ABIS NIEUWSBRIEF VOOR DB2 PROFESSIONALS

JAARGANG 7 - NUMMER 3, OKTOBER 2011

OPEN CURSOR

Deze maal hebben we voor u een themanummer samengesteld over "SQL PL", de SQL-component die u toelaat om procedurele logica te verweven met DML, bijvoorbeeld om stored procedures te bouwen. Vermits u ondertussen al een poosje met DB2 9 voor z/OS (of misschien al versie 10) aan de slag bent, en/of DB2 op LUW gebruikt, hetzij als DBA, designer, ontwikkelaar of als tester: de hoogste tijd om wat meer te weten te komen over de interessante mogelijkheden die door SQL PL.geboden worden.
Bekijk dit nummer als een voorsmaakje op de cursussen over SQL PL, triggers, stored procedures en user defined functions die bij ABIS op het programma staan!

Veel leesplezier,

Het ABIS DB2-team.

IN DIT NUMMER:

3

CLOSE CURSOR

Zoals reeds beloofd in het vorige nummer, krijgt u volgende keer deel twee van de bijdrage over temporele data in DB2 10; ook wordt de reeks "DB2 versus ..." voortgezet met een vergelijking tussen DB2 en SQL Server voor wat betreft scalaire functies.

SQL PL en PL/SQL – What's in a name?

Kris Van Thillo (ABIS)

Een definitie geven van SQL PL en PL/SQL (en eigenlijk ook van bijvoorbeeld T-SQL) is zeer gemakkelijk.

Inderdaad, het zijn typisch database server-based programmeertalen, die standaard SQL aanvullen met procedurele uitbreidingen en het aldus mogelijk maken om variabelen te gebruiken (en zelf te definiëren) en selectieconstructies (if, case), iteratieconstructies en exception handling te implementeren. Deze talen worden gebruikt voor het schrijven van server-side opgeslagen procedures, functies, triggers en client-side SQL blokken.

Tot zover de overeenkomst tussen beiden!

Een korte geschiedenisles.

PL/SQL is ontstaan als een variant op ADA/Pascal; de typische 'blockstructuur' (BEGIN...END-blokken) eigen aan deze omgevingen vind je in PL/SQL inderdaad makkelijk terug. De eerste versie van PL/SQL ziet het licht in Oracle v6 -- we schrijven 1988. Elke nieuwe release van Oracle voorziet in een uitbreiding van de PL/SQL 'feature set'.

SQL PL is voor het eerst geïmplementeerd in DB2 versie 7, beschikbaar vanaf 2001. De taal is gebaseerd op de ANSI/ISO/IEC 'SQL Persistent Stored Modules (SQL/PSM) language standard' (ANSI/ISO/IEC 9075-4:1999). Nieuwe DB2 releases implementeren telkens nieuwe onderdelen van deze standaard.

SQL PL is één van de meer volledige SQL/PSM implementaties (naast deze van PostgreSQL en MySQL). Belangrijke database vendors als Oracle en Microsoft met 'oudere' implementaties van een server-based programeertaal én een grote code-base hechten veel minder belang aan SQL/PSM compatibiliteit. Oracle geeft aan dat PL/SQL de mogelijkheden biedt van SQL/PSM ('functionally equivalent'), echter met een verschillende instructieset (andere instructies, instructievolgorde en/of spelling).

En dus - inderdaad - verschillen!

In wat volgt, *Code fragment 1 - SQL PL,* nemen we een standaard DB2 stored procedure zoals door IBM als 'sample' voorzien, en doen een poging deze te converteren naar PL/SQL. Na het aanmaken van de nodige tabellen in Oracle pogen we in een aantal compilatiestappen de procedure in Oracle aan de praat te krijgen.

Het resultaat, *Code fragment 2 - PL/SQL,* geeft aan wat minimaal moet gebeuren om de procedure te kunnen compileren. Minimaal, want een typische Oracle PL/SQL ontwikkelaar zou PL/SQL anders coderen, vooral gegeven de blokstructuur eigen aan PL/SQL.

Code fragment 1 - SQL PL

```
CREATE PROCEDURE IN_PARAMS (IN lowsal DOUBLE, IN medsal DOUBLE,
                            IN highsal DOUBLE, IN department CHAR(3),
                            OUT newsal DOUBLE)
DYNAMIC RESULT SETS 0
DETERMINISTIC
LANGUAGE SQL
MODIFIES SQL DATA
BEGIN
  /* declaratie van variabelen */
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE errorLabel CHAR(32) DEFAULT '';
  DECLARE v_firstnme VARCHAR(12);
  DECLARE v_midinit CHAR(1);
  DECLARE v_lastname VARCHAR(15);
  DECLARE v_salary DOUBLE;
  DECLARE at_end SMALLINT DEFAULT 0;

  /* declaratie (definitie) van cursors */
  DECLARE c1 CURSOR FOR
    SELECT firstnme, midinit, lastname, CAST(salary AS DOUBLE)
    FROM employee
    WHERE workdept = department
    FOR UPDATE OF salary;

  /* declaratie (definitie) van exception handles */
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET at_end = 1;

  DECLARE EXIT HANDLER FOR SQLEXCEPTION
    SIGNAL SQLSTATE value SQLSTATE SET MESSAGE_TEXT = errorLabel;

  /* het eigenlijke programma */
  SET errorLabel = 'OPEN CURSOR';
  OPEN c1;
  SET errorLabel = 'FIRST FETCH';
  FETCH c1 INTO v_firstnme, v_midinit, v_lastname, v_salary;

  WHILE (at_end = 0) DO
    IF (lowsal > v_salary)      THEN    SET newsal = lowsal;
    ELSEIF (medsal > v_salary)  THEN    SET newsal = medsal;
    ELSEIF (highsal > v_salary) THEN    SET newsal = highsal;
    ELSE                                SET newsal = salary * 1.10;
    END IF;
    SET errorLabel = 'UPDATE IN WHILE LOOP';
    UPDATE employee   SET salary = newsal   WHERE CURRENT OF c1;
    SET errorLabel = 'FETCH IN WHILE LOOP';
    FETCH c1 INTO v_firstnme, v_midinit, v_lastname, v_salary;
  END WHILE;

  SET errorLabel = 'CLOSE CURSOR';
  CLOSE c1;
END
```

