Der Oracle Cost-Based Optimizer ist eine zentrale Komponente der Datenbank und mittlerweile recht ausgereift. In den seltenen Fällen, in denen er allerdings keinen performanten Ausführungsplan ermitteln kann, empfiehlt Oracle – bei entsprechender Lizenzierung – die Verwendung des SQL Tuning Advisors zur automatischen Ermittlung von Verbesserungspotenzial. Ein DBA sollte allerdings auch ohne diesen Assistenten in der Lage sein, ein SQL-Statement manuell zu analysieren.

Manuelles Oracle SQL-Tuning

Martin Decker, ora-solutions.net

Der Artikel gibt eine Einführung in das manuelle SQL-Tuning anhand eines einfachen 2-Table-Join-Beispiels. Es werden die Entscheidungen des Cost-Based Optimizers nachvollzogen und die wichtigsten Aspekte anhand des Beispiels beleuchtet, etwa Join Method, Join Order, Access Paths, Optimizer Goal, Statistiken und insbesondere Histogramme im Zusammenhang mit Bind-Variablen. Im Abschluss werden die verfügbaren Werkzeuge zur Veränderung eines Ausführungsplans erläutert.

Identifikation des relevanten SQL-Statements

Als erstes gilt es, das relevante, problematische SQL-Statement zu identifizieren, das auf Performance-Optimierung untersucht werden soll. Hierfür gibt es verschiedene Möglichkeiten:

- AWR (Diagnostic Pack License)/ Statspack Report über bestimmtes Intervall
- SQL Tracing (Trace Event 10046 beziehungsweise DBMS_MONITOR)
- Active Session History (erfordert Diagnostic-Pack-Lizenz)
- Oracle Enterprise Manager – DB Performance Page (erfordert Diagnostic-Pack-Lizenz)
- SQL Performance Monitor (erfordert Tuning-Pack-Lizenz)

Ist das SQL-Statement identifiziert, beginnen die eigentliche Untersuchung beziehungsweise die Optimierung. Die Vorgehensweise ist in drei Phasen unterteilt:

- *Analyse-Phase*
  - Ermittlung der Ausführungs-Statistiken (Elapsed Time, Buffer Gets, Disk Reads, Executions) aus V$SQLSTATS beziehungsweise DBA_HIST_SQLSTAT
  - Ermittlung Ausführungsplan (DBMS_XPLAN)
  - optional: Ermittlung Bind-Variablen (v$sql_bind_capture beziehungsweise DBMS_XPLAN)
- *Reproduktions-Phase*
  - Reproduzieren der sub-optimalen Ausführungs-Performance mittels SQL*Plus beziehungsweise SQL Developer
- *Tuning-Phase*
  - Reduzierung der Buffer Gets des Statements durch iterative Veränderung und Messung der Wirkung

In der Analyse-Phase geht es darum, die Performance-Daten der problematischen Ausführung zu ermitteln. Dies sind vor allem „Elapsed Time“, „Buffer Gets“, „Disk Reads“ und „Executions“. Die wichtigste Metrik hierbei sind die „Buffer Gets“. Dieser Begriff definiert die logischen I/O-Operationen, um Datenblöcke aus dem Arbeitsspeicher zu lesen. Während die Metriken „Elapsed Time“ und „Disk Reads“ abhängig von der Caching-Ratio der Datenblöcke stark variieren können, bleibt die Metrik „Buffer Gets“ unabhängig vom Caching stets konstant. In der Praxis kann man feststellen, dass Entwicklungssysteme häufig mit sehr großen Buffer Caches und relativ kleinen Datenmengen betrieben werden. Zudem liegt der Fokus meist nur auf der Antwortzeit der Statements. Damit fällt ein problematischer Zugriffsplan sehr oft erst auf, wenn mit großen Datenmengen im Produktionsumfeld gearbeitet wird. Der DBA wird dann damit konfrontiert, im Produktionsumfeld unter Zeitdruck das Performance-Problem zu analysieren und zu beheben.

