Automatic Parallel Execution in der Praxis

Peter Heumel
Value Transformation Services
München, Deutschland

Schlüsselworte
PARALLEL_DEGREE_POLICY, Automatic Parallel Execution, Parallel Query, Data Warehouse, Performance

Einleitung

Mit Oracle 11 Release 2 wurde Parallel Query um drei neue Features erweitert: automatischer Parallelisierungsgrad (auto DOP), In-Memory Parallel Query (IMPQ) und Parallel Statement Queuing. Die Nutzung von Parallel Query sollte vereinfacht – nach Oracle's Ansicht sogar völlig automatisiert – werden. Was zeigt die Praxis?

Automatischer Parallelisierungsgrad (automatic Degree of Parallelism)

Seit der Einführung von Parallel Query in Oracle 7.1 mußte man dem Cost Based Optimizer (CBO) einen festen (statischen) Parallel Degree zur Berechnung mitgeben, wollte man, daß Abfragen mittels Parallel Query beschleunigt werden. Nur dann wurde Parallel Query bei der Plan Erstellung vom CBO in Erwägung gezogen.

Warum das so ist, läßt sich leicht überlegen. Oracle's CBO [1] berechnet für alternative Zugriffspfade die jeweiligen Kosten als Maß für die Laufzeit. Der Ausführungsplan mit den geringsten Kosten (= geringste geschätzte Laufzeit) wird ausgewählt und ausgeführt. In Anlehnung an Amdahl's Law lassen sich die Kosten eines Zugriffspfads durch $cost = cost_s + cost_p/dop$ abschätzen. $cost_s$ sind die Kosten des seriellen Teils des Zugriffspfads und $cost_p$ die Kosten für den Teil, der durch Parallelisierung beschleunigt werden kann. dop ist der Parallel Degree.
In einer idealen Betrachtung sind die Kosten für einen parallelen Zugriffspfad $0 + cost_p/dop$. Durch geeignet hohe Parallelisierung wird der Wert beliebig klein und damit besser als jeder serielle Zugriffspfad, der mit $cost_s$ anzusetzen ist. Der konkrete Parallelisierungsgrad spielt also eine entscheidende Rolle. Ohne diesen lassen sich die Kosten für die unterschiedlichen Zugriffspfade nicht vergleichen.

Man konnte den Parallel Degree auf Objektebene, Statementebene oder Sessionebene vorgeben. Es gab zwar auch einen DEFAULT Parallel Degree. Bei genauem Hinsehen ist das aber ebenfalls ein statischer Wert, der sich aus der Anzahl der zur Verfügung stehenden CPU-Threads berechnet.

Kurz: ohne vorgegebenen statischen Parallel Degree gab es vor Oracle 11.2 keine parallele Ausführung.

Ab Oracle 11.2 ändert sich dies nun. Mit dem Automatischen Parallelisierungsgrad kann man sich basierend auf Eigenschaften der Query (Größe der beteiligten Tabellen usw.) einen dynamischen Parallelisierungsgrad automatisch errechnen lassen. Dieses Feature ist die Geburtsstunde von Automatic Parallel Query.

Eingeschaltet wird das Feature durch **PARALLEL_DEGREE_POLICY=AUTO** (oder LIMITED).

Mit dieser Einstellung berechnet der CBO zunächst den besten seriellen Plan (dop=1).
Die Kosten, die Oracle in der Einheit "single block reads" ausgibt, werden in Zeit umgerechnet. Ist die Laufzeit des besten seriellen Plans kleiner als 10 Sekunden, dann wird dieser Plan ausgeführt.
Andernfalls errechnet der CBO einen "optimalen" Parallelisierungsgrad und führt die Zugriffsplanerstellung mit diesem noch einmal durch (also mit dop="optimaler" errechneter Parallelisierungsgrad).
Es werden also noch einmal die unterschiedlichen Zugriffspläne bewertet, wobei zur Kostenberechnung bei parallelen Zugriffspfaden der errechnete Parallelisierungsgrad genutzt wird.
Der beste Plan dieser zweiten Zugriffspfad Berechnung wird gewählt. Dieser kann parallele Abarbeitung enthalten.

