Wenn zwei sich streiten

Die Eigenheiten von InterBase-Transaktionen verstehen und nutzen – Teil 2

von Karsten Strobel

Wenn zwei Vögel an einem Wurm zerren, dann zerreißt er meistens in der Mitte. Als Datenbankentwickler kann man da nur den Kopf schütteln. Wie können zwei Anwender nur so unkoordiniert und chaotisch auf dieselbe Ressource zugreifen! Kein Wunder, dass es da zu Schäden an den Strukturen kommt! Zum Glück ist das bei InterBase besser organisiert...

Im ersten Teil dieses Artikels (als pdf-Datei auf der Profi-CD und als Online-Artikel unter www.derentwickler.de zu finden) wurden die von InterBase unterstützten Transaktions-Modi vorgestellt und deren Anwendung und Wirkung anhand einiger Bespiele erkundet. Dabei ging es vorwiegend um das Problem der Isolation, d.h. der Abschottung einer Transaktion von den Änderungen konkurrierender Transaktionen. In diesem zweiten Teil werden wir uns dem – ebenso wichtigen – Problem der Schreibkonflikte stellen.

Quellcode

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

Ein Schreibkonflikt tritt immer dann auf, wenn mehrere simultane Transaktionen versuchen, die selben Datensätze zu verändern (durch Einfügen, Ändern oder Löschen). Die Transaktion, die als erste einen Datensatz verändert, sperrt diesen für den (schreibenden) Zugriff durch andere Transaktionen. Die Sperre (engl. Lock) des Datensatzes wird erst aufgehoben, wenn die Transaktion, die den ersten Schreibzugriff durchgeführt und damit die Sperre bewirkt hatte, beendet wird. Dies kann bekanntlich entweder durch ein Commit oder ein Rollback geschehen.

Für das simultane Verändern eindeutiger Indexe (insb. PRIMARY oder UNIQUE KEYs) gelten die gleichen Regeln. Das bedeutet, dass zwar zwei Transaktionen gleichzeitig Datensätze in eine Tabelle einfügen können. Wenn der Tabelle aber ein eindeutiger Index angehört und beide Transaktionen den gleichen Schlüsselwert einzutragen versuchen, kommt es zu einem Schreibkonflikt.

Dank der besonderen Architektur von InterBase gelten Sperren immer für einzelne Datensätze und nicht – wie bei manchen anderen Datenbanken üblich – für mehrere Datensätze, die sich zufällig gemeinsam in einer Speicherseite befinden, oder gar für ganze Tabellen. Eine Ausnahme von dieser Regel besteht mit dem bereits besprochenen Isolationsmodus *snapshot table stability* (auch *consistency* genannt). In diesem (eher unüblichen) Modus dehnt sich eine Datensatzsperre auf eine ganze Tabelle aus, was für eine Transaktion ein höchst eigensinniges Verhalten ist und in einer Mehrbenutzerumgebung nur in seltenen Ausnahmefällen Anwendung findet.

Konfliktbewältigung – manchmal hilft Geduld

Wir hatten im ersten Teil bereits die *[no] wait*-Option bei der Parametrierung von Transaktionen verwendet, und zwar im Zusammenhang mit dem Isolationsmodus *read committed no record_version* und dem Verhalten einer so eingestellten Transaktion beim Lesen von Daten. Die *[no] wait*-Option ist aber keineswegs nur für das Leseverhalten von Bedeutung. Im Gegenteil: Auf das Lesen von Daten hat sie nämlich nur in ganz seltenen Situationen Einfluss, die – wie bereits ausgeführt – in der Praxis ohnehin meist vermieden werden. Viel wichtiger ist die *[no] wait*-Einstellung in Zusammenhang mit kollidierenden Schreibzugriffen, denn sie steuert das Verhalten einer Transaktion, die beim

Schreiben auf eine bereits bestehende Sperre trifft.

