Der Entwickler 06/1999

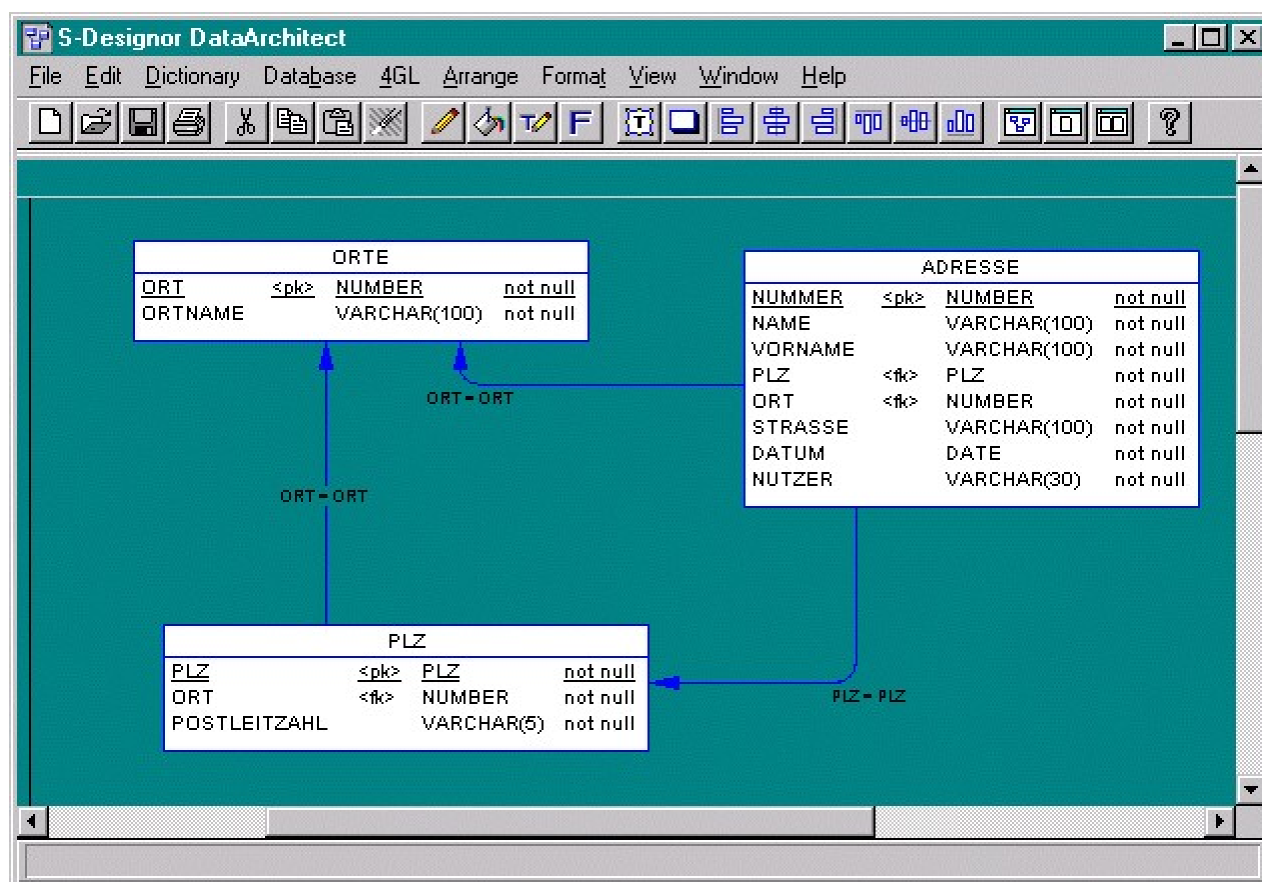
Das Oracle von Delphi I

von Henry Wolf

Viele Hersteller von Entwicklungswerkzeugen behaupten in stetiger Regelmäßigkeit das es möglich ist, mit ihren Werkzeugen Software zu entwickeln, welche auf den verschiedensten Datenbanken läuft. Theoretisch ist dies sicherlich richtig. Performante und stabile Anwendungen lassen sich jedoch nur unter Nutzung der spezifischen Funktionalitäten der jeweiligen Datenbank entwickeln. Dies gilt auch für Delphi. In diesem und einem folgenden Artikel möchten wir zeigen, wie man mit Delphi auf der Grundlage des (O)RDBMS ORACLE Software entwickelt und eine Einführung in die mächtige Funktionalität dieser Datenbank zeigen.

