Algemene richtlijnen binnen de mogelijkheden van RDBMS en 4GL

Foutafhandeling in SQL

Toon Loonen

Foutafhandeling van SQL-code is op veel projecten een achtergebleven gebied. Soms is er discussie of het nodig is en vaak is er erkenning dat het nodig is, maar wordt het niet of niet compleet toegepast. In dit artikel geeft Toon Loonen aan waar foutafhandeling wel of niet nodig is, hoe het toegepast kan worden en beschrijft enkele bijzondere situaties: concurrent updates en deadlocks.

Op verschillende manieren kan SQL worden gebruikt. Interactief, in een query tool in UNIX (Oracle 'sqlplus' of Sybase 'isql'), in Windows (Oracle SQLPlus, Sybase SQL Advantage, Microsoft SQL Server Query Analyzer) of in een third party tool zoals bijvoorbeeld DBArtisan of TOAD.

Maar ook in een SQL scriptfile die via een van deze tools wordt uitgevoerd, verder kan het embedded worden gebruikt in een 3GL (C, COBOL, Java) of 4GL (Uniface, Powerbuilder, Advantage: Gen, Oracle Developer, Visual Basic en diverse Java Generatoren) en tenslotte in een stored procedure, functie of trigger.

WANNEER FOUTAFHANDELING TOEPASSEN

Voor geheel interactief gebruik van SQL in bovengenoemde query tools is er geen foutafhandeling nodig. Bij een fout in de query, zowel een syntax-fout als een verkeerde tabelnaam of kolomnaam, komt er onmiddellijk een foutmelding in plaats van het verwachte antwoord. Men kan de fout direct corrigeren en de query opnieuw aanbieden. Natuurlijk worden fouten met betrekking tot logica in de query niet opgemerkt, men zou bijvoorbeeld bij een rapport met kosten gemakkelijk de BTW-bedragen kunnen vergeten! Voor dit soort fouten zijn testen nodig: probeer de query eerst uit op een eenvoudige situatie waarvan de goede uitkomst bekend is, voordat deze gebruikt wordt voor belangrijk nieuw werk.

Bij een fout in de query komt er onmiddellijk een foutmelding in plaats van het verwachte antwoord

Het volgende script bevat een fout: Een *insert duplicate key* op de werktabel:

```
create table wrk_names (name char(32))
create unique index i1 on wrk_names(name)
insert into wrk_names values ("XX")
insert into wrk_names values ("XX")
select name from wrk_names
```

Als dit SQL-script in een batch wordt uitgevoerd, bijvoorbeeld in UNIX met het Sybase tool 'isql', zal er een fout optreden bij de tweede insert. Zonder foutafhandeling gaat het script echter verder met het laatste statement, de select. In de uitvoer komt zowel de fout-melding als het resultaat van de select te staan. Komt er na de fout echter nog veel uitvoer, dan is het goed mogelijk dat de fout niet meer wordt opgemerkt, maar het resultaat is, zeer waarschijnlijk, fout.

Dit soort scripts wordt interactief uitgevoerd door de gebruiker of automatisch als onderdeel van een grotere batch, bijvoorbeeld als een nachtelijke run met rapportages of bij de installatie van (een nieuwe versie van) het systeem. In alle gevallen is het gewenst om via foutafhandeling de gebruiker of beheerder op de fouten te wijzen en de delen van een script na het foutief uitgevoerde statement niet meer uit te voeren.

Bovenstaande coding kan ook in een 3GL- of 4GL-programma staan. Zo'n programma zal elk statement afzonderlijk uitvoeren waarna het telkens controleert of dit statement correct is uitgevoerd of zal het geheel als één batch aan de databaseserver aanbieden.