Sehen wir uns das mal an einem Beispiel an. Um die Beispiele nachzuvollziehen, verwenden Sie bitte wieder zwei parallele Instanzen von (W)ISQL und die Testdatenbank mit den im ersten Teil definierten Tabellen. Die SQL-Anweisungen zum Erzeugen der Testdatenbank finden Sie auch auf der Profi-CD und im Web.

Wir sehen, dass der gleichzeitige Schreibzugriff zweier simultan aktiver Transaktionen auf ein- und denselben Datensatz – nämlich der Datensatz in der Tabelle *konten*, der über den Hauptschlüssel *kontonr=1600* eindeutig identifiziert wird – zu einem Schreibkonflikt führt. Mit *gleichzeitig* ist dabei übrigens nicht ein Zugriff durch beide Transaktionen im selben Augenblick oder innerhalb einer gewissen Zeit gemeint; es bedeutet einfach einen Konflikt zweier im selben Zeitraum aktiver Transaktionen – deren Lebensdauer, ob nun Sekundenbruchteile oder Stunden, ist dabei ohne Bedeutung.

Die erste Transaktion, die den Datensatz in *konten* aktualisiert, „gewinnt". Beim ersten Versuch (vgl. Tabelle 1) ist dies die linke Transaktion, beim zweiten Mal die rechte. Die andere Transaktion läuft gegen eine Sperre, die die erste Transaktion hinterlassen hat. Die Sperre gilt nur für diesen einzelnen Datensatz und sie dauert so lange, bis die sperrende Transaktion ein Commit oder Rollback ausführt. Von der Einstellung der *wait* Option ist es dann abhängig, ob der Konflikt sofort zur Auslösung einer Deadlock-Fehlermeldung führt, so wie rechts (Zl. 4), wo *wait* ausgeschaltet ist (Zl. 2). Auf der linken Seite dagegen ist die *wait*-Option eingeschaltet (Zl. 1); diese Transaktion wartet deshalb so lange, bis die Datensatzsperre wieder aufgehoben wird (Zl. 8) und versucht dann erneut, den UPDATE-Befehl auszuführen, was allerdings scheitert, weil sich inzwischen der Inhalt dieses Datensatzes geändert hat. Da der UPDATE nun definitiv unzulässig ist, kommt es schließlich doch zu der Deadlock-Fehlermeldung (nach Zl. 8).

	/* 1. Versuch */	
1	**set transaction wait read committed** record_version;	
2		**set transaction no wait read committed** record_version;
3	**update** konten **set** bezeichnung= 'Bargeldkasse' **where** kontonr=1600;	
4		**update** konten **set** bezeichnung='Hauptkasse' **where** kontonr=1600; <Fehler SQLCode=-901, lock conflict on no wait transaction, deadlock>
5	**commit**; /* 2. Versuch */	
6		**update** konten **set** bezeichnung='Hauptkasse' **where** kontonr=1600;
7	**update** konten **set** bezeichnung='Kasse' **where** kontonr=1600; <... wartet ...>	
8	<Fehler SQLCode=-901, lock conflict on no wait transaction, deadlock>	**commit**;

Tabelle 1

1	**set transaction wait read committed** record_version;	
2		**update** konten **set** bezeichnung='Bargeldkasse' **where** kontonr=1600;
3	**update** konten **set** bezeichnung='Kasse' **where** kontonr=1600; <... wartet ...>	
4	<update ist erfolgreich>	**rollback**;
5	**commit**;	

Tabelle 2

Wo aber soll nun der große Unterschied zwischen *wait* und *no wait* bestehen, wenn es schließlich doch auf die gleiche Fehlermeldung hinausläuft, was bei eingeschaltetem *wait* nur etwas länger dauert? Der Unterschied ist der, dass eine *no wait*-Transaktion gleich „aufgibt", wenn es zu einem Konflikt kommt, während die *wait*-Transaktion wartet, ob die versuchte Schreibaktion vielleicht – nach dem Ende der Datensatzsperre – doch noch möglich ist. Es kann nämlich durchaus sein, dass sich dieses Warten lohnt, vor allem dann, wenn die „gegnerische" Transaktion es sich quasi anders überlegt und ihre Datenänderung mit Rollback zurücknimmt (siehe Tabelle 2).