Um wichtige Aspekte für das Vorgehen bei der Entwicklung eines Delphi-Programmes für ORACLE darzustellen, wollen wir eine kleine Adressdatenbank entwickeln. Diese besteht aus den 3 Tabellen Adresse, Orte, PLZ (siehe Abb. 1).

Abb. 1 Beispieldatenbank



In der Tabelle Orte befindet sich ein Ortsverzeichnis. Die Tabelle PLZ repräsentiert die verfügbaren Postleitzahlen in Abhängigkeit vom Ort. In der Tabelle Adresse werden in einer einfachen Form die Adressen gespeichert, wobei für Ort und PLZ die Kennzeichen in Abhängigkeit von den Tabellen Orte und PLZ gespeichert werden. Auf die Aspekte der Entwicklung des Datenbank-Designs ist in verschiedenen Artikeln in vorherigen Ausgaben eingegangen worden. Diese sollen an dieser Stelle vernachlässigt werden.

1. Zugriff auf ORACLE via BDE

Um auf unsere ORACLE-DB via Delphi zugreifen zu können, müssen wir als erstes die BDE konfigurieren. Hierbei gibt es wie immer zwei Teile zu konfigurieren. Als erstes die Konfiguration des SQL-Links (Treiber – siehe Abb.2) und dann die Konfiguration des speziellen Datenbankzugriffs, unseres Aliases zum Zugriff auf die ORACLE-DB (siehe Abb. 3).

Abb. 2 Konfiguration des Oracle SQL-Links (Treiber)

Parameter	Oracle 7	Oracle 8
DLL32	SQLORA32.DLL	SQLORA8.DLL
Vendor Init	ORA73.DLL	OCI.DLL

Bei der Konfiguration des SQL-Links werden Einstellung der Basisverbindung zu Oracle (Definition der Variante des Zugriffes auf die OCI-Schnittstelle) und allgemeine Einstellungen, welche für alle Oracle DB's gelten, auf welche zugegriffen werden soll, durchgeführt.

Bei der Konfiguration des BDE-Aliases werden die spezifischen Einstellungen zum Zugriff auf ein bestimmtes Datenbankschema realisiert.

Abb. 3 Konfiguration des BDE-Aliases

SERVER NAME:	TNS - ALIAS zum Zugriff auf die Oracle - DB
OBJECT MODE:	Zugriff auf Objekteigenschaften in Oracle8
LIST SYNONYMS:	Zugriffsregelung auf in der DB definierte Synonyme
BLOBS TO CACHE:	Wieviele Datensätze mit Blobs (formatiert , unformatiert) sollen auf dem Client zwischengespeichert werden
ENABLE INTEGERS:	Wandelt NUMBER-Werte ohne Nachkommastellen in ORACLE-Integers um.
ENABLE BCD:	ENABLE BCD decimals (BCD)
LANGDRIVER:	Sprachtreiber zum Zugriff auf die Oracle DB

Die Eigenschaft Enable Shema Cache und die damit zusammenhängenden wurden weggelassen, da diese Parameter in Oracle 8 nicht mehr unterstützt werden.