Code changes.

Welke wijzigingen werden aan de originele source-code aangebracht (minimalistische benadering) om de SQL PL code door een Oracle PL/SQL compiler te laten uitvoeren? We doen een poging e.e.a. te bundelen in een aantal thematische groepen:

- *DDL:* het is in Oracle efficiënter om tijdens de ontwikkeling van een procedure de 'create or replace' instructie te gebruiken; het 'as' keyword moet worden toegevoegd om de DDL te scheiden van de PL/SQL 'code' en vervangt bij stored procedures de initiële 'declare' instructie;

- *Blokstructuur*: de instructies 'declare', 'begin', 'exception', en 'end' worden normaal gebruikt om een PL/SQL programma in logische blokken in te delen, voor respectievelijk: de declaratie van variabelen, constanten, exceptions, ...; de uit-te-voeren code; en de definitie van uitzonderingsroutines. Blokken worden standaard zeer vaak genest: definities aangemaakt op een bepaald niveau zijn lokaal voor dat niveau, en worden default op dat niveau geïnterpreteerd. Lukt dit niet, dan wordt stapsgewijs een hoger niveau bij de interpretatie betrokken. Dit laatste geldt ook in SQL PL, maar daar wordt minder dikwijls met BEGIN...END-blokken en locale variabelen gewerkt;

- *Excepties*: SQL PL 'exit handlers' zijn in Oracle geïmplementeerd als excepties. Een aantal zijn standaard beschikbaar en worden dus niet vooraf gedeclareerd (bijvoorbeeld no_data_found, too_many_rows, of catch-all others...); maar het staat de ontwikkelaar vrij eigen excepties aan te maken waar nodig;

- *Inputparameters*: Oracle hanteert een andere parameter definitie syntax wat betreft input/output parameters (positie van het IN, OUT en INOUT keyword). Merk trouwens op dat Oracle geen INOUT gebruikt. Parameters moeten in Oracle bovendien 'unconstrained' zijn: bij CHAR staat dan ook geen lengte-indicatie; dit is niet mogelijk met SQL PL;

- *Datatypes*: Oracle en DB2 hanteren verschillende datatype definities. Alhoewel de meeste DB2 datatypes door Oracle worden herkend en geïnterpreteerd, is dit niet het geval voor 'double' -- deze werd hier vervangen door number zonder precisie;

- *Toekenningen*: Het toewijzen van waarden aan variabelen in Oracle vereist de ':=' instructie; 'set' is niet beschikbaar; omgekeerd kan ':=' niet gebruikt worden bij SQL PL;

- *Instructies*: Wat instructies betreft zijn de verschillen tussen beide 'talen' aanzienlijk. Het 'declare' is een 'block' instructie en wordt dus slechts éénmaal opgegeven per block; cursordeclaratie gebruikt de 'cursor is' syntax. Merk op dat de 'while' instructie andere syntax gebruikt, en dat ook 'elseif' in het codevoorbeeld moet vervangen worden door 'elsif'. Ook andere, niet in het voorbeeld opgenomen instructies, zijn in SQL PL en PL/SQL niet identiek.

- *Commentaar*: SQL PL gebruikt '/* ... */' als comment delimiters, terwijl in PL/SQL gebruik gemaakt wordt van '--' (begin comment, tot einde regel).

Abstractie.

Om de 'conversiestap' zo minimalistisch mogelijk te houden hebben we een aantal belangrijke aandachtspunten bewust niet bekeken (vaak punten die ook buiten de PL/SQL context belangrijk zijn), bijvoorbeeld:

- datatypes: zijn de door Oracle gehanteerde datatypes correct/optimaal? Hoe interpreteert Oracle bijvoorbeeld het externe, niet gekende DB2 datatype 'int'? En wat met het DB2 datatype varchar dat overeenkomt met het Oracle datatype varchar maar semantisch 'mapt' op varchar2? De problematiek van 'blank padding/trimming' moet eigenlijk ook behandeld worden.
- 'null' handling.

Voor een gedetailleerde discussie omtrent de belangrijkste verschillen tussen Oracle en DB2 verwijzen we u graag naar onze cursussen!

Besluit!