Das *Rollback* der rechten Transaktion (Zl. 4) bewirkt hier, dass die Datensatzsperre ohne Änderung des Datensatzes aufgehoben wird. Anschließend kann der UPDATE der linken Transaktion (Zl. 3), der zuerst in einen Wartezustand führte, doch noch erfolgreich ausgeführt werden. Hätte die linke Transaktion die Option *no wait* gehabt, dann wäre der UPDATE sofort mit einer Deadlock-Fehlermeldung gescheitert.

Bei einem Schreibkonflikt auf einen Rückzieher der anderen Transaktion zu hoffen, ist in der Praxis zwar nicht

sonderlich aussichtsreich, denn Rollbacks sind ja eher die Ausnahme denn die Regel, aber es gibt durchaus auch Fälle, in denen das Warten immer die bessere Alternative ist, um einen Deadlock zu vermeiden. Auch hierzu ein Beispiel in Tabelle 3.

Der erste Versuch zeigt das Verhalten bei der Einstellung *no wait*. Der INSERT-Befehl links (Zl. 6) schlägt wegen eines Schreibkonfliktes fehl, und zwar weil der neue Datensatz in *buchungen* mit dem Feld *kontonr* einen Fremdschlüssel auf die Tabelle *konten* besitzt und sich darüber auf einen Datensatz in *konten* bezieht, der aufgrund des UPDATEs der rechten Transaktion (Zl. 5) gerade gesperrt ist. Dieser Konflikt resultiert in der *lock conflict*-Fehlermeldung.

Beim zweiten Versuch ist der Ablauf genau der gleiche, außer dass diesmal die linke Transaktion die Einstellung *wait* erhält (Zl. 8). Der gleiche INSERT wie beim ersten Versuch (Zl. 10) führt diesmal in einen Wartezustand. Sobald die Sperre auf den Datensatz in *konten* durch die rechte Transaktion per *Commit* aufgehoben ist, versucht die linke Transaktion denselben Befehl automatisch noch mal auszuführen, und zwar mit Erfolg, denn der UPDATE der rechten Transaktion (Zl. 9) hat zwar den Datensatz vorübergehend gesperrt, aber ihn nicht so verändert, dass das Schreiben des neuen Datensatzes in *buchungen* (mit Bezug auf *konten*) unmöglich wäre. Unmöglich wäre dies nur dann gewesen, wenn der UPDATE rechts nicht den Wert von *bezeichnung*, sondern von *kontonr* geändert hätte, denn dann wäre nach Aufheben der Sperre der Datensatz in *konten* mit *kontonr=1600* plötzlich nicht mehr da gewesen, und der INSERT wäre fehlgeschlagen.

	/* 1. Versuch */	
1	**delete from** buchungen;	
2	**commit**;	
3		**commit**;
4	**set transaction no wait read committed** record_version;	
5		**update** konten **set** bezeichnung='Hauptkasse' **where** kontonr=1600;
6	**insert into** buchungen values (1600, 'H', -80.00, 'Fachbuch'); <Fehler SQLCode=-901, lock conflict on no wait transaction, violation of FOREIGN KEY constraint „INTEG_4" on table „BUCHUNGEN">	
7		**commit**;
	/* 2. Versuch */	
8	**set transaction wait read committed** record_version;	
9		**update** konten **set** bezeichnung='Nebenkasse' **where** kontonr=1600;
10	**insert into** buchungen **values** (1600, 'H', -80.00, 'Fachbuch'); <... wartet ...>	
11		**commit**;
	<insert ist erfolgreich>	
12	**insert into** buchungen **values** (6820, 'S', 80.00, 'Fachbuch');	
13	**commit**;	

