Manuelles Oracle SQL Tuning

Martin Decker
ora-solutions.net
D/A/CH

Schlüsselworte
Optimizer, SQL Tuning, Performance, Explain Plan, Hints,

Einleitung
Der Oracle Cost-Based Optimizer ist die 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 Verbesserungspotential. Der DBA sollte allerdings auch ohne diesen Assistenten in der Lage sein, das SQL Statement manuell zu analysieren. Der Vortrag 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, z.B. 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 bzw. DBMS_MONITOR)
- Active Session History (Diagnostic Pack License)
- Oracle Enterprise Manager – DB Performance Page (Diagnostic Pack License)
- SQL Performance Monitor (Tuning Pack License)

Die Vorgehensweise zur Identifikation des relevanten Statements wird hier nicht beschrieben. Ist das SQL Statement identifiziert kann mit der eigentlichen Untersuchung bzw. der Optimierung begonnen werden.

Ablauf-Phasen
Die Vorgehensweise kann in 3 Phasen unterteilt werden:

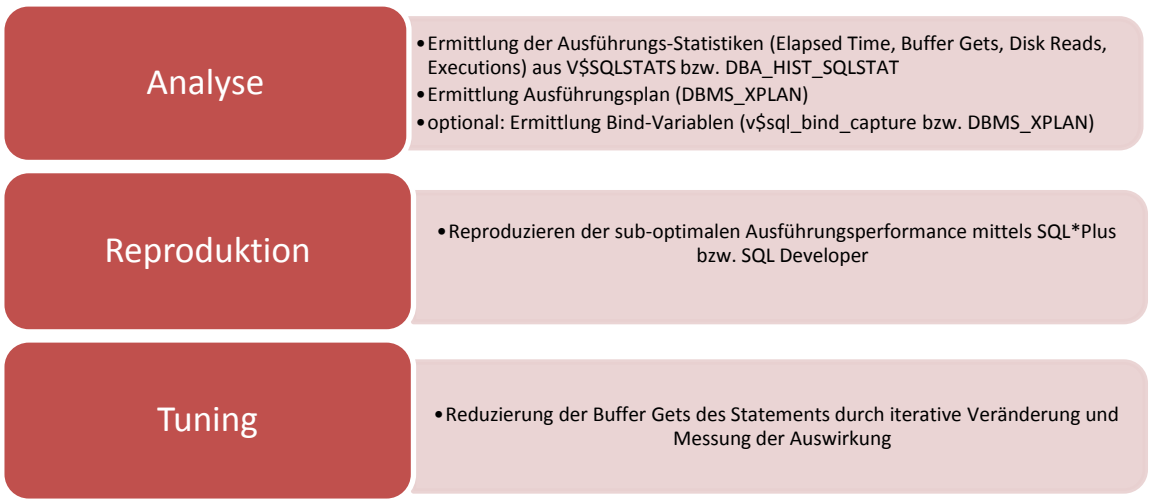


Abb.1: Ablauf-Phasen beim manuellen Tuning

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 festgestellt werden, 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 < 10 zu halten. Dies gilt nicht für Aggregationen (z.B. count, sum, avg, etc.) und Joins. Mit folgender Query wird die Belastung der Query für die Datenbank ermitelt:

```
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;
```

Zudem wird der Ausführungsplan inklusive Bind-Variablen zum Zeitpunkt des Parsings ermittelt:

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

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, z.B. optimizer_features_enable=11.1.0.7.0
- 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 Stablity Informationen (Stored Outlines, ab 10g: SQL Profiles, ab 11g: SQL Plan Baselines)
- Cardinality Feedback (ab 11gR2: Rück-Übermittlung der Row-Source-Operation Cardinalities nach Abschluss der Ausführung an den Optimizer)
- aktuelles Datum (z.B. wenn 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 „Plan Instability". D.h., durch Änderung des Ausführungsplans wird das selbe SQL Statement zeitweise performant, zu einem anderen Zeitpunkt unperformant ausgeführt. Zur Lösung dieses Problems gibt es mittlerweile mehrere verschiedene Werkzeuge, z.B. Stored Outlines (Standard Edition), SQL Profiles (Enterprise Edition), 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

