Der Artikel stellt Test-Ergebnisse zur Skalierbarkeit und Performance des Oracle RAC vor. Getestet wurde auf der 10*g*R2-Chip-Multithreading-Architektur UltraSPARC T2, die auch unter ihrem Codenamen „Niagara 2 CPU" bekannt ist. Die Tests erfolgten in einem realen Projekt für einen Kunden und brachten durchaus überraschende Resultate.

Vorsicht bei parallelen Abfragen eines Oracle RAC auf Multithreading-Chipsätzen

Piotr Sajda, OPITZ CONSULTING GmbH

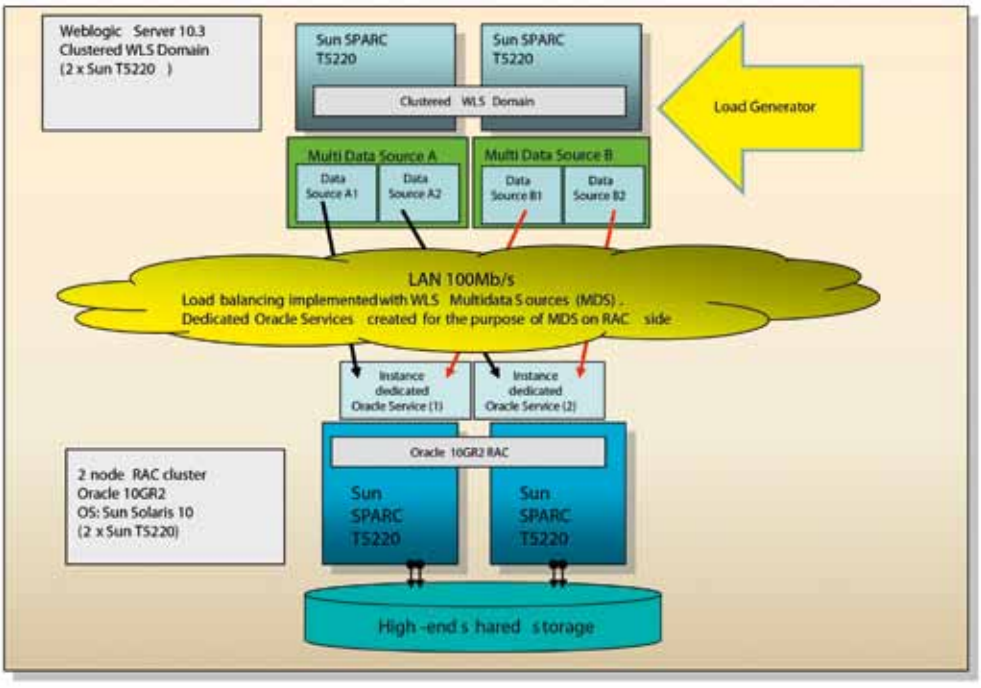
Ziel der Tests war es, die Nutzbarkeit der Chip-Multithreading-(CMT)-Architektur innerhalb einer skalierbaren und effizienten RAC-Umgebung, die mehrere OLTP-Datenbanken innerhalb eines Host-RAC umfasst, zu untersuchen.

Die Tests erfolgten unter hoher Auslastung mit unterschiedlichen Typen kommerzieller und Open-Source-Anwendungen. In diesem Artikel stelle ich die Tests mit DVD Store vor, einer Open-Source-Anwendung für E-Commerce.

Testumgebung

Die Untersuchungen fanden in den folgenden Umgebungen statt (siehe Beispiel 1):

- *Middleware*
  Zwei geclusterte Knoten mit dem WebLogic Server 10. Initial/Maximal Connection Pool von 200 Verbindungen/Datenbank (100 physikalische Verbindungen per Datenbank-Instanz)
- *Datenbank*
  Oracle 10*g* R2 RAC Datenbank
- *Betriebssystem*
  Solaris 10
- *Hardware*
  2 x Sun T5220
- *Storage*
  Storage der High-End-Klasse
- *Client*
  DVD-Shop-J2EE-konforme Applikation, eingesetzt auf dem geclusterten WLS-Server



Beispiel 1: Die Testumgebung

Mögliche Ursachen für die Überlastung

Eine erste kurze Analyse des AWR-Berichts brachte schnell die problematischen Bereiche ans Licht, die im Wesentlichen für die Überlastung des RAC Interconnect verantwortlich waren. Die weitere Analyse des „gc-buffer busy"-Wait-Events führte zu einer SQL-Abfrage, die im hohen Maße zur Leistungsminderung beitrug. Deutlich zeigt dies die Spitze kurz nach 5 Uhr bei den Average-Active-Sessions (siehe Beispiel 2).

