Die Protokollierung

Serverseitiges Auditing – Protokollierungsmechanismus für InterBase 7.1 selbst gemacht

Ist Ihnen Big Brother noch ein Begriff? Sie wissen schon, diese Fernsehsendung, in der man eine Anzahl von Personen in einen Container gesperrt hat, und die Zuschauer vor dem Bildschirm quasi als spionierendes Auge an diesem Geschehen teilhaben konnten. Etwas provokant gefragt, aber hätten Sie sich als Anwendungsentwickler oder Datenbankadministrator nicht schon öfters in einer ähnlichen Lage sehen wollen, wenn es darum ging, Ihren Benutzern bei der Dateneingabe etwas über die Schulter schauen zu können, ohne dass diese etwas davon mitbekommen? Ja? Dann begleiten Sie mich doch durch diesen Artikel. Ich werde Ihnen zeigen, wie man sich mit Bordmitteln selbst einen serverseitigen Protokollierungsmechanismus in einer InterBase-Datenbank integriert.

von Thomas Steinmaurer

Eine der herausragendsten Eigenschaften von InterBase gegenüber manchen Mitstreitern am Datenbankmarkt ist die Abhandlung konkurrierender Zugriffe im Mehrbenutzerbetrieb durch die Multi-Generationen-Architektur (MGA) [1] – auch „Versioning" genannt. Wird ein Datensatz geändert, so wird dieser nicht direkt bearbeitet, sondern es wird eine neue Version dieses Datensatzes angelegt, die unter anderem auch eine eindeutige Nummer der Transaktion beinhaltet, von der die Datenänderung durchgeführt wurde. Dieser Mechanismus stellt sicher, dass parallel laufende Transaktionen immer eine konsistente Sicht auf die zu verarbeitende Ergebnismenge besitzen, ohne sich gegenseitig sperren zu müssen. Erst wenn zwei Transaktionen denselben Datensatz ändern wollen, kommt es zu einem Konflikt, der abhängig von den verwendeten Transaktionsparametern entsprechend behandelt wird. In aller Kürze kann man die MGA so zusammenfassen, dass lesende Operationen keine schreibende Operationen und umgekehrt blockieren. Somit ist InterBase sowohl für den OLTP (Online Transaction Processing) als auch den DSS (Decision Support System) Betrieb bestens geeignet. Eine weitere positive Eigenschaft der MGA ist, dass im Falle eines Datenbankcrashs keine Transaction-Log-Datei durchlaufen werden muss, um die Datenbank bis zu einem bestimmten Zeitpunkt wieder in Betrieb nehmen zu können, da nicht-beendete Transaktionen beim Wiederanlauf einfach zurückgesetzt (Rollback) werden. Daher besitzt – im Gegensatz zu anderen Datenbankprodukten – InterBase kein Transaction-Log, da es hierfür einfach keine Verwendung gibt.

Quellcode

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

Anwendungsszenarien

Seit InterBase 7 existieren in der Server Edition die so genannten „Monitoring-Tables" [2] [3], die einem Datenbankadministrator das Leben erheblich erleichtern. Man ist hiermit in der Lage, das faktuelle Geschehen in einer Datenbank nachzuvollziehen, und z.B. Datenbankverbindungen, langandauernde Transaktionen oder Abfragen gezielt zu beenden. Leider ist dieses durchaus nützliche Feature nicht in der Lage, über durchgeführte datenmanipulierende Operationen (*Delete/Insert/Update*) Auskunft zu geben. Die Entwickler sind somit auf sich alleine gestellt, einen Protokollierungsmechanismus – der vollständig am Server abzulaufen hat – selbst zu entwickeln. Die Mühen der Umsetzung machen sich allemal bezahlt. Dies soll die folgende Auflistung der möglichen Anwendungsszenarien einer serverseitigen Protokollierung aufzeigen:

- Um folgende Frage zu beantworten: „Wann hat wer welchen Datensatz gelöscht, eingefügt oder geändert?"
- Für statistische Auswertungen (z.B. Preisschwankungen von Artikeln über einen bestimmten Zeitraum).
- Für den Anwendungsentwickler, um komplexe Geschäftsprozesse in Hinblick auf die notwendigen datenmanipulierenden Operationen (DML) zu optimieren.
- „Black Box Debugging", um zu sehen, was sich z.B. hinter einem Klick auf einen Button in einem Fremdprodukt bzgl. Datenänderungen abspielt.

