Fasten your seat belt

Transaction-Log und Redo-Mechanismus mit InterBase

Es ist nicht nur im Gesundheitswesen so, dass präventive Maßnahmen in der Regel viel billiger sind als aufwendige und kostspielige Behandlungen – wenn diese dann wirklich benötigt werden. Diese zugegebenermaßen nicht allzu neue Binsenweisheit trifft auch für Softwaresysteme zu, die im 24 x 7-Betrieb ihre Arbeit verlässlich verrichten müssen. Ist ein Datenbankmanagementsystem mit an Bord, dann sind es vor allem die Datenbankadministratoren, die gefordert sind, entsprechende Backup- und Recoverystrategien zu entwickeln, um für den Fall der Fälle gerüstet zu sein. Dies gilt auch für eine InterBase-Datenbank. Auf Nummer sicher zu gehen ist erst der Anfang. Aktiv dafür etwas zu tun ist das Ziel. Sehen wir uns ein mögliches Ziel, in Form des hier vorgestellten Roll-Forward-Log-Mechanismus, etwas näher an. Legen wir gemeinsam den Sicherheitsgurt an.

von Thomas Steinmaurer

In meinem letzten Artikel [1] habe ich Ihnen das Grundwissen vermittelt, wie man selbst einen serverseitigen Protokollierungsmechanismus in einer InterBase-Datenbank integriert. In diesem Artikel werden wir uns ein konkretes Anwendungsbeispiel für die Weiterverarbeitung der angefallenen Protokolldaten ansehen, nämlich die Haltung einer so genannten Transaction-Logdatenbank und einem darauf arbeitenden Redo-Mechanismus. Ein InterBase-Quereinsteiger, der vorher mit anderen namhaften Datenbankprodukten gearbeitet hat, würde nun vermutlich etwas verwundert grübeln, aber berechtigterweise fragen: „Besitzt InterBase kein Transaction-Log und wie verläuft dann das Wiederanfahren im Falle eines Servercrashs?" InterBase – und auch Firebird – hat in diesem Bereich durch deren Multi-Generationen-Architektur (MGA) [2] eine Sonderstellung inne, da eine konsistente Sicht auf Daten im Mehrbenutzerzugriff nicht durch Page/Table/Record-Locks sichergestellt wird, sondern für einen Datensatz während dessen Lebensdauer mehrere Datensatzversionen existieren. Dies wird auch als „Versioning" bezeichnet. Durch die MGA benötigt InterBase kein Transaction-Log, um die Datenbank im Fehlerfall für einen bestimmten Zeitpunkt wieder zur Verfügung zu stellen. Beim Wiederanlauf werden nicht beendete Transaktionen einfach zurückgesetzt.

Quellcode

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

Backup, Ausfallsicherheit und ähnliche Unannehmlichkeiten

InterBase ist beliebt wegen seiner einfachen Installation und Administration. Eine Herausforderung für Administratoren stellt sich jedoch spätestens dann, wenn man eine Datenbank im Gigabyte-Bereich sein Eigen nennen darf und hierfür auch für eine entsprechende Backup- und Ausfallsicherheitsstrategie sorgen muss. Frei nach Murphy heißt es: „Wenn etwas schief gehen kann, dann wird es auch schief gehen." In solch einer Situation kann man sich dann glücklich schätzen, ein funktionierendes Backup in den Händen zu halten. Backup ist jedoch nicht gleich Backup, vor allem bei einer InterBase-Datenbank. Ist bei der Konzipierung einer Backupstrategie für ein Firmennetzwerk eine InterBase-Datenbank mit im Spiel, dann müssen Sie eine Regel beachten: Lassen Sie die Backupsoftware nie eine aktive InterBase-Datenbank direkt mitsichern, da dies eine der häufigsten Ursachen darstellt, warum es zu einer Beschädigung von Datenbankdateien kommt [3].

InterBase bietet für das Backup einer Datenbank das Kommandozeilentool *gbak.exe* an. Verwenden Sie dieses Tool, um ein Backup Ihrer Datenbank zu erstellen, welches anschließend von der Backupsoftware des Servers mitgesichert wird. Kontrollieren Sie jedoch in regelmäßigen Abständen mit einem Restore (ebenfalls mit *gbak.exe*), ob man aus dem Backup wieder eine funktionsfähige Datenbank erhält. Bei den unzähligen Talenten, die *gbak.exe* besitzt, ist eines jedoch nicht möglich: Man kann hiermit keine inkrementellen Sicherungen durchführen, die Verwendung geschieht

nach dem „Alles oder Nichts"-Prinzip. Entweder es wird die komplette Datenbank gesichert oder gar nichts. Bei sehr großen Datenbanken kann und wird dies einiges an Zeit und Ressourcen in Anspruch nehmen. InterBase bietet zwar auch das Konzept der Erstellung von so genannten Schattendateien (Shadow) an, in denen Datenänderungen gespiegelt werden, aber diese sind auf Festplatten beschränkt, die sich lokal zur InterBase-Installation befinden müssen. Dieses Konzept kann daher eigentlich auch nicht wirklich für ein Fail-over eingesetzt werden.

