Richtlijnen voor batchprogrammatuur

Aanleiding

In de huidige programmatuur wordt weinig rekening gehouden met de verderop in dit document beschreven richtlijnen. Hierdoor ontstaat overlast voor de gehele organisatie en de Servicedesk in het bijzonder in de vorm van op deadlocks klappende batches, verloren wijzigingen of batches die geen voortgang lijken te vertonen. Het betreft dan overlast die gemakkelijk is te voorkomen. Door de gewoonte van ontwikkelaars om voor een nieuw stukje programmatuur af te kijken bij een bestaand programma, wordt het tij niet gekeerd. Vandaar deze poging om een aantal belangrijke principes nog eens tussen de oren te krijgen van eenieder die batchprogrammatuur ontwerpt, bouwt en onderhoudt. De richtlijnen zullen waar mogelijk in een volgende versie van de Standaarden en Richtlijnen worden opgenomen.

De onderwerpen zijn:

1: Voortgang
2: Herstartbaarheid
3: Verloren wijzigingen
4: Deadlocks
Appendix: Testresultaten diverse vergrendelingsstrategieën

1: Voortgang

Een regelmatig terugkerend verschijnsel is de situatie dat de Servicedesk 's avonds een batch opstart en de volgende dag 's ochtends de status nog steeds op "Running" ziet staan. Hoe weten zij nu wat ze moeten doen: laten lopen of afbreken? Vaak wordt dan advies ingewonnen bij TAB/Cockpit. Als er geen bijzondere situaties zijn zoals vergrendelingen, dan kunnen zij weinig anders doen dan melden: "de batch doet nog steeds zijn werk". Een voortgangsindicator is dan erg gewenst.

Advies: Gebruik de procedure *dbms_application_info.set_action* volop in je code. De informatie die je aan deze procedure meegeeft is dan te zien in de kolom *Action* van de view *v$session*. De eerste aanroep van *set_action* zou je bij voorkeur nog kunnen vervangen door een aanroep naar *set_module*, waar je zowel de *Action*- als de *Module*-kolom kunt meegeven. Als modulenaam geef je de naam van de batch op.

Bijvoorbeeld, direct na het woord BEGIN:

dbms_application_info.set_module('Mijnbatch','Initialisatie');

In de eventuele bulkverwerkingslus zet je dan:
dbms_application_info.set_action('Bezig met rijen ' || TO_CHAR(i) || ' tot en met' || TO_CHAR(i+99);.

Nu kun je dus door herhaaldelijk "select action from v$session where sid = …" uit te voeren in een andere sessie, inschatten wat de verwerkingssnelheid is. Het uitvoeren

van een COUNT(*) om het verwerkte aantal versus het totaal aantal te kunnen melden, raad ik af vanwege de extra kosten daarvan. Dit zou je los in een andere sessie kunnen doen als er vragen over de voortgang zijn.

2: Herstartbaarheid

Van alle batches wordt geëist dat ze herstartbaar zijn. Dit wordt het gemakkelijkst bereikt door er één transactie van te maken, oftewel één COMMIT aan het einde. Veel batches in ons systeem werken echter met transactievolumes en maximaal aantal transacties, zoals geregistreerd in de kolommen *Transacties_Per_Commit* en *Max_Aant_Transacties* uit BATCH_JOBS. Het idee hierachter is dat batches wel eens onverwacht stuklopen en dat je zodoende toch wat werk af kunt tikken doordat je regelmatig tussendoor een COMMIT hebt gegeven.

Er zijn echter extra eisen en nadelen verbonden aan tussentijdse COMMIT's:

- Het moet functioneel niet erg zijn als slechts een gedeelte wordt afgehandeld. Als functioneel alles of niets gewenst is, dan zijn tussentijdse COMMIT's dus uit den boze.
- Je batch moet herstartbaar blijven, dus je moet de afgehandelde rijen kunnen onderscheiden van de nog te verwerken rijen, door bijvoorbeeld een *Verwerkt_Ind* kolom. Door tussentijdse COMMIT's is herstartbaarheid niet triviaal meer.
- Het is langzamer dan een enkele COMMIT (zoek maar eens op AskTom)
- Je loopt ineens risico op de beruchte foutmelding *ORA-01555 snapshot too old*, zie het recente artikel van Marcel Hoefs hierover. Je zal de zogenaamde *fetch across commit* moeten omzeilen, waardoor je meerdere keren je gegevens moet ophalen in plaats van eenmaal met een hoofdcursor.

