Maßlos?

InterBase-Transaktionen verstehen und nutzen – Teil 1

von Karsten Strobel

Was ist eigentlich eine Transaktion? Der Duden gibt folgende Auskunft: „*Trans|ak|ti|on [...zion] die, (das ein normales Maß überschreitende finanzielle Geschäft)*. Der Transaktionszähler meiner Adressdatenbank steht zurzeit bei knapp über 17.000, und ich bin immer noch nicht reich. Ein Programmfehler? Auf jeden Fall Anlass genug, um einmal genauer zu beleuchten, was InterBase unter einer Transaktion versteht...

Ein altbewährtes Musterbeispiel für die Anwendung von Transaktionen ist die doppelte Buchführung. Diese verlangt, dass jede Buchung – sagen wir beispielsweise für das Bezahlen eines guten Fachbuches – *doppelt*, also zweifach niedergeschrieben wird, nämlich auf den beiden Konten, die der Vorgang betrifft, einmal im Soll und einmal im Haben. In unserem Fall wollen wir natürlich keine staubigen Kontenbücher, sondern eine InterBase-Datenbank zum Niederschreiben der Buchungen verwenden. Zur Veranschaulichung der Vorgänge definieren wir folgende Tabellen:

```
create table konten (
 kontonr integer not null primary key,
 bezeichnung varchar(50)
);
create table buchungen (
 kontonr integer not null references konten (kontonr),
 seite char(1) check (seite in ('S','H')),
 betrag numeric(9,2),
 bemerkung varchar(50)
);

insert into konten values (1600, 'Kasse');
insert into konten values (6820, 'Fachliteratur');
```

So könnten die Tabellen aussehen, in denen die Konten und Buchungssätze erfasst werden. Abbildung 1 stellt diese in sogenannten T-Konten dar, und zwar mit *einer* Buchung, die auf beiden Konten erscheinen muss (Doppelte Buchführung).

Buchhalter-Albträume

Der Horror eines guten und somit peniblen Buchhalters ist natürlich, dass seine Bücher durch Buchungsfehler durcheinander geraten. Dem Buchhalter, wie er wohl vor 50 Jahren zu Werke ging, mit Schirmmütze und Ärmelschonern bewaffnet, konnte sich vielleicht bei der ersten Eintragung die Hand verstauchen und darüber die zweite Eintragung vergessen. Dann wäre seine Buchung unvollständig und seine Konten unstimmig – für einen Buchhalter eine Katastrophe. Werden die Buchungsdaten in einer Datenbank gespeichert, dann entspricht die verstauchte Hand vielleicht einem Programmierfehler in dem Anwendungsprogramm, das die zwei *Insert*-Befehle an die Datenbank übergibt und der zwischen dem ersten und dem zweiten *Insert* zum Abbruch des Programms führt. Mithilfe der Transaktionssteuerung kann man zwar nicht das Auftreten des Fehlers, aber dessen Konsequenzen – nämlich die Inkonsistenz der Daten – vermeiden.

Wozu Transaktionen?

Einer Datenbank, die keine Transaktionen unterstützt, fehlt ein wichtiges Mittel, um Datenkonsistenz bei Mehrbenutzerbetrieb garantieren zu können. Eine Transaktion bietet den Kontext für den Zugriff auf die Datenbank und erlaubt es, einen oder mehrere Befehle, die eine logisch zusammenhängende Arbeitsmenge bilden, als eine zusammenhängende und nicht teilbare Datenbankaktion anzusehen.

Anfang und Ende einer Transaktion kann man als Programmierer selbst bestimmen. Wenn die beiden *Insert*-Befehle, mit denen die beiden Buchungen in die Datenbank geschrieben werden, zu einer

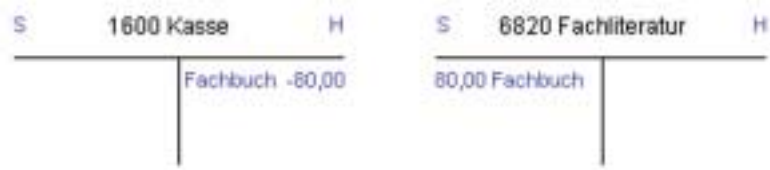

Abb. 1: Doppelte Buchführung

Transaktion zusammengefasst werden, dann werden am Ende der Transaktion entweder beide Datensätze in der Datenbank eingetragen sein, oder – falls irgendein Fehler dies verhindert – wird keiner der beiden Datensätze in der Datenbank stehen. Beide oder keiner – aber nie einer, nie nur ein Teil der (nicht teilbaren) Transaktionsarbeit.