In het laatste geval zal ook foutafhandeling tussen elk statement van de batch moeten worden ingebouwd, anders zal het programma waarschijnlijk de fout niet opmerken (de foutcode na het laatste statement is weer 0 omdat het laatste statement correct verwerkt is) en zal de coding na de fout voor niets worden uitgevoerd en zal mogelijk het resultaat ten onrechte als correct aan de gebruiker worden gepresenteerd.
Dezelfde coding zou ook in een *stored procedure* of *trigger* kunnen staan:

```
create procedure wrk_names_p as
begin
   insert into wrk_names values (“XX”)
   insert into wrk_names values (“XX”)
   select name from wrk_names
end
```

Bij het aanroepen van deze procedure (interactief of vanuit een programma) wordt de select na de foutieve insert ook uitgevoerd en mogelijk de fout niet opgemerkt. Bij een stored procedure moet daarom na elk statement op fouten gecontroleerd worden.

Bij batchprogramma's zal de verwerking van dit programma worden afgebroken

Bij een fout wordt de stored procedure afgebroken en wordt aan het aanroepende programma doorgegeven dat er een fout is opgetreden. Dit laatste programma moet hierop controleren en weer de daar gewenste foutafhandeling uitvoeren.

Bij nesting van stored procedures (programma A voert procedure A1 uit, die weer procedure A11 uitvoert) moet de fout doorgegeven worden naar het hoogste SQL-niveau, in programma A. Hetzelfde geldt voor triggers en andere coding.

HOE FOUTAFHANDELING TOEPASSEN

De werkwijze voor foutafhandeling is productafhankelijk. De onderstaande coding is uitgevoerd in Sybase en MS SQL Server, maar zal er bij andere producten wat anders uitzien.

Voor SQL-batches:

```
declare @error int
insert into wrk_names values (“XX”)
select @error = @@error
if @error = 0
begin
   insert into wrk_names values (“XX”)
   select @error = @@error
end
if @error = 0
begin
   select name from wrk_names
   select @error = @@error
end
```

Voor een stored procedure ziet de coding er als volgt uit:

```
create procedure wrk_names_p as
begin
   declare @error int
   insert into wrk_names values (“XX”)
   select @error = @@error
   if @error != 0
   begin
      return @error
   end

   insert into wrk_names values (“XX”)
   select @error = @@error
   if @error != 0
   begin
      return @error
   end
   select name from wrk_names
   select @error = @@error
   return @error
end
```

Deze procedure wordt als volgt uitgevoerd:

```
declare  @error int, @error_proc int
exec     @error_proc = wrk_names_p
select   @error = @@error
if not   (@error = 0 and @error_proc = 0) then ...
```

Bij het uitvoeren van deze code krijgt:

- @error de waarde 0: de procedure is correct uitgevoerd (indien de procedure niet (meer) bestaat zou hier een andere waarde kunnen komen);
- @error_proc de waarde 2601: De Sybase error voor duplicate key, de fout die in de procedure geconstateerd en teruggegeven was.

Deze controles kunnen worden uitgebreid met andere, meer functionele controles, bijvoorbeeld of, bij een update of delete, het aantal verwerkte rijen groter is dan 0. Hierbij horen dan eigen foutcodes.

Zoals hierboven te lezen is, maakt foutafhandeling de coding er niet echt leesbaar op. Sommige producten hebben in stored procedures een 'on exception' paragraaf, bijvoorbeeld in Informix:

```
create procedure wrk_names_p
on exception in (...)
   –  vul hier de specifieke foutnummers in
   –  bijvoorbeeld insert is niet uitgevoerd
      i.v.m. duplicate key error
   –  processing mag doorgaan na bijvoorbeeld een
      update i.p.v. de insert
   update wrk_names set …
end exception with resume

on exception
   –  bij andere fouten verwerking afbreken (geen
      'with resume' optie)
end exception
```

```
begin
   insert into wrk_names values (“XX”);
   insert into wrk_names values (“XX”);
   select name from wrk_names;
end
end procedure;
```

En in Oracle:

```
create procedure wrk_names_p
as
begin
   insert into wrk_names values (“XX”);
   insert into wrk_names values (“XX”);
   select name from wrk_names;
   return 0;
exception
   when no_data_found then raise_application_
   error(..., fouttekst);
   when other then raise_application_error(...,
   fouttekst);
end
```

