SQL SERVER CONCURRENCY, TEIL 3

Die sechs Gesichter der Deadlocks

Wie Sie verhindern, dass sich zwei Transaktionen gegenseitig blockieren.

Schon in den beiden vorangegangenen Teilen dieser Miniserie [1] [2] haben Sie erfahren, dass der SQL Server gleichzeitig arbeitende Benutzer abhängig vom gewählten Transaction Isolation Level voneinander isoliert. Dafür verwendet er diese Locks:

- Für schreibende Transaktionen werden immer exklusive Locks angefordert.
- Lesende Transaktionen können abhängig vom gewählten Transaction Isolation Level Shared Locks anfordern.

So weit, so gut. Das große Problem von Locks ist aber, dass diese nicht alle kompatibel zueinander sind. Ein Shared Lock ist zum Beispiel nicht kompatibel mit einem Exclusive Lock. Fordern Sie trotzdem einen Shared Lock an, wird dieser erst geliefert, wenn der exklusive Lock nicht mehr aktiv ist, es entsteht eine klassische Blocking-Situation.

Kritisch wird es, wenn die zweite Session ebenfalls einen inkompatiblen Lock anfordert. Dann wartet die erste Session auf die zweite, während die zweite auf die erste Session wartet. Nichts geht mehr, es ist eine Deadlocking-Situation entstanden.

Deadlock-Handling im SQL Server

Das Nette am SQL Server ist, dass er Deadlock-Situationen selbstständig auflösen kann. Das erledigt ein Hintergrundprozess namens Deadlock Monitor, der alle paar Sekunden prüft, ob Deadlocks vorhanden sind.

Hat er eine Deadlocking-Situation identifiziert, macht er die günstigste Transaktion rückgängig. Dadurch werden die angeforderten Locks wieder freigegeben und die blockierte Transaktion kann mit ihrer Arbeit fortfahren. Die günstigste Transaktion ist dabei diejenige, welche die wenigsten Daten ins Transaktions-Log geschrieben hat. Bei einem Deadlock zwischen einer lesenden und einer schreibenden Transaktion wird deshalb immer die lesende Transaktion zurückgerollt, da diese keine Daten ins Transaktions-Log schreibt. Die vom Deadlock Monitor zurückgerollte Transaktion wird auch als Deadlock Victim bezeichnet.

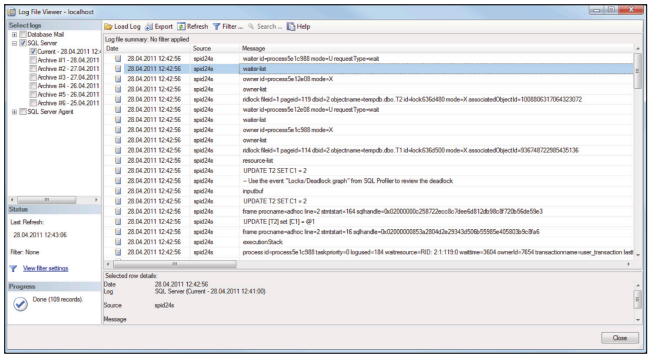
Die als Deadlock Victim ausgewählte Transaktion erhält die Fehlernummer 1205. Tritt diese Meldung auf, sollte der Datenzugriffscode der Anwendung die Transaktion einfach noch einmal ausführen. Da der Deadlock seitens des SQL Server beim Ausgeben der Fehlernummer bereits aufgelöst wurde, sollte beim nächsten Versuch kein Deadlock mehr auftreten. Allerdings sollte der Code nicht unendlich lange versuchen, eine mit dem Fehler 1205 quittierte Transaktion erneut auszuführen, sondern die Versuche nach mehrmaligem Fehlschlagen beenden und den Benutzer informieren, dass es Probleme beim Zugriff auf die Datenbank gibt.

Ein Deadlock ist aber eine Situation, die Ihre Anwendung in den meisten Fällen ohne Benutzerintervention lösen kann – solange Ihr Datenzugriffscode stets das Auftreten der Fehlernummer 1205 überprüft und angemessen darauf reagiert.

