

SQL Indexes Impact on Concurrent Data Access

www.ladiMolnar.com

Introduction	2
Why is this interesting?	2
Prerequisites	2
How to reproto the sample cases	3
1. Tables with no indexes	4
2. Tables with a primary key (clustered index)	6
2.1. The WHERE clause uses the primary key column (clustered index)	7
2.2. The WHERE clause uses a column that is not indexed in a table with a primary key (clustered index)	9
2.3. The WHERE clause uses the primary key column (clustered index) and the query changes the value of the primary key column	11
3. Tables with a non clustered index	13
3.1. The WHERE clause uses an indexed column in a table that has a non-clustered index	13
3.2. The WHERE clause uses a column that is not indexed in a table that has only one non-clustered index	15
4. Tables with a composite primary key	17
4.1. The WHERE clause uses all the columns that form the primary key	17
4.2. The WHERE clause uses the first column from the primary key	19
4.3. The WHERE clause uses the first column from the primary key and a non indexed column with an AND	21
4.4. The WHERE clause uses the first column from the primary key and a non indexed column with an OR	24
4.5. The WHERE clause uses the second column from the primary key	26
4.6. The WHERE clause uses the second column of the primary key. That column also has its own index	28
5. Tables with a primary key (clustered index) and a foreign key where the foreign key is not indexed	30
6. Tables with a primary key (clustered index) and a foreign key where the foreign key is indexed	35
The sample database	39

Introduction

This document shows how the presence or absence of indexes will impact the outcome of scenarios where different clients do concurrent access on SQL tables.

Why is this interesting?

Most SQL developers understand how indexes impact the performance of simple queries. The simplest and most direct consequences of indexes have to do with what happens when one client issues a query that has a `WHERE` clause. The presence or absence of the appropriate index can cause dramatic differences in performance if large table are involved. In one case SQL Server will scan the entire table. In other it will be able to directly access the rows targeted by the query.

There is also a different aspect of indexes which has to do with how concurrent clients will impact each other when accessing the same tables. The presence or absence of indexes will change the way SQL Server parses data from tables. Without indexes not only the operations that a client does will cause more data to be parsed but that client may be locked and have to wait needlessly until another client finishes its own operations. By needlessly I mean that the same results would be achieved if no locking and waiting was imposed. Without indexes, the SQL Server does not have a way of figuring that that is the case and the approach that is taken will cause one client to wait for another.

Prerequisites

This article assumes that you have at least a basic understanding of a few simple SQL concepts:

- SQL tables, columns.
- Simple queries especially update queries. Meaning you understand a query like `UPDATE Table1 SET Col1 = Col1 + 1 WHERE Id = 100`
- Transactions.
- Indexes.
- Locks.
- Foreign keys.

A brief refresher on some of these concepts:

- **Indexes**
Indexes are a way of collecting and maintaining information about the location of rows in SQL tables. They help locate rows more quickly and efficiently. SQL Server stores each index in an n-ary tree data structure. The leaf level of this tree is composed of either the actual data (as in the case of a clustered index) or of some kind of pointers to the actual data rows.

SQL Server has two types of indexes:

- **Clustered.**
If you have a clustered index, the data itself in the table will be sorted and stored based on the key values of the clustered index. In that case the leaf level in the index n-ary tree is the data itself.

- **Non-clustered.**
Non-clustered indexes have a structure that is independent of the data rows. The leaf level of a non-clustered index n-ary tree contains the non-clustered index key values. Each entry on that level has pointers to the corresponding actual data rows. The pointer from an index row in a non-clustered index to a data row is called a row locator. What this row locator actually is depends on whether the table has or does not have a clustered index. If there is a clustered index then the row locator is the clustered index key. If the table has no a clustered index then the row locator is a pointer to the row (Row ID or RID).

- **SQL Locks**

Locking happens when a SQL Server process takes ownership of a resource (rows, index keys, index ranges, pages, tables or the database) prior to performing a particular action like reading or updating. The SQL programmer does not have to manage this process although it can influence it. Locks are managed by SQL Server internally. Locks are held on SQL Server resources such as rows, index keys, index ranges, pages, tables or the database and so on. The locks are used to arbitrate concurrent use of resources by multiple transactions. For example, if during a transaction you delete a row from a table, an exclusive (X) lock is held on that row, and no other transaction can read or modify that row until the lock is released. In this case, the lock will be released at the end of a transaction.

How to repro the sample cases

To set up the sample database copy the script at the end of this document in Microsoft SQL Server Management Studio (SQL Server 2005) or in Query Analyzer (SQL Server 2000) and execute it.

The repro scenarios will illustrate things using two different clients. To repro a case, open in SQL Server Management Studio (or Query Analyzer for SQL Server 2000) two separate windows and run the repro statements mentioned for each client.

All the cases are based on similar tables. All these tables have the same columns and content. They all have 2000 rows. The columns are:

```
GroupId      int NOT NULL,    -- 0 to 199. One group for each 10 rows
ItemId       int NOT NULL,    -- 0 to 1999
ItemData     int NULL         -- 0 to 1999
```

The only difference is in the way primary key, indexes or foreign keys are created. Depending on those things we will see that for the same queries locks are applied differently. Since locks are applied differently, concurrent access made by different clients will perform differently.

1. Tables with no indexes

This case is exemplified by table Table1. It is a plain table with no indexes:

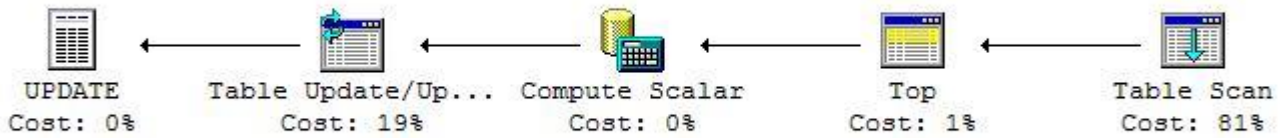
```
CREATE TABLE dbo.Table1 (  
  GroupId      int NOT NULL,  
  ItemId       int NOT NULL,  
  ItemData     int NULL  
)  
GO
```

Let's say we have a client that executes the query:

```
BEGIN TRANSACTION  
UPDATE Table1 SET ItemData = ItemData + 1 WHERE ItemId = 0
```

Even though only one row satisfies the **WHERE** clause, since we don't have an index on the column **ItemId**, SQL Server needs to parse all the rows in the table in order to evaluate the condition **ItemId = 0**. Hence the table scan seen in the execution plan. Each row that is examined is locked with an update (U) lock. If the row does not satisfy the condition it is unlocked. If the row satisfies the condition, the update (U) lock is changed in an exclusive (X) lock and the row is updated. The exclusive lock will remain in effect until the transaction ends. After this statement finishes (but the transaction is still active) the locks are as shown below:

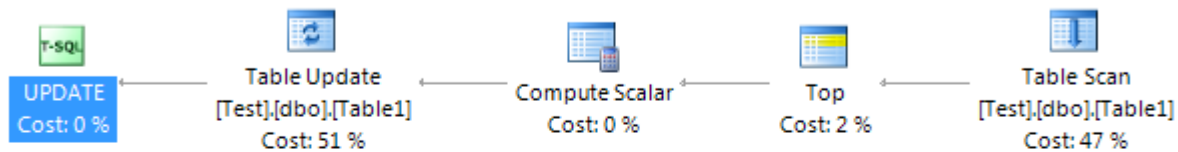
SQL Server 2000 Execution Plan



SQL Server 2000 Locks

Object	Lock Type	Mode	Status	Index	Resource
Test	DB	S	GRANT	Table1	
Test.dbo.Table1	TAB	IX	GRANT	Table1	
Test.dbo.Table1	PAG	IX	GRANT	Table1	1:91
Test.dbo.Table1	RID	X	GRANT	Table1	1:91:0

SQL Server 2005 Execution Plan



SQL Server 2005 Locks

Selected process:		51					
Object	Type	Object ID	Description	Request Mode	Request Type	Request Status	
(internal)	PAGE	72057594038321152	1:159	IX	LOCK	GRANT	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	OBJECT	2073058421		IX	LOCK	GRANT	
Test..Table1	RID	72057594038321152	1:159:0	X	LOCK	GRANT	

Note that since we don't have a clustered index the locks are applied to RIDs (row IDs).