Oracle 11.2 sieht für dieses Feature zwei Parameter zur Konfiguration vor:

- die Schwelle für die geschätzte Laufzeit, ab der der automatische Parallelisierungsgrad errechnet und die Planerstellung mit diesem erneut durchgeführt wird (PARALLEL_MIN_TIME_THRESHOLD)
- die obere Schranke für den Parallelisierungsgrad (PARALLEL_DEGREE_LIMIT)

Der Default für die Schwelle ist AUTO, was identisch mit dem Wert 10 Sekunden ist. Nur wenn die geschätzte Laufzeit größer als dieser Wert ist, tritt die Berechnung eines automatischen Parallelisierungsgrades auf den Plan.
Der Default für die Schranke berechnet sich gleich zum DEFAULT Parallel Degree. Der berechnete Parallelisierungsgrad hat maximal diesen Wert.
Lassen wir einmal diese beiden Parameter bei ihrem jeweiligen Default.

Es stellen sich sofort zwei Fragen: Wie berechnet der CBO den optimalen Parallelisierungsgrad? Und was, wenn dieser nicht zu meiner Anwendung paßt? Zu dem „Wie" gibt es kaum etwas in der Dokumentation nachzulesen und eine manuelle Konfiguration ist in 11.2 nicht vorgesehen.
Erschwerend kommt hinzu, daß sich die unterschiedlichen Oracle Versionen unterschiedlich verhalten.
Zu allem Übel ändert sich die Berechnung als Seiteneffekt von bestimmten Änderungen in der Datenbank, die man nicht mit diesem Feature in Verbindung bringen würde. Selbst Kerry Osborne äußerte sich bezüglich der Implementierung in Oracle 11: "auto DOP was too hard to control and thus too scary for production systems"[6]. Auf Kundenwunsch wurde das Feature bei uns allerdings bereits in Oracle 11 eingesetzt. Daher mußten wir uns mit diesem Feature auseinandersetzen und Lösungen finden.

Wie berechnet der CBO den optimalen Parallelisierungsgrad?

Durch Betrachtung von 10053 Traces und den Berechnungen des CBO in den Ausführungsplänen (explain plan) kommt man dem Optimizer auf die Spur. In Oracle 11.2 wird scheinbar nur das notwendige IO (io_dop) betrachtet.

Im Kern beruht die Berechnung in Oracle 11.2 auf der Objektgröße (num_blocks*block_size) und der Scanrate. Es gilt **dop = floor(object_size/(scanrate*ptu))**, wobei ptu = 10 sec ist. Mit der Funktion floor wird dabei auf eine ganze Zahl abgerundet.

Beispiel: ist eine Tabelle 6000 MB groß und die Scanrate 200 MB/s, dann ist der errechnete Parallelisierungsgrad 3.

Die **Objektgröße** beruht auf den Objektstatistiken (`DBMS_STATS.GATHER_TABLE_STATS` usw.) und wird bei Fehlen dieser durch Dynamic Sampling erhoben.

Die herangezoge **Scanrate** unterscheidet sich je Oracle Version. Oracle 11.2.0.1 und 12.1.0.2 nutzen einen Default von 200 MB/s. 11.2.0.2 bis 11.2.0.4 kennen keinen Default und setzen zwingend eine IO Kalibrierung des Systems mittels `DBMS_RESOURCE_MANAGER.CALIBRATE_IO` voraus. Der Wert fällt dann meist viel zu hoch aus. Fehlt die Kalibrierung so wird stillschweigend das Feature ignoriert und das Statement seriell ausgeführt (im Plan findet sich der Hinweis: `automatic DOP: skipped because of IO calibrate statistics are missing`). In 11.2.0.1 führen Systemstatistiken dazu, daß nicht die Default Scanrate sondern ein aus den Systemstatistiken errechneter (viel zu niedriger) Wert genutzt wird.

In Oracle 12 ist inzwischen eine manuelle Konfiguration mittels `DBMS_STATS.SET_PROCESSING_RATE` möglich. Diese wird aber aufgrund eines Bugs ignoriert, es sei denn man setzt `_optimizer_proc_rate_source='MANUAL'`. In 12.2 soll dieser Bug behoben sein. [2]

Ab Oracle 12 werden auch CPU Kosten zur Berechnung des automatischen Parallelisierungsgrades herangezogen (cpu_dop).