Wie so oft ist vieles vorgegeben, jedoch manches beeinflussbar. Ich werde Ihnen zeigen, wie Sie sich selbst eine Transaction-Log-Datenbank halten und protokollierte Operationen auf einer Backup-Datenbank erneut ausführen, um letztendlich eine Datenbank – für einen bestimmten Zeitpunkt – in einem konsistenten Zustand zu erhalten. Dies wird auch als Roll-Forward-Log bezeichnet.

Zutatenmix

Die Realisierung eines Roll-Forward-Log-Mechanismus für InterBase ist keine triviale Sache mehr, sondern umfasst mehrere Softwarekomponenten, die perfekt miteinander harmonieren müssen:

- serverseitiger Protokollierungsmechanismus.
- Datentransfer-Tool, um protokollierte Datenänderungen von der Produktionsdatenbank in eine separate „Transaction-Log"-Datenbank zu schaufeln, die sich durchaus auf einem anderen Rechner im Netzwerk befinden kann. Dies hat den Vorteil, dass die Produktionsdatenbank nicht mit den Logdaten „verunreinigt" wird. Wichtiger ist jedoch, dass man auf die Protokolldaten auch dann zurückgreifen kann, wenn die Produktionsdatenbank beschädigt wurde.
- Redo-Tool, welches das Änderungsprotokoll sequenziell durchläuft und auf einer Backup-Datenbank erneut ausführt.

Serverseitiger Protokollierungsmechanismus

Die Integration eines serverseitigen Protokollierungsmechanismus können wir, mit dem in [1] angeeigneten Know-how, als abgeschlossen ansehen. Führen Sie hierfür die SQL-Skriptdateien *erstelle_audit_tabellen_dialekt1.sql* und *erstelle_audit_trigger.sql* auf der Employee-Beispieldatenbank Ihrer InterBase-Installation aus. Sie erhalten hiermit eine vollständige Protokollierung aller Datenänderungen auf allen Feldern der Tabelle *CUSTOMER*. Sollten Sie eine Dialekt-3-Datenbank verwenden wollen, dann verwenden Sie zum Erstellen der Log-Tabellen das Skript *erstelle_audit_tabellen_dialekt3.sql*, da sich das Logging-Datenmodell durch die verwendeten Datentypen geringfügig unterscheidet.

Auf den in Listing 1 abgebildeten Trigger möchte ich kurz etwas näher eingehen. Es handelt sich hierbei um einen *AFTER INSERT*-Trigger auf der Log-Tabelle *LOG $OPERATIONLOG*, der in einer möglichen Kette von *AFTER INSERT*-Triggern am Schluss (Position 32767) gefeuert wird. In diesem Trigger wird über das *POST_EVENT*-Kommando ein Event *LOG$INSERTED* losgeschickt, das von interessierten Clients aufgefangen und entsprechend weiterverarbeitet werden kann. Diesen Eventmechanismus werden wir uns zu Nutze machen, um – mit einer geringen zeitlichen Verzögerung – protokollierte Datenänderungen in eine andere InterBase-Datenbank zu transferieren. Sollte nun der eine oder andere InterBase/Firebird-Kenner sagen, dass Events nicht zuverlässig bzw. verwendbar sind, vor allem wenn eine Firewall mit im Spiel ist, dann ist das mit InterBase 7.1 und Firebird 1.5 nicht mehr richtig. Verwendeten InterBase < 7.1 und Firebird 1.0 für die Eventkommunikation noch nicht vorhersagbare Ports, so geschieht dies in InterBase 7.1 jetzt über denselben Port, der auch für alle anderen Anfragen verwendet wird (Default: 3050). In Firebird 1.5 ist dieser Port über den Parameter *RemoteAuxPort* in der Konfigurationsdatei *firebird.conf* frei konfigurierbar.

