dbarc: Ausarbeitung SQLTuning

Yanick Eberle, Pascal Schwarz

19. April 2013

Inhaltsverzeichnis

1 Einleitung

2 Statistiken

2.1 Statistiken sammeln

Mit dem folgenden Befehl werden die Statistiken für alle Tabellen aufgebaut:

```
1  BEGIN
2    DBMS_STATS.GATHER_TABLE_STATS('dbarc02','customers');
3    DBMS_STATS.GATHER_TABLE_STATS('dbarc02','lineitems');
4    DBMS_STATS.GATHER_TABLE_STATS('dbarc02','nations');
5    DBMS_STATS.GATHER_TABLE_STATS('dbarc02','orders');
6    DBMS_STATS.GATHER_TABLE_STATS('dbarc02','parts');
7    DBMS_STATS.GATHER_TABLE_STATS('dbarc02','partsupps');
8    DBMS_STATS.GATHER_TABLE_STATS('dbarc02','regions');
9    DBMS_STATS.GATHER_TABLE_STATS('dbarc02','suppliers');
10 END;
```

2.2 Zeilen, Bytes, Blöcke und Extents der Tabellen

Um die Anzahl Extents festzustellen, haben wir uns Informationen der Tabelle *DBA_SEGMENTS* bedient. Eine kurze Google-Recherche führte uns auf die Seite `http://www.rocket99.com/techref/oracle8409.html`, die uns bei dieser Aufgabe behilflich war.

```
1  SELECT stat.table_name, stat.num_rows, stat.blocks, seg.extents,
2    stat.avg_row_len*stat.num_rows AS size_bytes
3  FROM user_tab_statistics stat
4  JOIN DBA_SEGMENTS seg ON (stat.table_name = seg.segment_name)
5  WHERE seg.owner = 'DBARC02'
6
7  TABLE_NAME                     NUM_ROWS     BLOCKS     EXTENTS SIZE_BYTES
8  ------------------------------ ---------- ---------- ---------- ----------
9  CUSTOMERS                          150000       3494         43   23850000
10 LINEITEMS                         6001215     109217        179  750151875
11 NATIONS                             25          4          1       2675
12 ORDERS                            1500000      24284         95  166500000
13 PARTS                              200000       3859         46   26400000
14 PARTSUPPS                          800000      16650         88  114400000
15 REGIONS                                5          4          1        480
16 SUPPLIERS                          10000        220         17    1440000
```

3 Ausführungsplan

Die Ausführung des EXPLAIN PLAN-Befehles erzeugt folgende Ausgabe:

```
1  plan FOR succeeded.
```

Und die Abfrage des Ausführungsplans zeigt erwartungsgemäss einen kompletten Tabellenzugriff, da das SELECT-Statement ja keine WHERE-Klausel verwendet.

```
1  PLAN_TABLE_OUTPUT
2  ------------------------------------------------------------------------------
3  Plan hash value: 3931018009
4
5  ------------------------------------------------------------------------------
6  | Id  | Operation          | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
7  ------------------------------------------------------------------------------
8  |   0 | SELECT STATEMENT   |       |   200K|    25M|  1051   (1)| 00:00:13 |
9  |   1 |  TABLE ACCESS FULL | PARTS |   200K|    25M|  1051   (1)| 00:00:13 |
10 ------------------------------------------------------------------------------
```

4 Versuche ohne Index

4.1 Projektion

4.1.1 * FROM

Das erste Statement (SELECT * FROM...) erzeugt einen Output sehr ähnlich dem bereits Gezeigten. Es werden sämtliche 1.5 Millionen Zeilen der Tabelle gelesen. Da es sich dabei primär um I/O handelt, ist der Anteil der CPU an den Kosten mit lediglich einem Prozent entsprechend gering.

```
1 ------------------------------------------------------------------------------
2 | Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
3 ------------------------------------------------------------------------------
4 |   0 | SELECT STATEMENT   |         |  1500K|   158M|  6610   (1)| 00:01:20 |
5 |   1 |  TABLE ACCESS FULL| ORDERS  |  1500K|   158M|  6610   (1)| 00:01:20 |
6 ------------------------------------------------------------------------------
```

