Einer für alle, alle für einen – verteilte Transaktionen

Transaktionen sind aus der Computerwelt nicht wegzudenken. Auch unser tägliches Leben besteht aus einer Vielzahl von Transaktionen, die wir nicht bewusst als solche wahrnehmen. Diese Artikelserie gibt eine umfassende praktische Einführung in die transaktionale Datenverarbeitung. Im ersten Teil stehen die Grundlagen und der SQL Server im Vordergrund.

___ Am Beginn eines Artikels über Transaktionen steht für gewöhnlich das klassische Zwei-Konten-Beispiel.

An einer Überweisung von einer Bank zur anderen sind mindestens zwei Computer beteiligt. Ein Computer subtrahiert Geld von einem Konto und der andere addiert Geld auf ein zweites Konto. Diese beiden Aktionen müssen koordiniert werden. Es wäre doch sehr schade, wenn das Geld nur von einem Konto abgebucht, aber nie auf dem anderen Konto ankommen würde. Deshalb laufen beide Aktionen in einer Transaktion ab, die sicherstellt, dass entweder beide Aktionen durchgeführt werden oder keine.

Aber es gibt auch andere Beispiele für eine Transaktion, beispielsweise eine Heirat. Die Braut und der Bräutigam sind Ressourcen-Manager, die sich selber verwalten, und der Pfarrer ist der Transaktionskoordinator. Erst wenn beide Ressourcen-Manager (Braut und Bräutigam) ja sagen, kann der Transaktionskoordinator (der Pfarrer) die Transaktion (die Heirat) bestätigen (*COMMIT*) und die Heiratsurkunde wird ausgefertigt.

Hier werden die Probleme einer verteilten Transaktion deutlich. Stellen Sie sich eine Fernhochzeit vor, bei der Braut und Bräutigam an zwei verschiedenen Orten sind. Es werden zwei Transaktionskoordinatoren (Pfarrer) benötigt, die sich untereinander abstimmen, vielleicht per Telefon. Die Braut sagt ja und der Pfarrer der Braut teilt dies dem Pfarrer des Bräutigams mit. Nun fragt der Pfarrer den Bräutigam ... aber leider, bevor der Pfarrer des Bräutigams dem Pfarrer der Braut die Antwort des Bräutigams mitteilen kann, bricht die Telefonverbindung zusammen. Beide haben zwar ja gesagt, aber sind sie jetzt verheiratet? Ist die Transaktion Hochzeit abgeschlossen? Oder muss ein Rollback durchgeführt werden?

In der Computerwelt treten ähnliche Probleme auf. Meist betreffen Transaktionen nur ein Datenbank-Management-System (DBMS). Dieses ist für die korrekte Verarbeitung der Transaktion verantwortlich. Es gibt allerdings viele Situationen, in denen nicht nur ein DBMS beteiligt ist; Transaktionen können über einen Oracle Server und einen SQL Server verteilt sein. Es kann ein anderes System wie MS Message Queue Server oder das Dateisystem beteiligt sein.

Was ist nun eine Transaktion? Wie in der Schule gibt es einen Merksatz: „Eine Transaktion ist eine Menge von Aktionen, die ein System von einem konsistenten Zustand in einem anderen konsistenten Zustand transformieren.“ Was ein konsistenter Zustand ist, wird dabei von der Anwendungslogik definiert.

ACID

Hinter dem Begriff verbirgt sich keine Säure, sondern er steht für die vier Anfangsbuchstaben der Grundbedingungen einer Transaktion auf Englisch.

„A“ steht für Atomicity oder Unteilbarkeit. Das bedeutet, dass entweder alle Aktionen durchgeführt werden oder keine. Schlägt eine Aktion fehl, wird für alle Aktionen ein Rollback durchgeführt. Wurde Geld von einem Konto abgebucht, ohne einem anderen Konto hinzugefügt zu werden, wird die Abbuchung rückgängig gemacht.

„C“ steht für Consistency. Durch eine Transaktion wird das System von einem konsistenten Zustand in einen anderen transformiert, wobei Konsistenz durch die Logik der Anwendung festgelegt ist. In Abbildung 1 wird ein Element in eine verkettete Liste eingefügt, in der jedes Element seinen Nachfolger kennt. Dazu ist es notwendig, beim vorletzten und beim einzufügenden Element die Adresse des Nachfolgers umzusetzen. Die Liste ist nur konsistent, wenn beide Adressen umgesetzt wurden.