Let's say that before the first client completes its transaction, a second client executes an update on a different row:

```
BEGIN TRANSACTION
```

```
UPDATE Table1 SET ItemData = ItemData + 1 WHERE ItemId = 100
```

The same process will happen where SQL Server has to examine all the rows in the table to see if they satisfy the condition in the WHERE clause which is ItemId = 100. Each row that is examined is locked with an update (U) lock. If the row does not satisfy the condition it is unlocked. If the row satisfies the condition, the update (U) lock is changed in an exclusive (X) lock and the row is updated. When SQL Server tries to examine the row with ItemId = 0, it finds it already exclusively locked by the first client. It has to wait and it does that until the first client will complete its transaction (or a timeout is raised). The locks at this moment are as shown here:

SQL Server 2000 Locks

Process ID	Lock Type	Mode	Status	Index	Resource
51	TAB	IX	GRANT	Table1	
51	PAG	IX	GRANT	Table1	1:91
51	RID	X	GRANT	Table1	1:91:0
53	TAB	IX	GRANT	Table1	
53	PAG	IU	GRANT	Table1	1:91
53	RID	U	WAIT	Table1	1:91:0

SQL Server 2005 Locks

Selected process:		51					
Object	Type	Object ID	Description ^	Request Mode	Request Type	Request Status	
(internal)	OBJECT	2073058421		IX	LOCK	GRANT	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	PAGE	72057594038321152	1:159	IX	LOCK	GRANT	
Test..Table1	RID	72057594038321152	1:159:0	X	LOCK	GRANT	

Selected process:		52					
Object	Type	Object ID	Description ^	Request Mode	Request Type	Request Status	
(internal)	OBJECT	2073058421		IX	LOCK	GRANT	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	PAGE	72057594038321152	1:159	IU	LOCK	GRANT	
Test..Table1	RID	72057594038321152	1:159:0	U	LOCK	WAIT	

Note the Status (or Request Status) = WAIT for one of the locks applied by the second client. That is the lock that will force the second client to wait until the first client completes its transaction.

2. Tables with a primary key (clustered index)

This case is exemplified by table Table2. It is similar with Table1 with the difference that it has a clustered index:

```
CREATE TABLE dbo.Table2 (  
  GroupId      int NOT NULL,  
  ItemId       int NOT NULL,  
  ItemData     int NULL  
)  
GO  
  
ALTER TABLE dbo.Table2 ADD CONSTRAINT  
PK_Table2 PRIMARY KEY CLUSTERED (ItemId)  
GO
```

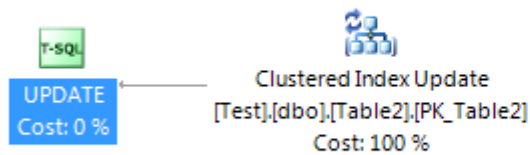
2.1. The WHERE clause uses the primary key column (clustered index)

The first client executes:

```
BEGIN TRANSACTION  
UPDATE Table2 SET ItemData = ItemData + 1 WHERE ItemId = 0
```

In this case SQL Server knows how to pinpoint the appropriate row based on the index. We no longer have a table scan. Since the access is done in a direct manner, the two clients will not lock each other out as long as they access different rows. Note also that the locks are applied on the key of the clustered index.

SQL Server 2005 Execution Plan



SQL Server 2005 Locks

Selected process: 51

	Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status
	(internal)	OBJECT	2089058478		IX	LOCK	GRANT
	(internal)	DATABASE	0		S	LOCK	GRANT
	(internal)	PAGE	72057594038452224	1:161	IX	LOCK	GRANT
	Test..Table2	KEY	72057594038452224	(0000e320bbde)	X	LOCK	GRANT

The second client executes:

```
BEGIN TRANSACTION
UPDATE Table2 SET ItemData = ItemData + 1 WHERE ItemId = 100
```

SQL Server can pinpoint the row that satisfies the condition `ItemId = 100` based on the index. It will no longer have to do the table scan that we saw at point 1. As shown here the second client is not blocked.

SQL Server 2005 Locks

Selected process: 51

	Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status
	(internal)	OBJECT	2089058478		IX	LOCK	GRANT
	(internal)	DATABASE	0		S	LOCK	GRANT
	(internal)	PAGE	72057594038452224	1:161	IX	LOCK	GRANT
	Test..Table2	KEY	72057594038452224	(0000e320bbde)	X	LOCK	GRANT

Selected process: 52

	Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status
	(internal)	DATABASE	0		S	LOCK	GRANT
	(internal)	PAGE	72057594038452224	1:161	IX	LOCK	GRANT
	(internal)	OBJECT	2089058478		IX	LOCK	GRANT
	Test..Table2	KEY	72057594038452224	(6400b740f6a)	X	LOCK	GRANT

Note the fact that both clients have all their locks granted.

The second client will however be blocked if it tries to access the same row as the first client:

```
BEGIN TRANSACTION
UPDATE Table2 SET ItemData = ItemData + 1 WHERE ItemId = 0
```

SQL Server 2005 Locks

Selected process: 51							
	Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status
	(internal)	OBJECT	2089058478		IX	LOCK	GRANT
	(internal)	DATABASE	0		S	LOCK	GRANT
	(internal)	PAGE	72057594038452224	1:161	IX	LOCK	GRANT
	Test..Table2	KEY	72057594038452224	(0000e320bbde)	X	LOCK	GRANT
Selected process: 52							
	Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status
	(internal)	DATABASE	0		S	LOCK	GRANT
	(internal)	PAGE	72057594038452224	1:161	IX	LOCK	GRANT
	(internal)	OBJECT	2089058478		IX	LOCK	GRANT
	Test..Table2	KEY	72057594038452224	(0000e320bbde)	X	LOCK	WAIT

Note the Request Status = WAIT for one of the locks of the second client.

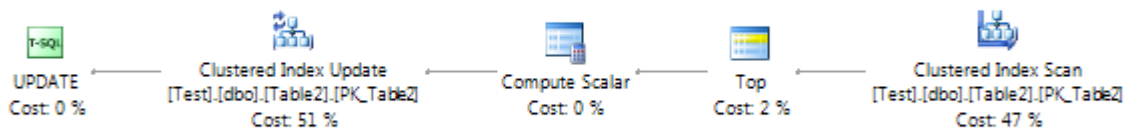
2.2. The WHERE clause uses a column that is not indexed in a table with a primary key (clustered index)

The first client executes:

```
BEGIN TRANSACTION
UPDATE Table2 SET ItemData = ItemData + 1 WHERE ItemData = 0
```

Even though we have an index, that index is useless when evaluating the WHERE clause. As a result SQL Server will have to parse and evaluate all the rows in the table. This scenario is similar with the one at point 1. There is a difference though. The locks are taken on the keys of the clustered index and not on RIDs.

SQL Server 2005 Execution Plan



SQL Server 2005 Locks

Selected process:		51					
	Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status
	(internal)	DATABASE	0		S	LOCK	GRANT
	(internal)	OBJECT	2089058478		IX	LOCK	GRANT
	(internal)	PAGE	72057594038452224	1:161	IX	LOCK	GRANT
	Test..Table2	KEY	72057594038452224	(0000e320bbde)	X	LOCK	GRANT

If the second client executes:

```
BEGIN TRANSACTION
UPDATE Table2 SET ItemData = ItemData + 1 WHERE ItemData = 100
```

Even though we have a clustered index, because that index is of no use for this particular WHERE clause, SQL Server still has to examine all the rows to evaluate the WHERE clause. The same mechanism as shown in case 1. applies. When the second client wants to examine the row with ItemId = 0 to evaluate the condition ItemData = 100, it finds that the row is locked. Since the second client needs to apply a update (U) lock to examine the row, it will have to wait for the exclusive (X) lock already placed by the first client to be lifted. That will only happen when the first client will complete its transaction.