Datentransfer-Tool

Das Datentransfer-Tool ist dafür zuständig, dass Logdaten von der Produktionsdatenbank in eine andere Datenbank verschoben werden. Die Zieldatenbank, ich nenne sie Transaction-Log-Datenbank, kann durchaus auch auf einem anderen Rechner im Netzwerk liegen. Die wichtigste funktionale Anforderung an dieses Tool ist, dass sich die Daten beider Datenbanken sowohl vor als auch nach dem Datentransfer, in einem konsistenten Zustand befinden müssen. Die Protokolldaten werden in die Transaction-Log-Datenbank eingefügt und nur wenn diese Folge von Einfügeoperationen erfolgreich abgeschlossen wurde, dürfen die Logdaten in der Produktionsdatenbank gelöscht werden. InterBase ist hierfür bestens gerüstet, da es das so genannte Two-Phase-Commit-Protokoll unterstützt, welches es ermöglicht, dass eine Transaktion über mehrere Datenbanken hinweg gestartet, bestätigt (Commit) bzw. zurückgesetzt (Rollback) werden kann. Dieses mächtige Werkzeug steht dem Anwendungsentwickler über die InterBase-Express-(IBX-)Komponenten oder auch mit IBObjects [4] transparent zur Verfügung. Weitere Punkte, die bei der Realisierung berücksichtigt werden sollten, sind:

- Benutzeroberfläche vs. Kommandozeile
- Zeitpunkt (synchron/asynchron) für das Anstoßen des Transferprozesses

Für den Echtbetrieb würde sich vielleicht eine kommandozeilenorientierte Variante besser eignen. Man wäre hiermit flexibler, was den Ausführungszeitpunkt betrifft, da man über einen Task-Scheduler des Betriebssystems den Datentransfer zu vorgegebenen Zeitpunkten automatisch starten könnte. Dies ist vor allem dann relevant, wenn man den Eventmechanismus von InterBase nicht einsetzen möchte. Um Ih-

Listing 1

```
// AFTER INSERT Trigger auf LOG$OPERATIONLOG, um
// Event zu posten, falls neuer Datensatz eingefügt wurde
SET TERM ^^ ;
CREATE TRIGGER TRI_LOG$OPERATIONLOG_AI FOR
LOG$OPERATIONLOG ACTIVE AFTER INSERT POSITION
32767 AS
BEGIN
 POST_EVENT 'LOG$INSERTED';
END
^^
SET TERM ; ^^

COMMIT WORK;
```

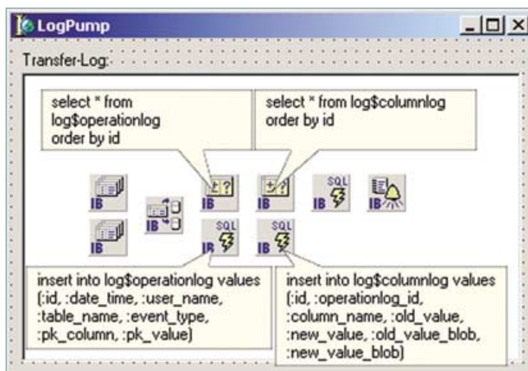

Abb.1: Entwickleransicht – LogPump

Abb. 2: LogPump in Aktion

nen das Zusammenspiel der unterschiedlichen IBX-Komponenten besser zu erläutern, habe ich mich allerdings für die Variante mit einer Benutzeroberfläche und dem synchronen Anstoßen des Transferprozesses über den bereits erwähnten Eventmechanismus entschieden. Abbildung 1 zeigt die Entwickleransicht einer einfach gehaltenen Version unseres Datentransfer-Tools.

Der Zugriff auf die Produktionsdatenbank geschieht über die Komponenten der oberen Reihe. Die beiden *TIBDataSet* stellen den Zugriff auf die Log-Tabellen *LOG$OPERATIONLOG* und *LOG$-COLUMNLOG* dar. Für beide wurde *UniDirectional* auf *True* gesetzt, da dies die schnellste und ressourcenschonendste Alternative zum Durchlaufen einer Ergebnismenge darstellt. *TIBSQL* wird verwendet, um die Logdaten nach dem Transfer in die Transaction-Logdatenbank aus der Produktionsdatenbank zu löschen. Die *TIBEvents*-Komponente ist dafür zuständig, den vom *AFTER INSERT*-Trigger gefeuerten Event *LOG$INSERTED* abzufangen, um dann im *OnEventAlert*-Ereignis den Datentransfer anzustoßen. Dazu wird dem Event *Property* eine Zeile mit dem Eventnamen *LOG$INSERTED* hinzugefügt. *AutoRegister* wird auf *True* gesetzt.