4.1.2 o_clerk FROM

Bei der Projektion auf eine einzige Spalte der Tabelle Orders fällt ein Grossteil der Daten weg (22M statt 158M), ansonsten sind die Unterschiede aber sehr gering. Vom Festspeicher müssen die selben Blöcke gelesen werden, erst danach können die Inhalte der nicht angefragten Spalten verworfen werden. Daher fallen auch die Kosten nur geringfügig tiefer aus.

```
1 ------------------------------------------------------------------------------
2 | Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
3 ------------------------------------------------------------------------------
4 |   0 | SELECT STATEMENT   |         |  1500K|    22M|  6607   (1)| 00:01:20 |
5 |   1 |  TABLE ACCESS FULL| ORDERS  |  1500K|    22M|  6607   (1)| 00:01:20 |
6 ------------------------------------------------------------------------------
```

4.1.3 DISTINCT o_clerk FROM

Für das SELECT DISTINCT Statement werden in einem ersten Schritt (Id:2) wiederum alle Daten der entsprechenden Spalte der Tabelle geladen (Kosten wiederum 6607). Danach werden mittels HASH UNIQUE die doppelt vorhandenen Werte ermittelt und entfernt. Dies erzeugt noch ein wenig CPU-Last, aber senkt die Anzahl Zeilen von 1.5 Millionen auf 1000 und verringert dadurch auch den Speicherbedarf von 22M auf 16000 Bytes.

```
1 ------------------------------------------------------------------------------
2 | Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
3 ------------------------------------------------------------------------------
4 |   0 | SELECT STATEMENT   |         |  1000 | 16000 |  6676   (2)| 00:01:21 |
5 |   1 |  HASH UNIQUE       |         |  1000 | 16000 |  6676   (2)| 00:01:21 |
6 |   2 |   TABLE ACCESS FULL| ORDERS  |  1500K|    22M|  6607   (1)| 00:01:20 |
7 ------------------------------------------------------------------------------
```

4.2 Selektion

4.2.1 Exact Point

Obwohl das Exact-Point Query lediglich eine einzige Zeile zurückliefert fallen die Kosten mit 6602 beinahe so hoch wie bei der Projektion auf eine einzige Spalte der selben Tabelle (ohne Selektion) aus. Da kein Index für diese Spalte vorhanden ist, kann das Datenbanksystem die Abfrage nicht effizienter als mittels linearer Suche ausführen.

```
1  ------------------------------------------------------------------------------
2  | Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT   |         |     1 |   111 |  6602   (1)| 00:01:20 |
5  |*  1 |  TABLE ACCESS FULL| ORDERS  |     1 |   111 |  6602   (1)| 00:01:20 |
6  ------------------------------------------------------------------------------
7
8  Predicate Information (identified by operation id):
9  ---------------------------------------------------
10
11    1 - filter("O_ORDERKEY"=44444)
```

4.2.2 Partial Point, OR

Die OR-Verknüpften Bedingungen und die daraus resultierende höhere Anzahl an zurückzugebenden Zeilen erhöhen die Kosten gegenüber dem Exact Point Query noch ein wenig. Weiterhin dürfte aber die Notwendigkeit des Lesens der gesamten Tabelle für die lineare Suche den grössten Teil der Kosten ausmachen.

```
1  ------------------------------------------------------------------------------
2  | Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT   |         |  1501 |  162K|  6629   (1)| 00:01:20 |
5  |*  1 |  TABLE ACCESS FULL| ORDERS  |  1501 |  162K|  6629   (1)| 00:01:20 |
6  ------------------------------------------------------------------------------
7
8  Predicate Information (identified by operation id):
9  ---------------------------------------------------
10
11    1 - filter("O_CLERK"='Clerk#000000286' OR "O_ORDERKEY"=44444)
```

4.2.3 Partial Point, AND

Wiederum muss die gesamte Tabelle geladen werden und die Kosten fallen ähnlich aus. Die gegenüber dem vorherigen Query leicht geringeren Kosten erklären wir uns folgendermassen:

- Es müssen je Zeile nur dann beide Bedingungen geprüft werden, wenn die erste Bedingung erfüllt ist.
- Nur eine einzige Zeile erfüllt beide Bedingungen.

```
1  ------------------------------------------------------------------------------
2  | Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT   |         |     1 |   111 |  6611   (1)| 00:01:20 |
5  |*  1 |  TABLE ACCESS FULL| ORDERS  |     1 |   111 |  6611   (1)| 00:01:20 |
6  ------------------------------------------------------------------------------
7
8  Predicate Information (identified by operation id):
9  ---------------------------------------------------
10
11    1 - filter("O_ORDERKEY"=44444 AND "O_CLERK"='Clerk#000000286')
```

