ABIS NIEUWSBRIEF VOOR DB2 PROFESSIONALS

JAARGANG 8 - NUMMER 1, AUGUSTUS 2012

OPEN CURSOR

Data management wordt een almaar complexere aangelegenheid. Enerzijds doordat steeds meer datavolume moet bijgehouden worden: terabyte-tabellen zijn al lang geen uitzondering meer. Maar vooral ook omdat de verscheidenheid van de data toeneemt: log-gegevens van webbezoeken, sociale media, telefoongesprekken, ... Deze data moeten bovendien snel toegankelijk zijn. Uitdagingen die het er ons werk niet gemakkelijker op maken; maar waar software zoals DB2 ons een stukje kan bij helpen. Data-trends zoals toegenomen volume, parallellisme, snelheid, verscheidenheid ... weerspiegelen zich in nieuwe of gewijzigde mogelijkheden van DB2 versie 10. Waarover we u, zoals in het verleden, proberen op de hoogte te houden via Exploring DB2!

Ook deze zomer veel leesgenot met dit nummer!

Het ABIS DB2-team.

IN DIT NUMMER:

- In "*Temporele data en DB2 10 for z/OS - deel 3*", het laatste in deze reeks, wordt de "business time period" onder de loep genomen, opnieuw aan de hand van een uitgewerkt voorbeeld.
- "*SQL PL foutafhandeling*" gaat in op enkele technische details in SQL PL, met name twee verschillende manieren waarop "exception handling" kan gebeuren in een stored procedure.
- En tenslotte vindt u in Dossier 10 "*Non-key kolommen in unieke indexen*", waarin één van de opvallende nieuwe mogelijkheden beschreven wordt van DB2 10 voor z/OS.

1

CLOSE CURSOR

We hopen dat u in dit nummer van Exploring DB2 opnieuw interessant lees- en experimenteermateriaal hebt gevonden. DB2 biedt vele, soms verrassende, mogelijkheden die u wellicht ook in uw dagelijkse praktijk kunt benutten. In een volgend nummer proberen we u opnieuw te verrassen en te verrijken met meer DB2-nieuws!

Temporele data en DB2 10 for z/OS (III)

Kris Van Thillo (ABIS)

In vorige nummers van Exploring DB2 stonden we reeds stil bij 'Temporele data'. In eerste instantie, om een aantal conceptuele elementen eigen aan dit onderwerp binnen de context van DB2 versie 10 nader uiteen te zetten; in een tweede nummer stonden we stil bij de implementatie van temporele data in DB2 versie 10 op basis van 'system time' als referentiekader.

In dit laatste artikel bespreken we de implementatie in DB2 op basis van 'business time' - zeg maar *business periode*, dus de door de business bepaalde geldigheidsperiode.

Achtergrond en opzet.

Temporeel databeheer geïmplementeerd op basis van 'system time' heeft tot doel wijzigingen aan DB2 data vanuit historisch perspectief te registreren, om waar nodig, deze wijzigingen te kunnen analyseren, interpreteren - er dus gebruik van te maken. Aldus ontstaat het idee van 'huidige', 'current' data, opgeslagen in een standaard (inderdaad temporele) DB2-tabel, en historische data, opgeslagen in een history table, met 'oude' data. Deze history-tabel wordt in applicaties nooit rechtstreeks gebruikt. Inderdaad, DB2 zal i.f.v. de voorliggende vraag op de standaard temporele tabel waneer nodig transparant van deze history-tabel gebruik maken.

De data opgenomen in deze history-tabel heeft evenwel geen verband met de actuele bedrijfsvoering, de 'business'-gerelateerde 'betekenis'! Want de data is niet meer dan een historisch overzicht, vormt louter een historiek.

Bij implementatie van temporele data op basis van *business time* is dit niet langer het geval. Data krijgt in deze context binnen zijn historisch kader nu een concrete betekenis: applicatief en/of bedrijfsspecifiek. Ziedaar de link met het concept *business periode*.

