ABIS NIEUWSBRIEF VOOR DB2 PROFESSIONALS

JAARGANG 2 - NUMMER 3, NOVEMBER 2003

OPEN CURSOR

Tijdens de recente IDUG user-bijkomst is opnieuw duidelijk de richting gebleken die IBM met DB2 voor OS/390 is ingeslagen. Kernachtig uitgedrukt: grotere beschikbaarheid en onderhoudbaarheid (e.g. schema evolutie, data partitioned secondary indexes), performantie (e.g. 64-bit problematiek, materialized query tables), schaalbaarheid. Maar ook integratie staat centraal: DB2 Replication, Information Integrator en natuurlijk Unicode.

Ook coördinatie en integratie van de 'feature set' overheen de verschillende leden van de DB2-familie was duidelijk aan de orde.

En voor applicatieontwikkelaars zijn er nieuwe evoluties m.b.t. Java, XML en SQL.

ABIS en Exploring DB2 zullen ervoor zorgen dat u van al deze evoluties op de hoogte blijft.

Het ABIS DB2-team.

IN DIT NUMMER:

- Stored objects worden steeds meer gebruikt; in *Triggers & redundancy* staan we stil bij een mogelijk gebruik.
- Het derde deel in onze reeks DB2 applicatieperformance - *DB2 performance - case 3*.
- *Dossier 8* behandelt multiple row inserts.

Cursusplanning december 2003 - maart 2004.

CLOSE CURSOR

In een volgend nummer bekijken we opnieuw .NET: .NET-configuratie en -tuning vanuit DB2-optiek is nu aan de orde. We spreken over utilities. En voor de laatste maal, een DB2 applicatieperformance case.

Tot dan!

Triggers & redundancy

Pieter Bedert (ABIS)

Verhoog de data access performance!

In nr. 2 van de vorige jaargang wijdde ik een artikel aan het gebruik, maar vooral aan het creëren van een trigger. Dit artikel zal u trachten te overtuigen dat triggers één van de meest onderschatte objecten zijn in uw database. Na het lezen zal u dan ook een onweerstaanbare drang voelen om uw eerste trigger te gaan implementeren. Maar voor het zover is, beginnen we misschien best bij het begin.

Wat is een trigger?

Triggers zorgen voor een automatische - voorwaardelijke of onvoorwaardelijke - uitvoering van een reeks SQL-statements voor of na een bepaalde datamanipulatie in een tabel. Dus voor of na een insert, update of delete wordt aan de hand van een conditie gecontroleerd of een reeks SQL-statements al dan niet moet uitgevoerd worden. Belangrijk in deze definitie is het feit dat een trigger automatisch en altijd wordt afgevuurd!

Triggers zijn bovendien database-objecten, aangemaakt of verwijderd met behulp van DDL-statements.

Waarvoor worden triggers gebruikt?

Triggers kunnen in een veelheid van situaties worden gebruikt.

- Constraint: Wanneer een check-constraint tekort schiet voor de controle van een bepaalde kolom - je moet bijvoorbeeld een controle uitvoeren in een andere tabel - dan kun je altijd een before-trigger gebruiken om die constraint af te dwingen.
- Business-logica: Een trigger kan bij bepaalde gebeurtenissen automatisch de logische daaropvolgende stap uitvoeren. Wanneer bijvoorbeeld het aantal stuks in voorraad beneden een limiet komt, kan de trigger automatisch een order inserten.
- Auditing: Een trigger kan bij elke manipulatie van een tabel automatisch informatie over de gebruiker en de uitgevoerde actie in een log-tabel inserten.
- Replicatie: Een trigger kan ervoor zorgen dat informatie automatisch gerepliceerd wordt in andere tabellen (schaduw-tabellen) - meestal wordt enkel bijgehouden welke actie op welke rij werd uitgevoerd (verschilt met 'Auditing' trigger).
- Redundancy: Triggers kunnen gebruikt worden om redundante informatie of afgeleide informatie te onderhouden. Daar gaan we straks dieper op in.

Voordelen van triggers

Een aantal interessante voordelen worden even op een rijtje gezet.

- Triggers worden altijd afgevuurd wanneer de geassocieerde actie zich voordoet. Ontwikkelaars moeten zich dus geen zorgen maken over het herhalen van bepaalde stukken coding in meerdere applicaties. Bovendien kunnen willekeurige ad-hoc gebruikers, bijvoorbeeld buiten een applicatie om, deze functionaliteit niet omzeilen.