SQL PL code en PL/SQL code is niet zonder meer uitwisselbaar -- het is andere code, op andere regels gestoeld, en vertrekkende van andere, specifieke uitgangspunten. Het hoeft daarom ook niet te verwonderen dat DB2 voor LUW sinds versie 9 naast 'native' SQL PL ook 'native' PL/SQL ondersteunt: want migratie/conversie van PL/SQL naar SQL PL of omgekeerd is echt niet evident.

Hiertoe bevat DB2 voor LUW twee onafhankelijke compilers, die los van mekaar virtuele 'instructies' genereren voor de DB2 SQL Unified Runtime Engine. Code wordt dus niet vertaald, en wordt als PL/SQL in de DB2 UDB cataloog opgeslagen. Een voordeel voor ontwikkelaars met PL/SQL ervaring, die verder kunnen blijven ontwikkelen in PL/SQL. En zoals te verwachten: procedures aangemaakt in SQL PL kunnen zonder problemen procedures oproepen die in PL/SQL gecodeerd zijn.

Werk je dus in een gemengde Oracle - DB2 LUW omgeving, en moet je met procedurele logica in de database aan de slag, dan ligt de conclusie voor de hand: PL/SQL rules, toch?

Code fragment 2 - PL/SQL

```
CREATE or REPLACE PROCEDURE IN_PARAMS (lowsal IN number, medsal IN number,
                                       highsal IN number, department IN CHAR,
                                       newsal OUT number)
as
BEGIN
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  errorLabel CHAR(32) DEFAULT '';
  v_firstnme VARCHAR(12);
  v_midinit CHAR(1);
  v_lastname VARCHAR(15);
  v_salary number;

  CURSOR c1 IS
    SELECT firstnme, midinit, lastname, CAST(salary AS number)
    FROM employee
    WHERE workdept = department
    FOR UPDATE OF salary;

BEGIN
  errorLabel := 'OPEN CURSOR';
  OPEN c1;
  errorLabel := 'FIRST FETCH';
  FETCH c1 INTO v_firstnme, v_midinit, v_lastname, v_salary;

  WHILE C1%FOUND loop
    IF (lowsal > v_salary)     THEN    newsal := lowsal;
    ELSIF (medsal > v_salary)  THEN    newsal := medsal;
    ELSIF (highsal > v_salary) THEN    newsal := highsal;
    ELSE                               newsal := salary * 1.10;
    END IF;
    errorLabel := 'UPDATE IN WHILE LOOP';
    UPDATE employee   SET salary = newsal   WHERE CURRENT OF c1;
    errorLabel := 'FETCH IN WHILE LOOP';
    FETCH c1 INTO v_firstnme, v_midinit, v_lastname, v_salary;
  END loop;

  errorLabel := 'CLOSE CURSOR';
  CLOSE c1;

EXCEPTION
  EXCEPTION
   WHEN OTHERS THEN
   raise_application_error
            (-20001,'Error - '||SQLCODE||' - '||SQLERRM||' - '||errorLabel );
ENDEND;
end;
```

Aan de slag met SQL PL in DB2

Peter Vanroose (ABIS)

SQL PL: wat en hoe?

SQL is een geïntegreerde taal voor het ondervragen en beheren van relationele databases, met consistente syntax voor zowel DML (data manipulatie: SELECT, INSERT, UPDATE, DELETE,...), DDL (data-definitie: CREATE/ALTER/DROP) en DCL (data contol: GRANT/REVOKE, COMMIT/ROLLBACK, ...). Naast deze drie aspecten van de taal is er ook een (misschien minder gekende) procedurele component: de "procedural language" of "PL". Jammer genoeg is de standaardisatie van deze component (SQL/PSM) veel later op gang gekomen dan voor DML, zodat er (vooral syntactisch) nog steeds grote verschillen zijn tussen wat DB2 aanbiedt (SQL PL) en wat andere relationele databases (i.h.b. Oracle met PL/SQL en SQL Server met Transact-SQL) voorzien.

DB2-ondersteuning voor procedurele SQL-extensies bestaat sinds versie 7 (ondertussen toch al 10 jaar), maar is nooit echt populair geweest, vooral niet op z/OS, om verschillende redenen:

- Geen van de DB2 front-end interfaces (zoals SPUFI en QMF voor z/OS, of Data Studio en de commando-interface voor LUW) ondersteunt rechtstreeks de SQL PL-syntax: deze kan enkel gebruikt worden in de body van een stored procedure (SP), een user-defined function (UDF, althans op LUW en met DB2 10 op z/OS), of een trigger.
- Bovendien is het op z/OS pas sinds versie 9 mogelijk om SPs echt "native" in SQL PL te schrijven: voorheen was er een C-compiler nodig op z/OS om van de SP toch nog een "externe" procedure te maken. De mogelijkheid om native SQL PL stored procedures te creëren is misschien wel de belangrijkste nieuwe feature van DB2 9 voor z/OS!

In wat volgt beperken we ons tot SQL PL in stored procedures; de meeste zaken gelden (mutatis mutandis) ook voor UDFs en triggers.