Deadlock-Troubleshooting

Das Bordwerkzeug des SQL Server enthält drei Technologien zur Analyse und Behandlung von Deadlocking-Situationen:

- Trace Flag 1222
- SQL Server Profiler
- Extended Events



Ein Blick in die Logs des SQL Server (Bild 1)

XML-Liste zu einem Deadlock (Bild 2)

Generell empfehle ich den Einsatz von Extended Events für den Umgang mit Deadlocks. Da es sich beim SQL Server Profiler bereits um eine von Microsoft abgekündigte Funktion (deprecated) handelt, wird sie in einer der nächsten Versionen der Datenbank nicht mehr enthalten sein. Das Trace Flag 1222 passt nicht mehr so recht in die Zeit, da hier lediglich XML-Informationen über den aufgetretenen Deadlock ins SQL-Server-Log geschrieben werden (Bild 1).

Die leistungsfähigste Möglichkeit zum Troubleshooting bei Deadlocks bieten die Extended Events, da der SQL Server hier eine Vielzahl unterschiedlicher Informationen in Form eines XML-Dokuments zurückliefert. Bild 2 zeigt einen Ausschnitt daraus. Es handelt sich dabei um den sogenannten Deadlock-Graphen, der die folgenden XML-Knoten auf oberster Ebene beinhaltet:

- *<process-list>*
- *<resource-list>*

Der Knoten *<process-list>* beschreibt, welche Abfragen beim Deadlock beteiligt waren. Dadurch können Sie sehr leicht identifizieren, bei welchen Transaktionen der Deadlock aufgetreten ist. Zudem liefert der Knoten Informationen darüber, welche Ressourcen (Locks) beim abgebildeten Deadlock involviert waren.

Sie sehen über eine *<owner-List>*, welche Locks bereits von den Transaktionen gehalten wurden, und die *<waiter-List>* gibt Auskunft, auf welche Locks aktuell gewartet wurde. Zusätzlich sind *<owner-List>* und *<waiter-List>* mit der entsprechenden Transaktion in der *<process-list>* verknüpft.

Anhand dieser Informationen im Deadlock-Graphen lässt sich recht einfach herausfinden, warum der Deadlock aufgetreten ist. Generell ist der Deadlock-Graph die erste Anlaufstelle, wenn Sie eine Deadlocking-Situation analysieren.

Ich bekomme sehr oft E-Mail-Anfragen, warum zwei verschiedene Abfragen in einen Deadlock gelaufen sind. Meist sind im Anhang der Mails noch die zugehörigen SQL-Statements zu finden. Das bringt aber nicht viel, da der Deadlock im SQL Server auf physischer Ebene auftritt, die SQL-Statements aber immer auf logischer Ebene beschreiben, was die Datenbank tun soll. Daher fordere ich für die Analyse immer den zugehörigen Deadlock-Graphen an, da nur dieser beschreibt, was auf physischer Ebene passiert ist. Ohne Deadlock-Graph ist die Analyse fast unmöglich!

Cycle Deadlock

Wie oben schon angedeutet, gibt es verschiedene Arten von Deadlocks. Die einfachste davon ist ein Cycle Deadlock. Er ensteht, wenn auf Ressourcen in unterschiedlicher Reihenfolge zugegriffen wird. Sehen Sie sich dazu Listing 1 und 2 an. Wie zu erkennen ist, greifen die beiden Sessions auf dieselben beiden Tabellen zu, aber in unterschiedlicher Reihenfolge: Die erste Session greift auf Tabelle *T1* und dann auf Tabelle *T2* zu, die zweite Session greift auf Tabelle *T2* und danach auf Tabelle *T1* zu.