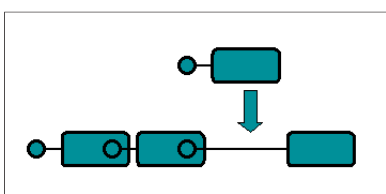

Abbildung 1 | Einfügen eines Elements in eine verkettete Liste.

SUMMARY

Auf einen Blick
Diese dreiteilige Artikelserie gibt eine umfassende praktische Einführung in die transaktionale Datenverarbeitung von der Theorie und dem SQL Server bis hin zu den .NET Enterprise Services und dem Distributed Transaction Coordinator.

Eingesetzte Anwendungen
.NET Framework, SQL Server, Windows 2000 oder Windows XP

Serie
Teil 1/3

Autor
Marcel Gnoth ist Senior Consultant bei der Berliner NTeam GmbH. Seine Arbeitsschwerpunkte liegen im Bereich COM, .NET und verteilte Informationssysteme, zu denen er auch Trainings durchführt. Sie erreichen ihn unter marcel@gnoth.net oder über www.gnoth.net.

Das „I“ wird uns im Folgenden noch ausführlicher beschäftigen. Es steht für Isolation, die Isolation der einzelnen Transaktionen untereinander. Auf einem DBMS arbeiten mehrere Benutzer gleichzeitig, deren Aktionen sich nicht gegenseitig beeinflussen dürfen. Im Idealfall ist es so, als arbeiteten alle Anwender nacheinander in der Datenbank.

Als Beispiel soll die verkettete Liste dienen: Anwender A möchte in einer Transaktion ein Element in die Liste einfügen und Anwender B möchte in einer Transaktion die Liste durchsuchen. Das Einfügen besteht aus zwei Aktionen. Es wäre sehr ungünstig, wenn Anwender B anfängt, die Liste zu durchsuchen, nachdem Anwender A die erste Aktion durchgeführt hat. Beim einzufügenden Element ist die Nachfolgeradresse noch nicht gesetzt. Also wird beim Durchsuchen das letzte Element nicht gefunden, das Ergebnis ist inkonsistent. Entweder B durchsucht die Liste vor oder nach dem Einfügen. B erhält zwar zwei verschiedene, aber jeweils konsistente Ergebnisse.

„D“ wie Durability bedeutet, dass bestätigte Änderungen permanent sind, auch wenn ein Computer oder Netzwerk ausfällt. Das heißt, im Normalfall werden die Ergebnisse auf der Festplatte gespeichert.

Transaktionen und der SQL Server

Der SQL Server ist ein Datenbank-Management-System und kann Änderungen an den von ihm verwalteten Daten transaktional durchführen. Dazu gehören verschiedene Transaktionsmodi, Sperrmechanismen für die Isolation der Transaktionen und Protokollfunktionen. Drei Transaktionsmodi stehen zur Verfügung. Standardmäßig arbeitet der SQL Server im Modus *AutoCommit*. Jede einzelne SQL-Anweisung wird in einer eigenen Transaktion ausgeführt.

Sollen mehrere Anweisungen in einer Transaktion durchgeführt werden, muss der Modus auf *Explicit* oder *Implicit* umgeschaltet werden. *begin transaction* startet eine explizite Transaktion. Alle folgenden Anweisungen sind Bestandteil dieser Transaktion, bis eine *commit-* oder eine *rollback*-Anweisung erreicht wird. In Listing 1 finden Sie eine explizite Transaktion. Diese Transact-SQL-Anweisungen (TSQL) können Sie im Query-Analyzer (*iSQLw.exe*) ausführen (siehe Abbildung 2). Sie beziehen sich auf

