Knowledge in 70-461

SQL中PERSISTED关键字

PERSISTED

指定 SQL Server 数据库引擎将在表中物理存储计算值，而且，当计算列依赖的任何其他列发生更新时对这些计算值进行更新。将计算列标记为 PERSISTED，可允许您对具有确定性、但不精确的计算列创建索引。有关详细信息，请参阅为计算列创建索引。用作已分区表的分区依据列的所有计算列都必须显式标记为 PERSISTED。指定 PERSISTED 时，computed\_column\_expression 必须具有确定性。

复制代码

create table orders

(

orderid int not null,

price money not null,

quantity int not null,

orderdate datetime not null,

total as price\*quantity ,

total2 as Price \* quantity persisted,

shipedate as dateadd (day,7,orderdate)

)

insert into orders (orderid,price,quantity,orderdate) values (1,2,6,'2008-8-8')

[clustered index implemented on view](https://stackoverflow.com/questions/16094204/how-clustered-index-implemented-on-view)

A view is just a stored query with no materialized data unless it has a clustered index.

In many cases, people think that any old view can be "sped up" by adding a clustered index. This is very seldom the case. Typically you create an indexed view to pre-aggregate certain computations, such as SUM or COUNT\_BIG. If you create an indexed view that doesn't aggregate and hence has the same number of rows / pages as the base table, you haven't achieved anything (of course, like anything, there are exceptions - the index on the view could be skinnier than the base table, for example, leading to fewer pages).

As an aside, you can't create a non-clustered index on a view unless you first create a clustered index. And you should also keep in mind that, like indexes on a table, an indexed view is not free. While it may speed up some queries, SQL Server has to maintain the index throughout the DML portion of your workload.

from your first line, when we create clustered index on view, that view become materialized view or we can create clustered index only on materialized view ? – [Ravi](https://stackoverflow.com/users/1162620/ravi) [Apr 23 '13 at 18:15](https://stackoverflow.com/questions/16094204/how-clustered-index-implemented-on-view#comment23122192_16095608)

* A view becomes materialized when you create a clustered index on it. – [Aaron Bertrand](https://stackoverflow.com/users/61305/aaron-bertrand) [Apr 23 '13 at 18:27](https://stackoverflow.com/questions/16094204/how-clustered-index-implemented-on-view#comment23122579_16095608)
* since i tagged sql server on my post, so, I'm asking one more question, does sql server have materialized view ? – [Ravi](https://stackoverflow.com/users/1162620/ravi) [Apr 23 '13 at 18:35](https://stackoverflow.com/questions/16094204/how-clustered-index-implemented-on-view#comment23122858_16095608)
* Yes, but it's called an indexed view. – [Aaron Bertrand](https://stackoverflow.com/users/61305/aaron-bertrand) [Apr 23 '13 at 18:36](https://stackoverflow.com/questions/16094204/how-clustered-index-implemented-on-view#comment23122865_16095608)
* ok.. means has same properties as materialized view in oracle. – [Ravi](https://stackoverflow.com/users/1162620/ravi) [Apr 23 '13 at 19:05](https://stackoverflow.com/questions/16094204/how-clustered-index-implemented-on-view#comment23123784_16095608)

add a comment

up vote3down vote

A clustered index on a view is basically a new table that gets automatically updated when the referenced tables get changed. So you buy quicker access time with pre collected data but you are paying with a significantly higher (double in many cases) storage use.

<https://docs.microsoft.com/en-us/sql/relational-databases/views/create-indexed-views?view=sql-server-2017>

Example:

CREATE TABLE MyBigTable(

ItemID INT PRIMARY KEY,

ItemDsc VARCHAR(20),

QTY INT)

GO

CREATE VIEW MyView WITH SCHEMABINDING AS

SELECT ItemID, QTY

FROM dbo.MyBigTable

WHERE QTY > 10

GO

CREATE UNIQUE CLUSTERED INDEX idx\_MyView ON MyView(QTY)

# datetimeoffset

Defines a date that is combined with a time of a day that has time zone awareness and is based on a 24-hour clock.

SELECT

CAST('2007-05-08 12:35:29. 1234567 +12:15' AS time(7)) AS 'time'

,CAST('2007-05-08 12:35:29. 1234567 +12:15' AS date) AS 'date'

,CAST('2007-05-08 12:35:29.123' AS smalldatetime) AS

'smalldatetime'

,CAST('2007-05-08 12:35:29.123' AS datetime) AS 'datetime'

,CAST('2007-05-08 12:35:29.1234567+12:15' AS datetime2(7)) AS

'datetime2'

,CAST('2007-05-08 12:35:29.1234567 +12:15' AS datetimeoffset(7)) AS

'datetimeoffset'

,CAST('2007-05-08 12:35:29.1234567+12:15' AS datetimeoffset(7)) AS

'datetimeoffset IS08601';

# CTE与视图之比较

一、CTE是什么

CTE是指公用表表达式(Common Table Expression)。它是一种临时结果集，可以在Select、Insert、Update、Delete语句的执行范围内定义并使用。也可用在 CREATE VIEW 语句中，作为该语句的 SELECT 定义语句的一部分。公用表表达式可以包括对自身的引用。这时它被称为递归公用表表达式。

语法：WITH < Expression\_Name > [ ( Column\_Name [ ,......n ] ) ]

              AS

                    (CTE\_query\_definition)

二、视图

视图是一个虚拟表，其内容由查询定义。同真实的表一样，视图包含一系列带有名称的列和行数据。视图在数据库中并不是以数据值存储集形式存在，除非是索引视图。行和列数据来自由定义视图的查询所引用的表，并且在引用视图时动态生成。

|语法：CREATE VIEW View\_Name

               AS

                        Select\_Statement [ ; ]

三、CTE与视图的不同：

1.可以创建递归查询

2.不用将定义存贮在元数据中（即不存贮为对象）

3.在同一语句中多次引用生成的表

4.可以在用户定义的例程（如函数、存储过程、触发器或视图）中定义 CTE

总之，使用 CTE 可以获得提高可读性（便于在代码中多次引用）和轻松维护复杂查询的优点（MSDN这样写的，呵呵）。

# 在存储过程中SET XACT\_ABORT ON 有什么用

2016年05月13日 15:00:53 [luoxq111](https://me.csdn.net/luoxq111) 阅读数：390更多

个人分类： [SQLServer管理开发](https://blog.csdn.net/luoxq111/article/category/3132273)

SET XACT\_ABORT ON设置事务回滚的 当为ON时，如果你存储中的某个地方出了问题，整个事务中的语句都会回滚 为OFF时，只回滚错误的地方.

# [数据库中存储过程和函数的区别](https://www.cnblogs.com/zhycyq/articles/2454758.html)

**函数**限制比较多，如不能用临时表，只能用表变量等，而**存储过程**的限制相对就比较少。

1.一般来说，存储过程实现的功能要复杂一点，而函数的实现的功能针对性比较强。

2.对于存储过程来说可以返回参数，而函数只能返回值或者表对象。

3.存储过程一般是作为一个独立的部分来执行，而函数可以作为查询语句的一个部分来调用，由于函数可以返回一个表对象，因此它可以在查询语句中位于FROM关键字的后面。

4.当存储过程和函数被执行的时候，SQLManager会到procedurecache中去取相应的查询语句，如果在procedurecache里没有相应的查询语句，SQLManager就会对存储过程和函数进行编译。

Procedurecache：中保存的是执行计划，当编译好之后就执行procedurecache中的executionplan，之后SQLSERVER会根据每个executionplan的实际情况来考虑是否要在cache中保存这个plan，评判的标准一个是这个executionplan可能被使用的频率；其次是生成这个plan的代价，也就是编译的耗时。保存在cache中的plan在下次执行时就不用再编译了。

**存储过程和函数具体的区别：**

存储过程：可以使得对的管理、以及显示关于及其用户信息的工作容易得多。存储过程是SQL语句和可选控制流语句的预编译集合，以一个名称存储并作为一个单元处理。存储过程存储在数据库内，可由应用程序通过一个调用执行，而且允许用户声明变量、有条件执行以及其它强大的编程功能。存储过程可包含程序流、逻辑以及对数据库的查询。它们可以接受参数、输出参数、返回单个或多个结果集以及返回值。

可以出于任何使用SQL语句的目的来使用存储过程，它具有以下优点：

(1)功能强大，限制少。

(2)可以在单个存储过程中执行一系列SQL语句。

(3)可以从自己的存储过程内引用其它存储过程，这可以简化一系列复杂语句。

(4)存储过程在创建时即在上进行编译，所以执行起来比单个SQL语句快。

(5)可以有多个返回值，即多个输出参数，并且可以使用SELECT返回结果集。

函数：是由一个或多个SQL语句组成的子程序，可用于封装代码以便重新使用。自定义函数诸多限制，有许多语句不能使用，许多功能不能实现。函数可以直接引用返回值，用表变量返回记录集。但是，用户定义函数不能用于执行一组修改全局数据库状态的操作。

补充：

前面有一句，“可以处于任何使用SQL语句的目的来使用存储过程”。这里想说的是，有些时候有些地方使用函数或许会更方便些。例如，存在这样一个表：Temperature(Year,Month,Day,T02,T08,T14,T20)，其中Year,Month,Day是时间字段，T02,T08,T14,T20是指2时、8时、14时、20时四个时刻对应的温度值，这些温度值可为空。现在，要求统计2008年5月份的平均温度。

或许大家会这样写：

SELECT(AVG(T02)+AVG(T08)+AVG(T14)+AVG(T20))/4FROMTemperatureWHEREYear=2008ANDMonth=5

如果不考虑空值的话，这样完全正确，但是考虑空值的话，如果根本没有统计02时的温度，那么AVG(T02)将为NULL，然后进行所有运算的结果都将为NULL。这显然是不正确的。

这里可以创建一个自定义函数，然后使用一个SELECT语句即可查询：

SELECTAVG(user.Average(T02,T08,T14,T20)FROMTemperatureWHEREYear=2008ANDMonth=5

总结：

用户自定义函数在处理同一数据行中的各个字段时，特别方便有用。虽然这里使用存储过程也能达到查询目的，但是显然没有使用函数方便。而且，即使使用存储过程也无法处理SELECT查询中的同一数据行中的各个字段的运算。因为存储过程不返回值，使用时只能单独调用；而函数却能出现在能放置表达式的任何位置。

# [对存储过程进行加密和解密(SQL 2008/SQL 2012)](https://www.cnblogs.com/wghao/archive/2012/12/30/2837642.html)

 在网络上，看到有SQL Server 2000和SQL Server 2005 的存储过程加密和解密的方法，后来分析了其中的代码，发现它们的原理都是一样的。后来自己根据实际的应用环境，编写了两个存储过程，一个加密存储过程（sp\_EncryptObject），和一个解密存储过程（sp\_EncryptObject），它们可以应用于SQL Server中的储过程，函数，视图，以及触发器。

感觉这两个存储过程蛮有意思的，拿来与大家分享；如果你看过类似的，就当作重温一下也好。

**用于加密的存储过程 （sp\_EncryptObject） ：**

 存储过程（sp\_EncryptObject）加密的方法是在存储过程，函数，视图的“As”位置前加上“with encryption”；如果是触发器，就在“for”位置前加“with encryption”。

如果触发器是{ AFTER | INSTEAD OF} 需要修改下面代码"For"位置：

if objectproperty(object\_id(@Object),'ExecIsAfterTrigger')=**0** set @Replace='As' ; else set @Replace='For ';

存储过程完成代码：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

Use master

Go

if object\_ID('[sp\_EncryptObject]') is not null

Drop Procedure [sp\_EncryptObject]

Go

create procedure sp\_EncryptObject

(

@Object sysname='All'

)

as

/\*

当@Object=All的时候，对所有的函数，存储过程，视图和触发器进行加密

调用方法：

1. Execute sp\_EncryptObject 'All'

2. Execute sp\_EncryptObject 'ObjectName'

\*/

begin

set nocount on

if @Object <>'All'

begin

if not exists(select **1** from sys.objects a where a.object\_id=object\_id(@Object) And a.type in('P','V','TR','FN','IF','TF'))

begin

--SQL Server 2008

raiserror **50001** N'无效的加密对象！加密对象必须是函数，存储过程，视图或触发器。'

--SQL Server 2012

--throw 50001, N'无效的加密对象！加密对象必须是函数，存储过程，视图或触发器。',1

return

end

if exists(select **1** from sys.sql\_modules a where a.object\_id=object\_id(@Object) and a.definition is null)

begin

--SQL Server 2008

raiserror **50001** N'对象已经加密！'

--SQL Server 2012

--throw 50001, N'对象已经加密！',1

return

end

end

declare @sql nvarchar(max),@C1 nchar(**1**),@C2 nchar(**1**),@type nvarchar(**50**),@Replace nvarchar(**50**)

set @C1=nchar(**13**)

set @C2=nchar(**10**)

declare cur\_Object

cursor for

select object\_name(a.object\_id) As ObjectName,a.definition

from sys.sql\_modules a

inner join sys.objects b on b.object\_id=a.object\_id

and b.is\_ms\_shipped=**0**

and not exists(select **1**

from sys.extended\_properties x

where x.major\_id=b.object\_id

and x.minor\_id=**0**

and x.class=**1**

and x.name='microsoft\_database\_tools\_support'

)

where b.type in('P','V','TR','FN','IF','TF')

and (b.name=@Object or @Object='All')

and b.name <>'sp\_EncryptObject'

and a.definition is not null

order by Case

when b.type ='V' then **1**

when b.type ='TR' then **2**

when b.type in('FN','IF','TF') then **3**

else **4** end,b.create\_date,b.object\_id

open cur\_Object

fetch next from cur\_Object into @Object,@sql

while **@@fetch\_status**=**0**

begin

Begin Try

if objectproperty(object\_id(@Object),'ExecIsAfterTrigger')=**0** set @Replace='As' ; else set @Replace='For ';

if (patindex('%'+@C1+@C2+@Replace+@C1+@C2+'%',@sql)>**0**)

begin

set @sql=Replace(@sql,@C1+@C2+@Replace+@C1+@C2,@C1+@C2+'With Encryption'+@C1+@C2+@Replace+@C1+@C2)

end

else if(patindex('%'+@C1+@Replace+@C1+'%',@sql)>**0**)

begin

set @sql=Replace(@sql,@C1+@Replace+@C1,@C1+'With Encryption'+@C1+@Replace+@C1)

end

else if(patindex('%'+@C2+@Replace+@C2+'%',@sql)>**0**)

begin

set @sql=Replace(@sql,@C2+@Replace+@C2,@C2+'With Encryption'+@C2+@Replace+@C2)

end

else if(patindex('%'+@C2+@Replace+@C1+'%',@sql)>**0**)

begin

set @sql=Replace(@sql,@C2+@Replace+@C1,@C1+'With Encryption'+@C2+@Replace+@C1)

end

else if(patindex('%'+@C1+@C2+@Replace+'%',@sql)>**0**)

begin

set @sql=Replace(@sql,@C1+@C2+@Replace,@C1+@C2+'With Encryption'+@C1+@C2+@Replace)

end

else if(patindex('%'+@C1+@Replace+'%',@sql)>**0**)

begin

set @sql=Replace(@sql,@C1+@Replace,@C1+'With Encryption'+@C1+@Replace)

end

else if(patindex('%'+@C2+@Replace+'%',@sql)>**0**)

begin

set @sql=Replace(@sql,@C2+@Replace,@C2+'With Encryption'+@C2+@Replace)

end

set @type =

case

when object\_id(@Object,'P')>**0** then 'Proc'

when object\_id(@Object,'V')>**0** then 'View'

when object\_id(@Object,'TR')>**0** then 'Trigger'

when object\_id(@Object,'FN')>**0** or object\_id(@Object,'IF')>**0** or object\_id(@Object,'TF')>**0** then 'Function'

end

set @sql=Replace(@sql,'Create '+@type,'Alter '+@type)