Wenn Sie die Statements Schritt für Schritt abwechselnd für die beiden Transaktionen durchführen, werden Sie in einen Deadlock laufen, der ein paar Sekunden später durch den ▶

Listing 1: Cycle Deadlock – Session 1

```
USE TempDB
GO

CREATE TABLE T1
(
   C1 INT
)
GO

INSERT INTO T1 VALUES (1)
GO

CREATE TABLE T2
(
   C1 INT
)

INSERT INTO T2 VALUES (1)
GO

SELECT * FROM T1
SELECT * FROM T2
GO

BEGIN TRANSACTION

UPDATE T1 SET C1 = 2

UPDATE T2 SET C1 = 2

-- COMMIT TRANSACTION

DROP TABLE T1
DROP TABLE T2
GO
```

Listing 2: Cycle Deadlock – Session 2

```
USE TempDB
GO

BEGIN TRANSACTION

UPDATE T2 SET C1 = 1

UPDATE T1 SET C1 = 1

COMMIT TRANSACTION
GO
```

Deadlock Monitor erkannt und schlussendlich durch das Zurückrollen der günstigsten Transaktion aufgelöst wird.

Dieser zyklische Deadlock lässt sich ganz einfach vermeiden: Es genügt, wenn Sie auf Tabellen immer in der gleichen Reihenfolge zugreifen. Ich habe schon mehrfach mit Kunden zusammengearbeitet, deren Entwickler angehalten waren, auf Tabellen immer in alphabetischer Reihenfolge (auf Basis der Namensgebung) zuzugreifen, um Cycle Deadlocks zu vermeiden.

Bookmark Lookup Deadlocks

Ich habe schon einige Male in der dotnetpro über Bookmark Lookups geschrieben – beispielsweise in [3] – und berichtet, dass diese hinsichtlich Performance und Indizierungsstrategie nicht immer das beste Mittel sind. Ein weiterer negativer Seiteneffekt, der sich aus Bookmark Lookups ergeben kann, ist die Tatsache, dass Sie hier ganz leicht in Deadlocks laufen, wenn Sie parallele schreibende Transaktionen auf derselben Tabelle ausführen. Sehen Sie sich dazu Listing 3 und 4 an.

Die Listings sind nicht komplex. In Listing 3 wird in einer Endlosschleife der Clustered Key der Tabelle aktualisiert. Da der Clustered Key auch als logischer Zeiger im Non-Clustered Index vorhanden ist, muss der SQL Server zunächst den Clustered Index und danach auch den Non-Clustered Index aktualisieren. Für das Aktualisieren beider Indizes sind jeweils exklusive Locks erforderlich.

Der Code in Listing 4 greift per Bookmark Lookup auf dieselbe Tabelle zu: im ersten Schritt lesend auf den Non-Clustered Index, im zweiten Schritt lesend auf den Clustered Index. Der Zugriff auf die beteiligten Ressourcen erfolgt auch hier nicht in der gleichen Reihenfolge, wodurch ein Deadlock ganz einfach provoziert werden kann!

Der Transaction Isolation Level wurde in beiden Transaktionen auf *Repeatable Read* gesetzt, damit der Deadlock einfacher zu reproduzieren ist. In einer Produktivumgebung mit entsprechendem Workload lässt sich dieser Deadlock auch im Standard-Isolation-Level *Read Committed* reproduzieren. Bild 3 veranschaulicht diese Problematik.

Wie kann ein solcher Deadlock vermieden werden? Das Problem entsteht ja dadurch, dass der SQL Server intern in der falschen Reihenfolge auf die Ressourcen zugreift. Einfaches Umschreiben der Abfragen ist somit keine Option.

Listing 3: Bookmark Lookup Deadlock – Session 1

```
USE master
GO

CREATE DATABASE BookmarkLookupDL
GO

USE BookmarkLookupDL
GO

CREATE TABLE Deadlock (
   Col1 INT NOT NULL PRIMARY KEY CLUSTERED,
   Col2 INT NOT NULL,
   Col3 INT NOT NULL
)
GO

CREATE NONCLUSTERED INDEX idx_Col3 ON Deadlock(Col3)
GO

INSERT INTO Deadlock VALUES (1, 1, 1)
GO

SELECT * FROM Deadlock
GO

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
GO

WHILE (1 = 1)
BEGIN
   UPDATE Deadlock
   SET Col1 = Col1 + 1
   WHERE Col3 = 1
END
GO
```