Listing 1 Eine explizite Transaktion.

```
BEGIN TRANSACTION
–Table Authors
INSERT INTO [pubs].[dbo].[authors]
([au_id], [au_lname], [au_fname], [phone], [address], [city], [state],
[zip], [contract])
VALUES
(‘111-22-3337’, ‘Pokemon’, ‘Gengar’, ‘123456’, ‘Spassgasse 7’, ‘Eden City’,
‘B’, ‘12345’, 1)
–Table Titles
INSERT INTO [pubs].[dbo].[titles]
([title_id], [title], [type], [pub_id], [price], [advance], [royalty],
[ytd_sales], [notes], [pubdate])
VALUES
(‘MG010’, ‘Japanisch Kochen für Mutige’, ‘mod_cook’,1389, 10.44, 5000.0000,
10, 5000, ‘’, 1991-06-07)
–Table TitleAuthors
INSERT INTO [pubs].[dbo].[titleauthor]([au_id], [title_id], [au_ord],
[royaltyper])
VALUES(‘111-22-3335’, ‘MG010’, 2, 40)
COMMIT
```

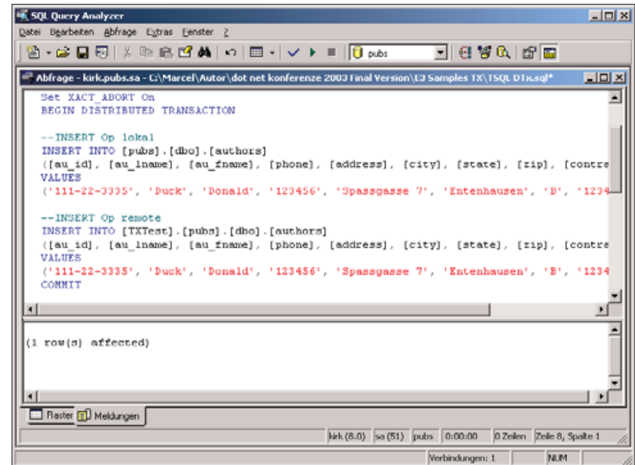

Abbildung 2 | Der iSQLw, das Werkzeug für SQL-Anweisungen, die zum SQL Server geschickt werden.

die beim SQL Server mitgelieferte Beispieldatenbank *Pubs*. Die Anweisung *set implicit _transactions on* startet den impliziten Transaktionsmodus. Alle folgenden Anweisungen laufen in der gleichen Transaktion, bis *commit* oder *rollback* aufgerufen wird. Die darauf folgende Anweisung startet automatisch (implizit) eine neue Transaktion, die dann wiederum bis zu einem *commit* oder *rollback* läuft. Dieser Modus ist wohl eher für Schreibfaule gedacht. Wer mit Transaktionen auf einem Datenbankserver arbeitet, sollte sie bewusst starten und beenden.

Der Mehrbenutzerzugriff und Probleme mit der Parallelität

Auf einem realen Datenbanksystem arbeiten viele Anwender gleichzeitig. Zusammenhängende Aktionen eines Benutzers werden in einer Transaktion zusammengefasst. Zu diesen Aktionen zählen *select*, *insert*, *update* und *delete*. Die Herausforderung besteht nun darin, dass sich die einzelnen Benutzer mit ihren Transaktionen nicht gegenseitig in die Quere kommen. Im Folgenden werden die dabei auftretenden Probleme untersucht, wobei deutlich wird, dass immer ein Kompromiss zwischen Isolation und Durchsatz gefunden werden muss.

Lost Update

Beim *Lost Update*, auch *Verlorene Aktualisierungen* genannt, lesen zwei Anwender dieselben Daten und ändern diese. Der erste Anwender speichert seine Änderungen und überschreibt damit die Originalwerte in der Datenbank. Wenn jetzt der zweite Anwender seine Änderungen ebenfalls speichert, überschreibt er die Änderungen des ersten Anwenders, ohne zu wissen, dass schon jemand vor ihm die Datensätze aktualisiert hat. Hätte er von den Änderungen seines Kollegen gewusst, würde er diese eventuell nicht überschreiben. In den Fällen, in denen das Prinzip „der Letzte gewinnt“ akzeptabel ist, muss verhindert werden, dass ein Anwender unwissend die Änderungen eines anderen überschreibt. Tabelle 1 zeigt ein solches Szenario. Anwender A liest den Kontostand und subtrahiert 100 Euro. Bevor Anwender A den neuen Kontostand in die Datenbank schreiben kann, liest Anwender B ebenfalls den Kontostand, subtrahiert 20 Euro und schreibt den Kontostand. Danach kommt A endlich dazu, seine Änderung in die Datenbank zu schreiben, und über-

Tabelle 1 Lost-Update-Problem – zwei Anwender ändern dieselben Daten.