- Änderungsprotokolle (Audit Trails) als Notwendigkeit zur Erfüllung bestimmter nationaler und internationaler Standards.
- Protokollierte Operationen als wichtiger Bestandteil des so genannten Briefkastenmodells beim Abgleichen/Replizieren von Datenbeständen.

Sie sehen, die möglichen Einsatzzwecke sind breit gefächert. Vielleicht wollen Sie aber auch nur zu einem bestimmten Zeitpunkt Ihren Mitarbeitern oder Kunden bei der Arbeit etwas über die Schulter blicken, ohne jedoch wirklich anwesend sein zu müssen?

Bestandsaufnahme

Bekanntlich führen viele Wege nach Rom. Dass dies vor allem in der Softwareentwicklung auch Umwege sein können, hat sicherlich schon jeder einmal selbst erfahren. In unserem Fall haben wir jedoch Glück, weil unsere Hauptanforderung für einen Protokollierungsmechanismus ist, dass dieser am Server – losgelöst von jeglicher Client-Anwendung – abzulaufen hat. Hier gibt es für InterBase nur eines, nämlich Trigger.

InterBase ist bekannt und beliebt für dessen leicht zu lernende, jedoch nicht mindermächtige Prozedurale Sprache (PSQL) für gespeicherte Prozeduren und Trigger. Die Programmlogik kann damit teilweise vom Client zum Server transferiert werden, um die Netzwerkbelastung zu reduzieren bzw. die Wiederverwendbarkeit und die Wartbarkeit zu erhöhen. Im Unterschied zu einigen anderen Datenbankherstellern handelt es sich bei InterBase PSQL um eine vollkommen eigenständige Programmiersprache, die vom Datenbankserver übersetzt wird und das Kompilat direkt in der Datenbank ablegt. Der Vorteil besteht darin, dass man die Datenbank sehr einfach auf eine andere Plattform portieren kann, ohne dafür serverseitigen Code für jede Plattform neu übersetzen zu müssen.

Das besondere an einem Trigger ist nun, dass dessen Ausführung nicht manuell angestoßen werden muss, sondern dieser wird beim Auftreten einer datenändernden Operation automatisch ausgeführt; man sagt auch dazu „gefeuert“. Ist z.B. ein *AFTER INSERT*-Trigger auf eine Tabelle definiert, so wird der Code des Triggers automatisch ausgeführt, nachdem ein neuer Datensatz in diese Tabelle eingefügt wurde. Die Trigger-Implementierung von InterBase gehört zu einer der Besten am Markt, und zeichnet sich durch folgende Merkmale aus:

- Trigger in InterBase arbeiten datensatzorientiert. Werden über eine *UPDATE*-Anweisung 100 Datensätze geändert, dann wird ein *BEFORE/AFTER UPDATE*-Trigger hundertmal ausgeführt.
- Es können mehrere Trigger für ein und dieselbe Aktion auf einer Tabelle definiert werden, wobei die Ausführungsreihenfolge durch das optionale Schlüsselwort *POSITION* angegeben werden kann.
- Über die Kontextvariablen *OLD.<spalte>* und *NEW.<spalte>* kann abhängig vom Aktionstyps des Triggers auf alte und/oder neue Spaltenwerte beliebigen Datentyps (auch BLOB) innerhalb des Triggers zugegriffen werden.
- Trigger können deaktiviert und zu einem späteren Zeitpunkt wieder aktiviert werden.