Die Latenz des RAC Interconnect lieferte keine klaren Hinweise auf Hardware-Probleme oder eine fehlerhafte Konfiguration (siehe Beispiel 3).

Dem Performance-Problem auf der Spur

Die vom „Average CR Block Receive Time" geschätzte Latenz schwankte zwischen zwei und drei Millisekunden und die verfügbare Bandbreite von einem Gigabyte pro Sekunde war auch ausreichend. Und doch gab es anscheinend Probleme in diesem Bereich. Letztendlich war die direkte Ursache dieses Peaks die Abfrage „select count(*) from ORDERS". Merkwürdigerweise wurde diese relativ kleine partitionierte Tabelle (70 MB) mit einem Full Table Scan (FTS) gelesen.

Bei der Analyse des Ausführungsplans der Abfrage und der Antwortzeit kam die entscheidende Abfrage:

```
SQL>  select count(*) from ds2.
orders ;
```

Der Execution-Plan zeigte, dass der Optimizer eine Parallel Query benutzte, die natürlich einen FTS auf die Tabelle ORDERS anwendete und etwa 34 Sekunden dauerte. Im Falle solcher Abfragen wie „select count (*) from <Tabelle>“ besteht keine Notwendigkeit, auf einen einzelnen Block der Tabelle zuzugreifen, wenn ein geeigneter Index existiert. In diesem Fall könnte ein (Fast) Full Index Scan das Ergebnis liefern.

Schon bald fand der Autor einen Index, der als perfekter Kandidat ein schnelleres Ergebnis der Abfrage lieferte. Er verwendete nun einen Hint, um zu testen, ob bei der Verwendung dieses Index bessere (schnellere) Ergebnisse zu erwarten waren:

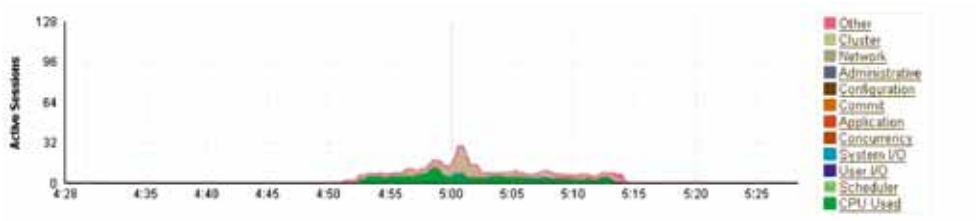
```
SQL> select /*+ index(orders
PK_ORDERS) */  count(*) from
ds2.orders;
```

Erwartungsgemäß lief die Abfrage viel schneller (siehe Beispiel 4), wenn der Optimizer einen INDEX FULL SCAN verwendete. Das Ergebnis stand jetzt nach 2,86 Sekunden fest.

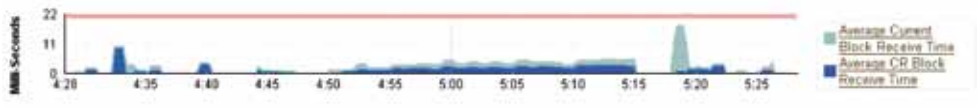
Warum aber entschied sich der Optimizer für einen so ungünstigen Ausführungsplan? Um die Entscheidungswege besser nachvollziehen zu können, wurde der Ausgang des Events 10053 analysiert. Auf diese Weise kam heraus, warum die Verwendung des bekannten Index nicht als bester Ausführungsplan gewählt wurde, warum die Kosten für einen FTS (COST 6) so gering und im Gegensatz dazu die Kosten für einen Full-Index-Scan (COST 3420) so hoch waren.

