Een reflectie van SQL Technieken

Tom Kyte over scalar subquery en caching

Er zijn verschillende technieken en trucjes om het schrijven van SQL eenvoudiger te maken. Een daarvan is scalar subqueries. Deze kunnen worden gebruikt om het aantal keren dat een PL/SQL-functie die wordt aangeroepen vanuit SQL, tot een minimum te beperken. Dit is belangrijk, omdat de overhead als gevolg van het overschakelen van SQL naar PL/SQL vrij duur is. De database functie scalar subquery in combinatie met caching kan deze overhead drastisch beperken.

Wat is nu precies een scalar subquery? Een scalar subquery is een subquery in een SQL statement die een enkele kolom en nul of één rijen oplevert. Die ene kolom kan een complex object type zijn en kan dus in feite bestaan uit vele attributen, maar je geeft een enkele scalar waarde terug (of NULL als de subquery nul records teruggeeft). Een scalar subquery kan overal worden gebruikt waar een literal had kunnen worden gebruikt. Een voorbeeld:

```
select deptno, dname,
       'Hello world'
 from dept;
```

of, met betekenisvollere data:

```
select deptno, dname
          (select count (*) from emp where emp.deptno = dept.deptno
   from dept;
```

We kunnen het ook (sinds versie 8.1.5 van Oracle) zo formuleren:

```
select deptno, dname,
       (select count(*) from emp where emp.deptno = dept.deptno)
  from dept;
```

Die laatste query is het semantische equivalent van:

```
select  dept.deptno, dept.dname,
        count(emp.empno)
   from dept left outer join emp
     on (dept.deptno = emp.deptno );
```

Je zou de vorige query kunnen gebruiken om de reactietijd van de zoekopdracht te optimaliseren. Oracle zou alleen de eerste rij van DEPT krijgen, de scalar subquery uitvoeren tegen EMP om de telling te krijgen en deze teruggeven. Het zou heel snel de eerste rij produceren. Met de laatste query zou het waarschijnlijk twee volledige scans uitvoeren, een hash outer join en een samenvoeging, voordat de eerste rij kan worden geretourneerd. Nu we weten wat een scalar subquery is moeten we nog de scalar subquery caching begrijpen. In de bovenstaande query met een scalar subquery zou de database onze scalar subquery vertalen om bind variabelen te gebruiken - dus het zou in feite deze query uitvoeren:

```
(select count(*) from emp where emp.deptno = ?)
```

voor elke rij in de DEPT tabel. Gezien het feit dat DEPTNO uniek is in DEPT zou het de scalar subquery fysiek moeten uitvoeren voor elke rij in DEPT. Maar wat als de DEPT tabel een andere tabel is met een kolom DEPTNO er in? Wellicht een PROJECTS tabel met de records PROJECT_NAME en DEPTNO. Dan zou een query, zoals:

```
select project_name, deptno,
       (select count(*) from emp where emp.deptno = projects.deptno)
  from projects;
```

de scalar subquery minstens zo vaak moeten uitvoeren als er unieke DEPTNO waarden in de tabel PROJECTEN zijn. Let op het 'minstens'; het hoeft de scalar subquery niet uit te voeren voor elke rij in de tabel PROJECTS als we enkele van de resultaten cachen en hergebruiken. Dat doet de database gelukkig op de achtergrond voor ons: bij gebruik van een scalar subquery zal Oracle een kleine in-memory hash tabel creëren voor de subquery en de resultaten ervan. Dus als we bovenstaande query uitvoeren zet Oracle in het geheugen een hash tabel op, die er als volgt uitziet:

```
Select count(*) from emp where emp.deptno = :deptno
:deptno    Count(*)

….          ….
```

Het zal gebruik maken van deze hash tabel om de scalar subquery en de input – in dit geval alleen DEPTNO - met de bijbehorende output op te slaan. Aanvankelijk is deze cache natuurlijk leeg. Wanneer de query gaat draaien zitten er geen waarden in. Maar stel dat we de query runnen en in de eerste PROJECT rij halen we een DEPTNO van 10 op. Oracle zal het getal 10 hashen op een waarde tussen 1 en 255 (de omvang van de hash table cache in Oracle 10g en 11g) en kijkt op die hashtabel plek om te zien of het antwoord bestaat. In dit geval zal dat niet zo zijn en moet de scalar subquery met de input van 10 draaien en het resultaat weergeven. Stel dat het antwoord 42 is, dan zou de hash table er nu ongeveer zo uitzien:

```
Select count(*) from emp where emp.deptno = :deptno
:deptno    Count(*)
….         …
10         42
….         ….
```