Pflichtübung

Das Bestehen einer Transaktion ist übrigens für die allermeisten Zugriffe auf die Datenbank obligatorisch. Für das Lesen und Schreiben von Daten gilt das grundsätzlich. Man sagt: Der Zugriff auf die Daten erfolgt *im Kontext* einer Transaktion.

Gemeint ist, dass für den Datenzugriff gewisse Spielregeln gelten, die der Datenbankbenutzer (der Client) beim Definieren der Transaktion vor dem eigentlichen Datenzugriff festgelegt haben muss. Bei diesen Spielregeln – man könnte auch sagen: Aspekte einer Transaktion – geht es um:

- Zusammenfassen von einem oder mehreren Zugriffen zu einer Einheit
- Sichtbarkeit von Änderungen anderer Transaktionen
- Behandlung von Konflikten bei Schreibzugriffen
- Exklusivität des Datenzugriffs

Eine Transaktion kann wenige Millisekunden oder auch einige Stunden dauern. Vielleicht sind Sie schon einmal über Benchmark-Werte für Server-Computer gestolpert, die in „Transaktionen pro Minute“ (tpm) o.ä. angegeben wurden. Nach dem vorher Gesagten ist klar, dass solchen Messwerten eine genaue Definition von Art und Umfang dieser Transaktionen zugrunde liegen muss, da die Werte sonst nicht vergleichbar wären. Daher werden solche Definitionen für Datenbank-Benchmarks vom TPC (Transaktion Processing Performance Council), einem nicht-kommerziellen Zusammenschluss von Hard- und Softwareherstellern, vorgenommen, um vergleichbare Maßstäbe für die Leistungsmessung von Datenbankservern anzubieten (siehe www.tpc.org).

Um auf eine InterBase-Datenbank zuzugreifen, muss das Client-Programm zunächst zwei Schritte vollziehen: Im ersten muss es sich mit der Datenbank verbinden. Das ist mit dem Wählen einer Telefonnummer beim Telefonieren vergleichbar. Erst wenn die Verbindung zustande gekommen ist, kann das Gespräch (der Zugriff auf die Daten) losgehen. Und dafür braucht man immer eine Transaktion. Wenn Sie schon mit Hilfe einer Client-Bibliothek wie BDE, IBObjects, IBX o.ä. von Delphi oder C++ auf InterBase zugegriffen haben, dann haben Sie Transaktionen vielleicht als eine Option verstanden. Hier braucht man eine Transaktion nämlich nur dann explizit zu starten (z.B. mit *TDatabase.StartTransaction*) und zu beenden (mit *TDatabase.Commit* oder *.Rollback*), wenn man die Transaktionsdauer aus einem besonderen Grund selbst kontrollieren will, etwa um mehrere SQL-Befehle zusammenzufassen. Tatsächlich gibt es die Transaktion aber auch bei dieser Form der Programmierung immer, die Client-Bibliothek (BDE, IBO...) kümmert sich nämlich selbst darum, eine Transaktion zu starten, wenn dies nicht explizit geschieht. Klarheit darüber erhält man, wenn man einen Blick in die Dokumentation der InterBase-API (API Guide) wirft. Bei jedem Zugriff auf Daten (z.B. mit den API-Befehlen *isc_dsql_prepare()*, *isc_dsql_execute()* etc.) muss der Client ein Transaktionshandle übergeben, das er vorher mit *isc_start_transaction()* angefordert haben muss. Im Klartext: Auch wenn Sie es als Delphi-Programmierer nicht gewöhnt sind, für jeden Zugriff auf die Datenbank eine Transaktion zu definieren – es geschieht doch.

Aber zurück zu unserem Beispiel mit der Buchhaltung: Die verschiedenen Aspekte der Wirkung der Transaktionsmodi möchte ich im Folgenden anhand von Beispielen illustrieren. Sie können – und sollten möglichst – diese Beispiele nachvollziehen und selbst experimentieren. Verwenden Sie dazu bitte entweder die Anwendung *WISQL32.exe* (aus InterBase V5, auch anwendbar auf V6) oder die Textmodusanwendung *ISQL.exe* (im *\Bin*-Verzeichnis Ihrer InterBase-Installation). Alle Beispiele funktionieren sowohl mit V5 als auch mit V6 (egal in welchem SQL-Dialekt). Sie sollten mit einer Test-Datenbank arbeiten, die Sie am besten extra für diesen Zweck anlegen. Erzeugen Sie die beiden Tabellen mit den angegebenen *Create*-Befehlen. Diese finden Sie auch auf der Profi-CD und im Web.