Bei der Nutzung von Installshield oder der BDE treten oft Probleme bei der Installation auf bzw. werden durch Änderungen der Anwender auf dem PC verursacht. Dies kann man dadurch vermeiden, das man mit sogenannten dynamischen Aliasen arbeitet, sprich in der BDE keine Alias einrichtet sondern dies im Programmcode realisiert (Abb. 4). Den Alias richtet man nur für die Zeit der Entwicklung ein, ansonsten arbeitet das Programm mit den Einstellungen, welche via Programm vorgenommen werden. Dabei übergibt man bei Programmstart via eines eigenen Anmeldedialoges den Nutzernamen und das Password bzw. den Oracle-Connectstring. Um für den Anwender des Programmes die Anmeldung nicht zu verkomplizieren, können alle Angaben außer dem Password in der Registry oder einer Ini-Datei gespeichert werden und bei jeder Anmeldung als Vorgaben angezeigt werden.

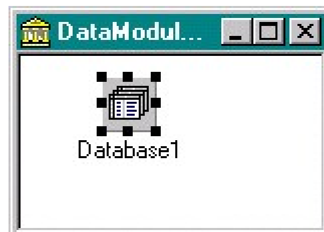
Abb. 4 Konfiguration der TDatabase-Komponente im Programmcode

```
Procedure TDatamoduleBase.PubProcConnectDB
                              ( Username:String;Password:String;Hoststring:String);
begin
  TRY
    Database1.Connected := FALSE;
    Database1.LoginPrompt := FALSE;
    Database1.Params.Clear;
    Database1.Params.Add ('USER NAME=' + UpperCase(Username) );
    Database1.Params.Add ('PASSWORD=' + UpperCase (Password) );
    Database1.Params.Add ('SERVER NAME=' + Uppercase ( HostString ));
    Database1.Params.Add ('LANGDRIVER=Borland DEU Latin-1' );
    Database1.Params.Add ('BLOBS TO CACHE= 128' );
    Database1.Connected := TRUE;
  EXCEPT
  ON A:EDBEngineError
  do begin
          ExceptionWithoutTransaction ( A ); { eigenes Exceptionhandling }
         raise;
        end;
  END;
END;
```

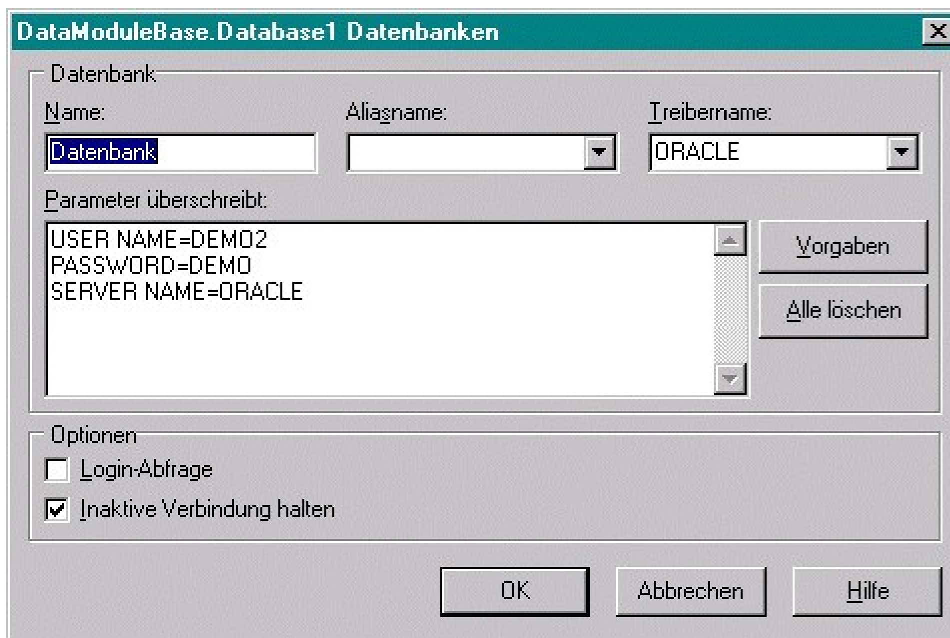
2. TDatabase und Table versus Query

Um die Verbindung zu unserer Datenbank herzustellen, benötigen wir eine TDatabase-Komponente, welche es gestattet eine Verbindung zu unserer 1. Datenbank herzustellen (siehe Abschnitt 1).