SQL Server 2005 Locks

Selected process:		51					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	OBJECT	2089058478		IX	LOCK	GRANT	
(internal)	PAGE	72057594038452224	1:161	IX	LOCK	GRANT	
Test..Table2	KEY	72057594038452224	(0000e320bbde)	X	LOCK	GRANT	
Selected process:		52					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	PAGE	72057594038452224	1:161	IU	LOCK	GRANT	
(internal)	OBJECT	2089058478		IX	LOCK	GRANT	
Test..Table2	KEY	72057594038452224	(0000e320bbde)	U	LOCK	WAIT	

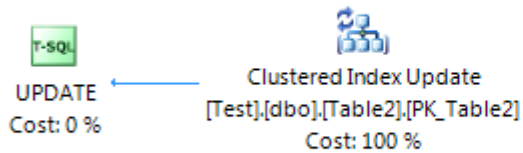
Note the Request Status = WAIT for one of the locks of the second client.

2.3. The WHERE clause uses the primary key column (clustered index) and the query changes the value of the primary key column

The first client executes:

```
BEGIN TRANSACTION
UPDATE Table2 SET ItemId = 2000 WHERE ItemId = 0
```

SQL Server 2005 Execution Plan



SQL Server 2005 Locks

Selected process:		51					
	Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status
	(internal)	PAGE	72057594038452224	1:161	IX	LOCK	GRANT
	(internal)	PAGE	72057594038452224	1:224	IX	LOCK	GRANT
	(internal)	DATABASE	0		S	LOCK	GRANT
	(internal)	OBJECT	2089058478		IX	LOCK	GRANT
	Test..Table2	KEY	72057594038452224	(d000ff8fa0fd)	X	LOCK	GRANT
	Test..Table2	KEY	72057594038452224	(0000e320bbde)	X	LOCK	GRANT







Note that we have two keys locked. One key corresponds to ItemId = 0. The second key corresponds to ItemId = 2000





If the second client executes:

```
BEGIN TRANSACTION
UPDATE Table2 SET ItemData = ItemData + 1 WHERE ItemId = 2000
```

Then the second client is blocked:

SQL Server 2005 Locks

Selected process:		51					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
 (internal)	PAGE	72057594038452224	1:161	IX	LOCK	GRANT	
 (internal)	PAGE	72057594038452224	1:224	IX	LOCK	GRANT	
 (internal)	DATABASE	0		S	LOCK	GRANT	
 (internal)	OBJECT	2089058478		IX	LOCK	GRANT	
 Test..Table2	KEY	72057594038452224	(d000ff8fa0fd)	X	LOCK	GRANT	
 Test..Table2	KEY	72057594038452224	(0000e320bbde)	X	LOCK	GRANT	

Selected process:		52					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
 (internal)	DATABASE	0		S	LOCK	GRANT	
 (internal)	PAGE	72057594038452224	1:224	IX	LOCK	GRANT	
 (internal)	OBJECT	2089058478		IX	LOCK	GRANT	
 Test..Table2	KEY	72057594038452224	(d000ff8fa0fd)	X	LOCK	WAIT	

Note the Request Status = WAIT for one of the locks of the second client.

3. Tables with a non clustered index

This case is exemplified by table Table3. It is a table with a non clustered index:

```
CREATE TABLE dbo.Table3 (  
    GroupId      int NOT NULL,  
    ItemId       int NOT NULL,  
    ItemData     int NULL  
)  
GO  
  
CREATE INDEX IX_Table3_ItemId ON dbo.Table3 (ItemId)  
GO
```

3.1. The WHERE clause uses an indexed column in a table that has a non-clustered index

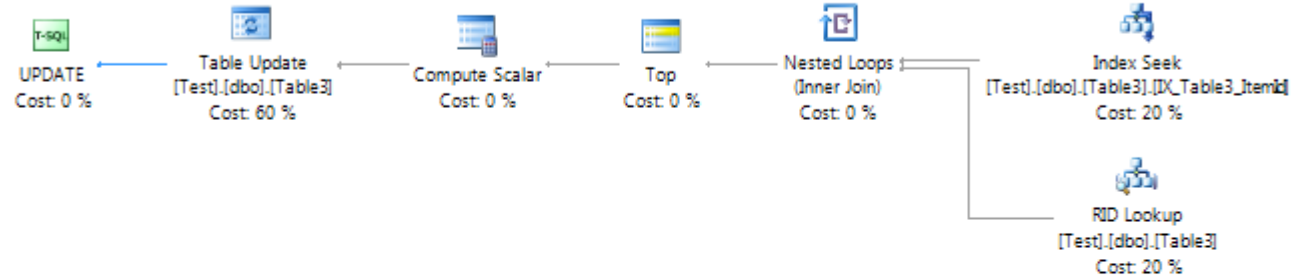
This scenario is similar to the one at point 2.1. The difference is that locks will now be placed on RIDs and not on index keys. The first client executes:

```
BEGIN TRANSACTION
```

```
UPDATE Table3 SET ItemData = ItemData + 1 WHERE ItemId = 0
```

In this case SQL Server knows where the row that satisfies the WHERE clause is based on the index. There is no need for examining all the rows. However, note that since we don't have a clustered index the locks are placed on row IDs (RID) and not on index keys (as it happened in case 2.1.).

SQL Server 2005 Execution Plan



SQL Server 2005 Locks

Selected process: 51

Object	Type	Object ID	Description	Request Mode	Request Type	Request Status
(internal)	DATABASE	0		S	LOCK	GRANT
(internal)	PAGE	72057594038517760	1:163	IX	LOCK	GRANT
(internal)	OBJECT	2121058592		IX	LOCK	GRANT
Test..Table3	RID	72057594038517760	1:163:0	X	LOCK	GRANT

The second client executes:

```
BEGIN TRANSACTION  
UPDATE Table3 SET ItemData = ItemData + 1 WHERE ItemId = 100
```

SQL Server 2005 Locks

Selected process:		51					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	PAGE	72057594038517760	1:163	IX	LOCK	GRANT	
(internal)	OBJECT	2121058592		IX	LOCK	GRANT	
Test..Table3	RID	72057594038517760	1:163:0	X	LOCK	GRANT	

Selected process:		52					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	PAGE	72057594038517760	1:163	IX	LOCK	GRANT	
(internal)	OBJECT	2121058592		IX	LOCK	GRANT	
Test..Table3	RID	72057594038517760	1:163:100	X	LOCK	GRANT	

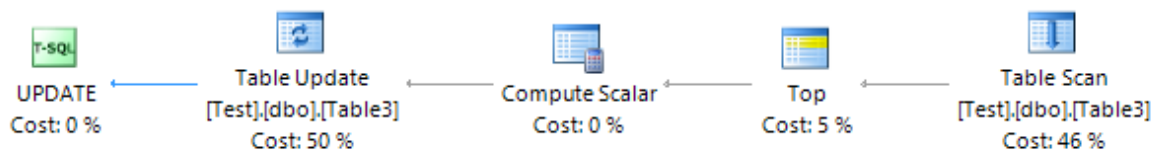
Note that all the locks are granted.

3.2. The WHERE clause uses a column that is not indexed in a table that has only one non-clustered index

This scenario is similar to the one at point 2.2. The difference is that locks will now be placed on RIDs and not on index keys. The first client executes:

```
BEGIN TRANSACTION
UPDATE Table3 SET ItemData = ItemData + 1 WHERE ItemData = 0
```

SQL Server 2005 Execution Plan



SQL Server 2005 Locks

Selected process:		51					
Object	Type	Object ID	Description	Request Mode	Request Type	Request Status	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	OBJECT	2121058592		IX	LOCK	GRANT	
(internal)	PAGE	72057594038517760	1:163	IX	LOCK	GRANT	
Test..Table3	RID	72057594038517760	1:163:0	X	LOCK	GRANT	

The second client executes:

```
BEGIN TRANSACTION
UPDATE Table3 SET ItemData = ItemData + 1 WHERE ItemData = 100
```

The second client is blocked:

SQL Server 2005 Locks

Selected process:		51					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	OBJECT	2121058592		IX	LOCK	GRANT	
(internal)	PAGE	72057594038517760	1:163	IX	LOCK	GRANT	
Test..Table3	RID	72057594038517760	1:163:0	X	LOCK	GRANT	

Selected process:		52					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	PAGE	72057594038517760	1:163	IU	LOCK	GRANT	
(internal)	OBJECT	2121058592		IX	LOCK	GRANT	
Test..Table3	RID	72057594038517760	1:163:0	U	LOCK	WAIT	

Note the Request Status = WAIT for one of the locks of the second client.