Die Komponenten der unteren Reihe sind für den Zugriff auf die Transaction-Log-Datenbank zuständig. Bei den *TIBSQL*-Komponenten handelt es sich um parametrisierte *INSERT INTO*-Anweisungen, die für die Einfügeoperationen in die Log-Tabellen der Transaction-Logdatenbank zuständig sind. Die Verwendung einer parametrisierten Anweisung besitzt folgende Vorteile:

- Bessere Performance: Bei jedem Schleifendurchlauf werden nur die Parameterwerte und nicht die vollständige SQL-Anweisung ersetzt. Dies entlastet den InterBase-Server und das Netzwerk.
- BLOB-Daten: BLOB-Daten – inklusive Binärdaten – können sehr einfach transferiert werden.

Beide Datenbankverbindungen werden über ein und dieselbe Transaktionskomponente durchgeschleust. Somit verwenden wir automatisch das Two-Phase- Commit-Protokoll von InterBase. Für die *TIBTransaction*-Komponente wird ein Isolationsgrad *SNAPSHOT* (concurrency, nowait) verwendet. So wird sichergestellt, dass wir innerhalb der Transaktion eine konsistente Sicht auf die zu übertragenden und zu löschenden Logdaten haben.

Erstellen Sie nun eine neue Datenbank. Achten Sie darauf, dass Sie denselben SQL-Dialekt verwenden, den auch die Produktionsdatenbank besitzt. In unserem Beispiel ist dies Dialekt 1. Geben Sie ihr den Namen *employee_log.gdb*. Führen Sie danach das Skript *erstelle_audit_tabellen_dialekt1.sql* auf der neu erstellten Datenbank aus. Öffnen Sie das Delphi-Projekt *LogPump.dpr* und ändern Sie entsprechend die *DatabaseName*-Eigenschaft beider *TIBDatabase*-Komponenten.

Starten Sie die *LogPump*-Anwendung. Ändern Sie einen Datensatz in der Tabelle *CUSTOMER* in *employee.gdb* (nicht in *employee_log. gdb*!) mit einem Tool Ihrer Wahl und bestätigen Sie diese Änderung mit einem Commit. Es wird nun am Server über den *AFTER UPDATE* Log-Trigger auf der Tabelle *CUSTOMER* diese Änderung mitprotokolliert, d.h., neue Datensätze in den Log-Tabellen *LOG$OPERATIONLOG* und *LOG$CLUMNLOG* werden hinzugefügt. Der *AFTER INSERT*-Trigger auf der Tabelle *LOG$OPERATIONLOG* wird hiermit gefeuert und ein Event *LOG$INSERTED* gepostet. Dieses Event wird von *LogPump* aufgefangen. Im *OnEventAlert*-Ereignis wird die *DoPump*-Methode aufgerufen, welche die Datensätze von der Produktionsdatenbank in die *Transaction-Logdaten*bank transferiert, und anschließend werden die transferierten Datensätze aus der Produktionsdatenbank entfernt. In Listing 2 sind die wichtigsten Methoden unseres Datentransfer-Tools zu sehen.

Sollte im ersten Schritt bei den Einfügeoperationen auf der Transaction-Log-Datenbank etwas schief gehen, dann ist durch die Verwendung des Two-Phase-Commit-Protokolls sichergestellt, dass die Logdaten aus der Produktionsdatenbank nicht gelöscht werden. Abbildung 2 zeigt *LogPump* in Aktion.

Redo-Tool

Wir haben nun die protokollierten Operationen nicht mehr in der Produktionsdatenbank, sondern in einer separaten Datenbank. Jetzt fehlt uns nur noch ein Redo-Tool, welches das Änderungsprotokoll in der Transaction-Log-Datenbank sequenziell durchläuft und Schritt für Schritt auf einem Backup unserer Produktionsdatenbank ausführt. Folgende Anforderungen bzw. Probleme können hierfür identifiziert werden:

- Metadatenänderungen: Da wir nur Änderungen an den Daten, jedoch nicht an den Metadaten mitprotokollieren, können auch nur „echte“ Datenänderungen

Anzeige

von unserem Redo-Tool auf der Backup-Datenbank ausgeführt werden. Für Metadatenänderungen könnte man im Redo-Tool eine Funktionalität vorsehen, welche das Ausführen eines SQL-Skripts vor dem Redo-Prozess ermöglicht.