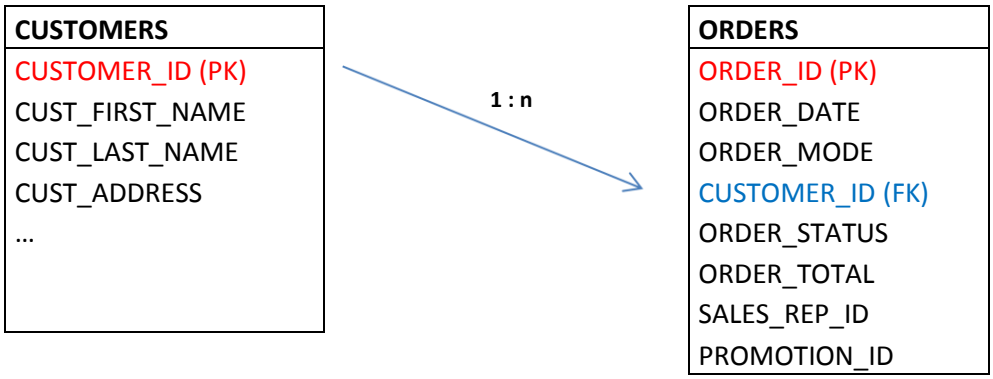
die Daten-Konstellation, die zum Problem führt, nur kurzzeitig vorhanden. Später, wenn der DBA die Analyse durchführt, ist das Problem aufgrund der anderen Datenkonstellation nicht mehr sichtbar. In diesen Fällen muss auf SQL Tracing mittels Trace Event 10046 oder DBMS_MONITOR ausgewichen werden.

Ist es gelungen, den selben problematischen Zugriffsplan bzw. die problematische Ausführungsdauer zu generieren so kann nun in der **Tuning-Phase** begonnen werden, die Ausführungsdauer und den Ressourcen-Verbrauch des Statements durch verschiedenste Maßnahmen zu reduzieren. Im Gegensatz zum Instance Tuning wird nicht versucht, durch Vergrößerung von Caches und Reduzierung von Disk Reads die Ausführungszeit zu verringern. Der Fokus liegt hier bei 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 eines bestimmten Optimizer-Version, Deaktivierung von bestimmten Optimizer-Features, Ändern der Datenverteilung durch Reorganisation und Sortierung, Rewrite des SQL Statements, etc.

Beispiel

Das folgende Beispiel verwendet die beiden Tabellen CUSTOMERS und ORDERS.



Datenverteilung:

- Customers: 100.000 rows,
- Orders: 1.000.000 rows,
    - CUSTOMER_ID: 70% von einem einzelnen Customer, 20% von einem einzelnen anderen Customer, die restlichen 10% von 64.234 verschiedenen Customer,
    - ORDER_STATUS: 2 verschiedene Order-Stati, ungleichmäßig verteilt: 900.000 COMPLETED, 100.000 PENDING

Die folgende SQL Query wird nun im Detail analysiert:

```
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
;
```

Step 1: Analyse:

Bei der Analyse kann festgestellt werden, dass das Statement 0,353 Sekunden pro Ausführung dauert und 22300 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 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 wurde. 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. Der Zweck von Bind-Variablen besteht darin, den Parsing-Overhead zu reduzieren und möglichst viele Statements im Shared Pool cachen und wiederverwenden zu können. 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 verschiedene CUSTOMER_ID (potentiell 100.000) ein einzelnes unabhängiges SQL Statement verwendet wird. Der Effekt des Cachings von SQL Statements und dessen 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.