Als Richtwert sollte versucht werden, die Anzahl der Buffer Gets pro zurückgelieferter Zeile kleiner als zehn zu halten. Dies gilt nicht für Aggregationen (count, sum, avg etc.) und Joins. Listing 1 zeigt eine Query, die die Belastung der Query für die Datenbank ermittelt. Zudem wird der Ausführungsplan inklusive Bind-Variablen zum Zeitpunkt des Parsing erstellt (siehe Listing 2).

Der Oracle Cost-Based Optimizer (CBO) erstellt zum Zeitpunkt des Parsing den Ausführungsplan. Dieser ist abhängig von einigen Input-Parametern:

- Datenbank-Version, z.B. 11.2.0.1
- Initialisierungsparameter, etwa „optimizer_features_enable=11.1.0.7.0“

```
select sql_fulltext, child_number, plan_hash_value, buffer_gets/
executions, disk_reads/executions, elapsed_time/executions from
v$sqlstats where sql_id = '<sql_id>' and executions > 0;
```

Listing 1

```
select * from table(dbms_xplan.display_cursor('<sql_id>', '<child_
number>', ,TYPICAL +PEEKDED_BINDS'));
```

Listing 2

- Objekt-Statistiken (Table, Column, Index Statistics gesammelt mit „dbms_stats.gather_database|schema|table_stats“)
- System-Statistiken (gesammelt mit „dbms_stats.gather_system_stats“)
- Datenbank-Schema (Tabellenstruktur, vorhandene Indizes etc.)
- Plan-Stability-Informationen (Stored Outlines, ab 10g: SQL Profiles, ab 11*g*: SQL Plan Baselines)
- Cardinality Feedback (ab 11*g* R2: Rück-Übermittlung der Row-Source-Operation-Cardinalities nach Abschluss der Ausführung an den Optimizer)
- aktuelles Datum (wenn beispielsweise Query die Funktion „sysdate“ enthält)

Ändert sich einer dieser Input-Parameter, kann dies dazu führen, dass der Optimizer einen neuen Ausführungsplan erstellt. Ein historisch häufiges Problem hierbei ist die sogenannte „Plan Instability“. Durch Änderung des Ausführungsplans wird ein SQL-Statement zeitweise performant, zu einem anderen Zeitpunkt unperformant ausgeführt. Zur Lösung dieses Problems gibt es mittlerweile mehrere verschiedene Werkzeuge wie Stored Outlines (Standard Edition), SQL Profiles (Enterprise Edition) oder SQL Plan Baselines (Enterprise Edition).

Anschließend wird in der Reproduktions-Phase versucht, die problematische Ausführungsperformance zu reproduzieren. Falls das Statement Bind-Variablen beinhaltet, dürfen diese vorerst nicht durch Literale ersetzt werden. Falls die Bind-Variablen vom Datentyp „DATE“ sind, empfiehlt es sich, das Statement mit SQL Developer statt SQL*Plus zu reproduzieren.

Unter Umständen ist das Problem allerdings nicht reproduzierbar. Teilweise ist die Daten-Konstellation, die zum Problem führt, nur kurzzeitig vorhanden. Später, wenn der DBA die Analyse durchführt, ist das Problem aufgrund der anderen Datenkonstellation nicht mehr sichtbar. In diesen Fällen muss auf SQL Tracing mittels „Trace Event 10046“ oder „DBMS_MONITOR“ ausgewichen werden.

Ist es gelungen, denselben problematischen Zugriffsplan beziehungsweise die problematische Ausführungsdauer zu generieren, so wird nun in der Tuning-Phase begonnen, die Ausführungsdauer und den Ressourcen-Verbrauch des Statements durch verschiedenste Maßnahmen zu reduzieren. Im Gegensatz zum Instance Tuning wird nicht versucht, die Ausführungszeit durch Vergrößerung von Caches und Reduzierung von Disk Reads zu verringern. Der Fokus liegt hier auf der Reduzierung der Logical I/Os (Buffer Gets) der SQL Query. Die einzelnen Maßnahmen sind mannigfaltig und abhängig vom Statement. Beispiele hierfür sind: Erstellung eines Index, Aktualisierung der Optimizer-Statistiken, Optimizer Hints, Aktivierung einer bestimmten Optimizer-Version, Deaktivierung von bestimmten Optimizer-Features, Ändern der Datenverteilung durch Reorganisation und Sortierung, Rewrite des SQL-Statements etc. Das folgende Beispiel verwendet die beiden Tabellen „CUSTOMERS“ und „ORDERS“ (siehe Abbildung 1). Die Datenverteilung hat folgendes Profil:

- Customers: 100.000 rows
- Orders: 1.000.000 rows
  - CUSTOMER_ID: 70 Prozent von einem einzelnen Customer, 20 Prozent von einem einzelnen anderen Customer, die restlichen 10 Prozent von ca. 64.000 verschiedenen Customern
  - ORDER_STATUS: zwei verschiedene Order-Stati, ungleichmäßig verteilt: 900.000 COMPLETED, 100.000 PENDING

Listing 3 zeigt eine SQL Query, die nun im Detail analysiert wird.

Step 1: Analyse-Phase

Bei der Analyse kann festgestellt werden, dass das Statement 0,353 Sekunden pro Ausführung dauert und 22.300 Buffer Gets dafür aufgewendet werden müssen. Der Ausführungsplan zeigt einen Nested Loop Join und als Access Paths werden Index Lookups verwendet. Das Statement enthält eine Bind-Variable, die auf eine bestimmte

```
SELECT * FROM DEMO.CUSTOMERS C,
              DEMO.ORDERS O
WHERE O.CUSTOMER_ID = C.CUSTOMER_ID        -- join predicate
      AND C.CUSTOMER_ID = :v1              -- filter predicate
      and O.ORDER_STATUS = 'PENDING'       -- filter predicate
ORDER BY O.ORDER_DATE                      -- sorting
;
```

Listing 3

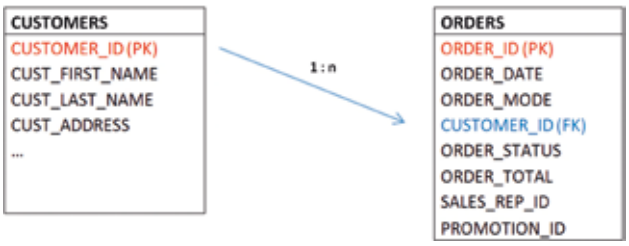

Abbildung 1: Vereinfachtes ER-Diagramm der Beispiel-Tabellen

```
SQL> select child_number as child, plan_hash_value as phv, buffer_
gets/executions as gets_per_exe, disk_reads/executions as disk_per_
exe, elapsed_time/executions ela_per_exe, executions as exe from v$sql
where sql_id = 'g7c03r8pn1ymq';
     CHILD        PHV GETS_PER_EXE DISK_PER_EXE ELA_PER_EXE        EXE
---------- ---------- ------------ ------------ ------------ ----------
         0 3311696933        22300            0  353112.938         81
```

Abbildung 2: Ressourcen-Verbrauch

```
SQL> select * from table(dbms_xplan.display_cursor('g7c03r8pn1ymq','0','TYPICAL +PEEKED_BINDS'));
Plan hash value: 3311696933
-------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |              |       |       |     8 (100)|          |
|   1 |  SORT ORDER BY                |              |     8 |  1560 |     8  (13)| 00:00:01 |
|   2 |   NESTED LOOPS                |              |     8 |  1560 |     7   (0)| 00:00:01 |
|   3 |    TABLE ACCESS BY INDEX ROWID| CUSTOMERS    |     1 |   150 |     2   (0)| 00:00:01 |
|*  4 |     INDEX UNIQUE SCAN         | CUSTOMERS_PK |     1 |       |     1   (0)| 00:00:01 |
|*  5 |    TABLE ACCESS BY INDEX ROWID| ORDERS       |     8 |   360 |     5   (0)| 00:00:01 |
|*  6 |     INDEX RANGE SCAN          | CUSTOMER_IDX |    16 |       |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------
Peeked Binds (identified by position):
--------------------------------------
   1 - :V1 (NUMBER): 1001
Predicate Information (identified by operation id):
PLAN_TABLE_OUTPUT
---------------------------------------------------
   4 - access("C"."CUSTOMER_ID"=:V1)
   5 - filter("O"."ORDER_STATUS"='PENDING')
   6 - access("O"."CUSTOMER_ID"=:V1)
```

Abbildung 3: Ressourcen-Verbrauch und Ausführungsplan

CUSTOMER_ID filtert. Damit ist klar, dass die Query nur eine einzelne Zeile aus der Tabelle „CUSTOMERS“ treffen wird, da die Spalte „CUSTOMER_ID“ als Primary Key definiert ist. Für diesen Kunden werden dann alle offenen Bestellungen abgefragt und nach dem Bestelldatum (ORDER_DATE) sortiert. Beim Ausführungsplan ist zu sehen, welche Bind-Variable beim Parsing vorhanden war. (hier: 1001).