Begin Transaction

exec(@sql)

print N'已完成加密对象('+@type+')：'+@Object

Commit Transaction

End Try

Begin Catch

Declare @Error nvarchar(**2047**)

Set @Error='Object: '+@Object+@C1+@C2+'Error: '+Error\_message()

Rollback Transaction

print @Error

print @sql

End Catch

fetch next from cur\_Object into @Object,@sql

end

close cur\_Object

deallocate cur\_Object

end

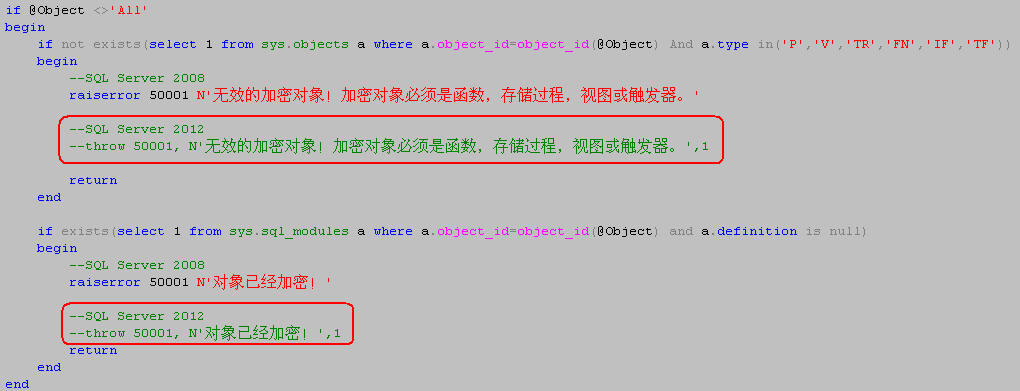
Go

exec sp\_ms\_marksystemobject 'sp\_EncryptObject' --标识为系统对象

go

[复制代码](javascript:void(0);)

如果SQL Server 2012，请修改下面两个位置的代码。在SQL Server 2012，建议在使用throw来代替raiserror。



**解密方法：**

 解密过程，最重要采用异或方法：

[字符1]经过函数 fn\_x（x）加密变成[加密后字符1]，如果我们已知[加密后字符1]，反过来查[字符1],可以这样：

[字符1]  =  [字符2]  ^  fn\_x([字符2])  ^  [加密后字符1]

这里我列举一个简单的例子：

[复制代码](javascript:void(0);)

--创建加密函数（fn\_x）

if object\_id('fn\_x') is not null drop function fn\_x

go

create function fn\_x

(

@x nchar(**1**)

)returns nchar(**1**)

as

begin

return(nchar((**65535**-unicode(@x))))

end

go

declare @nchar\_1\_encrypt nchar(**1**),@nchar\_2 nchar(**1**)

--对字符'A'进行加密，存入变量@nchar\_1\_encrypt

set @nchar\_1\_encrypt=dbo.fn\_x(N'A')

--參考的字符@nchar\_2

set @nchar\_2='x'

--算出@nchar\_1\_encrypt 加密前的字符

select nchar(unicode(@nchar\_2)^unicode(dbo.fn\_x(@nchar\_2))^unicode(@nchar\_1\_encrypt)) as [@nchar\_1]

/\*

@nchar\_1

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

A

\*/

[复制代码](javascript:void(0);)

**[注]：**从SQL Server 2000至 SQL Server 2012 采用异或方法都可以解密

**用于解密的存储过程（sp\_DecryptObject）：**

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

Use master

Go

if object\_ID('[sp\_DecryptObject]') is not null

Drop Procedure [sp\_DecryptObject]

Go

create procedure sp\_DecryptObject

(

@Object sysname, --要解密的对象名：函数，存储过程，视图或触发器

@MaxLength int=**4000** --评估内容的长度

)

as

set nocount on

/\* 1. 解密 \*/

if not exists(select **1** from sys.objects a where a.object\_id=object\_id(@Object) And a.type in('P','V','TR','FN','IF','TF'))

begin

--SQL Server 2008

raiserror **50001** N'无效的对象！要解密的对象必须是函数，存储过程，视图或触发器。'

--SQL Server 2012

--throw 50001, N'无效的对象！要解密的对象必须是函数，存储过程，视图或触发器。',1

return

end

if exists(select **1** from sys.sql\_modules a where a.object\_id=object\_id(@Object) and a.definition is not null)

begin

--SQL Server 2008

raiserror **50001** N'对象没有加密！'

--SQL Server 2012

--throw 50001, N'无效的对象！要解密的对象必须是函数，存储过程，视图或触发器。',1

return

end

declare @sql nvarchar(max) --解密出来的SQL语句

,@imageval nvarchar(max) --加密字符串

,@tmpStr nvarchar(max) --临时SQL语句

,@tmpStr\_imageval nvarchar(max) --临时SQL语句(加密后)

,@type char(**2**) --对象类型('P','V','TR','FN','IF','TF')

,@objectID int --对象ID

,@i int --While循环使用

,@Oject1 nvarchar(**1000**)

set @objectID=object\_id(@Object)

set @type=(select a.type from sys.objects a where a.object\_id=@objectID)

declare @Space4000 nchar(**4000**)

set @Space4000=replicate('-',**4000**)

/\*

@tmpStr 会构造下面的SQL语句

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

alter trigger Tr\_Name on Table\_Name with encryption for update as return /\*\*/

alter proc Proc\_Name with encryption as select **1** as col /\*\*/

alter view View\_Name with encryption as select **1** as col /\*\*/

alter function Fn\_Name() returns int with encryption as begin return(**0**) end/\*\*/

\*/

set @Oject1=quotename(object\_schema\_name(@objectID))+'.'+quotename(@Object)

set @tmpStr=

case

when @type ='P ' then N'Alter Procedure '+@Oject1+' with encryption as select 1 as column1 '

when @type ='V ' then N'Alter View '+@Oject1+' with encryption as select 1 as column1 '

when @type ='FN' then N'Alter Function '+@Oject1+'() returns int with encryption as begin return(0) end '

when @type ='IF' then N'Alter Function '+@Oject1+'() returns table with encryption as return(Select a.name from sys.types a) '

when @type ='TF' then N'Alter Function '+@Oject1+'() returns @t table(name nvarchar(50)) with encryption as begin return end '

else 'Alter Trigger '+@Oject1+'on '+quotename(object\_schema\_name(@objectID))+'.'+(select Top(**1**) quotename(object\_name(parent\_id)) from sys.triggers a where a.object\_id=@objectID)+' with encryption for update as return '

end

set @tmpStr=@tmpStr+'/\*'+@Space4000

set @i=**0**

while @i < (ceiling(@MaxLength\***1.0**/**4000**)-**1**)

begin

set @tmpStr=@tmpStr+ @Space4000

Set @i=@i+**1**

end

set @tmpStr=@tmpStr+'\*/'

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

set @imageval =(select top(**1**) a.imageval from sys.sysobjvalues a where a.objid=@objectID and a.valclass=**1**)

begin tran

exec(@tmpStr)

set @tmpStr\_imageval =(select top(**1**) a.imageval from sys.sysobjvalues a where a.objid=@objectID and a.valclass=**1**)

rollback tran

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

set @tmpStr=stuff(@tmpStr,**1**,**5**,'create')

set @sql=''

set @i=**1**

while @i<= (datalength(@imageval)/**2**)

begin

set @sql=@sql+isnull(nchar(unicode(substring(@tmpStr,@i,**1**)) ^ unicode(substring(@tmpStr\_imageval,@i,**1**))^unicode(substring(@imageval,@i,**1**)) ),'')

Set @i+=**1**

end

/\* 2. 列印 \*/

declare @patindex int

while @sql>''

begin

set @patindex=patindex('%'+char(**13**)+char(**10**)+'%',@sql)

if @patindex >**0**

begin

print substring(@sql,**1**,@patindex-**1**)

set @sql=stuff(@sql,**1**,@patindex+**1**,'')

end

else

begin

set @patindex=patindex('%'+char(**13**)+'%',@sql)

if @patindex >**0**

begin

print substring(@sql,**1**,@patindex-**1**)

set @sql=stuff(@sql,**1**,@patindex,'')

end

else

begin

set @patindex=patindex('%'+char(**10**)+'%',@sql)

if @patindex >**0**

begin

print substring(@sql,**1**,@patindex-**1**)

set @sql=stuff(@sql,**1**,@patindex,'')

end

else

begin

print @sql

set @sql=''

end

end

end

end

Go

exec sp\_ms\_marksystemobject 'sp\_DecryptObject' --标识为系统对象

go

[复制代码](javascript:void(0);)

 如果SQL Server 2012，请修改下面两个位置的代码。方法类似于前面的加密过程：



**搭建测试环境：**

在一个测试环境中（DB: Test），先执行上面的加密存储过程（sp\_EncryptObject）和解密存储过程（sp\_EncryptObject）;再创建两个表：TableA & TableB

[复制代码](javascript:void(0);)

use test

go

--创建表: TableA & TableB

if object\_id('myTableA') is not null drop table myTableA

if object\_id('myTableB') is not null drop table myTableB

go

create table myTableA (ID int identity,data nvarchar(**50**),constraint PK\_myTableA primary key(ID))

create table myTableB (ID int ,data nvarchar(**50**),constraint PK\_myTableB primary key(ID))

go

[复制代码](javascript:void(0);)

接下来，我们要创建6个未加密的对象（对象类型包含 'P','V','TR','FN','IF','TF'）：

 1.视图（myView）：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

if object\_id('myView') is not null drop view myView

go

create view myView

As

select \* from TableA;

go

2.触发器（MyTrigger）：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

if object\_id('MyTrigger') is not null drop Trigger MyTrigger

go

create trigger MyTrigger

on TableA

for update

As

insert into TableB(ID,data) select a.ID,a.Data From Inserted a

go

[复制代码](javascript:void(0);)

3.存储过程（MyProc）：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

if object\_id('MyProc') is not null drop proc MyProc

go

create proc MyProc

(

@data nvarchar(**50**)

)

As

insert into TableA(data) values(@data)

go

[复制代码](javascript:void(0);)

4.用户定义表值函数(TF)（MyFunction\_TF）：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

if object\_id('MyFunction\_TF') is not null drop function MyFunction\_TF

go

create function MyFunction\_TF

(

)

returns @t table

(

id int,

data nvarchar(**50**)

)

As

begin

insert @t(id,data) select id,data from TableA

return

end

go

[复制代码](javascript:void(0);)

5.内联表值函数(IF) （MyFunction\_IF）：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

if object\_id('MyFunction\_IF') is not null drop function MyFunction\_IF

go

create function MyFunction\_IF

(

)

returns table

As

return(select top(**3**) id, data from TableA order by id desc)

go

[复制代码](javascript:void(0);)

6.标量函数(FN)（MyFunction\_FN）：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

if object\_id('MyFunction\_FN') is not null drop function MyFunction\_FN

go

create function MyFunction\_FN

(

)

returns nvarchar(**50**)

As

begin

return(select top(**1**) data from TableA order by id desc)

end

go

[复制代码](javascript:void(0);)

 当执行完了上面的1-6步骤的脚本，我们通过查询系统视图sys.sql\_modules，可以看到未加密前的定义信息：

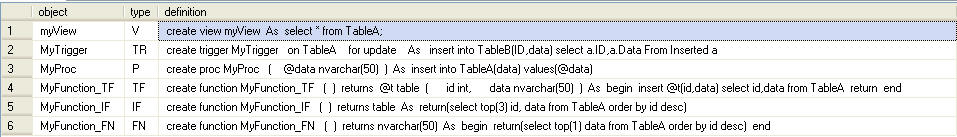
select b.name as object,b.type,a.definition

from sys.sql\_modules a

inner join sys.objects b on b.object\_id=a.object\_id

where b.create\_date>=convert(date,getdate())

order by b.object\_id



**加密测试：**

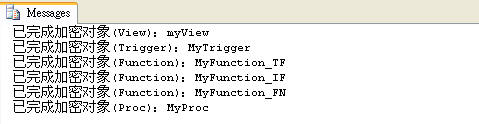
下面我就通过调用加密存储过程（sp\_EncryptObject），一次性对它们进行加密：

use test

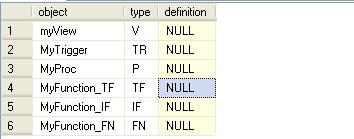
go

exec sp\_EncryptObject 'all'

go



当我们再查回系统视图sys.sql\_modules，会发现definition列返回的是null值，说明定义内容已经给加密：



**解密测试：**

解密过程，必须在DAC连接SQL Server，我们这里例子是从 SSMS(SQL Server Management Studio) 查询编辑器启动 DAC，如图：



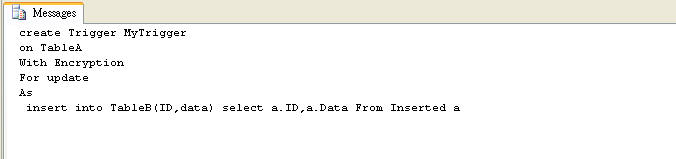
解密存储过程（sp\_DecryptObject），只能一次对一个存储过程、函数、视图或触发器，进行解密：

use test

go

exec sp\_DecryptObject MyTrigger

go

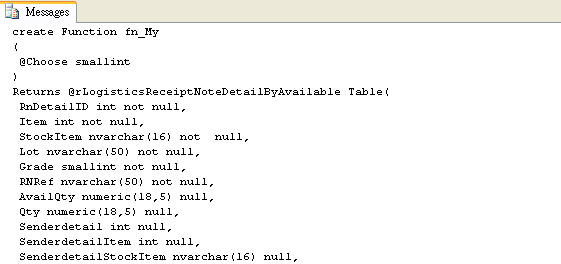


当定义内容长度超过4000，我们可以指定@MaxLength的值，如：

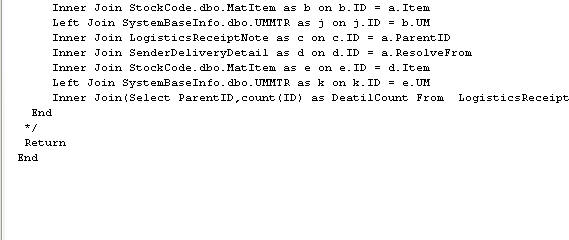
exec sp\_DecryptObject fn\_My,**20000**

go

这里（fn\_My）是一个函数，定义内容超过了8000：



... ...