Anwender A	Anwender B
Begin Transaktion 1 read (Kontostand) Kontostand += 100	
	Begin Transaktion 2 read (Kontostand) Kontostand –= 20 write (Kontostand) Commit Transaktion 2
write (Kontostand) Commit Transaktion 1	

schreibt damit die Subtraktion von B. Beide Transaktionen wurden erfolgreich durchgeführt, aber auf dem Konto sind jetzt 20 Euro zu viel. Ob das konsistent ist?

Das *Lost-Update*-Problem muss besonders in der Welt des unverbundenen *DataSet* berücksichtigt werden. Viele Anwender erhalten Daten aus der Datenbank und ändern diese lokal auf ihrem PC, ohne Informationen über Änderungen anderer Anwender zu haben. Der *DataAdapter* verfügt aber über einen Mechanismus, um beim Zurückschreiben der Änderungen des *DataSet* in die Datenbank mit dem *Lost-Update*-Problem umzugehen, siehe auch [1].

Dirty Read

Das Dirty Read wird im Deutschen oft als Lesen unbestätigter Daten bezeichnet. Zwei Transaktionen laufen parallel, Transaktion A ändert Daten. Die zweite Transaktion B liest diese Änderungen, bevor diese mit einem *Commit* von Transaktion A bestätigt sind. Wenn jetzt Transaktion A ein *Rollback* durchführt, dann arbeitet Transaktion B mit Daten, die nicht mehr in der Datenbank stehen (siehe Tabelle 2). Angenommen, der Kontostand beträgt 3000 Euro. Transaktion 1 setzt den Kontostand auf 5000 Euro und schreibt diesen Wert in die Datenbank. In der Zwischenzeit liest Transaktion 2 diesen vorläufigen Kontostand und subtrahiert davon 20 Euro. Transaktion 1 führt jetzt ein Rollback durch, sodass in der Datenbank der ursprüngliche Wert von 3000 Euro wiederhergestellt wird. Jetzt schreibt Transaktion 2 die errechneten 4980 Euro in die Datenbank. Ein konsistentes System?

Tabelle 2 Dirty-Read-Phänomen.

Anwender A	Anwender B
Begin Transaktion 1 Kontostand = 5000 write (Kontostand)	
	Begin Transaktion 2 read (Kontostand) Kontostand –= 20
Rollback Transaction 1	
	write (Kontostand) Commit Transaction 2

Nonrepeatable Reads

Diese Situation ähnelt den Dirty Reads und wird auch als „Nicht wiederholbare Lesevorgänge" bezeichnet. Eine Transaktion liest zweimal einen Datensatz und erhält dabei unterschiedliche Werte. Das kann passieren, weil in der Zwischenzeit eine andere Transaktion den Datensatz verändert und diese Änderung mit einem *Commit* bestätigt hat (siehe Tabelle 3).

Tabelle 3 Nonrepeatable Reads.

Anwender A	Anwender B
Begin Transaktion 1 read (Kontostand)	
	Begin Transaktion 2 read (Kontostand) Kontostand–= 20 write (Kontostand) Commit Transaktion 2
read (Kontostand)	

Phantome

Diese Phantome haben nichts mit Opernhäusern zu tun, es handelt sich vielmehr um Datensätze, die plötzlich auftauchen oder verschwinden. In Tabelle 4 führt Transaktion 1 ein *select* aus und erhält eine Ergebnismenge von zehn Datensätzen. In der Zwischenzeit ändert Transaktion 2 den Kontostand eines Datensatzes auf 80 Euro und bestätigt diese Änderung. Führt jetzt Transaktion 1 erneut dasselbe *select* aus, enthält die Ergebnismenge nur noch neun Datensätze, einer ist verloren gegangen. Das gleiche Problem tritt auf, wenn Transaktion 2 einen der selektierten Datensätze löscht.

Tabelle 4 Phantome, Datensätze gehen verloren.