Listing 4: Bookmark Lookup Deadlock – Session 2

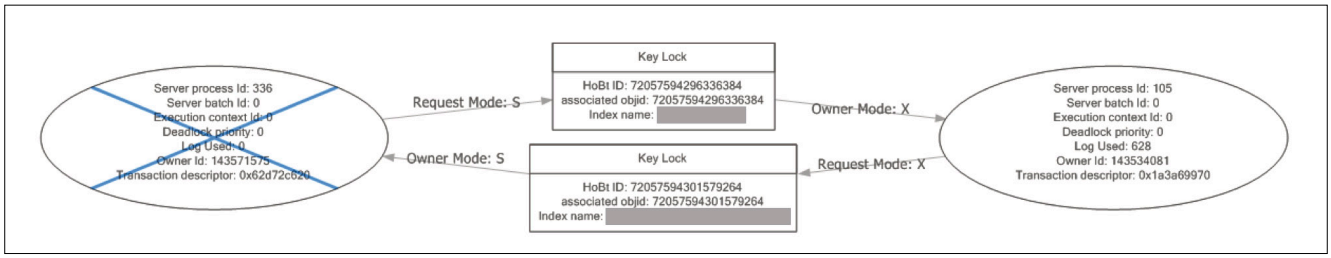
```
USE BookmarkLookupDL
GO

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
GO

WHILE (1 = 1)
BEGIN
   SELECT * FROM Deadlock WITH (INDEX(idx_Col3))
   -- The hint is necessary to overcome the Tipping
   -- Point to produce a Bookmark Lookup
   WHERE Col3 = 1
END
GO
```

Listing 5: Covering Non-Clustered Index

```
-- Creates a Covering Non-Clustered Index
CREATE NONCLUSTERED INDEX idx_Col3 ON Deadlock(Col3)
INCLUDE (Col1, Col2)
WITH (DROP_EXISTING = ON)
GO
```



Bookmark Lookup Deadlocks (Bild 3)

Sie können den Deadlock jedoch verhindern, indem Sie den Bookmark Lookup ganz vermeiden – ein Covering Non-Clustered Index hilft dabei! Wird der lesende Zugriff auf den Clustered Index vermieden, lässt sich der Deadlock lösen. Listing 5 zeigt den dazu erforderlichen Index.

Bookmark Lookup Deadlocks sind folglich ein weiterer Grund, warum Sie Bookmark Lookups wirklich nur in Spezialfällen in Kauf nehmen sollten.

Deadlocks aufgrund fehlender Indizes

Auch durch fehlende Indizes können Deadlocking-Situationen entstehen. Stellen Sie sich vor, Sie haben eine Heap Table (eine Tabelle ohne Clustered Index) und zusätzlich ist für diese Tabelle kein einziger Non-Clustered Index für den Datenzugriff definiert. In diesem Fall gibt es nur eine Möglichkeit, auf diese Tabelle zuzugreifen – nämlich über einen Table Scan Operator im Ausführungsplan.

Dadurch müssen Sie bei jedem Zugriff auf die Tabelle die komplette Tabelle lesen (und für jeden Datensatz einen Shared Lock anfordern), auch wenn Sie nur an einem spezifischen Datensatz interessiert sind. Sehen Sie sich dazu Listing 6 und 7 näher an. Sie werden erkennen, dass hier auf die beiden beteiligten Tabellen wieder in unterschiedlicher Reihenfolge zugegriffen wird. Dadurch entsteht, wie gehabt, ein Deadlock zwischen den beiden Transaktionen. Diesen Deadlock vermeiden Sie, indem Sie zusätzliche Non-Clustered ▶