# [binary 和 varbinary 用法全解](https://www.cnblogs.com/ljhdo/p/4530293.html)

在SQL Server中，使用数据类型 binary(n) 和 varbinary(n) 存储二进制数据，n是指字节数量：

* binary(n)：固定长度为 n 字节，其中 n 值从 1 到 8,000 ，存储空间为 n 字节；
* varbinary( n | max)：可变长度，n 的取值范围为 1 至 8,000，max 是指最大存储空间是 2^31-1 个字节，即最大4GB；
* n：在表列定义或变量声明语句中没有指定 n，则默认长度为 1；在CAST 函数中没有指定 n，则默认长度为 30；

在将数据转换为二进制数据时，SQL Server会对生成的二进制数据进行填充或截断，详细的规则是：

* 填充（或扩展）的二进制数据是16进制的**0x00的整数倍，这就是说，填充的字节中每一个bit都是0；**
* 将字符类型转换成二进制数据时，在数据的**右侧**填充或截断数据，填充数值是0x00；
* 将其他类型数据转换成二进制数据时，在数据的**左侧**填充或截断数据，填充数值是0x00；在截断数据时，保留低位的数据，将高位数值截断；

**一，初试binary类型定义二进制变量**

**1，存储单位是Byte**

数据类型binary 和 varbinary 占用的存储空间的单位不是bit，而是 byte

declare @varBi binary(**1**)

set @varBi=**1**

输出结果是：0x01，结果以16进制编码显示，共8bit。

**2，为binary变量赋值**

可以使用10进制数据为binary类型的变量赋值，也可以直接使用16进制数值为binary变量赋值；

declare @biA binary(**1**)

set @biA= **0x01**

**二，二进制数据的类型转换**

**1， 数值类型隐式转换成binary类型**

系统自动将 int类型隐式转换成binary类型，但是，binary必须显式强制转换成int类型。

declare @varBi binary(**1**)

set @varBi=**1**

select @varBi,cast(@varbi as int)

**2， 字符类型不能隐式转换成binary类型，需要显式转换**

将字符类型转换成binary类型的过程是：逐个将字符的ASCII码转换成二进制数据，末尾补0，或截断末尾多余的字符

declare @varBi binary(**8**)

set @varBi=cast('1h' as binary(**8**))

select @varBi,ascii('1'),ascii('h'),cast(@varbi as varchar),DATALENGTH(cast(@varbi as varchar))

https://images2015.cnblogs.com/blog/628084/201610/628084-20161016112718420-1286117706.png

输出结果分析：

* 0x31 是16进制，换算成10进制数值是49，和字符“1”的ASCII值相同；
* 0x68 是16进制，换算成10进制数值是104，和字符“h”的ASCII值相同；
* 末尾补0：由于字符串的长度是2Byte，定长binary(8)的长度是8Byte，在binary(8) 类型的末尾补0，补齐8Byte；
* 在将binary转换成字符类型时，输出结果是字符 “1h”，而不是数字，说明，按照ASCII将数值转换成字符；
* 转换成字符串类型时，长度是8Byte，未显示的字符是不可打印的；

**三，将字符串翻译（translate）成二进制类型**

从字符数据类型翻译成binary或varbinary类型时，将在右侧发生截断或填充，填充的是16进制的0x00。当从字符类型（char、varchar、nchar、nvarchar）转换为不同长度的 binary 或 varbinary 数据类型时，SQL Server 将在二进制数据的右侧填充或截断数据。从字符串的左边逐字节翻译，如果字符串的长度大于binary的长度，则右边截断；如果字符串的长度小于binary的长度，则右边填充0x00。

翻译的字符必须是0-9、a-f，逐个把字符的ASCII码翻译成二进制数字，或者逐个把二进制数字对应ASCII码，转换为字符。默认情况下，对于二进制数据，0x是自动填充导结果的最左侧。

**1，转换成binary或varbinary类型时，从数据的右侧截断数据**

declare @varBi binary(**2**)

set @varBi=cast('123' as binary(**3**))

select @varBi,cast(@varbi as varchar)

https://images2015.cnblogs.com/blog/628084/201610/628084-20161016113308202-68144069.png

**2，转换成binary或varbinary类型时，从数据的右侧填充数据，填充的数值是0x00**

declare @varBi binary(**8**)

set @varBi=cast('123' as binary(**8**))

select @varBi,cast(@varbi as varchar)

https://images2015.cnblogs.com/blog/628084/201610/628084-20161016113515014-1230231347.png

**3，使用convert的风格0（style=0）把binary（或varbinary）翻译成字符串**

select convert(varchar(**8**), **0x4E616d65**, **0**),cast(**0x4E616d65** as varchar(**8**))

https://images2017.cnblogs.com/blog/628084/201711/628084-20171108173339669-224345526.png

**4，使用convert的风格0（style=0）把字符串翻译成binary（或varbinary）**

select convert(varbinary(**8**), 'name', **0**),cast('name' as varbinary(**8**))

https://images2017.cnblogs.com/blog/628084/201711/628084-20171108173537809-1096089704.png

**四，使用convert进行字符串和binary(或varbinary)的转换**

如果要把二进制数据，按照显示的字符转换为字符串，那么就必须使用convert转换，convert的第三个参数是style，共有两种风格，分别是1 和 2，风格1要求0x必须在二进制数据的最左侧，风格2要求0x不能出现在二进制数据的最左侧。

**1，使用convert把binary转换为字符串**

分别设置style是1和2，查看转换的结果：当style为1时，字符串前面添加0x；当style为2时，字符串前面没有0x；

declare @b binary(**8**)

set @b=**12**

select convert(varchar(**64**),@b,**1**),convert(varchar(**64**),@b,**2**)

https://images2017.cnblogs.com/blog/628084/201711/628084-20171108170408559-709880213.png

**2，使用convert把字符串转换为binary类型**

如果字符串前面有0x字符，那么使用风格1（设置style=1）；如果字符串前面没有0x字符，那么使用风格2（设置style=2）；

declare @c varchar(**64**),@d varchar(**64**)

select @c='0x000000000000000C',@d='000000000000000C'

select convert(binary(**8**),@c,**1**), convert(binary(**8**),@d,**2**)

https://images2017.cnblogs.com/blog/628084/201711/628084-20171108172605888-624461048.png

**五，将整数类型转换为二进制类型**

从整数类型转换为 binary 或 varbinary 时，在数据的左侧填充或截断数据，填充将通过使用十六进制的零来完成。

**1，从整数类型转换为二进制类型时，在数据的左侧截断数据，即保留低位的数据，将高位数值截断**

declare @biToInt binary(**1**)

set @biToInt=**300**

select @biToInt,cast(@biToInt as int)

https://images2015.cnblogs.com/blog/628084/201610/628084-20161016114318717-1985408064.png

分析：由于binary(1)是1Byte，最大值是255，发生上溢，左侧发生截断，舍弃左边高位数值 256，保留低位数值44

**2，从整数类型转换为二进制类型时，在数据的左侧填充数据，即在高位填充数值：0x00**

declare @biToInt binary(**3**)

set @biToInt= **12**

select @biToInt,cast(@biToInt as int)

https://images2015.cnblogs.com/blog/628084/201610/628084-20161016114725327-344806908.png

如果没有发生溢出，那么不管int类型数值有多大，都能生成相同的值，如果binary长度较长，那么左侧填充为0x00。

**六，二进制数据的比较，将自动提升操作数的数据类型**

在SQL Server中，字节大的二进制数据类型，其类型处于“高”位。在进行二进制数值比较之前，首先匹配两个操作数的数据类型，将两个操作数提升为相同的数据类型，这体现在比较两个binary类型的变量时，将字节数低的数据类型提升，保持两个操作数的字节数相同。提升的字节使用0x00填充，并且是在右侧填充，这将增大binary的数值。

**1，自动提升数据类型，在右侧填充0x00**

declare @biA binary(**1**)

declare @biB binary(**2**)

set @biA=**1**

set @biB=**2**

select @biA,@biB,iif(@biA >= @biB,'>=','<')

https://images2015.cnblogs.com/blog/628084/201610/628084-20161016115721202-285700901.png

输出结果分析：在比较时，SQL Server将变量@biA 的数据类型提升为binary(2)，在右侧填充1byte，变成0x0100，数值是256，所以比较的结果是 >=；

如果变量@biB的值是257，那么输出的结果将是<，因为，@biA填充之后，是0x0100，比0x0101小；

declare @biA varbinary(**1**)

declare @biB varbinary(**2**)

set @biA=**1**

set @biB=**257**

select @biA,@biB,iif(@biA >= @biB,'>=','<')

https://images2015.cnblogs.com/blog/628084/201610/628084-20161016115807061-276923728.png

[x86](https://www.baidu.com/s?wd=x86&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)、x64是处理器的通用型号，[x86](https://www.baidu.com/s?wd=x86&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)的处理能力是32位(32 bit)，x64的处理能力是64位(64 bit)，所以就没有86位的说法。以前的电脑处理能力是32位，所以要装32位系统，现在的电脑基本上都有64位的处理能力。64位系统占用内存稍大，运行64位软件速度更快。

# [字节、字、bit、byte的关系](https://www.cnblogs.com/chansblogs/p/6542215.html)

**字     word**  
**字节  byte**  
**位     bit**  
**字长**是指字的长度

1字节=8位(1 byte = 8bit)  
1字=2字节(1 word = 2 byte)

一个字节的字长是8  
一个字的字长为16

**bps**    是 bits per second 的简称。一般数据机及网络通讯的传输速率都是以「bps」为单位。如56Kbps、100.0Mbps 等等。   
**Bps**即是Byte per second 的简称。而电脑一般都以Bps 显示速度，如1Mbps 大约等同 128 KBps。   
**bit**     电脑记忆体中最小的单位，在二进位电脑系统中，每一bit 可以代表0 或 1 的数位讯号。   
**Byte**一个Byte由8 bits 所组成，可代表一个字元(A~Z)、数字(0~9)、或符号(,.?!%&+-\*/)，是记忆体储存资料的基本单位，至於每个中文字则须要两Bytes。当记忆体容量过大时，位元组这个单位就不够用，因此就有千位元组的单位KB出现，以下乃个记忆体计算单位之间的相关性：

**1 Byte = 8 Bits**

**1 KB = 1024 Bytes**

**1 MB = 1024 KB**

**1 GB = 1024 MB**

usb2.0标准接口传输速率。许多人都将“480mbps”误解为480兆/秒。其实，这是错误的，事实上“480mbps”应为“480兆比特/秒”或“480兆位/秒”，它等于“60兆字节/秒”，大家看到差距了吧。

这要从bit和byte说起：bit和byte同译为"比特"，都是数据量度单位，bit=“比特”或“位”。   
byte=字节即1byte=8bits,两者换算是1：8的关系。   
mbps=mega bits per second(兆位/秒)是速率单位，所以正确的说法应该是说usb2.0的传输速度是480兆位/秒,即480mbps。   
mb=mega bytes(兆比、兆字节)是量单位，1mb/s（兆字节/秒）=8mbps（兆位/秒）。

我们所说的硬盘容量是40gb、80gb、100gb，这里的b指是的byte也就是“字节”。   
1 kb = 1024 bytes =2^10 bytes   
1 mb = 1024 kb = 2^20 bytes   
1 gb = 1024 mb = 2^30 bytes

比如以前所谓的56kb的modem换算过来56kbps除以8也就是7kbyte，所以真正从网上下载文件存在硬盘上的速度也就是每秒7kbyte。   
也就是说与传输速度有关的b一般指的是bit。   
与容量有关的b一般指的是byte。

最后再说一点: usb2.0 480mbps=60mb/s的传输速率还只是理论值，它还要受到系统环境的制约（cpu、硬盘和内存等），其实际读、取写入硬盘的速度约在11～16mb/s。但这也比usb1.1的12mbps(1.5m/s)快了近10倍。

# 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://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms187373%28v%3dsql.105%29).

**Note**

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.

Copy

USE AdventureWorks2008R2;

GO

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

GO

BEGIN TRANSACTION;

GO

SELECT Title

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 2008, 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://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms190273%28v%3dsql.105%29).

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

## **Examples**

### A. Using an explicit transaction

This example uses AdventureWorks.

Copy

BEGIN TRANSACTION;

DELETE FROM HumanResources.JobCandidate

WHERE JobCandidateID = 13;

COMMIT;

### B. Rolling back a transaction

The following example shows the effect of rolling back a transaction. In this example, the ROLLBACK statement will roll back the INSERT statement, but the created table will still exist.

Copy

CREATE TABLE ValueTable (id int);

BEGIN TRANSACTION;

INSERT INTO ValueTable VALUES(1);

INSERT INTO ValueTable VALUES(2);

ROLLBACK;

### C. Naming a transaction

The following example shows how to name a transaction.

Copy

DECLARE @TranName VARCHAR(20);

SELECT @TranName = 'MyTransaction';

BEGIN TRANSACTION @TranName;

USE AdventureWorks2012;

DELETE FROM AdventureWorks2012.HumanResources.JobCandidate

WHERE JobCandidateID = 13;

COMMIT TRANSACTION @TranName;

GO

### D. Marking a transaction

The following example shows how to mark a transaction. The transaction CandidateDelete is marked.

Copy

BEGIN TRANSACTION CandidateDelete

WITH MARK N'Deleting a Job Candidate';

GO

USE AdventureWorks2012;

GO

DELETE FROM AdventureWorks2012.HumanResources.JobCandidate

WHERE JobCandidateID = 13;

GO

COMMIT TRANSACTION CandidateDelete;

GO

# [查看SQL Server被锁的表以及如何解锁](https://www.cnblogs.com/shy1766IT/p/6225694.html)

前段时间\*\*公司DBA来我们这培训。讲了一大堆MYSQL的优化。 QA环节一程序员问“SQL语句中的 with nolock 除了不锁表外,是否能读其他锁住的数据"。

　　讲课的人嘟嘟了半天没解释清楚（有可能是MYSQL里没有这个机制），公司的另一程序员给出了一个很简洁明了的回答：

　　WITH NOLOCK 除了本身不锁表（不加任何锁） 也不会受其他的已存在的锁影响， 锁住的行数据也照样读，

个人认为这句话说得很清楚明了，一句话就能说明白的事，不过好奇怪的是程序员经常用这个语句竟然也不去试一下。 这里顺便总结一下 其他的 SQLSERVER 中的with锁级别：

　　WITH NOLOCK：无锁

　　WITH HOLDLOCK：挂一个保持锁

　　WITH UPDLOCK：挂一个更新锁

　　WITH XLOCK：挂一个排他锁

　　需要注意的是 with nolock 是不能用于update,delete insert 这种更新语句的，说绕了。简单的说 with nolock 只能用于select。

　　例如:update dbo.test with(NOLOCK) set username='wokofo' --这样的语句是错误的

　　弹回：INSERT、UPDATE、DELETE 或 MERGE 语句的目标表不允许使用 NOLOCK 和 READUNCOMMITTED 锁提示。

实际使用：

selecttop10\*from dbo.test with(NOLOCK) selecttop10\*from dbo.test with(HOLDLOCK) selecttop10\*from dbo.test with(XLOCK) selecttop10\*from dbo.test with(UPDLOCK) update dbo.test with(HOLDLOCK) set username='wokofo' update dbo.test with(XLOCK) set username='wokofo' update dbo.test with(UPDLOCK) set username='wokofo'

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

SELECT语句中的各项“加锁选项”以及相应的功能说明。   
NOLOCK（不加锁）   
此选项被选中时，SQL Server 在读取或修改数据时不加任何锁。 在这种情况下，用户有可能读取到未完成事务（Uncommited Transaction）或回滚(Roll Back)中的数据, 即所谓的“脏数据”。  
NOLOCK 语句执行时不发出共享锁，允许脏读 ，等于 READ UNCOMMITTED事务隔离级别

HOLDLOCK（保持锁）   
此选项被选中时，SQL Server 会将此共享锁保持至整个事务结束，而不会在途中释放。   
HOLDLOCK 持有共享锁，直到整个事务完成，应该在被锁对象不需要时立即释放，等于SERIALIZABLE事务隔离级别

UPDLOCK（修改锁）   
此选项被选中时，SQL Server 在读取数据时使用修改锁来代替共享锁，并将此锁保持至整个事务或命令结束。使用此选项能够保证多个进程能同时读取数据但只有该进程能修改数据。

TABLOCK（表锁）   
此选项被选中时，SQL Server 将在整个表上置共享锁直至该命令结束。 这个选项保证其他进程只能读取而不能修改数据。

PAGLOCK（页锁）  
此选项为默认选项， 当被选中时，SQL Server 使用共享页锁。  
PAGLOCK 在使用一个表锁的地方用多个页锁

TABLOCKX（排它表锁）   
此选项被选中时，SQL Server 将在整个表上置排它锁直至该命令或事务结束。这将防止其他进程读取或修改表中的数据。 TABLOCKX 强制使用独占表级锁，这个锁在事务期间阻止任何其他事务使用这个表

READPAST 让sql server跳过任何锁定行，执行事务，适用于READ UNCOMMITTED事务隔离级别只跳过RID锁，不跳过页，区域和表锁  
ROWLOCK 强制使用行锁   
UPLOCK 强制在读表时使用更新而不用共享锁

查看被锁表：  
select request\_session\_id spid,OBJECT\_NAME(resource\_associated\_entity\_id) tableName   
from sys.dm\_tran\_locks where resource\_type='OBJECT'   
spid 锁表进程   
tableName 被锁表名

解锁：  
declare @spid int   
Set @spid = 57 --锁表进程  
declare @sql varchar(1000)  
set @sql='kill '+cast(@spid as varchar)  
exec(@sql)

USE master;

GO

EXEC sp\_who 'active';

GO

EXEC sp\_who2 'active';

GO

USE master;

GO

KILL spid | UOW [WITH STATUSONLY]

GO

[复制代码](javascript:void(0);)

select

object\_name(P.object\_id) as TableName,

resource\_type, resource\_description,request\_session\_id

from

sys.dm\_tran\_locks L

join sys.partitions P on L.resource\_associated\_entity\_id = p.hobt\_id

kill ID

--Other Information about Locks:

select \* from sys.dm\_tran\_locks

SELECT \* FROM sys.dm\_exec\_requests WHERE blocking\_session\_id<>**0**

select \* from sys.sysprocesses

select

object\_name(resource\_associated\_entity\_id) as 'TableName' ,\*

from

sys.dm\_tran\_locks

where resource\_type = 'OBJECT'

and resource\_database\_id = DB\_ID()

GO

[复制代码](javascript:void(0);)

# UPDATE STATISTICS (Transact-SQL)

Updates query optimization statistics on a table or indexed view. By default, the query optimizer already updates statistics as necessary to improve the query plan; in some cases you can improve query performance by using UPDATE STATISTICS or the stored procedure [sp\_updatestats](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-updatestats-transact-sql?view=sql-server-2017) to update statistics more frequently than the default updates.