Wichtig für das Verständnis der Beispiele: *WISQL32.exe* und *ISQL.exe* starten unmittelbar nach dem Verbinden mit einer Datenbank eine Transaktion, die erst mit einem *Commit*- oder *Rollback*-Befehl beendet wird, woraufhin automatisch wieder eine neue Transaktion gestartet wird. Es ist bei diesen Programmen also praktisch immer eine Transaktion gestartet, solange man mit einer Datenbank verbunden ist. Dem Benutzer steht dabei je Anwendung nur eine Transaktion zur Verfügung, nicht mehrere, was zwar theoretisch durchaus möglich ist, praktisch aber von (W)ISQL nicht unterstützt wird.

Die Würfel sind gefallen

Der erste Aspekt einer Transaktion ist – wie gesagt – die Zusammenfassung von (meist mehreren) Operationen zu einer logischen Einheit. Wenn wir die zwei *Insert*-Befehle im Kontext einer Transaktion ausführen und anschließend die gemachten Änderungen mit *Commit* bestätigen, dann sind anschließend beide Datensätze in der Tabelle „buchungen“ gespeichert. Wird statt dem *Commit* ein *Rollback*-Befehl gegeben, dann werden beide Änderungen zurückgenommen und keiner der beiden Datensätze erscheint in der Tabelle „buchungen“. Das Motto lautet also: Alles oder nichts – halbe Sachen zählen nicht. Unser Experiment 1:

```
delete from buchungen;
commit;
select count(*) from buchungen;
<Ergebnis: 0>
insert into buchungen values
                        (1600, 'H', -80.00, 'Fachbuch');
select count(*) from buchungen;
<Ergebnis: 1>
insert into buchungen values (6820, 'S', 80.00, 'Fachbuch');
select count(*) from buchungen;
<Ergebnis: 2>
rollback;
select count(*) from buchungen;
<Ergebnis: 0>
```

Eine Anwendung sollte den Ablauf einer expliziten Transaktion immer so kontrollieren, dass am Ende der Transaktion

im Normalfall ein *Commit* und im Fehlerfall ein *Rollback* aufgerufen wird. Folgende Programmstruktur bietet sich in Delphi dafür an:

```
IBODatabase.StartTransaction;
try

 {... datenbank-operation ...}
 {... datenbank-operation ...}

 IBODatabase.Commit;
finally
 if IBODatabase.InTransaction then IBODatabase.Rollback;
end;
```

Eine solche Kontrollstruktur kann man immer einsetzen, wenn man in einer nichtinteraktiven Form auf die Datenbank zugreift.

Es kann natürlich auch passieren, dass eine Transaktion weder mit *Commit* noch mit *Rollback* abgeschlossen wird, etwa weil der Clientrechner sang- und klanglos abstürzt und nicht einmal der *finally*-Block mehr bearbeitet wird, oder weil plötzlich die Netzwerkverbindung zum Server gestört ist, wegen eines Stromausfalls oder Ähnlichem. Wenn der Datenbankserver feststellt, dass eine Clientverbindung abgebrochen wurde, seitens derer noch gestartete Transaktionen existieren, dann werden diese automatisch mit einem *Rollback* abgeschlossen. Sie können dieses Verhalten experimentell nachvollziehen: Fügen Sie einfach in den obigen Beispielcode für Delphi vor die Commit-Anweisung den Win32-API-Aufruf *TerminateProcess (GetCurrentProcess, 0);* ein. Das erzwingt ein „hartes" Beenden des laufenden Prozesses, bei dem nicht einmal mehr der *finally*-Block abgearbeitet wird. Das ist zwar nicht gerade die feine englische Art, aber sehr wirkungsvoll: Der Prozess wird beendet, ohne dass ein *Commit* oder *Rollback* auf die gestartete Transaktion ausgeführt werden kann und ohne dass die Datenbankverbindung geschlossen wird. Der Datenbankserver wird feststellen, dass die Clientverbindung plötzlich „gestorben" ist und wird daher auch die Transaktionen dieser Verbindung für tot erklären, was einem *Rollback* der noch offenen Transaktionen gleichkommt. Sie können sich danach wieder mit der Datenbank verbinden und überprüfen, dass die Datenbankoperationen des zwangsweise abgestürzten Delphi-Programms keine Spuren hinterlassen haben und die Datenänderungen nicht sichtbar sind.

Wann also werden Änderungen im Rahmen einer Transaktion sichtbar? Die Antwort: Für die ändernde Transaktion sofort, aber für alle anderen Transaktionen frühestens nachdem die ändernde Transaktion ein *Commit* ausgeführt hat.

Machen wir mal ein Experiment, um die Sichtbarkeit von Änderungen zu erforschen. Starten Sie dazu *ISQL.exe* (oder *WISQL32.exe*) in zwei Instanzen und arrangieren Sie die Fenster nebeneinander am Bildschirm, so dass Sie die Ausgaben beider Programminstanzen sehen und leicht zwischen beiden Fenstern hin- und her wechseln können. Verbinden Sie die beiden Instanzen mit Ihrer Testdatenbank.