- Stabile Primärschlüsselwerte: Es ist ein Muss, dass eine Tabelle einen Primärschlüssel besitzt und Primärschlüsselwerte „stabil" sind, d.h. nicht geändert werden können, da sonst der richtige Datensatz nicht mehr identifiziert werden könnte.
- Generatorenwerte: Auch Änderungen an den Generatorenwerten werden nicht mitprotokolliert und können somit auch nicht in die Zieldatenbank übertragen werden. Hier könnte das *LogPump*-Utility in regelmäßigen Abständen einen speziellen Eintrag in *LOG$OPERATIONLOG* erstellen, wobei die dazugehörigen Einträge in *LOG$COLUMNLOG* den Generatorennamen und dessen Momentanwerten entsprechen. Dieser spezielle Logdatensatz müsste dann vom Redo-Tool erkannt werden, um die Generatoren über eine Folge von *SET*

Listing 2

```
procedure TfmLogPump.ievLogPumpEventAlert(Sender:
                                              TObject;
 EventName: String; EventCount: Integer;
                             var CancelAlerts: Boolean);
begin
 DoPump;
end;

procedure TfmLogPump.DoPump;
begin
 itrLogPump.StartTransaction;
 try
  // Datensätze in LOG$OPERATIONLOG transferieren
  DoPumpOperationLog;
  // Datensätze in LOG$COLUMNLOG transferieren
  DoPumpColumnLog;
  // Transferierte Logdaten in der Quelldatenbank löschen
  DoDeleteLogdata;
  itrLogPump.Commit;
 except
  itrLogPump.Rollback;
 end;
end;

procedure TfmLogPump.DoPumpOperationLog;
begin
 meLogPump.Lines.Add('Beginn - Datentransfer
                            LOG$OPERATIONLOG ...');
 with idstOperationLogSelect do
 begin
  try
   try
    if not isqlOperationLogInsert.Prepared then isql
                            OperationLogInsert.Prepare;
    Open;
    while not EOF do
    begin
     isqlOperationLogInsert.ParamByName('id').Value :=
                               FieldByName('id').Value;

isqlOperationLogInsert.ParamByName('date_time').Value
                  := FieldByName('date_time').Value;
     isqlOperationLogInsert.ParamByName('user_name').
                 alue := FieldByName('user_name').Value;
     isqlOperationLogInsert.ParamByName('table_name').
              Value := FieldByName('table_name').Value;
     isqlOperationLogInsert.ParamByName('event_type').
              Value := FieldByName('event_type').Value;
     isqlOperationLogInsert.ParamByName('pk_column').
              Value := FieldByName('pk_column').Value;
     isqlOperationLogInsert.ParamByName('pk_value').
                Value := FieldByName('pk_value').Value;
     isqlOperationLogInsert.ExecQuery;
     meLogPump.Lines.Add(Format('Datensatz (id=%s)
       wurde transferiert.', [FieldByName('id').AsString]));
     Next;
    end;
   except
    meLogPump.Lines.Add(Format('Transfer von Datensatz
 (id=%s) fehlgeschlagen!', [FieldByName('id').AsString]));
   end;
  finally
   Close;
  end;
 end;
 meLogPump.Lines.Add('Ende - Datentransfer
                            LOG$OPERATIONLOG ...');
end;

procedure TfmLogPump.DoPumpColumnLog;
begin
 meLogPump.Lines.Add('Beginn - Datentransfer
                               LOG$COLUMNLOG ...');
 with idstColumnLogSelect do
 begin
  try
   try
    if not isqlColumnLogInsert.Prepared then isql
                            ColumnLogInsert.Prepare;
    Open;
    while not EOF do
    begin
     isqlColumnLogInsert.ParamByName('id').Value :=
                               FieldByName('id').Value;

isqlColumnLogInsert.ParamByName('operationlog_id').
        Value := FieldByName('operationlog_id').Value;
     isqlColumnLogInsert.ParamByName('column_name').
           Value := FieldByName('column_name').Value;
     isqlColumnLogInsert.ParamByName('old_value').
               Value := FieldByName('old_value').Value;
     isqlColumnLogInsert.ParamByName('new_value').
               Value := FieldByName('new_value').Value;

isqlColumnLogInsert.ParamByName('old_value_blob').
          Value := FieldByName('old_value_blob').Value;

isqlColumnLogInsert.ParamByName('new_value_blob').
          Value := FieldByName('new_value_blob').Value;
     isqlColumnLogInsert.ExecQuery;
     meLogPump.Lines.Add(Format('Datensatz (id=%s)
       wurde transferiert.', [FieldByName('id').AsString]));
     Next;
    end;
   except
    meLogPump.Lines.Add(Format('Transfer von Datensatz
 (id=%s) fehlgeschlagen!', [FieldByName('id').AsString]));
   end;
  finally
   Close;
  end;
 end;
 meLogPump.Lines.Add('Ende - Datentransfer
                              LOG$COLUMNNLOG ...');
end;

procedure TfmLogPump.DoDeleteLogData;
begin
 meLogPump.Lines.Add('Beginn - Löschen der Logdaten in
                                   Quelldatenbank...');
 with isqlEmployee do
 begin
  SQL.Text := 'delete from log$operationlog';
  ExecQuery;
  (* Hinweis:
   Durch den in der Datenbank definierten kaskadierenden
       FOREIGN-KEY Constraint werden die Datensätze in
         LOG$COLUMNLOG automatisch entfernt, wenn ein
          übergeordneter Datensatz in LOG$OPERATIONLOG
                                       gelöscht wird.
  *)
 end;
 meLogPump.Lines.Add('Ende - Löschen der Logdaten in
                                   Quelldatenbank...');
end;
```