Updating statistics ensures that queries compile with up-to-date statistics. However, updating statistics causes queries to recompile. We recommend not updating statistics too frequently because there is a performance tradeoff between improving query plans and the time it takes to recompile queries. The specific tradeoffs depend on your application. UPDATE STATISTICS can use tempdb to sort the sample of rows for building statistics.

## Examples

### A. Update all statistics on a table

The following example updates the statistics for all indexes on the SalesOrderDetail table.

SQLCopy

USE AdventureWorks2012;

GO

UPDATE STATISTICS Sales.SalesOrderDetail;

GO

### B. Update the statistics for an index

The following example updates the statistics for the AK\_SalesOrderDetail\_rowguid index of the SalesOrderDetailtable.

SQLCopy

USE AdventureWorks2012;

GO

UPDATE STATISTICS Sales.SalesOrderDetail AK\_SalesOrderDetail\_rowguid;

GO

### C. Update statistics by using 50 percent sampling

The following example creates and then updates the statistics for the Name and ProductNumber columns in the Product table.

SQLCopy

USE AdventureWorks2012;

GO

CREATE STATISTICS Products

ON Production.Product ([Name], ProductNumber)

WITH SAMPLE 50 PERCENT

-- Time passes. The UPDATE STATISTICS statement is then executed.

UPDATE STATISTICS Production.Product(Products)

WITH SAMPLE 50 PERCENT;

### D. Update statistics by using FULLSCAN and NORECOMPUTE

The following example updates the Products statistics in the Product table, forces a full scan of all rows in the Product table, and turns off automatic statistics for the Products statistics.

SQLCopy

USE AdventureWorks2012;

GO

UPDATE STATISTICS Production.Product(Products)

WITH FULLSCAN, NORECOMPUTE;

GO

## Examples

# 在存储过程中使用 EXECUTE AS

# execute as是用来定义模块的执行上下文

# EXECUTE AS { CALLER | SELF | OWNER | 'user\_name' }

# 其中CALLER指模块调用方，SELF指创建或更改模块的用户，OWNER指模块的当前所有者， ‘user\_name’ 则是指定的用户。

有一个有意思的实验：

1. 创建一个测试数据库及表：

CREATE DATABASE mydb

go

USE mydb

go

create table dbo.tbl\_TestPer(ID INT, name sysname)

go

INSERT INTO tbl\_TestPer SELECT 1, 'Iori'

2. 创建如下存储过程：

CREATE PROC [dbo].[sp\_TestPer1]

WITH EXECUTE AS CALLER

AS

BEGIN

SELECT SYSTEM\_USER, CURRENT\_USER

SELECT \* FROM dbo.tbl\_TestPer

END

3. 创建一个测试用户并修改其权限：

CREATE LOGIN testusr WITH PASSWORD='Password@123!'

go

CREATE USER testusr FROM LOGIN testusr

go

GRANT CONNECT TO testusr

go

DENY SELECT ON dbo.tbl\_TestPer TO testusr

go

GRANT EXECUTE ON dbo.sp\_TestPer1 TO test

go

4. 以testusr登录并执行如下语句，可以成功查询到记录

EXEC mydb..[sp\_TestPer1]

这个时候的testusr用户实际上是没有权限查看tbl\_TestPer表的，那么它是如何成功执行SELECT语句的呢？答案就在于Ownership chaining。由于sp\_TestPer1和tbl\_TestPer都属于schema dbo，而dbo这个schema的拥有者是用户dbo（好像有点绕），所以第二次对SELECT 语句的权限检查直接被by pass了。在联机丛书中有如下描述：

Regardless of the execution context that is specified in the module, the following actions always apply:

When the module is executed, the Database Engine first verifies that the user executing the module has EXECUTE permission on the module.

Ownership chaining rules continue to apply. This means if the owners of the calling and called objects are the same, no permissions are checked on the underlying objects.

因此要让SQL在用户在执行存储过程的时候对引用的对象进行权限检查，需要将存储过程与其引用的对象放在不同的schema下面，被不同的数据库用户所拥有。比如可以将表转到guest schema下面：

ALTER SCHEMA guest TRANSFER dbo.tbl\_TestPer

go

再修改存储过程使其指向转移过的表：

ALTER PROC dbo.sp\_TestPer1

WITH EXECUTE AS CALLER

AS

BEGIN

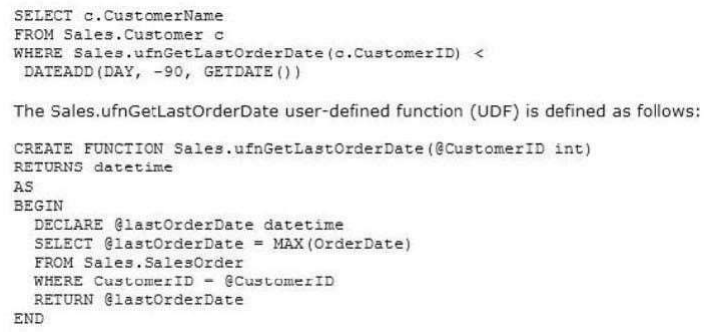
SELECT SYSTEM\_USER, CURRENT\_USER

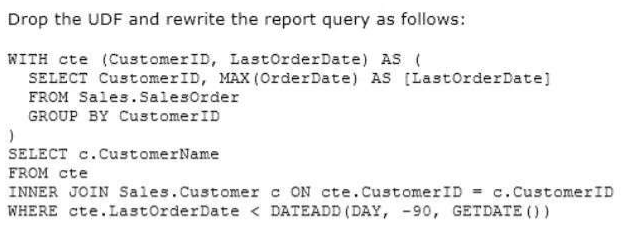
SELECT \* FROM guest.tbl\_TestPer

END

这时再用testuser执行存储过程将遇到错误229。

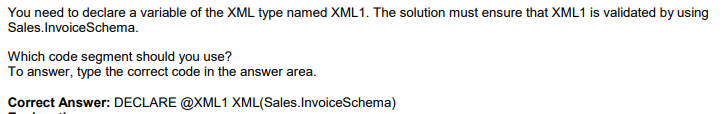
**CTE is faster than user defined function**





**Declare an xml variable**

**CREATE XML SCHEMA COLLECTION**

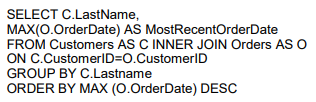
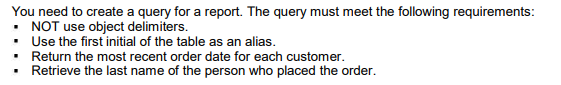


DECLARE @MySchemaCollection nvarchar(max)

Set @MySchemaCollection = N' copy the schema collection here'

CREATE XML SCHEMA COLLECTION MyCollection AS @MySchemaCollection

**Aggregation function can be used in order clause**



**Advanced update statement**

UPDATE

Table\_A

SET

Table\_A.col1 = Table\_B.col1,

Table\_A.col2 = Table\_B.col2

FROM

Some\_Table AS Table\_A

INNER JOIN Other\_Table AS Table\_B

ON Table\_A.id = Table\_B.id

WHERE

Table\_A.col3 = 'cool'

UPDATE BlogEntry

SET Summary.WRITE(N' This is in a draft stage', NULL, 0)

FROM ( SELECT TOP(10) Id FROM BlogEntry ORDER BY EntryDateTime DESC) AS s

WHERE BlogEntry.Id = s.ID

**SQL Server游标**

什么是游标

结果集，结果集就是select查询之后返回的所有行数据的集合。

游标则是处理结果集的一种机制吧，它可以定位到结果集中的某一行，多数据进行读写，也可以移动游标定位到你所需要的行中进行操作数据。

一般复杂的存储过程，都会有游标的出现，他的用处主要有：

定位到结果集中的某一行。

对当前位置的数据进行读写。

可以对结果集中的数据单独操作，而不是整行执行相同的操作。

是面向集合的数据库管理系统和面向行的程序设计之间的桥梁。

游标的分类

根据游标检测结果集变化的能力和消耗资源的情况不同，SQL Server支持的API服务器游标分为一下4种：

静态游标: 静态游标的结果集，在游标打开的时候建立在TempDB中，不论你在操作游标的时候，如何操作数据库，游标中的数据集都不会变。例如你在游标打开的时候，对游标查询的数据表数据进行增删改，操作之后，静态游标中select的数据依旧显示的为没有操作之前的数据。如果想与操作之后的数据一致，则重新关闭打开游标即可。

动态游标:这个则与静态游标相对，滚动游标时，动态游标反应结果集中的所有更改。结果集中的行数据值、顺序和成员在每次提取时都会变化。所有用户做的增删改语句通过游标均可见。如果使用API函数或T-SQL Where Current of子句通过游标进行更新，他们将立即可见。在游标外部所做的更新直到提交时才可见。

只进游标：只进游标不支持滚动，只支持从头到尾顺序提取数据，数据库执行增删改，在提取时是可见的，但由于该游标只能进不能向后滚动，所以在行提取后对行做增删改是不可见的。

键集驱动游标：打开键集驱动游标时，该有表中的各个成员身份和顺序是固定的。打开游标时，结果集这些行数据被一组唯一标识符标识，被标识的列做删改时，用户滚动游标是可见的，如果没被标识的列增该，则不可见，比如insert一条数据，是不可见的，若可见，须关闭重新打开游标。

静态游标在滚动时检测不到表数据变化，但消耗的资源相对很少。动态游标在滚动时能检测到所有表数据变化，但消耗的资源却较多。键集驱动游标则处于他们中间，所以根据需求建立适合自己的游标，避免资源浪费。。

游标的生命周期

游标的生命周期包含有五个阶段：声明游标、打开游标、读取游标数据、关闭游标、释放游标。

DECLARE vend\_cursor CURSOR

FOR SELECT \* FROM Purchasing.Vendor

OPEN vend\_cursor

FETCH NEXT FROM vend\_cursor;

[MSSQL]CROSS 和 OUTER apply 区别详解

SQL Server 2005 新增cross 和outer 联接语句，增加这两个东东有啥作用呢？

我们知道有个 SQL Server 2000 中有个cross join 是用于交叉联接的。实际上增加cross 和outer

是用于交叉联接表值函数（返回表结果集的函数）的， 更重要的是这个函数的参数是另一个表中的字段。这个解释可能有些含混不请，请看下面的例子：

-- 1. cross join 联接两个表

select \* from TABLE\_1as T1

cross join TABLE\_2as T2

-- 2. cross join 联接表和表值函数，表值函数的参数是个“常量”

select \* from TABLE\_1 T1

cross join FN\_TableValue(100)

-- 3. cross join 联接表和表值函数，表值函数的参数是“表T1中的字段”

select \*

from TABLE\_1 T1

cross join FN\_TableValue(T1.column\_a)

Msg 4104,Level 16, State 1, Line1

The multi-part identifier "T1.column\_a" couldnot be bound.

最后的这个查询的语法有错误。在 cross join 时，表值函数的参数不能是表 T1 的字段， 为啥不能这样做呢？我猜可能微软当时没有加这个功能：），后来有客户抱怨后， 于是微软就增加了cross 和outer 来完善，请看cross ,outer 的例子：

-- 4. cross

select \* from TABLE\_1 T1 cross FN\_TableValue(T1.column\_a)

-- 5. outer

select \* from TABLE\_1 T1 outer FN\_TableValue(T1.column\_a)

cross 和outer 对于 T1 中的每一行都和派生表（表值函数根据T1当前行数据生成的动态结果集） 做了一个交叉联接。cross 和outer 的区别在于： 如果根据 T1 的某行数据生成的派生表为空，cross 后的结果集 就不包含 T1 中的这行数据，而outer 仍会包含这行数据，并且派生表的所有字段值都为NULL。

下面的例子摘自微软 SQL Server 2005 联机帮助，它很清楚的展现了cross

和outer 的不同之处：

-- cross

select \* from Departmentsas D cross fn\_getsubtree(D.deptmgrid)as ST

deptid deptname deptmgrid empid empname mgrid lvl

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

1 HR 2 2 Andrew 1 0

1 HR 2 5 Steven 2 1

1 HR 2 6 Michael 2 1

2 Marketing 7 7 Robert 3 0

2 Marketing 7 11 David 7 1

2 Marketing 7 12 Ron 7 1

2 Marketing 7 13 Dan 7 1

2 Marketing 7 14 James 11 2

3 Finance 8 8 Laura 3 0

4 R&D 9 9 Ann 3 0

5 Training 4 4 Margaret 1 0

5 Training 4 10 Ina 4 1

(12 row(s) affected)

-- outer

select \* from Departmentsas D outer fn\_getsubtree(D.deptmgrid)as ST

deptid deptname deptmgrid empid empname mgrid lvl

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

1 HR 2 2 Andrew 1 0

1 HR 2 5 Steven 2 1

1 HR 2 6 Michael 2 1

2 Marketing 7 7 Robert 3 0

2 Marketing 7 11 David 7 1

2 Marketing 7 12 Ron 7 1

2 Marketing 7 13 Dan 7 1

2 Marketing 7 14 James 11 2

3 Finance 8 8 Laura 3 0

4 R&D 9 9 Ann 3 0

5 Training 4 4 Margaret 1 0

5 Training 4 10 Ina 4 1

6 Gardening NULL NULL NULL NULL NULL

(13 row(s) affected)

注意 outer

结果集中多出的最后一行。 当 Departments 的最后一行在进行交叉联接时：deptmgrid 为NULL，fn\_getsubtree(D.deptmgrid) 生成的派生表中没有数据，但outer

仍会包含这一行数据，这就是它和cross join 的不同之处。

下面是完整的测试代码，你可以在 SQL Server 2005 联机帮助上找到：

-- create Employees table and insert values

IF OBJECT\_ID('Employees')IS NOTNULL

DROP TABLE Employees

GO

CREATE TABLE Employees

(

empid INT NOTNULL,

mgrid INT NULL,

empname VARCHAR(25)NOT NULL,

salary MONEY NOT NULL

)

GO

IF OBJECT\_ID('Departments')IS NOTNULL

DROP TABLE Departments

GO

-- create Departments table and insert values

CREATE TABLE Departments

(

deptid INT NOT NULL PRIMARYKEY,

deptname VARCHAR(25)NOT NULL,

deptmgrid INT

)

GO

-- fill datas

INSERT INTO employeesVALUES (1,NULL,'Nancy',00.00)

INSERT INTO employeesVALUES (2,1,'Andrew',00.00)

INSERT INTO employeesVALUES (3,1,'Janet',00.00)

INSERT INTO employeesVALUES (4,1,'Margaret',00.00)