De genoemde beperkingen impliceren dat het *uitvoeren* van SQL PL bij DB2 enkel kan door eerst een SP te creëren, en die dan expliciet op te roepen m.b.v. het SQL CALL-statement. Ook daar worden de klassieke front-end interfaces ietwat stiefmoederlijk behandeld: zo ondersteunen SPUFI en QMF het CALL-statement niet, zodat noodgedwongen een (COBOL-, PLI- of REXX-) programma moet geschreven worden, alleen maar om een stukje SQL PL daadwerkelijk uit te voeren. Anderzijds kan de DB2 Command Interface op Unix of Windows wel overweg met het SQL CALL-statement. Ook voor DB2 op z/OS kan dat een interessante optie zijn, b.v. om snel een nieuwe SP op z/OS remote uit te proberen via een interface op Unix of Windows. (Dit vereist weliswaar DB2 Connect.)

Mogelijkheden.

SQL PL ondersteunt uiteraard alle syntactische ingrediënten die nodig zijn om programma-logica te implementeren, en de mogelijkheid om dit te combineren met DML-statements (inclusief `SELECT ... INTO`):

- declaratie & gebruik van variabelen van elk gewenst DB2-datatype; toekenning van een waarde aan een variabele gebeurt met
  `SET <variabelenaam> = <scalaire expressie>`
  waarbij voor het rechterlid precies dezelfde mogelijkheden bestaan als bij het UPDATE-statement. Er kan dus i.h.b. ook een scalaire subquery gebruikt worden;
- programma-sequentie (separator ";") en optioneel ook samengestelde statements (BEGIN...END blocks);
- conditionele uitvoering van statements (IF...THEN...ELSE en CASE);
- iteratieve programma-logica (WHILE, REPEAT, LOOP, FOR);
- exception handling

Wat datatypes betreft: uiteraard worden alle DB2-types ondersteund, dus op z/OS sinds versie 9 ook b.v. BIGINT, (VAR)BINARY en DECFLOAT. Maar nog belangrijker is het te beseffen dat elke variabele ook de (pseudo-)waarde NULL kan aannemen. Naadloze integratie dus met wat DML verwacht en teruggeeft, maar vooral ook: geen onnodig geknoei met NULL-indicatoren.

Verder valt het nog op bij een SQL PL-programma dat er in de embedded DML geen onderscheid lijkt gemaakt te zijn tussen kolomnamen en (host)variabelen; er is dus geen ":var" notatie zoals in embedded SQL in COBOL, of geen "@var" zoals in MySQL of SQL Server.
Ambiguïteiten worden opgelost zoals gebruikelijk in nested SQL-statements: eerst wordt een naam geïnterpreteerd als kolomnaam van één van de tabellen in de FROM; daarna van een tabel in een eventuele externe query block; en tenslotte als variabele-naam, eerst van het huidige BEGIN...END-blok, daarna van het hoofdprogramma. Codefragment 1 van het vorige artikel heeft daarvan een duidelijk voorbeeld: in

```
UPDATE employee SET salary = newsal WHERE CURRENT OF c1 ;
```

is `salary` een kolomnaam (en dat blijft zo, ook al zou er een variabele met die naam bestaan), terwijl `newsal` een (host)variabele is omdat er geen kolom met die naam bestaat.

Procedurele logica: kort overzicht van de syntax.

De "body" van een SP, maar ook elk BEGIN...END blok, bestaat uit een strikte opeenvolging van een aantal secties (allemaal optioneel):

1. Declaratie van variabelen (die --indien gewenst-- dadelijk kunnen geïnitialiseerd worden m.b.v. een DEFAULT-clause);

2. Declaratie van cursors;

3. Definitie van “exception handlers” die automatisch geactiveerd worden wanneer een bepaalde DB2-conditie (SQLCODE) optreedt: hetzij exit, of continue (d.w.z.: ignore), of undo (dus rollback):

```
DECLARE { CONTINUE | EXIT | UNDO } HANDLER FOR
        { SQLSTATE <value> | SQLEXCEPTION | SQLWARNING | NOT FOUND }
 <single SQL-statement> ;
```

4. De eigenlijke programma-logica. Hierin kunnen o.a. de volgende twee constructies gebruikt worden; bedenk dat, zoals hierboven, elk `SQL-statement` op een “;” moet eindigen, en dat een `SQL-statement` ook een `BEGIN...END` blok mag zijn:

```
IF <condition>     THEN  <SQL-statement(s)> ;
ELSEIF <condition> THEN  <SQL-statement(s)> ; /* optioneel, mogelijk herhaald */
ELSE                     <SQL-statement(s)> ; /* optioneel */
END IF ;

WHILE <condition>   DO   <SQL PL statement(s)> ;  END WHILE ;
```

“`condition`” staat uiteraard voor alles wat in een WHERE-conditie mogelijk is, inclusief het gebruik van NOT, AND, OR, scalaire functies, en scalaire subqueries.

Achter de schermen: packages en versioning.

Elke “create procedure” is impliciet altijd ook een “bind package”: DB2 heeft, net zoals bij externe stored procedures, twee gescheiden objecten (beide met dezelfde naam, maar met gescheiden verantwoordelijkheden en onafhankelijke autorisaties): een procedureel object (gedocumenteerd in `SYSIBM.SYSROUTINES`) dat de SQL PL “source code” bevat, en een package (zie `SYSIBM.SYSPACKAGE`) dat het geoptimiseerde access path bevat.