- Triggers worden centraal beheerd. Ze worden eenmalig gecodeerd, uitvoerig getest en het toegangspad wordt geoptimaliseerd. Dit wil zeggen dat ze over het algemeen zeer efficiënt hun werk doen.

- Vooral in gedistribueerde (client-server) omgevingen realiseren ze een performance voordeel. De trigger voert een reeks SQL-statements uit waarvoor slechts één database-access nodig is vanuit de client, dus ook maar één communicatie over het netwerk.

- Een online 'load resume' simuleert inserts - de trigger-definities worden gerespecteerd, en de trigger code uitgevoerd.

Nadelen van triggers

Denk na over de volgende uitdagingen.

- Omdat triggers altijd uitgevoerd worden, moeten we wel heel voorzichtig zijn: een inefficiënte trigger op een zeer frequent uitgevoerde manipulatie kan de database performance drastisch naar beneden halen.

- Een klassieke load omzeilt de triggeractie.

- Te veel triggers kunnen 'trigger-chains' activeren. Inderdaad, een trigger vuurt door een DML-manipulatie een tweede trigger af en deze op zijn beurt een derde... Nog erger is een 'trigger-cycle' waarbij we in een trigger-lus verzeild raken. DB2 heeft wel een mechanisme dat na 16 opeenvolgende trigger-calls een SQLCODE -724 zal genereren. We moeten dus omzichtig omspringen met te veel triggers in de database.

- Triggers worden uitgevoerd in volgorde van definitie. Men moet er zorg voor dragen impliciet noch expliciet van deze ordening gebruik te maken (afhankelijk te worden).

Redundancy

Een perfect genormaliseerde database die aan alle relationele regels voldoet, mag geen redundante informatie bevatten. Dit is informatie die kan afgeleid worden met behulp van andere informatie in de database. Bijvoorbeeld totalen van rijen worden in een andere tabel bijgehouden; deze informatie is redundant want kan met de sum-functie rechtstreeks afgeleid worden van de bedragen in die rijen zelf.

Toch wordt deze vorm van denormalisatie dikwijls toegepast. Het geeft ons een aanzienlijk performance-voordeel bij het opvragen van die totalen want die moeten op dat moment niet meer berekend worden. We mogen deze strategie echter alleen maar overwegen wanneer veel meer data-access gedaan wordt dan datamanipulatie. De manipulatie van deze data vergt nu namelijk intensief onderhoud. In alle applicaties moeten we rekening houden met het up-to-date blijven van die afgeleide informatie.

Triggers & redundancy

Juist op dit vlak ligt de sterkte van triggers.

We kunnen door een eenmalige creatie van een aantal triggers, het onderhoud van de redundante informatie verzekeren terwijl ontwikkelaars daar in geen enkele applicatie hoeven rekening mee te houden.

Om dit alles te verduidelijken geven we een voorbeeld waarbij triggers en redundancy kunnen helpen om de performance van de data-access te verhogen.

Genormaliseerde database

We vertrekken van een genormaliseerde database die bestaat uit een stock-, order- en orderline-tabel (zie voorbeeld 1).

Voorbeeld 1: genormaliseerde structuur

stock	*order*	*orderline*
productId (PK)	orderId (PK)	orderId (PK1, FK -> order)
product	clientInfo	orderlineNr (PK2)
productPrice		productId (FK -> stock)
		nrOfProducts
		deliveryDate

De nodige views zijn aangemaakt op de database: vwstock, vworder en vworderline. En er zijn applicaties geschreven die de administratie van bestellingen afwerken en gebruik maken van deze tabellen. Deze tabellen blijken echter zeer veel geraadpleegd te worden en de database lijdt onder de vele joins die moeten gebeuren om ordertotalen te berekenen. Het totale bedrag van een order is telkens een join over de 3 tabellen en het bedrag per orderlijn moet berekend worden met info uit zowel de tabellen orderline als stock.

Belangrijk in ons voorbeeld is het feit dat het te betalen bedrag bepaald wordt op het ogenblik van de levering.

Denormalisatie: invoeren van redundancy

Denormalisatie van het datamodel ligt voor de hand, we zouden een kolom kunnen toevoegen aan de orderline-tabel: orderlineAmnt = productPrice * nrOfProducts. En we kunnen een kolom toevoegen aan de order-tabel: ordertotal die gelijk is aan SUM(orderlineAmnt) van alle orders met dezelfde orderId (zie voorbeeld 2).