Achten Sie auf die genaue Abfolge der im linken bzw. im rechten Fenster auszuführenden Anweisungen.

Alles wie erwartet? Beinahe. Wir haben zwei Sitzungen jeweils mit einer eigenen Transaktion verwendet. Links haben wir Datensätze in die Tabelle „buchungen" geschrieben und konnten die gemachten Änderungen dort dann auch sofort sehen; schließlich waren es ja Änderungen im Kontext derselben Transaktion. Rechts sehen wir diese Änderungen erst mal nicht. Zunächst muss die linke Seite die Änderungen mit *Commit* endgültig machen (Zeile 12 in Tabelle 1), d.h. die laufende Transaktion unter Bestätigung der gemachten Änderungen abschließen. Aber auch danach sieht die rechte Sitzung die Änderungen der linken Seite noch nicht (Zeile 14), obwohl sie auf der linken Seite schon mit *Commit* abgesegnet worden waren! Erst nachdem auch die rechte Seite ein *Commit* machte (Zeile 15) und dadurch die bisherige Transaktion beendete und eine neue startete, werden die Änderungen der linken Seite auch rechts sichtbar (Zeile 16). Warum?

Selbst-Isolation

Diese Frage führt uns direkt zum zweiten zu besprechenden Aspekt, nämlich der Sicht-

1	**delete from** buchungen;	
2	**commit;**	
3		**commit;**
4	**select count(*) from** buchungen; *<Ergebnis: 0>*	
5		**select count(*) from** buchungen; *<Ergebnis: 0>*
6	**insert into** buchungen values (1600, 'H', -80.00, 'Fachbuch');	
7	**select count(*) from** buchungen; *<Ergebnis: 1>*	
8		**select count(*) from** buchungen; *<Ergebnis: 0>*
9	**insert into** buchungen values (6820, 'S', 80.00, 'Fachbuch');	
10	**select count(*) from** buchungen; *<Ergebnis: 2>*	
11		**select count(*) from** buchungen; *<Ergebnis: 0>*
12	**commit;**	
13	**select count**(*) **from** buchungen; *<Ergebnis: 2>*	
14		**select count**(*) **from** buchungen; *<Ergebnis: 0>*
15		**commit;**
16		**select count**(*) **from** buchungen; *<Ergebnis: 2>*

Tabelle 1: Das Experiment 2

barkeit von Änderungen anderer Transaktionen, auch Isolation genannt. Eine Transaktion kann ihren Isolationsmodus und damit den Grad der Sichtbarkeit von Änderungen anderer Transaktionen selbst bestimmen. Der Isolationsmodus steuert also, inwieweit sich eine Transaktion selbst von neuen Datensatzversionen isoliert. Es geht dabei also *nicht* darum, andere Transaktionen zu isolieren, sondern darum, dass eine Transaktion sich selbst von fremden Änderungen abschottet.

Die üblichen Transaktionsmodi für eine InterBase-Anwendung sind *concurrency* oder *read-committed* mit *record_version*.

Read Committed bedeutet, dass eine in diesem Modus gestartete Transaktion alle Änderungen anderer, simultaner Transaktionen „sehen" kann, sofern diese anderen Transaktionen ihre Änderungen schon mit *Commit* permanent gemacht haben. *Read Committed* ist also der Modus, in dem eine Transaktion immer über „das Neueste" auf dem laufenden bleibt, und zwar auch dann, wenn die Neuigkeiten erst nach dem Start der eigenen Transaktion hereinkommen.

Eine Transaktion im *concurrency*-Modus dagegen behält immer die selbe Sicht der Daten, so wie sie in dem Augenblick, in dem die Transaktion gestartet wurde, aktuell waren. Daher wird dieser Modus auch als *snapshot* oder *repeatable read* bezeichnet. Solange die Transaktion läuft, liefert sie dem Anwender immer die selben Daten, als hätte man am Anfang einen „Schnappschuss" von der ganzen Datenbank gemacht.

In den meisten Fällen möchte man den gleichzeitigen Zugriff mehrerer Transaktionen auf die Datenbank gar nicht verhindern. Im Gegenteil: Man ist froh, wenn dies so konfliktfrei wie möglich geschehen kann. Daher kommen in der Praxis fast ausschließlich die Modi *read committed* und *concurrency* zur Anwendung. Zur Erinnerung: *read committed* erlaubt es der Transaktion, jederzeit die Änderungen anderer Transaktionen zu lesen, sofern diese schon „committed" sind. *Concurrency* hingegen liefert immer dasselbe Bild, abgesehen natürlich von den Änderungen, die diese Transaktion selbst an den Daten vornimmt.