Anwender A	Anwender B
Begin Transaktion 1 Select * From Konto Where Kontostand > 100	
	Begin Transaktion 2 read (Kontostand) Kontostand = 80 write (Kontostand) Commit Transaktion 2
Select * From Konto Where Kontostand > 100	

Sperren und Isolationsstufen

Zur Behandlung dieser Probleme verfügen DBMS über Sperrmechanismen. Benötigt eine Transaktion Datensätze, so werden diese für andere Anwender und Transaktionen gesperrt, wobei zwischen Lese- und Schreibsperren unterschieden wird. Das kann bedeuten, dass andere Transaktionen warten müssen, bis die sperrende Transaktion beendet ist. Je sicherer beziehungsweise konsistenter eine Transaktion ausgeführt werden soll, umso mehr Sperren sind erforderlich. Je mehr Sperren existieren, umso mehr behindern sich die Transaktionen gegenseitig. Schon wenige Benutzer können mit komplizierten Transaktionen einen Datenbankserver außer Gefecht setzen.

Es gilt einen Kompromiss zwischen Isolation und Konsistenz auf der einen Seite und Performanz und Skalierbarkeit auf der anderen Seite zu finden.

Der SQL Server und COM+ verfügen über vier Isolationsstufen, die festlegen, wie Datensätze gesperrt werden (Tabelle 5). Die TSQL-Anweisung *set transaction isolation level* legt die Isola-

Tabelle 5 Die vier Isolationsstufen des SQL Servers.

Isolationsstufe	Dirty Read	Nonrepeatable Read	Phantom
Read Uncommitted	Ja	Ja	Ja
Read Committed	Nein	Ja	Ja
Repeatable Read	Nein	Nein	Ja
Serializable	Nein	Nein	Nein

tions-Eigenschaft einer Transaktion fest. *Serializable* bietet den höchsten Schutz vor Inkonsistenzen, benötigt aber die meisten Sperren. In der Praxis wird häufig *Read Committed* eingesetzt.

Verteilte Transaktionen mit dem SQL Server

Ein System, das den Zugriff auf seine Daten über Transaktionen erlaubt, wird als Ressourcen-Manager bezeichnet. Für die meisten Anwendungen in der Geschäftswelt ist der Zugriff auf einen Datenbankserver beziehungsweise Ressourcen-Manager ausreichend. Was aber, wenn es notwendig wird, Datensätze in zwei verschiedene Datenbankserver einzufügen? Dazu müssten zwei Transaktionen gestartet werden, eine für jeden Ressourcen-Manager. Beide Transaktionen könnten direkt nacheinander mit *commit* bestätigt werden. Wie helfen Sie sich aber, wenn der unwahrscheinliche Fall eintritt, dass nach dem *commit* der ersten Transaktion ein Computer- oder Netzwerkausfall eintritt? Nach Murphy's Law, wird alles, was schief gehen kann, irgendwann schief gehen. Also ist die Lösung mit zwei getrennten Transaktionen unbefriedigend. Schöner wäre es, wenn die Änderungen auf beiden Ressourcen-Managern in einer *Verteilten Transaktion* durchgeführt werden könnten. Wie soll das realisiert werden, wo doch jeder Ressourcen-Manager nur seine eigenen Daten verwalten kann?

Die Lösung in der Windows-Welt heißt *Distributed Transaction Coordinator* (DTC). Der DTC wurde erstmalig mit dem SQL Server 6.5 ausgeliefert und ist inzwischen Bestandteil von Windows und COM+. Er läuft als Dienst und hat die Aufgabe, die Zusammenarbeit mehrerer Ressourcen-Manager zu koordinieren. Ausführlicher wird er im zweiten Teil der Artikelserie beschrieben.

Linked Server

Der SQL Server bietet die Möglichkeit, andere Datenquellen als *Verbundene Server* oder *Linked Server* einzubinden. Um eine solche Verbindung einzurichten, kann der Enterprise Manager des SQL Servers benutzt werden. Wer jedoch ein echter SQLer ist, wird TSQL-Anweisungen bevorzugen. Listing 2 zeigt die Anweisungen, die nötig sind, um einen Verbundenen Server einzurichten und wieder zu löschen. SPOCK ist dabei der Name des zu verbindenden Datenbankservers. Nachdem der Server angelegt wurde, muss ein Login festgelegt werden. Im Enterprise Manager ist ein Linked Server unter *Sicherheit->Verbindungsserver* zu finden. (siehe auch *Zentraler Zugriff auf verteilte Daten: Linked Server*, Seite 90 ff.)