Hier blijft de functionele coding goed leesbaar en is de foutafhandeling in een eigen hoekje van de procedure weggestopt. Door voor procedures en triggers een template of sjabloon te maken waar alle standaard coding en commentaar reeds in staat, kost het schrijven van een procedure met goede foutafhandeling niet veel extra werk (zie ook [1]). Sybase heeft deze ‘on exception’ mogelijkheid niet. Oracle kent de resume optie niet om door te gaan na een acceptabele fout.

De procedure die ik in Sybase hanteer bij het bouwen van procedures is:

- Eerst de procedure bouwen en testen zonder foutafhandeling;
- Daarna bij alle statements de standaard coding toevoegen, bijvoorbeeld:

```
select @error = @@error
if @error != 0 return @error
```

In een 3GL- of 4GL-programma zal na een onverwachte fout bij de uitvoering van een SQL-statement verdere afhandeling moeten volgen. Bij on-line programma’s kan de verwerking van de erop volgende programmacode worden afgebroken en een melding op het scherm worden getoond. Hierop moet worden aangegeven dat er een onverwachte fout was bij de verwerking en er moet voldoende informatie worden meegegeven om een medewerker (DBA) in staat te stellen het probleem nader te analyseren (bijvoorbeeld de foutcode, de naam van de stored procedure waarin het probleem zich voordoet, tabelnaam, sleutelwaarden). De gebruiker kan gevraagd worden deze informatie te noteren of een schermprint te maken, maar beter is het om deze gegevens (ook) naar een logfile te schrijven zodat medewerkers daar de noodzakelijke informatie vinden om het probleem uit te zoeken of een analyse gemaakt kan worden van veel voorkomende problemen.

Bij batchprogramma’s zal de verwerking van dit programma en eventueel ervan afhankelijke volgende programma’s, worden afgebroken. Andere programma’s die niet hiervan afhankelijk zijn, kunnen wel doorgaan. Ook de batch moet een goede logging verzorgen van de problemen.

WELKE STATEMENTS MOETEN GECONTROLEERD WORDEN

Niet alle statement behoeven gecontroleerd te worden. De controle is alleen van belang als er I/O-opdrachten worden uitgevoerd. Dit betekent dat de volgende statements (Sybase en Microsoft SQL Server syntax) geen foutafhandeling nodig hebben:

- Declaratie van variabelen: `declare @error int`
- Een constante waarde toekennen aan een variabele: `select @error = 0`
- Programma flow-statements: `if, while, goto, return, …`
- Set opties: `set rowcount 99, …`
- Enkele andere statements: `print, raiserror`

Voor de volgende statements is foutafhandeling wel gewenst:

- Datamanipulatie: `select, insert, update, delete`
- Transactie-afhandeling: `begin work, commit, rollback`

ENKELE BIJZONDERE SITUATIES

Als er in een SQL-statement een syntaxfout staat, bijvoorbeeld:

```
insert into wrk_names values (“XX”)
insrt into wrk_names values (“XX”)
select name from wrk_names
```

Dan zal dit gevonden worden bij het parsen van de coding. De gehele batch wordt dan afgekeurd. De statements voorafgaand aan (en volgend op) het foutieve statement worden ook niet uitgevoerd. Een eventuele foutafhandeling in deze batch wordt dus ook niet uitgevoerd.

Syntaxfouten komen voornamelijk voor bij het interactief intoetsen van coding maar kunnen ook in productiesystemen optreden, bijvoorbeeld bij:

- Dynamisch opgebouwde (gegenereerde) SQL-statements;
- De waarde van variabelen: een foutieve datum (04okt2002 in plaats van 04oct2002, afkomstig van een invoerscherm of invoerbestand) die in een datumveld in de database wordt geplaatst, zal bij Sybase ook tot een syntaxfout leiden.