Al met al zorgt het bovenstaande ervoor dat ik persoonlijk liever mijn batch sneller maak met het risico dat de batch een keer extra opgestart moet worden, dan gebruik te maken van tussentijdse COMMIT's. Hierbij zorg je er dan voor dat de foutmeldingen die te maken hebben met de data zelf, alleen worden gerapporteerd en dat de rij wordt overgeslagen. De totale verwerking wordt zo niet verstoord. De batch mag alleen foutlopen op echt onverwachte zaken die buiten je macht liggen, zoals bijvoorbeeld een volgelopen tabelruimte.

Wat je ook kiest, je moet voor jezelf de vraag: "is de consistentie van mijn gegevens gewaarborgd als de batch onverwachts stukloopt, ongeacht de plaats en het moment waarop dit gebeurt?" altijd met een volmondig "Ja!" kunnen beantwoorden.

3: Verloren wijzigingen

Doordat in batches niet op de juiste manier de gegevens worden vergrendeld, is de kans erg groot dat er in het verleden wijzigingen verloren zijn gegaan en dat dit in de nabije toekomst zal blijven gebeuren. Een fictief voorbeeldje om dit uit te leggen:

```
DECLARE
  t_nieuwe_account_credit account.account_credit%TYPE;
BEGIN
  FOR r IN
```

```
  ( SELECT act.klt_rle_nummer
    ,      act.volgnr
    ,      act.account_credit
    FROM   accounts act
    WHERE  ROWNUM <= 10000
  )
  LOOP
    pkg$lock.act_lock(r.klt_rle_nummer,r.volgnr)
    ;
    <complexe berekening om t_nieuwe_account_credit vast te stellen>
    ;
    UPDATE accounts
    SET    account_credit = t_nieuwe_account_credit
    WHERE  klt_rle_nummer = r.klt_rle_nummer
    AND    volgnr = r.volgnr
    ;
  END LOOP;
END;
```

Stel hierbij voor dat de batch een uur loopt. Voor de laatste te verwerken account geldt dat deze verwerkt wordt met het account credit zoals die gold op het moment dat de batch begon, een klein uur geleden dus. Het account wordt echter pas vergrendeld aan het eind van de batch. Als een andere sessie het account credit heeft bijgewerkt gedurende dat uur, bijvoorbeeld door er 100 bij op te tellen, dan is na deze batch die wijziging verloren gegaan. Zoals de naam van deze paragraaf al doet vermoeden, treedt het probleem alleen op bij wijzigingen en niet in situaties waar er alleen data toegevoegd of verwijderd wordt.

De opzet van veel batches lijkt op het voorbeeld hierboven en leidt dus aan het beschreven fenomeen. Dit hoort een groot probleem voor de organisatie te zijn, maar kennelijk is deze werkwijze nog onvoldoende afgestraft in de praktijk. Of het bedrijf heeft tot nu toe altijd geluk gehad, of (waarschijnlijker) is het al diverse malen fout gegaan, maar heeft niemand het gemerkt. Het is dus wachten op het moment dat het echt een keer opgemerkt verkeerd gaat. Tot die tijd is het goed dat een ieder van ons zich hiervan bewust is en de situatie herstelt als de mogelijkheid er is.

Een aantal te volgen vergrendelingsstrategieën die verloren wijzigingen tegengaan, zijn:

- Alleen SQL
- PL/SQL: pessimistisch vergrendelen
- PL/SQL: optimistisch vergrendelen
- PL/SQL: vergrendelde rijen overslaan

De te kiezen strategie is niet zomaar algemeen aan te geven. Dit hangt af van de functionele eisen, van de omgeving en op welke tabellen de wijzigingen plaatsvinden. De technisch ontwerper kan dan met zijn gezond verstand een keuze maken.

Hieronder volgt een korte toelichting op de vier strategieën.

SQL

```
UPDATE accounts
```

```
SET    account_credit = <complexe expressie>
WHERE  ROWNUM <= 10000
```

Helaas is de wereld niet altijd zo simpel dat het zo te schrijven is. Het zou echter wel altijd de eerste gedachte moeten zijn om tot deze constructie te komen. Het is het snelst doordat je maar 1 contextswitch (= relatief dure overgang van PL/SQL naar SQL) doet, Oracle doet al het vergrendelingswerk voor jou en je voorkomt zo veel complexiteit in je programma. Kijk ook eens naar het gereduceerde aantal regels code. Verloren wijzigingen kunnen met deze constructie niet voorkomen. De wijziging wordt echter pas uitgevoerd als alle te wijzigingen rijen vrij zijn.

