Verdeckte Operationen

Programmierung benutzerdefinierter Funktionen für InterBase – Teil 3

von Karsten Strobel

Ein Kamel durch ein Nadelöhr bewegen, einen Raddampfer vom Amazonas über ein Bergmassiv transportieren, Aggregatfunktionen mit InterBase-UDFs programmieren... Der letzte Teil dieser Artikelserie soll natürlich nicht zu Wahnsinnstaten verleiten, aber aufzeigen, welche Möglichkeiten jenseits der Spezifikation noch in den benutzerdefinierten Funktionen stecken.

In den ersten beiden Teilen haben wir diverse Anwendungsgebiete benutzerdefinierter Funktionen ausgelotet und deren Tücken und Gefahren kennen gelernt. Außerdem wurden die Abbildung der InterBase-Datentypen auf die Funktionsparameter sowie deren korrekte Deklaration beschrieben und anhand vieler Beispiele illustriert. Alle bisherigen Beispiele haben eines gemeinsam: Es handelte sich immer um skalare Funktionen, die bei einmaligem Aufruf für einen Satz von Eingabeparametern genau einen Ausgabeparameter ausspucken.

Quellcode

Den Quellcode finden Sie auf der aktuellen Profi-CD sowie auf unserer Homepage unter

www.derentwickler.de

Dies ist nicht weiter überraschend, weil es auch genau dem Konzept der UDFs von InterBase entspricht. Für andere Funktionstypen (z.B. für Matrizenoperationen) taugen sie eigentlich nicht.

Vor allem Aggregatfunktionen lassen sich mit UDFs nicht realisieren. Die *SUM()*- oder die *COUNT()*-Funktion lässt sich damit auf keinen Fall nachprogrammieren, denn dies würde ein anderes Aufrufkonzept erfordern, worauf wir aber keinen Einfluss haben. Dennoch möchte ich eine Möglichkeit zeigen, mit der sich UDFs – speziell in Verbindung mit Stored Procedures – zur Verarbeitung von Parameterreihen nutzen lassen, womit sich die Funktionalität einer Aggregatfunktion quasi nachbilden lässt. Dabei werden wir uns allerdings „hart am Rande der Legalität" bewegen.

Nehmen wir ein Beispiel aus einem Bereich, auf den viele derzeit wahrscheinlich mit gemischten Gefühlen blicken: Der Aktienmarkt. Wenn man sich über eine der unzähligen Info-Seiten im Internet einen Aktienchart anzeigen lässt, dann sieht man meistens eine oder mehrere Kurven in die Grafik eingezeichnet, die man als „gleitender Durchschnitt" bezeichnet und die zum Beispiel aus dem Mittelwert der Kurse der letzten 30 Tage berechnet werden. Dabei wird für jeden Tag der durchschnittliche Kurs der vergangenen N Tage berechnet und grafisch eingetragen. Bei einem gleitenden Durchschnitt über 30 Tage steht für die ersten 29 Tage der Notierung kein Ergebnis zur Verfügung. Ab dem 30. Tag wird der Durchschnittswert eingezeichnet. Am 31. Tag wird der Durchschnitt der Tage 2 bis 31 gebildet und der Tag 1 wird nicht mehr berücksichtigt und so fort. Eine Berechnung dieser Art beruht also auf einer Reihe von Eingangswerten, die sich mit einer (skalaren) benutzerdefinierten Funktionen eigentlich nicht verarbeiten lässt.

Listing 1 zeigt ein Objekt, das für die fortlaufende Berechnung eines gleitenden Durchschnitts verwendet werden könnte. Mit der Methode *Learn()* werden die täglichen neuen Kurswerte eingespeist; das Funktionsergebnis zeigt an, ob schon genügend Werte für einen Durchschnittswert gesammelt worden sind. Wenn ja, kann der Durchschnittswert über die Methode *Avg()* abgeholt werden.

Wenn wir die Aktienkurse in einer Tabelle hinterlegen und für jeden Handelstag einen Datensatz speichern, dann wäre es sehr nützlich, mit Hilfe dieses Objektes den gleitenden Durchschnitt zusammen mit dem Tageskurs ausgeben zu können. Die Anzahl der Parameter einer Externen Funktion ist aber auf zehn begrenzt. Schon allein deswegen können wir nicht 30 oder mehr Werte mit einem einzigen Aufruf an die Funktion übergeben, zumal so ein Vorgehen auch sehr unpraktisch wäre.