Haben wir die TDatabase-Komponente konfiguriert, können wir nun auf Objekte in unserer Oracle-DB zugreifen.



Der Zugriff erfolgt wie allgemein bei SQL-Datenbanken via SQL und der Komponente TQuery. TTable eignet sich nicht, da bei TTable sogenanntes lokales SQL (für dBase und Paradox u.a.) genutzt wird, welches bei der Ausführung von Abfragen zu einem enormen Overhead führt und die Nutzung der zusätzlichen SQL-Funktionalität der SQL-Datenbank verhindert. Wer sich dies exakt anschauen will, kann dies über den SQL-Monitor tun. Er nehme ein TTable und eine TQquery (SELECT * FROM TABELLE) und schalte diese beiden Objekte auf aktiv=true. Dann sieht man deutlich, weshalb man TQuery bei SQL-Datenbanken nutzen sollte.



3. Views

Views stellen dem Entwickler eine aufbereitetet Sicht auf die Daten zur Verfügung. Aber nicht nur dies wird von einem View geleistet, sondern ein View stellt auch einen Weg zur Verbesserung der Performance von Datenbank-Abfragen dar. SQL-Statements werden in 3 Phasen abgearbeitet:

1. Parse - Vorbereitung zur Ausführung des SQL-Statements (Execute-Plan zusammenstellen e.t.c.)
2. Execute - Ausführung des SQL-Statements
3. Fetch - Übermittlung der Daten an den Clienten.

Ein View stellt ein bereits geparstes SQL-Statement dar, das angesprochen werden kann wie eine Tabelle. Für unser Beispiel wäre z.B. folgender View sinnvoll, welcher die Adresse in einer für die Anwendersicht aufbereiteten Form darstellt:

```
CREATE OR REPLACE VIEW V_ADRESSE_1
AS
SELECT UPPER(A.NAME)||','||UPPER(A.VORNAME) "NAME" ,
                    B.POSTLEITZAHL "POSTLEITZAHL" ,
                    C.ORTNAME "ORTBEZEICHNER" ,
                    A.STRASSE "STRASSE"
FROM ADRESSE A , PLZ B , ORTE C
WHERE A.PLZ = B.PLZ
AND A.ORT = C.ORT
```

Auf diesen View kann jetzt wie auf eine Tabelle zugegriffen werden, z.B.:

SELECT NAME,VORNAME,POSTLEITZAHL, ORTBEZEICHNER,STRASSE FROM V_ADRESSE_1 WHERE SUBSTR(NAME,1,3) = ‚KAM';

4. Sequences

Sequences bieten die Möglichkeit, sogenannte autoincrement-Werte zu generieren. Dies ist besonders bei der Generierung von Primärindexwerten hervorragend geeignet. Angelegt werden diese Sequences via SQL. Für unsere Beispielanwendung benötigen wir 3 Sequences zur Primärindexbildung:

```
Tabelle Adresse :
CREATE SEQUENCE S_ADRESSE START WITH 1 INCREMENT BY 1;
Tabelle ORT :
CREATE SEQUENCE S_ORT START WITH 1 INCREMENT BY 1;
Tabelle PLZ
CREATE SEQUENCE S_PLZ START WITH 1 INCREMENT BY 1;
```

Beim Insert in die Tabelle Adresse ist bei der Nutzung der Sequence wie folgt zu verfahren:

INSERT INTO ORT (ORT , ORTNAME) VALUES (S_ORT.NEXTVAL , ‚NORDHAUSEN'
);

Mittels der "Methode" nextval der Sequence wird automatisch der nächste Wert der Zählfolge abgerufen und vergeben. Dieser Wert steht weder außerhalb noch innerhalb der aktuellen Transaction zur Verfügung sondern ist unwiederbringlich weg. Dies ist besonders in Multiuserumgebungen wichtig.