Exkurs: Binds vs. Literals

Noch ein Gedanke zur Query: Immer wieder gibt es Konflikte zwischen DBAs und Entwicklern über die Verwendung von Bind-Variablen. Deren Zweck besteht darin, den Parsing-Overhead zu reduzieren und möglichst viele Statements im Shared Pool zu cachen und wiederzuverwenden. Die Query enthält eine Bind-Variable für die CUSTOMER_ID und ein Literal-String für den ORDER_STATUS. Wenn statt der Bind-Variablen für die CUSTOMER_ID ein Literal verwendet worden wäre, dann hätte das die Konsequenz, dass für jede CUSTOMER_ID (potenziell 100.000) ein einzelnes, unabhängiges SQL-Statement verwendet wird. Der Effekt des Caching von SQL-Statements und deren Metadaten wäre damit nicht möglich.

Bei der Spalte „ORDER_STATUS“ gibt es nur zwei verschiedene Werte (NDV), die zudem ungleich verteilt sind: „COMPLETED“ und „PENDING“. Deshalb ist dieses Prädikat ein idealer Kandidat für ein Literal. Im schlimmsten Fall gibt es dadurch zwei verschiedene SQL-Statements, die bis auf den Literal-Wert identisch sind. Hingegen gibt es bei Verwendung des Literals die Möglichkeit, ein Histogramm auf die Spalte zu berechnen und damit dem Optimizer die genaue Verteilung der Rows auf die beiden Werte mitzuteilen. Die Abbildungen 2 und 3 zeigen die Belastung der Datenbank durch die Query.

Die Spalte „Rows“ definiert die „Cardinality“, also die geschätzte Anzahl der Zeilen, die mit dieser Operation zurückgeliefert werden. Es ist zu beachten, dass alle Informationen des Ausführungsplans lediglich Schätzungen des Optimizer zum Zeitpunkt des Parsing sind und es nicht ersichtlich ist, an welcher Zeile (Row Source Operation) des Ausführungsplans der Großteil der Buffer Gets aufgewendet wird. Für die weitere Analyse ist deshalb diese Darstellung unzureichend.

Step 2: Reproduktions-Phase

Bei der Reproduktion wird nun der Optimizer Hint „GATHER_PLAN_STATISTICS“ eingefügt. Direkt nach dem eigentlichen SQL-Statement wird dann der Ausführungsplan mittels „DBMS_XPLAN“ abgefragt, wobei der Format-Parameter auf „ALLSTATS LAST“ gesetzt wird. Um den eigentlichen Query-Output auszublenden, kann „termout off“ verwendet werden. Dabei ist die Definition der Bind-Variablen in SQL*Plus-Syntax zu beachten. Beim Datentyp ist zu bedenken, dass derselbe Datentyp verwendet wird, wie vorher im Ausführungsplan angezeigt wurde (siehe Listing 4).

Abbildung 4 zeigt den Ausführungsplan, nachdem diese SQL-Datei mit SQL*Plus ausgeführt wurde. Man sieht nun einen wesentlich detaillierteren Ausführungsplan mit zusätzlichen beziehungsweise anderen Spalten. Die Spalte „STARTS“ gibt an, wie oft eine „Row Source Operation“ ausgeführt wurde. Dies ist vorwiegend bei Nested Loop Joins relevant. Die Spalten „E-(estimated) Rows“ und „A-(actual) Rows“ geben Aufschluss darüber, wie akkurat die Cardinality-Schätzung des Optimizer war. In unserem Fall gibt es eine starke Abweichung.