REBIND is dus apart beschikbaar op het package-object, met alle opties (zoals b.v. `EXPLAIN(YES)` of `QUALIFIER(...)` of `REOPT`); deze opties zijn trouwens ook beschikbaar als CREATE PROCEDURE-opties.

Bovendien kan ALTER PROCEDURE gebruikt worden om het procedureel object te beheren: zo zal de REPLACE-clause toelaten, de sourcecode te vervangen zonder te raken aan de “object dependency”: zo blijven b.v. autorisaties bewaard en worden andere packages (static SQL) die een CALL bevatten naar deze SP niet invalid.

Het is mogelijk om een tweede (en derde enz.) procedure te creëren met dezelfde naam en signatuur(*), maar met een nieuw versie‘nummer’:

`CREATE PROCEDURE <naam> VERSION <versie>`

waarbij “`versie`” een willekeurige tekstuele waarde kan hebben. Default (bij afwezigheid van “VERSION”) wordt de waarde “`V1`” gebruikt. Elke versie heeft uiteraard z’n eigen package, maar er kan slechts één versie actief zijn (en dus opgeroepen worden); toekennen van een andere versie als de actieve kan m.b.v. “`ALTER PROCEDURE ... ALTER ACTIVE VERSION`”. Versies toevoegen en verwijderen gebeurt eveneens met ALTER PROCEDURE.

(*) De *signatuur* van een functie of procedure is het geheel van z'n naam, z'n schema-naam, en z'n parameter"structuur": aantal, volgorde, datatype en aard (IN, OUT of INOUT). De parameternamen doen er dus niet toe.

SQL PL versus COBOL.

Bij het ontwikkelen van (nieuwe) programmatuur, i.c. hier dus een SP, gelden drie belangrijke criteria: (1) de SP moet correct werken, volgens de specificaties dus; (2) de implementatie moet zo leesbaar mogelijk zijn (want dat bepaalt de onderhoudbaarheid, en dus onrechtstreeks de prijs bij latere enhancements); en pas in laatste instantie (3) performance: de implementatie moet uiteraard zo efficiënt mogelijk zijn.

Natuurlijk conflicteren (2) en (3) soms, maar verrassend genoeg is een leesbaarder implementatie (van software in het algemeen) meestal niet minder performant, en is een performantere implementatie meestal leesbaarder!

In elk geval: wanneer de keuze zich stelt tussen het implementeren van een bepaalde SP met hetzij COBOL, hetzij SQL PL, dan is (1) alleen maar een punt wanneer de SP bij voorbeeld externe bestanden moet lezen of schrijven of andere externe programma's (geen SPs) moet kunnen oproepen. NULLs en VARCHARs kunnen ook in COBOL correct afgehandeld worden, zij het op een (meestal) iets omslachtiger en dus minder leesbare manier dan in SQL PL. Wat ons brengt bij punt (2). De overzichtelijke paragraaf-structuur van COBOL daarentegen spelen we weliswaar kwijt in SQL PL, en ook de duidelijker scheiding tussen embedded SQL ("EXEC SQL" en de `:var` notatie) en programma-logica is een pluspunt voor COBOL. Ook het ingebouwde "exception handling"-mechanisme en de BEGIN...END blocks kunnen SQL PL minder leesbaar maken dan het COBOL-equivalent. Blijft nog het performance-aspect. Er wordt dikwijls beweerd (en meestal terecht) dat niets de performance van COBOL kan kloppen. In de vergelijking met SQL PL mag echter niet vergeten worden dat een native SP binnen de DBM1 address space draait en er dus (veel) minder inter-address space datatrafiek nodig is dan wanneer de SP extern (buiten DB2, b.v. onder WLM-controle) draait. Daar staat dan weer tegenover dat de WLM-flexibiliteit wat betreft resource-toekenning een systeem-globaal performance-voordeel kan zijn van de COBOL-oplossing t.o.v. de "not fenced" SQL PL-oplossing.

Conclusie.

Zeker sinds DB2 9 kunnen we er niet meer onderuit: vooral bij het ontwikkelen van een nieuwe SP moet native SQL PL als implementatiemogelijkheid als valabele optie overwogen worden. Pro's en contras moeten uiteraard nauwgezet tegen elkaar afgewogen worden, i.h.b. wat betreft onderhoudbaarheid en performance. Maar zeker voor relatief kleine programma's (d.w.z.: met vooral DML en verder relatief weinig programma-logica) zal de keuze bijna altijd in het voordeel van SQL PL uitvallen.

SQL PL als programmeertaal – onze ervaringen

Steven Scheldeman (ABIS)

Situatieschets.

Bij ABIS gebruiken we voor de cursusadministratie en het klantenbeheer een zelf ontwikkelde applicatie met de naam ACCA ("ABIS Client & Course Administration"). Deze bestaat uit een interface opgebouwd met ISPF Dialog Manager en aangedreven door COBOL programmatuur. Onze data is opgeslagen in DB2 for z/OS. Net zoals vele van haar tijdgenoten is ACCA doorheen de tijd als het ware organisch gegroeid. Er kwamen nieuwe noden aan het licht, bepaalde administratieve zaken moesten aangepast worden, integratie met andere pakketten was nodig..., kortom, er was reden genoeg om vele kleine en minder kleine wijzigingen aan te brengen.