Linked Server können jetzt in TSQL-Anweisungen benutzt werden. *begin distributed transaction* oder *begin transaction* starten eine Transaktion. Die Anweisungen laufen so lange als lokale Transaktion, wie sie nur einen Ressourcen-Manager betreffen. Sobald innerhalb der Transaktion auf den Linked Server zugegriffen wird, nimmt der Ressourcen-Manager Kontakt mit dem lokalen DTC auf, der versucht, die Transaktion zu einer verteilten

Listing 2 Einen Verbunden Server einrichten.

```
–Linked Server anlegen
EXEC sp_addlinkedserver
      @server='TXTest', provider='SQLOLEDB', @datasrc='SPOCK'
–Login für Linked Server einrichten
EXEC sp_addlinkedsrvlogin 'TXTest', 'false', NULL, 'sa', ''
–Linked Server löschen
sp_dropserver 'TXTest', 'droplogins'
```

Listing 3 Eine verteilte Transaktion zwischen zwei SQL Servern.

```
Set XACT_ABORT On
BEGIN DISTRIBUTED TRANSACTION
–INSERT Op lokal
INSERT INTO [pubs].[dbo].[authors]
([au_id], [au_lname], [au_fname], [phone], [address], [city], [state],
[zip], [contract])
VALUES
('111-22-3334', 'Duck', 'Donald', '123456', 'Spassgasse 7', 'Entenhausen',
'B', '12345', 1)

–INSERT Op remote
INSERT INTO [TXTest].[pubs].[dbo].[authors]
([au_id], [au_lname], [au_fname], [phone], [address], [city], [state],
[zip], [contract])
VALUES
('111-22-3334', 'Duck', 'Donald', '123456', 'Spassgasse 7', 'Entenhausen',
'B', '12345', 1)
COMMIT
```

Transaktion hochzustufen. Dazu nimmt der lokale DTC Kontakt mit dem DTC des zweiten Computers auf. Wenn der Ressourcen-Manager des verbundenen Servers mit dem DTC zusammenarbeitet, wird aus der einfachen Transaktion eine verteilte Transaktion, die vom DTC zwischen den beiden Ressourcen-Managern koordiniert wird. Listing 3 zeigt ein Beispiel, in dem eine Transaktion zwischen zwei SQL Servern gesteuert wird. Für dieses Beispiel muss die Option *xact abort* eingeschaltet werden. Normalerweise wird bei einem Fehler nur für die Anweisung einer Transaktion, die den Fehler verursacht hat, ein Rollback durchgeführt; die Transaktion läuft weiter. Ist diese Option eingeschaltet, erfolgt im Fehlerfall für die gesamte Transaktion ein Rollback. Unterstützt einer der Ressourcen-Manager keine verschachtelten Transaktionen, muss diese Option eingeschaltet sein. Nähere Informationen entnehmen Sie bitte der SQL-Server-Online-Dokumentation.

Die erste *insert*-Anweisung fügt einen Datensatz in die lokale *Pubs*-Datenbank ein. Die zweite Anweisung fügt den Datensatz in die *Pubs*-Datenbank des verbundenen Servers *TXTest* ein. Das *commit* gilt für beide Aktionen. Sie finden innerhalb einer logischen Transaktion auf zwei getrennten SQL Servern statt.

Ausblick

Nicht immer reichen die Bordmittel des SQL Servers aus, und TSQL zu programmieren ist auch nicht jedermanns Sache. Im nächsten Teil der Serie geht es um Alternativen und die .NET Enterprise Services. Diese ermöglichen, mit COM+-Komponenten verteilte Transaktionen zu programmieren. Außerdem wird der DTC genauer unter die Lupe genommen.

[1] Datasets: lokal bearbeitet, zentral gespeichert, Marcel Gnoth, dotnetpro 1/2003, Seite 102 ff.