Pessimistisch vergrendelen

```
DECLARE
  CURSOR c_act
  IS
  SELECT act.acc_credit
  FROM   accounts act
  WHERE  ROWNUM <= 20000
  FOR UPDATE OF act.acc_credit
  ;
  TYPE t_acc_credit IS TABLE OF accounts.acc_credit%TYPE INDEX BY PLS_INTEGER;
  r_acc_credit t_acc_credit;
BEGIN
  OPEN c_act;
  LOOP
    FETCH c_act BULK COLLECT INTO r_acc_credit LIMIT 100
    ;
    FOR i IN 1..r_acc_credit.COUNT
    LOOP
      <Complexe berekening om r_acc_credit(i) te vullen.>
    END LOOP
    ;
    FORALL i IN 1..r_acc_credit.COUNT
    UPDATE accounts
    SET    acc_credit = r_acc_credit(i)
    WHERE  CURRENT OF c_act
    ;
    EXIT WHEN c_act%NOTFOUND
    ;
  END LOOP;
  CLOSE c_act;
END;
```

Selecteer de accounts met behulp van een FOR UPDATE clausule. Dit zorgt ervoor dat de geselecteerde rijen meteen vergrendeld worden op het tijdstip dat de hoofdcursor wordt geopend. Andere sessies zijn dan dus gedurende de draaitijd van de batch niet in staat om deze rijen te muteren. Zo worden verloren wijzigingen uitgesloten. De werking is verder eigenlijk hetzelfde als de SQL-variant, maar dan langzamer, vanwege meer contextswitches en zou dus alleen gebruikt moeten worden als SQL alleen niet meer volstaat. Ook hier zal de verwerking pas starten als alle rijen vrij zijn.

Optimistisch vergrendelen

```
DECLARE
  CURSOR c_act
  IS
  SELECT act.klt_rle_nummer
  ,      act.volgnr
  ,      act.acc_credit
  FROM   accounts act
  WHERE  ROWNUM <= 20000
  ;
  TYPE t_rle_nummer IS TABLE OF accounts.klt_rle_nummer%TYPE INDEX BY PLS_INTEGER;
  TYPE t_act_volgnr IS TABLE OF accounts.volgnr%TYPE INDEX BY PLS_INTEGER;
  TYPE t_acc_credit IS TABLE OF accounts.acc_credit%TYPE INDEX BY PLS_INTEGER;
  r_rle_nummer       t_rle_nummer;
  r_act_volgnr       t_act_volgnr;
  r_acc_credit       t_acc_credit;
  r_acc_credit_nieuw t_acc_credit;
BEGIN
  OPEN c_act;
  LOOP
    FETCH c_act BULK COLLECT INTO r_rle_nummer,r_act_volgnr,r_acc_credit LIMIT 100
    ;
    FOR i IN 1..r_rle_nummer.COUNT
    LOOP
      r_acc_credit_nieuw(i) := nvl(r_acc_credit(i) * 2, 100);
    END LOOP
    ;
    FORALL i IN 1..r_rle_nummer.COUNT
    UPDATE accounts
    SET    acc_credit = r_acc_credit_nieuw(i)
    WHERE  klt_rle_nummer = r_rle_nummer(i)
    AND    volgnr = r_act_volgnr(i)
    AND    acc_credit = r_acc_credit(i)
    ;
    EXIT WHEN c_act%NOTFOUND
    ;
  END LOOP;
  CLOSE c_act;
END;
```

De implementatie kan op meerdere manieren geschieden, die ook nog eens per Oracle-versie verschillen. Je vergrendelt de gegevens niet op het moment van ophalen, maar je gaat er vanuit dat de gegevens wel vrij zullen zijn. Tijdens de wijziging controleer je dan of je nog wel met dezelfde gegevens van doen hebt. Drie mogelijkheden zijn:

- Je haalt de oude waarden op van de gebruikte kolommen en later vergelijk je deze met de dan geldende waarden. Je krijgt hierdoor veel extra predikaten, maar het voordeel is dat andere kolommen in de te bewerken rijen gewoon gewijzigd mogen worden, zonder dat dit problemen oplevert voor jouw batch. Dit is de manier die in het voorbeeld hierboven is gekozen, door middel van het predikaat “acc_credit = r_acc_credit(i)”
- Je voegt aan iedere tabel een tijdstempelkolom (datatype TIMESTAMP) met een nauwkeurigheid tot op microseconden, een trigger om deze kolom te voorzien van het tijdstempel waarop deze als laatste is gewijzigd, om vervolgens slechts een extra predikaat met deze nieuwe kolom aan ieder

update-commando toe te voegen. Als een andere kolom wordt gewijzigd dan de kolom waar je mee bezig bent, dan wordt in dit scenario de rij ook als gewijzigd gezien en wordt deze overgeslagen.
- Je werkt met versie 10g release 1 of hoger en je laat het bovenstaande automatisch doen door de clausule ROWDEPENDENCIES mee te geven aan je tabel en de ORA_ROWSCN pseudokolom te gebruiken in je update-commando's. Hiervoor gelden dezelfde voor- en nadelen als voor het tijdstempel alternatief, alleen heb je geen (zichtbare) extra kolom in je tabel.

Optimistisch vergrendelen is met name geschikt voor toestandsloze omgevingen, zoals het web, waar pessimistisch vergrendelen niet kan. Voor batches kan het ook gebruikt worden, maar er is wel een belangrijk functioneel verschil met pessimistisch vergrendelen. Het is nu namelijk mogelijk dat voor een aantal rijen voor niks een berekening wordt gedaan, omdat blijkt dat een andere sessie de rij voor jou heeft gewijzigd. Je hebt dus niet meer de garantie dat alle rijen worden verwerkt. Dit is dan ook een belangrijk criterium of je een pessimistische of een optimistische strategie wilt kiezen. Je loopt bij optimistisch vergrendelen net als bij pessimistisch het risico dat je moet wachten op rijen die vergrendeld zijn, al wordt het "aantal vergrendelingen per tijdseenheid" gemiddeld gezien gehalveerd.

Vergrendelde rijen overslaan

```
DECLARE
  CURSOR c_act
  IS
  SELECT act.klt_rle_nummer
  ,      act.volgnr
  ,      null acc_credit
  FROM   accounts act
  WHERE  ROWNUM <= 20000
  ;
  TYPE t_rle_nummer IS TABLE OF accounts.klt_rle_nummer%TYPE INDEX BY PLS_INTEGER;
  TYPE t_act_volgnr IS TABLE OF accounts.volgnr%TYPE INDEX BY PLS_INTEGER;
  TYPE t_acc_credit IS TABLE OF accounts.acc_credit%TYPE INDEX BY PLS_INTEGER;
  r_rle_nummer t_rle_nummer;
  r_act_volgnr t_act_volgnr;
  r_acc_credit t_acc_credit;
BEGIN
  OPEN c_act;
  LOOP
    FETCH c_act BULK COLLECT INTO r_rle_nummer,r_act_volgnr,r_acc_credit LIMIT 100
    ;
    FOR i IN 1..r_rle_nummer.COUNT
    LOOP
      BEGIN
        SELECT act.acc_credit
        INTO   r_acc_credit(i)
        FROM   accounts act
        WHERE  act.klt_rle_nummer = r_rle_nummer(i)
        AND    act.volgnr = r_act_volgnr(i)
        FOR UPDATE OF act.acc_credit NOWAIT
        ;
        <Complexe berekening om r_acc_credit(i) te vullen.>
        ;
      EXCEPTION
```

```
      WHEN NO_DATA_FOUND THEN
        NULL; -- selectiecriteria zijn in de tussentijd gewijzigd
      WHEN pkg$lock.e_rij_vergrendeld THEN
        NULL; -- rij vergrendeld, overslaan dus
      END;
    END LOOP
    ;
    FORALL i IN 1..r_rle_nummer.COUNT
    UPDATE accounts
    SET    acc_credit = r_acc_credit(i)
    WHERE  klt_rle_nummer = r_rle_nummer(i)
    AND    volgnr = r_act_volgnr(i)
    AND    r_acc_credit(i) IS NOT NULL -- vergrendelde rijen worden zo overgeslagen
    ;
    EXIT WHEN c_act%NOTFOUND
    ;
  END LOOP;
  CLOSE c_act;
END;
```

Er bestaat een stukje ongedocumenteerde syntax "FOR UPDATE SKIP LOCKED". Helaas kan deze door gebrek aan ondersteuning van Oracle zelf niet gebruikt worden, maar het doet wel precies hetzelfde als hierboven beschreven met de uitzondering dat je niet te weten kan komen welke rijen zijn overgeslagen. Deze variant slaat alle vergrendelde rijen over, in plaats van te wachten totdat ze vrij komen. Dit is dus functioneel heel anders dan met name de eerste twee scenario's en soms niet gewenst en soms juist wel. Denk je maar eens in dat je het salaris van alle medewerkers eenmalig wil verhogen met 10%. Toch jammer als jouw rij net vergrendeld was …