Geheimes Wissen

Die UDF müsste also die Eingangsparameter „sammeln" können, und zwar sukzessive durch mehrmaliges Aufrufen. Dafür benötigt die Funktion aber ein „Gedächtnis", das über die Dauer der ein-

maligen Abarbeitung hinausgeht. Globale Variablen kommen nicht als Lösung in Frage, weil eine UDF (wie wir schon im Teil 1 gesehen haben) von mehreren Threads gleichzeitig aufgerufen werden kann, sodass es zu ungewollten „Kurzschlüssen" zwischen verschiedenen Sitzungen käme. Am besten lässt sich dieser Kreis durchbrechen, indem die Datensammlung in dynamisch allokiertem Speicher vorgenommen wird, auf dem während der Sitzung ein Handle (ein Zeiger) aufbewahrt und nach getaner Arbeit schließlich freigegeben wird. Wenn man sich diese Möglichkeit erschließt, spricht auch nichts dagegen, mit der Objektklasse aus Listing 1 zu arbeiten und einen Zeiger auf dieses Objekt zu verwenden. Dabei muss man gewährleisten, dass der Objektzeiger an den „Kunden" der UDF zurückgeliefert und jedes Mal wieder an die UDF übergeben wird, wenn auf die gleichen Daten zugegriffen werden soll. Da ein Zeiger nichts anderes als ein 32-Bit-Wert ist, kann dies ohne Weiteres mit einem *INTEGER*-Funktionsergebnis (ebenfalls 32 Bit) realisiert werden. Listing 2 zeigt eine benutzerdefinierte Funktion, die zum Erzeugen bzw. Freigeben einer Objektinstanz dient. Der erste Parameter gibt den Umfang der Reihe für den gleitenden Durchschnitt an (d.h. Anzahl der Tage). Der Wert 0 zeigt an, dass ein vorhandenes Objekt freigegeben werden soll. In diesem Fall steht der Zeiger im zweiten Parameter.

Listing 1: Durchschnittsberechnung als Objektklasse

```
type
 TMovAvg = class
  FData: array of Double;
  FMax: Integer;
  FCount: Integer;
  FPos: Integer;
  constructor Create(AMax: Integer);
  function IsComplete: Boolean;
  function Learn(Value: Double): Boolean;
  function Avg: extended;
 end;

constructor TMovAvg.Create(AMax: Integer);
begin
 SetLength(FData, AMax);
 inherited Create;
end;

function TMovAvg.IsComplete: Boolean;
begin
 Result := FCount = (High(FData) - Low(FData) + 1);
end;

function TMovAvg.Avg: extended;
begin
 if not IsComplete then
  raise Exception.Create('Daten nicht ausreichend');
 Result := Math.Mean(FData);
end;

function TMovAvg.Learn(Value: Double): Boolean;
begin
 if FPos > High(FData) then FPos := 0;
 FData[FPos] := Value;
 Inc(FPos);
 if not IsComplete then Inc(FCount);
 Result := IsComplete;
end;
```

Auf diese Weise lässt sich sogar ein Hauch von OOP in die Sphären von InterBase tragen. Das Funktionsergebnis des *INIT_CLEAR_MOVAVG*-Aufrufs wird zwar als ordinäre Ganzzahl definiert, ist aber in Wahrheit ein Zeiger auf die Instanz einer Objektklasse. Dieser Zeiger, den wir vielleicht treffender Handle nennen können, kann bei weiteren UDF-Aufrufen dazu verwendet werden, den Arbeitskontext zu definieren, wenn mit der Funktion *CALC_MOVAVG* (siehe Listing 3) Kurswerte übergeben werden. Als erster Parameter wird der zuvor angeforderte Handle übergeben; der zweite Parameter ist der Kurswert eines Tages. Das Funktionsergebnis ist ein Fließkommawert, der den gleitenden Durchschnitt angibt, oder 0, wenn es mangels Daten noch keinen gibt. Die Quellcodes finden Sie auch auf der Profi-CD und im Web.

Gefährlich ist diese Art der UDF-Programmierung vor allem aus dem folgenden Grund: Wenn ein instanziertes Objekt nicht mit *INIT_CLEAR_MOVAVG(0, <handle>)* wieder freigegeben wird, dann bedeutet dies ein Speicherleck. Je öfter dieser Fall eintritt, umso mehr dynamischer Speicher geht verloren. Wenn solches sehr oft hintereinander passiert, wird der Speicher irgendwann knapp und der Serverprozess, in dessen Adressraum UDFs ja ausgeführt werden, wird (gelinde gesagt) beeinträchtigt.