We hebben de waarde 10 en het antwoord 42 in een slot opgeslagen - waarschijnlijk niet in de eerste, waarschijnlijk niet in de laatste slot - maar in het slot waarnaar de waarde 10 is gehashed. Stel nu dat de tweede rij die we terug krijgen van PROJECTS een DEPTNO = 20 bevat. We kijken opnieuw in de hash tabel na hashing de waarde 20 en we zouden terugkrijgen 'no results in the cache yet'. We zouden de scalar subquery draaien en het resultaat in de hash table cache zetten- nu kan onze cache er zo uitzien:

```
Select count(*) from emp where emp.deptno = :deptno
:deptno    Count(*)
…          …
10         42
….         ….
20         55
…          …
```

Stel dat we op de derde rij opnieuw DEPTNO = 10 krijgen. Ditmaal hashen we DEPTNO = 10, ontdekken dat we die waarde al in de hash table cache hebben - en we kunnen eenvoudigweg 42 uit de cache teruggeven in plaats van de scalar subquery uit te voeren. In feite zouden we de scalar subquery voor de DEPTNO waarden van 10 of 20 nooit meer voor deze query hoeven uit te voeren, want we zouden het antwoord al hebben.
De vraag doet zich voor wat er gebeurt als het aantal unieke DEPTNO waarden de omvang van onze hash table overschrijdt? Wat als er meer dan 255 zijn? Of, meer in het algemeen, wat gebeurt er als er meer dan een DEPTNO waarde wordt gehashed op dezelfde slot in de hash tabel. Wat gebeurt er als we een hash collision krijgen?
Het antwoord is vrij simpel: Oracle zal niet in staat zijn om de tweede of N-de waarde die naar dat slot in de tabel hasht te cachen. Stel dat de derde rij in de tabel de waarde DEPTNO = 30 heeft. En laten we ervan uitgaan dat 30 hasht naar exact dezelfde slot als DEPTNO 10 deed. We zullen in dit geval niet in staat zijn om DEPTNO = 30 te cachen - dit zal de hash tabel nooit bereiken. Het zal echter 'gedeeltelijk' worden gecached. Oracle heeft de hash tabel met alle voorgaande executies - maar het houdt ook het laatste scalar subquery resultaat 'naast' de hash tabel vast. Tenminste, als de vierde rij ook voor DEPTNO = 30 is, dan zal Oracle ontdekken dat het resultaat niet in de hash tabel staat maar 'naast' de hash table aangezien de subquery de laatste keer werd uitgevoerd met de input 30. Aan de andere kant, als de vierde rij DEPTNO = 40 is, zouden we de scalar subquery uitvoeren met 40 (omdat we het tijdens deze query-run nog niet hebben gezien) en overschrijven het resultaat van DEPTNO = 30. De volgende keer dat we DEPTNO = 30 in het resultaat zien moeten we de scalar subquery weer uitvoeren.
Dit alles als inleiding tot het eigenlijke onderwerp: Hoe kan ik het aantal keren dat een PL/SQL-functie wordt aangeroepen vanuit SQL verminderen?

Stel je hebt deze PL/SQL-functie:

```
SQL> create or replace function f( x in varchar2 ) return number
  2  as
  3  begin
  4   dbms_application_info.set_client_info(userenv('client_info')+1 );
  5          return length(x);
  6  end;
  7  /
Function created.
```

Deze functie zal gewoon elke keer dat hij wordt aangeroepen een teller ophogen - de teller zal worden opgeslagen in de CLIENT_INFO kolom van V$SESSION en geeft dan de lengte van zijn input. Als we deze gewone query uitvoeren (de tabel STAGE is gewoon een kopie van ALL_OBJECTS):

```
SQL> exec :cpu := dbms_utility.get_cpu_time; dbms_application_info.set_
client_info(0);
PL/SQL procedure successfully completed.

SQL> select owner, f(owner) from stage;
…
72841 rows selected.

SQL> select dbms_utility.get_cpu_time-:cpu cpu_hsecs, userenv('client_
info') from dual;

 CPU_HSECS USERENV('CLIENT_INFO')
---------- --------------------------
       118 72841
```