Het grootste nadeel aan deze benadering is dat de programma's als het ware doorspekt zijn met vele kleine DB2 queries aan de ene zijde, afgewisseld met Dialog Manager calls. Met andere woorden, ook wij hebben te lijden onder een redelijke vergevorderde variant van spaghetti-programmatuur. Vooral de verwevenheid van de businesslogica en de GUI-logica bemoeilijkt het onderhoud en staat de modulariteit van de programmatuur in de weg.

Nieuwe noden, nieuwe aanpak.

Terwijl we enerzijds onze interne administratie verzorgen met ACCA, hebben we ook een website en een online-applicatie voor onze klanten: MyABIS. Eén van de beoogde doelen van MyABIS is onze klanten de mogelijkheid bieden zich online in te schrijven voor onze cursussen. Het spreekt voor zich dat we zo'n online inschrijving volgens dezelfde regels willen laten gebeuren, als wanneer we die inschrijving zelf zouden inbrengen met ACCA. Dit heeft ertoe geleid dat we besloten hebben een drastische verandering in de originele programmatuur van ACCA door te voeren.

Waar we tot voor kort de programmatorische vereisten van Dialog Manager totaal verweven hadden met onze Business Logica-programmatuur, willen we die nu gescheiden houden. Enerzijds hebben we alles wat nodig is om de interface aan te drijven, anderzijds hebben we de regels van onze Business Logica. Daar het niemand zal verwonderen dat we al onze administratieve programma's graag op dezelfde wijze willen laten werken, hebben we toen de beslissing genomen de bijhorende business-logica in een Stored Procedure te gieten en als DB2 object op te slaan. Het doel is hier om deze businesslogica onafhankelijk te houden van GUI's en i.h.b. niet te moeten dupliceren. Hierdoor kan elk programma die een inschrijving verzorgt, onafhankelijk van het platform waarop dit programma draait, en on-

afhankelijk van de programmeertaal waarin het programma geschreven is, deze Stored Procedure aanroepen, en wordt een inschrijving volgens gegarandeerd dezelfde businessregels afgewerkt.

COBOL of SQL PL?

Origineel werd er gespeeld met het idee de bestaande COBOL programmatuur uit ACCA te lichten en deze ongewijzigd als Stored Procedure op te slaan. De organische groei van ACCA doorheen de jaren, heeft er echter voor gezorgd dat we niet zomaar de Business Logica van de I/O-logica kunnen scheiden. Een grondig herschrijven bleek nodig te zijn. U begrijpt dat we op dat ogenblik een kans hadden die we met beide handen gegrepen hebben.

Enerzijds hebben we COBOL, een taal waar we zeer goed mee vertrouwd zijn. We kennen de mogelijkheden en beperktheden hiervan zeer goed. De interactie tussen COBOL en DB2 is goed gedocumenteerd en stelt ons voor weinig verrassingen, maar COBOL is nog steeds “niet eigen aan” DB2.

Sinds versie 9 kunnen we in DB2 ook ware programmatuur schrijven door het gebruik van SQL PL. Zoals de naam “Procedural Language” al aangeeft, biedt SQL PL ons de mogelijkheid een procedure te schrijven die eigen is aan DB2. En aangezien we toch onze Business Logica uit de originele programmatuur van ACCA moesten losweken, konden we net zo goed overstappen op een taal die inherent is aan DB2.

Bijkomend houdt dit meteen in dat de database nu onze businesslogica bevat als object dat we kunnen exporteren naar andere databases -- op LUW-omgevingen bv. -- zonder dat we daar ook moeten beschikken over COBOL.

Aandachtspunten en uitdagingen.

Natuurlijk zijn de praktische implicaties van SQL PL totaal anders. Ik heb hieronder enkele aandachtspunten opgesomd, en de overwegingen die we gemaakt hebben om tot bepaalde keuzes te komen.

1. *Compilatie*: Indien men COBOL gebruikt om een Stored Procedure te schrijven, dient men deze nog steeds te pre-compileren en te compileren. Deze noodzaak valt weg indien men SQL PL gebruikt. Een simpele Create Procedure volstaat. De “compilatie” gebeurt dan automatisch door de DB2 optimizer, meer bepaalt tijdens het uitvoeren van deze DDL. Zoals we allemaal weten correspondeert elke procedure -- na zijn creatie -- aan een package dat --net zoals de op COBOL gebaseerde packages-- verder autonoom kan beheerd worden (BIND-opties, REBIND, ...).

2. *SQLcodes*: In de Stored Procedure gaat men ettelijke SQL queries uitvoeren die elk hun eigen SQLcode genereren. Men mag dus niet vergeten dat men deze zal moeten testen en opvangen in de procedure zelf. Het oproepen van de Stored Procedure vanuit een COBOL programma (met een SQL CALL statement) zal immers zelf ook een SQLcode teruggeven, maar die vertelt ons niets over het resultaat van de SQL statements intern aan de procedure.

Dit heeft er ons toe geleid een eigen return-code systeem te bedenken. Meer specifiek komt dit erop neer dat de procedure altijd een getal en een boodschap terug geeft. Het getal is een unieke aanduiding van wat de procedure gedaan heeft --of niet gedaan heeft-- en wordt tijdens de procedure "berekend" (zie punt 4) en aan het eind ervan teruggegeven aan de gebruiker. De boodschap --op dit ogenblik nog niet actief-- zal uiteindelijk de leesbare (fouten)boodschap bevatten. We willen deze als rijen in een foutentabel opslaan met als Primary Key het getal dat door de procedure berekend wordt. Dit zal er dan ook toe leiden dat al onze verschillende GUI's dezelfde (fouten)boodschappen zullen presenteren.