Es gäbe noch weitere interessante Merkmale der Triggerimplementierung in InterBase. Eine Auflistung würde hier den Rahmen sprengen, und wäre auch unnötig, weil für unseren Protokollierungsmechanismus die hier angeführten Punkte am wichtigsten sind. Ein datensatzorientierter Trigger wird uns für jeden geänderten Datensatz einen Eintrag in einer Log-Tabelle erstellen. Die Möglichkeit mehrere Trigger für eine Aktion auf einer Tabelle zu definieren unterstützt uns, einen bereits existierenden Trigger nicht ändern zu müssen, sondern unser Trigger kann als abgeschlossene Einheit angesehen werden. Die *OLD/NEW*-Kontextvariablen ermöglichen uns den einfachen Zugriff auf alte/neue Spaltenwerte beliebigen Datentyps, und ersparen uns die Basistabelle mit temporären Tabellen verknüpfen (joinen) zu müssen, um zu den geänderten Werten zu kommen, wie dies in anderen Datenbankprodukten der Fall ist. Zu guter Letzt kann es auch hilfreich sein, einen Logging-Trigger vorübergehend zu deaktivieren, ohne diesen aus der Datenbank vollständig entfernen zu müssen.

Logging-Datenmodell

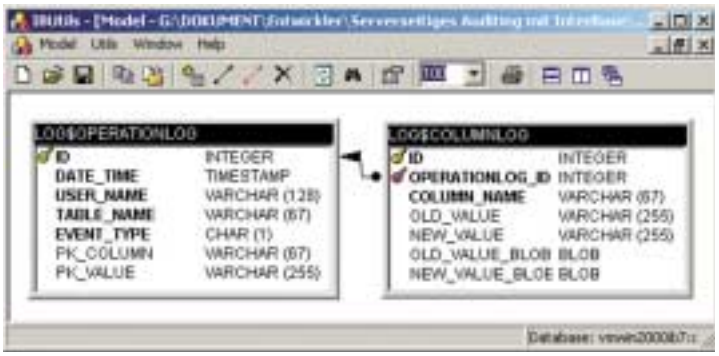
Was brauchen wir nun außer der Implementierung von Triggern noch? Richtig: Um Datenänderungen mitprotokollieren zu können, benötigen wir noch Tabellen in der Datenbank, in denen protokollierte Datenänderungen abgespeichert werden. Diese Tabellen müssen sich in der zu überwachenden Datenbank befinden, da InterBase keine so genannten Cross-Database-Queries unterstützt. Im Klartext bedeutet das, dass eine *INSERT INTO*-Anweisung in einem Trigger nur auf einer Tabelle in derselben Datenbank durchgeführt werden kann. Das Logging-Datenmodell kann abhängig von den Anforderungen, die der Protokollierungsmechanismus zu erfüllen hat, unterschiedlich aussehen. Folgende Einsatzwecke sind denkbar:

- Eine Log-Tabelle, die nur für eine spezielle Anforderung herangezogen wird. Ein konkretes Beispiel ist die Protokollierung von Preisänderungen in einer Artikeltabelle, wobei die Log-Tabelle zumindest den Primarschlüsselwert der Artikeltabelle, sowie den alten und neuen Artikelpreis zusammen mit dem Änderungsdatum speichern sollte.
- Verwendung von „Schattentabellen“. Hierfür wird für jede zu überwachende Tabelle eine Schattentabelle (Log-Tabelle) gleicher Struktur angelegt. Nachteil dieser Variante ist, dass beide Tabellen bzgl. Struktur synchron gehalten werden müssen.
- Generische Log-Tabellen, die neben dem Änderungsdatum auch den Tabellennamen, Aktionstyp (*Delete/Insert/Update*), den/die Namen der Primärschlüsselspalte(n) und deren Wert(e) speichert. In einer weiteren Tabelle können dann die alten/neuen Spaltenwerte zusammen mit dem Feldnamen abgelegt werden, wobei diese Tabelle eine *1:n*-Verbindung zu der vorher angeführten Tabelle besitzt.