Manuelle Konfiguration des automatischen Parallelisierungsgrades in Oracle 11.2

Offensichtlich hatten auch andere Oracle Kunden Probleme mit der durch die IO Kalibrierung ermittelten Werte. Es bestand Bedarf die Konfiguration manuell anzupassen. Oracle behalf sich, indem in My Oracle Support (MOS) Implementierungsdetails bekannt und ein Workaround an die Hand gegeben wurde [5]. Die von `DBMS_RESOURCE_MANAGER.CALIBRATE_IO` ermittelten Werte werden in der Tabelle `SYS.RESOURCE_IO_CALIBRATE$` geschrieben und dürfen per SQL geändert werden. Die Scanrate findet sich in der Spalte `MAX_PMBPS` (maximal parallel MB/s). Um eine Scanrate von 30 MB/s einzutragen – um die gewünschte Parallelisierung zu erreichen - ist das Vorgehen:

```
delete from sys.resource_io_calibrate$;
insert into sys.resource_io_calibrate$ values
(systimestamp,systimestamp,0,0,30,0,0);
commit;
```

Und dann ist ein Instance Restart notwendig, damit die neuen Werte auch tatsächlich herangezogen werden. Durch Experimentieren mit unterschiedlichen Werten, wurden die für unsere Anwendungen passenden Werte ermittelt.
Allerdings gibt es einen Haken: wird versehentlich eine IO Kalibrierung vorgenommen, dann wird der manuell eingefügte Wert überschrieben und dieser zieht dann auch noch sofort. Der manuelle Wert kann nur über eine Downtime (Restart) wiederhergestellt werden. In Produktion ist dies höchst unangenehm.

Manuelle Konfiguration des automatischen Parallelsierungsgrades in Oracle 12c

Das eben beschriebene Verfahren funktioniert auch in Oracle 12c. Allerdings ist hier der Umstieg auf `DBMS_STATS.SET_PROCESSING_RATE` zu empfehlen, da mit dieser Prozedur jetzt die manuelle Konfiguration allgemein unterstützt wird. Um die Scanrate auf 30 MB/s zu setzen, ist folgendes Kommando notwendig:

```
exec dbms_stats.set_processing_rate('IO_ACCESS',30);
```

Der Wert zieht sofort (kein Durchstarten notwendig) und eine unbeabsichtigt durchgeführte IO Kalibrierung hat auf die Berechnung des automatischen Parallelisierungsgrades keine Wirkung, da dieser Wert an anderer Stelle und nicht in `SYS.RESOURCE_IO_CALIBRATE$` abgespeichert wird. Über die View `V$OPTIMIZER_PROCESSING_RATE` kann man sich die für auto DOP wichtigen Werte in Oracle 12 anschauen. Ab 12.2 ist vermutlich auch der Parameter `_optimizer_proc_rate_source='MANUAL'` obsolet, der in 12.1.0.2 zusätzlich gesetzt werden muß. Oracle hat gegenüber 11g also deutlich nachgebessert.

Was ist das Optimierungsziel für den automatischen Parallelisierungsgrad?

Man kann experimentell – durch iterative Anpassung der Scanrate – den für die eigene Anwendung passenden Wert ermitteln und konfigurieren. Ein bestehender Parallel Degree auf Objektebene kann mittels der Formel scanrate = object_size/(dop * ptu) in einen passenden Startwert für die Iteration umgerechnet werden.
Unabhängig von diesem iterativen Verfahren kann man sich aber auch fragen, welches Optimierungsziel Oracle mit dieser Berechnung im Sinn hatte. Wenn man die Scanrate auf den tatsächlichen Wert setzt, die ein parallel Query Slave maximal verarbeiten kann – und das scheint die Intention in 11.2 zu sein -, dann gibt ptu die gewünschte Laufzeit in Sekunden an. Das Ziel für den errechneten Parallelisierungsgrad wäre folglich, SQL-Statements im worst case in 10 Sekunden abzuarbeiten. Dies paßt auch zum Default von `PARALLEL_MIN_TIME_THRESHOLD`. Ist die Laufzeit des besten seriellen Plans größer als 10 Sekunden, dann tritt auto DOP auf den Plan mit dem Ziel, die Laufzeit auf 10 Sekunden herunterzudrücken.

