SQL Indexes Impact on Concurrent Data Access

www.ladiMolnar.com

Introduction .................................................................................................................................................. 2
Why is this interesting? .................................................................................................................................. 2
Prerequisites .................................................................................................................................................. 2
How to repro 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 tables 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 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:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GroupId</td>
<td>int</td>
<td>NOT NULL, 0 to 199. One group for each 10 rows</td>
</tr>
<tr>
<td>ItemId</td>
<td>int</td>
<td>NOT NULL, 0 to 1999</td>
</tr>
<tr>
<td>ItemData</td>
<td>int</td>
<td>NULL, 0 to 1999</td>
</tr>
</tbody>
</table>
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:
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

<table>
<thead>
<tr>
<th>Process ID</th>
<th>Lock Type</th>
<th>Mode</th>
<th>Status</th>
<th>Index</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
<td>TAB</td>
<td>IX</td>
<td>GRANT</td>
<td></td>
<td>Table1</td>
</tr>
<tr>
<td>51</td>
<td>PAG</td>
<td>IX</td>
<td>GRANT</td>
<td></td>
<td>Table1</td>
</tr>
<tr>
<td>51</td>
<td>RDI</td>
<td>X</td>
<td>GRANT</td>
<td></td>
<td>Table1</td>
</tr>
<tr>
<td>53</td>
<td>TAB</td>
<td>IX</td>
<td>GRANT</td>
<td></td>
<td>Table1</td>
</tr>
<tr>
<td>53</td>
<td>PAG</td>
<td>IU</td>
<td>GRANT</td>
<td></td>
<td>Table1</td>
</tr>
<tr>
<td>53</td>
<td>RDI</td>
<td>U</td>
<td>WAIT</td>
<td></td>
<td>Table1</td>
</tr>
</tbody>
</table>

### SQL Server 2005 Locks

<table>
<thead>
<tr>
<th>Object</th>
<th>Type</th>
<th>Object ID</th>
<th>Description</th>
<th>Request Mode</th>
<th>Request Type</th>
<th>Request Status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DATABASE</td>
<td>2073058421</td>
<td></td>
<td>IX</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td></td>
<td>PAGE</td>
<td>72057594038321152</td>
<td>1:159</td>
<td>IX</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
</tbody>
</table>

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:

```sql
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:

```sql
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.
The second client executes:

```sql
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.
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**

<table>
<thead>
<tr>
<th>Selected process:</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Type</td>
<td>database 0</td>
</tr>
<tr>
<td>Object ID</td>
<td>2089058478</td>
</tr>
<tr>
<td>Description</td>
<td>IX LOCK Grant</td>
</tr>
<tr>
<td>Request Status</td>
<td>WAIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Selected process:</th>
<th>52</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Type</td>
<td>page 1:161</td>
</tr>
<tr>
<td>Object ID</td>
<td>72057594038452224</td>
</tr>
<tr>
<td>Request Status</td>
<td>WAIT</td>
</tr>
</tbody>
</table>

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.
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.
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
```
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:
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:

```sql
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:

```sql
BEGIN TRANSACTION
```
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.).

The second client executes:

```
BEGIN TRANSACTION
UPDATE Table3 SET ItemData = ItemData + 1 WHERE ItemId = 100
```
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
```
The second client executes:

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

The second client is blocked:
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:

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

---

**SQL Server 2005 Execution Plan**

![Execution Plan Diagram]

**SQL Server 2005 Locks**

<table>
<thead>
<tr>
<th>Selected process</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Type</td>
<td></td>
</tr>
<tr>
<td>(internal)</td>
<td>DATABASE 0</td>
</tr>
<tr>
<td>(internal)</td>
<td>PAGE 72057594038714368 1:167</td>
</tr>
<tr>
<td>(internal)</td>
<td>OBJECT 2137058649</td>
</tr>
<tr>
<td>Test_Table4</td>
<td>KEY 72057594038714368 (00009620dd5a)</td>
</tr>
</tbody>
</table>

---

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:
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
```
The second client executes:

```sql
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:
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**

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:
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:
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
```
If the second client executes:

```sql
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:
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:

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

```sql
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

<table>
<thead>
<tr>
<th>Selected process:</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Type</td>
<td>Description</td>
</tr>
<tr>
<td>(internal) DATABASE 0</td>
<td>S</td>
</tr>
<tr>
<td>(internal) OBJECT 21575115</td>
<td>IX</td>
</tr>
<tr>
<td>(internal) PAGE 72057594038845440 1:169</td>
<td>IX</td>
</tr>
<tr>
<td>Test.Table4b KEY 72057594038845440 (00009620dd9a)</td>
<td>X</td>
</tr>
</tbody>
</table>

The second client executes:

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

The second client is not blocked:
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.