Ich werde mich im weiteren Verlauf nur dem generischen Ansatz widmen, da dies die flexibelste Variante darstellt. Lassen Sie uns nun zur Implementierung kom-

men. Dazu verwende ich die Dialekt 1 Beispieldatenbank *employee.gdb*, die mit jeder InterBase-Installation mitkommt. Abbildung 1 zeigt für den im vorherigen Abschnitt angeführten generischen Ansatz ein einfaches Datenmodell, bestehend aus zwei Tabellen.

Abb. 1: Logging-Datenmodell



In der Tabelle *LOG$OPERATIONLOG* wird, von einem Trigger für jeden geänderten Datensatz einer Tabelle, ein Eintrag mit dem Zeitstempel, dem verbundenen Benutzer, dem Tabellennamen und der ausgeführten Aktion eingefügt. Falls die Tabelle einen Primarschlüssel besitzt, werden des Weiteren der Name des Primärschlüsselfeldes und dessen Wert mitprotokolliert. Damit kann man zu einem späteren Zeitpunkt die protokollierte Änderung dem zugrunde liegenden Datensatz wieder zuordnen, sofern sich natürlich der Primärschlüsselwert nicht geändert hat. Um die Sache nicht unnötig zu verkomplizieren, unterstützen wir nur Tabellen mit nicht-zusammengesetzten Primärschlüsseln. Der Datentyp von *TABLE_NAME* und *PK_COLUMN* wurde so gewählt, dass die seit InterBase 7 neue maximale Länge für Datenbankobjektnamen von 67 Zeichen berücksichtigt wird. Die Tabelle *LOG$COLUMNLOG* referenziert einen „übergeordneten“ Datensatz in *LOG$OPERATIONLOG*, und speichert Datenänderungen mit dem alten und neuen Wert zusammen mit dem Feldnamen. Hierbei muss jedoch zwischen Datentypen, die implizit auf den Datentyp *VARCHAR* umgewandelt (gecasted) werden können, und dem Datentyp BLOB unterschieden werden. Unser Beispiel verwendet einen BLOB Subtyp 0 (binär), der in der Lage ist, einen BLOB beliebigen Subtyps zu speichern. Den Array-Datentyp unterstützen wir nicht. Es handelt sich hierbei um Log-Tabellen für eine Dialekt 1 Datenbank. Für eine Dialekt 3 Datenbank kann dieses Datenmodell fast unverändert übernommen werden. Statt dem *DATE*-Datentyp sollten Sie dann *TIMESTAMP* verwenden. Für die Primär/Fremd-Schlüsselfelder der Log-Tabellen wäre es von Vorteil, einen 64-Bit Integer Datentyp – *NUMERIC (18, 0)* – zu verwenden. Des Weiteren werden zwei Generatoren *GEN_LOG$OPERATIONLOG* und *GEN_LOG$COLUMNLOG* verwendet, um

den Primärschlüsselfeldern der Log-Tabellen eindeutige Werte zuzuweisen.

Logging-Trigger

Bei einem Logging-Trigger sollte es sich immer um einen *AFTER*-Trigger handeln. Dieser ist in der Ausführungsreihenfolge immer der letzte im Bunde, auch wenn benutzerdefinierte Constraints (die intern durch System-Trigger realisiert sind) vorhanden sind. Des Weiteren sollte ein Logging-Trigger immer der letzte in der Kette aller vorhandenen *AFTER*-Trigger für eine Operation einer Tabelle sein. Dies wird erreicht, indem man bei der Triggererstellung eine POSITION von 32767 verwendet. InterBase bietet leider im Unterschied zu anderen Produkten wie z.B. Oracle, Microsoft SQL Server oder Firebird 1.5 nicht die Möglichkeit, nur einen Trigger für alle datenmanipulierenden Operationen auf einer Tabelle zu implementieren. Daher muss für jeden zu protokollierenden Änderungstyp (*Delete*/*Insert*/*Update*) ein eigener Trigger erstellt werden. Deshalb sind drei Trigger für eine Tabelle nötig, falls alle drei Änderungstypen protokolliert werden sollen.