In een aantal gevallen, is het echter helemaal niet erg als je een rij laat zitten en die de volgende dag of de volgende week oppakt. Deze methode is echter wel een stuk trager dan de vorige twee en belast daarbij de database ook nog eens zwaarder (meer latches, ook relatief), dus je moet wel een heel goede reden hebben om dit te willen. Zo'n reden kan zijn dat jouw batch altijd gelijktijdig loopt met andere batches die op dezelfde tabel werken en dat jouw batch dusdanig belangrijk is dat deze altijd door moet kunnen lopen.

Technisch gezien komt de implementatie erop neer dat je eerst de primaire sleutelwaarden ophaalt (rowid kan ook) op basis van jouw selectiecriteria, om later de overige waarden erbij te halen en direct te vergrendelen met de NOWAIT clausule. Let er hierbij wel op dat de selectiecriteria uit de buitenste cursor herhaald moeten worden in de binnenste cursor, als het mogelijk is dat de gebruikte kolommen in de selectiecriteria ondertussen gewijzigd worden. Vandaar dat ook een NO_DATA_FOUND afgevangen moet worden, ook al is dit in het huidige voorbeeld vanwege het ontbreken van echte selectiecriteria overbodig.

4: Deadlocks

Deadlocks treden op in de situatie dat sessie A eerst rij X wijzigt en daarna probeert rij Y te wijzigen, wat niet lukt omdat rij Y al is vergrendeld door sessie B, die daarna rij X probeert te wijzigen, wat ook niet lukt omdat sessie A deze heeft vergrendeld. Beide sessies zijn dan op elkaar aan het wachten. Oracle probeert iedere drie

seconden een deadlock-situatie te detecteren. Als Oracle zo'n situatie vindt, dan laat het willekeurig een van de twee sessies foutlopen op een *ORA-00060: deadlock detected while waiting for resource*. De andere sessie gaat dan gewoon door.

De nummer 1 oorzaak van een deadlock blijken in de praktijk niet-geindexeerde foreign keys te zijn, waarbij een rij uit de oudertabel worden verwijderd. Hierdoor wordt dan de gehele kindtabel vergrendeld. Gelukkig lijdt ons systeem hier momenteel niet aan. Als het bij ons optreedt, is het vanwege het gelijktijdig draaien van batches die gegevens in een verschillende volgorde verwerken.

Wat je vooral NIET moet doen om deadlocks te voorkomen, is om die reden extra vaak een COMMIT uitvoeren, zie hiervoor ook de paragraaf over herstartbaarheid. Dit heeft negatieve gevolgen voor data-integriteit en de verwerkingssnelheid. Je kan het enerzijds afvangen door de afhankelijkheden in kaart te brengen en zo er voor te zorgen dat de Servicedesk de twee betrokken batches niet meer gelijktijdig inroostert. Anderzijds kan je ervoor zorgen dat batches de gegevens in dezelfde volgorde verwerkt. Dus altijd eerst tabel X en dan tabel Y, of altijd de gegevens uit tabel X op volgorde van kolom A. De oplossing is dus altijd afhankelijk van de situatie. Het kan zijn dat je beide gebruik laat maken van een zelfde API, waardoor een zelfde volgorde wordt gehanteerd. Een andere mogelijkheid is dat je een *ORDER BY* clausule introduceert in de hoofdcursor van beide batches, ook al is die functioneel niet echt noodzakelijk.

Appendix: Testresultaten diverse vergrendelingsstrategieën

In een test heb ik uitgezocht hoe snel elke van de vier vergrendelingsstrategieën is, en hoe schaalbaar de oplossing is.

De snelheid is gemeten met een simpele "set timing on" in SQL*Plus en is gestaafd door middel van tkprof-bestanden. Hierbij is de eerste decimaal van de verstreken tijd in SQL*Plus genegeerd, aangezien deze altijd en foutief 0 (nul) is. De tweede decimaal is dus eigenlijk de eerste decimaal. Ook deze eigenaardigheid is geconstateerd door middel van het vergelijken met tkprof-bestanden.