Ganz egal, welchen Transaktionsmodus man wählt: Änderungen an den Daten sind für andere Transaktionen frühestens sichtbar, nachdem die Änderungen mittels *Commit* endgültig geworden sind. Andere Datenbankprodukte unterstützen einen Modus namens *Dirty read*, in dem auch noch nicht endgültige Änderungen sichtbar sind. InterBase kennt diesen Modus nicht.

Die Eigenheiten des *concurrency*-Modus sind auch der Grund dafür, warum im Experiment Nr. 2 auf der rechten Seite noch ein *Commit* (Zeile 15) notwendig ist, um die Änderungen der linken Seite sehen zu können (Zeile 16). Es könnte übrigens hier genauso gut ein *Rollback* sein, denn die Transaktion hat ja gar keine Änderungen vorgenommen und es geht nur darum, die gestartete Transaktion zu beenden und eine neue Transaktion zu starten. Der voreingestellte Isolationsmodus für Transaktionen bei (W)ISQL ist nämlich *concurrency* (alias *snapshot*).

Man kann mit dem Befehl *Set Transaction* bei (W)ISQL die Parameter der Transaktion selbst festlegen. Hier das Syntaxschema des Befehls:

Isolationsmodus	Bedeutung
read committed	Die Transaktion liest die Änderungen anderer simultaner Transaktionen, sofern diese abgeschlossen (committed) sind. Bei diesem Modus gibt es noch die Auswahl zwischen der Option *record_version* oder *no record_version*; Letztere verhindert den Zugriff auf Datensätze, die von simultanen Transaktionen gerade geändert werden.
Concurrency	Die Transaktion hat keinen Zugriff auf abgeschlossene Änderungen anderer simultaner Transaktionen. Dieser Modus wird auch *snapshot* oder *repeatable-read* genannt.
Consistency	Die Transaktion erhält keinen Zugriff auf Tabellen, die von simultanen Transaktionen beschrieben werden. Umgekehrt können simultane Transaktionen Tabellen nicht beschreiben, wenn diese von einer aktiven consistency-Transaktion verändert wurden. Ansonsten entspricht dieser Modus dem concurrency-Modus. Dieser Modus wird auch *snapshot table stability* genannt.

Tabelle 2: Isolationsmodi

1	**delete from** buchungen;	
2	**commit;**	
3		**set transaction read committed** record_version;
4	**select count(*) from** buchungen; *<Ergebnis: 0>*	
5		**select count(*) from** buchungen; *<Ergebnis: 0>*
6	**insert into** buchungen values (1600, 'S',80.00, 'Fachbuch')	
7	**select count(*) from** buchungen; *<Ergebnis: 1>*	
8		**select count(*) from** buchungen; *<Ergebnis: 0>*
9	**insert into** buchungen values (6820, 'H',80.00, 'Fachbuch')	
10	**select count(*) from** buchungen; *<Ergebnis: 2>*	
11		**select count(*) from** buchungen; *<Ergebnis: 0>*
12	commit;	
13	**select count(*) from** buchungen; *<Ergebnis: 2>*	
14		**select count(*) from** buchungen; *<Ergebnis: 2>*

Tabelle 3: Experiment 3

Abb. 2: Kontoauszug als Bericht

```
SET TRANSACTION
[READ WRITE | READ ONLY]
[WAIT | NO WAIT]
[[ISOLATION LEVEL] {SNAPSHOT [TABLE STABILITY]
| READ COMMITTED [[NO] RECORD_VERSION]}]
[RESERVING <reserving_clause>];

<reserving_clause> = table [, table ...]
[FOR [SHARED | PROTECTED] {READ | WRITE}]
                          [, <reserving_clause>]
```

Set Transaction erlaubt es in (W)ISQL, beinahe alle bei InterBase möglichen Transaktionsparameter einzustellen. Für die Isolationsmodi werden hierbei allerdings z.T. andere Begriffe verwendet. „snapshot" steht hier für *concurrency* und „snapshot table stability" steht für *consistency*.

Set Transaction startet übrigens eine neue Transaktion, denn für eine bereits gestartete Transaktion können die Parameter nicht geändert werden. Die bestehende Transaktion wird dazu mit *Rollback* beendet (bei WISQL) oder der Benutzer wird gefragt, ob auf die bisherige Transaktion *Commit* oder *Rollback* ausgeübt werden soll (bei ISQL). Wiederholen wir das letzte Experiment und wählen auf der rechten Seite diesmal den Isolationsmodus *read committed* (siehe Tabelle 3).