Bijvoorbeeld. Een klant van ABIS is van plan te migreren van DB2 z/OS naar Oracle en zal daartoe in de periode 01.09.2012 - 31.08.2013 een aanzienlijk aantal opleidingen bij ABIS afnemen. Met deze klant wordt een raamovereenkomst afgesloten en een korting toegekend. Prijzen en kortingen gehanteerd vóór 01.09.2012, tussen 01.09.2012 en 31.08.2013 (inbegrepen), en na 31.08.2013 hebben (naast hun evidente historische relevantie) ook een bedrijfsspecifieke relevantie, gerelateerd aan het bestaan van de bewuste raamovereenkomst (hier dus de relevante *business periode*).

Bovenstaande casus kan op een eenvoudige manier geïmplementeerd wordenop basis van de op volgende pagina opgenomen tabel.

De temporele tabel moet in dit geval enkel een 'start' en 'stop' kolom bevatten, van het type timestamp(6) of date. De applicaties zijn daarna verantwoordelijk voor het verstrekken van waarden voor deze beide kolommen; waarden, die het begin/eind van een *business periode* aangeven. Andere, bijkomende kolommen (b.v. om een transactie ID op te slaan) zijn niet vereist.

Tabelcreatie

```
create table reductions
(rno         int           not null,
 r_companyid int           references companies on delete cascade,
 rrebate     decimal(2,2)  not null,
 rstart     date not null,
 rstop      date not null,
 period business_time (rstart, rstop),
 primary key (rno, business_time without overlaps));
```

De instructie PERIOD BUSINESS_TIME voegt *business time* gebaseerde temporele ondersteuning aan deze tabel toe. Een history-tabel (schaduwtabel) wordt niet gebruikt: alle data wordt in één tabel opgeslagen, met indicatie van de voor die data relevante *business periode*. De primary key definitie geeft aan dat binnen een bepaald *business time interval* keys niet mogen worden herbruikt (uniek moeten zijn). DB2 voegt transparant een CHECK CONSTRAINT toe die nagaat of - toegepast op ons voorbeeld - rstart < rstop. Merk op dat de bovengrens geen deel uitmaakt van de eigenlijke business-periode.

Temporele ondersteuning, i.e. versioning, kan trouwens achteraf aan een bestaande tabel worden toegevoegd, met standaard ALTER TABLE statements. Merk op dat het naderhand wijzigen van een basistabel (aan de hand van ALTER TABLE statements) enkel mogelijk is na het opheffen van de temporele ondersteuning.

Het moment is nu aangebroken om bovenstaande opzet ook daadwerkelijk te gaan gebruiken! Vooraf is het belangrijk even stil te staan bij volgende bedenkingen.

- Alle in deze context relevante data wordt in één enkele DB2-tabel opgeslagen - niet over twee tabellen verspreid! Standaard DB2 queries, die van de temporele features van DB2 in deze context *geen* gebruik maken, zullen *extra data* opleveren, data m.b.t. alle gedefinieerde business periodes, met inbegrip van mogelijk oude én toekomstige data!

- temporele ondersteuning wordt mogelijk door het opnemen van de instructie FOR PORTION OF BUSINESS TIME, zoals in wat volgt nader uiteengezet wordt.

Een belangrijk gevolg van bovenstaande opmerkingen is dat het gebruiken van temporele ondersteuning op basis van *business periode* niet transparant is voor gebruikers, en aanpassingen aan applicaties vereist!

DML - Update, delete en insert.

Traditionele, standaard DML-instructies kunnen gewoon worden gebruikt.