Voorbeeld 2: gedenormaliseerde structuur

```
stock              order          orderline

productId (PK)     orderId (PK)   orderId     (PK1, FK -> order)
product            clientInfo     orderlineNr (PK2)
productPrice       ordertotal     productId   (FK -> stock)
                                  nrOfProducts
                                  deliveryDate
                                  orderlineAmnt
```

Deze beslissing is niet zo eenvoudig. De applicaties die zorgen voor het bevolken van deze tabellen zouden namelijk bij elke update, insert en delete moeten aangepast worden met extra coding met het oog op het bijwerken van deze redundante kolominformatie.

Invoeren van triggers

We beslissen om de kolommen toch toe te voegen maar de applicaties in oorspronkelijke staat te behouden (de views zullen toch niets ondervinden van de extra kolommen). Inserts, updates en deletes zullen applicatief onveranderd blijven. We zorgen ervoor dat de triggers op de achtergrond automatisch het onderhoud van de redundante informatie op zich nemen. De performance van onze data-access zal nu spectaculair stijgen. Totalen staan kant en klaar in de databases en kunnen nu met een één-tabel-select te voorschijn getoverd worden.

Welke triggers hebben we nodig?

Per tabel maken we een inventaris op over hoe een manipulatie de redundante informatie kan beïnvloeden. Om het overzicht te bewaren, bekijken we dit per kolom met redundante informatie; eerst kijken we naar welke manipulaties de orderlineAmnt onrechtstreeks aanpassen.

- stock-tabel: Delete- en insert-operaties zullen geen problemen geven, maar een update van de productPrice zal rechtstreeks de prijs van de nog niet geleverde maar reeds verkochte producten beïnvloeden (vergeef ons deze onrealistische business-rule)! Hier is dus een trigger nodig.

Voorbeeld 3: trigger bij prijswijziging

```
CREATE TRIGGER updateProductPrice
AFTER update of productPrice ON stock
REFERENCING new AS newrow
FOR EACH ROW MODE DB2SQL
    UPDATE orderline
       SET orderlineAmnt = newrow.productPrice * nrOfProducts
     WHERE productId = newrow.productId
       AND deliveryDate > CURRENT DATE
```

- order-tabel: Geen enkele manipulatie in deze tabel zal de orderlineAmnt veranderen.
- orderline-tabel: Bij een insert zullen we onmiddellijk de orderlineAmnt moeten berekenen (1 trigger), bij een update van productId

of nrOfProducts moeten we ook een trigger afvuren om de orderlineAmnt te wijzigen. Met een delete hoeven we hier geen rekening te houden.

Voorbeeld 4: trigger bij nieuwe of gewijzigde orderlijn

```
CREATE TRIGGER insertOrderline
AFTER insert ON orderline
REFERENCING new AS newrow
FOR EACH ROW MODE DB2SQL
UPDATE orderline
   SET orderlineAmnt =
       newrow.nrOfProducts * (SELECT productPrice
                              FROM stock
                              WHERE productId = newrow.productId)
 WHERE orderlinenr = newrow.orderlinenr
   AND orderId = newrow.orderId

CREATE TRIGGER updateOrderline
AFTER update OF nrOfProducts, productId ON orderline
REFERENCING new AS newrow
FOR EACH ROW MODE DB2SQL
UPDATE orderline
   SET orderlineAmnt =
       newrow.nrOfProducts * (SELECT productPrice
                              FROM stock
                              WHERE productId = newrow.productId)
 WHERE orderlinenr = newrow.orderlinenr
   AND orderId = newrow.orderId
```

Dit wil zeggen dat we voor het onderhouden van de orderlineAmnt slechts 3 triggers nodig hebben.

Wanneer deze triggers gecreëerd zijn, kunnen we dezelfde inventaris maken voor de ordertotal-kolom:

- stock: Zowel update, insert of delete op deze tabel hebben geen rechtstreekse invloed op de ordertotal-kolom. Wel onrechtstreeks natuurlijk via de orderlineAmnt, maar die invloed wordt opgevangen door de triggers die we hierboven hebben gedefinieerd.
- order: Ook hier zullen update, insert of delete geen invloed hebben op de ordertotal-kolom, tenzij de PK wordt geüpdated, maar dit wordt in een normale DB-omgeving niet toegelaten.
- orderline: Ook hier laten we geen PK-updates toe, zodanig dat we geen orderline-rijen kunnen veranderen van order. Maar een update van de orderlineAmnt zal de ordertotal-kolom wel rechtstreeks beïnvloeden. Ook na een delete van een orderline zal de ordertotal-kolom moeten aangepast worden. Bij een insert van een nieuwe rij, zullen we rekenen op trigger-cascading. Een insert zal immers automatisch een update van de orderlineAmnt teweegbrengen, en de update van de orderlineAmnt zal dan op zijn beurt een trigger afvuren die het ordertotal aanpast.