Die Spalte „A-(actual)Time“ zeigt die Ausführungsdauer. Die Spalte „Buffers“ gibt an, wie viele logische I/O-Operationen pro Row-Source-Opera-

```
-----------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name         | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |              |      1 |        |      1 |00:00:00.58 |   22300 |
|   1 |  SORT ORDER BY                |              |      1 |      8 |      1 |00:00:00.58 |   22300 |
|   2 |   NESTED LOOPS                |              |      1 |      8 |      1 |00:00:00.58 |   22300 |
|   3 |    TABLE ACCESS BY INDEX ROWID| CUSTOMERS    |      1 |      1 |      1 |00:00:00.01 |       3 |
|*  4 |     INDEX UNIQUE SCAN         | CUSTOMERS_PK |      1 |      1 |      1 |00:00:00.01 |       2 |
|*  5 |    TABLE ACCESS BY INDEX ROWID| ORDERS       |      1 |      8 |      1 |00:00:00.57 |   22297 |
|*  6 |     INDEX RANGE SCAN          | CUSTOMER_IDX |      1 |     16 |    700K|00:00:11.72 |    1467 |
-----------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access("C"."CUSTOMER_ID"=:V1)
   5 - filter("O"."ORDER_STATUS"='PENDING')
```

Abbildung 4: Reproduziertes SQL-Statement und erweiterter Ausführungsplan

```
set termout off
variable v1 number;
exec :v1 := 1001;
select /*+ GATHER_PLAN_STATISTICS */ * from DEMO.CUSTOMERS C,
             DEMO.ORDERS O
WHERE O.CUSTOMER_ID = C.CUSTOMER_ID -- join predicate
      AND C.customer_id = :v1 -- filter predicate
      and O.order_status = ‚PENDING' --filter predicate
order by order_date -- sorting
;
set termout on
set lines 300
set pages 1000
select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS
LAST'));
```

Listing 4

tion notwendig waren. Man sieht, dass der Großteil davon auf die Zeile 5 fällt. Die Zahlen in der Spalte „Buffers" sind kumulativ, abhängig davon, ob es eine untergeordnete Row-Source-Operation gibt. In unserem Beispiel beinhaltet die Zahl „Buffers=3" in Zeile 3 bereits die Zahl „Buffers=2" der Zeile 4. Ebenso enthält die Zahl „22297" aus der Zeile 5 bereits die 1467 Buffer Gets aus Zeile 6. In der Row-Source-Operation der Zeile 2 (NESTED LOOPS) enthält die Zahl „22300" bereits die beiden Nested Loop Join Childs 3 und 5 (beziehungsweise Buffers 3 und 22297).

Der Optimizer hat nun folgende Entscheidungen getroffen:

- *Join Order*
  Aufgrund der Tatsache, dass mit einem Equal-Prädikat auf den Primary-Key zugegriffen wird, weiß der Optimizer, dass die Tabelle „Customer" nur eine Zeile zurückliefern wird. Deshalb ist diese Tabelle diejenige, mit der begonnen wird. Grundsätzlich ist das Ziel, bei der Ausführung möglichst früh möglichst viele Rows (durch Filter etc.) zu eliminieren.
- *Access Paths*
  Der Optimizer weiß, dass auf der Spalte „CUSTOMERS.CUSTOMER_ID" ein Unique-Index liegt und dies der effizienteste Zugriffspfad zur Tabelle ist. Bei der Tabelle „ORDERS" gibt es zwei Prädikate: erstens die Einschränkung auf „ORDER_STATUS" und zweitens die auf „ORDERS.CUSTOMER_ID". Der Optimizer weiß anhand der Column-Statistics (DBA_TAB_COL_STATISTICS), dass die Anzahl der verschiedenen Werte (NDV, Number of Distinct Values) für „ORDERS.CUSTOMER_ID" 64236 beträgt.
  Die Selektivität wird berechnet mit der Formel: „1/NDV", das heißt „1/64236=0,0000155". Nun wird die Anzahl der Rows der Tabelle (1.000.000) mit der Selektivität multipliziert: „1.000.000*0,0000155=15,5". Der Optimizer rundet das auf 16 und schätzt, dass der Kunde 16 Bestellungen hat. Aufgrund der geringen Selektivität entscheidet sich der Optimizer gegen einen Full Table Scan der Tabelle „Orders" (1 Million Rows) und für einen Index Range Scan auf den Foreign Key Index auf der Spalte „CUSTOMER_ID".
- *Join Method*
  Zur Auswahl stehen Nested Loop, Hash Join, Sort Merge Join und Merge Join Cartesian. Der Optimizer wählt den Nested Loop Join. Dieser besteht aus einer äußeren Schleife, dem Outer Loop (Zeile 3-4), und einem Inner Loop (Zeile 5-6). Für jede zutreffende Zeile des Outer Loop wird der Inner Loop einmal ausgeführt. Die Anzahl der Wiederholungen wird in der Spalte „Starts" angezeigt. Aufgrund der Tatsache, dass der Outer Loop nur eine Zeile (einen Kunden) zurückliefert, muss der Inner Loop nur einmal ausgeführt werden.