Im Gegensatz zum Experiment Nr. 2 war diesmal auf der rechten Seite kein *Commit* (und somit Starten einer neuen Transaktion) notwendig, denn der Isolationsmodus *read committed* bewirkt, dass für die rechte Transaktion die Änderungen der linken Transaktion nach deren Commit (Zeile 12 in Tabelle 3) sichtbar werden (Zeile 14).

Der *read-committed*-Isolationsmodus wird häufig für interaktive Anwendungen verwendet, besonders dann, wenn über einen längeren Zeitraum eine Transaktion gestartet bleibt. Der Benutzer könnte das Fenster, in dem er die Daten anzeigt und bearbeitet, vielleicht stundenlang offen lassen und unter Umständen wird so lange auch die Transaktion gestartet bleiben. Im *concurrency*-Modus hätte der Benutzer nach Stunden noch immer die selbe (evtl. veraltete) Sicht auf die Daten wie zu Anfang, obwohl inzwischen andere Benutzer massenweise Änderungen gemacht haben können.

Eine Transaktion auf dem Ego-Trip

Schließlich gibt es noch den *consistency*-Modus, der die Transaktion in eine geradezu „egoistische" Betriebsart versetzt. In diesem Modus erhält die Transaktion exklusiven Zugriff auf alle Datensätze in den Tabellen, die sie berührt und außerdem auf Tabellen, die beim Starten der Transaktion aufgelistet werden können. Alle anderen Transaktionen bleiben ausgesperrt. Dieser Modus wird höchst selten verwendet, weil er für die Verwendbarkeit einer Datenbank durch mehrere Benutzer eher hinderlich ist. Man kann diesen Modus für besondere Datenbankaufgaben einsetzen, bei denen simultane Schreibzugriffe auf bestimmte Tabellen nicht erlaubt werden sollen. Bei einer Buchhaltung zum Beispiel könnte eine solche besondere Aufgabe ein Jahresabschluss sein, bei dem gewisse abschließende Schreibzugriffe auf die Daten des alten Jahres erfolgen und das neue Jahr initialisiert wird, währenddessen kein anderer Benutzer mehr irgendwelche Buchungen auf das alte Jahr vornehmen darf.

Eine Transaktion im Modus *consistency* versetzt ganze Tabellen in einen reservierten Zustand, sobald sie diese einmal beschrieben hat. Außerdem können ganze Tabellen beim Starten der Transaktion von vornherein zu „Privateigentum" der Transaktion erklärt werden. Wie gesagt: Es handelt sich um den „Ego-Modus".

Es ist klar, dass die Datenbank einen ziemlich hohen Aufwand betreiben muss, um für Transaktionen im *concurrency*- oder *consistency*-Modus dauerhaft die gleiche Sicht auf die Daten aufrecht zu erhalten. Riesige Datenbestände können in der Zwischenzeit von anderen Transaktionen gelöscht und große Mengen neuer Datensätze hinzugefügt worden sein, aber eine *concurrency*-Transaktion geht das alles nichts an – sie schwelgt in der Vergangenheit und InterBase muss für sie die Vergangenheit lebendig halten, d.h. noch die alten Versionen aller inzwischen geänderten und gelöschten Datensätze bereit-

1	**delete from** buchungen;	
2	**insert into** buchungen **values** (1600, 'H',-80.00, 'Fachbuch');	
3	**insert into** buchungen **values** (6820, 'S',80.00, 'Fachbuch');	
4	**commit;**	
5		commit;
6	**set transaction read committed** record_version;	
7	**select* from** buchungen **where** kontonr=1600; *<Ergebnis: 1600|H|-80.00|Fachbuch>*	
8		**insert into** buchungen values (1600, 'H',-13.50, 'Kaffee');
9		**insert into** buchungen values (6820, 'S',13.50, 'Kaffee');
10		**commit;**
11	**select** sum(betrag) **from** buchungen **where** kontonr=1600; *<Ergebnis: -93.50>*	

Tabelle 4: Experiment 4

halten. Eine *read committed*-Transaktion ist da weniger anspruchsvoll, denn sie „geht mit der Zeit".

Man sollte also besonders *concurrency*- (und erst recht *consistency*-) Transaktionen nicht unnötig lange laufen lassen. Meist werden sie ohnehin für automatische, d.h. nicht vom Benutzer gelenkte Aufgaben eingesetzt, etwa für das Erzeugen von Berichten, das Generieren von zusammenfassenden Datensätzen o.ä. Notwendig ist die Verwendung einer *concurrency*-Transaktion dann, wenn ein Vorgang dauerhaft auf die Konsistenz von Daten, die auf verschiedene Datensätze verteilt sind, angewiesen ist.