4. Tables with a composite primary key

This case is exemplified by table Table4. It is a table with a composite clustered index:

```
CREATE TABLE dbo.Table4 (  
    GroupId      int NOT NULL,  
    ItemId       int NOT NULL,  
    ItemData     int NULL  
)  
GO  
  
ALTER TABLE dbo.Table4 ADD CONSTRAINT  
PK_Table4 PRIMARY KEY CLUSTERED (GroupId, ItemId)  
GO
```

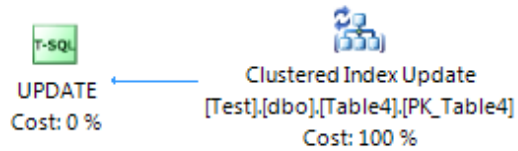
4.1. The WHERE clause uses all the columns that form the primary key

This is quite similar with the case at point 2.1. The first client executes:

```
BEGIN TRANSACTION
```

```
UPDATE Table4 SET ItemData = ItemData + 1
WHERE GroupId = 0 AND ItemId = 0
```

SQL Server 2005 Execution Plan



SQL Server 2005 Locks

Selected process: 51





	Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status
	(internal)	DATABASE	0		S	LOCK	GRANT
	(internal)	PAGE	72057594038714368	1:167	IX	LOCK	GRANT
	(internal)	OBJECT	2137058649		IX	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(00009620dd9a)	X	LOCK	GRANT





If the second client executes:

```
BEGIN TRANSACTION
UPDATE Table4 SET ItemData = ItemData + 1
WHERE GroupId = 0 AND ItemId = 1
```

The second client is not blocked:

SQL Server 2005 Locks

Selected process:		51					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
 (internal)	DATABASE	0		S	LOCK	GRANT	
 (internal)	PAGE	72057594038714368	1:167	IX	LOCK	GRANT	
 (internal)	OBJECT	2137058649		IX	LOCK	GRANT	
 Test..Table4	KEY	72057594038714368	(00009620dd9a)	X	LOCK	GRANT	

Selected process:		52					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
 (internal)	DATABASE	0		S	LOCK	GRANT	
 (internal)	PAGE	72057594038714368	1:167	IX	LOCK	GRANT	
 (internal)	OBJECT	2137058649		IX	LOCK	GRANT	
 Test..Table4	KEY	72057594038714368	(0100f3476122)	X	LOCK	GRANT	

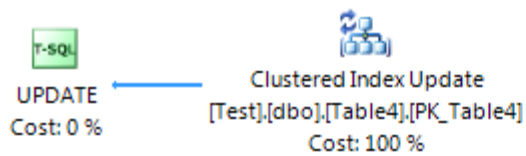
Note that all the locks are granted.

4.2. The WHERE clause uses the first column from the primary key

The first client executes:

```
BEGIN TRANSACTION
UPDATE Table4 SET ItemData = ItemData + 1 WHERE GroupId = 0
```

SQL Server 2005 Execution Plan



SQL Server 2005 Locks

Selected process: 51

	Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status
	(internal)	DATABASE	0		S	LOCK	GRANT
	(internal)	PAGE	72057594038714368	1:167	IX	LOCK	GRANT
	(internal)	OBJECT	2137058649		IX	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(0500a4d003ad)	X	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(0100f3476122)	X	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(0300788f6888)	X	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(02001de8d430)	X	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(00009620dd9a)	X	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(09001c6fd5e7)	X	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(07002f180a07)	X	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(06004a7b6bf)	X	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(08007908695f)	X	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(0400c1b7bf15)	X	LOCK	GRANT

The second client executes:

```
BEGIN TRANSACTION
UPDATE Table4 SET ItemData = ItemData + 1 WHERE GroupId = 1
```

The second client is not blocked. Note however that locks are placed individually on key values for each of the 20 rows that are updated by the two clients:

SQL Server 2005 Locks

Selected process:		51					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	PAGE	72057594038714368	1:167	IX	LOCK	GRANT	
(internal)	OBJECT	2137058649		IX	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(0500a4d003ad)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(0100f3476122)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(0300788f6888)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(02001de8d430)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(00009620dd9a)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(09001c6fd5e7)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(07002f180a07)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(06004a7b6bf)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(08007908695f)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(0400c1b7bf15)	X	LOCK	GRANT	

Selected process:		52					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	PAGE	72057594038714368	1:167	IX	LOCK	GRANT	
(internal)	OBJECT	2137058649		IX	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(0d00b09fa11c)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(140079d8db14)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(0e00d5f81da4)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(0f003b57a8b6)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(0c0009a77681)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(110097776e06)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(0b006cc0ca39)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(10005e30140e)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(1200f210d2be)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(13001cbf67ac)	X	LOCK	GRANT	

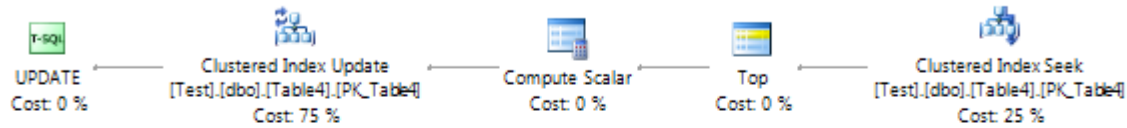
4.3. The WHERE clause uses the first column from the primary key and a non indexed column with an AND

The first client executes:

```
BEGIN TRANSACTION
UPDATE Table4 SET ItemData = ItemData + 1
```

```
WHERE GroupId = 0 AND ItemData = 0
```

SQL Server 2005 Execution Plan



SQL Server 2005 Locks





Selected process:		51					
Object	Type	Object ID	Description	Request Mode	Request Type	Request Status	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	PAGE	72057594038714368	1:167	IX	LOCK	GRANT	
(internal)	OBJECT	2137058649		IX	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(00009620dd9a)	X	LOCK	GRANT	





The second client accesses a different row in a different group. It executes:

```
BEGIN TRANSACTION
UPDATE Table4 SET ItemData = ItemData + 1
WHERE GroupId = 1 AND ItemData = 10
```

For each client there is only one row that satisfies the WHERE clause. However, SQL Server will have to access and examine more rows than that one row. Since the combination (GroupId, IndexData) is not indexed, SQL Server cannot pinpoint the rows that satisfy the WHERE clause based on an index. Nevertheless it does not have to examine all the rows in the table. It can still take advantage of the condition placed on the GroupId. The first client examines all 10 rows that have GroupId = 0. The second client examines all 10 rows that have GroupId = 1. All in all, the two clients will manage to avoid stepping on each other toes and the result is that the second client is not blocked:

SQL Server 2005 Locks

Selected process:		51					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
 (internal)	DATABASE	0		S	LOCK	GRANT	
 (internal)	PAGE	72057594038714368	1:167	IX	LOCK	GRANT	
 (internal)	OBJECT	2137058649		IX	LOCK	GRANT	
 Test..Table4	KEY	72057594038714368	(00009620dd9a)	X	LOCK	GRANT	

Selected process:		52					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
 (internal)	DATABASE	0		S	LOCK	GRANT	
 (internal)	PAGE	72057594038714368	1:167	IX	LOCK	GRANT	
 (internal)	OBJECT	2137058649		IX	LOCK	GRANT	
 Test..Table4	KEY	72057594038714368	(0b006cc0ca39)	X	LOCK	GRANT	

Note that all the locks are granted.