INSERT INTO employeesVALUES (5,2,'Steven',00.00)

INSERT INTO employeesVALUES (6,2,'Michael',00.00)

INSERT INTO employeesVALUES (7,3,'Robert',00.00)

INSERT INTO employeesVALUES (8,3,'Laura',00.00)

INSERT INTO employeesVALUES (9,3,'Ann',00.00)

INSERT INTO employeesVALUES (10,4,'Ina',00.00)

INSERT INTO employeesVALUES (11,7,'David',00.00)

INSERT INTO employeesVALUES (12,7,'Ron',00.00)

INSERT INTO employeesVALUES (13,7,'Dan',00.00)

INSERT INTO employeesVALUES (14,11,'James',00.00)

INSERT INTO departmentsVALUES (1,'HR',2)

INSERT INTO departmentsVALUES (2,'Marketing',7)

INSERT INTO departmentsVALUES (3,'Finance',8)

INSERT INTO departmentsVALUES (4,'R&D',9)

INSERT INTO departmentsVALUES (5,'Training',4)

INSERT INTO departmentsVALUES (6,'Gardening',NULL)

GO

--SELECT \* FROM departments

-- table-value function

IF OBJECT\_ID('fn\_getsubtree')IS NOTNULL

DROP FUNCTION fn\_getsubtree

GO

CREATE FUNCTION dbo.fn\_getsubtree(@empidAS INT)

RETURNS TABLE

AS

RETURN(

WITH Employees\_Subtree(empid, empname, mgrid, lvl)

AS

(

-- Anchor Member (AM)

SELECT empid, empname, mgrid,0

FROM employees

WHERE empid= @empid

UNION ALL

-- Recursive Member (RM)

SELECT e.empid, e.empname, e.mgrid, es.lvl+1

FROM employeesAS e

join employees\_subtreeAS es

ON e.mgrid= es.empid

)

SELECT \* FROM Employees\_Subtree

)

GO

-- cross query

SELECT \* FROM DepartmentsAS D

CROSS

fn\_getsubtree(D.deptmgrid)AS ST

-- outer query

SELECT \* FROM DepartmentsAS D

[**SQL SERVER函数——表值函数的处理**](https://www.cnblogs.com/peaceli/archive/2009/09/08/1561451.html)

     有些情况可能用下表值函数，表值函数主要用于数据计算出来返回结果集，可以带参数（和视图的一个大的区别），如果函数中没有过多的逻辑处理，如变量的定义，判断等，  
表值函数返回结果集可以简单向下面这么写：

[复制代码](javascript:void(0);)

CREATE FUNCTION Fun\_GetReportNews(@type varchar(**10**))  
RETURNS TABLE  
AS   
RETURN  
(  
  SELECT TPR\_ID,TPR\_Title,TPR\_Date FROM TP\_ReportNews WHERE TPR\_Type = @type  
)

[复制代码](javascript:void(0);)

调用的时候就 SELECT XX FROM Fun\_GetReprotNews('xx')  
  
如果函数中要定义变量，进行判断计算处理什么的，写法有点不一样了，要定义表变量才行，表值函数里是不允许创建临时表的，只能是表变量。  
举个简单的写法样式，如下：

[复制代码](javascript:void(0);)

CREATE FUNCTION FUN\_GetInfoList(@type varchar(**10**))  
RETURNS @Table TABLE(TPR\_ID int,TPR\_Title nvarchar(**100**),TPR\_PubDate datetime)  
AS  
BEGIN  
  DECLARE @a varchar(**10**)  
  SELECT @a = xx FROM XX WHERE xx = @type  
    INSERT @Table SELECT XX,XX,XX FROM TableName WHERE XX = @a --表变量里定义的列数和取值列数要一致  
RETURN  
END

[复制代码](javascript:void(0);)

如果进行多表操作，可以在函数体内定义表变量来存放结果集再进行关联查询。  
标量值函数也贴一个样子好了，老掉牙的了，呵呵~~

[复制代码](javascript:void(0);)

CREATE FUNCTION FUN\_DataFormat (@strDate datetime)    
RETURNS varchar(**20**)  AS    
BEGIN   
      
    declare @date varchar(**20**)  
      set @date = DATENAME(YY,@strDate)+'年'+Convert(VARCHAR,MONTH(@strDate))+'月'+Convert(VARCHAR,DAY(@strDate))+'日'  
    return @date  
END

[复制代码](javascript:void(0);)

访问标量值函数时一般在函数名前加dbo，不然会被认为是系统内置函数，却因又不是系统内置函数而会报错。  
上面的可以这么测试   
select dbo.FUN\_DataFormat(getdate())  
就忽悠这些了~~~~~~~  
表变量和临时表的区别及函数和存储过程的区别，大伙可以在博客园上博一把，有很多说明文章比较详细的了

[SQL Server 2012中的ColumnStore Index](https://www.cnblogs.com/CareySon/archive/2012/03/09/2387475.html)

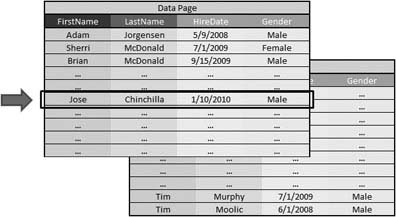
Columnstore index is a new type of index introduced in SQL Server 2012. It is a column-based non-clustered index geared toward increasing query performance for workloads that involve large amounts of data, typically found in data warehouse fact tables.

This new type of index stores data column-wise instead of row-wise, as indexes currently do. For example, consider an Employee table containing employee data, as shown in Table 1.

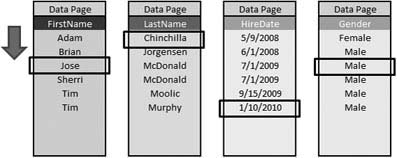
**Table 1: Sample Employee Table**

|  |  |  |  |
| --- | --- | --- | --- |
| FirstName | LastName | HireDate | Gender |
| Adam | Jorgensen | 5/9/2008 | Male |
| Sherri | McDonald | 7/1/2009 | Female |
| Brian | McDonald | 09/15/2009 | Male |
| Jose | Chinchilla | 1/10/2010 | Male |
| Tim | Murphy | 7/1/2009 | Male |
| Tim | Moolic | 6/1/2008 | Male |

In a row-based index, the data in the Employee table is stored in one or more data pages, as shown in Figure 1.



 In a column-based index, the data in the Employee table is stored in separate pages for each of the columns, as shown in Figure 2.



 Performance advantages in columnstore indexes are possible by leveraging the VertiPaq compression technology, which enables large amounts of data to be compressed in-memory. This in-memory compressed store reduces the number of disk reads and increases buffer cache hit ratios because only the smaller column-based data pages that need to satisfy a query are moved into memory.

For wide tables, such as those commonly found in data warehouses, columnstore indexes come in handy as you essentially reduce the amount and size of data needed to be accessed for any given query. For example, consider the following query:

SELECT FirstName, LastName, FROM EmployeeTable

WHERE HireDate >= ‘1/1/2010’

A column-store index is more efficient for this example because only one smaller-sized (compressed) data page is needed to satisfy the query. In this case, the columnstore index for the HireDate column satisfies the WHERE clause. A row-based index is not as efficient because it may need to load one or more larger-sized data pages into memory and read the entire rows, including columns not needed to satisfy the query. A larger-sized data page and additional unnecessary columns increases data size, memory usage, disk reads, and overall query time. Imagine if this table had 20 or more columns!

Columnstore indexes have some requirements and limitations, as shown in Table 2.

**Table 2: Requirements and Limitations of Columnstore Index**

|  |  |
| --- | --- |
| Description | Requirement/Limitation |
| No. of columnstore indexes per table | 1 |
| Index record size limit of 900 bytes | No limit/Not applicable |
| Index limit of 16 key columns | No limit/Not applicable |
| Table partitioning support | Yes, as a partition aligned index. |
| Can be combined with row-based indexes? | Yes, if clustered index, all columns must be present in columnstore index. |
| Update, Delete, Insert, Merge supported? | No, columnstore indexes are read-only but workarounds exist. Refer to Books Online: Best Practices: Updating Data in a Columnstore Index. |
| Data types that can be included in a columnstore index | Char, varchar except varchar(max), nchar, nvarchar except nvarchar(max), decimal and numeric except with precision greater than 18 digits, int, bigint, smallint, tinyint, float, real, bit, money, smallmoney, all date and time data types except datetimeoffset with scale greater than 2. |
| Data types that cannot be included in a columnstore index | Binary, varbinary, ntext, text, image, varchar(max), nvarchar(max), uniqueidentifier, rowversion, timestamp, sql\_variant, decimal and numeric with precision greater than 18 digits, datetimeoffset with scale greater than 2, CLR types including hierarchyid and spatial types, xml. |

 Following is the basic syntax to create a columnstore index:

 CREATE COLUMNSTORE INDEX idx\_cs1

ON EmployeeTable (FirstName, LastName, HireDate, Gender)

You can also create columnstore indexes using SQL Server Management Studio. Simply navigate to the Indexes section of the table, and select New Index > Non-clustered Columnstore Index.

# 使用 FORCESEEK 表提示

FORCESEEK 表提示强制查询优化器仅使用索引查找操作作为访问查询引用的表或者视图中的数据的路径。 您可使用该表提示覆盖查询优化器选择的默认计划，从而避免因低效的查询计划而导致的性能问题。 例如，如果计划中包含表扫描运算符或者索引扫描运算符，且相应的表导致执行查询期间读取十分频繁，如 [STATISTICS IO](https://docs.microsoft.com/zh-cn/previous-versions/sql/sql-server-2008-r2/ms184361%28v%3dsql.105%29) 输出中所示，则强制索引查找操作可能会获得更好的查询性能。 如果优化器因基数或开销估计不准确而在编译计划时倾向于选择扫描操作，则更是如此。

FORCESEEK 适用于聚集索引查找和非聚集索引查找操作。 可以在 SELECT 语句的 FROM 子句和 UPDATE 或 DELETE 语句的 FROM <table\_source> 子句中为任何表或视图指定 FORCESEEK。

| **注意事项注意** |
| --- |
| 由于 SQL Server 查询优化器通常会为查询选择最优执行计划，因此我们建议，只有在最后迫不得已的情况下才可由资深的开发人员和数据库管理员使用提示。 |

## 评估查询计划对 FORCESEEK 的适用性

如果查询计划对表或视图使用表扫描或索引扫描运算符，FORCESEEK 表提示可能有用，但索引查找运算符可能更为高效。 请考虑以下查询和后续执行计划。

复制

USE AdventureWorks2008R2;

GO

SELECT \*

FROM Sales.SalesOrderHeader AS h

INNER JOIN Sales.SalesOrderDetail AS d

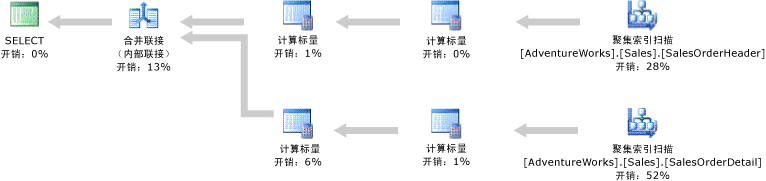
ON h.SalesOrderID = d.SalesOrderID

WHERE h.TotalDue > 100

AND (d.OrderQty > 5 OR d.LineTotal < 1000.00);

GO

下列执行计划显示查询优化器选择了聚集索引扫描运算符来访问两个表中的数据。



如下列查询所示，可通过指定 FORCESEEK 提示强制查询优化器对 Sales.SalesOrderDetail 表执行查找操作。

复制

USE AdventureWorks2008R2;

GO

SELECT \*

FROM Sales.SalesOrderHeader AS h

INNER JOIN Sales.SalesOrderDetail AS d WITH (FORCESEEK)

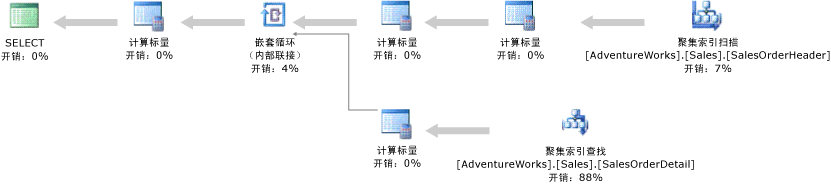
ON h.SalesOrderID = d.SalesOrderID

WHERE h.TotalDue > 100

AND (d.OrderQty > 5 OR d.LineTotal < 1000.00);

GO

下列执行计划显示了在查询中使用 FORCESEEK 提示的结果。 聚集索引查找操作被用来访问 Sales.SalesOrderDetail 表中的数据。



### 支持索引并集和交集

FORCESEEK 提示支持索引并集和交集。 此提示使查询优化器更可能使用这些技术。 为了避免延长简单查询的编译时间，通常仅在根据将列的基数和选择性考虑在内的规则时才会选择索引并集和交集。 不过，指定 FORCESEEK 提示后，会跳过此类规则，而始终将这些技术考虑在内。 例如，考虑以下查询：

复制

SELECT \* FROM T WITH(FORCESEEK) WHERE T.a = 1 AND T.b = 2;

如果表 T 中的列 a 和列 b 具有单独的非聚集索引，则可选择索引交集计划。 也就是说，此计划包含对列 a 的非聚集索引查找操作和对列 b 的非聚集索引查找操作，并且在对基表执行查找操作之前对所得的索引键集求交集。

下例中选择了一个索引并集计划。 也就是说，计划包含对列 a 的查找操作和对列 b 的查找操作，并且在对基表执行查找操作之前对所得的索引键集求并集。

复制

SELECT \* FROM T WITH(FORCESEEK) WHERE T.a = 1 OR T.b = 2;

### 对使用 LIKE 或 IN 的查询应用 FORCESEEK

当查询将 IN 或 LIKE 用做搜索谓词时，查询优化器规则和不准确的基数估计也会导致优化器执行表扫描或索引扫描操作，而不是索引查找操作。

下面的示例演示将 LIKE 或 IN 用做搜索谓词时，FORCESEEK 提示如何强制查询优化器执行索引查找操作，而不是表扫描操作。 若要查看查询执行计划，请在运行此示例前单击\*\*“包括实际的执行计划”\*\*工具栏按钮。

复制

USE tempdb;

GO

DROP TABLE t;

GO

CREATE TABLE t(i int UNIQUE, j int, vc varchar(100));

CREATE INDEX t\_vc ON t(vc);

GO

DECLARE @p1 int, @p2 int, @p3 int, @p4 int, @p5 int;

SELECT \* FROM t WHERE i IN (@p1, @p2, @p3, @p4, @p5);

GO

DECLARE @p1 int, @p2 int, @p3 int, @p4 int, @p5 int;

SELECT \* FROM t WITH (FORCESEEK) WHERE i IN (@p1, @p2, @p3, @p4, @p5);

GO

SELECT \* FROM t WHERE vc LIKE 'Test%';

GO

SELECT \* FROM t WITH (FORCESEEK) WHERE vc LIKE 'Test%';

GO

DECLARE @vc varchar(100);

SELECT \* FROM t WHERE vc LIKE @vc;

GO

DECLARE @vc varchar(100);

SELECT \* FROM t WITH (FORCESEEK) where vc like @vc;

GO

## 对视图使用 FORCESEEK

指定 FORCESEEK 时有无索引提示均可。 将 FORCESEEK 表提示应用到某一视图或索引视图时，FORCESEEK 提示将递归传播到该视图扩展版本中的所有表。 如果已指定索引提示，则会将其忽略。 如果并非每个基础表都至少包含一个索引，则无法找到计划并返回错误 8622。

对索引视图的引用同时使用 FORCESEEK 和 NOEXPAND 提示时，则使用此索引视图时无需事先将其展开。 FORCESEEK 提示将直接应用到索引视图，这与处理表的方式相同。

如果将 FORCESEEK 提示应用至表引用，则表引用无法参与索引视图匹配。 但是，查询中其他不受 FORCESEEK 提示影响的部分可参与索引视图匹配。 这一点与使用 INDEX 提示时索引视图匹配的行为是相似的。

## 最佳做法注意事项

以下为建议的最佳做法：

* 在使用 FORCESEEK 表提示之前，请确保数据库中的统计信息是最新的，且准确无误。

最新的统计信息有助于优化器实现对不同查询计划的开销的准确估算，并选择高质量的计划。 因此，建议对于每个用户数据库，都将 AUTO\_CREATE\_STATISTICS 和 AUTO\_UPDATE\_STATISTICS 设置为 ON（默认值）。 或者，您可以使用 [UPDATE STATISTICS](https://docs.microsoft.com/zh-cn/previous-versions/sql/sql-server-2008-r2/ms187348%28v%3dsql.105%29) 语句手动更新表或视图的统计信息。

* 评估查询看其是否存在可能导致不准确基数估计或开销估计的项，如果可能则删除这些项。 例如，用参数或文字替换局部变量并限制查询中多语句表值函数和表变量的使用。 有关要查找的其他项的更多信息，请参阅 [Microsoft SQL Server 2005 中查询优化器使用的统计信息](http://go.microsoft.com/fwlink/?linkid=62104)。
* 除非在必要情况下，否则请不要将 INDEX 提示与 FORCESEEK 一起使用。 也就是说，如果单独使用 FORCESEEK 就可生成满足需要的计划，则同时使用 INDEX 提示会过分限制优化器可做的选择。 而且，在更改表的物理架构以删除 INDEX 提示中指定的索引时，此提示将导致查询失败。 相比之下，只要应用 FORCESEEK 提示的表中至少存在一个可使用的索引，则即使在您更改索引结构时查询仍能编译。
* 请不要将 INDEX 提示 INDEX (0) 与 FORCESEEK 提示一起使用。 INDEX (0) 强制对基表进行扫描。 在与 FORCESEEK 同时使用时，将无法找到计划并返回错误 8622。
* 请不要将 USE PLAN 查询提示与 FORCESEEK 提示一起使用。 否则，将忽略 FORCESEEK 提示。

# SET TRANSACTION ISOLATION LEVEL

SET TRANSACTION ISOLATION LEVEL

{ READ UNCOMMITTED

| READ COMMITTED

| REPEATABLE READ

| SNAPSHOT

| SERIALIZABLE

}

READ UNCOMMITTED  
Specifies that statements can read rows that have been modified by other transactions but not yet committed.