Unheimliche Begegnungen

Nehmen wir an, Sie wollen einen Kontoauszug für ein bestimmtes Konto der Buchhaltung als Bericht ausdrucken (siehe Abb. 2).

Um diesen Ausdruck erstellen zu können, müssen Sie die Buchungen für das gewünschte Konto abrufen (z.B. *„select * from buchungen where kontonr=1600"*) und schließlich die Summe (den Saldo) des Kontos ermitteln (mit *„select sum(betrag) from buchungen where kontonr=1600"*). Da es sich bei der Buchhaltung um ein Mehrbenutzersystem handelt, kann man nicht ausschließen, dass – parallel zu der Berichterstellung – eine andere Transaktion auf dasselbe Konto Buchungen vornimmt. Mit dem Isolationsmodus *read committed* könnte das aussehen wie in Tabelle 4.

Die Datenbank ist zu jedem Zeitpunkt konsistent, aber der Bericht wäre dennoch falsch! Der Saldo (Zeile 11 in Tabelle 4) passt nicht zu den ausgegebenen Buchungszeilen (Zeile 7), weil inzwischen die rechte Transaktion eine neue Buchung auf das fragliche Konto geschrieben (Zeile 8) und diese Änderung committed hat (Zeile 10). Wegen des links gewählten Isolationsmodus *read committed* (Zeile 6) ist die Änderung der rechten Seite schon sichtbar, als der Saldo ermittelt wird und fließt in die Summe mit ein. Man spricht hier übrigens von dem Phantom-Problem. Dieselbe Situation, allerdings mit dem Isolationsmodus *concurrency* (= *snapshot*) zeigt Tabelle 5.

Diesmal stimmt der Saldo, denn die linke Transaktion behält Ihre Sicht der Daten ab dem Starten der Transaktion (Zeile 6 in Tabelle 5) bei. Der so ausgedruckte Kontoauszug ist zwar nicht mehr topaktuell, denn inzwischen gibt es ja neue Buchungen für dieses Konto, aber zumindest ist kein Fehler im Ausdruck.

Neu-Gierig

In den bisherigen Beispielen habe ich beim Isolationsmodus *read committed* stillschweigend auch immer die Option *record_version* angegeben. Wenn Sie einen Blick auf das Syntaxdiagramm von *Set Transaction* werfen, dann sehen Sie, dass speziell und nur bei *read committed* noch die Auswahl zwischen *record_version* und *no record_version* (default) besteht. Die Option *record_version* gibt an, dass die Transaktion Datensätze auch dann lesen soll, wenn andere, simultane Transaktionen gerade Änderungen an diesen Datensätzen vornehmen, aber diese Änderungen noch nicht committed sind. *record_version* bedeutet also in etwa: „Lese auch ältere Datensatzversionen, auch wenn sich diese Datensätze gerade in Bearbeitung befinden." Das sollte eigentlich das Standardverhalten für eine *read committed*-Transaktion sein, seltsamerweise ist aber *no record_version* die Defaulteinstellung bei *Set Transaction* (d.h. sie gilt, wenn man keine der beiden Optionen angibt). *No record_version* bewirkt, dass sich die Transaktion „weigert", Datensätze zu lesen, die sich gerade anderweitig in Bearbeitung befinden.

Ich selbst hatte noch nie Verwendung für die Option *no record_version*, denn sie garantiert mir zwar, dass im Augenblick des Lesens der Daten niemand sonst etwas an diesen Daten verändert, aber dies schützt mich natürlich nicht davor, dass dies im allernächsten Augenblick (nach dem Lesen) doch geschieht. Ein praktisches Anwendungsbeispiel sehe ich am ehesten im Bereich von Informationssystemen, bei denen es auf allerhöchste Aktualität ankommt. Wenn Sie etwa ein Buchungssystem für Flugreisen programmieren und für eine bestimmte Flugnummer die Anzahl der verfügbaren Sitzplätze abfragen wollen, dann könnten Sie mit den Optionen *wait* und *no record_version* sicherstellen, dass Ihnen diese Auskunft erst dann angezeigt wird, wenn eine vielleicht gerade parallel auf dieselbe Flugnummer laufende Buchung abgeschlossen worden ist. Aber: Auch in diesem etwas konstruierten Beispiel ist der Nutzen von *no record_version* doch recht zweifelhaft, denn sie verzögert nur die Ausgabe einer Information, falls diese