The story changes if the second client accesses a different row in the same group as the first client:

```
BEGIN TRANSACTION
UPDATE Table4 SET ItemData = ItemData + 1
WHERE GroupId = 0 AND ItemData = 5
```

This time the second client is blocked although the row it tries to modify is different from the row modified by the first client. This is because GroupId that participates in the primary key cannot discriminate between the two rows. Both these rows have GroupId = 0. When the second client issues its query, SQL Server will have to examine all the 10 rows in the group that has GroupId = 0. For each row that is examined, SQL Server will start by placing an update (U) lock for the period it reads the row. When it tries to examine (and place the U lock) the row that has ItemId = 0 it finds it already exclusively locked. The second client will be blocked until the first client finishes its transaction:

SQL Server 2005 Locks

Selected process:		51					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	PAGE	72057594038714368	1:167	IX	LOCK	GRANT	
(internal)	OBJECT	2137058649		IX	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(00009620dd9a)	X	LOCK	GRANT	
Selected process:		52					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	PAGE	72057594038714368	1:167	IU	LOCK	GRANT	
(internal)	OBJECT	2137058649		IX	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(00009620dd9a)	U	LOCK	WAIT	

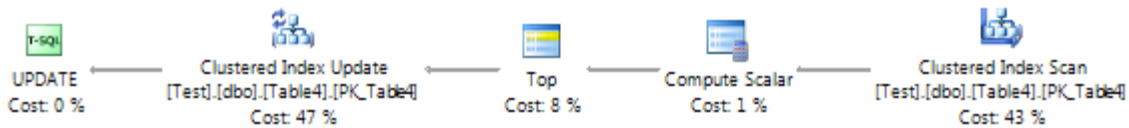
Note the Request Status = WAIT for one of the locks of the second client.

4.4. The WHERE clause uses the first column from the primary key and a non indexed column with an OR

The first client executes:

```
BEGIN TRANSACTION
UPDATE Table4 SET ItemData = ItemData + 1
WHERE GroupId = 0 OR ItemData = 10
```


SQL Server 2005 Execution Plan



SQL Server 2005 Locks

Selected process:		51					
	Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status
	(internal)	DATABASE	0		S	LOCK	GRANT
	(internal)	PAGE	72057594038714368	1:167	IX	LOCK	GRANT
	(internal)	OBJECT	2137058649		IX	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(0500a4d003ad)	X	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(09001c6fd5e7)	X	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(08007908695f)	X	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(0400c1b7bf15)	X	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(06004a7b6bf)	X	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(0b006cc0ca39)	X	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(0300788f6888)	X	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(0100f3476122)	X	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(02001de8d430)	X	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(07002f180a07)	X	LOCK	GRANT
	Test..Table4	KEY	72057594038714368	(00009620dd9a)	X	LOCK	GRANT

If the second client executes:

```
BEGIN TRANSACTION
UPDATE Table4 SET ItemData = ItemData + 1
WHERE GroupId = 2 OR ItemData = 100
```

The second client accesses different rows compared with the ones accessed by the first client. However for this particular WHERE clause the index we have is useless and the SQL Server will have to examine all the rows in the table. For each row that is examined, SQL Server will try to place an update (U) lock just to evaluate the WHERE clause. When SQL Server reaches a row that was already exclusively locked by the first client it will have to wait until the first client completes the transaction. Hence the second client is blocked:

SQL Server 2005 Locks

Selected process:		51					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	OBJECT	2137058649		IX	LOCK	GRANT	
(internal)	PAGE	72057594038714368	1:167	IX	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(0100f3476122)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(07002f180a07)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(09001c6fd5e7)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(08007908695f)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(02001de8d430)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(0400c1b7bf15)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(0300788f6888)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(0500a4d003ad)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(00009620dd9a)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(06004a7fb6bf)	X	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(0b006cc0ca39)	X	LOCK	GRANT	

Selected process:		52					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
(internal)	OBJECT	2137058649		IX	LOCK	GRANT	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	PAGE	72057594038714368	1:167	IU	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(00009620dd9a)	U	LOCK	WAIT	

Note the Request Status = WAIT for one of the locks of the second client.

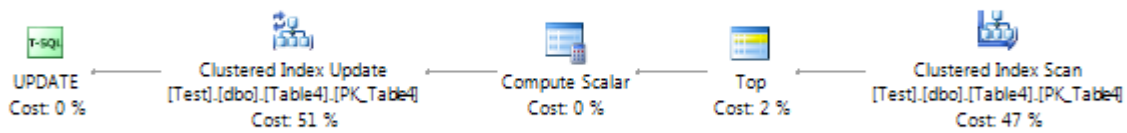
4.5. *The WHERE clause uses the second column from the primary key*

Using the second column in a composite index will not allow SQL Server to take advantage of the index. The two clients will end up stepping on each other toes.

The first client executes:

```
BEGIN TRANSACTION
UPDATE Table4 SET ItemData = ItemData + 1 WHERE ItemId = 0
```

SQL Server 2005 Execution Plan



SQL Server 2005 Locks

Selected process:		51					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	PAGE	72057594038714368	1:167	IX	LOCK	GRANT	
(internal)	OBJECT	2137058649		IX	LOCK	GRANT	
Test..Table4	KEY	72057594038714368	(00009620dd9a)	X	LOCK	GRANT	





The second client executes:





```
BEGIN TRANSACTION
UPDATE Table4 SET ItemData = ItemData + 1 WHERE ItemId = 1
```

Again, this is similar with having in the WHERE clause a column that does not participate at all in the index. Since ItemId is the second column of a composite index it is useless for this WHERE clause. SQL Server has to loop through all rows to identify the rows that satisfy the condition ItemId = 1.

The second client is blocked:

SQL Server 2005 Locks

Selected process:		51					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
 (internal)	DATABASE	0		S	LOCK	GRANT	
 (internal)	PAGE	72057594038714368	1:167	IX	LOCK	GRANT	
 (internal)	OBJECT	2137058649		IX	LOCK	GRANT	
 Test..Table4	KEY	72057594038714368	(00009620dd9a)	X	LOCK	GRANT	

Selected process:		52					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
 (internal)	DATABASE	0		S	LOCK	GRANT	
 (internal)	PAGE	72057594038714368	1:167	IU	LOCK	GRANT	
 (internal)	OBJECT	2137058649		IX	LOCK	GRANT	
 Test..Table4	KEY	72057594038714368	(00009620dd9a)	U	LOCK	WAIT	

Note the Request Status = WAIT for one of the locks of the second client.

4.6. The WHERE clause uses the second column of the primary key. That column also has its own index

This case is exemplified by table Table4b. It is a table with a composite clustered index and a separate non clustered index:

```
CREATE TABLE dbo.Table4b (
  GroupId      int NOT NULL,
  ItemId       int NOT NULL,
  ItemData     int NULL
)
GO

ALTER TABLE dbo.Table4b ADD CONSTRAINT PK_Table4b PRIMARY KEY
CLUSTERED (GroupId, ItemId)
GO

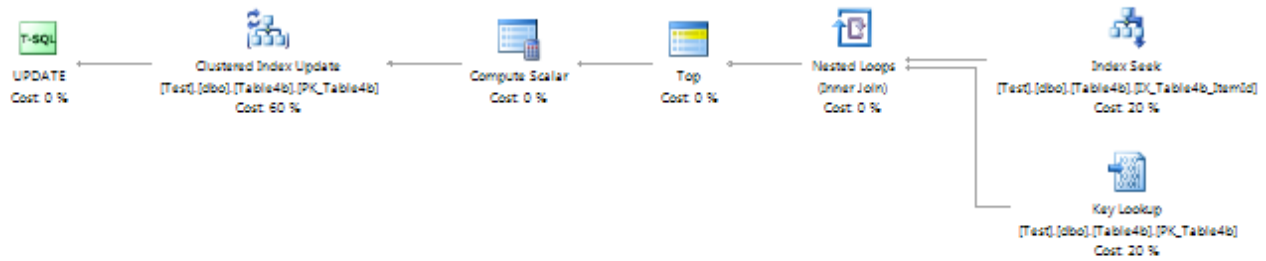
CREATE INDEX IX Table4b ItemId ON dbo.Table4b (ItemId)
GO
```

This case is similar with case 4.5. However, to improve the query performance we added a separate index on the second column (ItemId) of the primary key.