- voor *insert* operaties dient een waarde te worden verstrekt voor alle in de tabel relevante kolommen (standaard gedrag dus), met inbegrip van de kolommen die een periode aanduiding meegeven; de opgegeven waarden worden door DB2 toegevoegd aan de tabel;
- voor *standaard update* operaties zal DB2 de originele rij gewoon wijzigen als aangegeven. Indien evenwel de optie FOR PORTION OF BUSINESS_TIME wordt meegegeven, zal de update enkel betrekking hebben op dat deel van de business periode. Voor het/de overige deel/delen wordt een *nieuwe rij* toegevoegd met de oude waarden. De *business periode* grenzen worden aangepast;
- voor *standaard delete* statements wordt de rij uit de tabel weggehaald als verwacht. Indien evenwel de optie FOR PORTION OF BUSINESS_TIME wordt meegegeven, zal de delete enkel betrekking hebben op dat deel van de business periode. Voor het resterende deel wordt de *bestaande rij* aangepast (de business periode wordt aangepast voor het niet gedeleted deel) en/of een nieuwe rij toegevoegd. De *business periode* grenzen worden aangepast;

Bovenstaande processing wordt automatisch en transparant door DB2 uitgevoerd.

Over primary keys!

Indien bij het aanmaken van een tabel een primary key wordt gedeclareerd met de optie BUSINESS_TIME WITHOUT OVERLAPS, kan per business periode één primary key maar één maal voorkomen. Als het dus nodig zou zijn bijvoorbeeld met één primary key waarde een tweede business periode te associëren, zal dit enkel lukken indien de bestaande periode ge-updated wordt. Dit zal DB2 aanzetten tot het updaten van de bestaande rij en het toevoegen van een nieuwe rij voor de resterende periode.

Voorbeelden.

Een aantal voorbeelden volgen om e.e.a. nader te verduidelijken. We baseren ons op de business case die hierboven werd uiteengezet.

DML (1)

```
insert into reductions
values (100, 18661, 0, '01.01.2012', '01.09.2012');

insert into reductions
values (100, 18661, 8, '01.09.2012', '01.09.2013');

insert into reductions
values (100, 18661, 0, '01.09.2013', '01.09.2015');
```

Tabel reductions

rno	r_companyid	rrebate	rstart	rstop
100	18661	0	01.01.2012	01.09.2012
100	18661	8	01.09.2012	01.09.2013
100	18661	0	01.09.2013	01.09.2015

Als bovenstaande statements zijn uitgevoerd met de opgenomen referentietijden, bevat de tabel zoals verwacht 3 rijen.

Het project loopt evenwel uit - concreet wordt de raamovereenkomst met 6 maand verlengd aan dezelfde voorwaarden, van 01.09.2013 tot en met 31.03.2014. Het uit te voeren update statement wordt in wat volgt aangegeven.

DML (2)

```
update reductions
       for portion of business_time from '01.09.2013' to '01.04.2014'
set rrebate = 8
where rno = 100;
```

Tabel reductions

rno	r_companyid	rrebate	rstart	rstop	
100	18661	0	01.01.2012	01.09.2012	
100	18661	8	01.09.2012	01.09.2013	
100	18661	8	01.09.2013	01.04.2014	<<< insert
100	18661	0	01.04.2014	01.09.2015	<<< update

De inhoud van bovenstaande tabel geeft aan dat de laatste rij uit voorbeeld DML(1) werd gesplitst in twee business periodes; de update werd feitelijk verwerkt als een update gevolgd door een insert. Merk op dat als de bij de update opgegeven business periode twee reeds aanwezige rijen 'omspant', beide betrokken rijen indien relevant (afhankelijk van de opgegeven *portion of business_time*) zullen worden opgesplitst!

Voor een delete operatie gebeurt identiek hetzelfde bijvoorbeeld als de duur van de raamovereenkomst met 1 maand wordt ingekort.