De schaalbaarheid is gemeten door voor en na het stukje testcode in v$latch te kijken naar het toegenomen aantal *gets* van de twee latches die de library cache beschermen, genaamd "library cache pin" en "library cache". Dit is natuurlijk niet het enige dat de schaalbaarheid in de weg kan zitten, maar wel één van de belangrijkste, zeker in ons systeem. Door het feit dat er gelijktijdig meer sessies actief kunnen zijn, kan dit aantal hoger zijn dan alleen voor het stukje testcode nodig was. Hierdoor geldt dat ik hier niet het gemiddelde neem, maar het laagste aantal. Dit aantal had dus het minste last van gelijktijdig actieve sessies.

Anno 2007 gebruiken we alleen SQL en bulkverwerking in PL/SQL. Echter heb ik voor diegenen die nog steeds graag rij-voor-rij-verwerking programmeren, deze varianten ook meegenomen in de test. Voor hen: bekijk de testresultaten maar eens goed en laat dit nog een reden zijn om deze methode voorgoed vaarwel te zeggen …

Het testscript ziet er als volgt uit:

```
set echo on
create table accounts as select * from own.accounts where rownum <= 100000
/
alter table accounts add constraint act_pk primary key (klt_rle_nummer,volgnr)
/
exec dbms_stats.gather_table_stats(user,'ACCOUNTS',method_opt=>'FOR ALL INDEXED COLUMNS')
set timing on
alter session set tracefile_identifier = 'BATCH';
alter session set events '10046 trace name context forever, level 8';
rem
rem   De situatie nu: risico op verloren wijzigingen
rem
DECLARE
  t_nieuwe_account_credit accounts.acc_credit%TYPE;
BEGIN
  FOR r IN
  ( SELECT act.klt_rle_nummer
    ,      act.volgnr
    ,      act.acc_credit
    FROM   accounts act
    WHERE  ROWNUM <= 20000
  )
  LOOP
    pkg$lock.act_lock(r.klt_rle_nummer,r.volgnr,null,'N')
    ;
    t_nieuwe_account_credit := nvl(r.acc_credit * 2,100)
    ;
    UPDATE accounts
    SET    acc_credit = t_nieuwe_account_credit
    WHERE  klt_rle_nummer = r.klt_rle_nummer
    AND    volgnr = r.volgnr
    ;
  END LOOP;
END;
```

```
/
rollback
/
rem
rem   Het enkele UPDATE-commando
rem
UPDATE accounts
SET    acc_credit = nvl(acc_credit * 2, 100)
WHERE  ROWNUM <= 20000
/
rollback
/
rem
rem   Pessimistisch vergrendelen, bulkverwerking
rem
DECLARE
  CURSOR c_act
  IS
  SELECT act.acc_credit
  FROM   accounts act
  WHERE  ROWNUM <= 20000
  FOR UPDATE OF act.acc_credit
  ;
  TYPE t_acc_credit IS TABLE OF accounts.acc_credit%TYPE INDEX BY PLS_INTEGER;
  r_acc_credit t_acc_credit;
BEGIN
  OPEN c_act;
  LOOP
    FETCH c_act BULK COLLECT INTO r_acc_credit LIMIT 100
    ;
    FOR i IN 1..r_acc_credit.COUNT
    LOOP
      r_acc_credit(i) := nvl(r_acc_credit(i) * 2, 100);
    END LOOP
    ;
    FORALL i IN 1..r_acc_credit.COUNT
    UPDATE accounts
    SET    acc_credit = r_acc_credit(i)
    WHERE  CURRENT OF c_act
    ;
    EXIT WHEN c_act%NOTFOUND
    ;
  END LOOP;
  CLOSE c_act;
END;
/
rollback
/
rem
rem   Optimistisch vergrendelen, bulkverwerking
rem
DECLARE
  CURSOR c_act
  IS
  SELECT act.klt_rle_nummer
  ,      act.volgnr
  ,      act.acc_credit
  FROM   accounts act
  WHERE  ROWNUM <= 20000
  ;
  TYPE t_rle_nummer IS TABLE OF accounts.klt_rle_nummer%TYPE INDEX BY PLS_INTEGER;
  TYPE t_act_volgnr IS TABLE OF accounts.volgnr%TYPE INDEX BY PLS_INTEGER;
  TYPE t_acc_credit IS TABLE OF accounts.acc_credit%TYPE INDEX BY PLS_INTEGER;
  r_rle_nummer       t_rle_nummer;
  r_act_volgnr       t_act_volgnr;
  r_acc_credit       t_acc_credit;
  r_acc_credit_nieuw t_acc_credit;
BEGIN
  OPEN c_act;
  LOOP
    FETCH c_act BULK COLLECT INTO r_rle_nummer,r_act_volgnr,r_acc_credit LIMIT 100
```

```
    ;
    FOR i IN 1..r_rle_nummer.COUNT
    LOOP
      r_acc_credit_nieuw(i) := nvl(r_acc_credit(i) * 2, 100);
    END LOOP
    ;
    FORALL i IN 1..r_rle_nummer.COUNT
    UPDATE accounts
    SET    acc_credit = r_acc_credit_nieuw(i)
    WHERE  klt_rle_nummer = r_rle_nummer(i)