The first client executes:

```
BEGIN TRANSACTION
UPDATE Table4b SET ItemData = ItemData + 1 WHERE ItemId = 0
```

SQL Server 2005 Execution Plan



SQL Server 2005 Locks

Selected process: 51





	Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status
	(internal)	DATABASE	0		S	LOCK	GRANT
	(internal)	OBJECT	21575115		IX	LOCK	GRANT
	(internal)	PAGE	72057594038845440	1:169	IX	LOCK	GRANT
	Test..Table4b	KEY	72057594038845440	(00009620dd9a)	X	LOCK	GRANT





The second client executes:

```
BEGIN TRANSACTION
UPDATE Table4b SET ItemData = ItemData + 1 WHERE ItemId = 1
```

The second client is not blocked:

SQL Server 2005 Locks

Selected process:		51					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
 (internal)	DATABASE	0		S	LOCK	GRANT	
 (internal)	OBJECT	21575115		IX	LOCK	GRANT	
 (internal)	PAGE	72057594038845440	1:169	IX	LOCK	GRANT	
 Test..Table4b	KEY	72057594038845440	(00009620dd9a)	X	LOCK	GRANT	

Selected process:		52					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
 (internal)	DATABASE	0		S	LOCK	GRANT	
 (internal)	PAGE	72057594038845440	1:169	IX	LOCK	GRANT	
 (internal)	OBJECT	21575115		IX	LOCK	GRANT	
 Test..Table4b	KEY	72057594038845440	(0100f3476122)	X	LOCK	GRANT	

Note that all the locks are granted.

5. Tables with a primary key (clustered index) and a foreign key where the foreign key is not indexed

This case is exemplified by the tables Group5 and Table5. Both tables have primary keys on the columns we'll use in the WHERE clause. We also have a FK from Table5 to Group6. However we did not indexed column GroupId in table Table5.

```
CREATE TABLE dbo.Group5 (  
    GroupId      int NOT NULL,  
    GroupName    nvarchar(10) NULL  
)  
GO  
  
ALTER TABLE dbo.Group5 ADD CONSTRAINT PK_Group5 PRIMARY KEY CLUSTERED  
(GroupId)  
GO  
  
CREATE TABLE dbo.Table5 (  
    GroupId      int NOT NULL,  
    ItemId       int NOT NULL,  
    ItemData     int NULL  
)  
GO  
  
ALTER TABLE dbo.Table5 ADD CONSTRAINT PK_Table5 PRIMARY KEY CLUSTERED  
(ItemId)  
GO  
  
ALTER TABLE dbo.Table5 ADD CONSTRAINT FK_Table5_GroupId FOREIGN KEY  
(GroupId) REFERENCES dbo.Group5 (GroupId)
```

```
GO
```

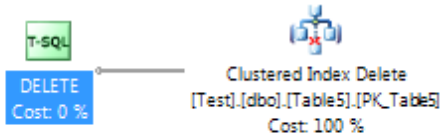
The fact that Table5.GroupId is not indexed can have negative effects regarding concurrency. That happens even though we do not have any WHERE condition that directly involves Table5.GroupId.

The first client deletes the rows from table Table5 with ItemId in the range [0, 10). These rows correspond to group GroupId = 0. Next, the first client deletes the entry from the Group5 table that has GroupId = 0. It executes:

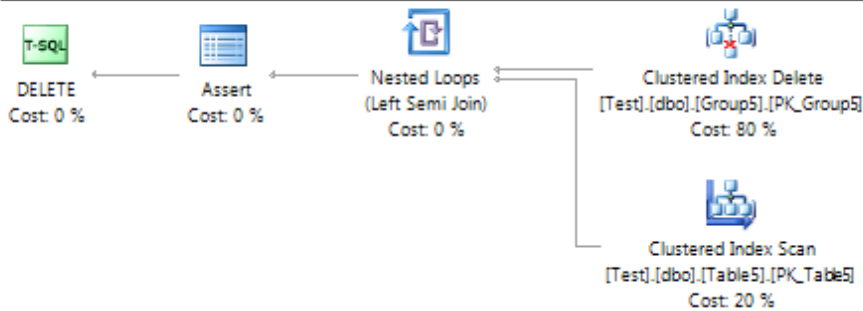
```
BEGIN TRANSACTION
DELETE Table5 WHERE ItemId >= 0 AND ItemId < 10
DELETE Group5 WHERE GroupId = 0
```

SQL Server 2005 Execution Plan

Query 1: Query cost (relative to the batch): 45%
 DELETE [Table5] WHERE [ItemID]>=@1 AND [ItemID]<@2



Query 2: Query cost (relative to the batch): 55%
 DELETE [Group5] WHERE [GroupId]=@1



SQL Server 2005 Locks

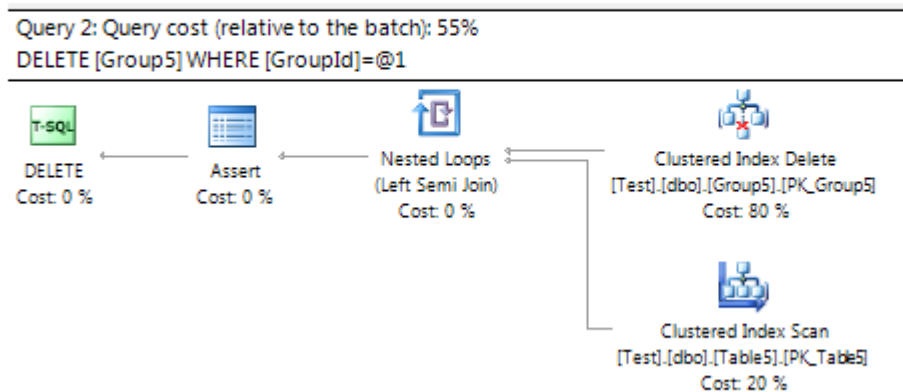
Selected process:		51					
Object	Type	Object ID	Description	Request Mode	Request Type	Request Status	
(internal)	PAGE	72057594039042048	1:155	IX	LOCK	GRANT	
(internal)	PAGE	72057594039173120	1:173	IX	LOCK	GRANT	
(internal)	OBJECT	53575229		IX	LOCK	GRANT	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	OBJECT	85575343		IX	LOCK	GRANT	
Test..Group5	KEY	72057594039042048	(0000e320bbde)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(0000e320bbde)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(07005a186c43)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(020068e8b274)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(03000d8f0ecc)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(010086470766)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(0500d1d065e9)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(0900696fb3a3)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(08000c080f1b)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(0400b4b7d951)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(06003f7d0fb)	X	LOCK	GRANT	

The second client deletes the rows from table Table5 with ItemId in the range [10, 20). These rows correspond to group GroupId = 1. Next, the second client deletes the entry from the Group5 table that has GroupId = 1. It executes:

```
BEGIN TRANSACTION
DELETE Table5 WHERE ItemId >= 10 AND ItemId < 20
DELETE Group5 WHERE GroupId = 1
```

What is interesting is that the two clients operate on different rows and use in their WHERE clauses conditions that are fully covered by indexes. And yet the second client ends up being blocked. To understand why we need to look at the execution plan of the query that deletes from Group5.

SQL Server 2005 Execution Plan



When an entry is deleted from the Group5 table, SQL Server needs to make sure that no foreign key constraint is violated. That would be the case when Table5 still had entries with GroupId = 1. For this, entries in Table5 must be examined. Since GroupId column in table Table5 does not have an index, SQL Server will scan all the rows in that table. This is shown in the execution plan by the Index Scan on the Table5. When SQL Server reads a row in Table5 it will start by putting a shared (S) lock. This shared lock is lifted as soon as the row was examined. However, the lock cannot be obtained on rows that are already exclusively locked by client 1. Hence when such a row is scanned, client 2 has to wait until the first client completes its transaction.