Visualisierung des auto DOP Optimierungsziels

Der automatische Parallelisierungsgrad ist abhängig von der Objektgröße (Objektstatistiken oder Dynamic Sampling) und der Scanrate. Das Ergebnis des CBO Rechenmodels hinsichtlich des automatischen Parallelisierungsgrades und der sich daraus ergebenden geschätzten Laufzeit läßt sich visualisieren. So errät man am Schnellsten das Optimierungsziel.
Im Diagramm wird die geschätzte Laufzeit und der gewählte Parallelisierungsgrad in Abhängigkeit zur Objektgröße dargestellt. Die Objektgröße wurde durch `exec DBMS_STATS.SET_TABLE_STATS('SCOTT','T_EMP_20',numblks=>&1);` schrittweise vergößert. Der errechnete Parallelisierungsgrad wurde dem Ausführungplan entnommen.

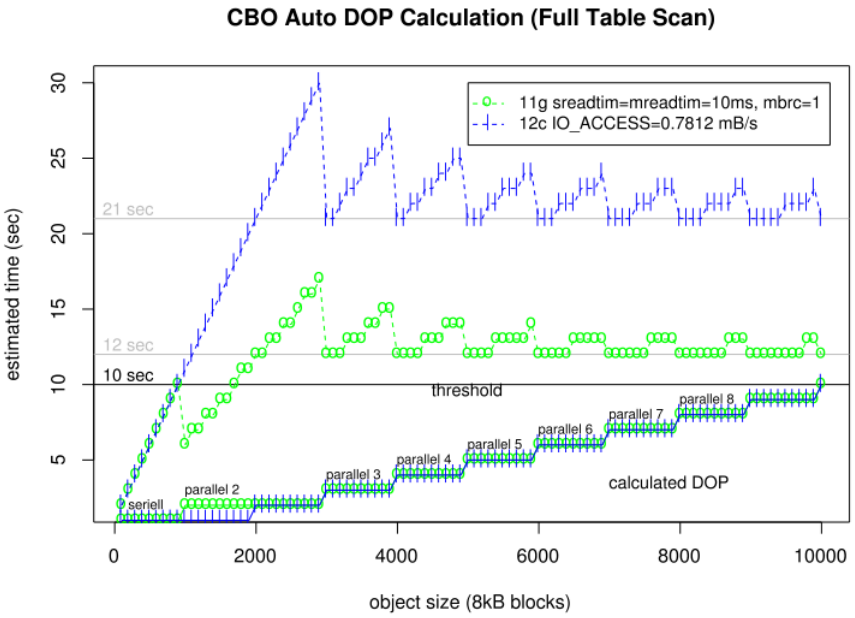
```
select avg(sal) from t_emp_20;
```

```
-------------------------------------------------------------------------------------------------------------
| Id  | Operation              | Name     | Rows  | Bytes | Cost (%CPU)| Time     |    TQ  |IN-OUT| PQ Distrib |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |          |     1 |     4 |   557   (0)| 00:00:06 |        |      |            |
|   1 |  SORT AGGREGATE        |          |     1 |     4 |            |          |        |      |            |
|   2 |   PX COORDINATOR       |          |       |       |            |          |        |      |            |
|   3 |    PX SEND QC (RANDOM) | :TQ10000 |     1 |     4 |            |          | Q1,00 | P->S | QC (RAND)  |
|   4 |     SORT AGGREGATE     |          |     1 |     4 |            |          | Q1,00 | PCWP |            |
|   5 |      PX BLOCK ITERATOR |          |    14M|    56M|   557   (0)| 00:00:06 | Q1,00 | PCWC |            |
|   6 |       TABLE ACCESS FULL| T_EMP_20 |    14M|    56M|   557   (0)| 00:00:06 | Q1,00 | PCWP |            |
-------------------------------------------------------------------------------------------------------------

Note
-----
   - automatic DOP: Computed Degree of Parallelism is 2
```

Zu beachten ist, daß auf der y-Achse zum einen die geschätzte Laufzeit (für die beiden oberen Kurven) und zum anderen der errechnete Parallelisierungsgrad (für die beiden unteren Kurven) aufgetragen ist.

Auf der x-Achse ist die Objektgröße (Anzahl von 8 kB Blöcken) aufgetragen.



Für Oracle 11.2 sieht man sehr schön, daß das Optimierungsziel die Laufzeit (scheinbar 12 sec) ist und auto DOP nach Überschreiten der Schwelle einsetzt, um die Laufzeit bei wachsender Objektgröße auf 12 Sekunden zu senken. Alle 1000 Blöcke erhöht sich der errechnete Parallelisierungsgrad um eins.