3. *Parameters*: Zoals bij elke procedure moet er duchtig nagedacht worden over alle parameters die de procedure zal verwerken of teruggeven. In ons geval is de lijst redelijk lang: alle I/O velden van de schermen in Dialog Manager moesten als in/out parameters meegegeven worden.
We moesten een parameter voorzien om te bepalen wat we verwachten van de procedure -- een zogenaamde CRUD(P) parameter die enkel een input parameter is. Ruwweg vertaald: Create (= Insert), Retrieve (= Select), Update, Delete, en --specifiek voor ons eigen gebruik-- Preview. Dit laatste voert alle berekeningen uit die nodig zijn om een geldige Insert of Update te doen, zonder echter de actie uit te voeren in DB2. Dit geeft de mogelijkheid om dit --optioneel-- eerst interactief na te kijken en te bevestigen.
Tot slot moesten we ook enkele output parameters voorzien die onze eigen ontwikkelde returncode-set kunnen opvangen. Zoals eerder vermeld, bestaat deze uit enerzijds een getal en anderzijds een alfanumerieke string.

4. *Fouten-afhandeling*: Men zou kunnen zeggen dat de eigenheid van een Stored Procedure er in feite een soort van Batch programma van maakt. Mijn interface --in dit geval Dialog Manager-- aangedreven door COBOL roept een procedure op die volledig doorlopen dient te worden voor er kan ingegrepen worden. Dit houdt in dat we er rekening mee moeten houden dat de procedure niets in de database wijzigt, tenzij alles volledig correct is. Liefst zonder al te veel ROLLBACKs. We hebben dus heel wat controle queries in onze procedure gebouwd. Daarnaast is het karakter van SQL PL van dien aard dat ze bij een fout -- een exceptie -- de body van de procedure verlaat om rechtstreeks naar het 'exception handling' gedeelte van de procedure te springen. Dit willen we natuurlijk alleen maar indien de business-logica effectief spreekt over een fout, of indien er een systeem fout optreedt. Zogenaamde uitzonderingen die kunnen verwacht worden, dienen intern opgevangen en afgehandeld te worden. Klassiek voorbeeld: een "duplicate key" exception na een insert kan in het vervolg van de programma-logica als een (verwachte) "exists"-indicatie gebruikt worden, en hoeft zeker niet aan het oproepende programma gemeld te worden.

Natuurlijk dient in beide gevallen (echte fouten, voorzienbare uitzonderingen) de Stored Procedure dit te melden aan de gebruiker. Door het gebruik van evaluaties, optellingen en aftrekkingen, bere-

kent de procedure een getal bestaande uit 5 decimalen, die elk hun eigen betekenis hebben. De eerste twee verwijzen naar het onderdeel van ACCA (logisch gesproken) waar deze procedure bij hoort. De derde positie verwijst naar de onderdeel van de Stored Procedure waar de fout of uitzondering is opgetreden (Business-logica, Create, Retrieve, Update, Delete of Preview). De laatste twee vermelden dan de exacte aard ervan. Dit getal zal dan gebruikt worden om in de foutenboodschappen-tabel de bijhorende boodschap op te zoeken en deze samen met de "returncode" als uitgaande parameters terug te geven.

5. *NULL-waarden*: Niet elke programmeertaal kan even eenvoudig omgaan met NULL waarden. Het doel van de creatie van deze Stored Procedure --die onze Business Logica bevat-- is nu juist dat meerdere programma's deze kunnen oproepen, zonder dat we de interne logica keer op keer naar een nieuwe taal moeten omzetten. En dan hebben we twee keuzes: of we laten elke taal zelf de NULL waarden opvangen en afhandelen, of we zorgen ervoor dat de Stored Procedure dit intern doet. Hiermee bedoelen we dat we intern de overeenkomstige waarden omzetten van b.v. blanco naar NULL bij het binnenkomen van de parameters en vice versa.

We hebben er echter voor gekozen om de procedure NULL waarden te laten teruggeven aan de oproepende programma's. Uiteindelijk kan bijna elke programmeertaal omgaan met NULLs -- zij het soms op minder eenvoudige wijze. De talen die echter zelf gebruik maken van NULL waarden, hoeven er zich dan niet om te bekommeren: een teruggegeven string van blanco's is effectief een string van blanco's en een NULL is effectief een NULL.

6. *Security*: Het vraagstuk van de security zal hier dan ook anders bekeken worden dan bij traditionele programma's. De Stored Procedure is en blijft een DB2 object en valt als dusdanig onder de controle en security van de DBA. In mijn ogen vergroot dit de security. Een enkele SQL PL programmeur kan de autorisatie gegeven worden om de procedure te creëren, waarna anderen het recht krijgen ze uit te voeren. Hoe men dit intern regelt is natuurlijk een zaak van elk bedrijf op zich. Hoe men dit praktisch uitvoert weet elke DBA.