Natürlich stellt sich die Frage, was tue ich, wenn ich sogenannte request – Live Abfragen zur Eingabe von Werten nutze. Wie komme ich an die Werte der Sequence. Hierbei kommt die Methode OnNewRecord der Querykomponente zur Anwendung.

Neben der eigentlichen Haupt-TQuery , wird eine zweite TQuery (SQL = ‚SELECT S_ORTE.NEXTVAL FROM DUAL'). eingebunden, welche beim Einfügen eines neuen Datensatzes den aktuellen Sequencwert holt.

Dann wird bei onNewRecord das Primärindexfeld vorbelegt.

```
procedure TFormSequence.Query1NewRecord(DataSet: TDataSet);
begin
   inherited;
   Query2.Close;
   Query2.Open;
   Query1.FieldByName ('ORT').AsInteger := trunc( Query2.FieldByName ('NEXTVAL').AsFloat );
   Query2.Close;
end;
```

5. Sequences und Trigger

Ein wesentlich einfacherer Weg ist die Vergabe von Primärindizes via Trigger. Trigger sind PL/SQL-Programmmodule, welche beim ändernden Zugriff auf eine Tabelle (INSERT,UPDATE,DELETE) automatisch vorher oder danach ausgeführt werden. PL/SQL ist die mächtige procedurale Erweiterung von SQL. Von der Syntax her kommt PL/SQL Pascal ziemlich nahe. Eigentlich kann man sagen, PL/SQL ist Pascal + SQL. Unser Trigger für dieTabelle PLZ müßte wie folgt aussehen:

```
CREATE OR REPLACE TRIGGER T1_PLZ
BEFORE INSERT
ON PLZ
FOR EACH ROW
BEGIN
   SELECT S_PLZ.NEXTVAL INTO :NEW.PLZ FROM DUAL;
END;
/
```

Jetzt muß nicht mehr in Delphi die Vergabe des Primärindizes erfolgen, sondern dies geschieht auf Datenbankebene. Neben der einfacheren Programmierung kommt noch ein wesentlicher Aspekt hinzu. Wir müssen immer damit rechnen, das auch andere Programme auf unsere Datenbank zugreifen. Steuern wir die Primärindexvergabe bzw. die Vorbelegung von Feldern via Programm, ist es dem Fremdprogramm möglich, unsere Vorbelegungen zu umgehen. Haben wir diese Vorbelegung via Trigger implementiert, ist dies nicht möglich. Als Beispiel hierfür soll bei der Tabelle Adresse immer der Nutzer, welcher als letzter den Datensatz geändert hat und das Änderungsdatum automatisch via Trigger protokolliert werden. Unser Trigger müßte wie folgt aussehen:

```
CREATE OR REPLACE TRIGGER T1_ADRESSE
BEFORE INSERT OR UPDATE
ON ADRESSE
FOR EACH ROW
BEGIN
   SELECT S_ADRESSE.NEXTVAL INTO :NEW.NUMMER FROM DUAL;
   :NEW.NUTZER := USER;
   :NEW.DATUM := SYSDATE;
END;
```

```
/
```

Dabei werden die vordefinierten Oracle-Funktionen User (aktueller Nutzer) und Sysdate (Systemdatum) verwendet. Kein Nutzer kann nun Datensätze einfügen oder ändern ohne vermerkt zu werden.