We zien dat onze functie eenmaal per rij wordt aangeroepen, zelfs als de input naar onze functie telkens wordt herhaald. Als we een scalar subquery gebruiken en 'f (owner)' vervangen door '(select f(owner) from dual', zullen we een forse vermindering van oproepen naar onze functie zien.

```
SQL> exec :cpu := dbms_utility.get_cpu_time; dbms_application_info.set_
client_info(0);
PL/SQL procedure successfully completed.
```

```
SQL> select owner, (select f(owner) from dual) f from stage;
…
72841 rows selected.

SQL> select dbms_utility.get_cpu_time-:cpu cpu_hsecs, userenv('
client_info') from dual;

 CPU_HSECS USERENV('CLIENT_INFO')
---------- --------------------------
         29 66
```

Zoals je ziet gaan we van 72.841 aanroepen naar 66. En de CPU-tijd daalt ook dramatisch omdat we de functie niet zo vaak aanroepen (de context switch van SQL naar PL/SQL en geen beroep doen op DBMS_APPLICATION_INFO en LENGTH).

Wat als we de functie als deterministisch markeren - want het is in feite deterministisch aangezien bij gelijke input altijd dezelfde output wordt geretourneerd - zou dat niet ook een vermindering van het aantal functie-aanroepen opleveren? In principe wel, maar niet zo goed als scalar subquery caching dat kan. Bijvoorbeeld:

```
SQL> create or replace function f( x in varchar2 ) return number
  2  DETERMINISTIC
  3  as
  4  begin
  5           dbms_application_info.set_client_info(userenv('client_
info')+1 );
  6           return length(x);
  7  end;
  8  /

Function created.

SQL> exec :cpu := dbms_utility.get_cpu_time; dbms_application_info.set_
client_info(0);

PL/SQL procedure successfully completed.

SQL> select owner, f(owner) from stage;
…
72841 rows selected.

SQL> select dbms_utility.get_cpu_time-:cpu cpu_hsecs, userenv('client_
info') from dual;

 CPU_HSECS USERENV('CLIENT_INFO')
---------- ---------------------
         69 8316
```

Zoals je ziet vermindert het aantal calls tot 8316. De scalar subquery cache is in dit geval beter dan de markering als deterministisch (let op: deterministisch beïnvloedt de caching alleen in Oracle 10g en hoger).

Nog een stap verder - je zou kunnen vragen 'maar wat als we de function result cache in 11g zouden gebruiken. Dan zouden de functie aanroepen op nul uitkomen. Het antwoord daarop is 'ja, zoiets'. De functie aanroepen zouden dalen tot nul, maar het schakelen van SQL naar PL/SQL zou op een zeer hoge waarde blijven. In dit geval 72.841 keer. Bijvoorbeeld:

```
SQL> create or replace function f( x in varchar2 ) return number
  2  RESULT_CACHE
  3  as
  4  begin
  5           dbms_application_info.set_client_info(userenv('client_
             info')+1 );
  6           return length(x);
  7  end;
  8  /
Function created.

SQL> exec :cpu := dbms_utility.get_cpu_time; dbms_application_info.set_
client_info(0);
PL/SQL procedure successfully completed.

SQL> select owner, f(owner) from stage;
…
72841 rows selected.

SQL> select dbms_utility.get_cpu_time-:cpu cpu_hsecs, userenv('client_
info') from dual;

 CPU_HSECS USERENV('CLIENT_INFO')
---------- ------------------------------
         73 32
```

Het aantal aanroepen is 32 (ik heb toevallig 32 schema's in mijn database), maar de CPU-tijd is met 73 ongeveer hetzelfde als met de deterministische functie en ver en boven de 66 aanroepen van de scalar subquery! Bovendien, als we deze query opnieuw draaien, zouden we ontdekken dat het onze functie nul keer oproept - maar nog steeds een hoge CPU-tijd heeft:

```
SQL> exec :cpu := dbms_utility.get_cpu_time; dbms_application_info.set_
client_info(0);
PL/SQL procedure successfully completed.

SQL> select owner, f(owner) from stage;
…
72841 rows selected.

SQL> select dbms_utility.get_cpu_time-:cpu cpu_hsecs, userenv('client_
info') from dual;

 CPU_HSECS USERENV('CLIENT_INFO')
---------- ----------------------------
         63 0
```

Dit laat zien dat, zelfs als de functie deterministisch is, zelfs als de functie 'result cached' is, er een goede reden is om de functie aanroep in een SELECT from dual onder te brengen. Ik heb in de loop der jaren geleerd nooit dit te coderein:

```
Select * from t where column = plsql_function(..);
```

Maar wel:

```
Select * from t where column = (select plsql_function(…) from dual);
```

Om optimaal te profiteren van de scalar subquery cache.

Tom Kyte is vice president and senior technology architect bij Oracle.