Voor het onderhoud van de ordertotal kolom zijn slechts 2 eenvoudige triggers nodig.

Voorbeeld 5: triggers op de orderline tabel

```
CREATE TRIGGER updateOrderlinePrice
AFTER update OF orderlinePrice ON orderline
REFERENCING old AS oldrow
            new AS newrow
FOR EACH ROW MODE DB2SQL
    UPDATE order
       SET ordertotal =
           ordertotal + newrow.orderlinePrice - oldrow.orderlinePrice
     WHERE orderId = newrow.orderId

CREATE TRIGGER deleteOrderline
AFTER delete ON orderline
REFERENCING old AS oldrow
FOR EACH ROW MODE DB2SQL
    UPDATE order
        SET ordertotal = ordertotal - oldrow.orderlinePrice
WHERE orderId = oldrow.orderId
```

Conclusie

Door het invoeren van redundancy vermijden we joins en vermijden we het gebruik van kolomfuncties om totalen te berekenen in queries; de performance van onze select's gaat er dus spectaculair op vooruit. Bovendien hebben we dit bereikt zonder applicatief wijzigingen aan te brengen, wat nog goedkoop is ook. Slechts 5 eenvoudige triggers, die we uiteraard eerst uitvoerig getest en geoptimaliseerd hebben, staan in voor het onderhoud van de redundante informatie. Ook data die we via een load moeten gaan toevoegen, zullen met een online 'load resume' de triggerfunctionaliteit respecteren.

Triggers zijn nuttig indien ze enkel gebruikt worden waar het voordeel overduidelijk is, waar we met een minimum aan administratie, een maximum aan rendement uit halen. En nu op zoek naar opportuniteiten in uw eigen database omgeving. Vergeet natuurlijk niet te overleggen met collega-databasegebruikers. Maar voor u daaraan begint, legt u ze best even dit artikel voor. Veel succes!

DB2 performance - case 3

Eric Venmans (ABIS)

Inleiding

In een eerste artikel over DB2 performance werd aangetoond dat optimaliseren te ver kan gaan. Het artikel had te maken met een toegangspad dat na optimalisering, enkel beter leek, maar niet beter was.

We beschrijven in dit artikel een ander voorbeeld van hetzelfde verschijnsel, maar in een andere context. We kunnen hierbij vooral leren hoe we kritisch moeten omgaan met de DB2 hulpmiddelen.

CASE 3: duurder via index!

Via een online transactie moeten we aan een gebruiker alle voorraden tonen die horen bij een magazijn. Voor het ophalen van de betrokken gegevens moeten we ons richten tot de tabel met voorraadinformatie. Deze heeft de structuur in voorbeeld 1 uiteengezet.

Voorbeeld 1: de tabel voorraden

```
Create TABLE Producten
...
Create TABLE Magazijnen
...

Create TABLE Voorraden
(VR_ProduktID      Char(6)        Not Null,
 VR_MagazijnID     Char(4)        Not Null,
 VRAantalStuks     Integer        Not Null with default,
 VRStatus          Char(1),
 VRLaatsteLevering Date,
 VRLaatsteVerkoop  Date,
Primary Key (VR_ProduktID, VR_MagazijnID),
Foreign key FK1 (VR_ProduktID)
        references Produkten on delete cascade,
Foreign key FK2 (VR_MagazijnID)
        references Magazijnen on delete restrict)

Create unique index PKVrrdIX
on Voorraden(VR_ProduktID, VR_MagazijnID) ...
```

De SQL vraag die we wensen te bestuderen is een erg eenvoudige query. Ze kan als volgt geformuleerd worden:

```
SELECT *
FROM Voorraden
WHERE VR_MagazijnID = :X
```

We onderzoeken de query in een testomgeving waar in de Catalog de productiestatistieken gekopieerd zijn. We gebruiken hiervoor een aantal alternatieve DB2 hulpmiddelen, en stellen het volgende vast:

- via de Plan_Table: dat DB2 een table scan verkiest;
- via de DSN_Statemnt_Table: dat de processing tijd wordt berekend op +/- 3500 msec;
- via DB2 Estimator: dat het fysiek lezen van de tabel wordt geschat op 15 sec.