4.2.4 Partial Point, AND und Funktion

Die Multiplikation des Feldes *O_ORDERKEY* sowie die erhöhte Anzahl an zurückzugebenden Zeilen erhöhen die Kosten gegenüber dem vorherigen Query in geringem Masse.

```
1  ------------------------------------------------------------------------------
2  | Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT   |         |    15 |  1665 |  6615   (1)| 00:01:20 |
5  |*  1 |  TABLE ACCESS FULL| ORDERS  |    15 |  1665 |  6615   (1)| 00:01:20 |
6  ------------------------------------------------------------------------------
7
8  Predicate Information (identified by operation id):
```

```
9  ---------------------------------------------------
10
11     1 - filter("O_ORDERKEY"*2=44444 AND "O_CLERK"='Clerk#000000286')
```

4.2.5 Range Query

Für das Range Query muss aufgrund der nicht vorhandenen Indices die komplette Tabelle geladen werden. Die AND-Verknüpfung erlaubt es wiederum, für viele Zeilen die Überprüfung der zweiten Bedingung zu überspringen.

```
1  ------------------------------------------------------------------------------
2  | Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT   |         | 27780 |  3011K|  6603   (1)| 00:01:20 |
5  |*  1 |  TABLE ACCESS FULL| ORDERS  | 27780 |  3011K|  6603   (1)| 00:01:20 |
6  ------------------------------------------------------------------------------
7
8  Predicate Information (identified by operation id):
9  ---------------------------------------------------
10
11     1 - filter("O_ORDERKEY"<=222222 AND "O_ORDERKEY">=111111)
```

Die grösse des Intervalls spielt in diesem Fall praktisch keine Rolle:

```
1  ------------------------------------------------------------------------------
2  | Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT   |         |  249K |   26M |  6605   (1)| 00:01:20 |
5  |*  1 |  TABLE ACCESS FULL| ORDERS  |  249K |   26M |  6605   (1)| 00:01:20 |
6  ------------------------------------------------------------------------------
7
8  Predicate Information (identified by operation id):
9  ---------------------------------------------------
10
11     1 - filter("O_ORDERKEY"<=999222 AND "O_ORDERKEY">=000111)
```

4.2.6 Partial Range Query

Das Partial Range Query weist gegenüber dem einfachen Range Query praktisch keine Unterschiede auf. Wiederum muss die gesamte Tabelle durchsucht werden und nur für wenige Zeilen brauchen alle vier Bedingungen geprüft zu werden.

```
1  ------------------------------------------------------------------------------
2  | Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT   |         |     6 |   666 |  6611   (1)| 00:01:20 |
5  |*  1 |  TABLE ACCESS FULL| ORDERS  |     6 |   666 |  6611   (1)| 00:01:20 |
6  ------------------------------------------------------------------------------
7
8  Predicate Information (identified by operation id):
9  ---------------------------------------------------
10
11     1 - filter("O_ORDERKEY"<=55555 AND "O_CLERK"<='Clerk#000000139' AND
12                "O_ORDERKEY">=44444 AND "O_CLERK">='Clerk#000000130')
```

4.3 Join

Das Query in der gegebenen Form führt auf dieser Datenbasis ohne Indices dazu, dass beide im Join beteiligten Tabellen zunächst vollständig geladen werden müssen. Die Bedingung auf Orders führt dazu, dass lediglich 25 Zeilen aus dieser Tabelle verwendet werden.

Der HASH JOIN der beiden Relationen (25 Zeilen gejoint mit 150000 Zeilen) führt zu Kosten von 953.

```
1  ------------------------------------------------------------------------------
2  | Id  | Operation           | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT    |           |    25 |  6750 |   7555   (1)| 00:01:31 |
5  |*  1 |  HASH JOIN          |           |    25 |  6750 |   7555   (1)| 00:01:31 |
6  |*  2 |   TABLE ACCESS FULL| ORDERS    |    25 |  2775 |   6602   (1)| 00:01:20 |
7  |   3 |   TABLE ACCESS FULL| CUSTOMERS |  150K|    22M|    951   (1)| 00:00:12 |
8  ------------------------------------------------------------------------------
9
10 Predicate Information (identified by operation id):
11 ---------------------------------------------------
12
13    1 - access("O_CUSTKEY"="C_CUSTKEY")
14    2 - filter("O_ORDERKEY"<100)
```

Die Formulierung des Joins mittels JOIN ... ON (bedingung) führt zum selben Ausführungsplan.

```
1  SELECT *
2  FROM orders
3  JOIN customers ON (c_custkey = o_custkey)
4  WHERE o_orderkey < 100;
```