Tatsächlich hat die Berechnung des automatischen Parallelisierungsgrades das Ziel von 10 Sekunden (=ptu). Der CBO rechnet zusätzlich mit einem Overhead bei Parallel Query. Daraus ergibt sich dann die höher geschätzte Laufzeit von 12 Sekunden.

Für Oracle 12.1 ergibt sich eine noch größere Differenz (21 sec), da der CBO jetzt merkwürdigerweise die Kosten für DOP=2 identisch mit den seriellen Kosten ansetzt. Hinsichtlich der Berechnung des automatischen Parallelisierungsgrades gibt es aber nur einen minimalen Unterschied (bei einer Objektgröße von 1000-1900 Blöcken), sodaß die beiden unteren Kurven weitgehend übereinader liegen.

In-Memory Parallel Query

Sehr früh nach der Einführung von Parallel Query in 7.1 traf Oracle die für die damalige Zeit richtige Entscheidung, daß im Falle von Parallel Query Blöcke nicht gecached werden. Parallel Query nutzte von da an "direct reads" an der SGA (buffer cache) vorbei. Mit Oracle 11 Release 2 änderte sich das für diejenigen, die `PARALLEL_DEGREE_POLICY=AUTO` setzten. Falls das Objekt, das gescannt werden soll, kleiner als 80% des Buffer Caches ist, ist nun ein Caching ebenfalls möglich. Die 80% (default) können per `_parallel_cluster_cache_pct` angpaßt werden.
Allerdings ist offiziell nicht vorgesehen, dieses Feature auch unabhängig vom automatischen Parallelisierungsgrad einzusetzen.
Sinn macht ein Caching natürlich nur für Objekte, auf die wiederholt zugegriffen wird. Das Ziel von 10 Sekunden für den Parallelisierungsgrad kann durch Caching deutlich unterschritten werden, sodaß die tatsächliche Antwortzeit deutlich niedriger liegen kann.
Für RAC Umgebungen bringt das Feature eine besondere "Intelligenz" mit [3]. Erstens wird - sofern der Parallelisierungsgrad sich nicht ändert, was mit auto DOP der Fall ist - jeder Datenblock nur auf

einer einzigen Instanz gecached. Zweitens lesen die PX Slaves nur lokal gecachedte Blöcke. Falls ein Block lokal nicht gecached ist, wird er von Platte gelesen, selbst wenn er im Cache einer anderen Instanz vorhanden ist. Über ein Hashverfahren wird sichergestellt, daß das Scannen von Objekten so auf PX Slaves aufgeteilt wird, dass sie die ihnen zugeteilten Blöcke lokal vorfinden und nicht von Platte lesen müssen. Im Ausführungsplan steht `- parallel scans affinitized for buffer cache` falls IMPQ für ein Statement vorgesehen wird.

Diese Optimierung geht verloren, wenn unterschiedliche Parallelisierungsgrade genutzt werden. So könnten bei global konfiguriertem auto DOP einzelne Sessions auch einen manuell gesetzten Parallel Degree nutzen. In-Memory Parallel Query bleibt auch für diese Abfragen mit maunellem DOP aktiviert. Da das Hashverfahren vom DOP abhängt, geschieht aber eine andere Zuordnung der Blöcke zu Instanzen, wenn sich der DOP ändert. Manche Blöcke werden in diesem Fall erneut von Platte gelesen und zusätzlich in einer weiteren Instanz gecached. Dies kostet Zeit. Wenn sich der Parallelisierungsgrad häufig genug ändert, dann hat man zudem irgendwann alle Blöcke eines Objektes mehrfach und schlußendlich auf allen Instanzen gecached. Dadurch wird mehr Hauptspeicher als bei gleichbleibendem Parallelisierungsgrad belegt.

In-Memory Parallel Query wird durch `_parallel_cluster_cache_policy=CACHED` eingeschaltet. Durch `PARALLEL_DEGREE_POLICY=AUTO` geschieht das implizit.