Step 3: Tuning-Phase

Die Vorgehensweise beim Tuning ist abhängig vom SQL-Statement. Die folgenden Aspekte werden unter anderem berücksichtigt:

- Welcher Teil des Ausführungsplans die meisten Buffer Gets verursacht

```
-------------------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                     |      1 |        |      1 |00:00:00.40 |    6332 |
|   1 |  SORT ORDER BY                |                     |      1 |      1 |      1 |00:00:00.40 |    6332 |
|   2 |   NESTED LOOPS                |                     |      1 |      1 |      1 |00:00:00.40 |    6332 |
|   3 |    TABLE ACCESS BY INDEX ROWID| CUSTOMERS           |      1 |      1 |      1 |00:00:00.01 |       3 |
|*  4 |     INDEX UNIQUE SCAN         | CUSTOMERS_PK        |      1 |      1 |      1 |00:00:00.01 |       2 |
|*  5 |    TABLE ACCESS BY INDEX ROWID| ORDERS_SORTED       |      1 |      1 |      1 |00:00:00.40 |    6329 |
|*  6 |     INDEX RANGE SCAN          | ORDERS_SORTED_CUST  |      1 |      2 |    700K|00:00:08.14 |    1467 |
-------------------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access("C"."CUSTOMER_ID"=:V1)
   5 - filter("O"."ORDER_STATUS"='PENDING')
   6 - access("O"."CUSTOMER_ID"=:V1)
```

Abbildung 5: Ressourcen-Verbrauch und erweiterter Ausführungsplan nach Table-Reorganisation

und daher am meisten Potenzial für Verbesserung hat

- Ob es bei den Row Source Operations der Access-Paths eine starke Abweichung zwischen E-(estimated) Rows und A-(actual) Rows gibt
- Ob im Ausführungsplan ein Nested Loop Join existiert, bei dessen Inner Loop ein ineffizienter Access-Path gewählt wurde, etwa Full Table Scan oder Index Fast Full Scan
- Ob bei Betrachtung der Spalte „A-Rows“ möglichst früh Zeilen im Ausführungsplan gefiltert werden. Wenn ja, könnte die Join Order, speziell die Auswahl der Start-Tabelle, suboptimal sein. Ob es unrealistisch hohe A-Rows-Werte in bestimmten Row Source Operations gibt. Dies würde auf suboptimale Joins oder kartesische Produkte hinweisen.

Idee 1: Verbesserung der Cardinality-Schätzung durch Histogramme

Offensichtlich lag der Optimizer bei der Schätzung der Cardinality beim Hauptkunden etwas daneben. Statt der geschätzten 16 Rows werden hier 700.000 Rows geliefert. Das sind 70 Prozent der gesamten Zeilen der Tabelle „ORDERS“. Die Bestell-Daten sind also in Bezug auf Kunden ungleich verteilt (engl.: skewed). Oracle bietet hierfür sogenannte „Histogramme“ an. Diese geben dem Optimizer die Möglichkeit, genauere Daten über die ungleiche Verteilung zu liefern. Allerdings gibt es ein Problem bei Verwendung von Histogrammen und Bind-Variablen. In unserem Fall wird dieselbe Query für häufige und für seltene Kunden benutzt und nur CUSTOMER_ID als Wert der Bind-Variablen wird angepasst. Oracle wird dafür also einen einzigen Cursor generieren, der für alle verschiedenen CUSTOMER_IDs verwendet wird. Falls wir also ein Histogramm erzeugen würden, bestände das Risiko, dass Oracle denselben Plan, der für den Großkunden verwendet wird (FULL TABLE SCAN auf ORDERS), auch für den seltenen Kunden anwendet. Hier wäre aber ein Index-basierter Access Path zu bevorzugen. Wir verwerfen deshalb die Idee mit dem Histogramm und überlegen uns eine Alternative.