Tabelle 3

Dies ist ein sehr schönes – weil praxisnahes – Beispiel dafür, dass die Verwendung der Option *wait* oft sinnvoll ist. Eine Transaktion mit eingeschalteter *wait* Option wartet solange, bis tatsächlich feststeht, ob ein Konflikt aufgetreten ist oder nicht, und setzt ihre Arbeit ohne Fehlermeldung fort, sofern dies möglich ist. Eine *no wait*-Transaktion dagegen meldet schon „beim ersten Anzeichen" eines Konfliktes sofort einen Fehler. Der Nachteil von *wait* liegt auf der Hand: Die Wartezeit ist nicht begrenzt, außer durch das Verhalten der anderen Transaktion(en), mit denen der Konflikt besteht. Solange diese anderen Transaktionen noch aktiv sind, solange verharrt die wartende Transaktion und gibt die Kontrolle nicht an das aufrufende Programm zurück. Wenn durch die Programmierung der Anwendung garantiert ist, dass insbesondere schreibende Transaktionen nicht lange laufen, dann kann die Wartezeit akzeptiert werden. Wenn die Transaktionsdauer dagegen von den Launen eines Benutzers abhängt, dann sollte man auf *wait* eher verzichten.

Variationen eines Happy Ends

In allen unseren Beispielen haben wir *Commit* verwendet, um eine Transaktion zu beenden und die Änderungen zu übernehmen. Tatsächlich gibt es hierzu noch zwei Alternativen, die allerdings nicht mit (W)ISQL angewendet werden können: Erstens kann man eine InterBase-Transaktion so parametrieren, dass sie automatisch nach jeder Datenänderung ein Commit ausführt. Zweitens gibt es die Variante „Commit Retaining", was man in etwa mit „ausführen, aber bewahren" übersetzen könnte. Commit Retaining wird mitunter auch als „weiches Commit" bezeichnet. Diese Einstellung ist auch als „Auto-Commit" oder „Server-Auto-Commit" bekannt. Ein normales (hartes) Commit beendet eine Transaktion und schließt dabei alle noch offenen Ergebniszeiger. Ein weiches Commit bewirkt ebenfalls das permanente Speichern der Änderungen, aber erhält die gegenwärtigen Ergebniszeiger und (falls vorhanden) auch den aktuellen Snapshot. Ein solches „weiches Commit" kann man entweder durch Aufruf

der API-Funktion *isc_commit_retaining()* bzw. der entsprechenden Bibliotheksfunktion erreichen, oder man setzt für die Transaktion den Parameter *isc_tpb_autocommit* bzw. die entsprechende Eigenschaft der Komponentenbibliothek. Ein vom Server durchgeführtes automatisches Commit ist immer ein weiches Commit.

Auto-Commit wird häufig für *read committed*-Transaktionen bei interaktiven Client-Anwendungen verwendet. Speziell beim Navigieren und Editieren mit den bei Windows-Entwicklern so populären DBGrid-Steuerelementen ist diese Parameterkombination sehr sinnvoll, denn meistens soll jede Änderung des Benutzers sofort nach dem Speichern auch permanent und für andere sichtbar sein, und ein hartes Commit würde den Client zwingen, entweder immer die komplette Datenmenge vom Server in den Hauptspeicher zu laden oder sie nach dem Commit jedes Mal neu abzufragen.

Hinter den Kulissen

Es lohnt sich, auch einmal einen Blick hinter die Kulissen zu werfen und sich anzusehen, wie das InterBase-Transaktionsmodell eigentlich funktioniert. Wenn man eine Produktbeschreibung von InterBase liest, dann trifft man sehr bald auf den Begriff *Multigenerationsarchitektur* (MGA). Was hat es damit auf sich?

Grob gesagt heißt es, dass die Datenbank einen existierenden Datensatz aufgrund einer Update- oder Delete-Operation eines Anwenders zunächst einmal nicht verändert. Statt dessen entsteht eine neue „Generation“ des Datensatzes, der an die Stelle des Vorgängers tritt und an der selben Stelle gespeichert wird. Die ältere Generation wird von ihrem angestammten Platz verdrängt und stattdessen an einem anderen Ort gespeichert. Die neue Datensatzgeneration merkt sich einen Zeiger auf die vorherige, und jeder Datensatz erhält auch noch den Stempel derjenigen Transaktion, die diese Daten erzeugt hat, aufgeprägt, und zwar in Form einer Transaktions-Identnummer. Eine solche wird beim Starten einer jeden neuen Transaktion aus dem Feld „Next Transaction Id“ (in der Kopfseite der Datenbankdatei) vergeben.