```sql
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)
```
The fact that Table5.GroupId is not indexed can have negative effects regarding concurrency. That happens even though we do not have any WHER 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

<table>
<thead>
<tr>
<th>Selected process</th>
<th>Type</th>
<th>Object ID</th>
<th>Description</th>
<th>Request Mode</th>
<th>Request Type</th>
<th>Request Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>(internal)</td>
<td>PAGE</td>
<td>72057594039042048</td>
<td>1:155</td>
<td>IX</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>(internal)</td>
<td>PAGE</td>
<td>72057594039173120</td>
<td>1:173</td>
<td>IX</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>(internal)</td>
<td>OBJECT</td>
<td>53575229</td>
<td></td>
<td>IX</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>(internal)</td>
<td>DATABASE</td>
<td>0</td>
<td></td>
<td>S</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>(internal)</td>
<td>OBJECT</td>
<td>85575343</td>
<td></td>
<td>IX</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Group5</td>
<td>KEY</td>
<td>72057594039042048</td>
<td>(0000e320bbde)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Table5</td>
<td>KEY</td>
<td>72057594039173120</td>
<td>(0000e320bbde)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Table5</td>
<td>KEY</td>
<td>72057594039173120</td>
<td>(07005a186c43)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Table5</td>
<td>KEY</td>
<td>72057594039173120</td>
<td>(020069e8b274)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Table5</td>
<td>KEY</td>
<td>72057594039173120</td>
<td>(03000d5480ec)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Table5</td>
<td>KEY</td>
<td>72057594039173120</td>
<td>(010086470766)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Table5</td>
<td>KEY</td>
<td>72057594039173120</td>
<td>(0500d1d05e5e9)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Table5</td>
<td>KEY</td>
<td>72057594039173120</td>
<td>(0900696b3a3)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Table5</td>
<td>KEY</td>
<td>72057594039173120</td>
<td>(08000c080f1b)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Table5</td>
<td>KEY</td>
<td>72057594039173120</td>
<td>(040db4b7d551)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Table5</td>
<td>KEY</td>
<td>72057594039173120</td>
<td>(060037fe00b)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Object</th>
<th>Type</th>
<th>Object ID</th>
<th>Description</th>
<th>Request Mode</th>
<th>Request Type</th>
<th>Request Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>(internal)</td>
<td>PAGE</td>
<td>7205794039042048</td>
<td>1:155</td>
<td>IX</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>(internal)</td>
<td>PAGE</td>
<td>7205794039173120</td>
<td>1:173</td>
<td>IX</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>(internal)</td>
<td>OBJECT</td>
<td>53575229</td>
<td></td>
<td>IX</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>(internal)</td>
<td>DATABASE</td>
<td>0</td>
<td></td>
<td>S</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>(internal)</td>
<td>OBJECT</td>
<td>85575343</td>
<td></td>
<td>IX</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Group5</td>
<td>KEY</td>
<td>7205794039042048</td>
<td>(0000e320bbde)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Table5</td>
<td>KEY</td>
<td>7205794039173120</td>
<td>(0000e320bbde)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Table5</td>
<td>KEY</td>
<td>7205794039173120</td>
<td>(07005a186c43)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Table5</td>
<td>KEY</td>
<td>7205794039173120</td>
<td>(020068e8b274)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Table5</td>
<td>KEY</td>
<td>7205794039173120</td>
<td>(0300d8f0ecc)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Table5</td>
<td>KEY</td>
<td>7205794039173120</td>
<td>(01008470766)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Table5</td>
<td>KEY</td>
<td>7205794039173120</td>
<td>(0500d16065e9)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Table5</td>
<td>KEY</td>
<td>7205794039173120</td>
<td>(0900696b3a3)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Table5</td>
<td>KEY</td>
<td>7205794039173120</td>
<td>(08000c0b91b)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Table5</td>
<td>KEY</td>
<td>7205794039173120</td>
<td>(0400bb7d551)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>Test..Table5</td>
<td>KEY</td>
<td>7205794039173120</td>
<td>(060037d0b)</td>
<td>X</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
</tbody>
</table>

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.

```sql
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:

```sql
BEGIN TRANSACTION
DELETE Table6 WHERE ItemId >= 0 AND ItemId < 10
DELETE Group6 WHERE GroupId = 0
```
One interesting thing is that we have a lot more exclusive locks that 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.

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

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Object ID</th>
<th>Description</th>
<th>Request Mode</th>
<th>Request Type</th>
<th>Request Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>(internal) OBJECT</td>
<td>165575628</td>
<td>IX</td>
<td>LOCK</td>
<td>GRANT</td>
<td></td>
</tr>
<tr>
<td>(internal) OBJECT</td>
<td>133575514</td>
<td>IX</td>
<td>LOCK</td>
<td>GRANT</td>
<td></td>
</tr>
<tr>
<td>(internal) PAGE</td>
<td>72057594039500800</td>
<td>1:177</td>
<td>IX</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>(internal) PAGE</td>
<td>72057594039435264</td>
<td>1:175</td>
<td>IX</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
<tr>
<td>(internal) DATABASE</td>
<td>0</td>
<td>S</td>
<td>LOCK</td>
<td>GRANT</td>
<td></td>
</tr>
<tr>
<td>(internal) PAGE</td>
<td>72057594039304192</td>
<td>1:157</td>
<td>IX</td>
<td>LOCK</td>
<td>GRANT</td>
</tr>
</tbody>
</table>

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
USE tempdb
GO