Dies gilt auch für die Variante mit CROSS JOIN und der custkey-Bedingung in WHERE.

```
1  SELECT *
2  FROM orders
3  CROSS JOIN customers
4  WHERE o_orderkey < 100
5  AND
6  c_custkey = o_custkey;
```

5 Versuche mit Index

5.1 Erzeugung Indices

Die Indices werden gemäss den Befehlen aus der Aufgabenstellung erstellt (Zeilen 4 und 5 sind Output):

```
1  CREATE INDEX o_orderkey_ix ON orders(o_orderkey);
2  CREATE INDEX o_clerk_ix ON orders(o_clerk);
3
4  index O_ORDERKEY_IX created.
5  index O_CLERK_IX created.
```

Die Indices sind 30 resp. 48 MByte gross. Zusammen kommen sie somit beinahe auf die halbe Grösse der Tabelle *ORDERS* (ca. 160 MByte). Die Grösse der Indices haben wir gemäss folgendem Output festgestellt:

```
1  SELECT SEGMENT_NAME, BYTES
2  FROM DBA_SEGMENTS seg
3  WHERE seg.owner = 'DBARC02'
4  AND seg.segment_type = 'INDEX'
5
6  SEGMENT_NAME                              BYTES
7  ---------------------------------------- ----------
8  O_ORDERKEY_IX                             30408704
9  O_CLERK_IX                                48234496
```

5.2 Projektion

5.2.1 DISTINCT o_clerk FROM

Im Gegensatz zum Output ohne Index (siehe Abschnitt 4.1.3 auf Seite 3) fallen die Kosten bei der Abfrage mit Index merklich geringer aus. Der Schritt *HASH UNIQUE* verursacht Kosten von 69, was gegenüber der Abfrage ohne Index keinen Unterschied darstellt. Allerdings ist

der *INDEX FAST FULL SCAN* viel günstiger als *TABLE ACCESS FULL* (Kosten sinken von 6607 auf 1546).

Wir erklären uns dies dadurch, dass die Daten für das Query (lediglich Spalte *O_CLERK*) in diesem Fall direkt aus dem Index gelesen werden während ohne Index für *O_CLERK* die gesamte Tabelle von der Disk gelesen werden muss. Da der Index ca. vier mal kleiner ist als die Tabelle fallen auch die Kosten ca. viermal kleiner aus.

```
1  ------------------------------------------------------------------------------
2  | Id  | Operation                | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT         |             |  1000 | 16000 |  1615   (5)| 00:00:20 |
5  |   1 |  HASH UNIQUE             |             |  1000 | 16000 |  1615   (5)| 00:00:20 |
6  |   2 |   INDEX FAST FULL SCAN   | O_CLERK_IX  |  1500K|   22M |  1546   (1)| 00:00:19 |
7  ------------------------------------------------------------------------------
```

5.2.2 * FROM

Führen wir dasselbe Query wie in Abschnitt 4.1.1 auf Seite 3 aus, sehen wir einen gegenüber der Variante ohne Index unveränderten Ausführungsplan.

5.3 Selektion

5.3.1 Exact Point

Das Exact Point Query profitiert in enormem Ausmass vom Index auf *O_ORDERKEY* (Kosten sinken von 6602 auf 4, vgl. Abschnitt 4.2.1 auf Seite 3). Zunächst wird im Index der Eintrag mit dem entsprechenden Wert gesucht und dann die im Index enthaltene ROWID für den Zugriff auf die Tabelle benutzt (*TABLE ACCESS BY INDEX ROWID*). Der Zugriff auf die Tabelle ist notwendig, da wir die ganze Zeile und nicht nur das Feld mit dem Index ausgeben möchten.

```
1  ------------------------------------------------------------------------------
2  | Id  | Operation                    | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT             |                |     1 |   111 |     4   (0)| 00:00:01 |
5  |   1 |  TABLE ACCESS BY INDEX ROWID | ORDERS         |     1 |   111 |     4   (0)| 00:00:01 |
6  |*  2 |   INDEX RANGE SCAN           | O_ORDERKEY_IX  |     1 |       |     3   (0)| 00:00:01 |
7  ------------------------------------------------------------------------------
8
9  Predicate Information (identified by operation id):
10 ---------------------------------------------------
11
12    2 - access("O_ORDERKEY"=44444)
```

5.4 Join