1	**delete from** buchungen;	
2	**insert into** buchungen values (1600, 'H',-80.00, 'Fachbuch');	
3	**insert into** buchungen values (6820, 'S',80.00, 'Fachbuch');	
4	**commit**;	
5		**commit**;
6	**set transaction snapshot**;	
7	**select * from** buchungen **where** kontonr=1600; *<Ergebnis: 1600|HI-80.00|Fachbuch>*	
8		**insert into** buchungen values (1600, 'H',-13.50, 'Kaffee');
9		**insert into** buchungen values (6820, 'S',13.50, 'Kaffee');
10		**commit**;
11	**select** sum(betrag) **from** buchungen **where** kontonr=1600; *<Ergebnis: -80.00>*	

Tabelle 5: Experiment 5

gerade verändert wird, doch nach der Ausgabe hindert dies andere Weltenbummler nicht daran, Ihnen den letzten Platz vor der Nase wegzuschnappen. Wenn also nur noch zwei Plätze für den abgefragten Flug verfügbar sind, dann muss man sich auf jeden Fall mit der Buchung beeilen, egal ob die Abfrage im Kontext einer *no record_version*-Transaktion läuft oder nicht.

Ich möchte also grundsätzlich empfehlen, bei *read committed* auch *record_version* zu wählen (nicht umsonst ist dies auch die Voreinstellung bei Client-Bibliotheken wie IBObjects und anderen). Nachfolgend noch ein Beispiel, das die Wirkung von *[no] record_version* zusammen mit *wait* und *no wait* illustrieren soll (siehe Tabelle 6).

	/* 1. Versuch */	
1	**set transaction wait read committed no** record_version;	
2		**commit**;
3	**select * from** konten **where** kontonr=1600; <Ergebnis: 1600lKasse>	
4		**update** konten **set** bezeichnung='Bargeldkasse' **where** kontonr=1600;
5	**select** * **from** konten **where** kontonr=1600; <... wartet ...>	
6		**commit**;
	<Ergebnis: 1600lBargeldkasse>	
	/* 2. Versuch */	
7	**set transaction no wait read committed no** record_version;	
8	**select * from** konten **where** kontonr=1600; <Ergebnis: 1600lBargeldkasse>	
9		**update** konten **set** bezeichnung='Hauptkasse' **where** kontonr=1600;
10	**select * from** konten **where** kontonr=1600; <Fehler SQLCode=-901, lock conflict on no wait transaction, deadlock>	
11		**commit;**
12	**select * from** konten **where** kontonr=1600; <Ergebnis: 1600lHauptkasse>	
	/* 3. Versuch */	
13	**set transaction wait read committed** record_version;	
14		**update** konten **set** bezeichnung='Kasse' **where** kontonr=1600;
15	**select * from** konten **where** kontonr=1600; <Ergebnis: 1600lHauptkasse>	
16		**commit;**
17	**select * from** konten **where** kontonr=1600; <Ergebnis: 1600lHauptkasse>	

Tabelle 6: Experiment 6

Das Ergebnis auf das *Select* in Zeile 5 (Tabelle 6) wird erst ausgegeben, nachdem die Transaktion rechts mit *Commit* beendet wurde (Zeile 6). Bis dahin blockiert der SELECT-Befehl die Sitzung, denn die Option *no record_version* verlangt, dass keine Datensätze gelesen werden, die gerade von anderen Transaktionen bearbeitet werden. Genau dies ist aber nach dem Update der rechten Seite (Zeile 4) der Fall. Man sollte wissen, dass es keinen Timeout-Mechanismus für dieses Verhalten gibt. Wenn die Transaktion rechts eine Stunde benötigt, um die Transaktion und damit die Datensatzänderung abzuschließen, dann dauert es auch auf der linken Seite eine Stunde, bis das *Select* ein Ergebnis liefert.

Dieselbe Situation führt in Zeile 10 zu einer Fehlermeldung, denn diesmal wurde für die Transaktion links die Option *no wait* angegeben (Zeile 7). Mit der Option *wait* würde das *Select* (genauso wie in Zeile 4) blockieren; *no wait* verbietet dies und meldet stattdessen sofort einen Deadlock-Fehler.

Im dritten Versuch (Zeile 13-17) wird nochmals das Verhalten einer *read committed*-Transaktion, für die mit *record_version* das Lesen von Datensatzversionen erlaubt ist, demonstriert. Man sieht: Diese Variante bereitet keine Probleme.

Ausblick

Wir haben und bisher hauptsächlich mit den Aspekten Atomität und Isolation, also den eher defensiven Qualitäten von Transaktionen, beschäftigt. Im zweiten Teil dieses Artikels werden wir in die Offensive gehen und die Probleme der Schreibkonflikte konkurrierender Transaktionen beleuchten.

Quellcode

Den Quellcode finden Sie auf der aktuellen Profi-CD sowie auf unserer homepage unter
www.derentwickler.de