Mit dem folgenden Statement wird nun die Belastung der Datenbank durch die Query ermittelt:

```
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

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)
```

Die Spalte „Rows“ definiert die „Cardinality“, d.h. die geschätzte Anzahl der Zeilen die mit dieser Operation zurückgeliefert wird. Bitte beachten Sie, dass alle Informationen des gezeigten Ausführungsplans lediglich Schätzungen des Optimizers zum Zeitpunkt des Parsings sind! Es ist hier nicht ersichtlich, an welcher Zeile (Row Source Operation) des Ausführungsplans der Großteil der Buffer Gets aufgewendet wird. Für die weitere Analyse ist deshalb diese Darstellung des Ausführungsplans unzureichend.

Step 2: Reproduktion

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. Beachten Sie ebenfalls die Definition der Bind Variable in SQL*Plus Syntax. Beim Datentyp ist darauf zu achten, dass der selbe Datentyp verwendet wird, wie vorher im Ausführungsplan angezeigt wurde.

```
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'));

Plan hash value: 3311696933

-----------------------------------------------------------------------------------------------------------------
| 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')
```

```
6 - access("O"."CUSTOMER_ID"=:V1)
```

Sie sehen nun einen wesentlich detaillierteren Ausführungsplan mit zusätzlichen bzw. 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 Optimizers 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 Operation notwendig waren. Man sieht, dass der Großteil davon auf die Zeile 5 fällt. Beachten Sie, dass die Zahlen in der Spalte Buffers kumulativ sind, abhängig, 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 (bzw. Buffers 3 und 22297).

Welche Entscheidungen hat der Optimizer 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, möglichst früh bei der Ausführung möglichst viele Rows (durch Filter, etc.) zu eliminieren.
- **Access Path:** Der Optimizer weiß, das 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 Einschränkung auf ORDERS.CUSTOMER_ID. Der Optimizer weiß anhand der Column-Statistics (DBA_TAB_COL_STATISTICS), dass die Anzahl der verschiedenen Werte (NDV, **N**umber of **D**istinct **V**alues) für ORDERS.CUSTOMER_ID 64236 beträgt. Die Selektivität wird berechnet mit der Formel: 1 / NDV, d.h. 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 MIO 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 Join besteht aus einer äußeren Schleife, dem Outer Loop (Zeile 3-4) und einem Inner Loop (Zeile 5-6). Für jede zutreffende Zeile des Outer-Loops 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

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

- Welcher Teil des Ausführungsplans verursacht die meisten Buffer Gets und hat daher am meisten Potential für Verbesserung?
- Gibt es bei den Row Source Operations der Access-Paths eine starke Abweichung zwischen E-(estimated) Rows und A-(actual) Rows?
- Existiert ein Nested-Loop Join im Ausführungsplan, bei dessen Inner Loop ein ineffizienter Access-Path gewählt wurde, z.B. Full Table Scan oder Index Fast Full Scan?
- Bei Betrachtung der Spalte „A-Rows“: werden Zeilen möglichst früh im Ausführungsplan gefiltert? Wenn ja, ist die Join Order, speziell die Auswahl der Start-Tabelle suboptimal? Gibt es unrealistisch

hohe A-Rows Werte in bestimmten Row Source Operations? Dies würde auf suboptimale Joins oder kartesische Produkte hinweisen.

Idee 1: Verbesserung der Cardinality-Schätzung durch Histogramm
Offensichtlich lag der Optimizer bei der Schätzung der Cardinality bei unserem Hauptkunden etwas daneben. Statt der geschätzten 16 Rows werden hier 700.000 Rows geliefert. Das sind 70% der gesamten Zeilen der Tabelle ORDERS. Die Bestell-Daten sind also im Bezug auf Kunden ungleich verteilt (engl.: skewed). Oracle bietet hierfür „Histogramme". Histogramme 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 die selbe Query für häufige und 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_ID verwendet wird. Falls wir also ein Histogramm erzeugen würden, besteht das Risiko, dass Oracle den selben Plan, der für den Großkunden verwendet wird (FULL TABLE SCAN auf ORDERS), auch für den seltenen Kunden angewendet wird. Hier wäre aber ein Index-basierter Access Path zu bevorzugen. Wir verwerfen deshalb die Idee mit dem Histogramm und überlegen uns eine andere 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 / Row in Block*. In Zeile 5 wird also jede ROWID aufgelöst und gelesen. Die Bestelldaten für die einzelnen Kunden sind unsortiert, d.h. die Rows für den Kunden 1001 sind quer über alle Tabellen-Blöcke verteilt. Wenn man nun die Tabelle ORDERS reorganisieren und nach CUSTOMER_ID sortiert speichern würde, dann 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 ist im Verhältnis zu den Rows. Ein Test zeigt, dass damit die Anzahl der Buffer Gets von 20830 auf 4862 reduziert werden kann.

```
-------------------------------------------------------------------------------------------------------------------
| 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)
```