DML (3)

```
delete from reductions
       for portion of business_time from '01.08.2015' to '01.09.2015'
where rno = 100;
```

Tabel reductions

rno	r_companyid	rrebate	rstart	rstop	
100	18661	0	01.01.2012	01.09.2012	
100	18661	8	01.09.2012	01.09.2013	
100	18661	8	01.09.2013	01.04.2014	
100	18661	0	01.04.2014	01.08.2015	<<< update

Sommige delete operaties hebben evenwel interessante neveneffecten! Stel bijvoorbeeld dat we om één of andere reden de raamovereenkomst niet willen toepassen in de maand januari van 2013.

DML (4)

```
delete from reductions
       for portion of business_time from '01.01.2013' to '01.02.2013'
where rno = 100;
```

Tabel reductions

rno	r_companyid	rrebate	rstart	rstop	
100	18661	0	01.01.2012	01.09.2012	
100	18661	8	01.09.2012	01.01.2013	
100	18661	8	01.02.2013	01.09.2013	<<< insert
100	18661	8	01.09.2013	01.04.2014	
100	18661	0	01.04.2014	01.08.2015	

En wat betreft het select statement...

... is e.e.a. gemakkelijk uit te leggen. De SELECT zonder indicatie van een business periode heeft steeds betrekking op alle aanwezige data, zoals standaard het geval. Echter, dit is niet meer het geval wanneer het select statement uitgebreid wordt met een uitdrukkelijke verwijzing naar een business periode.

SELECT statement

```
select count(*)
from reductions
where rno = 100; -- resultaat = 5


select rrebate
from reductions
     for business_time as of '01.10.2013'
where rno = 100; -- resultaat = 8 (één rij dus)
```

Andere alternatieven om de business time in rekening te brengen maken gebruik van de functies FOR BUSINESS_TIME FROM ... TO ... (de 'to' tijd *niet* inbegrepen) of FOR SYSTEM_TIME BETWEEN ... AND ... (waarbij de 'and' tijd *wel* is inbegrepen in het zoekcriterium). Merk op dat om de 'current' waarde, van één specifieke *business periode* te bekomen, het gebruik van de FOR BUSINESS_TIME AS OF optie vereist is. Deze optie beperkte de output tot de opgegeven *business periode*.

Ook hier gebeurt e.e.a. achter de schermen: de verwijzing naar een business periode zorgt er inderdaad voor dat DB2 een aantal passende WHERE condities aan de oorspronkelijke SELECT toevoegt (query rewrite), die allen verwijzen naar de *business periode* 'start' en 'stop' gegevens aanwezig in de tabel.

Implementatie van deze functionaliteit was in vorige versies niet mogelijk en kon dan ook enkel op niveau van applicaties worden opgevangen.

Tot slot.

Het is mogelijk tabellen aan te maken met 'bi-temporele' ondersteuning, waarbij zowel 'system_time' als 'business_time' implementaties worden gecombineerd. Merk op dat de implementatiebeperkingen aangegeven aan het eind van het vorige artikel ook in deze context van toepassing zijn.

SQL PL foutafhandeling

Peter Vanroose

(ABIS)

Wat vooraf ging.

Het onderwerp "*SQL PL*" is al enkele malen aan bod gekomen in Exploring DB2: voor het eerst in mei 2003 (in nummer 8 van onze eerste jaargang!) waar de eerste implementatie (in DB2 versie 7 voor zowel LUW als OS/390) van deze procedurele component van ons aller RDBMS onder de loep werd genomen, en waar ook een verhelderend voorbeeld is uitgewerkt. Bij het nalezen van die bijdrage viel het me op dat er in 10 jaar tijd syntactisch zo goed als niets veranderd is. Wat ik vooral als een pluspunt zie: SQL PL -de taal- was vanaf dag 1 een weldoordacht geheel. Relatief eenvoudig maar toch zeer volledig. Zonder overdadig te zijn.

Zie "*SQL/PL - een beknopt overzicht*", in J1N8.