Transactions running at the READ UNCOMMITTED level do not issue shared locks to prevent other transactions from modifying data read by the current transaction. READ UNCOMMITTED transactions are also not blocked by exclusive locks that would prevent the current transaction from reading rows that have been modified but not committed by other transactions. When this option is set, it is possible to read uncommitted modifications, which are called dirty reads. Values in the data can be changed and rows can appear or disappear in the data set before the end of the transaction. This option has the same effect as setting NOLOCK on all tables in all SELECT statements in a transaction. This is the least restrictive of the isolation levels.

In SQL Server, you can also minimize locking contention while protecting transactions from dirty reads of uncommitted data modifications using either:

* The READ COMMITTED isolation level with the READ\_COMMITTED\_SNAPSHOT database option set to ON.
* The SNAPSHOT isolation level. For more information about snapshot isolation, see [Snapshot Isolation in SQL Server](https://docs.microsoft.com/dotnet/framework/data/adonet/sql/snapshot-isolation-in-sql-server).

READ COMMITTED  
Specifies that statements cannot read data that has been modified but not committed by other transactions. This prevents dirty reads. Data can be changed by other transactions between individual statements within the current transaction, resulting in nonrepeatable reads or phantom data. This option is the SQL Server default.

The behavior of READ COMMITTED depends on the setting of the READ\_COMMITTED\_SNAPSHOT database option:

* If READ\_COMMITTED\_SNAPSHOT is set to OFF (the default), the Database Engine 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. The shared lock type determines when it will be released. Row locks are released before the next row is processed. Page locks are released when the next page is read, and table locks are released when the statement finishes.
* If READ\_COMMITTED\_SNAPSHOT is set to ON, 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.

**Important**

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. Additionally, an update made at the READ\_COMMITTED isolation level uses update locks on the data rows selected, whereas an update made at the SNAPSHOT isolation level uses row versions to select rows to update. For read operations, transaction isolation levels primarily define the level of protection from the effects of modifications made by other transactions. See the [**Transaction Locking and Row Versioning Guide**](https://docs.microsoft.com/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide) for more information.

**Note**

Snapshot isolation supports FILESTREAM data. Under snapshot isolation mode, FILESTREAM data read by any statement in a transaction will be the transactionally consistent version of the data that existed at the start of the transaction.

When the READ\_COMMITTED\_SNAPSHOT database option is ON, you can use the READCOMMITTEDLOCK table hint to request shared locking instead of row versioning for individual statements in transactions running at the READ COMMITTED isolation level.

**Note**

When you set 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.

REPEATABLE READ  
Specifies that statements cannot read data that has been modified but not yet committed by other transactions and that no other transactions can modify data that has been read by the current transaction until the current transaction completes.

Shared locks are placed on all data read by each statement in the transaction and are held until the transaction completes. This prevents other transactions from modifying any rows that have been read by the current transaction. Other transactions can insert new rows that match the search conditions of statements issued by the current transaction. If the current transaction then retries the statement it will retrieve the new rows, which results in phantom reads. Because shared locks are held to the end of a transaction instead of being released at the end of each statement, concurrency is lower than the default READ COMMITTED isolation level. Use this option only when necessary.

SNAPSHOT  
Specifies that data read by any statement in a transaction will be the transactionally consistent version of the data that existed at the start of the transaction. The transaction can only recognize data modifications that were committed before the start of the transaction. Data modifications made by other transactions after the start of the current transaction are not visible to statements executing in the current transaction. The effect is as if the statements in a transaction get a snapshot of the committed data as it existed at the start of the transaction.

Except when a database is being recovered, SNAPSHOT transactions do not request locks when reading data. SNAPSHOT transactions reading data do not block other transactions from writing data. Transactions writing data do not block SNAPSHOT transactions from reading data.

During the roll-back phase of a database recovery, SNAPSHOT transactions will request a lock if an attempt is made to read data that is locked by another transaction that is being rolled back. The SNAPSHOT transaction is blocked until that transaction has been rolled back. The lock is released immediately after it has been granted.

The ALLOW\_SNAPSHOT\_ISOLATION database option must be set to ON before you can start a transaction that uses the SNAPSHOT isolation level. If a transaction using the SNAPSHOT isolation level accesses data in multiple databases, ALLOW\_SNAPSHOT\_ISOLATION must be set to ON in each database.

A transaction cannot be set to SNAPSHOT isolation level that started with another isolation level; doing so will cause the transaction to abort. If a transaction starts in the SNAPSHOT isolation level, you can change it to another isolation level and then back to SNAPSHOT. A transaction starts the first time it accesses data.

A transaction running under SNAPSHOT isolation level can view changes made by that transaction. For example, if the transaction performs an UPDATE on a table and then issues a SELECT statement against the same table, the modified data will be included in the result set.

**Note**

Under snapshot isolation mode, FILESTREAM data read by any statement in a transaction will be the transactionally consistent version of the data that existed at the start of the transaction, not at the start of the statement.

SERIALIZABLE  
Specifies the following:

* Statements cannot read data that has been modified but not yet committed by other transactions.
* No other transactions can modify data that has been read by the current transaction until the current transaction completes.
* Other transactions cannot insert new rows with key values that would fall in the range of keys read by any statements in the current transaction until the current transaction completes.

Range locks are placed in the range of key values that match the search conditions of each statement executed in a transaction. This blocks other transactions from updating or inserting any rows that would qualify for any of the statements executed by the current transaction. This means that if any of the statements in a transaction are executed a second time, they will read the same set of rows. The range locks are held until the transaction completes. This is the most restrictive of the isolation levels because it locks entire ranges of keys and holds the locks until the transaction completes. Because concurrency is lower, use this option only when necessary. This option has the same effect as setting HOLDLOCK on all tables in all SELECT statements in a transaction.

## Remarks

Only one of the isolation level options can be set at a time, and it remains set for that connection until it is explicitly changed. All read operations performed within the transaction operate under the rules for the specified isolation level unless a table hint in the FROM clause of a statement specifies different locking or versioning behavior for a table.

The transaction isolation levels define the type of locks acquired on read operations. Shared locks acquired for READ COMMITTED or REPEATABLE READ are generally row locks, although the row locks can be escalated to page or table locks if a significant number of the rows in a page or table are referenced by the read. If a row is modified by the transaction after it has been read, the transaction acquires an exclusive lock to protect that row, and the exclusive lock is retained until the transaction completes. For example, if a REPEATABLE READ transaction has a shared lock on a row, and the transaction then modifies the row, the shared row lock is converted to an exclusive row lock.

With one exception, you can switch from one isolation level to another at any time during a transaction. The exception occurs when changing from any isolation level to SNAPSHOT isolation. Doing this causes the transaction to fail and roll back. However, you can change a transaction started in SNAPSHOT isolation to any other isolation level.

When you change a transaction from one isolation level to another, resources that are read after the change are protected according to the rules of the new level. Resources that are read before the change continue to be protected according to the rules of the previous level. For example, if a transaction changed from READ COMMITTED to SERIALIZABLE, the shared locks acquired after the change are now held until the end of the transaction.

If you issue SET TRANSACTION ISOLATION LEVEL in a stored procedure or trigger, when the object returns control the isolation level is reset to the level in effect when the object was invoked. For example, if you set REPEATABLE READ in a batch, and the batch then calls a stored procedure that sets the isolation level to SERIALIZABLE, the isolation level setting reverts to REPEATABLE READ when the stored procedure returns control to the batch.

**Note**

User-defined functions and common language runtime (CLR) user-defined types cannot execute SET TRANSACTION ISOLATION LEVEL. However, you can override the isolation level by using a table hint. For more information, see [**Table Hints (Transact-SQL)**](https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-table?view=sql-server-2017).

When you use sp\_bindsession to bind two 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 sessions bound to it.

SET TRANSACTION ISOLATION LEVEL takes effect at execute or run time, and not at parse time.

Optimized bulk load operations on heaps block queries that are running under the following isolation levels:

* SNAPSHOT
* READ UNCOMMITTED
* READ COMMITTED using row versioning

Conversely, queries that run under these isolation levels block optimized bulk load operations on heaps. For more information about bulk load operations, see [Bulk Import and Export of Data (SQL Server)](https://docs.microsoft.com/en-us/sql/relational-databases/import-export/bulk-import-and-export-of-data-sql-server?view=sql-server-2017).

FILESTREAM-enabled databases support the following transaction isolation levels.

## Isolation level Transact SQL access File system access

## Read uncommitted SQL Server 2017 Unsupported

## Read committed SQL Server 2017 SQL Server 2017

## Repeatable read SQL Server 2017 Unsupported

## Serializable SQL Server 2017 Unsupported

## Read committed snapshot SQL Server 2017 SQL Server 2017

## Snapshot SQL Server 2017 SQL Server 2017

## Examples

The following example sets the TRANSACTION ISOLATION LEVEL for the session. For each Transact-SQL statement that follows, SQL Server holds all of the shared locks until the end of the transaction.

Copy

USE AdventureWorks2012;

GO

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

GO

BEGIN TRANSACTION;

GO

SELECT \*

FROM HumanResources.EmployeePayHistory;

GO

SELECT \*

FROM HumanResources.Department;

GO

COMMIT TRANSACTION;

GO

**Transaction Isolation Level**

# [理解Sql Server 事务隔离层级(](https://www.cnblogs.com/dimg/p/7269008.html)

关于Sql Server 事务隔离级别，百度百科是这样描述的

隔离级别：一个事务必须与由其他事务进行的资源或数据更改相隔离的程度。隔离级别从允许的并发副作用（例如，脏读或虚拟读取）的角度进行描述。

隔离级别共5种：

read uncommitted | 0 未提交读, 允许脏读, 读出uncommitted数据  
read committed | 1 已提交读, 不允许脏读, 不可重复读,default  
repeatable read | 2 可重复读, 不允许脏读, 可重复读, 允许”幻读, 保证在一个事务中的两个读操作之间，其他的事务不能修改当前事务读取的数据，该级别事务获取数据前必须先获得共享锁同时获得的共享锁不立即释放一直保持共享锁至事务完成  
serializable | 3 可序列化, 这是最高级别的隔离，串行化读，事务只能一个一个执行，不允许脏读,不可重复读, 不允许幻读。

snapshot(snapshot, READ COMMITTED SNAPSHOT)  
snapshot 快照(2005版本以后新加）读取数据时，可以保证读操作读取的行是事务开始时可用的最后提交版本。这意味着这种隔离级别可以保证读取的是已经提交过的数据，并且可以实现可重复读，也能确保不会幻读。

READ COMMITTED SNAPSHOT也是基于行版本控制,但是READ COMMITTED SNAPSHOT的隔离级别是读操作之前的最后已提交版本,而不是事务前的已提交版本，有点类似前面的READ COMMITTED能保证已提交读,但是不能保证可重复读,不能避免幻读,但是又比 READ COMMITTED隔离级别多出了不需要获取共享锁就可以读取数据

要启用READ COMMITTED SNAPSHOT隔离级别同样需要修改数据库选项，在回话1，回话2中执行以下操作(执行下面的操作当前连接必须是数据库的唯一连接，可以通过查询已连接当前数据库的进程,然后KILL掉那些进程,然后再执行该操作,否则可能无法执行成功)

以下面的图为例，在事务A中会根据条件读取TABLE1，且读两次，事务B中会对同样的表TABLE1同样的条件进行UPDATE，在对表数据更新时会获得排他锁直至事务提交，

这样在事务A中如果设置不同的隔离级别就会有不同的效果。

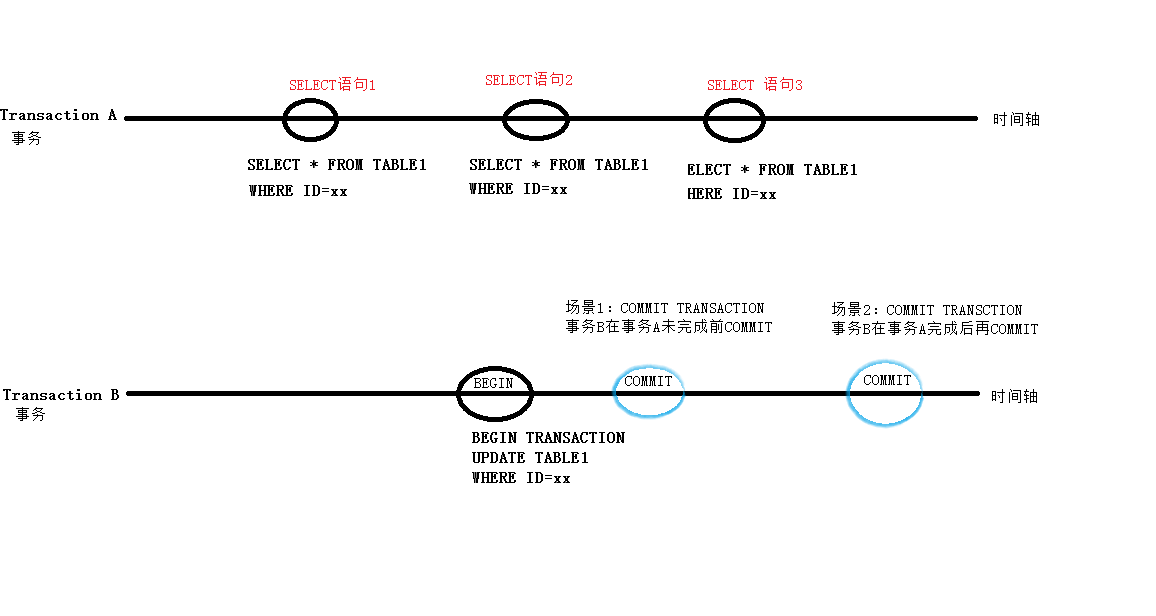
1、隔离级别设置为read uncommitted，允许脏读，不管事务B何时Commit，事务A中的SELECT语句2都可以读出数据（有可能是脏数据，因事务B可能会ROLLBACK)，且与语句1的数据不同。

2、隔离级别设置为read committed，不允许脏读，但允许“不可重复读”，即事务A中可以多次读，不管事务B中是否Commit，如果SELECT语句3执行时事务B还没有Commit，读取结果与SELECT 语句1相同，反之则不同。

3、隔离级别设置为repeatable read，不允许脏读，也不允许“不可重复读”，但允许”幻读“，示例中，事务A中的SELECT语句3的执行一定要等到事务B Commit之后才能执行成功。

4、隔离级别设置为serializable，这是最高级别的隔离，串行化读，事务只能一个一个执行，避免了脏读、不可重复读、幻读。

5、隔离级别设置为snapshot ，读取数据时，可以保证读操作读取的行是事务开始时可用的最后提交版本。这意味着这种隔离级别可以保证读取的是已经提交过的数据，并且可以实现可重复读，也能确保不会幻读。



**SQL 事务隔离级别**

**概述**

     隔离级别用于决定如果控制并发用户如何读写数据的操作，同时对性能也有一定的影响作用。

**步骤**

事务隔离级别通过影响读操作来间接地影响写操作；可以在回话级别上设置事务隔离级别也可以在查询（表级别）级别上设置事务隔离级别。  
事务隔离级别总共有6个隔离级别：  
READ UNCOMMITTED(未提交读,读脏),相当于(NOLOCK)  
READ COMMITTED(已提交读,默认级别)  
REPEATABLE READ(可以重复读),相当于(HOLDLOCK)  
SERIALIZABLE(可序列化)  
SNAPSHOT(快照)  
READ COMMITTED SNAPSHOT(已经提交读隔离)  
对于前四个隔离级别：READ UNCOMMITTED<READ COMMITTED<REPEATABLE READ<SERIALIZABLE  
隔离级别越高,读操作的请求锁定就越严格,锁的持有时间久越长;所以隔离级别越高,一致性就越高,并发性就越低,同时性能也相对影响越大.