Hoe op deze fouten gereageerd wordt, is weer productafhankelijk. Sybase zal bij syntaxfouten en enkele andere fouten niet doorgaan met het volgende statement (de foutafhandeling) maar de gehele verwerking afbreken tot het hoogste niveau: de 3GL of 4GL of het interactieve SQL tool. Aldaar moet de fout door het programma respectievelijk de gebruiker worden opgepakt.

CONCURRENT UPDATES

We stellen ons een programma voor waarin de gegevens van een crediteur (crediteurnummer, naam, adres en gironummer) gemuteerd kunnen worden. Gebruiker x haalt de gegevens van crediteur 123 op om deze te muteren en wijzigt het adres. Een ander veld, bijvoorbeeld het gironummer, blijft ongewijzigd op het scherm staan. Een andere gebruiker y haalt ook de gegevens van dezelfde crediteur 123 op om deze te muteren en wijzigt het gironummer. Het adres blijft ongewijzigd op het scherm staan.

Gebruiker x drukt op de toets 'Verwerk' of 'Opslaan'. Het programma zal alle velden (naam, adres en gironummer) van het crediteurrecord van het scherm naar de database schrijven, ook als deze niet door de gebruiker gewijzigd zijn.

Gebruiker y drukt nu op de toets 'Verwerk'. Als er geen rekening gehouden is met het gelijktijdig wijzigen van gegevens door meer gebruikers, zal het programma nu alle velden van het crediteurrecord op het scherm van gebruiker y naar de database schrijven, dus:

- de ongewijzigde naam;
- het gewijzigde gironummer;
- maar ook het zojuist door een andere gebruiker gewijzigde adres.

Hierbij wordt het adres, dat zojuist door gebruiker x is gewijzigd, weer met de oude waarde overschreven. De mutatie van gebruiker x is dus verloren gegaan, zonder gebruiker x hierop te wijzen!

Het programma moet daarom altijd controleren of een opgehaald record niet tussentijds is gewijzigd. Bij een tussentijdse wijziging zal het programma de verwerking afbreken en een foutboodschap laten zien, waarin vermeld wordt dat de gegevens tussentijds gewijzigd zijn en de gebruiker vragen om de gegevens opnieuw op te halen en zonodig de wijziging opnieuw aan te brengen. Dit opnieuw aanbrengen van een mutatie is misschien niet gebruikersvriendelijk, maar het komt niet vaak voor en het verloren gaan van een mutatie is in het geheel niet acceptabel.

Om problemen met een concurrent update te voorkomen zijn er in theorie drie mogelijke *locking* strategieën:

- Paranoid locking: de in een onderhoudsfunctie te lezen gegevens worden reeds direct bij het lezen gelockt (andere gebruikers mogen nog wel lezen maar niet wijzigen);
- Cautious locking: de in een onderhoudsfunctie te lezen gegevens worden gelockt zodra de gebruiker de eerste wijziging aanbrengt op het scherm. Mocht er reeds door een andere gebruiker een wijziging zijn aangebracht of een lock gelegd, dan krijgt de gebruiker op dit moment een melding;
- Optimistic locking: pas op het moment dat de gebruiker op de knop 'Verwerk' drukt controleert het programma of er iets door een andere gebruiker gemuteerd is.

Voor administratieve systemen wordt meestal optimistic locking gebruikt. Dit heeft enkele voordelen en nadelen. Het nadeel is dat de gebruiker bij een eventueel probleem opnieuw de gegevens moet ophalen en de wijzigingen aanbrengen, maar het voordeel is dat er geen gegevens in de database gelockt blijven terwijl de gebruiker een kopje koffie aan het drinken is. Deze situatie zou andere gebruikers (en de batchverwerking) flink in de weg kunnen zitten (zie ook [2]). Omdat een gelijktijdige wijziging in de praktijk weinig voorkomt, is het voordeel groter dan het nadeel en daarom is 'optimistic locking' voor de meeste administratieve systemen de beste optie.