In der Praxis mußten wir in einer sehr frühen Version (11.2.0.2) IMPQ vorübergehend (bis zum Vorliegen eines Bugfixes) durch `_parallel_cluster_cache_policy=adaptive` abschalten, da sporadisch falsche Ergebnisse bei Tabellen mit chained rows auftraten (Bug 10026972). An dieser Stelle sei auf die Reference Note zu Automatic Parallel Execution in MOS verwiesen, wo alle bekannten Bugs und Fixes für alle drei Features aufgelistet sind [4].

Auf Exadatas muß man zudem zwischen Smart Scans (direct read) und IMPQ abwägen. Laut Oracle [3] sollte in Oracle 12.1.0.2 der Optimizer hier selbst die beste Wahl treffen. In der Praxis sollte man aber besser testen und nachjustieren.

Parallel Statement Queuing

Auch dieses Feature ist eine interessante und nützliche Erweiterung für Parallel Query allerdings mit ein paar Fallstricken. Schon der Name ist irreführend, da Sessions und nicht Statements gequeued werden. Aber der Reihe nach: mit `PARALLEL_DEGREE_POLICY=AUTO` wird dieses Feature aktiviert. Wenn der CBO für eine Abfrage einen parallelen Ausführungsplan auswählt, dann wird geprüft, ob bereits mehr als `PARALLEL_SERVERS_TARGET` Parallel Query Slaves (PX Slaves) aktiv sind.

- Die Session darf die Query sofort ausführen, wenn dieses “soft limit” noch nicht überschritten ist. Zudem bekommt die Session die tatsächlich angeforderte Anzahl an Parallel Query Slaves unabhängig davon, wie weit das “soft limit” dadurch überschritten wird.
- Sind im System bereits mehr Parallel Query Slaves aktiv als dieses “soft limit” erlaubt, wird die Abfrage solange gequeued bis genügend Resourcen vorhanden sind (d.h. die aktive Anzahl an Parallel Query Slaves im System kleiner gleich dem Limit ist).

Lediglich die Anzahl an Parallel Query Slaves, die im System aktiv sein dürfen ohne daß Queuing auftritt, ist mit `PARALLEL_SERVERS_TARGET` an die eigene Anwendungssituation anzupassen. Soweit die Oracle Dokumentation.

Die Motivation für die Implementierung dieses Features ist zweierlei:

- Queries sollen immer mit ihrem optimalen Parallel Degree ohne Downgrades ausgeführt werden.
- Unterstützung eines gemischten Betriebes von OLTP und Data Warehouse Abfragen in einem "operativen Data Warehouse".

Ohne dieses Feature würden zum einen Parallel Queries die System Resourcen übermäßig stark in Anspruch nehmen und OLTP Abfragen stark ausbremsen. Zum anderen würden Parallel Queries mit einem niedrigeren Parallel Degree ausgeführt als angfordert, wenn nicht genügend Resourcen (Parallel Query Slaves) zur Verfügung stehen. Dieser "parallel degree downgrade" Mechanismus ist defaultmäßig durch `PARALLEL_ADAPTIVE_MULTI_USER=TRUE` aktiv. Mit Parallel Statement Queuing sollte dieser Mechanismus ausgeschaltet werden.

Um das Feature richtig einzuschätzen, muß man sich zudem klarmachen, daß Parallel Statement Queuing ein komplett neues Abarbeitungsmodell innerhalb der Datenbank ist. Die durchschnittliche Antwortzeit für Einzelabfragen ist bei hoher Last niedriger als ohne Parallel Statement Queuing, was für interaktive Abfragen wünschenswert ist. Warum das so ist, kann man sich leicht überlegen. Angenommen wir haben eine Parallel Query, die unser Datenbanksystem allein bereits voll auslastet und eine Servicezeit t_s hat. Diese Abfrage wird von n-Benutzern gestartet. Im Idealfall startet der nächste seine Abfrage genau dann, wenn die vorige Abfrage fertig ist. Das System ist dann optimal ausgelastet. Die Laufzeit ist für jeden genau der Servicezeit t_s (sieht man von Caching usw. ab). Was aber wenn alle ihre Abfrage gleichzeitig starten (worst case)? Ohne Parallel Statement Queuing liegt die Laufzeit dann bei etwa $n*t_s$. Denn das Betriebssystem wird für jede Abfrage immer ein bestimmtes "time slice" zur Verfügung stellen, bis die nächste Abfrage der Reihe nach ihr "time slice" bekommt. Alle Abfragen werden ungefähr gleichzeitig fertig. Je mehr Last im System ist, desto langsamer werden alle Abfragen. Mit Parallel Statement Queuing wird dagegen die erste Abfrage in t_s laufen (also wie ohne Last), da alle anderen n-1 Abfragen in der Queue warten. Die zweite Abfrage hat eine Laufzeit von $2*t_s$ (t_s + Wartezeit $1*t_s$). Die dritte Abfrage hat eine Laufzeit von $3*t_s$ (t_s + Wartezeit $2*t_s$) usw. Die durchschnittliche Antwortzeit ohne Queuing ist t_s*n und mit Queuing $t_s*(n+1)/2$. Der Durchsatz an Statements ist dabei für beide Methoden gleich.