GENERATOR-Anweisungen auf die protokollierten Werte zu setzen.

- Aktive Trigger: Aktive Trigger sollten in der Backup-Datenbank vor dem Redo deaktiviert und danach wieder aktiviert werden. Welche Trigger dies sind, lässt sich über eine entsprechende Abfrage auf die Systemtabelle *RDB$TRIGGERS* ermitteln.

Abbildung 3 zeigt die Entwickleransicht eines einfachen Redo-Tools. Die IBX-Komponenten auf der linken Seite realisieren den Zugriff auf die Transaction-Log-Datenbank *employee_log.gdb*. Die Zugriffskomponenten auf der rechten Seite stellen die Verbindung zur Backup-Datenbank *employee_redo.gdb* dar. Die *TIB SQL*-Komponente wird verwendet, um die generierte DML-Anweisung auf der Redo-Datenbank auszuführen.

Erstellen Sie nun eine Kopie von *employee.gdb* und benennen Sie die Datei auf *employee_redo.gdb* um. Öffnen Sie das Delphi-Projekt *LogRedo.dpr* und passen Sie erneut die Datenbankpfade an. Starten Sie das *LogRedo*-Utility. Definieren Sie nun den gewünschten Zeitraum, in dem Sie sich in den Logdaten bewegen wollen. Öffnen

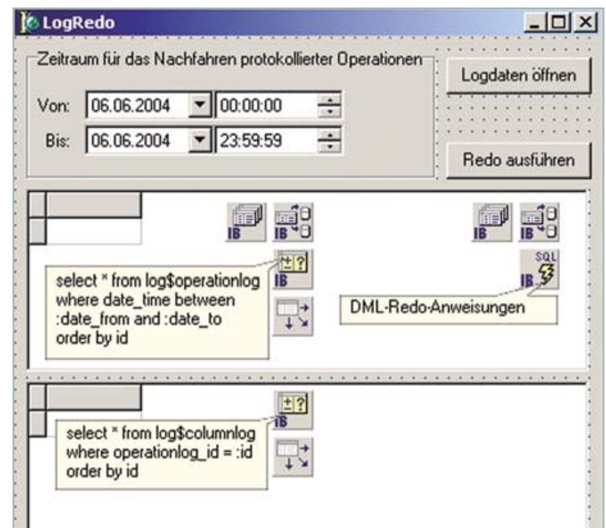

Abb. 3: Entwickleransicht – LogRedo

Sie die Logdaten (Abb. 4). Setzen Sie nun den Datensatzzeiger in der Tabelle *LOG $OPERATIONLOG* an die Position, von wo das Redo gestartet werden soll. Klicken Sie auf den Button REDO AUSFÜHREN. Es wird die Methode *DoRedo* aufgerufen, die alle protokollierten Operationen ab der Startposition sequenziell durchläuft. Für jeden Log-Eintrag wird die Methode *DoSingleRedoStatement* aufgerufen, die als Eingabeparameter den protokollierten Operationstyp (*D*, *I*, *U*) aus der Spalte *EVENT_TYPE* verwendet. Abhängig vom übergebenen Operationstyp wird die entsprechende *DELETE*-, *INSER0T*- oder *UPDATE*-Anweisung mit den Methoden *GetDeleteSQLStatement*, *GetInsertSQL -Statement* bzw. *GetUpdateSQLStatement* zusammengebaut und auf *employee_redo.gdb* ausgeführt. Beim Zusammenstellen der DML-Anweisung muss darauf geachtet werden, dass BLOB-Daten wiede-