Idee 2: Erhöhung des Clustering durch Sortierung der ORDER-Daten nach CUSTOMER_ID

Der Index-Zugriff von Zeile 6 liefert also 700.000 ROWIDs mit der gewünschten CUSTOMER_ID. Die ROWID ist das physische Identifikationsmerkmal eines Datensatzes der Tabelle und besteht aus „File“, „Block“ und „Row in Block“. In Zeile 5 wird also jede ROWID aufgelöst und gelesen. Die Bestelldaten für die einzelnen Kunden sind unsortiert, die Rows für den Kunden „1001“ sind also quer über alle Tabellen-Blöcke verteilt. Wenn man nun die Tabelle „ORDERS“ reorganisieren und nach „CUSTOMER_ID“ sortiert speichern würde, müsste man sehr wahrscheinlich weniger Tabellen-Datenblöcke lesen. Als Richtwert kann man die Anzahl der Rows (700.000) mit der Anzahl der Buffers (22297 minus 1467=20830) ins Verhältnis setzen: „700.000/20830“. Am effizientesten ist diese Reorganisation, wenn die Anzahl der Blöcke sehr hoch im Verhältnis zu der Rows ist. Ein Test zeigt, dass damit die Anzahl der Buffer Gets von 20830 auf 4862 reduziert werden kann (siehe Abbildung 5). Nachteil ist, dass die Reorganisation nicht nur einmalig, sondern regelmäßig durchgeführt werden muss. Es ist also in diesem Fall nicht die optimale Lösung. Zudem ist der Ressourcen-Verbrauch mit über 6000 Buffer Gets für eine einzelne zurückgelieferte Zeile noch deutlich zu hoch. Wir verwerfen deshalb auch diese Idee und testen eine dritte.

Idee 3: Erhöhung der Selektivität durch Index auf ORDERS.ORDER_STATUS

Es wird davon ausgegangen, dass die meisten Bestellungen der Tabelle „ORDERS“ im Status „COMPLETED“ sind und nur recht wenige im Status „PENDING“. Beim Nested Loop Join findet der Einstieg in die Tabelle „ORDERS“ über die Spalte „CUSTOMER_ID“ statt. Es wird ein Index angelegt, der neben „CUSTOMER_ID“ auch die Spalte „ORDER_STATUS“ enthält. Da es nur zwei verschiedene Stati gibt und die Zeilen im Index sortiert gespeichert werden, kann Index Compression für die erste Spalte aktiviert werden und sich wiederholende Status-Strings können vermieden werden. Dies führt zu einer Reduzierung der Index-Leaf-Blocks (siehe Listing 5). Abbildung 6 zeigt den Status nach erneuter Ausführung der Query.

```
-----------------------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name                | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                     |      1 |        |      1 |00:00:00.01 |       7 |
|   1 |  SORT ORDER BY               |                     |      1 |     16 |      1 |00:00:00.01 |       7 |
|   2 |   NESTED LOOPS               |                     |      1 |     16 |      1 |00:00:00.01 |       7 |
|   3 |    TABLE ACCESS BY INDEX ROWID| CUSTOMERS          |      1 |      1 |      1 |00:00:00.01 |       3 |
|*  4 |     INDEX UNIQUE SCAN        | CUSTOMERS_PK        |      1 |      1 |      1 |00:00:00.01 |       2 |
|   5 |    TABLE ACCESS BY INDEX ROWID| ORDERS             |      1 |     16 |      1 |00:00:00.01 |       4 |
|*  6 |     INDEX RANGE SCAN         | ORDERS_CUST_STATUS  |      1 |     16 |      1 |00:00:00.01 |       3 |
-----------------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access("C"."CUSTOMER_ID"=:V1)
   6 - access("O"."ORDER_STATUS"='PENDING' AND "O"."CUSTOMER_ID"=:V1)
```