Der relevante Teil der Trace-Datei zeigte nach dem Tracing der Entscheidungspfade des Optimizers folgendes Ergebnis (siehe Beispiel 5).

In der Tat war der (full) TableScan (siehe Access Path: TableScan) für den Optimizer der effizienteste Plan. Die Zugriffskosten durch einen FTS wurden mit 6,39 berechnet. Im Gegensatz dazu wertete der Optimizer den bevorzugten Zugriff mit dem Full Index Scan mit enormen Kosten in Höhe von 809,89. Doch was bringt den Optimizer dazu, die Kosten des Ausführungsplans mit FTS mit 6,39 zu berechnen,



Beispiel 2: Überblick über die Anzahl der Zugriffe, Average Active Sessions (Current Up instances: 2/2)



Beispiel 3: Überblick über die Zugriffsgeschwindigkeit, Global Cache Block Access Latency (Current Up instances: 2/2)

```
Execution Plan
----------------------------------------------------------------------------
| Id  | Operation             | Name     | Rows  | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |          |   1   |  3420  (3)| 00:00:54 |
|   1 | SORT AGGREGATE        |          |   1   |           |          |
|   2 | INDEX FULL SCAN       | PK_ORDERS | 1423K |  3420  (3 |   00:00:54 |
----------------------------------------------------------------------------

Statistics
-----------------------------------------------------------
      4335  consistent gets
      4480  physical reads
```

Beispiel 4: Ergebnisse des Hints

```
SINGLE TABLE ACCESS PATH
  Table: ORDERS  Alias: ORDERS
    Card: Original: 1207210  Rounded: 1207210  Computed: 1207210.00  Non
Adjusted: 1207210.00
   Access Path: TableScan
    Cost:  1472.06  Resp: 6.39  Degree: 0
      Cost_io: 1410.00  Cost_cpu: 181081500
      Resp_io: 6.12  Resp_cpu: 785944
  Access Path: index (index (FFS))
    Index: PK_ORDERS
    resc_io: 749.00  resc_cpu: 177670962
    ix_sel: 0.0000e+00  ix_sel_with_filters: 1
  Access Path: index (FFS)
    Cost:  809.89  Resp: 809.89  Degree: 1
      Cost_io: 749.00  Cost_cpu: 177670962
      Resp_io: 749.00  Resp_cpu: 177670962
  Access Path: index (FullScan)
    Index: PK_ORDERS
    resc_io: 3420.00  resc_cpu: 279905125
    ix_sel: 1  ix_sel_with_filters: 1
    Cost: 3515.92  Resp: 3515.92  Degree: 1
  Best:: AccessPath: TableScan
         Cost: 6.39  Degree: 256  Resp: 6.39  Card: 1207210.00  Bytes: 0
```

Beispiel 5: Ergebnis nach dem Tracing der Entscheidungspfade des Optimizers

was anschließend zu einem suboptimalen Ausführungsplan führt?

Resp steht für „Kosten für ‚full parallel execution'", die Antwort war wohl in der Architektur des Ausführungsmechanismus' der parallelen Query zu finden. Des Weiteren fiel auf, dass Resp_cpu, also die Kosten für die Nutzung der CPU-Ressourcen, wenn die parallele Option verwendet wird, relativ niedrig war: 785944. Damit war Resp_cpu rund 230-mal niedriger als die „serielle" Option (Cost_cpu: von 181081500).