Voor deze controle wordt meestal een vorm van *timestamp* (datum en tijd van laatste wijziging of een volgnummer) gebruikt. Een extra kolom in elke tabel bevat deze timestamp. Bij de update wordt gecontroleerd of de waarde van deze kolom niet gewijzigd is:

```
update tabel
set     – gewenste updates
,       mutatietijd = <nieuwe waarde>
where   primary_key = <primary_key waarde>
and     mutatietijd = <oude waarde>
```

Hierna volgt een controle: als er 0 rijen gewijzigd zijn, is het record intussen door een andere gebruiker gewijzigd of verwijderd en moet er een melding worden gegeven aan de gebruiker.

DEADLOCKS

Stel een mutatie van een gebruiker kan bestaan uit twee (of meer) acties welke altijd gecombineerd uitgevoerd moeten worden, bijvoorbeeld het toevoegen van een order met enkele orderregels. Voor elke orderregel moet ook het aantal van het artikel dat in voorraad is, worden bijgewerkt. Deze 'transactie' moet altijd in zijn geheel WEL of NIET verwerkt worden. Bij gedeeltelijke verwerking zou de database logisch niet meer consistent zijn!

Als veel gebruikers met een ordersysteem bezig zijn, kan de volgende situatie zich voordoen:

- Gebruiker x brengt een order in met twee orderregels: order-regel 1 op artikel 123 en orderregel 2 op artikel 321;
- Gebruiker y brengt ook een order in met twee orderregels: orderregel 1 op artikel 321 en orderregel 2 op artikel 123.

Beide gebruikers drukken op de 'Opslaan' toets. Voor beide gebruikers zullen nu orderheader en orderregels in de database worden toegevoegd.

- Voor gebruiker x zal artikel 123 worden bijgewerkt. Let op: een andere gebruiker mag de nieuwe situatie van dit artikel niet zien tot de gehele transactie van gebruiker x is afgewerkt;
- Voor gebruiker y zal artikel 321 worden bijgewerkt;
- Voor gebruiker x zou artikel 321 worden bijgewerkt maar dat kan niet want artikel 321 is door gebruiker y bijgewerkt maar nog niet vrijgegeven, gebruiker x moet dus even wachten tot gebruiker y klaar is;
- Voor gebruiker y zou nu artikel 123 worden bijgewerkt, maar dat kan niet want artikel 123 is door gebruiker x bijgewerkt maar nog niet vrijgegeven, gebruiker y moet dus even wachten tot gebruiker x klaar is.

Vergeten commit

Een project had een menuprogramma dat de andere functies aanriep. In één van deze functies zat een fout: onder specifieke omstandigheden werd een transactie niet afgesloten door een commit of rollback. Dit betekende dat het werk van alle volgende functies ook binnen de door de foutieve functie gestarte transactie vielen, mogelijk uren werk van de gebruiker. Andere gebruikers liepen natuurlijk tegen gelockte gegevens aan en zaten naar een zandlopertje te staren. Tot er voor de eerste gebruiker een rollback werd uitgevoerd, mogelijk na een deadlock. Toen werd al het werk van deze gebruiker teruggedraaid tot de foutieve functie.

Natuurlijk werd de fout in deze functie hersteld, maar het menuprogramma ging daarna ook controleren of elke functie op een correcte manier was afgesloten, maar er werden ook nog enkele andere controles aan toegevoegd; fouten werden gelogd. Zonodig werd nog een extra rollback uitgevoerd. Hiermee werd voorkomen dat een fout in een functie pas veel later in een andere functie tot een probleem voor de gebruiker leidt.

Nu wacht gebruiker x tot gebruiker y klaar is en wacht gebruiker y tot gebruiker x klaar is. Dit zou erg lang kunnen duren. Het RDBMS zal op deze situatie controleren en voor één van de twee gebruikers de gehele transactie terugdraaien. Deze gebruiker krijgt dan de foutmelding op het scherm. Voor de andere gebruiker kan hierna de transactie worden afgemaakt.