Listing 1 zeigt einen Trigger, der Löschoperationen auf der Tabelle *CUSTOMER* mitprotokolliert, wobei ein Logging auf Feldebene nur für das Feld *CONTACT_LAST* durchgeführt wird. Zu Beginn wird der nächste Generatorwert von *GEN_LOG$OPERATIONLOG* in die Variable *VAR_ID* zwischengespeichert, da dieser Wert wegen der *1:n*-Verbindung in unserem Logging-Datenmodell, für die spätere Einfüge-Operation des Spalten-Loggings benötigt wird. Die erste *INSERT INTO*-Anweisung fügt einen neuen Datensatz in *LOG$OPERATIONLOG* ein. Dieser Datensatz enthält Informationen wann wer welchen Datensatz aus der Tabelle *CUSTOMER* gelöscht hat. Der eingefügte Datensatz in *LOG$CLUMNLOG* gibt nähere Auskunft darüber, welches Feld welchen Wert zum Zeitpunkt der Löschoperation gehabt hat. In unserem Fall gibt es nur einen Eintrag für das Feld *CONTACT_LAST* pro protokollierte Löschoperation, sofern *CONTACT_LAST* nicht *NULL* war, da wir das vorher explizit mit der IF-Bedingung überprüfen.

Listing 2 zeigt dieselbe Vorgehensweise für einen Trigger, der Einfüge-Operationen mitprotokolliert. Der einzige Unterschied besteht in der Verwendung der *OLD*/*NEW*-Kontextvariablen. In einem *INSERT*-Trigger steht der Zugriff auf Spaltenwerte nur über die Kontextvariable *NEW* zur Verfügung, wohingegen in einem *DELETE*-Trigger dies nur über *OLD.<spalte>* möglich ist. In einem *UPDATE*-Trigger stehen beide Kontextvariablen zur Verfügung, und beinhalten sowohl den alten als auch den neuen Spaltenwert.

Listing 3 zeigt einen Trigger, der Änderungsoperationen in der Tabelle *CUSTOMER* mitprotokolliert, und bei einer Änderung im Feld *CONTACT_LAST* eine Protokollierung auf Feldebene durchführt. Für den *UPDATE*-Trigger möchte ich noch zusätzlich auf zwei Dinge kurz eingehen, die manchmal übersehen werden:

- Vergleich von BLOB-Feldern: Der Ungleichheitsoperator <> kann auf BLOB-Feldern nicht angewendet werden. Um zu überprüfen, ob sich in einem BLOB-Feld etwas geändert hat, muss eine UDF verwendet werden. Diese kann z.B. eine Überprüfung mittels einer CRC-Prüfsumme oder einen Binärvergleich durchführen. Eine UDF-Bibliothek mit einer Funktion für den Binärvergleich zweier BLOBs, finden Sie für Windows unter [4].

Listing 1

```
SET TERM ^^ ;
CREATE TRIGGER TRI_CUSTOMER_L_D FOR CUSTOMER
                ACTIVE AFTER DELETE POSITION 32767 AS
DECLARE VARIABLE VAR_ID INTEGER;
BEGIN
 VAR_ID = GEN_ID(GEN_LOG$OPERATIONLOG, 1);
 // Operations-Log
 INSERT INTO LOG$OPERATIONLOG (ID, TABLE_NAME,
                EVENT_TYPE, PK_COLUMN, PK_VALUE)
  VALUES (:VAR_ID, 'CUSTOMER', 'D', 'CUST_NO',
OLD.CUST_NO);
 // Spalten-Log
 IF (OLD.CONTACT_LAST IS NOT NULL) THEN
  INSERT INTO LOG$COLUMNLOG (ID, OPERATIONLOG_ID,
                COLUMN_NAME, OLD_VALUE)
   VALUES (GEN_ID(GEN_LOG$COLUMNLOG, 1), :VAR_ID,
                'CONTACT_LAST', OLD.CONTACT_LAST);
 // Hier können weitere Spalten kommen ...
END
^^
SET TERM ; ^^
```