Beim Verändern eines Datensatzes kommt es häufig vor, dass nur der Inhalt eines einzigen oder nur weniger Felder abgeändert wird, die übrigen Felder aber unberührt bleiben. Um das Volumen der Datenbank nicht durch das Schreiben immer neuer, fast gleicher Datensatzgenerationen aufzublähen, bedient sich InterBase sogenannter Änderungsdatensätze (Delta Records), um nur die tatsächlichen Unterschiede zweier Datensatzgenerationen auf möglichst kleinem Raum zu notieren.

Wenn ein Anwender im Kontext einer Transaktion auf einen Datensatz zugreifen will, so überprüft der Datenbankkern anhand des Verzeichnisses aller Transaktionen, welches in den sogenannten *Transaction Inventory Pages* (TIP) festgehalten wird, ob die vorgefundene Datensatzgeneration für diese Transaktion sichtbar ist. Dies ist *nicht* der Fall, wenn diese Datensatzgeneration von einer anderen Transaktion stammt, die entweder noch nicht beendet ist oder deren Änderungen inzwischen mit Rollback zurückgenommen worden sind. Ist die vorgefundene Generation also nicht gültig, so wird die Vorgängergeneration (sofern es eine gibt) der selben Prüfung unterzogen, und so fort. Auf diese Weise sorgt die MGA von InterBase für die Isolation der Transaktionen.

Transaktionen isolieren

Der Unterschied zwischen einer *read committed*- und einer *concurrency*-Transaktion ist dabei, dass für erstere die zuvor beschriebene Prüfung anhand des allgemeinen Transaktionsverzeichnisses durchgeführt wird, wobei die Statusänderungen konkurrierender Transaktionen berücksichtigt werden; letztere dagegen verwendet eine Kopie der TIP-Seiten (einen *snapshot*), die beim Transaktionsstart angelegt wurde. Auf diese Weise liest eine Transaktion im *read committed* Isolationsmodus auch Änderungen anderer Transaktionen (sofern diese committed sind), eine Transaktion mit *concurrency*-Isolationsmodus hat einen konstante Sicht auf die Daten, ohne dass dafür eine Kopie ganzer Tabellen angelegt werden müsste.

Auskehren

Natürlich muss InterBase nicht die Zustände sämtlicher Transaktionen seit dem allerersten Zugriff auf eine Datenbank aufzeichnen. Änderungen an Datensätzen können irgendwann durch weitere Änderungen überschrieben werden. Wenn keine aktive Transaktion mehr die vorherige Datensatzgeneration sehen will, dann ist diese Transaktion und deren Änderungen nicht mehr „interessant“ und wird auch nicht mehr weiter verfolgt. In der Kopfseite einer InterBase-Datenbank werden zwei Variablen gespeichert, die für das Transaktionsmanagement sehr wichtig sind: Die Oldest Interesting Transaction (OIT) gibt die älteste Transaktion an, deren Änderungen noch von Bedeutung sind. Die Oldest Active Transaction (OAT) ist die älteste zur Zeit noch aktive (d.h. laufende) Transaktion. Das Fortschreiben dieser Zähler wird beim Zugriff neuer Transaktionen auf die Datenbank sozusagen nebenbei erledigt. (Mit dem Kommando „*gstat.exe –h <lokale Datenbank>*“ kann man sich übrigens den Inhalt der Kopfseite anzeigen lassen.) Neue Transaktionen machen sich auch nützlich, wenn sie auf Datensätze stoßen, zu denen ältere Generationen existieren, die von inzwischen nicht mehr interessanten Transaktionen stammen. Diese älteren, nicht mehr relevanten Datenversionen werden bei der Gelegenheit gleich beseitigt (Garbage Collection). Bei InterBase 5 führt diese zwanghafte Reinlichkeit manchmal zu unschönen Geschwindigkeitseinbußen, wenn eine Transaktion viele Datensätze berührt, die von einer früheren Transaktion verändert worden sind. Mit InterBase 6 wurde dieses Verhalten erheblich verbessert, denn hier interagieren die Transaktionen mit einem niedrig priorisierten Thread, der die Aufräumarbeiten besorgt. Neben der – eher beiläufigen – Selbstbereinigung der Datenbank durch Transaktionen, die über irrelevante Datenversionen stolpern und diese beseitigen, gibt es noch einen dedi-