Daarna was het blijkbaar een poosje windstil op het SQL PL-front, maar vorig jaar (in nummer 3 van jaargang 7 van Exploring DB2) brachten we u een themanummer rond SQL PL. Met daarin enerzijds een vergelijking met Oracle's PL/SQL, anderzijds een beknopt syntactisch overzicht, en ten slotte een verslag van onze ervaringen met het herschrijven van een stukje COBOL-programmatuur naar een stored procedure in SQL PL. Daarin worden o.a. een aantal design-keuzes beschreven die we gemaakt hebben, zowel infrastructureel als syntactisch.

Zie "*Exploring DB2 J7N3*".

In deze bijdrage wordt dieper ingegaan op mogelijke alternatieve design-keuzes die SQL PL biedt, in het bijzonder wat interne foutafhandeling betreft. Om alles voldoende concreet te houden, zijn de uitgewerkte voorbeelden gebaseerd op de probleemstelling van het artikel "*SQL PL als programmeertaal -- onze ervaringen*" uit het genoemde themanummer van Exploring DB2. Met name dus een stored procedure die een inschrijving op een ABIS-cursus verwerkt in ons CRM-systeem "ACCA".

Foutafhandeling - "exception handling"

Voor de Java-programmeurs onder jullie is het concept "exception handling" (m.n. "throw" en "catch") wellicht zeer vertrouwd. Kort samengevat komt het erop neer dat een "module" (dus een subprogramma of een stored procedure) enerzijds een aantal onverwachte (exceptionele) situaties zelf kan inschatten en ook zelf afhandelt ("*catch*") zodat de oproeper (laat ons zeggen "het hoofdprogramma") er niets van merkt, en anderzijds een aantal andere exceptions helemaal niet verwacht en ze gewoon "doorgeeft" aan z'n oproeper ("*throw*"). Die er hopelijk wel iets mee doet, of ze op z'n beurt doorgeeft naar diens oproeper: hetzij de interactieve gebruiker (foutenboodschap op het scherm) of de runtime-omgeving van het batchprogramma (b.v. returncode 8 of 12), of nog een tussenlaag met z'n eigen oproeper.

In het “3-tier model” dat we bij ACCA gebruiken (database-laag / business logica / presentatielaag) kan elk van de drie lagen een exception *throw*en naar z’n oproeper, die op z’n beurt kan *catch*en of doorgeven: DB2 geeft b.v. een negatieve SQL-code terug bij een ongeldige INSERT; die door de business-laag (geschreven in COBOL) al dan niet opgevangen wordt, of doorgegeven aan een ISPF Dialog Manager routine (had trouwens ook CICS kunnen zijn), die ervoor zorgt dat de gebruiker de SQL-code op z’n scherm ziet. Daarnaast kan de business-laag ook zelf een ongeldige situatie ontdekken (b.v. een inschrijving zonder facturatie-adres, of een ongeldig email-adres) en een ACCA-foutencode op het scherm laten zetten.

Zoals beschreven in de vorige bijdrage, hebben we het oorspronkelijke COBOL-programma met daarin een mix van business-logica en ISPF Dialog Manager “GUI” logica netjes opgedeeld in twee functioneel gescheiden modules: de business-logica voor de inschrijving in een nieuwe stored procedure, en de GUI-logica plus natuurlijk de oproep van die procedure in COBOL.

Wat foutafhandeling betreft hebben we nu eigenlijk een laag meer: de back-end is nog steeds DB2 (voor alles wat SELECTs en INSERTs e.d. betreft); de volgende laag is de stored procedure (weliswaar ook DB2, maar logisch gesproken een tweede *layer*) die de SQL-codes al dan niet interpreteert, opvangt, of doorgeeft aan laag 3, het COBOL-programma met enkel nog de GUI-logica, dat op z’n beurt al dan niet verder doorgeeft aan de eindgebruiker.