Wenn man die hier vorgestellte Technik verwenden möchte, empfiehlt es sich, dies innerhalb einer Stored Procedure zu tun. Dadurch lässt sich der gesamte Lebenszyklus dieser ungewöhnlichen UDF-Operationen – von der Initialisierung über die Verarbeitung bis zur Freigabe – in einen überschaubaren Rahmen fassen und von störenden Einflüssen weitgehend isolieren. Listing 4 zeigt eine solche Lösung. Nur wenn die Stored Procedure zwischen dem ersten und dem abschließenden Aufruf von *INIT_CLEAR_MOVAVG()* unterbrochen würde – zum Beispiel durch das Auftreten eines Deadlock – könnte hier noch der Fall eintreten, dass der für die Objektinstanz angeforderte Speicher nicht wieder freigegeben würde. Es gibt also bei diesem Verfahren keine hundertprozentige Sicherheit, aber das Risiko ist halbwegs überschaubar.

Es soll nicht unerwähnt bleiben, dass man die für unser Beispiel konstruierte Aufgabe wohl auch mit reinem SQL lösen könnte und die Objektimplementierung sogar bewusst umständlich gemacht wurde. Die Berechnung des gleitenden Durchschnitts dient zwar als Modell für die gezeigte Pseudo-Aggregatfunktion, ließe sich aber auch innerhalb einer Stored Procedure ohne Zuhilfenahme externer Programmierung einfach realisieren. Bei anderen, komplizierteren Anwendungsfällen kann die Methode des wiederholten UDF-Aufrufs aber die einzige Alternative zur clientseitigen Programmierung und damit einen sinnvollen Ausweg darstellen.

Listing 2: Anlegen und Freigeben der Objektinstanz

```
function Init_Clear_MovAvg(var Size, Handle: Integer):
                                        Integer; cdecl;
{
DECLARE EXTERNAL FUNCTION INIT_CLEAR_MOVAVG
                                  INTEGER, INTEGER
RETURNS INTEGER BY VALUE
ENTRY_POINT 'INIT_CLEAR_MOVAVG' MODULE_NAME
                                        'MeineUDFs';
}
begin
 if Handle <> 0 then TMovAvg(Handle).Free;
 if Size <> 0 then Result := Integer(TMovAvg.Create
                                  (Size)) else Result := 0;
end;
```

Gefährliche Versuchung

Derart auf den Geschmack gekommen, drängen sich Fragen nach weiteren Tricks zur artfremden aber vielleicht umso praktischeren Verwendung benutzerdefinierter Funktionen auf. In den einschlägigen Newsgroups wird häufig gefragt, ob man aus einer UDF heraus irgendwie Zugriff auf die Datenbank erlangen könnte, um von Fall zu Fall Daten zu lesen oder sogar verändern zu können. Die Antwort ist ein klares Nein. Es gibt zumindest keinen direkten Weg für so einen Griff nach den Daten und eine UDF verfügt auch über keinerlei Kontextinformationen, außer ihren Parametern, die der Anwender der Funktion beim Aufruf übergibt.

Eine UDF könnte zwar als Datenbankclient auf die aufrufende Datenbank zugreifen, aber dazu müsste aus der Funktionsbibliothek heraus eine „offizielle" Datenbankverbindung hergestellt werden. Weder kann die UDF hierfür den aktuellen Transaktionskontext des Aufrufers nutzen, noch steht auch nur die Information zur Verfügung, welche Datenbank gerade die Funktion verwendet. Eine derartige Client-im-Server-Lösung erlangt also keinerlei Vorteil aus der Beheimatung im InterBase-Prozess. Generell sollte man derart komplizierte Operationen sowieso nicht in einer UDF durchführen, denn mit der Stabilität und der Geschwindigkeit solcher Lösungen steht und fällt auch die Zuverlässigkeit und Performance des Servers. Ergo: Ein bisschen Tricksen ist erlaubt, aber man sollte nicht versuchen, einen Elefanten aus dem Ärmel zu zaubern.