zierten Aufräumvorgang, das sogenannte *Sweeping* (= Fegen), das entweder manuell gestartet werden kann (mit „*gfix.exe –sweep*“) oder automatisch in Aktion tritt, wenn der Abstand zwischen OIT und OAT größer als ein einstellbarer Wert (default = 20000) geworden ist.

Je größer der Abstand zwischen der ältesten interessanten Transaktion und der nächsten Transaktion ist, umso größer ist der Aufwand, der für die Verwaltung des Transaktionsverzeichnisses und älterer Datensatzgenerationen betrieben werden muss. Eine einzige Transaktion, die einmal gestartet wurde und auf unbestimmte Zeit läuft, verhindert, dass OIT und OAT mit neuen Transaktionen Schritt halten und lässt den Wasserkopf der Datenbank immer mehr anschwellen. Es spricht also einiges dafür, Transaktionen immer so kurz wie möglich zu halten, und zwar umso mehr, wenn die Anwendung mit einer großen Zahl simultaner Benutzer (also auch Transaktionen) arbeitet. An dieser Stelle möchte ich für Entwickler, die mit Delphi oder Borland C++Builder arbeiten, auf IBObjects (www.ibobjects.com), eine sehr ausgereifte Komponentenbibliothek für die Programmierung von InterBase Client-Anwendungen, hinweisen. IBObjects beinhaltet sogenannte *OAT-Management*-Funktionen. Diese bieten eine weitgehende Automatisierung der Lebenszeitbegrenzung von *read committed*- Transaktionen, was einem viele Sorgen mit den Folgen lange laufender Transaktionen ersparen kann.

Die Multigenerationsarchitektur hat zwei wichtige Vorteile: Erstens ermöglicht sie eine friedliches Nebeneinander von lesenden und schreibenden Transaktionen. Im Gegensatz zu anderen Architekturen ist es für eine Transaktion nicht notwendig, einen gelesenen Datensatz mit einer Sperre zu belegen, um konkurrierende Transaktionen daran zu hindern, diesen Datensatz zu modifizieren, denn sie kann weiterhin auf die gelesene Version der Daten zugreifen, auch wenn eine andere Transaktion diese ändert und eine neue Generation des Datensatzes erzeugt. Zweitens benötigt man bei dieser Architektur keine Transaktions-Logs, wie andere Datenbanken sie pflegen, um Änderungen bei einem Rollback oder einem Systemabsturz zurückzunehmen. Nach einem Stromausfall z.B. kann InterBase mit geringem Aufwand und ohne manuellen Eingriff die Konsistenz wiederherstellen, indem alle vor dem Absturz aktiven Transaktionen für tot erklärt werden, womit alle ihre Änderungen für neue Transaktionen unsichtbar bleiben.

Fazit

InterBase verfügt über ausgereifte Transaktionsmechanismen, die auch bei gleichzeitigem Benutzerzugriff dank punktgenauer Datensatzsperren und der Multigenerationsarchitektur nur wenig Konflikte erzeugen. Das Transaktionsmanagement geschieht völlig automatisch und es ist wenig oder kein Administrationsaufwand für den Sysop notwendig. Wenn man die Wirkung der verschiedenen Transaktionsparameter kennt und je nach Anwendungsfall das richtige Mittel wählt, und außerdem die Achillesferse (lange laufende Transaktionen) nicht reizt, dann kann man mit InterBase sehr zuverlässige und leistungsfähige Mehrbenutzer-Anwendungen realisieren.