De procedure “cursist inschrijven op een ABIS-cursus”

De communicatie tussen laag 3 (COBOL) en laag 2 (de procedure) bestaat uit de volgende invoer (dus “IN”-parameters):

- identificatie van de cursist (persoonsnummer: `pno`)
- identificatie van zijn bedrijf (facturatie-adres: `cono`)
- identificatie van de cursus, of eigenlijk van de sessie (datum/locatie: `seno`)

De procedure zorgt in de eerste plaats voor validatie van de invoer, zoals cursist reeds ingeschreven; cursus volzet; ongeldige gegevens; onvolledige gegevens; ... Verder bestaat de business-logica van de procedure uit het aflezen van de basisprijs van de cursus uit DB2-tabellen (SELECT), een eventuele korting berekenen op basis van bedrijfsgegevens of voucher, en ten slotte het opslaan van de inschrijving in een DB2-tabel (INSERT).

De procedure geeft volgende gegevens terug aan zijn oproeper (in de vorm van “OUT”-parameters, om te tonen op het scherm): het gegenereerd inschrijvingsnummer (`eno`) als identificator, en alle afgeleide gegevens (naam van cursist en bedrijf; cursusdatum; berekende prijs; ...) voor visuele verificatie. Een eventuele exception wordt momenteel teruggegeven via drie extra OUT-parameters: een ACCA-foutencode, de bijhorende foutenboodschap, en de laatste SQLCODE (uit laag 1 dus).

Codevoorbeeld 1 - de stored procedure "InsEnrol"

```
CREATE PROCEDURE InsEnrol
( IN    p_seno INT
,   OUT p_eno SMALLINT
, IN    p_studpno INT
,   OUT p_studname VARCHAR(80)
, IN    p_invcono INT
,   OUT p_compname VARCHAR(80)
, INOUT p_redpercent INT
, INOUT p_invprice DECIMAL(10,2)
,   OUT p_msg VARCHAR(80)
,   OUT p_sqlcode INT
)
dynamic result sets 0
language SQL   modifies SQL data
disable debug mode
BEGIN
  DECLARE SQLCODE INT;
  SELECT 0 INTO p_sqlcode FROM sessions WHERE seno = p_seno;
  IF SQLCODE <> 0 THEN
    [ exception handling & return ]
  END IF;
  [ analoog voor p-studpno & p-invcono; & ophalen p-studname & p-compname ]
  IF p_invprice IS NULL THEN /* d.w.z.: prijs niet opgegeven door de oproeper */
    [ business-logica om p_redpercent op te zoeken, en p_invprice te berekenen ]
  END IF;
  SELECT coalesce(MAX(eno),0)+1 INTO p_eno FROM enrolments WHERE e_seno=p_seno;
  INSERT INTO enrolments VALUES (p_seno, p_eno, p_studpno, ....);
  IF SQLCODE <> 0 THEN [ foutenboodschap ] END IF;
  SET p_sqlcode = SQLCODE; SET p_msg = 'Enrolment inserted';
END
```

Bemerk de twee INOUT-parameters: bij de oproep kan al een prijs of een kortingspercentage meegegeven worden; zo niet, dan worden deze berekend en teruggegeven door het (niet uitgeschreven) stukje business-logica in de procedure. Dat stukje kan er b.v. als volgt uit zien:

Codevoorbeeld 2 - fragment van de stored procedure "InsEnrol"

```
  IF p_redpercent IS NULL THEN
    SELECT repercent INTO p_redpercent FROM reductions WHERE re_cono=p_invcono;
    IF SQLCODE <> 0 THEN SET p_redpercent = 0 END IF;
  END IF;
  SELECT cfprice INTO p_invprice FROM coursefees INNER JOIN currency
         ON cf_crcode = crcode WHERE cf_seno = p_seno;
  /* korting: */
  SET p_invprice = p_invprice * ( 1.00 - p_redpercent * 0.01 );
```

Alternatieve foutafhandeling

In de hierboven uitgeschreven implementatie van *InsEnrol* gebeurt zeer minimale exception handling, eigenlijk volledig in de stijl zoals het vanuit een COBOL-programma zou gebeuren: de variabele SQLCODE wordt systematisch geïnspecteerd na elke "EXEC SQL" (hier dus na elke SELECT/INSERT/...), op basis waarvan de program flow kan aangepast worden.

