

DOAG

Ausgabe 02/2009
Seite 43-44

Kompression, Ausführungspläne und Bind-Peeking

Matthias Mann, maihiro GmbH

Tabellenkompression liefert oftmals beeindruckende Ergebnisse, allerdings sind auch einige Nachteile dieses Features wie der leichte CPU-Overhead bekannt. Weitere Effekte können sich allerdings durch das Zusammenspiel von Bind Variable Peeking mit der Tabellenkompression ergeben, die sich in 10g und 11g unterschiedlich ausprägen.

Es war eine interessante Konstellation: In einer Datenbank läuft ein ETL-Prozess, der täglich die Daten der Vortage überträgt und verdichtet. Der Vorgang dauerte in der Regel etwas mehr als eine Stunde, benötigte aber nach einer Reorganisation der beteiligten Tabellen mehrere, manchmal sogar mehr als 10 Stunden. Dabei verlief die Tabellenpflege durchaus erfolgreich. So gelang es, durch das Löschen von nicht benötigten Daten und den Einsatz von Tabellenkompression die Datenmenge von mehr als 10 GB auf nur 1,5 GB zu reduzieren.

Der Grund für die Laufzeitverlängerung war schnell gefunden: Die Ausführungspläne einiger SELECT-Statements, die ihre Arbeit vorher in einigen Millisekunden erledigten, waren „geklippt“, so dass die Anweisungen für ihre Arbeit nun mehrere Sekunden oder sogar Minuten brauchten. Einige Experimente mit den Statements zeigten, dass der Optimizer die alten und besseren (sprich in der gegebenen Situation passenderen) Ausführungspläne mit höheren Kosten bewertete als die neuen Pläne, die oft einen Table-Full-Scan beinhalteten. Eine weitere Analyse zeigte, dass nach der Tabellenkompression jeder Block nun durchschnittlich die dreifache Anzahl von Zeilen speicherte. Damit wurde ein Full-Scan für den Optimizer wesentlich attraktiver.

Da die ersten Ausführungen der Statements meistens eine relativ hohe Anzahl von Datensätzen zurücklieferten, wurde beim Parsen dieser mit Bind-Variablen versehenen Abfragen ein für den Gesamtprozess unpassender Plan gewählt (Bind Variable Peeking). Vor der Reorganisation der Tabellen führ-

ten die gleichen Datenmengen immer zu dem „richtigen“ Ausführungsplan, da trotz der großen Ergebnismenge der abgefragte Anteil der benötigten Blöcke im Verhältnis zu der Gesamtanzahl der Blöcke in der Tabelle gering war.

Komprimierung und Ausführungspläne in 10g

Nachvollziehbar wird das beschriebene Verhalten durch einen einfachen Test, bei dem man eine Abfrage auf einen definierten Datenbereich durchführt. Im ersten Schritt wird eine Tabelle mit 100.000 Zeilen auf Basis der DBA_OBJECTS erstellt, ein Unique Index wird auf der Spalte OBJ_ID erzeugt und anschließend wird die Tabelle analysiert:

```
create table OBJS as
(select * from (
  select rownum as obj_
  id, a.*
  from dba_objects a,
  dba_objects b
  where b.object_id < 4
)
where rownum < 100001);
create unique index OBJS_N1 on
OBJS(obj_id);
BEGIN
  dbms_stats.gather_table_
  stats(
    ownname => .XYZ', tabna-
    me => ,OBJS', cascade => true,
    method_opt => .for all
    indexed columns size auto',
    estimate_percent =>
    dbms_stats.auto_sample_size);
END;
/
```

Die zu untersuchende Abfrage soll einen bestimmten Datenbereich der Tabelle zurückliefern:

```
select * from OBJS where obj_id
> 80000
```

Nun kann mittels einer Halbierungssuche die OBJ_ID herausgefunden werden, unterhalb derer sich der Optimizer für einen Full-Scan entscheidet. In diesem Beispiel (und in dieser Testumgebung) wählt der Optimizer ab OBJ_ID=82631 den Index-Zugriff, darunter einen Full-Scan (Ausgabe angepasst):

```
set autotrace trace explain

select * from OBJS where obj_id
>82630;
| Operation |
Name | Rows
-----|-----
| SELECT STATEMENT |
| 17370
| TABLE ACCESS BY INDEX ROWID|
OBJS | 17370
| INDEX RANGE SCAN
OBJS_N1 | 17370
-----|-----

select * from OBJS where obj_id
>82629;
| Operation | Name |
Rows
-----|-----
| SELECT STATEMENT | |
17371
| TABLE ACCESS FULL| OBJS |
17371
-----|-----

Nun werden die Tabelle kom-
primiert und die Statistiken
erneuert:

alter table OBJS compress move;
alter index OBJS_N1 rebuild;
BEGIN
  dbms_stats.gather_table_
  stats(
```

```

ownname => ,XYZ', tabname
=>'OBJ_'. cascade => true,
method_opt => ,for all in-
dexed columns size auto',
estimate_percent => dbms_
stats.auto_sample_size);
END;
/