Der Nachteil dieser Methode besteht darin, dass die Reorganisation nicht nur einmalig, sondern regelmäßig durchgeführt werden müsste. Es ist also in diesem Fall nicht die optimale Lösung. Zudem ist der Ressourcenverbrauch mit über 6000 Buffer Gets für eine einzelne zurückgelieferte Zeile noch deutlich zu hoch. Wir verwerfen also auch diese Idee und testen eine dritte Idee.

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.

```
create index DEMO.ORDERS_CUST_STATUS on DEMO.ORDERS(ORDER_STATUS,CUSTOMER_ID)
COMPRESS 1;
```

Dann wird die Query erneut ausgeführt:

```
-----------------------------------------------------------------------------------------------------------------
| 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)
```

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 700.00 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.

```
drop index DEMO.ORDERS_CUST_STATUS;
create index DEMO.ORDERS_CUST_STATUS_DATE on
DEMO.ORDERS(ORDER_STATUS,CUSTOMER_ID,ORDER_DATE)
COMPRESS 1;
```

Anschließend bestätigt der Ausführungsplan die Vermeidung der Sort-Operation: (SORT ORDER BY)

```
Plan hash value: 2123382235

-----------------------------------------------------------------------------------------------------------------
| 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 |
-----------------------------------------------------------------------------------------------------------------
```

Hinweis 1: Optimizer Goal

Der Optimizer unterstützt verschiedene Optimierungen, die 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 bzw. 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.

Hinweis 2: Fixierung des Zugriffsplans – Plan Instability
Bei Problemen mit Plan Instability kann es notwendig sein, den optimalen Zugriffsplan zu fixieren. Persönlich bevorzuge ich dafür die Methode, ein SQL Profil mittels DBMS_SQLTUNE.IMPORT_SQL_PROFILE zu setzen. Dabei wird in der Tuning-Phase mittels Hints der Zugriffsplan bewusst verändert und anschließend werden die Ausführungsplan-Direktiven (Outlines) dem produktiven Statement zugewiesen. Bei Verwendung der Standard Edition ist ein ähnliches Vorgehen mittels Stored Outlines möglich.

Fazit
Manuelles Oracle SQL Tuning ist auch in Zeiten von DBA 2.0 und SQL Tuning- /Access Advisor noch relevant. Der DBA bzw. Developer soll verstehen, wie „teuer“ eine Abfrage ist und ob diese unnötige Resourcen verbraucht. Oft sind durch kleine Anpassungen am Statement oder am Ausführungsplan wesentliche Verbesserungen zu erzielen. Der Optimizer erhält in jeder neuen Version dutzende neue Optimierungen. Trotzdem muss berücksichtigt werden, dass er nur auf einem Modell beruht und die Realität mitsamt der Komplexität von Daten und Statement nicht wiederspiegeln kann. Zudem muss der Optimizer innerhalb von wenigen Millisekunden einen akzeptablen Plan erstellen. Das Beispiel hat gezeigt, wie anhand von manuellem SQL Tuning ein fehlender Index identifiziert wurde und die Ausführungszeit und der Ressourcenverbrauch des Statements dadurch minimiert werden konnte.

Kontaktadresse:

Martin Decker

Oracle Certified Master 10g
Oracle Certified Master 11g

ora-solutions.net
Franz-Fischer-Str. 7
D-81677 München

Telefon: +49 (0) 176 787 627 88
E-Mail martin.decker@ora-solutions.net
Internet: http://www.ora-solutions.net