Listing 3: Gleitenden Durchschnitt berechnen

```
Class CustomersCompanyNameControl
        Subclass Of AcTextControl
  Sub SetProperties( )
   Super::SetProperties( )
    Position.X = 900
    Position.Y = 180
    Size.Height = 362
    Size.Width = 6400
  End Sub

  'ValueExp: [Customers.CompanyName]

  Sub SetValue( row As AcDataRow )
   DataValue = row.GetValue( "Customers.CompanyName"
)
  End Sub

End Class

AddComponent (
 "NewReportApp::Frame1",
 "Content", 1,
 "NewReportApp::Frame1::CustomersCompanyName-
Control"
)
```

Von Redmond in die Antarktis

Bisher haben wir wie selbstverständlich immer Windows als Serverbetriebssystem vorausgesetzt. InterBase steht aber auch auf anderen Plattformen zur Verfügung, nämlich auf Sun Solaris und diversen Linux-Distributionen. InterBase-Datenbanken sind zwischen den Plattformen völlig stressfrei portierbar, nur UDFs sind leider ein Bremsklotz beim sorglosen Reisen zwischen den Betriebssystemwelten, denn der Bibliothekscode ist plattformspezifisch. Um die gleiche Funktionalität unter Unix/Linux zu erreichen, müssen Sie Ihre Funktionsbibliothek also für die jeweilige Plattform umwandeln.

Für Linux bietet Borland das Delphi-ähnliche Kylix an. Aufgrund der weitgehenden Kompatibilität von Delphi und Kylix sind erstaunlich wenige Änderungen am Pascal-Quellcode notwendig, um daraus mit Kylix eine .so Datei (für „Shared Object", das Linux-Pendant zur .dll) für InterBase erzeugen zu können. Falls Sie diesen Weg gehen möchten, müssen Sie für ein erfolgreiches Kompilieren mit Kylix folgende Punkte beachten:

- Verwenden Sie bedingte Kompilierung, um plattformspezifische Unterschiede zu machen: *{$IFDEF MSWINDOWS} ... {$ENDIF}* bzw. *{$IFDEF LINUX} ... {$ENDIF}*
- Entfernen Sie (mit *IFDEF.*.) die Unit „Windows" aus den Uses-Klauseln.
- Passen Sie die Dateinamen der zu ladenden Bibliotheken an: Aus *ib_util.dll* wird *ib_util.so* und aus *gds32.dll* wird *gds.so*.
- Testen Sie den Ergebniswert von *LoadLibrary('gds.so')* nicht auf < 32, sondern auf 0, um einen Fehlerfall festzustellen. Die *LoadLibrary/FreeLibrary*-Aufrufe brauchen Sie nicht zu ändern, da Kylix diese WinAPI-Funktionen in der *Sys Utils*-Unit implementiert und in Linux-Kommandos umsetzt.

Bei der Deklaration in SQL müssen Sie nichts weiter beachten, sofern Sie (wie in unseren Beispielen geschehen), für die externe Funktion als *MODULE_NAME* den Bibliotheksnamen *ohne* Namenserweiterung, also ohne „.dll" bzw „.so" angeben. In diesem Fall ergänzt nämlich InterBase automatisch die für das jeweilige Betriebssystem passende Extension.

Schlusslicht

Damit ist die Serie über InterBase-UDFs abgeschlossen und es dürfte Ihnen jetzt nicht mehr schwer fallen, Ihre Datenbankanwendung mit eigenen Funktionen zu erweitern. Dabei sollte man immer die Risiken für die Serverstabilität im Auge behalten und ausreichend Zeit auf Kontrolle und Tests – vor allem auch im Mehrbenutzerbetrieb – verwenden.

Listing 4: Anwendung der Spezial-UDF

```
CREATE TABLE KURSE (DATUM DATE, KURS NUMERIC
                                        (9, 2));
CREATE PROCEDURE TEST_MOVAVG (AVON DATE, ABIS
  DATE) RETURNS (RDATUM DATE, RKURS NUMERIC(9,2),
                    RMOVAVG DOUBLE PRECISION) AS
DECLARE VARIABLE UDF_HANDLE INTEGER;
BEGIN
 UDF_HANDLE = INIT_CLEAR_MOVAVG(7, 0);
      /* Objekt erzeugen (für 7-Tage-Durchschnitt) */
 FOR
  SELECT k.DATUM, k.KURS, CALC_MOVAVG(:UDF_
                                    HANDLE, k.KURS)
  FROM KURSE k WHERE k.DATUM BETWEEN :AVON
                                    AND :ABIS
  ORDER BY k.DATUM
  INTO :RDATUM, :RKURS, :RMOVAVG
 DO
  SUSPEND;
 UDF_HANDLE = INIT_CLEAR_MOVAVG(0, UDF_
                    HANDLE); /* Objekt freigeben */
END
```