SQL Server 2005 Locks

Selected process:		51					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
(internal)	PAGE	72057594039042048	1:155	IX	LOCK	GRANT	
(internal)	PAGE	72057594039173120	1:173	IX	LOCK	GRANT	
(internal)	OBJECT	53575229		IX	LOCK	GRANT	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	OBJECT	85575343		IX	LOCK	GRANT	
Test..Group5	KEY	72057594039042048	(0000e320bbde)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(0000e320bbde)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(07005a186c43)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(020068e8b274)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(03000d8f0ecc)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(010086470766)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(0500d1d065e9)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(0900696fb3a3)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(08000c080f1b)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(0400b4b7d951)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(06003f7fd0fb)	X	LOCK	GRANT	
Selected process:		52					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
(internal)	OBJECT	85575343		IX	LOCK	GRANT	
(internal)	OBJECT	53575229		IX	LOCK	GRANT	
(internal)	PAGE	72057594039173120	1:173	IX	LOCK	GRANT	
(internal)	DATABASE	0		S	LOCK	GRANT	
(internal)	PAGE	72057594039042048	1:155	IX	LOCK	GRANT	
Test..Group5	KEY	72057594039042048	(010086470766)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(0d003ef8d12c)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(0b00e2a7ba09)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(0e00d057643e)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(0000e320bbde)	S	LOCK	WAIT	
Test..Table5	KEY	72057594039173120	(0c005b9fd94)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(0f00b530d886)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(10007c77a28e)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(110019101e36)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(130092d8179c)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(0a0087c006b1)	X	LOCK	GRANT	
Test..Table5	KEY	72057594039173120	(1200f7bfab24)	X	LOCK	GRANT	

Note the Request Status = WAIT for one of the locks of the second client.

6. Tables with a primary key (clustered index) and a foreign key where the foreign key is indexed

This case is exemplified by the tables Group6 and Table6. It is similar with case 5. However, here the foreign key GroupId in Table6 is indexed and the scenario does not result in the second client being blocked.




























```
CREATE TABLE dbo.Group6 (  
    GroupId      int NOT NULL,  
    GroupName    nvarchar(10) NULL  
)  
GO  
  
ALTER TABLE dbo.Group6 ADD CONSTRAINT PK_Group6 PRIMARY KEY CLUSTERED  
(GroupId)  
GO  
  
CREATE TABLE dbo.Table6 (  
    GroupId      int NOT NULL,  
    ItemId       int NOT NULL,  
    ItemData     int NULL  
)  
GO  
  
ALTER TABLE dbo.Table6 ADD CONSTRAINT PK_Table6 PRIMARY KEY CLUSTERED  
(ItemId)  
GO  
  
CREATE INDEX IX_Table6_GroupId ON dbo.Table6 (GroupId)  
GO  
  
ALTER TABLE dbo.Table6 ADD CONSTRAINT FK_Table6_GroupId FOREIGN KEY  
(GroupId) REFERENCES dbo.Group6 (GroupId)  
GO
```

We'll execute the same steps as for case 5.

The first client deletes the rows from table Table5 with ItemId in the range [0, 10). These rows correspond to group GroupId = 0. Next, the first client deletes the entry from the Group5 table that has GroupId = 0. It executes:

```
BEGIN TRANSACTION  
DELETE Table6 WHERE ItemId >= 0 AND ItemId < 10  
DELETE Group6 WHERE GroupId = 0
```

SQL Server 2005 Locks

Selected process:		51					
Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status	
 (internal)	DATABASE	0		S	LOCK	GRANT	
 (internal)	PAGE	72057594039304192	1:157	IX	LOCK	GRANT	
 (internal)	PAGE	72057594039435264	1:175	IX	LOCK	GRANT	
 (internal)	PAGE	72057594039500800	1:177	IX	LOCK	GRANT	
 (internal)	OBJECT	133575514		IX	LOCK	GRANT	
 (internal)	OBJECT	165575628		IX	LOCK	GRANT	
 Test..Group6	KEY	72057594039304192	(0000e320bbde)	X	LOCK	GRANT	
 Test..Table6	KEY	72057594039500800	(07002f180a07)	X	LOCK	GRANT	
 Test..Table6	KEY	72057594039435264	(0000e320bbde)	X	LOCK	GRANT	
 Test..Table6	KEY	72057594039500800	(09001c6fd5e7)	X	LOCK	GRANT	
 Test..Table6	KEY	72057594039500800	(0500a4d003ad)	X	LOCK	GRANT	
 Test..Table6	KEY	72057594039435264	(020068e8b274)	X	LOCK	GRANT	
 Test..Table6	KEY	72057594039435264	(03000d8f0ecc)	X	LOCK	GRANT	
 Test..Table6	KEY	72057594039500800	(08007908695f)	X	LOCK	GRANT	
 Test..Table6	KEY	72057594039500800	(0400c1b7bf15)	X	LOCK	GRANT	
 Test..Table6	KEY	72057594039500800	(06004a7b6bf)	X	LOCK	GRANT	
 Test..Table6	KEY	72057594039435264	(010086470766)	X	LOCK	GRANT	
 Test..Table6	KEY	72057594039500800	(0300788f6888)	X	LOCK	GRANT	
 Test..Table6	KEY	72057594039435264	(08000c080f1b)	X	LOCK	GRANT	
 Test..Table6	KEY	72057594039435264	(0400b4b7d951)	X	LOCK	GRANT	
 Test..Table6	KEY	72057594039500800	(0100f3476122)	X	LOCK	GRANT	
 Test..Table6	KEY	72057594039435264	(06003f7d0fb)	X	LOCK	GRANT	
 Test..Table6	KEY	72057594039500800	(02001de8d430)	X	LOCK	GRANT	
 Test..Table6	KEY	72057594039435264	(07005a186c43)	X	LOCK	GRANT	
 Test..Table6	KEY	72057594039435264	(0500d1d065e9)	X	LOCK	GRANT	
 Test..Table6	KEY	72057594039435264	(0900696fb3a3)	X	LOCK	GRANT	
 Test..Table6	KEY	72057594039500800	(00009620dd9a)	X	LOCK	GRANT	

One interesting thing is that we have a lot more exclusive locks than in case 5. This is because we have 10 more locks applied on the index on column GroupId of table Table6.

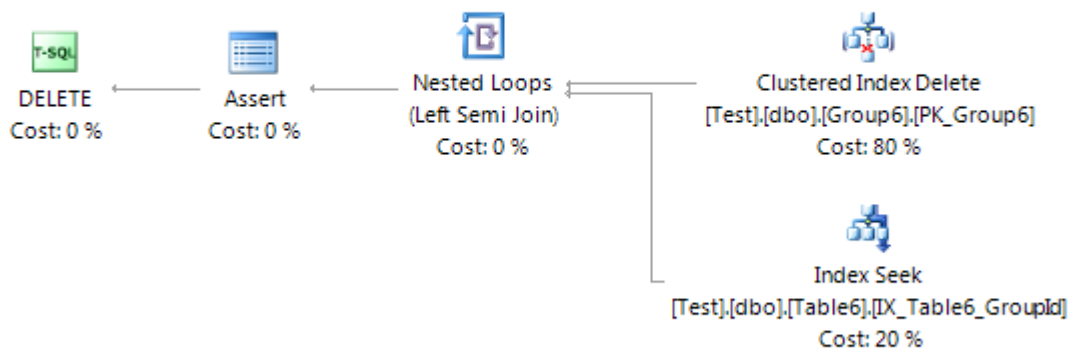
The second client deletes the rows from table Table5 with ItemId in the range [10, 20). These rows correspond to group GroupId = 1. Next, the first client deletes the entry from the Group5 table that has GroupId = 1. It executes:

```
BEGIN TRANSACTION
DELETE Table6 WHERE ItemId >= 10 AND ItemId < 20
DELETE Group6 WHERE GroupId = 1
```

This time the second client is not blocked.

Here is the execution plan of the query that deletes from Group6. Note that we have an index seek (not a index scan) where the table Table6 is examined to ensure the foreign key is not violated.

SQL Server 2005 Execution Plan



For brevity we'll show here only the locks taken by the second client. The ones taken by the first client are shown a few paragraphs above and remain the same.