Listing 6: Fehlende Indizes – Session 1

```
USE master
GO

-- Create a new database
CREATE DATABASE DeadlockingDemo
GO

-- Use it
USE DeadlockingDemo
GO

-- Create a table without any indexes
CREATE TABLE Table1 (
   Column1 INT, Column2 INT )
GO

-- Insert a few records
INSERT INTO Table1 VALUES (1, 1)
INSERT INTO Table1 VALUES (2, 2)
INSERT INTO Table1 VALUES (3, 3)
INSERT INTO Table1 VALUES (4, 4)
GO

-- Create a table without any indexes
CREATE TABLE Table2 (
   Column1 INT, Column2 INT )
GO

-- Insert a few records
INSERT INTO Table2 VALUES (1, 1)
INSERT INTO Table2 VALUES (2, 2)
INSERT INTO Table2 VALUES (3, 3)
INSERT INTO Table2 VALUES (4, 4)
GO

BEGIN TRANSACTION

   -- Acquires an Exclusive Lock on the row
   UPDATE Table1 SET Column1 = 3 WHERE Column2 = 1
   -- Execute the query from Session 2...
   -- ...

   -- This query now requests a Shared Lock, but gets
   -- blocked, because the other session/transaction
   -- has an Exclusive Lock on one row, that is
   -- currently updated
   SELECT Column1 FROM Table2
   --WITH (INDEX = idx_Column2)
   -- The index hint is necessary, because in this
   -- tiny table, SQL Server will just scan the whole
   -- table...
   WHERE Column2 = 3

ROLLBACK TRANSACTION
GO
```

Listing 7: Fehlende Indizes – Session 2

```
-- Use the previous created database
USE DeadlockingDemo
GO

BEGIN TRANSACTION
   -- Acquires an Exclusive Lock on the row
   UPDATE Table2 SET Column1 = 5 WHERE Column2 = 2

   SELECT * FROM sys.dm_tran_locks
   WHERE request_session_id = @@SPID

   -- Continue with the query from Session 2...
   -- ...

   -- This query now requests a Shared Lock, but
   -- gets blocked, because the other session/
   -- transaction has an Exclusive Lock on one row,
   -- that is currently updated
   SELECT Column1 FROM Table1
   -- WITH (INDEX = idx_Column2) -- The index hint
   -- is necessary, because in this tiny table SQL
   -- Server will just scan the whole table...
   WHERE Column2 = 4

ROLLBACK TRANSACTION
GO
```

Indizes für beide Tabellen definieren. Durch den Einsatz eines Non-Clustered Index kann der SQL Server über eine *Seek*-Operation die gewünschten Datensätze im Leaf Level des Non-Clustered Index finden, ohne auf die eigentliche Tabelle über einen Table Scan Operator zugreifen zu müssen. Und dadurch wird der Deadlock eliminiert. Listing 8 zeigt die dazu erforderlichen Non-Clustered Indizes.

Listing 8: Non-Clustered Indizes

```
-- Create new indexes so that SQL Server has alter
-- native access paths to the data. The previous
-- 2 SELECT statements can be now done through the
-- Nonclustered Index without acquiring a Shared
-- Lock on the table itself (which currently holds
-- an Exclusive Lock from the UPDATE statement).
CREATE NONCLUSTERED INDEX idx_Column2 ON
Table1(Column2)

CREATE NONCLUSTERED INDEX idx_Column2 ON
Table2(Column2)
GO
```

Listing 9: Deadlock – Session 1

```
USE AdventureWorks2014
GO

-- Causes a deadlock when we access data in the
-- wrong order
BEGIN TRANSACTION
   -- 1st range of data
   SELECT * FROM Person.Person
   WHERE ModifiedDate = '20120208'

   UPDATE Person.Person
   SET FirstName = '...'
   WHERE ModifiedDate = '20120208'
   -- Switch to session 2
   -- ...
   -- 2nd range of data
   -- This statement will cause a deadlock!
   SELECT * FROM Person.Person
   WHERE ModifiedDate = '20120209'

   UPDATE Person.Person
   SET FirstName = '...'
   WHERE ModifiedDate = '20120209'

ROLLBACK
GO
```

Deadlocks beim Zugriff auf unterschiedliche Datenbereiche

Greifen Sie auf unterschiedliche Datenbereiche in unterschiedlicher Reihenfolge zu, droht Ihnen ebenfalls ein Deadlock. Hier ein konkretes Beispiel: Listing 9 und 10 zeigen wieder zwei Transaktionen, die sich gegenseitig blockieren können. Hier handelt es sich wieder um einen klassischen Cycle Deadlock, da auf unterschiedliche Datenbereiche in unterschiedlicher Reihenfolge zugegriffen wird. Auch dieser Deadlock lässt sich vermeiden, wenn Sie auf die Datenbereiche in identischer Reihenfolge zugreifen.