    AND    volgnr = r_act_volgnr(i)
    AND    acc_credit = r_acc_credit(i)
    ;
    EXIT WHEN c_act%NOTFOUND
    ;
  END LOOP;
  CLOSE c_act;
END;
/
rollback
/
rem
rem   Vergrendelde rijen overslaan, bulkverwerking
rem
DECLARE
  CURSOR c_act
  IS
  SELECT act.klt_rle_nummer
  ,      act.volgnr
  ,      null acc_credit
  FROM   accounts act
  WHERE  ROWNUM <= 20000
  ;
  TYPE t_rle_nummer IS TABLE OF accounts.klt_rle_nummer%TYPE INDEX BY PLS_INTEGER;
  TYPE t_act_volgnr IS TABLE OF accounts.volgnr%TYPE INDEX BY PLS_INTEGER;
  TYPE t_acc_credit IS TABLE OF accounts.acc_credit%TYPE INDEX BY PLS_INTEGER;
  r_rle_nummer t_rle_nummer;
  r_act_volgnr t_act_volgnr;
  r_acc_credit t_acc_credit;
BEGIN
  OPEN c_act;
  LOOP
    FETCH c_act BULK COLLECT INTO r_rle_nummer,r_act_volgnr,r_acc_credit LIMIT 100
    ;
    FOR i IN 1..r_rle_nummer.COUNT
    LOOP
      BEGIN
        SELECT act.acc_credit
        INTO   r_acc_credit(i)
        FROM   accounts act
        WHERE  act.klt_rle_nummer = r_rle_nummer(i)
        AND    act.volgnr = r_act_volgnr(i)
        FOR UPDATE OF act.acc_credit NOWAIT
        ;
        r_acc_credit(i) := nvl(r_acc_credit(i) * 2, 100)
        ;
      EXCEPTION
      WHEN NO_DATA_FOUND THEN
        NULL; -- selectiecriteria zijn in de tussentijd gewijzigd
      WHEN pkg$lock.e_rij_vergrendeld THEN
        NULL; -- vergrendeld, dus overslaan
      END;
    END LOOP
    ;
    FORALL i IN 1..r_rle_nummer.COUNT
    UPDATE accounts
    SET    acc_credit = r_acc_credit(i)
    WHERE  klt_rle_nummer = r_rle_nummer(i)
    AND    volgnr = r_act_volgnr(i)
    AND    r_acc_credit(i) IS NOT NULL  /* de vergrendelde rijen voldoen hier niet aan */
    ;
```

```
    EXIT WHEN c_act%NOTFOUND
    ;
  END LOOP;
  CLOSE c_act;
END;
/
rollback
/
rem
rem   Pessimistisch vergrendelen, rij-voor-rij
rem
DECLARE
  CURSOR c_act
  IS
  SELECT act.acc_credit
  FROM   accounts act
  WHERE  ROWNUM <= 20000
  FOR UPDATE OF act.acc_credit
  ;
  t_nieuwe_account_credit accounts.acc_credit%TYPE;
BEGIN
  FOR r_act IN c_act
  LOOP
    t_nieuwe_account_credit := nvl(r_act.acc_credit * 2, 100)
    ;
    UPDATE accounts
    SET    acc_credit = t_nieuwe_account_credit
    WHERE  CURRENT OF c_act
    ;
  END LOOP;
END;
/
rollback
/
rem
rem   Optimistisch vergrendelen, rij-voor-rij
rem
DECLARE
  t_nieuwe_account_credit accounts.acc_credit%TYPE;
BEGIN
  FOR r IN
  ( SELECT act.klt_rle_nummer
    ,      act.volgnr
    ,      act.acc_credit
    FROM   accounts act
    WHERE  ROWNUM <= 20000
  )
  LOOP
    t_nieuwe_account_credit := nvl(r.acc_credit * 2, 100)
    ;
    UPDATE accounts
    SET    acc_credit = t_nieuwe_account_credit
    WHERE  klt_rle_nummer = r.klt_rle_nummer
    AND    volgnr = r.volgnr
    AND    acc_credit = r.acc_credit
    ;
  END LOOP;
END;
/
rollback
/
rem
rem   Vergrendelde rijen overslaan, rij-voor-rij
rem
BEGIN
  FOR r IN
  ( SELECT act.klt_rle_nummer
    ,      act.volgnr
    FROM   accounts act
    WHERE  ROWNUM <= 20000
  )
```

```
  LOOP
    DECLARE
      t_nieuwe_account_credit accounts.acc_credit%TYPE;
    BEGIN
      SELECT act.acc_credit
      INTO   t_nieuwe_account_credit
      FROM   accounts act
      WHERE  act.klt_rle_nummer = r.klt_rle_nummer
      AND    act.volgnr = r.volgnr
      FOR UPDATE OF act.acc_credit NOWAIT
      ;
      t_nieuwe_account_credit := nvl(t_nieuwe_account_credit * 2, 100)
      ;
      UPDATE accounts
      SET    acc_credit = t_nieuwe_account_credit
      WHERE  klt_rle_nummer = r.klt_rle_nummer
      AND    volgnr = r.volgnr
      ;
    EXCEPTION
    WHEN NO_DATA_FOUND THEN
      NULL; -- selectiecriteria zijn in de tussentijd gewijzigd
    WHEN pkg$lock.e_rij_vergrendeld THEN
      dbms_output.put_line('rij vergrendeld');
    END;
  END LOOP;
END;
/
rollback
/
set echo off
set timing off
drop table accounts
/
```