Abbildung 6: Ressourcen-Verbrauch und erweiterter Ausführungsplan nach Index-Erstellung

```
-----------------------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name                     | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                          |      1 |        |      1 |00:00:00.01 |       8 |
|   1 |  NESTED LOOPS                |                          |      1 |      8 |      1 |00:00:00.01 |       8 |
|   2 |   TABLE ACCESS BY INDEX ROWID| CUSTOMERS                |      1 |      1 |      1 |00:00:00.01 |       3 |
|*  3 |    INDEX UNIQUE SCAN         | CUSTOMERS_PK             |      1 |      1 |      1 |00:00:00.01 |       2 |
|   4 |   TABLE ACCESS BY INDEX ROWID| ORDERS                   |      1 |      8 |      1 |00:00:00.01 |       5 |
|*  5 |    INDEX RANGE SCAN          | ORDERS_CUST_STATUS_DATE  |      1 |      8 |      1 |00:00:00.01 |       4 |
-----------------------------------------------------------------------------------------------------------------
```

Abbildung 7: Ressourcen-Verbrauch und erweiterter Ausführungsplan nach Index-Erstellung zur Vermeidung der Sort-Operation

Die Maßnahme war erfolgreich und wir konnten somit die Anzahl der Buffer Gets von 22.300 auf 7 reduzieren. Dieser Ausführungsplan ist nun sowohl für den Kunden mit 700.000 Bestellungen als auch für den Kunden mit wenigen Bestellungen optimal.

Idee 4: Vermeidung der Sortierung durch Erweiterung des Index

Im obigen Beispiel wird nur eine Zeile zurückgeliefert. Die Sortierung ist deshalb nicht aufwändig. Falls die Anzahl der Rows allerdings relativ hoch wäre, könnte man durch Erweiterung des Index die Sortierung eliminieren (siehe Listing 6). Abbildung 7 zeigt den Ausführungsplan, der die Vermeidung der Sort-Operation (SORT ORDER BY) bestätigt.

Hinweis 1: Optimizer Goal

Der Optimizer unterstützt verschiedene Optimierungen, abhängig davon, ob die Gesamtausführungszeit des Statements (ALL_ROWS) oder die Antwortzeit, bis die ersten X Rows (FIRST_ROWS_n) geliefert werden, optimiert wird. Bei der vorherigen Technik zur Vermeidung einer Sortierung hat man auch gleichzeitig den „FIRST_ROWS_n"-Zugriff optimiert. Die Resultate können direkt von der Row-Source-Operation „NESTED LOOPS" zeilenweise an die Anwendung beziehungsweise den Benutzer zurückgeliefert werden. Er kann somit sehr schnell mit den ersten Ergebnissen arbeiten. Bei einer Query mit der Row-Source-Operation „SORT ORDER BY" muss zuerst die gesamte Ergebnismenge ermittelt und sortiert werden, bevor die erste Zeile an den Anwender zurückgeliefert werden kann.

```
create index DEMO.ORDERS_CUST_
STATUS on DEMO.ORDERS(ORDER_
STATUS,CUSTOMER_ID) COMPRESS 1;
```

Listing 5

```
drop index DEMO.ORDERS_CUST_
STATUS;
create index DEMO.ORDERS_
CUST_STATUS_DATE on DEMO.
ORDERS(ORDER_STATUS,CUSTOMER_
ID,ORDER_DATE)
COMPRESS 1;
```

Listing 6

Hinweis 2: Fixierung des Zugriffsplans – Plan Instability

Bei Problemen mit „Plan Instability" kann es notwendig sein, den optimalen Zugriffsplan zu fixieren. Der Autor bevorzugt dafür die Methode, ein SQL-Profil mittels „DBMS_SQLTUNE.IMPORT_SQL_PROFILE" zu setzen. Dabei wird in der Tuning-Phase der Zugriffsplan durch Hints bewusst verändert und anschließend werden die Ausführungsplan-Direktiven (Outlines) dem produktiven Statement zugewiesen. Bei Verwendung der Standard Edition ist ein ähnliches Vorgehen mithilfe von Stored Outlines möglich.

Martin Decker
martin.decker
@ora-solutions.net