SQL Server 2005 Locks

Selected process:		52					
	Object ^	Type	Object ID	Description	Request Mode	Request Type	Request Status
	(internal)	OBJECT	165575628		IX	LOCK	GRANT
	(internal)	OBJECT	133575514		IX	LOCK	GRANT
	(internal)	PAGE	72057594039500800	1:177	IX	LOCK	GRANT
	(internal)	PAGE	72057594039435264	1:175	IX	LOCK	GRANT
	(internal)	DATABASE	0		S	LOCK	GRANT
	(internal)	PAGE	72057594039304192	1:157	IX	LOCK	GRANT
	Test..Group6	KEY	72057594039304192	(010086470766)	X	LOCK	GRANT
	Test..Table6	KEY	72057594039435264	(0f00b530d886)	X	LOCK	GRANT
	Test..Table6	KEY	72057594039500800	(13001cbf67ac)	X	LOCK	GRANT
	Test..Table6	KEY	72057594039500800	(1200f210d2be)	X	LOCK	GRANT
	Test..Table6	KEY	72057594039500800	(0b006cc0ca39)	X	LOCK	GRANT
	Test..Table6	KEY	72057594039500800	(10005e30140e)	X	LOCK	GRANT
	Test..Table6	KEY	72057594039435264	(0d003ef8d12c)	X	LOCK	GRANT
	Test..Table6	KEY	72057594039500800	(110097776e06)	X	LOCK	GRANT
	Test..Table6	KEY	72057594039435264	(0c005b9f6d94)	X	LOCK	GRANT
	Test..Table6	KEY	72057594039500800	(0d00b09fa11c)	X	LOCK	GRANT
	Test..Table6	KEY	72057594039435264	(0e00d057643e)	X	LOCK	GRANT
	Test..Table6	KEY	72057594039435264	(0a0087c006b1)	X	LOCK	GRANT
	Test..Table6	KEY	72057594039435264	(0b00e2a7ba09)	X	LOCK	GRANT
	Test..Table6	KEY	72057594039435264	(130092d8179c)	X	LOCK	GRANT
	Test..Table6	KEY	72057594039435264	(110019101e36)	X	LOCK	GRANT
	Test..Table6	KEY	72057594039435264	(10007c77a28e)	X	LOCK	GRANT
	Test..Table6	KEY	72057594039500800	(140079d8db14)	X	LOCK	GRANT
	Test..Table6	KEY	72057594039500800	(0f003b57a8b6)	X	LOCK	GRANT
	Test..Table6	KEY	72057594039500800	(0e00d5f81da4)	X	LOCK	GRANT
	Test..Table6	KEY	72057594039500800	(0c0009a77681)	X	LOCK	GRANT
	Test..Table6	KEY	72057594039435264	(1200f7bfab24)	X	LOCK	GRANT

Note that all locks are granted.

The sample database

```
USE tempdb
GO

IF EXISTS (SELECT * FROM master.dbo.sysdatabases WHERE Name = 'Test')
BEGIN
    DROP DATABASE Test
END
GO

CREATE DATABASE Test
GO

USE Test

CREATE TABLE dbo.Table1 (
    GroupId      int NOT NULL,
    ItemId       int NOT NULL,
    ItemData     int NULL
)
GO

CREATE TABLE dbo.Table2 (
    GroupId      int NOT NULL,
    ItemId       int NOT NULL,
    ItemData     int NULL
)
GO

ALTER TABLE dbo.Table2 ADD CONSTRAINT
PK_Table2 PRIMARY KEY CLUSTERED (ItemId)
GO

CREATE TABLE dbo.Table3 (
    GroupId      int NOT NULL,
    ItemId       int NOT NULL,
    ItemData     int NULL
)
GO

CREATE INDEX IX_Table3_ItemId ON dbo.Table3 (ItemId)
GO

CREATE TABLE dbo.Table4 (
    GroupId      int NOT NULL,
    ItemId       int NOT NULL,
    ItemData     int NULL
)
GO

ALTER TABLE dbo.Table4 ADD CONSTRAINT
```

```

PK_Table4 PRIMARY KEY CLUSTERED (GroupId, ItemId)
GO

CREATE TABLE dbo.Table4b (
    GroupId      int NOT NULL,
    ItemId       int NOT NULL,
    ItemData     int NULL
)
GO

ALTER TABLE dbo.Table4b ADD CONSTRAINT PK_Table4b PRIMARY KEY CLUSTERED
(GroupId, ItemId)
GO

CREATE INDEX IX_Table4b_ItemId ON dbo.Table4b (ItemId)
GO

CREATE TABLE dbo.Group5 (
    GroupId      int NOT NULL,
    GroupName    nvarchar(10) NULL
)
GO

ALTER TABLE dbo.Group5 ADD CONSTRAINT PK_Group5 PRIMARY KEY CLUSTERED
(GroupId)
GO

CREATE TABLE dbo.Table5 (
    GroupId      int NOT NULL,
    ItemId       int NOT NULL,
    ItemData     int NULL
)
GO

ALTER TABLE dbo.Table5 ADD CONSTRAINT PK_Table5 PRIMARY KEY CLUSTERED
(ItemId)
GO

ALTER TABLE dbo.Table5 ADD CONSTRAINT FK_Table5_GroupId FOREIGN KEY
(GroupId) REFERENCES dbo.Group5 (GroupId)
GO

CREATE TABLE dbo.Group6 (
    GroupId      int NOT NULL,
    GroupName    nvarchar(10) NULL
)
GO

ALTER TABLE dbo.Group6 ADD CONSTRAINT PK_Group6 PRIMARY KEY CLUSTERED
(GroupId)
GO

CREATE TABLE dbo.Table6 (
    GroupId      int NOT NULL,
    ItemId       int NOT NULL,
    ItemData     int NULL
)

```



```

)
GO

ALTER TABLE dbo.Table6 ADD CONSTRAINT PK_Table6 PRIMARY KEY CLUSTERED
(ItemId)
GO

CREATE INDEX IX_Table6_GroupId ON dbo.Table6 (GroupId)
GO

ALTER TABLE dbo.Table6 ADD CONSTRAINT FK_Table6_GroupId FOREIGN KEY
(GroupId) REFERENCES dbo.Group6(GroupId)
GO

DECLARE @CrtIndex int
DECLARE @GroupId int

SET @GroupId = 0
WHILE @GroupId < 200
BEGIN
    INSERT INTO Group5 (GroupId, GroupName)
    VALUES (@GroupId, 'Group ' + CAST(@GroupId AS nvarchar(3)))

    INSERT INTO Group6 (GroupId, GroupName)
    VALUES (@GroupId, 'Group ' + CAST(@GroupId AS nvarchar(3)))

    SET @GroupId = @GroupId + 1
END

SET @CrtIndex = 0
WHILE @CrtIndex < 2000
BEGIN
    SET @GroupId = @CrtIndex / 10

    INSERT INTO Table1 (GroupId, ItemId, ItemData)
    VALUES (@GroupId, @CrtIndex, @CrtIndex)

    INSERT INTO Table2 (GroupId, ItemId, ItemData)
    VALUES (@GroupId, @CrtIndex, @CrtIndex)

    INSERT INTO Table3 (GroupId, ItemId, ItemData)
    VALUES (@GroupId, @CrtIndex, @CrtIndex)

    INSERT INTO Table4 (GroupId, ItemId, ItemData)
    VALUES (@GroupId, @CrtIndex, @CrtIndex)

    INSERT INTO Table4b (GroupId, ItemId, ItemData)
    VALUES (@GroupId, @CrtIndex, @CrtIndex)

    INSERT INTO Table5 (GroupId, ItemId, ItemData)
    VALUES (@GroupId, @CrtIndex, @CrtIndex)

    INSERT INTO Table6 (GroupId, ItemId, ItemData)
    VALUES (@GroupId, @CrtIndex, @CrtIndex)

    SET @CrtIndex = @CrtIndex +1

```

```
END
```

```
GO
```

```
USE tempdb
```

```
GO
```