Listing 2

```
SET TERM ^^ ;
CREATE TRIGGER TRI_CUSTOMER_L_I FOR CUSTOMER
                ACTIVE AFTER INSERT POSITION 32767 AS
DECLARE VARIABLE VAR_ID INTEGER;
BEGIN
 VAR_ID = GEN_ID(GEN_LOG$OPERATIONLOG, 1);
 // Operations-Log
 INSERT INTO LOG$OPERATIONLOG (ID, TABLE_NAME,
                EVENT_TYPE, PK_COLUMN, PK_VALUE)
  VALUES (:VAR_ID, 'CUSTOMER', 'I', 'CUST_NO',
                NEW.CUST_NO);
 // Spalten-Log
 IF (NEW.CONTACT_LAST IS NOT NULL) THEN
  INSERT INTO LOG$COLUMNLOG (ID, OPERATIONLOG_ID,
                COLUMN_NAME, NEW_VALUE)
   VALUES (GEN_ID(GEN_LOG$COLUMNLOG, 1), :VAR_ID,
                'CONTACT_LAST', NEW.CONTACT_LAST);
 // Hier können weitere Spalten kommen ...
END
^^
SET TERM ; ^^
```

Listing 3

```
SET TERM ^^ ;
CREATE TRIGGER TRI_CUSTOMER_L_U FOR CUSTOMER
                ACTIVE AFTER UPDATE POSITION 32767 AS
DECLARE VARIABLE VAR_ID INTEGER;
BEGIN
 VAR_ID = GEN_ID(GEN_LOG$OPERATIONLOG, 1);
 // Operations-Log
 INSERT INTO LOG$OPERATIONLOG (ID, TABLE_NAME,
                EVENT_TYPE, PK_COLUMN, PK_VALUE)
  VALUES (:VAR_ID, 'CUSTOMER', 'U', 'CUST_NO',
                NEW.CUST_NO);
 // Spalten-Log
 IF (((OLD.CONTACT_LAST IS NULL) AND
                (NEW.CONTACT_LAST IS NOT NULL)) OR
  ((NEW.CONTACT_LAST IS NULL) AND
                (OLD.CONTACT_LAST IS NOT NULL)) OR
  (OLD.CONTACT_LAST <> NEW.CONTACT_LAST)
  ) THEN
  INSERT INTO LOG$COLUMNLOG (ID, OPERATIONLOG_ID,
                COLUMN_NAME, OLD_VALUE, NEW_VALUE)
   VALUES (GEN_ID(GEN_LOG$COLUMNLOG, 1), :VAR_ID,
                'CONTACT_LAST', OLD.CONTACT_LAST,
                NEW.CONTACT_LAST);
 // Hier können weitere Spalten kommen ...
END
^^
SET TERM ; ^^
```

- Eine Änderung in Feldern, die *NULL* Zustände erlauben: Um Änderungen in einem Feld festzustellen, das nicht als *NOT NULL* definiert wurde, reicht es nicht aus, auf Ungleichheit abzufragen, da dies bei *NULL* Zuständen nicht zum erwünschten bzw. erwarteten Ergebnis führt. Es muss hierfür die IF-Bedingung mit *IS NULL* und *IS NOT NULL* für alte/neue Spaltenwerte erweitert werden. Wurde ein Feld als *NOT NULL* deklariert, so kann diese zusätzliche Überprüfung wegfallen, und es reicht aus, nur den Ungleichheitsoperator <> zu verwenden.

Testen Sie nun den installierten Protokollierungsmechanismus durch Ausführen von Lösch-, Einfüge- und Änderungsoperationen auf der Tabelle *CUSTOMER*, und im speziellen auf der Spalte *CONTACT_LAST*.