Anzeige

Listing 3

```
procedure TfmMainForm.DoRedo;
begin
 itrEmployeeRedo.StartTransaction;
 try
  with idstOperationLog do
  begin
   while not EOF do
   begin
    DoSingleRedoStatement(FieldByName('event_type').
                                    AsString[1]);
    Next;
   end;
  end;
  itrEmployeeRedo.Commit;
 except
  itrEmployeeRedo.Rollback;
 end;
end;

procedure TfmMainForm.DoSingleRedoStatement
                              (AEventType: Char);
var
 pCount: Integer;
 aRedoStatement: String;
begin
 aRedoStatement := '';
 case AEventType of
  'D': begin
   aRedoStatement := GetDeleteSQLStatement;
   if aRedoStatement <> '' then
   begin
    isqlEmployeeRedo.SQL.Text := aRedoStatement;
    isqlEmployeeRedo.ExecQuery;
   end;
  end;
  'I', 'U': begin
   case AEventType of
    'I': aRedoStatement := GetInsertSQLStatement;
    'U': aRedoStatement := GetUpdateSQLStatement;
   end;
   if aRedoStatement <> '' then
   begin
    isqlEmployeeRedo.SQL.Text := aRedoStatement;
    if not isqlEmployeeRedo.Prepared then
                      isqlEmployeeRedo.Prepare;
    if isqlEmployeeRedo.Params.Count > 0 then
    begin
     (* In der SQL-Anweisung können auch Parameter
          vorhanden sein, z.B. umBLOB-Daten zu
                       berücksichtigen *)
     pCount := 0;
     idstColumnLog.First;
     while (not idstColumnLog.EOF) and (isqlEmployee
                Redo.Params.Count <> pCount) do
     begin
      if idstColumnLog.FieldByName('NEW_VALUE').IsNull
then
      begin
       isqlEmployeeRedo.ParamByName(idstColumnLog.
             FieldByName('COLUMN_NAME').AsString).
                  Value := idstColumnLog.FieldByName
                           ('NEW_VALUE_BLOB').Value;
       INC(pCount);
      end;
      idstColumnLog.Next;
     end;
    end;
    isqlEmployeeRedo.ExecQuery;
   end;
  end;
 end;
end;

function TfmMainForm.GetDeleteSQLStatement: String;
begin
 Result := '';
 with idstOperationLog do
 begin
  Result := Format('delete from %s where %s = %s', [
                  FieldByName('TABLE_NAME').AsString,
                  FieldByName('PK_COLUMN').AsString,
   QuotedStr(FieldByName('PK_VALUE').AsString)]);
 end;
end;

function TfmMainForm.GetInsertSQLStatement: String;

 procedure GetFieldParamList(const ATable: String; var
                AFieldList, AFieldValueList: String);
 begin
  with idstColumnLog do
  begin
   First;
   while not EOF do
   begin
    if AFieldList <> '' then AFieldList := AFieldList + ', ';
    AFieldList := AFieldList + FieldByName
                         ('COLUMN_NAME').AsString;
    if AFieldValueList <> '' then AFieldValueList :=
                            AFieldValueList + ', ';
    if not FieldByName('NEW_VALUE').IsNull then
     // Kein Parameter nötig, daher direkt zuweisen
     AFieldValueList := AFieldValueList + QuotedStr
                (FieldByName('NEW_VALUE').AsString)
    else
     // Mit Parametern arbeiten, da BLOB
     AFieldValueList := AFieldValueList + ':' + FieldByName
                         ('COLUMN_NAME').AsString;
    Next;
   end;
  end;
 end;

var aFields, aFieldsValues, aTable: String;
begin
 Result := ''; aFields := ''; aFieldsValues := '';
 aTable := idstOperationLog.FieldByName
                          ('TABLE_NAME').AsString;
 GetFieldParamList(aTable, aFields, aFieldsValues);
 if (aFields <> '') and (aFieldsValues <> '') then
  Result := Format('INSERT INTO %s (%s) VALUES (%s)',
                 [aTable, aFields, aFieldsValues]);
end;

function TfmMainForm.GetUpdateSQLStatement: String;

 procedure GetFieldParamList(const ATable: String; var
                               AFieldList: String);
 begin
  with idstColumnLog do
  begin
   First;
   while not EOF do
   begin
    if AFieldList <> '' then AFieldList := AFieldList + ', ';
    if not FieldByName('NEW_VALUE').IsNull then
     // Kein Parameter nötig, daher direkt zuweisen
     AFieldList := AFieldList + FieldByName
          ('COLUMN_NAME').AsString + '=' + QuotedStr
               (FieldByName('NEW_VALUE').AsString)
    else
     // Mit Parametern arbeiten, da BLOB
     AFieldList := AFieldList +
FieldByName('COLUMN_NAME').AsString + '=:' +
                   FieldByName('COLUMN_NAME').AsString;
    Next;
   end;
  end;
 end;

var aFields, aTable: String;
begin
 Result := ''; aFields := '';
 aTable := idstOperationLog.FieldByName
                          ('TABLE_NAME').AsString;
 GetFieldParamList(aTable, aFields);
 if (aFields <> '') then
  Result := Format('UPDATE %s SET %s WHERE %s = %s', [
   aTable,
   aFields,
   idstOperationLog.FieldByName('PK_COLUMN').
                                       AsString,
   idstOperationLog.FieldByName('PK_VALUE').
                                      AsString]);
end;
```