Deadlocks bei Repeatable Read

Der Transaction Isolation Level *Repetable Read* ist ebenfalls sehr anfällig für Deadlocking-Situationen, da hier bekanntlich (siehe [1] [2]) die Shared Locks bis zum Ende der Transaktion gehalten werden, damit Repeatable Reads gewährleistet werden können. Sehen Sie sich dazu Listing 11 an.

Führen Sie diese Transaktion in zwei Sessions gleichzeitig aus, können Sie auch hier in einen Deadlock laufen. Das Problem dabei ist, dass keine der beiden Transaktionen den Exclusive Lock für das *UPDATE*-Statement anfordern kann, da bereits die andere Transaktion einen inkompatiblen Shared Lock auf den Datensatz angefordert hat. Dadurch blockieren beide *UPDATE*-Statements – Deadlock!

Listing 10: Deadlock – Session 2

```
USE AdventureWorks2014
GO
-- Causes a deadlock when we access data in the
-- wrong order
BEGIN TRANSACTION
   -- 2nd range of data
   SELECT * FROM Person.Person
   WHERE ModifiedDate = '20120209'

   UPDATE Person.Person
   SET FirstName = '...'
   WHERE ModifiedDate = '20120209'
   -- Switch to session 1
   -- ...
   -- 1st range of data
   -- This statement will cause a deadlock!
   SELECT * FROM Person.Person
   WHERE ModifiedDate = '20120208'

   UPDATE Person.Person
   SET FirstName = '...'
   WHERE ModifiedDate = '20120208'

ROLLBACK
GO
```

Diesen Deadlock lösen Sie auf, indem Sie zum Beispiel beim *SELECT*-Statement explizit einen Update Lock anfordern. Dieser ist kompatibel mit einem Shared Lock, aber inkompatibel mit sich selbst, und auch inkompatibel mit einem Exclusive Lock. Dadurch kann nur das erste *SELECT*-Statement ausgeführt werden. Das *SELECT*-Statement der anderen Session wird blockieren, da der Update Lock nicht angefordert werden kann.

Dank dieser Vorgehensweise serialisiert der SQL Server beide Transaktionen und führt sie der Reihe nach aus. Die Performance sinkt dadurch zwar, aber als Gegenleistung haben Sie keinen Deadlock mehr, vergleiche Listing 12.

Intra-Parallelism Deadlocks

Ein äußerst gemeiner Deadlock im SQL Server ist der Intra-Parallelism Deadlock (Bild 4). Das Gemeine daran ist, dass es sich um einen Bug im SQL Server handelt, der bewusst nicht gefixt wird. Microsoft hat sich für diese Vorgehensweise entschieden, da die Behebung des Bugs zu risikoreich wäre. Wichtig ist für Sie zu erkennen, wann Sie es mit einem Intra-Parallelism Deadlock zu tun haben. Der Graph eines solchen Deadlocks verrät, dass es sich immer um die gleiche Session ID handelt, die Abfrage folglich selbst in einen Deadlock gelaufen ist. Dies ist möglich, weil ein Ausführungsplan eines solchen Deadlocks immer mit mehreren Worker-Threads umgesetzt wird, die schlussendlich gegenseitig aufeinander ▶

Listing 11: Deadlock in Repeatable Read – Session 1

```
USE AdventureWorks2014
GO

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
GO

BEGIN TRANSACTION
   SELECT * FROM Person.Person
   WHERE ModifiedDate = '20120208'
   -- Switch to session 2
   -- ...

   -- This statement will cause a deadlock!
   UPDATE Person.Person
   SET FirstName = '...'
   WHERE ModifiedDate = '20120208'

ROLLBACK
GO
```