Merk hierbij nog op dat de pkg$lock package lokaal is geïnstalleerd om deze te laten werken op mijn lokale ACCOUNTS tabel.

Voor de test op schaalbaarheid zijn exact dezelfde stukken code opgepakt en tussen een runstats_pkg.rs_start, rs_middle en rs_stop geplaatst. Voor de code van runstats_pkg: zie mijn schema RWK op O.

De testresultaten zijn dan als volgt:

	SQL	Pessimistisch	Optimistisch	Vergrendelde rijen overslaan	Pessimistisch rij-voor-rij	Optimistisch rij-voor-rij	Vergrendelde rijen overslaan rij-voor-rij
Doorlooptijd, alle rijen vrij, SQL*Plus timing							
Test 1	1,0	3,1	2,1	8,2	7,9	9,6	15,2
Test 2	0,9	2,1	1,9	6,8	6,9	5,8	11,1
Test 3	0,9	2,0	1,4	6,8	6,6	5,9	14,7
Test 4	0,9	2,0	1,4	6,9	7,0	5,9	11,5
Test 5	0,9	2,1	1,5	6,8	6,9	6,2	11,7
Gemiddelde (negeer hoogste en laagste)	**0,9**	**2,1**	**1,4**	**6,8**	**6,9**	**6,0**	**12,6**
Schaalbaarheid, #library cache latches							
Test 1	40	1071	899	81146	80189	80278	160206
Test 2	577	1304	1242	81479	81337	81531	168169
Test 3	540	882	889	81748	80346	80398	177919
Test 4	4702	9489	7805	102748	105346	99227	195330
Test 5	40	918	1210	80976	82048	80116	161702
Laagste aantal	**40**	**882**	**889**	**80976**	**80189**	**80116**	**160206**

Deze getallen ondersteunen de eerder geroepen kwalificaties over de verschillende strategieën:

- Puur SQL is het snelst en meest schaalbaar.
- Pessimistisch en optimistisch vergrendelen, mits met bulkverwerking geïmplementeerd, is ook lang niet slecht.

- Rij-voor-rij verwerking in drukke systemen kan écht niet meer: ongeveer 4 keer langere doorlooptijd (ruim 6 seconden ten opzichte van 1,5 a à 2 seconden met bulkverwerking) en 25 maal minder schaalbaar (80000 library cache latches/6 seconden ten opzichte van 800 latches per 1,5 seconde)
- Je moet wel een heel goede bedrijfsreden hebben om de strategie te kiezen om vergrendelde rijen over te slaan.

Bij dit alles wil ik als laatste nog opmerken dat we dankzij de vele database triggers de diverse verwerkingen enorm worden vertraagd. Immers, deze triggers zijn veelal "FOR EACH ROW" gedefinieerd en gaan dus af voor iedere rij. Alle validaties en acties in deze triggers gaan dus één voor één af, waarmee ze dus het effect van de bulkverwerking nivelleren. Zo heb ik in de praktijk gemerkt dat we niet een factor 4, maar met moeite een factor twee versnelling konden halen.