**获取事务隔离级别(isolation level)**

DBCC USEROPTIONS

**设置隔离**

[复制代码](javascript:void(0);)

设置回话隔离

SET TRANSACTION ISOLATION LEVEL <ISOLATION NAME>

--注意:在设置回话隔离时(REPEATABLE READ)两个单词需要用空格间隔开,但是在表隔离中可以粘在一起(REPEATABLEREAD)

设置查询表隔离

SELECT ....FROM <TABLE> WITH (<ISOLATION NAME>)

[复制代码](javascript:void(0);)

**1.READ UNCOMMITTED**

READ UNCOMMITTED:未提交读，读脏数据  
默认的读操作:需要请求共享锁,允许其他事物读锁定的数据但不允许修改.  
READ UNCOMMITTED:读操作不申请锁,运行读取未提交的修改,也就是允许读脏数据,读操作不会影响写操作请求排他锁.

 创建测试数据

[复制代码](javascript:void(0);)

IF OBJECT\_ID('Orders','U') IS NOT NULL DROP TABLE Orders

GO

CREATE TABLE Orders

(ID INT NOT NULL,

Price FLOAT NOT NULL

);

INSERT INTO Orders VALUES(**10**,**10.00**),(**11**,**11.00**),(**12**,**12.00**),(**13**,**13.00**),(**14**,**14.00**);

GO

SELECT ID,Price FROM Orders

[复制代码](javascript:void(0);)

新建回话1将订单10的价格加1

[复制代码](javascript:void(0);)

BEGIN TRANSACTION

UPDATE Orders

SET Price=Price+**1**

WHERE ID=**10**

SELECT ID,Price FROM Orders

WHERE ID=**10**

[复制代码](javascript:void(0);)



在另一个回话2中执行查询操作

[复制代码](javascript:void(0);)

首先不添加隔离级别,默认是READ COMMITTED,由于数据之前的更新操作使用了排他锁，所以查询一直在等待锁释放\*/

SELECT ID,Price FROM Orders

WHERE ID=**10**

---将查询的隔离级别设置为READ UNCOMMITTED允许未提交读，读操作之前不请求共享锁。

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT ID,Price FROM Orders

WHERE ID=**10**;

--当然也可以使用表隔离,效果是一样的

SELECT ID,Price FROM Orders WITH (NOLOCK)

WHERE ID=**10**

[复制代码](javascript:void(0);)



假设在回话1中对操作执行回滚操作,这样价格还是之前的10,但是回话2中则读取到的是回滚前的价格11,这样就属于一个读脏操作

ROLLBACK TRANSACTION

**2.READ COMMITTED**

READ COMMITTED（已提交读）是SQL SERVER默认的隔离级别，可以避免读取未提交的数据，隔离级别比**READ UNCOMMITTED**未提交读的级别更高;  
该隔离级别读操作之前首先申请并获得共享锁，允许其他读操作读取该锁定的数据，但是写操作必须等待锁释放，一般读操作读取完就会立刻释放共享锁。

新建回话1将订单10的价格加1,此时回话1的排他锁锁住了订单10的值

[复制代码](javascript:void(0);)

BEGIN TRANSACTION

UPDATE Orders

SET Price=Price+**1**

WHERE ID=**10**

SELECT ID,Price FROM Orders

WHERE ID=**10**

[复制代码](javascript:void(0);)



在回话2中执行查询，将隔离级别设置为READ COMMITTED

[复制代码](javascript:void(0);)

SET TRANSACTION ISOLATION LEVEL READ COMMITTED

SELECT ID,Price FROM Orders

WHERE ID=**10**

---由于READ COMMITTED需要申请获得共享锁，而锁与回话1的排他锁冲突，回话被堵塞，

----在回话1中执行事务提交

COMMIT TRANSACTION

/\*由于回话1事务提交，释放了订单10的排他锁，此时回话2申请共享锁成功查到到订单10的价格为修改后的价格11，READ COMMITTED由于是已提交读隔离级别，所以不会读脏数据.

\*/

[复制代码](javascript:void(0);)

重置数据

UPDATE Orders

SET Price=**10**

WHERE ID=**10**

注意：但是由于READ COMMITTED读操作一完成就立即释放共享锁,读操作不会在一个事务过程中保持共享锁,也就是说在一个事务的的两个查询过程之间有另一个回话对数据资源进行了更改,会导致一个事务的两次查询得到的结果不一致,这种现象称之为不可重复读.

**3.REPEATABLE READ**

REPEATABLE READ(可重复读)：保证在一个事务中的两个读操作之间，其他的事务不能修改当前事务读取的数据，该级别事务获取数据前必须先获得共享锁同时获得的共享锁不立即释放一直保持共享锁至事务完成，所以此隔离级别查询完并提交事务很重要。

在回话1中执行查询订单10,将回话级别设置为REPEATABLE READ

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

BEGIN TRANSACTION

SELECT ID,Price FROM Orders

WHERE ID=**10**

新建回话2修改订单10的价格

UPDATE Orders

SET Price=Price+**1**

WHERE ID=**10**

---由于回话1的隔离级别REPEATABLE READ申请的共享锁一直要保持到事务结束，所以回话2无法获取排他锁，处于等待状态

在回话1中执行下面语句，然后提交事务

SELECT ID,Price FROM Orders

WHERE ID=**10**

COMMIT TRANSACTION



回话1的两次查询得到的结果一致，前面的两个隔离级别无法得到一致的数据，此时事务已提交同时释放共享锁，回话2申请排他锁成功，对行执行更新

REPEATABLE READ隔离级别保证一个事务中的两次查询到的结果一致，同时保证了丢失更新  
丢失更新：两个事务同时读取了同一个值然后基于最初的值进行计算，接着再更新，就会导致两个事务的更新相互覆盖。  
例如酒店订房例子，两个人同时预定同一酒店的房间，首先两个人同时查询到还有一间房间可以预定，然后两个人同时提交预定操作，事务1执行number=1-0，同时事务2也执行number=1-0最后修改number=0,这就导致两个人其中一个人的操作被另一个人所覆盖，REPEATABLE READ隔离级别就能避免这种丢失更新的现象，当事务1查询房间时事务就一直保持共享锁直到事务提交，而不是像前面的几个隔离级别查询完就是否共享锁，就能避免其他事务获取排他锁。

**4.SERIALIZABLE**

SERIALIZABLE(可序列化)，对于前面的REPEATABLE READ能保证事务可重复读，但是事务只锁定查询第一次运行时获取的数据资源（数据行），而不能锁定查询结果之外的行，就是原本不存在于数据表中的数据。因此在一个事务中当第一个查询和第二个查询过程之间，有其他事务执行插入操作且插入数据满足第一次查询读取过滤的条件时，那么在第二次查询的结果中就会存在这些新插入的数据，使两次查询结果不一致，这种读操作称之为幻读。  
为了避免幻读需要将隔离级别设置为SERIALIZABLE

[复制代码](javascript:void(0);)

IF OBJECT\_ID('Orders','U') IS NOT NULL DROP TABLE Orders

GO

CREATE TABLE Orders

(ID INT NOT NULL PRIMARY KEY,

Price FLOAT NOT NULL,

type INT NOT NULL

);

INSERT INTO Orders VALUES(**10**,**10.00**,**1**),(**11**,**11.00**,**1**),(**12**,**12.00**,**1**),(**13**,**13.00**,**1**),(**14**,**14.00**,**1**);

GO

[复制代码](javascript:void(0);)

在回话1中执行查询操作,并将事务隔离级别设置为REPEATABLE READ(先测试一下前面更低级别的隔离)

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

BEGIN TRANSACTION

SELECT ID,Price,type FROM Orders

WHERE TYPE=**1**



在回话2中执行插入操作

INSERT INTO Orders VALUES(**15**,**15.00**,**1**)

返回回话1重新执行查询操作并提交事务

SELECT ID,Price,type FROM Orders

WHERE TYPE=**1**

COMMIT TRANSACTION



结果回话1中第二次查询到的数据包含了回话2新插入的数据，两次查询结果不一致（验证之前的隔离级别不能保证幻读）

重新插入测试数据

[复制代码](javascript:void(0);)

IF OBJECT\_ID('Orders','U') IS NOT NULL DROP TABLE Orders

GO

CREATE TABLE Orders

(ID INT NOT NULL PRIMARY KEY,

Price FLOAT NOT NULL,

type INT NOT NULL

);

INSERT INTO Orders VALUES(**10**,**10.00**,**1**),(**11**,**11.00**,**1**),(**12**,**12.00**,**1**),(**13**,**13.00**,**1**),(**14**,**14.00**,**1**);

GO

[复制代码](javascript:void(0);)

接下来将回话级别设置为SERIALIZABLE,在回话1中执行查询操作,并将事务隔离级别设置为SERIALIZABLE

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

BEGIN TRANSACTION

SELECT ID,Price,type FROM Orders

WHERE TYPE=**1**



在回话2中执行插入操作

INSERT INTO Orders VALUES(**15**,**15.00**,**1**)

返回回话1重新执行查询操作并提交事务

SELECT ID,Price,type FROM Orders

WHERE TYPE=**1**

COMMIT TRANSACTION



两次执行的查询结果相同

重置所有打开回话的默认隔离级别

SET TRANSACTION ISOLATION LEVEL READ COMMITTED

**5.SNAPSHOT**

SNAPSHOT快照：SNAPSHOT和READ COMMITTED SNAPSHOT两种隔离（可以把事务已经提交的行的上一版本保存在TEMPDB数据库中）  
SNAPSHOT隔离级别在逻辑上与SERIALIZABLE类似  
READ COMMITTED SNAPSHOT隔离级别在逻辑上与 READ COMMITTED类似  
不过在快照隔离级别下读操作不需要申请获得共享锁，所以即便是数据已经存在排他锁也不影响读操作。而且仍然可以得到和SERIALIZABLE与READ COMMITTED隔离级别类似的一致性；如果目前版本与预期的版本不一致，读操作可以从TEMPDB中获取预期的版本。

如果启用任何一种基于快照的隔离级别，DELETE和UPDATE语句在做出修改前都会把行的当前版本复制到TEMPDB中，而INSERT语句不需要在TEMPDB中进行版本控制，因为此时还没有行的旧数据

无论启用哪种基于快照的隔离级别都会对更新和删除操作产生性能的负面影响，但是有利于提高读操作的性能因为读操作不需要获取共享锁；

**5.1SNAPSHOT**

SNAPSHOT 在SNAPSHOT隔离级别下,当读取数据时可以保证操作读取的行是事务开始时可用的最后提交版本  
同时SNAPSHOT隔离级别也满足前面的已提交读,可重复读,不幻读;该隔离级别实用的不是共享锁,而是行版本控制  
使用SNAPSHOT隔离级别首先需要在数据库级别上设置相关选项

在打开的所有查询窗口中执行以下操作

ALTER DATABASE TEST SET ALLOW\_SNAPSHOT\_ISOLATION ON;

重置测试数据

[复制代码](javascript:void(0);)

IF OBJECT\_ID('Orders','U') IS NOT NULL DROP TABLE Orders

GO

CREATE TABLE Orders

(ID INT NOT NULL PRIMARY KEY,

Price FLOAT NOT NULL,

type INT NOT NULL

);

INSERT INTO Orders VALUES(**10**,**10.00**,**1**),(**11**,**11.00**,**1**),(**12**,**12.00**,**1**),(**13**,**13.00**,**1**),(**14**,**14.00**,**1**);

GO

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

在回话1中打开事务,将订单10的价格加1,并查询跟新后的价格

BEGIN TRANSACTION

UPDATE Orders

SET Price=Price+**1**

WHERE ID=**10**

SELECT ID,Price,type FROM Orders

WHERE ID=**10**

---查询到更新后的价格为11

---在回话2中将隔离级别设置为SNAPSHOT,并打开事务(此时查询也不会因为回话1的排他锁而等待,依然可以查询到数据)

SET TRANSACTION ISOLATION LEVEL SNAPSHOT

BEGIN TRANSACTION

SELECT ID,Price,type FROM Orders

WHERE ID=**10**

---查询到的结果还是回话1修改前的价格,由于回话1在默认的READ COMMITTED隔离级别下运行,SQL SERVER必须在更新前把行的一个副本复制到TEMPDB数据库中

--在SNAPSHOT级别启动事务会请求行版本

---现在在回话1中执行提交事务,此时订单10的价格为11

COMMIT TRANSACTION

---再次在回话二中查询订单10的价格并提交事务,结果还是10,因为事务要保证两次查询的结果相同

SELECT ID,Price,type FROM Orders

WHERE ID=**10**

COMMIT TRANSACTION

---此时如果在回话2中重新打开一个事务,查询到的订单10的价格则是11

BEGIN TRANSACTION

SELECT ID,Price,type FROM Orders

WHERE ID=**10**

COMMIT TRANSACTION

/\*SNAPSHOT隔离级别保证操作读取的行是事务开始时可用的最后已提交版本,由于回话1的事务未提交,所以订单10的最后提交版本还是修改前的价格10,所以回话2读取到的价格是回话2事务开始前的已提交版本价格10,当回话1提交事务后,回话2重新新建一个事务此时事务开启前的价格已经是11了，所以查询到的价格是11，同时SNAPSHOT隔离级别还能保证SERIALIZABLE的隔离级别\*/

[复制代码](javascript:void(0);)

**5.2READ COMMITTED SNAPSHOT**

READ COMMITTED SNAPSHOT也是基于行版本控制,但是READ COMMITTED SNAPSHOT的隔离级别是读操作之前的最后已提交版本,而不是事务前的已提交版本，有点类似前面的READ COMMITTED能保证已提交读,但是不能保证可重复读,不能避免幻读,但是又比 READ COMMITTED隔离级别多出了不需要获取共享锁就可以读取数据  
  
要启用READ COMMITTED SNAPSHOT隔离级别同样需要修改数据库选项，在回话1，回话2中执行以下操作(执行下面的操作当前连接必须是数据库的唯一连接，可以通过查询已连接当前数据库的进程,然后KILL掉那些进程,然后再执行该操作,否则可能无法执行成功)

[复制代码](javascript:void(0);)

ALTER DATABASE TEST SET READ\_COMMITTED\_SNAPSHOT ON

IF OBJECT\_ID('Orders','U') IS NOT NULL DROP TABLE Orders

GO

CREATE TABLE Orders

(ID INT NOT NULL PRIMARY KEY,

Price FLOAT NOT NULL,

type INT NOT NULL

);

INSERT INTO Orders VALUES(**10**,**10.00**,**1**),(**11**,**11.00**,**1**),(**12**,**12.00**,**1**),(**13**,**13.00**,**1**),(**14**,**14.00**,**1**);

GO

-----在回话1中打开事务,将订单10的价格加1,并查询跟新后的价格,并保持事务一直处于打开状态

BEGIN TRANSACTION

UPDATE Orders

SET Price=Price+**1**

WHERE ID=**10**

--查询到的价格是11

SELECT ID,Price,type FROM Orders

WHERE ID=**10**

---在回话2中打开事务查询订单10并一直保持事务处于打开状态(此时由于回话1还未提交事务,所以回话2中查询到的还是回话1执行事务之前保存的行版本)

BEGIN TRANSACTION

SELECT ID,Price,type FROM Orders

WHERE ID=**10**

--查询到的价格还是10

---在回话1中提交事务

COMMIT TRANSACTION

---在回话2中再次执行查询订单10的价格,并提交事务

SELECT ID,Price,type FROM Orders

WHERE ID=**10**

COMMIT TRANSACTION

--此时的价格为回话1修改后的价格11,而不是事务之前已提交版本的价格,也就是READ COMMITTED SNAPSHOT隔离级别在同一事务中两次查询的结果不一致.