6. Storage Functions

Die Syntax von Oracle SQL ist eine der mächtigsten SQL-Implementierungen. Neben sehr vielen besonderen System-Funktionen, welche via SQL gehändelt werden können, verfügt ORACLE-SQL im Vergleich zu Interbase und vielen anderen Datenbanken über einen unerschöpflichen Vorrat an Funktionen, welche in SQL-Statements verwendet werden können. Wer jedoch eigene Funktionen implementieren will, und dies lohnt sich in vielen Fällen, kann dies mittels SQL realisieren. In unserem Beispiel wollen wir eine einfache Funktion zur Formatierung der Ausgabe unserer Adresse schreiben.

```
CREATE OR REPLACE FUNCTION FUNC_FORMAT_NAME ( VORNAME VARCHAR,NACHNAME VARCHAR )
RETURN VARCHAR
IS
BEGIN
   RETURN ( UPPER(NACHNAME)||', '||UPPER(VORNAME) );
END;
```

Diese Funktion können wir jetzt immer zur Ausgabe des Namens in diesem Format nutzen. Z.B. in unserem View:

```
CREATE OR REPLACE VIEW V_ADRESSE_2
AS
SELECT FUNC_FORMAT_NAME ( A.VORNAME , A.NAME ) "NAME",
             B.POSTLEITZAHL "POSTLEITZAHL" ,
             C.ORTNAME "ORTBEZEICHNER" ,
             A.STRASSE "STRASSE"
FROM ADRESSE A , PLZ B , ORTE C
WHERE A.PLZ = B.PLZ
AND A.ORT = C.ORT
```

Jetzt wird der Name immer in der Form "YOUNG.NEIL" ausgegeben. Aus Sicht von Delphi ergibt sich in diesem Beispiel ein Problem. Obwohl die Funktion einen Varchar < 255 zurück gibt, gibt die BDE das formatierte Namensfeld im Ergebnis der Funktion als Memo zurück. Das allein ist noch kein Problem. Es gilt hierbei aber des BDE-Parameters **BLOBS TO CACHE**zu gedenken. Dieser muß entsprechend eingestellt werden, da sonst z. B. beim Scrollen im Grid die Meldung "ungültiges Blobhandle" erscheint. Für diesen Anwendungsfall ist also klar der View V_ADRESSE_1 günstiger. Effizient lassen sich Funktionen insbesondere für Berechnungen und viele andere Zwecke implementieren.

7. Storage Procedures - PL/SQL-Packages

Der Umfang der serverseitigen Implementierung von Anwendungsfunktionalität läßt sich zusätzlich noch durch sogenannte Storage Proceduren bzw. Packages (Procedure – und Funktionsbibliotheken) erweitern. Dabei kann der volle Umfang der Möglichkeiten von PL/SQL, welcher dem Umfang von Pascal in nichts nachsteht, benutzt werden. Wann ich Code auf dem Server implementiere, hängt davon ab, was will ich machen. Ein sinnvolles Beispiel ist sicherlich das Löschen eines Ortes aus unserer Datenbank. Bevor ich einen Ort löschen möchte, muß ich auf Grund der relationalen Verknüpfung alle entsprechenden Datensätze aus der Tabelle Adresse und PLZ löschen. Die entsprechende Storage Procedure sieht wo folgt aus:

```
CREATE OR REPLACE PROCEDURE PROC_ERASE_ORT ( VAR_ORT NUMBER )
IS
BEGIN
  DELETE FROM ADRESSE WHERE ORT = VAR_ORT;
  DELETE FROM PLZ WHERE ORT = VAR_ORT;
  DELETE FROM ORTE WHERE ORT = VAR_ORT;
END;
/
```

Statt 3 TQuery – Komponenten benötige ich jetzt nur noch eine TStoredProc-Komponente um diese von

Delphi auszuführen:

Der entsprechende Delphi-Quelltext zur Ausführung der Procedure sähe wie folgt aus:

```
procedure TFormStorageProc.SpeedButton2Click(Sender: TObject);
begin
   inherited;
   StoredProc1.ParamByName ( 'VAR_ORT' ).AsInteger := RunQuery.FieldByName ('ORT' ).AsInteger;
   StoredProc1.ExecProc;
end;
```

Zu erwähnen wäre noch, das man Proceduren aus einer Bibliothek (Package) bei dem Property StoredProcName der TStoredProc-Komponente mit Packagename.Procedurename zu referenzieren hat. Klappt man die Combobox bei StoredProcName der TStoredProc-Komponente auf, erscheinen diese nicht, sondern man muß sie kennen. Dies gilt auch innerhalb von PL/SQL. Sollen Proceduren aus einem Oracle-Standard-Package genutzt werden, gilt diese Form der Referenzierung von Proceduren ebenfalls (siehe Beispiel zu Abschnitt 9).