Jetzt aber zu den Fallstricken: während automatischer Parallelisierungsgrad und In-Memory Parallel Query – sieht man einmal von Bugs in frühen Versionen dieser beiden Features ab – relativ wenig anwendungsspezifische Probleme bereiten, ist dies bei Parallel Statement Queuing anders. Die Anwendung muß sich für dieses Feature eignen.

Meist (abgesehen von interaktiven Abfragen) besteht eine Anwendung aus vielen SQL-Statements, die notwendig sind, um einen Verarbeitungsschritt durchzuführen oder einen Report zu erstellen. Der Vorteil von Parallel Statement Queuing nutzt wenig, wenn zwar die erste Abfrage schnell durchläuft, die nächste aber – ohne die ein Report z.B. nicht angezeigt werden kann – wieder ganz hinten in der Queue landet. `DBMS_RESOURCE_MANAGER.BEGIN_SQL_BLOCK/END_SQL_BLOCK`, mit dem man Statements in der Queue nach vorne bringen kann, hilft nur sehr begrenzt.
Schlimmer noch: es kann regelrecht zu Deadlocks in der Anwendung kommen, die nicht vom Datenbanksystem aufgelöst werden.

Mit "Spread Sheet" Anwendungen – also Anwendungen, die Ergebnismengen tabular wiedergeben, die ein Benutzer interaktiv durchblättern kann - kann sich noch ein ganz anderes Problem ergeben: es wird gequeued, obwohl im Datenbanksystem kaum oder keine Last zu sehen ist. Browsen Benutzer Abfrageergebnisse mit Endbenutzer Tools wie SQL*Developer oder TOAD, dann endet man bei hohen Parallelisierungsgraden schnell in einer Situation, daß das Limit von `PARALLEL_SERVERS_TARGET` erreicht wird, ohne daß man nennenswerte Last am System feststellt. Das liegt daran, daß diese Tools einen Curser öffnen und dann die ersten n Sätze (z.B. 50)

anzeigen. Die Parallel Query Slaves für diese Abfragen bleiben im System, haben aber nichts zu tun, bis der Benutzer im Ergebnis weiterblättert. Sie werden als aktiv gezählt und führen dazu, daß weitere Abfragen, die parallelisieren wollen, in der Queue landen, weil das Limit bereits überschritten ist.

Beispiel: Wir setzen `PARALLEL_SERVERS_TARGET=2` (unrealistisch niedrig). Im SQL*Developer wird dann die Abfrage `select empno, sal from t_emp_20 order by 1,2;` abgesetzt. Die Abfrage bekommt die per auto DOP gewünschen 4 (=2*2) Parallel Query Slaves (`- automatic DOP: Computed Degree of Parallelism is 2)`. Es werden die ersten 50 Sätze angezeigt ("Fetched 50 rows in 25.246 seconds"). Eine weitere Abfrage eines anderen Benutzers aus SQL*Plus (`select * from t_emp_20)` wird jetzt gequeued und der Benutzer bekommt keinerlei Ergebnisse, obwohl die Slaves der ersten Abfrage im Moment nichts zu tun haben (bis der Benutzer weiterblättert).
In `GV$SESSION` und `GV$PX_SESSION` sieht man die relevanten Informationen.

```
PXStat    PXDegre SQL_HASH_VALUE QC_SID EXEC_ROWS  EVENT              P1RAW      STMT
--------- ------- -------------- ------ ---------- ------------------ ---------- ------------------------------
QC                            0 43@1                                 SQL Develo -- cursor stmt --
1:Slave1  2<=2        2217789247               150                    SQL Develo /* + NO_PARALLEL */select empn
          2<=2                                                        SQL Develo
1:Slave2  2<=2        2217789247                                     SQL Develo
          2<=2                                                        SQL Develo

                      3437852417                 0 resmgr:pq queued   SQL*Plus   select * from t_emp_20
```