Auf den ersten Blick könnte man meinen, der Optimizer hätte den Ausführungsplan mit 230 parallelen Prozessen (Slaves) mitgerechnet. Die Analyse des nächsten Abschnitts ergab, dass die Maximalzahl der laufenden parallelen Slaves das Minimum der beiden folgenden Zahlen war:

```
Effective # of Parallel Max
Servers =
min ( parallel_max_servers ,
(cpu_count * parallel_threads_
per_cpu) ) = 128
```

Dies bestätigte die obige Vermutung: 128 Slaves je Instanz ergaben in diesem Fall 256 Slaves für das ganze System, das auf einen Cluster mit zwei Knoten lief. Dies erklärte auch, warum die ermittelten „parallelen" CPU-Kosten etwa 230 mal geringer waren als die der seriellen Option. Getestet wurde auf einem Sun T5220-Rechner mit Multithreading-Chipsätzen mit 64 Threads in einer physikalischen CPU. Damit war die Herkunft dieser Zahlen einfach zu erklären: Der Optimizer-Algorithmus unterscheidet offensichtlich nicht zwischen den physikalischen CPU-Einheiten auf der Platine und den Verarbeitungs-Threads innerhalb einer solchen physikalischen CPU. Die Hardware-Threads erscheinen dem OS als logische CPU-Einheiten. Diese wiederum verwendet der Optimizer zum Erstellen des Werts des cpu_count-Parameters, der beim Starten der Instanz ausgewertet wird. Die Arithmetik, die auf diesen Daten zur Anwendung kommt, führt letztendlich zu einer falschen Entscheidung beim Berechnen der Kosten der einzelnen Ausführungsszenarien.

Ob die parallele Abfrage wirklich 256 Slaves benutzt, ist leicht zu testen, indem man die Abfrage verwendet (siehe Beispiel 6), während im Hintergrund die Abfrage „select count(*) from orders" ausgeführt wird.

Damit ist eindeutig belegt, dass die Abfrage 256 Slaves verwendete, die den Output an den Query Coordinator übergaben (SID=2503). Die Slaves verteilten sich auf zwei Instanzen, die jeweils 128 Slaves erzeugten. Die Zwischenergebnisse wurden an den Query Coordinator, in diesem Fall auf Instanz #2 geschickt.

```
   INST_ID        SID      QCSID    SERVER#          DEGREE REQ_DEGREE
---------- ---------- ---------- ---------- --------------- ----------
         1       3139       2503          1             256        256
         1       3113       2503          2             256        256
(…)
         1       3211       2503        126             256        256
         1       3224       2503        127             256        256
         1       3209       2503        128             256        256
         2       2399       2503        129             256        256
         2       1744       2503        130             256        256
(…)
         2       2663       2503        254             256        256
         2        396       2503        255             256        256
         2       3169       2503        256             256        256
         2       2503       2503
257 rows selected.
```

Beispiel 6: Ergebnisse der Abfrage „SQL> select inst_id, sid, qcsid, server#, degree, req_degree from gv$px_session where QCSID = 2503 order by INST_ID, SERVER# ;"

An dieser Stelle lassen sich folgenden Schlüsse ziehen:

1. Offensichtlich lag kein RAC-System mit 128 CPUs auf jedem Knoten vor, sodass Contention in den inneren Knoten der CPU zu erwarten war.
2. Selbst wenn es diese Ressourcen gegeben hätte, wäre ein derart gestalteter Ausführungsplan nicht skalierbar, da er alle verfügbaren CPU-Ressourcen nur für diese eine Abfrage verwendet. Eine Abfrage, die zur gleichen Zeit ausgeführt würde, müsste also um die CPU konkurrieren.
3. Da die Hälfte der 256 Slaves, die auf Knoten #1 liefen, ihr Ergebnis an den Query Coordinator (auf Knoten #2) übergab, wurde der Interconnect stark beansprucht und die Reaktionszeit nochmals stark reduziert.

Das Letztere kann man in den „Average Active Sessions" unmittelbar nach 5 Uhr beobachten. Interessant ist hierzu der Peak in „Cluster Waits" (siehe Beispiel 2).

Tuning-Möglichkeiten

Es gibt ein paar Wege, die das Problem sofort lösen, doch alle sind weit davon entfernt, mustergültig zu sein. Man könnte beispielsweise Hints einbauen oder den DEGREE-Parameter der Tabelle ändern. Aber diese Ansätze hätten die Situation nur lokal verbessert und nicht global auf andere Objekte gewirkt. Deshalb sollte man zuerst einmal den Optimizer wissen lassen, dass ihm nicht die Anzahl an CPUs zur Verfügung stand, die er angenommen hatte. Ein weiterer Grund für die Reduzierung dieser Größe war, das System so skalierbarer zu machen.