8. Hinweise zur Implementierung von ASCII Import/Export-Routinen

Häufig ist es in Programmen erforderlich einen Datenaustausch via ASCII-Dateien zu realisieren. Um Daten im ACII-Format aus-/bzw. einzuspielen werden meist kleine Delphi-Programme entwickelt. Erfahrene Oracle-Entwickler nutzen den SQL-Loader. Der Austausch via eines Delphi-Programmes ist meist langsam. Bei Änderungen muß eine neue Version des Programmes ausgeliefert werden. Der SQL-Loader ist oft für unerfahrene aber auch erfahrene ORACLE-Entwickler ein Buch mit 7 Siegeln. Hat man sich einmal mit PL/SQL beschäftigt, bietet sich eine weitere und sehr effiziente Möglichkeit solche Austauchprozesse zu implementieren und zu automatisieren. Seit der Version ORACLE 7.3 existiert das PL/SQL-Package UTL_FILE, mit welchem diese Austauschroutinen datenbankbasierend als PL/SQL-Procedure implementiert werden können. Diese können dann normal via TStoredProc von einem Delphi-Programm aus gestartet werden oder mittels des Enterprise-Manager von ORACLE als JOB gestartet und gesteuert werden. Bei Änderungen der Austauschformate kann via SQL einfach ein neuer Procedure-Quelltext eingespielt werden und es müssen keinerlei Anpassungen an irgendwelchen Programmodulen realisiert werden.

Funktionen des Package UTL_FILE

Funktion	Beschreibung
FOPEN	ASCII-Datei öffnen
IS_OPEN	Prüfung, ob Datei offen ist
FCLOSE	ASCII-Datei schließen
FCLOSE_ALL	Alle geöffneten Dateien schließen
GET_LINE	Zeile aus ASCII-Datei lesen
PUT	Zeile in ASCII-Datei schreiben
PUT_LINE	Zeile in ASCII-Datei schreiben mit Zeilenumbruch
PUTF	Formatiertes schreiben einer Zeile in eine ASCII-Datei
NEW_LINE	Schreibt Zeilenumbruch in ASCII-Datei
FFLUSH	Physikalisches "Festschreiben" einer ASCII-Datei auf Platte