SQL PL laat echter een veel flexibeler benadering toe, door gebruik te maken van zgn. CONTINUE HANDLERs en EXIT HANDLERs: beide worden getriggerd door een DB2 “exception” (dus een niet-nul-SQLCODE); de eerste “catcht” deze exception, de tweede “throwt” de exception (of een andere naar keuze) naar de oproeper. Het interessante van deze handlers is dat hun definitie volledig buiten de procedure-body valt, zodat de (hoofd)flow van het programma niet visueel onderbroken wordt door het afhandelen van “exceptionele” gebeurtenissen:

Codevoorbeeld 3 - zeer eenvoudige exception handling

```
BEGIN
  DECLARE EXIT HANDLER FOR SQLEXCEPTION SET p_sqlcode = SQLCODE;
  DECLARE EXIT HANDLER FOR SQLWARNING SET p_sqlcode = SQLCODE;
  DECLARE EXIT HANDLER FOR NOT FOUND SET p_sqlcode = 100;
  SET p_msg = 'No such session';
  SELECT 0 INTO p_sqlcode FROM sessions WHERE seno = p_seno;
  IF p_redpercent IS NULL THEN BEGIN
    DECLARE eof INT DEFAULT 0;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET eof = 1;
    SELECT repercent INTO p_redpercent FROM reductions WHERE ... ;
    IF eof = 1 THEN SET p_redpercent = 0 END IF;
  END; END IF;
  SET p_msg = 'Insert failed';
  INSERT INTO enrolments VALUES (p_seno, p_eno, p_studpno, ....);
  SET p_msg = 'Enrolment inserted';
END
```

Ook de gebeurtenissen “end-of-cursor” en “not found” (beide SQLCODE = 100) kunnen dus een handler triggeren; in het eenvoudige voorbeeld wordt gewoon een waarde gegeven aan een vlag-variabele (die hier, voor de eenvoud een output-parameter is).

Er is nu dus geen test op SQLCODE meer nodig; maar er komen drie handlers voor in de plaats. Men kan er trouwens voor kiezen, meer selectieve handlers te declareren, b.v. voor enkel de SQLCODE -811 (“select into met meer dan 1 rij”). Strikt genomen zal de handler niet de SQLCODE specificeren, maar aangeven dat de SQLSTATE = ‘21000’ wordt opgevangen.

Bemerk in het bijzonder de “embedded” continue handler (en de BEGIN ... END eromheen): deze handler overschrijft tijdelijk de exit handler die in het buitenste niveau werd gedeclareerd. Uiteraard kunnen er meerdere geneste blokken gebruikt worden, elk met meerdere EXIT of CONTINUE handlers.

In de praktijk

Uiteraard zal een stored procedure geschreven in SQL PL er in de praktijk iets complexer uit zien dan wat hier getoond werd. Maar de boodschap is hopelijk duidelijk: het scheiden van exception-logica van de eigenlijke, “gewone” business-logica maakt een programma een heel stuk leesbaarder en dus beter onderhoudbaar.

Wie meer wil weten over SQL PL en z’n mogelijkheden is uiteraard welkom op onze cursus *Software-ontwikkeling met SQL PL*.

DOSSIER 10

Non-key kolommen in unieke indexen!

'Index-only' toegangspaden naar data hebben het alomgekende voordeel: datatoegang kan worden vermeden tijdens het uitvoeren SQL-instructies. Bij het evalueren van alternatieve index-strategieën wordt met deze bedenking dan ook steeds rekening gehouden.