Deze resultaten maken duidelijk dat de query niet meteen in aanmerking komt om in een online transactie te worden uitgevoerd (veronderstel dat de maximale antwoordtijd hiervoor is vastgelegd op 2 sec).

Optimalisatiepoging

Voor het optimaliseren van deze query zouden we DB2 kunnen 'verplichten' om toch een index te gebruiken. Een eerste manier om dit te doen is het gebruik van HINTs. Het toegangspad (table scan) overschrijven we in de Plan_Table met een toegangspad via de primary key index. Een andere manier om DB2 de index te laten gebruiken is het toevoegen van een 'dummy'-voorwaarde op VR_ProduktID (VR_ProduktID > :X maar met low-values in de variabele X).

Maar levert dit performance winst?

Volgens DB2 niet. De processing tijd, berekend voor het uitwerken van de query via de index, wordt geschat op ongeveer +/- 4500 msec. Bovendien wordt ook het fysiek lezen zwaarder.

Vanwaar deze hogere kost?

Bij nader onderzoek blijken er 3 belangrijke elementen mee te spelen:

- als de index gebruikt wordt, is dit via een sequentieel lezen van de ganse index; VR_MagazijnID is het tweede deel in elke indexelement; d.w.z. dat alle verwijzingen voor het gevraagde magazijn verspreid zitten over de ganse index;
- de index is niet veel kleiner dan de tabel: indexelement (1 per rij) is fysiek 17 bytes lang; terwijl een rij een fysieke lengte heeft van 31 bytes; d.w.z. dat het sequentieel lezen van de tabel maar dubbel zolang duurt als het sequentieel lezen van de index;
- er zijn bovendien maar een 100-tal magazijnen; dus zowat één op honderd rijen worden via onze query aangeduid; dat is meer dan 1 per page (er kunnen voor onze tabel een 130-tal rijen in één page); indexinformatie weet dan wel aan te geven waar de gezochte rijen zitten in die pages, maar de pages zelf zullen toch zo goed als allemaal moeten gelezen worden.

We kunnen nu een optelsom maken van wat er moet gebeuren bij het lezen van de tabel via de index. Het resultaat hiervan is duidelijk hoger dan het aantal acties nodig voor het direct lezen van de tabel zonder de 'omweg' via de index:

- Indexgebruik: lezen van alle index pages; controleren van alle indexelementen (= aantal rijen); lezen van bijna alle pages; verwerken van +/- 1% van de rijen;
- Table scan: lezen van alle table pages; controleren van alle rijen; verwerken van +/- 1% van de rijen.

Besluit

Het forceren van indexgebruik is in het aangehaalde voorbeeld geen verbetering van de performance. De selectiviteit van de voorwaarde is te laag en bovendien is de index minder geschikt. De betrokken informatie zit niet vooraan in de indexelementen.

Kunnen we deze query dan niet verder optimaliseren?

Toch wel, maar dan moeten we aan de indexen zelf sleutelen. Ofwel maken we een extra index op alleen VR_MagazijnID (geen rechtstreekse invloed op bestaande select statements), ofwel wijzigen we de volgorde van de kolommen in de bestaande primary key index. Daardoor komen de indexelementen in volgorde van VR_MagazijnID. Dit laatste geeft allicht ernstige performanceproblemen voor queries die dan weer selecteren via VR_ProduktID. Bovendien zal echte performancewinst maar gehaald worden wanneer deze index (de nieuwe of de gewijzigde) als cluster index worden gedefinieerd. Hierdoor komen alle rijen met eenzelfde VR_MagazijnID fysiek bij mekaar in de tablespace (nog slechts +/- 1% van de table pages moet gelezen worden).

Uiteindelijk bepaalt de database administrator wat er eventueel verandert aan de datastructuren. We mogen immers veronderstellen dat hij of zij een overzicht heeft over hoe de betrokken tabel gebruikt wordt (of zal gebruikt worden).

DOSSIER 8

Multiple row insert

In het verleden was het mogelijk meerdere rijen in één instructie toe te voegen aan een DB2-tabel, als deze rijen afkomstig waren uit een andere DB2-tabel. Vanaf DB2 versie 8 is het echter ook mogelijk een host-rij-structuur als bron te gebruiken voor dergelijke insert. En kunnen meerdere rijen aan de hand van één instructie worden opgehaald uit DB2. Hierover later meer.