Wichtig ist, das in der Init.ORA der Instance via des Parameters UTL_FILE_DIR = C:\Export. der Datenbank erlaubt wird, aus den Verzeichnissen auf dem Server zu lesen oder zu schreiben. Auf der beigefügten CD befindet sich ein kleines Beispiel für die Realisierung eines solchen PL/SQL-Jobs entsprechend unserem kurzen Beispiel.

```
PROCEDURE EXPORT_ORTE
IS
  F UTL_FILE.FILE_TYPE; /* Filehandle */
  CURSOR C1 IS SELECT * FROM ORTE; /* Query */
  C1VAR C1%ROWTYPE; /* Datensatzrecord */
  UTL_EXPORT_AFA_ERROR EXCEPTION;
BEGIN
  /* Datei öffnen */
  F := UTL_FILE.FOPEN ( 'C:\EXPORT' , 'EXPORT.ASC' , 'W' );
  /* Query öffnen */
  OPEN C1;
  LOOP
    /* Datensatz lesen */
    FETCH C1 INTO C1VAR;
    EXIT WHEN C1%NOTFOUND;
    /* Zeile schreiben */
    UTL_FILE.PUT_LINE ( F , C1Var.ORT||';'||C1Var.ORTNAME );
  END LOOP;
  /* Query schliessen '/
```

```
 CLOSE C1;
 /* ASCII-Datei schliessen */
 UTL_FILE.FCLOSE ( F );
/* Fehlerhandling */
EXCEPTION
WHEN UTL_FILE.INVALID_FILEHANDLE
THEN DBMS_OUTPUT.PUT_LINE ( 'FILEHANDLE NICHT GÜLTIG' );
        RAISE UTL_EXPORT_AFA_ERROR;
WHEN UTL_FILE.INVALID_PATH
THEN DBMS_OUTPUT.PUT_LINE ( 'VERZEICHNIS UNGÜLTIG ODER NICHT VERFÜGBAR');
        RAISE UTL_EXPORT_AFA_ERROR;
WHEN UTL_FILE.INVALID_MODE
THEN DBMS_OUTPUT.PUT_LINE ( 'DATEI IM FALSCHEN MODUS GEÖFFNET' );
        RAISE UTL_EXPORT_AFA_ERROR;
WHEN UTL_FILE.INVALID_OPERATION
THEN DBMS_OUTPUT.PUT_LINE ( 'UNGÜLTIGE OPERATION MIT DATEI' );
        RAISE UTL_EXPORT_AFA_ERROR;
WHEN UTL_FILE.WRITE_ERROR
THEN DBMS_OUTPUT.PUT_LINE ( 'SCHREIBVERSUCH IN DATEI GESCHEITERT' );
        RAISE UTL_EXPORT_AFA_ERROR;
WHEN OTHERS
THEN DBMS_OUTPUT.PUT_LINE ( 'FEHLER während des Exports' );
        RAISE UTL_EXPORT_AFA_ERROR;
END;
```

9. Transactionhandling

Bezüglich des Transaction-Managements hat man prinzipiell drei Möglichkeiten:

1. Keine explizite Transaktionssteuerung
   Nach jeder Änderungsaktion (Query1.Post) erfolgt im Normalfall automatisch ein Commit.
2. Transaktionssteuerung von Delphi aus
   Hierbei werden die Methoden StartTransaction,Commit und Rollback der TDatabasekomponente benutzt.

```
Procedure TDataModulebase.Insert;
begin
   try
     Datamodulebase.StartTransaction;
     Datamodulebase.Commit;
   Except
   On A:EDBEngineError
   DO Datamodulebase.Rollback;
   end;
end;
```

- Transaktionssteuerung per Oracle SQL bzw. PL-SQL

```
CREATE OR REPLACE PROCEDURE PROC_ERASE_ORT ( VAR_ORT NUMBER )
IS
BEGIN
 COMMIT;
 DELETE FROM ADRESSE WHERE ORT = VAR_ORT;
 DELETE FROM PLZ WHERE ORT = VAR_ORT;
 DELETE FROM ORTE WHERE ORT = VAR_ORT;
 COMMIT;
EXCEPTION
WHEN OTHERS
THEN ROLLBACK;
        RAISE_APPLICATION_ERROR ( -20000 , 'FEHLER IN PL/SQL!!' );
```

```
END;
/
```

Ist man als Delphi-Entwickler bereit, sich dem Thema PL/SQL zu stellen bzw. andere Funktionalitäten von ORACLE zu nutzen, lassen sich viele Dinge sowohl für den Entwickler als auch für den Anwender effizient gestalten und auch für sehr große Datenmengen echt performante Anwendungen mit Delphi oder anderen Werkzeugen entwickeln. Zur Programmierung in PL/SQL existieren mittlerweile einige preiswerte Werkzeuge (z.B. www.sdctec.net), welche die Programmierung in PL/SQL einfach gestalten.

In einem weiteren Artikel in einem der nächsten Hefte werden wir auf weitere Aspekte der Programmierung gegen Oracle, z.B. die Besonderheiten der neuesten Version des (O)RDBMS ORACLE (Nestedt Tables, Array's u.a.) und deren Nutzung durch Delphi-Applikationen, eingehen.