Native PL/SQL-compilatie

Nieuwe features Oracle 11g (4)

Ook in dit deel weer aandacht voor een performanceverbetering, ditmaal in het gebruik van PL/SQL. Deze procedurele uitbreiding is al vele malen dood verklaard, ten gunste van die andere embedded taal, Java. Ik heb echter steeds, als deze vraag gesteld werd aan Oracle-medewerkers, stellig ontkennende antwoorden hierop gehoord. Dat dit serieus is, blijkt wel uit het feit dat er op het gebied van PL/SQL nog steeds wordt doorontwikkeld.

Zo is er in Oracle 11g een zeer eenvoudige manier beschikbaar gekomen om PL/SQL (procedures, triggers en functies) te vertalen naar machinecode. In vorige versies van Oracle was het al mogelijk om PL/SQL-code te laten vertalen in 'C', en vervolgens te compileren tot deze machinecode, ook wel 'native' code genoemd. Dat betekent dat de code wordt uitgevoerd in instructies die ofwel direct op de beschikbare CPU kunnen worden verwerkt, ofwel op een niveau dat daar zeer dichtbij staat. Dit in tegenstelling tot de geïnterpreteerde uitvoeringswijze die normaal wordt gehanteerd. Een stuk PL/SQL-code wordt dan aangeboden aan een interpreter, die iedere keer als de code opnieuw wordt aangeboden, interpreteert. Zo'n interpreter compileert de code 'onderweg', en vertaalt al doende de instructies naar functies die op processorniveau kunnen worden uitgevoerd. Dat vertaalproces kost natuurlijk extra processorcapaciteit en daarmee tijd.
Om bij eerdere versies van Oracle, die 'native code' ondersteunden met betrekking tot PL/SQL, gebruik te maken van deze mogelijkheid was nogal wat werk nodig. Zo moest er een 'C'-compiler beschikbaar zijn, en moest er rondom het geheel het een en ander worden geconfigureerd. In Oracle 10g was het al wat gemakkelijker dan in 9i, maar in 11g is het een kwestie van 'aanzetten'. Oracle zal zelf de vertaalslag naar machinecode doen.

Het aanzetten van de 'native' code optie is erg simpel:

```
SQL> alter system set plsql_code_type = native;

System altered.
```

De parameter kan ook op sessieniveau worden ingesteld. Verder kan een al bestaande PL/SQL-procedure worden omgezet in machinecode en terug:

```
SQL> alter procedure tn compile plsql_code_type = native;

Procedure altered.

SQL> alter procedure tn compile plsql_code_type = interpreted;

Procedure altered.
```

Om te kijken hoe effectief het gebruik van PL/SQL in native mode is heb ik een drietal testen opgezet. De eerste test voert geen enkele actie op de database uit, er wordt alleen een programmalus met een enkele assignment doorlopen:

```
CREATE OR REPLACE PROCEDURE tn(C IN NUMBER) AS
  i1 NUMBER;
  i2 NUMBER;
BEGIN
  FOR i1 IN 1..C
  LOOP
    i2 := i1;
  END LOOP;
END;
/
```

Deze procedure wordt eerst tien keer aangeroepen op de oude manier, met een interpreter voor de verwerking. Vervolgens wordt de procedure opnieuw gecompileerd in 'native' mode. Daarna worden er opnieuw tien aanroepen uitgevoerd. De verwerkingstijden worden door SQL*Plus bijgehouden: met het commando 'set timing on' worden deze na elke aanroep getoond.
De resultaten van de twee kunnen vervolgens worden vergeleken. Het gebruikte scriptje is laat de procedure een lus 100 miljoen keer doorlopen:

```
set timing on
alter procedure tn compile plsql_code_type = interpreted;
PROMPT met procedure interpreted
exec tn (100000000);
.
.
exec tn (100000000);

alter procedure tn compile plsql_code_type = native;
PROMPT met procedure native
exec tn (100000000);
.
.
exec tn (100000000);
```

Tien aanroepen in op interpreter-basis kostten in totaal 197.26 seconden, in machinecode waren er 146.28 seconden nodig. De machinecodeversie was daarmee 26% sneller dan de geïnterpreteerde versie.