[复制代码](javascript:void(0);)

关闭所有连接,然后打开一个新的连接,禁用之前设置的数据库快照隔离级别选项

ALTER DATABASE TEST SET ALLOW\_SNAPSHOT\_ISOLATION OFF;

ALTER DATABASE TEST SET READ\_COMMITTED\_SNAPSHOT OFF;

**总结**

理解了事务隔离级别有助于理解事务的死锁。

一、数据库事务隔离级别

数据库事务的隔离级别有4个，由低到高依次为Read uncommitted 、Read committed 、Repeatable read 、Serializable ，这四个级别可以逐个解决脏读 、不可重复读 、幻读 这几类问题。

√: 可能出现    ×: 不会出现

脏读 不可重复读 幻读

Read uncommitted √ √ √

Read committed × √ √

Repeatable read × × √

Serializable × × ×

注意：我们讨论隔离级别的场景，主要是在多个事务并发 的情况下，因此，接下来的讲解都围绕事务并发。

Read uncommitted 读未提交

公司发工资了，领导把5000元打到singo的账号上，但是该事务并未提交，而singo正好去查看账户，发现工资已经到账，是5000元整，非常高 兴。可是不幸的是，领导发现发给singo的工资金额不对，是2000元，于是迅速回滚了事务，修改金额后，将事务提交，最后singo实际的工资只有 2000元，singo空欢喜一场。

出现上述情况，即我们所说的脏读 ，两个并发的事务，“事务A：领导给singo发工资”、“事务B：singo查询工资账户”，事务B读取了事务A尚未提交的数据。

当隔离级别设置为Read uncommitted 时，就可能出现脏读，如何避免脏读，请看下一个隔离级别。

Read committed 读提交

singo拿着工资卡去消费，系统读取到卡里确实有2000元，而此时她的老婆也正好在网上转账，把singo工资卡的2000元转到另一账户，并在 singo之前提交了事务，当singo扣款时，系统检查到singo的工资卡已经没有钱，扣款失败，singo十分纳闷，明明卡里有钱，为 何......

出现上述情况，即我们所说的不可重复读 ，两个并发的事务，“事务A：singo消费”、“事务B：singo的老婆网上转账”，事务A事先读取了数据，事务B紧接了更新了数据，并提交了事务，而事务A再次读取该数据时，数据已经发生了改变。

当隔离级别设置为Read committed 时，避免了脏读，但是可能会造成不可重复读。

大多数数据库的默认级别就是Read committed，比如Sql Server , Oracle。如何解决不可重复读这一问题，请看下一个隔离级别。

Repeatable read 重复读

当隔离级别设置为Repeatable read 时，可以避免不可重复读。当singo拿着工资卡去消费时，一旦系统开始读取工资卡信息（即事务开始），singo的老婆就不可能对该记录进行修改，也就是singo的老婆不能在此时转账。

虽然Repeatable read避免了不可重复读，但还有可能出现幻读 。

singo的老婆工作在银行部门，她时常通过银行内部系统查看singo的信用卡消费记录。有一天，她正在查询到singo当月信用卡的总消费金额 （select sum(amount) from transaction where month = 本月）为80元，而singo此时正好在外面胡吃海塞后在收银台买单，消费1000元，即新增了一条1000元的消费记录（insert transaction ... ），并提交了事务，随后singo的老婆将singo当月信用卡消费的明细打印到A4纸上，却发现消费总额为1080元，singo的老婆很诧异，以为出 现了幻觉，幻读就这样产生了。

注：Mysql的默认隔离级别就是Repeatable read。

Serializable 序列化

Serializable 是最高的事务隔离级别，同时代价也花费最高，性能很低，一般很少使用，在该级别下，事务顺序执行，不仅可以避免脏读、不可重复读，还避免了幻像读。

二、脏读、幻读、不可重复读

1.脏读：

脏读就是指当一个事务正在访问数据，并且对数据进行了修改，而这种修改还没有提交到数据库中，这时，另外一个事务也访问这个数据，然后使用了这个数据。

2.不可重复读：

是指在一个事务内，多次读同一数据。在这个事务还没有结束时，另外一个事务也访问该同一数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改，那么第一个事务两次读到的的数据可能是不一样的。这样就发生了在一个事务内两次读到的数据是不一样的，因此称为是不可重复读。（即不能读到相同的数据内容）

例如，一个编辑人员两次读取同一文档，但在两次读取之间，作者重写了该文档。当编辑人员第二次读取文档时，文档已更改。原始读取不可重复。如果只有在作者全部完成编写后编辑人员才可以读取文档，则可以避免该问题。

3.幻读:

是指当事务不是独立执行时发生的一种现象，例如第一个事务对一个表中的数据进行了修改，这种修改涉及到表中的全部数据行。同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象

发生了幻觉一样。

例如，一个编辑人员更改作者提交的文档，但当生产部门将其更改内容合并到该文档的主复本时，发现作者已将未编辑的新材料添加到该文档中。如果在编辑人员和生产部门完成对原始文档的处理之前，任何人都不能将新材料添加到文档中，则可以避免该问题。

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

**不可重复读的重点是修改** :  
 同样的条件, 你读取过的数据,再次读取出来发现值不一样了  
**幻读的重点在于新增或者删除** 同样的条件, 第 1 次和第 2 次读出来的记录数不一样

SQL SERVER锁的机制   
   SQL server的所有活动都会产生锁。锁定的单元越小，就越能越能提高并发处理能力，但是管理锁的开销越大。如何找到平衡点，使并发性和性能都可接受是SQL Server的难点。  
SQL Server有如下几种琐：  
   SQL server的所有活动都会产生锁。锁定的单元越小，就越能越能提高并发处理能力，但是管理锁的开销越大。如何找到平衡点，使并发性和性能都可接受是SQL Server的难点。

**SQL Server有如下几种琐：**

**1、 共享锁**  
用于只读操作(SELECT)，锁定共享的资源。**共享锁不会阻止其他用户读，但是阻止其他的用户写和修改。**

**2、 更新锁**  
更新锁是一种意图锁，**当一个事务已经请求共享琐后并试图请求一个独占锁的时候发生更新琐。**例如当两个事务在几行数据行上都使用了共享锁，并同时试图获取独占锁以执行更新操作时，就发生了死锁：都在等待对方释放共享锁而实现独占锁。更新锁的目的是只让一个事务获得更新锁，防止这种情况的发生。

**3、 独占锁**  
一次只能有一个独占锁用在一个资源上，并且**阻止其他所有的锁**包括共享缩。写是独占锁，可以有效的防止’脏读’。

**4、 意图缩**  
在使用共享锁和独占锁之前，使用意图锁。从表的层次上查看意图锁，以判断事务能否获得共享锁和独占锁，提高了系统的性能，不需从页或者行上检查。

**5、 计划锁**  
Sch-M,Sch-S。对数据库结构改变时用Sch-M，对查询进行编译时用Sch-S。这两种锁不会阻塞任何事务锁，包括独占锁。  
**读是共享锁，写是排他锁，先读后更新的操作是更新锁，更新锁成功并且改变了数据时更新锁升级到排他锁**

l     DEFAULT 使用数据库设置的隔离级别 ( 默认 ) ，由 DBA 默认的设置来决定隔离级别 .   
l     READ\_UNCOMMITTED 会出现脏读、不可重复读、幻读 (**隔离级别最低，并发性能高** )   
l     READ\_COMMITTED  会出现不可重复读、幻读问题（**锁定正在读取的行**）   
l     REPEATABLE\_READ 会出幻读（**锁定所读取的所有行**）   
l     SERIALIZABLE 保证所有的情况不会发生（**锁表**）



ReadCommitted：   
假设A事务对正在读取数据Data放置了共享锁，那么Data不能被其它事务改写，所以当B事务对Data进行读取时总和A读取的Data数据是一致的，所以避免了脏读。由于在A没有提交之前可以对Data进行改写，那么B读取到的某个值可能会在其读取后被A更改从而导致了该值不能被重复取得；或者当B再次用相同的where字句时得到了和前一次不一样数据的结果集，也就是幻像数据。

ReadUncommitted：  
假设A事务即不发布共享锁，也不接受独占锁，那么并发的B或者其它事务可以改写A事务读取的数据，那么并发的C事务读取到的数据的状态和A的或者B的数据都可能不一致，那么。脏读、不可重复读、幻象数据都可能存在。

RepeatableRead：  
（注意MSDN原文中的第一句话：在查询中使用的所有数据上放置锁，所以不存在脏读的情况）。  
假设A事务对读取的所有数据Data放置了锁，以阻止其它事务对Data的更改，在A没有提交之前，新的并发事务读取到的数据如果存在于Data中，那么该数据的状态和A事务中的数据是一致的，从而避免了不可重复的读取。但在A事务没有结束之前，B事务可以插入新记录到Data所在的表中，那么其它事务再次用相同的where字句查询时，得到的结果数可能上一次的不一致，也就是幻像数据。

Serializable：  
 在数据表上放置了排他锁，以防止在事务完成之前由其他用户更新行或向数据集中插入行，这是最严格的锁。它防止了脏读、不可重复读取和幻象数据。

### SQL Server : Merge replication fails due to timeout errors

You administer several Microsoft SQL Server 2012 database servers. Merge replication

has been configured for an application that is distributed across offices throughout a wide

area network (WAN). Many of the tables involved in replication use the XML and varchar

(max) data types. Occasionally, merge replication fails due to timeout errors. You need to

reduce the occurrence of these timeout errors. What should you do?

A. Set the Merge agent on the problem subscribers to use the slow link agent profile.

B. Create a snapshot publication, and reconfigure the problem subscribers to use the

Question No : 40

Question No : 41

snapshot publication.

C. Change the Merge agent on the problem subscribers to run continuously.

D. Set the Remote Connection Timeout on the Publisher to 0.

**Answer:**

When you synchronize data rows with a large amount of data, such as rows with LOB columns, Web synchronization can require additional memory allocation and hurt performance. This occurs when the Merge Agent generates an XML message that contains too many data rows with large amounts of data. If the Merge Agent is consuming too many resources during Web synchronization, reduce the number of rows sent in a single message in one of the following ways:

* **Use the slow link agent profile for the Merge Agent.**
* Decrease the **-DownloadGenerationsPerBatch** and **-UploadGenerationsPerBatch** parameters for the Merge Agent to a value of 10 or less. The default value of these parameters is 50.

Note: Merge Agent has a "slow link" profile designed for low bandwidth connections.

**How to allow merge agent to use "slow link profile" :**

**Change Existing Agents:**

Select a profile (On the General page of the Distributor Properties - <Distributor> dialog box, click Profile Defaults), and then click Change Existing Agents to specify that all existing jobs for an agent of a given type should use the selected profile. For example, if you have created a number of subscriptions to a merge publication, and you want to change the profile to specify that the Merge Agent job for each of these subscriptions should use the Slow link agent profile, select that profile, and then click Change Existing Agents.

**How Merge Replication (合并复制)Works**

Merge replication is implemented by the SQL Server Snapshot Agent and Merge Agent. If the publication is unfiltered or uses static filters, the Snapshot Agent creates a single snapshot. If the publication uses parameterized filters, the Snapshot Agent creates a snapshot for each partition of data. The Merge Agent applies the initial snapshots to the Subscribers. It also merges incremental data changes that occurred at the Publisher or Subscribers after the initial snapshot was created, and detects and resolves any conflicts according to rules you configure.

**Merge Replication(合并复制)**

Masha at MSFT olprod OpenLocalizationService Saisang Cai

适用对象：yesSQL ServeryesAzure SQL 数据库noAzure SQL 数据仓库no并行数据仓库

与事务复制相同，合并复制通常也是从发布数据库对象和数据的快照开始， 并且用触发器跟踪在发布服务器和订阅服务器上所做的后续数据更改和架构修改。 订阅服务器在连接到网络时将与发布服务器进行同步，并交换自上次同步以来发布服务器和订阅服务器之间发生更改的所有行。

合并复制通常用于服务器到客户端的环境中。 合并复制适用于下列各种情况：

多个订阅服务器可能会在不同时间更新同一数据，并将其更改传播到发布服务器和其他订阅服务器。

订阅服务器需要接收数据，脱机更改数据，并在以后与发布服务器和其他订阅服务器同步更改。

每个订阅服务器都需要不同的数据分区。

可能会发生冲突，并且在冲突发生时，您需要具有检测和解决冲突的能力。

应用程序需要最终的数据更改结果，而不是访问中间数据状态。 例如，如果在订阅服务器与发布服务器进行同步之前，订阅服务器上的行更改了五次，则该行在发布服务器上仅更改一次来反映最终数据更改（也就是第五次更改的值）。

合并复制允许不同站点自主工作，并在以后将更新合并成一个统一的结果。 由于更新是在多个节点上进行的，同一数据可能由发布服务器和多个订阅服务器进行了更新。 因此，在合并更新时可能会产生冲突，合并复制提供了多种处理冲突的方法。

合并复制是由 SQL Server 快照代理和合并代理实现的。 如果发布未经筛选或使用静态筛选器，快照代理将创建单个快照。 如果发布使用参数化筛选器，则快照代理将为每个数据分区创建一个快照。 合并代理将初始快照应用于订阅服务器。 它还将合并自初始快照创建后发布服务器或订阅服务器上所发生的增量数据更改，并根据所配置的规则检测和解决任何冲突。

若要跟踪更改，合并复制（和带有排队更新订阅的事务复制）必须能够唯一地标识每个已发布表中的每一行。 为了完成这一合并复制，需要向每个表添加列 rowguid ，除非表中已包含一个数据类型为 uniqueidentifier 且设置了 ROWGUIDCOL 属性的列（这种情况下将使用此列）。 如果从发布中删除表，则删除 rowguid 列；如果将现有列用于跟踪，则不删除该列。 筛选器不能包含复制所用的 rowguidcol 来标识行。 提供 newid() 函数，为 rowguid 列提供默认值，但用户可以按需为每行提供 GUID。 不过，请不要提供值 00000000-0000-0000-0000-000000000000。

