EMBEDDED SQL

Inleiding

In het boek Databases & SQL wordt beschreven hoe opdrachten in de programmeertaal SQL gebruikt worden om de inhoud van een relationele database te raadplegen en te bewerken. SQL wordt daarbij als een op zichzelf staande taal gebruikt (self contained language).
Opdrachten geschreven in SQL kunnen ook toegepast worden als onderdeel van programma's die bijvoorbeeld in COBOL of C geschreven zijn. We spreken dan van 'embedded SQL'. Talen zoals bijvoorbeeld COBOL, C of FORTRAN zijn in dat geval hosttalen.
In dit artikel beschrijven we hoe SQL als embedded taal gebruikt wordt.

Het volgende komt aan de orde:
- hoe we SQL-opdrachten tussen de programmaopdrachten van een toepassingsprogramma opnemen
- hoe we met variabelen van de hosttaal in SQL-opdrachten omgaan
- hoe we met de resultaten van queries en met updateopdrachten werken
- hoe we met transacties omgaan in embedded SQL
- hoe we met foutmeldingen omgaan
- overzicht van de voorbeelden van embedding in COBOL

Voor de voorbeelden maken we gebruik van de voorbeelddatabases uit het boek Databases & SQL, paragraaf 6.2:
- werkverdelingdatabase
- bibliotheekdatabase

Embedden en hostvariabelen

SQL-opdrachten scheiden van opdrachten van het hostprogramma
Belangrijk is dat de compiler van het toepassingsprogramma de SQL-opdrachten herkent en aan SQL zelf aanbiedt. Een COBOL-compiler verwacht daarvoor dat iedere SQL-opdracht wordt voorafgegaan door:

```
EXEC SQL
```

en wordt gevolgd door:

```
END-EXEC.
```

voorbeeld

```
EXEC SQL
UPDATE VESTIGING
SET PLAATS = 'EDE'
WHERE VESNAAM = 'SNELHAP'
END-EXEC.
```

hostvariabelen in SQL-opdrachten
In de SQL-opdrachten kunnen we op de daarvoor bestemde plaatsen hostvariabelen van het toepassingsprogramma gebruiken. Deze hostvariabelen worden voorafgegaan van een dubbele punt om ze van de kolomnamen van tabellen te onderscheiden:
`:hostvariabelenaam`

voorbeeld

```
EXEC SQL
DELETE FROM VESTIGING
WHERE VESNAAM = :VESTIGINGNAAM
END-EXEC.
```

Queries en update-opdrachten

werken met een cursor
Aangezien een toepassingsprogramma slechts één rij uit een tabel tegelijk kan afhandelen en een SELECT-opdracht (in SQL) een tabel met mogelijk meer dan één rij oplevert, is er voor het toepassingsprogramma een mechanisme nodig om rij voor rij door de aangeboden tabel te gaan. Dit mechanisme is een soort wijzer die naar een rij van de aangeboden tabel wijst. Het toepassingsprogramma kan met de waarden van die rij aan de slag. We noemen zo'n wijzer een cursor.

Een SELECT-opdracht kan alleen met behulp van zo'n cursor worden benaderd (uitzondering hierop volgt verderop met SELECT INTO). Een cursor moet gedeclareerd worden, geopend worden, gebruikt worden en gesloten worden met aparte opdrachten.

De opdracht voor het declareren is:
`DECLARE cursornaam CURSOR FOR select-opdracht`

(waar kleine letters gebruikt zijn moet iets ingevuld worden)

voorbeeld

```
EXEC SQL
DECLARE CUR1 CURSOR FOR
     SELECT W1.W#, W1.SALARIS,
               W2.W#, W2.SALARIS
     FROM WERKNEMER W1,
          WERKNEMER W2,
          VERVANGING V
     WHERE W1.W#  = V.VERVANGENE
     AND V.VERVANGER = W2.W#
     AND W1.FNAAM = 'DIRECTEUR'
     ORDER BY W1.WNAAM
END-EXEC.
```

openen cursor

De opdacht voor het openen van de cursor is:

```
OPEN cursornaam
```

voorbeeld

```
EXEC SQL
OPEN CUR1
END-EXEC.
```

opdracht FETCH

De opdracht voor het binnenhalen van rijen vanaf een cursor is:

```
FETCH cursornaam INTO :variabele1
                   [,variabale2]...
```