rum nur über eine parametrisierte Anweisung übertragen werden können. Außerdem müssen berechnete (COMPUTED BY) Felder ausgenommen werden, da sonst die DML-Anweisung fehlschlagen würde. Dies wurde hier allerdings nicht berücksichtigt, da die Tabelle *CUSTOMER* keine berechneten Felder beinhaltet. Die wichtigsten Methoden für den Redo-Mechanismus sind in Listing 3 abgebildet.

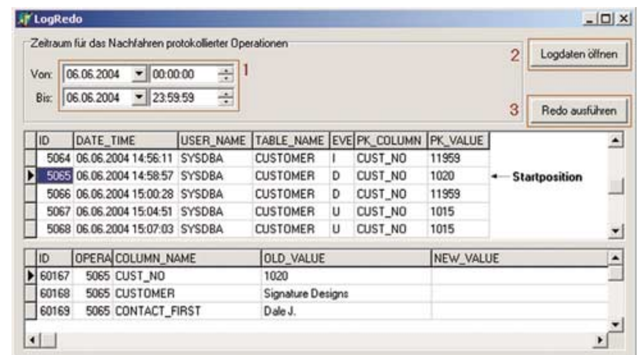

Abb.4: Log-Redo – Browsen in den protokollierten Operationen

Undo

Man könnte diese Vorgehensweise eines Redo auch in die andere Richtung durchführen. Angenommen, es wurden Änderungen durchgeführt, die über ein Commit bestätigt wurden. Diese Änderungen sollen nun zurückgenommen werden. Ein solcher Undo-Mechanismus müsste, verglichen zu unserem hier vorgestellten *Log-Redo*-Utility, die Protokolldaten in umgekehrter Reihenfolge abarbeiten, wobei die auszuführende Operation das Gegenstück zur protokollierten Operation sein muss – außer es handelt sich um eine protokollierte *UPDATE*-Anweisung, da dies beim Undo ebenfalls wieder ein *UPDATE* sein wird, jedoch mit den protokollierten alten Feldwerten.

Fazit

Sie sehen, protokollierte Datenänderungen können durchaus ihren Reiz haben, da diese vielseitig weiterverarbeitet werden können. Eine interessante Möglichkeit haben wir mit dem hier vorgestellten Roll-Forward-Log-Mechanismus gesehen. Sollten Sie auf der Suche nach einem kommerziellen Produkt in diesem Bereich sein, dann kann ich Ihnen die IB LogManager-Produktfamilie [5] ans Herz legen. Neben dem eigentlichen Protokollierungsmechanismus werden auch unterschiedliche Add-ons angeboten, welche die hier vorgestellten Konzepte umsetzen. Möchten Sie diese Thematik noch weiter vertiefen, kann ich Ihnen [6] als Lektüre ans Herz legen. Dann rollen Sie jetzt mal ordentlich vorwärts! Viel Spaß beim Ausprobieren.

Links & Literatur

[1] Thomas Steinmaurer: Die Protokollierung – Serverseitiges Auditing für InterBase, in *Der Entwickler* 3.2004
[2] www.dbginc.com/tech_pprs/IB.html
[3] bdn.borland.com/article/0,1410,29515,00.html
[4] www.ibobjects.com/
[5] www.upscene.com/
[6] firebird.sourceforge.net/index.php?op=useful&id=calford_1

Anzeige