In het verleden betekende dit vaak de creatie van extra indexen, met identieke 'leading columns': één unique index, en één (of een aantal) extra (al dan niet unieke) indexen, waaraan kolommen werden toegevoegd, vereist voor index-only toegang naar deze extra kolommen. Met extra kosten (CPU-kosten, beheerskosten, opslag, ...) als gevolg tijdens het aanpassen van de tabeldata.

Dit kan nu worden vermeden, door de mogelijkheid die in DB2 versie 10 wordt geboden om aan unieke indexen extra - *included* - kolommen toe te voegen.

Een voorbeeld verduidelijkt:

```
CREATE UNIQUE INDEX ix1
ON course_sessions (sessionnumber)
INCLUDE (se_coursenumber).
```

Een aantal aandachtspunten kort samengevat:

- INCLUDE kolommen kunnen eveneens met alter index aan een index worden toegevoegd (als deze zelf nog geen included kolommen heeft); de index wordt in een 'reorganisatie pending' state geplaatst (afhankelijk van een aantal factoren AREO, of PSRBD);

- INCLUDE kolommen kunnen enkel voor unieke indexen worden opgegeven. Ze hebben geen invloed op de functionaliteit achter het concept unieke index: de extra kolommen worden niet gebruikt om uniciteit af te dwingen, en ook binnen de context van RI hebben ze geen specifieke invloed;

- Een aantal indextypes zijn niet beschikbaar voor uitbreiding met INCLUDE kolommen: o.a. index-on-expression indexen, auxiliary indexen, XML-indexen, ...

- sysibm.sysindexes bevat informatie over het aantal (leading) kolommen in de index gebruikt voor het afdwingen van uniciteit (unique_count);

Afhankelijk van de workload, en van de omvang van de extra toegevoegde kolommen, kunnen de besparingen ten gevolge van de consolidatie van het aantal indexen aanzienlijk zijn.

Koen De Backer (ABIS)

CURSUSPLANNING SEP. - DEC. 2012

DB2 for z/OS, een totaaloverzicht	2125 EUR	17.09(L), 22.10(W))
DB2 UDB for LUW, totaaloverzicht	2125 EUR	17.09(L), 03.12(W)
RDBMS-concepten	395 EUR	17.09(L), 15.10(L), 22.10(W), 26.11(L), 03.12(W)
Basiskennis SQL	395 EUR	18.09(L), 16.10(L), 23.10(W), 27.11(L), 04.12(W)
DB2 for z/OS basiscursus	1335 EUR	19.09(L), 24.10(W)
DB2 UDB for LUW basiscursus	1335 EUR	19.09(L), 10.12(W)
SQL-QMF voor eindgebruikers	1335 EUR	op aanvraag
SQL workshop	840 EUR	27.09(L), 12.11(W), 13.12(W), 17.12(L)
SQL voor gevorderden	470 EUR	08.10(L), 26.11(W), 14.12(L)
Software-ontwikkeling met SQL PL	940 EUR	22.10(L), 20.12(W)
DB2 triggers, stored procedures, UDFs	470 EUR	16.11(L), 27.11(W)
DB2 for z/OS programmeren voor gevorderden	940 EUR	04.10(L)
DB2 for z/OS: SQL performance	1410 EUR	09.10(L), 21.11(W)
CICS applicatieprogrammering	1410 EUR	01.10(L)
TSO/E REXX	890 EUR	17.09(W), 22.11(L)
XML in DB2	470 EUR	op aanvraag
DB2 for z/OS database administratie	1980 EUR	05.11(W), 10.12(L)
DB2 for z/OS installation and migration	850 GBP	22.10(UK)
DB2 for z/OS data recovery	825 GBP	30.10(UK)
DB2 for z/OS systems performance and tuning	850 GBP	19.11(UK)
DB2 LUW DBA - Kernvaardigheden	1880 EUR	08.10(L), 17.12(W)
DB2 10 for z/OS: new features	850 GBP	15.10(UK), 03.12(UK)
SQL voor BI	470 EUR	19.11(L)

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