De gebruiker, die de foutmelding heeft gekregen, kan mogelijk opnieuw op de 'Opslaan' toets drukken als de gegevens nog op het scherm staan, anders moet hij zijn order geheel opnieuw intoetsen. Omdat een deadlock in de praktijk weinig voorkomt is dit opnieuw

Syntaxfouten komen voornamelijk voor bij het interactief intoetsen van coding

intoetsen voor de meeste administratieve systemen acceptabel. Er kunnen ook situaties zijn waarin de programmatuur zelf op een deadlock moet controleren, namelijk als de kans op een deadlock of het nadelig effect groter is. Bijvoorbeeld een programma dat in batch-order mutaties in de database verwerkt die afkomstig zijn van een groot bestand met veel orders.

Als bij de hierboven beschreven foutafhandeling een deadlock geconstateerd wordt, dan kan het programma de gehele transactie terugdraaien (voor zover dit al niet door het RDBMS is gedaan) en opnieuw aanbieden, eventueel in een loop en maximaal 10 keer.

Deadlocks kunnen vaak voorkomen worden door eenvoudige wijzigingen in het ontwerp, bijvoorbeeld:

- Kleine transacties: Een transactie per orderregel in plaats van per order;
- Sortering: Door in dit voorbeeld de orderregels gesorteerd op artikelnummer te verwerken wordt de deadlock al voorkomen.

TESTEN VAN DE FOUTAFHANDELING

De algemene regel voor testen (*unit test* of *white box test*) luidt: Alle coding tenminste 1 keer getest, of beter: Alle condities 1 keer waar en 1 keer onwaar.

Voor de coding en condities in de foutafhandeling is dit niet meer praktisch. De standaard foutafhandeling zal daarom meestal niet meer getest worden. Als het mogelijk is om deze coding 1 keer in een template onder te brengen en het template goed te testen, dan zijn deze testen ook niet meer nodig in elk programma (stored procedure) dat van het template is afgeleid. Deze werkwijze voorkomt fouten en vermindert de noodzaak om de coding van de foutafhandeling te testen.

Naast het bovenstaande is het aan te bevelen om, voor zover mogelijk en economisch rendabel, de scripts zo te schrijven dat na een fout het script opnieuw kan worden opgestart. Bijvoorbeeld: een batch met 100 create table en create procedure statements (de installatie van een nieuw systeem) is niet herstartbaar. Maar als voor elke create wordt bekeken of dit object al bestaat wordt deze procedure mogelijk wel herstartbaar:

```
if (tabel bestaat)
begin
   drop table
end
create table
```

CONCLUSIE EN AANBEVELING

Dit artikel geeft algemene richtlijnen voor foutafhandeling in SQL-coding, maar elke organisatie of project zal hiervoor eigen regels moeten opstellen, afhankelijk van de eisen van de organisatie en de mogelijkheden van het RDBMS en de 4GL.

De grootste fout is het geheel weglaten van de foutafhandeling in SQL-coding (batches, stored procedures, triggers) en in 3GL- of 4GL-programma's na een SQL-opdracht. Als er een fout optreedt en de gebruiker hierop niet gewezen wordt, is de kans groot dat met foutieve gegevens gewerkt gaat worden. Bedenk zelf wat er kan gebeuren als hierop bijvoorbeeld de aankoop van aandelen of opties zou zijn gebaseerd.

Door de coding voor foutafhandeling op een standaardmanier op één plaats (de on exception paragraaf) in een template op te nemen kost foutafhandeling niet veel extra werk. ●

LITERATUUR

1. Loonen, Ontwikkelstraat, hergebruik door inzet van architectuur. Software Release 2000/7-8.
2. Loonen, Performance. Database Magazine 2002/1-3.

Toon Loonen (toon.loonen@cgey.nl, toon.loonen@inter.nl.net) is als consultant werkzaam bij Cap Gemini Ernst & Young.