Nach Setzen des Parameters „parallel_max_servers" auf 8 und „parallel_threads_per_cpu" auf 1 war tatsächlich eine signifikante Verbesserung der Reaktionszeit zu erkennen. Der folgende Code erzeugte 16 parallel ausgeführte Slaves, die auf zwei Instanzen laufen und Daten an den Query Coordinator zurückgeben:

```
SQL> alter system set paral-
lel_max_servers=8 scope=both
sid='*' ;
SQL> alter system set parallel_
threads_per_cpu=1 scope=both
sid='*' ;
```

Nach Neustart beider Instanzen läuft die Abfrage erneut, nun mit den folgenden neuen Einstellungen (siehe Beispiel 7):

```
SQL> select count(*) from or-
ders
  COUNT(*)
----------
1423615
Dauer: 00:00:03.21
```

Als Erstes ist zu sehen, dass die Reaktionszeit drastisch gesunken ist – bis auf etwa drei Sekunden im Gegensatz zu den vorherigen 34 Sekunden, als 128 Slaves pro Instanz verwendet wurden. Auch die Zahl der nun ausgeführten Slaves wurde bestätigt: Sie war beim Ausführen der Abfrage mit den neuen Instanz-Einstellungen in der Tat 16 (8 pro Instanz) plus ein Query Coordinator.

Obwohl es nun viel bessere Ergebnisse gibt, erstellt der Optimizer den Ausführungsplan weiterhin mit Full Table Scan (FTS). Unter bestimmten Umständen hätte man damit zufrieden sein können, nämlich dann, wenn die Größe der zugrundeliegenden Tabelle winzig gewesen wäre. Die Segmente dieser Tabelle belegten zu dieser Zeit nur 8905 Blöcke (dies entspricht etwa 70 MB). Diese Tabelle konnte aber durchaus wachsen und viele Gigabyte belegen, womit die Reaktionszeit inakzeptabel würde.

Beim Untersuchen des Outputs von „autotrace“ in Hinblick auf „consistent gets“ (beziehungsweise „physical reads“, falls sich die Blöcke nicht im Buffer Cache befanden), kommt man zu dem Schluss, dass FTS im Vergleich zu Fast Index Scan eine weniger gut skalierbare Lösung darstellt. Um dies zu belegen, reicht ein Blick in die „Statistics“ der letzten Ausführung mit FTS, wo 8755 physikalische Lesevorgänge vorgenommen wurden. In der Tat korrelieren diese mit der Anzahl der von der Tabelle belegten Blöcke (8905). Sobald man den Optimizer zwingt, den Index zu benutzen, fällt die Zahl der physikalischen (und logischen) Lese- und Schreibvorgänge ungefähr auf die Anzahl der Index-Blöcke, die der Index belegt (4401).

Die Reaktionszeit ist im zweiten Fall (bei serieller Ausführung von Full Index Scan) langsamer als bei Verwendung von FTS (Full Table Scan mit paralleler Abfrage).

Mit der parallelen Ausführung von Full Index Scan nutzen wir die Chance, dass der Optimizer einen Full Index Scan mit paralleler Abfrage als beste Ausführung wählt, ohne Hints zu benötigen. Und wirklich, nachdem parallele Operationen auf den Index erlaubt wurden, wählte der Optimizer den parallel ausgeführten Fast Full Index Scan als den besten Ausführungsplan. Nachfolgend die einzelnen Schritte dazu:

1. Zuerst den Buffer Cache leeren, um physikalisches I/O zu erzwingen. Dies wurde auf beiden Instanzen ausgeführt:
   SQL> alter system flush buffer_ cache; System altered.
2. Anschließend die Abfrage ohne Hint ausführen, also trifft der Optimizer die Entscheidung basierend auf seinem Wissensstand (siehe Ab-

```
Execution Plan
----------------------------------------------------------------------------------------------------------------
| Id  | Operation                | Name      | Rows  | Cost (%CPU)| Time     |    TQ  |IN-OUT| PQ Distrib |
----------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |           |     1 |     9  (12)| 00:00:01 |        |      |            |
|   1 |  SORT AGGREGATE          |           |     1 |            |          |        |      |            |
|   2 |   PX COORDINATOR         |           |       |            |          |        |      |            |
|   3 |    PX SEND QC (RANDOM)   | :TQ10000  |     1 |            |          |  Q1,00 | P->S | QC (RAND)  |
|   4 |     SORT AGGREGATE       |           |     1 |            |          |  Q1,00 | PCWP |            |
|   5 |      PX BLOCK ITERATOR   |           |  1423K|     9  (12)| 00:00:01 |  Q1,00 | PCWC |            |
|   6 |       INDEX FAST FULL SCAN| PK_ORDERS |  1423K|     9  (12)| 00:00:01 |  Q1,00 | PCWP |            |

Statistics
----------------------------------------------------------
         48  recursive calls
          0  db block gets
       6817  consistent gets
       4404  physical reads
          0  redo size
        518  bytes sent via SQL*Net to client
        488  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Beispiel 7: Ergebnis der Abfrage

```
Execution Plan
----------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation              | Name     | Rows  | Cost (%CPU)| Time     | Pstart| Pstop |    TQ  |IN-OUT| PQ Distrib |
----------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |          |     1 |    18   (6)| 00:00:01 |       |       |        |      |            |
|   1 |  SORT AGGREGATE        |          |     1 |            |          |       |       |        |      |            |
|   2 |   PX COORDINATOR       |          |       |            |          |       |       |        |      |            |
|   3 |    PX SEND QC (RANDOM) | :TQ10000 |     1 |            |          |       |       |  Q1,00 | P->S | QC (RAND)  |
|   4 |     SORT AGGREGATE     |          |     1 |            |          |       |       |  Q1,00 | PCWP |            |
|   5 |      PX BLOCK ITERATOR |          |  1423K|    18   (6)| 00:00:01 |     1 |    26 |  Q1,00 | PCWC |            |
|   6 |       TABLE ACCESS FULL| ORDERS   |  1423K|    18   (6)| 00:00:01 |     1 |    26 |  Q1,00 | PCWP |            |
----------------------------------------------------------------------------------------------------------------------------------
Statistics
----------------------------------------------------------
       6296  recursive calls
          4  db block gets
      11241  consistent gets
       8755  physical reads
        672  redo size
        518  bytes sent via SQL*Net to client
        488  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
```

Beispiel 8: Ergebnis der Abfrage ohne Hint

bildung 8). Wie der Output in Abbildung 7 zeigt, wählt der Optimizer nun tatsächlich einen Index Fast Full Scan in paralleler Ausführung als den besten Ausführungsplan. Die Reaktionszeit verkürzt sich auf etwa zwei Sekunden gegenüber 3.21 Sekunden mit FTS und etwa 34 Sekunden bei FTS mit 256 Slaves (als Vergleich zum Ausgangspunkt).

3. Zum Schluss die Optimierung bei großer Auslastung (sechs aktive DVD-Store-Clients, die etwa 1020 Transaktionen pro Sekunde auf einen RAC-Cluster mit zwei Knoten generieren) überprüfen. Diese Abfrage wird bei stark ausgelasteter Datenbank ausgeführt.

Im Vergleich zur Reaktionszeit der Abfrage, die zuvor über zwei Minuten benötigte, haben wir mit vier Sekunden zwar erheblich bessere Ergebnisse, aber im Hinblick auf die Anforderungen an effiziente OLTP-Systeme konnten wir diese Zeitersparnis nicht als akzeptabel betrachten.

Fazit

Dieser Artikel zeigt nur einen geringen Teil der Testergebnisse. Die hier angerissenen Skalierbarkeits- und Performance-Tests wurden weitergeführt und es haben sich noch weitere interessante Aspekte ergeben. Für den gegenseitigen Austausch und weitere Informationen steht der Autor gerne zur Verfügung.

Piotr Sajda
OPITZ CONSULTING GmbH
piotr.sajda@opitz-consulting.com