Er moeten precies evenveel variabelen zijn als dat de SELECT-opdracht kolommen in de output oplevert. Bovendien moeten de variabelen zo gedefinieerd zijn dat ze de aangeboden waarden kunnen bevatten.
Een door een cursor aangeduide rij kan ook gewijzigd en zelfs verwijderd worden.

voorbeeld

```
EXEC SQL
FETCH CUR1
     INTO :VERVANGENE,
          :SAL1,
          :VERVANGER,
          :SAL2
END-EXEC.
```

opdracht UPDATE

Voor het wijzigen is de opdracht:

```
UPDATE tabelnaam
SET setbeschrijving
WHERE CURRENT OF cursornaam
```

voorbeeld

(aan de hand van een cursosr CUR3 die langs de rijen van de tabel WERKNEMER loopt. De werknemer krijgt nadat in het host-programma besloten is dat deze er voor in aanmerking komt een loonsverhoging)

```
EXEC SQL
UPDATE WERKNEMER
SET SALARIS =
SALARIS * (1+
  ( :VERHOGINGSPERCENTAGE / 100))
WHERE CURRENT OF CUR3
END-EXEC.
```

opdracht DELETE

De opdracht voor het verwijderen van de rij die door de cursor wordt aangewezen is:

```
DELETE FROM tabelnaam
WHERE CURRENT OF cursornaam
```

(de in de UPDATE- en DELETE-opdrachten genoemde tabelnamen moeten aansluiten op de door de cursor aangeduide tabel. Dit is alleen mogelijk als de cursor een SELECT-opdracht bevat met bij FROM slechts één tabel. Verder moet de output van de SELECT-opdracht betrekking hebben op één bepaalde rij van de tabel)

voorbeeld

(in het voorbeeld wordt ook weer met de cursor langs rijen van de tabel WERKNEMER gelopen. Het hostprogramma besluit welke werknemers weg moeten. Is dat het geval dan wordt de volgende opdracht uitgevoerd)

```
EXEC SQL
DELETE FROM WERKNEMER
WHERE CURRENT OF CUR6
END-EXEC.
```

afsluiten cursor

De opdracht om een cursor af te sluiten is:

```
CLOSE cursornaam
```

(de verderop behandelde opdracht COMMIT heeft een impliciete close in zich voor alle cursors)

voorbeeld

```
EXEC SQL
CLOSE CUR1
END -EXEC.
```

werken met SELECT INTO

Op het bovenstaande is één uitzondering. Dat is de SELECT INTO-opdracht. Deze opdracht sluit in zijn werking aan op het werken van het hostprogramma met slechts één rij tegelijkertijd. Dit komt omdat deze

opdracht slechts één rij uit de database mag opleveren. Doet het dat niet, dan volgt een foutsituatie.
De SELECT INTO-opdracht ziet er als volgt uit:

```
SELECT outputspecificatie
INTO :hostvariabele1
     [, hostvariabele2]...
FROM tabel(-len)
[WHERE search condition]
```

voorbeeld 1

```
     EXEC SQL
     SELECT    LNAAM,
               GEBDAT,
               ADRES,
               WPLAATS
     INTO :LENERNAAM,
          :GEBDAT,
          :ADRES,
          :WOONPLAATS
     FROM LENER
     WHERE LNAAM = 'VONK'
     END-EXEC.
```

voorbeeld 2

```
     EXEC SQL
     SELECT    SUM(SALARIS),
               MAX(SALARIS),
               MIN(SALARIS)
     INTO      :TOTAALSALARIS,
               :HOOGSTESALARIS,
               :LAAGSTESALARIS
     FROM WERKNEMER
     WHERE VESNAAM = 'HOK-O-TEL'
     END-EXEC.
```

Transacties

wat is een transactie

Een transactie is een logische eenheid van werk dat door een programma wordt uitgevoerd. De hoeveelheid werk moet zo gekozen zijn dat na de transactie de gegevens in de database voldoen aan alle integriteitseisen. Een probleem bij een database is dat tegelijkertijd meer dan één transactie (door verschillende programma's) op de database losgelaten kunnen worden. Een transactie kan hierdoor te maken krijgen met de tussenresultaten van andere transacties. Dit kan gedeeltelijk of geheel vermeden worden door transacties van elkaar te isoleren. Dit isoleren moet echter wel vanuit het programma ingesteld worden. We zullen hierna aangeven hoe dit moet.