```

Die Tabelle belegt nach der Komprimierung nur noch ein Drittel des ursprünglichen Speicherplatzes. Wieder ist ein Wert zu bestimmen, unterhalb dessen der Optimizer einen Full-Scan wählt. In diesem Fall ist es OBJ_ID=85597:

```

SQL> select * from OBJ_ where
obj_id >85596;
| Operation |
| Name | Rows |
|-----|-----|-----|
| SELECT STATEMENT |
| 14404 |
| TABLE ACCESS BY INDEX ROWID |
| OBJ_ | 14404 |
| INDEX RANGE SCAN |
| OBJ_N1 | 14404 |

```

```

SQL> select * from OBJ_ where
obj_id >85595;
| Operation | Name |
| Rows |
|-----|-----|
| SELECT STATEMENT | |
| 14405 |
| TABLE ACCESS FULL | OBJ_ |
| 14405 |

```

Die OBJ_ID, ab der ein Full-Scan gewählt wird, hat sich durch das Komprimieren der Tabelle verändert. Wurde vorher bei einer Abfrage ab OBJ_ID=82631 ein Index-Scan benutzt, ist es nachher die OBJ_ID=85597. Damit entsteht ein „unsicherer Bereich“, in diesem Fall die OBJ_IDs zwischen 82631 und 85597, in dem sich der Ausführungsplan des Statements durch die Komprimierung ändert.

Dies kann im Zusammenspiel mit dem Bind-Peeking-Mechanismus der 10g-Datenbank zu einem Problem führen: der erste Parse der mit Bind-Variablen versehenen Anweisung nach der Komprimierung und Statistikberechnung ist ein Hard-Parse, und der resultierende Plan wird für jede

weitere Ausführung des Statements beibehalten. Übergibt man zum Hard-Parse ein Wert von beispielsweise OBJ_ID=84000, wird sich der Optimizer für einen Full-Scan entscheiden – vor der Komprimierung hat er bei diesen Wert einen Index-Scan bevorzugt. Diese Entscheidung für den Full-Scan kann, wenn im Laufe des weiteren Prozesses die Anweisung vorwiegend mit sehr hohen OBJ_IDs abgesetzt wird, ungünstig sein. Deutlich ist dieser Effekt im Trace zu sehen:

```

alter session set sql_
trace=true;
var bl number
exec :bl := 84000;
select * from OBJ_ where obj_
id > :bl;
exec :bl := 89000;
select * from OBJ_ where obj_
id > :bl;

```

Im Trace-File kann man nun den Ausführungsplan der beiden Statements betrachten. Wie erwartet, wird die Anweisung zweimal mit einem Full-Scan ausgeführt:

```

STAT #33 id=1 cnt=16000 pld=0
pos=1 obj=56398 op='TABLE AC-
CESS FULL OBJ_ (cr=2503 pr=0
pw=0 time=70470 us)'

STAT #34 id=1 cnt=11000 pld=0
pos=1 obj=56398 op='TABLE AC-
CESS FULL OBJ_ (cr=2171 pr=0
pw=0 time=50909 us)'

```

An dieser Stelle hat der Bind-Peeking-Mechanismus der 10g-Datenbank also dazu geführt, dass eine Abfrage, die vor der Komprimierung über einen Range-Scan auf die Tabelle zugriff, nach der Komprimierung einen Full-Scan ausführt.

Adaptive Cursor Sharing in 11g

In Release 11g werden mit Bind-Variablen versehene Abfragen anders behandelt als in 10g. Das Adaptive Cursor Sharing ist in der Lage, zu erkennen, ob der Ausführungsplan einer Abfrage empfindlich auf die Änderungen der übergebenen Werte reagiert. Wenn der

Optimizer dies feststellt, erzeugt er bei Veränderung der übergebenen Werte einen neuen Ausführungsplan. Das folgende Beispiel verdeutlicht, dass sich der Optimizer bei Übergabe der OBJ_IDs 73000, 76000 und 79000 für einen Full-Scan entscheidet, für OBJ_IDs ab 82000 jedoch einen Index-Scan bevorzugt:

```

SELECT sbc.child_number as cn,
sbc.value_string as bin-
dval,
sp.operation || ' . . . ' ||
sp.options as plan_operation
FROM v$sql_plan sp,
v$sql_bind_capture sbc
WHERE sp.sql_id = sbc.sql_id
AND sbc.sql_id = .7q7unbw9f-
wgpu*
AND sp.operation = ,TABLE
ACCESS*
AND sp.child_address = sbc.
child_address
ORDER BY sbc.child_number;

```

```

CN BINDVAL PLAN_OPERATION
-----
1 73000 TABLE ACCESS FULL
2 76000 TABLE ACCESS FULL
3 79000 TABLE ACCESS FULL
4 82000 TABLE ACCESS BY IN-
DEX ROWID
5 85000 TABLE ACCESS BY IN-
DEX ROWID
6 90000 TABLE ACCESS BY IN-
DEX ROWID

```

Fazit

Sowohl in 10g als auch in 11g kann eine Komprimierung von Tabellen den Ausführungsplan eines Statements verändern. In 10g wird durch Bind-Variablen-Peeking eine mit Bind-Variablen versehene Abfrage unter ungünstigen Umständen mit einem für fast alle Abfragen unpassenden Ausführungsplan versehen. Das in 11g eingeführte Adaptive Cursor Sharing schafft in dieser Situation Abhilfe, da der Optimizer bei ausreichend verschiedenen Werten der Variablen neue Ausführungspläne für die Abfrage erstellt.

Kontakt:

Matthias Mann
matthias.mann@maihiro.com