7. *Versioning*: Iets wat altijd een heet hangijzer blijft is de versioning van onze programmatuur. In het geval een Stored Procedure is dit in feite vrij simpel. Iedere maal we de Stored Procedure willen wijzigen, creëren eerst we een kopie met een andere naam. Daarna kunnen we naar hartenlust wijzigingen aanbrengen zonder enig gevaar dat de originele code --en haar functionaliteit-- verloren gaat.

Natuurlijk heeft DB2 zelf een versioning-mechanisme. Wij hebben er echter voor gekozen om dit niet te gebruiken. Traditioneel werken we met ACCA nog steeds met de beproefde methode van een Development/Test omgeving en een Productie omgeving. Dit wil ook zeggen dat een in productie genomen Stored Procedure nooit

rechtstreeks gewijzigd wordt, maar enkel in de development/test omgeving. Het enige dat wij er aan toe gevoegd hebben is een “historische” kopie van de productie Stored Procedure.

8. *Cursors*: SQL PL staat de creatie en het gebruik van cursors evenzeer toe als een taal zoals COBOL. Helaas is dit niet zo dynamisch als in PL/SQL --de Oracle-variant-- maar verder dan dat zijn cursors niet gelimiteerd. Je kan ze openen en sluiten, sequentieel doorlopen of scrollable maken. Het omzetten van COBOL-gebaseerde cursors -- zoals ze gebruikt werden in de oude implementatie -- naar SQL PL-cursors gebeurde zonder veel te moeten wijzigen. Het gebruik ervan wijzigde niet.

9. *Performance*: De business-logica vereist vele controles van reeds in de database opgenomen data (namen van bedrijven, verantwoordelijken, contactpersonen, email-adressen, ...) en berekeningen gebaseerd op dergelijke data (kortingspercentages, dagprijzen, koerswisselingen, ...). Door deze in een Stored Procedure te gieten houden we alle data-I/O binnen DB2 zelf. Dit verhoogt de performance van onze programmatuur.

Tot slot...

Persoonlijk vind ik dat SQL PL meer is dan ‘gadget’. De volledige Business Logica is vertaald naar een volwaardig programma, geschreven in SQL PL en opgeslagen als een Stored Procedure, een object binnen DB2. Sequenties, selecties, iteraties, alle bouwstenen van een traditionele programmeertaal zijn aanwezig en makkelijk bruikbaar. De vergevorderde Exception Handling, de mogelijkheid SQL PL recursief te programmeren, het gebruik van cursors, dit alles maakt mij een grote fan van het gebruik van SQL PL als de standaard taal om een Stored Procedure te schrijven, en niet alleen om enkele bewerkingen op de database te automatiseren, te stroomlijnen, maar om daadwerkelijk business beslissingen uit te voeren. Als men dan ook nog de performance in overweging neemt, dan raad ik het helemaal aan.

CURSUSPLANNING OKT – DEC 2011

DB2 concepten	460 EUR	op aanvraag
DB2 for z/OS, een totaaloverzicht	2075 EUR	10.10(W), 28.11(L)
DB2 UDB for LUW, totaaloverzicht	2075 EUR	17.10(L), 05.12(W)
RDBMS-concepten	385 EUR	10.10(W), 17.10(L), 28.11(L), 05.12(W)
Basiskennis SQL	385 EUR	11.10(W), 18.10(L), 29.11(L), 06.12(W)
DB2 for z/OS basiscursus	1305 EUR	24.10(W), 30.11(L)
DB2 UDB for LUW basiscursus	1305 EUR	19.10(L), 07.12(W)
SQL-QMF voor eindgebruikers	1305 EUR	19.10(W)
SQL workshop	820 EUR	10.10(L), 27.10(W), 12.12(L), 19.12(W)
SQL voor gevorderden	460 EUR	17.10(L), 14.11(W), 16.12(L)
DB2 SQL PL	920 EUR	27.10(L), 21.11(W)
DB2 triggers, stored procedures, UDFs	460 EUR	21.10(L), 18.11(W)
DB2 for z/OS programmeren voor gevorderden	920 EUR	op aanvraag
DB2 for z/OS: SQL performance	1380 EUR	18.10(L), 15.11(W)
XML in DB2	460 EUR	24.11(L)
DB2 for z/OS database administratie	1940 EUR	07.11(L), 12.12(W)
DB2 for z/OS data recovery	825 GBP	13.12(UK)
DB2 for z/OS systems performance and tuning	1020 EUR	14.11(L)
DB2 LUW DBA – Kernvaardigheden	1840 EUR	07.11(L), 13.12(W)
DB2 9 upgrade, DB2 10 upgrade	460 EUR	op aanvraag
Data warehouse concepten	460 EUR	29.11(W)
SQL voor BI	460 EUR	05.12(L)
Exploring DB2 – live!	185 EUR	op aanvraag

Plaats: L = Leuven, W = Woerden, UK = High Wycombe (bij Londen);
voor details en andere cursussen, zie `http://www.abis.be/html/nlTraining.html`

Postbus 220
Diestsevest 32
BE-3000 Leuven
Tel. 016/245610
Fax 016/245639
http://www.abis.be/
training@abis.be

abis
TRAINING & CONSULTING

Postbus 122
Zaagmolenlaan 4
NL-3440 AC Woerden
Tel. 0348-413663
Fax +32-16-245639
http://www.abis.be/
training@abis.be