Vervolgens heb ik een PL/SQL-procedure gemaakt die in een loop een update op een tabel uitvoert. Dat ziet er zo uit:

```
CREATE TABLE x (N NUMBER);
INSERT INTO X values (0);

CREATE OR REPLACE PROCEDURE tn2(C IN NUMBER) AS
  i1 NUMBER;
BEGIN
  FOR i1 IN 1..C
  LOOP
    UPDATE x SET n = i1;
  END LOOP;
  COMMIT;
END;
/
```

Ook nu wordt de procedure twintig keer gemeten: tien keer geïnterpreteerd, en tien keer in machinecode:

```
set timing on
alter procedure tn2 compile plsql_code_type = interpreted;
PROMPT met procedure interpreted
exec tn2 (100000);
.
.
exec tn2 (100000);

alter procedure tn2 compile plsql_code_type = native;
PROMPT met procedure native
exec tn2 (100000);
.
.
exec tn2 (100000);
```

Geen 100 miljoen keer door de programmalus dit keer, maar honderdduizend keer. Omdat er nu gegevensmanipulatie in de database plaatsvindt, wordt er veel meer van het systeem gevergd. Voor een meetbare doorlooptijd kan worden volstaan met een kortere lus. Mijn verwachting was dat de machinecodeversie niet of nauwelijks sneller zou zijn dan de geïnterpreteerde versie. Immers, de hoeveelheid werk voor de kernel die achter het updatestatement verscholen gaat is aanzienlijk.
Er vindt latch-activiteit plaats, er moet undo en redo worden gegenereerd. Het bleek echter wel mee te vallen: Voor de interpretervariant kostten tien aanroepen 107,52 seconden, de machinecodevariant duurde 93,9 seconden. Er is dus toch nog een versnelling van bijna 13% bereikt.

Ten slotte heb ik een trigger gemaakt:

```
CREATE TABLE t3 (N NUMBER)
/

CREATE OR REPLACE TRIGGER t3u
BEFORE UPDATE ON t3
FOR    EACH ROW
DECLARE
BEGIN
  :NEW.N := :NEW.N + 1;
END;
/
```

De tabel t3 werd gevuld met 100.000 rijen. Het testscript ziet er dan als volgt uit:

```
set timing on
alter trigger t3u compile plsql_code_type = interpreted;
PROMPT met trigger interpreted
update t3 set N = N;
commit;
.
.
update t3 set N = N;
commit;

alter trigger t3u compile plsql_code_type = native;
PROMPT met procedure native
update t3 set N = N;
commit;
.
.
update t3 set N = N;
commit;
```

Er worden per keer 100.000 rijen geüpdate. De bijbehorende trigger wordt 100.00 keer aangeroepen, en voert steeds één assignment uit voordat de rij in de database wordt bijge-

werkt. Opnieuw wordt iedere actie tien keer uitgevoerd. De tijden voor de interpreter en de machinecode zijn nu respectievelijk 69,61 en 65,94 seconden. Dat zou een verbetering van 5% aanduiden. Echter, gezien de het verschil in minimale en maximale meetwaarden (voor de metingen met interpreter en machinecode respectievelijk 3,9 en 4,1 seconden) durf ik hier geen conclusie aan te verbinden. Ik heb de test nog een aantal keren herhaald, met steeds dezelfde variatie in meetresultaten. Deze waren bij de eerste twee testen aanzienlijk stabieler.

Mijn conclusie is dat het gebruik van native gecompileerd PL/SQL aanzienlijk is vereenvoudigd, en met name bij het gebruik van rekenintensieve taken in PL/SQL een stevige performanceverbetering te zien geeft. Zodra er ook een forse component gegevensmanipulatie in de database aan te pas komt, wordt het effect minder, tot zelfs verwaarloosbaar. Toch lijkt het me zeker de moeite waard standaard van native gecompileerd PL/SQL gebruik te gaan maken. Wel zou ik daarbij alle PL/SQL na hercompilatie nog eens zorgvuldig testen, maar als het goed is wordt dat bij de overgang naar een nieuwe versie toch al gedaan.

Carel-Jan Engel werkt als onafhankelijk Oracle-consultant. Hij is lid van het Oak Table Network. E-mail: careljan@dbalert.eu.