transacties in embedded SQL

In embedded SQL kunnen transacties uit meer dan één SQL-opdracht bestaan. Voor het in goede banen leiden van transacties moet duidelijk zijn wanneer een transactie begint en wanneer dezelfde transactie eindigt. Verder moet een transactie ongedaan gemaakt kunnen worden. Tenslotte moeten transacties van elkaar geen last hebben.

begin en einde van een transactie

Een transactie begint met een willekeurige SQL-opdracht en eindigt met de opdracht COMMIT. Dit betekent dat er achter elkaar heel veel SQL-opdrachten uitgevoerd kunnen worden in één transactie. De syntaxis is:

```
COMMIT
```

voorbeeld:

```
     EXEC SQL
     COMMIT
     END-EXEC.
```

ongedaan maken van een transactie

Een transactie kan ongedaan worden gemaakt door de opdracht:

```
ROLLBACK
```

Deze opdracht zorgt ervoor dat de transactie waar het initiërende programma mee bezig is, wordt teruggedraaid. Alles wat de transactie aan veranderingen in de database heeft aangebracht wordt ongedaan gemaakt. Een neveneffect van de opdracht ROLLBACK is dat alle cursors weer worden gesloten.

voorbeeld

```
     EXEC SQL
     ROLLBACK
     END-EXEC.
```

programma's die tegelijkertijd transacties laten uitvoeren

Wanneer we transacties tegelijkertijd uitvoeren dan kunnen verschillende problemen optreden. Deze problemen zijn:
- dirty read
- non-repeatable read
- phantoms

dirty read

Dirty read houdt in dat een transactie werkt met rijen die niet hebben bestaan of niet hebben bestaan in de vorm waarin de transactie ze heeft ingelezen.
Dit zou als volgt kunnen voorkomen. Stel transactie

T1 wijzigt een rij of maakt een nieuwe rij aan. Stel dat transactie T2 deze gewijzigde of nieuwe rij leest. Stel dat transactie T1 een ROLLBACK doet waardoor de wijziging op de rij of de rij in zijn geheel nooit heeft bestaan. Dan heeft transactie T2 gewerkt met een rij waarin een wijziging zit die nooit heeft bestaan of met een rij die zelf nooit heeft bestaan..

non-repeatable read
Non-repeatable read houdt in dat een transactie voordat de transactie klaar is te maken krijgt met wijzigingen van rijen waarmee de transactie bezig is.
Dit zou als volgt kunnen voorkomen. Stel transactie T1 leest een rij. Stel transactie T2 wijzigt of verwijdert deze rij. Stel transactie T1 probeert de rij nogmaals te lezen. Dan ontdekt transactie T1 dat de rij gewijzigd is of verwijderd is. Dit terwijl transacties graag willen dat tijdens het uitvoeren van de transactie andere transacties geen wijzigingen op de door hen gebruikte rijen uitvoeren.

phantoms
Het verschijnsel phantoms houdt in dat er in de loop van een transactie ineens nieuwe rijen opduiken in de database die er eerder tijdens de transactie niet waren.
Dit zou als volgt kunnen voorkomen. Stel transactie T1 werkt met een verzameling rijen. Deze rijen zijn ingelezen voordat transactie T2 een rij toevoegt die voldoet aan de voorwaarden waaronder de verzameling rijen door transactie T1 is opgehaald. Stel transactie T1 wil op basis van de eerder opgehaalde verzameling een wijziging in de database aanbrengen. Het zou kunnen zijn dat deze wijziging niet meer lukt door de nieuw toegevoegde rij. Dit kan komen doordat bijvoorbeeld een CHECK-constraint een overtreding constateert die op basis van de eerdere verzameling niet zou zijn geconstateerd. Dan heeft transactie T1 last van een phantom.