Vergessen Sie nicht, diese Änderungen über ein *COMMIT* zu bestätigen. In den Log-Tabellen sollten nun entsprechende Datensätze erscheinen. Verwenden Sie dazu eine gewöhnliche *SELECT*-Abfrage auf den Log-Tabellen *LOG$OPERATIONLOG* bzw. *LOG$COLUMNLOG*.

Was ist mit Firebird?

Firebird [5] ist eine sehr aktive Weiterentwicklung der im Jahre 2000 veröffentlichten Sourcen von InterBase 6. Sie können das hier erworbene Know-how natürlich auch für Firebird verwenden. Folgende Punkte sollten Sie allerdings beachten:

- Die maximale Länge von Datenbankobjektnamen beträgt nicht wie in InterBase 7 67 Zeichen, sondern nur 31 Zeichen (Stand: Firebird 1.5). Dies sollte im Logging-Datenmodell Berücksichtigung finden.
- Firebird 1.5 unterstützt so genannte Multi-Aktionen-Trigger, d.h. Sie können (müssen aber nicht) eine Protokollierung für alle Aktionstypen mit nur einem Trigger realisieren. Die neuen booleschen Kontextvariablen *DELETING*, *INSERTING* und *UPDATING* werden Ihnen dabei gute Dienste leisten, wenn Sie innerhalb des Triggers überprüfen wollen, von welcher datenmanipulierenden Operation der Trigger ausgelöst wurde.
- Firebird 1.5 stellt die neuen Kontextvariablen *CURRENT_TRANSACTION* und *CURRENT_CONNECTION* zur Verfügung, die ebenfalls im Logging-Datenmodell, im speziellen in der Tabelle *LOG$OPERATIONLOG*, integriert werden könnten. Bei beiden handelt es sich um einen 32-Bit Integer Wert. Vor allem *CURRENT_TRANSACTION* wäre von Interesse, da hiermit alle Änderungen innerhalb einer Transaktion über die Log-Tabelle abgefragt werden können. Berücksichtigen Sie aber, dass *CURRENT_TRANSACTION* nach einem Backup/Restore wieder von vorne beginnt.

Tool-Unterstützung

Das hier angewandte Konzept für einen serverseitigen Protokollierungsmechanismus mag durchaus einleuchtend sein, allerdings stellt sich schon sehr schnell die Frage, ob es nicht eine Tool-Unterstützung zum Erstellen der Trigger gibt. Für InterBase und Firebird stellt die IB LogManager-Produktfamilie eine sehr gute Lösung dar. Dieses Produkt wurde bereits in [6] kurz vorgestellt. Eine Testversion kann unter [7] heruntergeladen werden. Für andere Datenbankprodukte wird man schneller fündig. Suchen Sie im Internet einfach nach den Begriffen „Database Auditing“ oder „Database Logging“.

Fazit

Wie wir gesehen haben, lässt sich ein serverseitiger Protokollierungsmechanismus mit InterBase-Bordmitteln relativ einfach in Ihrer Datenbank integrieren. Man sollte den dabei gewonnen Nutzen nicht unterschätzen, weil es sich bei dem anfallenden Datenmaterial nicht immer um Abfall handeln muss. Versuchen Sie es einfach selbst. Vielleicht können Sie schon den nächsten reklamierten Programmfehler als Eingabefehler widerlegen.
Happy Auditing!

Links & Literatur

[1] www.dbginc.com/tech_pprs/IB.html
[2] www.dbginc.com/tech_pprs/ib7/IB7.htm
[3] K. Strobel: Ungleiche Zwillinge *Der Entwickler* 2.2003
[4] www.ibphoenix.com/downloads/freeudflib20010211.zip
[5] www.firebirdsql.org
[6] K. Strobel: Werkzeugkoffer – Tools für InterBase *Der Entwickler* 6.2003
[7] www.upscene.com