Über `GV$RSRC_CONSUMER_GROUP` kann man sich einen Überblick über aktive PQ Sessions und solche, die in der Queue sind, verschaffen.

```
NAME                           ACTIVE_SESSIONS EXECUTION_WAITERS CURRENT_PQ_SERVERS_ACTIVE CURRENT_PQS_QUEUED
------------------------------ --------------- ----------------- ------------------------- -------------------
OTHER_GROUPS                                 3                 0                         4                   1
```

Ein weitere Eigenschaft von Parallel Statement Queuing, kann man ebenfalls im SQL*Developer demonstrieren. Hat eine Session eine Parallel Query offen, dann können in dieser Session beliebig viele weitere Parallel Queries gestartet werden, ohne daß diese in der Queue landen. Das Target für Parallel Query Server kann dabei beliebig hoch überschritten werden. Dies ist der Grund dafür, daß sich das Feature eher wie ein Session Queuing als Statement Queuing anfühlt.

Parallel Statement Queuing wird durch `_parallel_statement_queuing=TRUE` eingeschaltet. Durch `PARALLEL_DEGREE_POLICY=AUTO` geschieht das implizit. Man kann diese implizite Setzung aber überschreiben `_parallel_statement_queuing=FALSE`.

Fazit

Mit `PARALLEL_DEGREE_POLICY=AUTO` schaltet man drei Features für Parallel Query ein:

- Automatischer Parallelisierungsgrad und damit Automatic Parallel Execution
- In-Memory Parallel Query
- Parallel Statement Queuing

Der Name des Parameters ist irreführend, da er nur auf das erste dieser drei Features verweist: dem automatischen Parallelisierungsgrad. Die Setzung `AUTO` ist aufgrund der Bünderlung dieser drei Features gut geeignet für interaktive Abfragen in einem operativen Data Warehouse. Denn für Data Warehouse Abfragen, die typischerweise größere Datenmengen verarbeiten und von Parallel Query stark profitieren, wird durch diese Setzung folgendes erreicht:

- der CBO berücksichtigt Parallel Query bei der Planerstellung automatisch
- der Parallel Degree wird dynamisch gewählt

- Parallel Degree Downgrades werden durch Queuing vermieden
- niedrige durchschnittliche Antwortzeiten werden selbst unter Last erzielt
- Caching wird auch für Parallel Query genutzt
- die Resourcen für Parallel Query werden limitiert, sodaß der OLTP Betrieb nicht behindert wird

Für andere Anwendungen (z.B. ETL im Data Warehouse, Batch) muß man im einzelnen überlegen, welche der drei Features von Vorteil sind und wo es Probleme geben könnte. Die drei Features können mittels Underscore Parameter getrennt an- und ausgeschaltet werden. Generell gilt für Underscore Parameter, daß diese mit Vorsicht (und nach Bestätigung von Oracle Customer Support) eingesetzt werden. Zudem hat man keine Garantie, daß bei Versionswechseln die Parameter und ihre Funktionalität erhalten bleiben.

References

[1] J. Lewis, Cost-Based Oracle Fundamentals, Berkeley: Apress, 2006.

[2] Y. Baskan, Optimizer Processing Rates for Auto DOP (The Data Warehouse Insider).

[3] Y. Baskan, In-Memory Parallel Query (The Data Warehouse Insider), 2016.

[4] MOS, Init.ora Parameter "PARALLEL_DEGREE_POLICY" Reference Note (Doc ID 1216277.1), Oracle.

[5] MOS, Automatic Degree of Parallelism in 11.2.0.2 (Doc ID 1269321.1), Oracle.

[6] K. Osborne, 12c - parallel_degree_level (control for auto DOP), Kerry Osborne's Oracle Blog.

[7] Kerry Osborne, Randy Johnson und Tanel Poder, „Expert Oracle Exadata," Berkeley, Apress, 2011, pp. 143-173.

Kontaktadresse

Peter Heumel
Value Transformation Services
Am Tucherpark 12
D-80538 München

Telefon: +49 (0) 89-378 26920
E-Mail Peter.Heumel_V-TServices@de.ibm.com