EXEC SQL INSERT INTO t FOR :n ROWS VALUES (:r1, :r2) ATOMIC;

Hostvariabele 'n' geeft weer hoeveel rijen moeten worden toegevoegd aan tabel 't', met 'r1' en 'r2' de host variabelen, gedefinieerd als 'rijen'. Hoe de rij wordt gedefinieerd, hangt af van de specificatie van de gebruikte host-taal, bijvoorbeeld:

```
01 INV.
   05 r1 PIC S9(4) COMP-4 occurs 10 times.
    05 r2 PIC X(10) occurs 10 times.
```

De lengte van beide rijen moet minstens gelijk zijn aan 'n', of DB2 genereert een fout. Indien één van beide variabelen een gewone variabele is, wordt deze 'n' maal toegevoegd. Er treden geen problemen op, indien 'n' kleiner is dan de lengte van de gebruikte rijen.

Null-indicator-rijen worden ondersteund.

De specificatie 'ATOMIC' geeft weer dat een fout tijdens de insert voldoende is om alle rijen te weigeren. Optie 'NOT ATOMIC CONTINUE ON SQLEXCEPTION' zorgt ervoor dat rijen die niet kunnen worden toegevoegd aan een tabel worden verworpen; de andere rijen worden gewoon toegevoegd.

Een nieuw statement 'GET DIAGNOSTICS' laat toe om deze individuele rijen te identificeren en te corrigeren. Het gebruik van de SQLCA-structuur is hier minder voor geschikt. Immers, dit nieuwe statement biedt naast een SQL-returncode voor het volledige statement, ook een SQL-returncode voor elke individuele actie, bijvoorbeeld in de context van een multiple row insert. De output van dit statement is typisch een rij-structuur. Aan de hand van een eerste oproep van dit statement wordt bepaald hoeveel gegevens (e.g. SQL-returncodes) in deze rij aanwezig zijn; middels een lusstructuur worden deze codes één na één opgevraagd, en op een passende wijze afgehandeld.

Kris Van Thillo (ABIS)

CURSUSPLANNING DEC 2003 - MRT 2004

DB2 concepten	375 EUR	09/12(W), 08/03(L)
DB2 for OS/390, een totaaloverzicht	1625 EUR	08-12/12 (W), 12-16/01 (L), 26-30/01(W), 01-05/03 (L), 29/03-02/04 (W)
DB2 UDB, een totaaloverzicht	1625 EUR	01-05/03 (L)
RDBMS concepten	325 EUR	08/12 (W), 12/01 (L), 26/01 (W), 01/03 (L), 29/03 (W)
Basiskennis SQL	325 EUR	09/12 (W), 13/01(L), 27/01(W), 02/03 (L), 30/03 (W)
DB2 for OS/390 basiscursus	975 EUR	10-12/12 (W), 14-16/01 (L), 28-30/01(W), 03-05/03 (L), 31/03-02/04 (W)
DB2 UDB basiscursus	975 EUR	03-05/03 (L)
SQL workshop	700 EUR	18-19/12 (W), 09-10/02 (W), 11-12/03 (L)
DB2 for OS/390 programmering voor gevorderden	1050 EUR	03-05/12 (L), 08-09/03 (L)
DB2 for OS/390: SQL performance	1200 EUR	15-17/12 (L), 24-26/03 (L)
DB2 UDB applicatieperformance	400 EUR	08/06 (W)
Database applicatieprogrammering met Java	800 EUR	29-30/04 (L)
Fysiek ontwerp van relationele databases.	700 EUR	03-04/05 (L)
DB2 for OS/390 database administratie	1600 EUR	02-05/12 (L), 15-18/03 (W)
DB2 for OS/390 restart and recovery	1650 EUR	10-12/12 (UK), 11-13/02 (UK)
DB2 UDB systeembeheer en performance	400 EUR	15/12 (L)
DB2 UDB en zijn extenders: XML en text search	200 EUR	16/12 (L), 12/03 (L)
DB2 UDB integratie met MQSeries	200 EUR	16/12 (L), 12/03 (L)

Plaats: L = Leuven; W = Woerden; details en extra cursussen: www.abis.be

Postbus 220
Diestsevest 32
BE-3000 Leuven
Tel. 016/245610
Fax 016/245691
training@abis.be

abis
TRAINING & CONSULTING

Postbus 122
Pelmolenlaan 1-K
NL-3440 AC Woerden
Tel. 0348-435570
Fax 0348-432493
training@abis.be