Listing 12: Deadlock in Repeatable Read – Session 2

```
USE AdventureWorks2014
GO

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
GO

BEGIN TRANSACTION
    SELECT * FROM Person.Person
   WHERE ModifiedDate = '20120208' WITH (UPDLOCK)
    -- Switch to session 1
   -- ...

   -- This statement will cause a deadlock!
   UPDATE Person.Person
   SET FirstName = '...'
   WHERE ModifiedDate = '20120208'

ROLLBACK
GO
```

warten und dadurch den Deadlock verursachen. Trotzdem lässt sich ein solcher Deadlock vermeiden, denn ein Intra-Parallelism Deadlock tritt ausschließlich in parallelen Ausführungsplänen auf. Sorgen Sie also dafür, dass solche Abfragen seriell mit nur einem Worker-Thread ausgeführt werden. Das klappt auf zweierlei Arten:

- Anpassen der Indizierungsstrategie, damit sichergestellt werden kann, dass sich die Gesamtkosten des Ausführungsplanes unterhalb des Cost Threshold for Parallelism [4] bewegen.
- Verwenden des Query Hints MAXDOP 1, da Sie dadurch einen seriellen Ausführungsplan erzwingen.

Read Committed Snapshot Isolation

In der vorigen Ausgabe der dotnetpro haben Sie Optimistic Concurrency und die beiden neuen Transaction Isolation Level *Read Committed Snapshot Isolation* und *Snapshot Isolation* kennengelernt [2].

Die Grundidee hinter Optimistic Concurrency ist, dass lesende Vorgänge – also *SELECT*-Statements – keine Shared Locks mehr anfordern. Und dieses Verhalten kann Ihnen ebenfalls beim Eliminieren von Deadlocks helfen.

Wie Sie auf den vorangegangenen Seiten gelesen haben, treten Deadlocks sehr oft zwischen lesenden und schreibenden Transaktionen auf. Wenn Sie bei lesenden Transaktionen die Shared Locks eliminieren, verhindern Sie zugleich die Deadlocks zwischen diesen beiden Datenzugriffsarten.

Registrieren Sie daher sehr viele Deadlocks zwischen lesenden und schreibenden Transaktionen und haben auch keine direkte Möglichkeit, diese zu beheben, schalten Sie einfach mal *Read Committed Snapshot Isolation* für die betreffende Datenbank ein – das kann Wunder bewirken.

Fazit

Die wichtigste Nachricht dieses Artikels: Der SQL Server kann über seinen Deadlock Monitor Blockaden selbstständig erkennen und auflösen. Generell gilt, dass Sie möglichst immer in derselben Reihenfolge auf Tabellen und Daten zugreifen sollten. Dies verhindert viele Deadlocks bereits im Vorfeld. Kann diese Regel nicht befolgt werden, kann auch das Aktivieren von *Read Committed Snapshot Isolation* sehr viele Deadlocks beseitigen.

[1] Klaus Aschenbrenner, SQL Server Concurrency, Teil 1, Pessimistic Concurrency, dotnetpro 2/2017, Seite 84ff., www.dotnetpro.de/A1702SQLServer

[2] Klaus Aschenbrenner, SQL Server Concurrency, Teil 2, Optimistic Concurrency, dotnetpro 3/2017, Seite 80ff., www.dotnetpro.de/A1703SQLServer

[3] Klaus Aschenbrenner, Nachschlagen mit Köpfchen, dotnetpro 3/2015, Seite 94ff., www.dotnetpro.de/A1503TSQL

[4] Cost Threshold for Parallelism, www.dotnetpro.de/SL1704SQLServer1

Klaus Aschenbrenner
berät Unternehmen in Europa beim Einsatz des Microsoft SQL Server und beschäftigt sich mit Windows-Programmierung und .NET. Zweimal zeichnete ihn Microsoft für sein Engagement als Microsoft MVP aus. Er ist zu erreichen über **www.SQLpassion.at.**

dnpCode A1704SQLServer