vrijwaren transacties voor problemen veroorzaakt door gelijktijdig voorkomen andere transacties
Voor het omgaan met de problemen die het gevolg zijn van het gelijktijdig gebruik van de database door meer dan één toepassingsprogramma moeten twee zaken worden ingesteld:
- access mode
- isolation level

access mode
De access mode geeft aan of een programma alleen uit de database leest of naast lezen ook wijzigingen in de database mag aanbrengen. Dit geven we als volgt aan:

```
SET TRANSACTION
        {READ ONLY | READ WRITE}
```

De access mode READ ONLY geeft aan dat (met uitzondering van tijdelijke tabellen) een programma alleen mag lezen uit de database.
De access mode READ WRITE geeft aan dat een programma zowel mag lezen uit de database als veranderingen mag aanbrengen in de database. Dit is tevens de default instelling.

voorbeeld
```
EXEC SQL
SET TRANSACTION READ ONLY
END-EXEC.
```

isolation level
Het isolation level geeft aan in hoeverre transacties elkaar één of meer van de genoemde problemen (dirty read, non repeatable read, phantom read) mogen bezorgen. Bij het schrijven van een programma dient men zich dus bewust te zijn welke problemen wel en welke problemen niet kunnen voorkomen. Het programma dient te toetsen of de problemen voorkomen. De mogelijke isolation levels zijn:
- READ UNCOMMITTED; hierbij kunnen dirty read, repeatable read en phantom read problemen voorkomen
- READ COMMITTED; hierbij kunnen non repeatable read en phantom read problemen voorkomen
- REPEATABLE READ; hierbij kunnen phantom read problemen voorkomen
- SERIALIZABLE; hierbij kunnen geen van de genoemde problemen voorkomen (deze instelling vreet wel aan de performance om aan deze eisen te voldoen)

Het isolation level kan als volgt worden aangegeven:
```
SET TRANSACTION ISOLATION LEVEL
     { READ UCOMMITTED |
       READ COMMITTED  |
       REPEATABLE READ |
       SERIALIZABLE}
```

voorbeeld
```
EXEC SQL
SET TRANSACTION ISOLATION LEVEL
                REPEATABLE READ
END-EXEC.
```

Foutmeldingen

Met het uitvoeren van de SQL-opdrachten kan van

alles mis gaan. Het host-programma moet dit kunnen merken. Op basis van de fout moet de loop van het hostprogramma kunnen worden veranderd. Er kunnen twee meldingsmechanismen worden gebruikt:

```
SQLERROR
SQLSTATE (in plaats van het vroegere
                        SQLCODE)
```

SQLERROR

SQLERROR kan gebruikt worden door in het host-programma de opdracht WHENEVER op te nemen. We zullen dit laten zien voor COBOL als hosttaal. De syntaxis is:

```
WHENEVER SQLERROR GOTO :hostlabel
```

(de opdracht maakt deel uit van de SQL-opdrachten)

voorbeeld

```
EXEC SQL
WHENEVER SQLERROR
          GO TO :HANDEL-ERROR-AF
END-EXEC.
```

SQLSTATE

SQLSTATE kan voor alles en nog wat gebruikt worden. Er kan precies uit afgelezen worden wat als fout geconstateerd is. Het belangrijkste gebruik is echter het constateren dat er geen verdere data gevonden kan worden. Dit levert de code '02000'op. SQLSTATE moet als variabele van het hostprogramma worden gedeclareerd en op de waarde '00000' gezet worden. SQLSTATE wordt vanuit het systeem van waarden voorzien.

voorbeeld

```
PERFORM BEWERKING
     UNTIL SQLSTATE = '02000'.
```

Voorbeelden

Hierna worden de voorbeelden herhaald die eerder werden gegeven. Voor het voorbeeld van het gebruik van een cursor geldt dat hier ook de samenhang van de opdrachten in beeld komt. De voorbeelden betreffen:

- voorbeeld UPDATE-opdracht zonder gebruik van cursor of hostvariabelen
- voorbeeld UPDATE-opdracht bij het gebruik van een cursor
- voorbeeld DELETE-opdracht bij het gebruik van een cursor
- voorbeeld van het gebruik van een cursor
- twee voorbeelden van het gebruik van SELECT INTO

voorbeeld UPDATE-opdracht

```
EXEC SQL
UPDATE VESTIGING
SET PLAATS = 'EDE'
WHERE VESNAAM = 'SNELHAP'
END-EXEC.
```

voorbeeld UPDATE-opdracht met gebruik van hostvariabele

```
EXEC SQL
DELETE FROM VESTIGING
WHERE VESNAAM = :VESTIGINGNAAM
END-EXEC.
```

voorbeeld UPDATE-opdracht bij het gebruik van een cursor

(aan de hand van een cursosr CUR3 die langs de rijen van de tabel WERKNEMER loopt. De werknemer krijgt nadat in het host-programma besloten is dat de werknemer er voor in aanmerking komt een loonsverhoging)

```
EXEC SQL
UPDATE WERKNEMER
SET SALARIS = SALARIS *
  (1+
  ( :VERHOGINGSPERCENTAGE / 100))
WHERE CURRENT OF CUR3
END-EXEC.
```

voorbeeld van het gebruik van een cursor

Onderstaand voorbeeld laat zien hoe een deel van een COBOL-programma zich bezig houdt met het toekennen van een hoger salaris aan vervangers van werknemers. Als database is genomen de voorbeeld database werkverdeling uit het boek Databases & SQL, paragraaf 6.2. Het voorbeeld is in stukjes al gebruikt in het voorafgaande.

Het deel van het COBOL-programma ziet er als volgt uit:

```
*VERVANGERS VAN DIRECTEUREN KRIJGEN
*HETZELFDE SALARIS ALS DEGENE DIE
*ZE VERVANGEN.
     EXEC SQL
          DECLARE CUR1 CURSOR FOR
               SELECT    W1.W#,
                         W1.SALARIS,
                         W2.W#,
                         W2.SALARIS
               FROM WERKNEMER W1,
                    WERKNEMER W2,
```

```
                    VERVANGING V
               WHERE W1.W#  =
                         V.VERVANGENE
               AND V.VERVANGER = W2.W#
               AND W1.FNAAM =
                         'DIRECTEUR'
               ORDER BY W1.WNAAM
     END-EXEC.
     EXEC SQL
          WHENEVER SQLERROR
               GO TO :HANDEL-ERROR-AF
     END-EXEC.
     EXEC SQL
          OPEN CUR1
     END-EXEC.
     MOVE '00000' TO SQLSTATE.
     EXEC SQL
          FETCH CUR1
               INTO :VERVANGENE,
                    :SAL1,
                    :VERVANGER,
                    :SAL2
     END-EXEC.
     IF SQLSTATE = '02000'
                         GO TO FINISH.
     PERFORM BEWERKING
          UNTIL SQLSTATE = '02000'.
     EXEC SQL
          CLOSE CUR1
     END-EXEC.
     EXEC SQL
          COMMIT
     END-EXEC.
     GO TO FINISH.
BEWERKING.
     MOVE SAL1 TO SAL2.
     EXEC SQL
          UPDATE WERKNEMER
          SET SALARIS = :SAL2
          WHERE W#  = :VERVANGER
     END-EXEC.
     EXEC SQL
          FETCH CUR1
               INTO :VERVANGENE,
                    :SAL1,
                    :VERVANGER,
                    :SAL2
     END-EXEC.
HANDEL-ERROR-AF.
     DISPLAY
          "ER IS EEN FOUT OPGETREDEN".
FINISH.
     STOP RUN.
```

twee voorbeelden van het gebruik van SELECT INTO

Hieronder staan de twee voorbeelden die al eerder als voorbeeld van het gebruik van de 'SELECT INTO'-opdracht zijn gegeven.
Let op hoe gezorgd is dat in de output niet meer dan één rij kan voorkomen.

voorbeeld 1

```
SELECT    LNAAM,
          GEBDAT,
          ADRES,
          WPLAATS
INTO      :LENERNAAM,
          :GEBDAT,
          :ADRES,
          :WOONPLAATS
FROM LENER
WHERE LNAAM = 'VONK'
```

voorbeeld 2

```
SELECT    SUM(SALARIS),
          MAX(SALARIS),
          MIN(SALARIS)
INTO      :TOTAALSALARIS,
          :HOOGSTESALARIS,
          :LAAGSTESALARIS
FROM WERKNEMER
WHERE VESNAAM = 'HOK-O-TEL'
```