dbarc: Ausarbeitung SQLTuning

Yanick Eberle, Pascal Schwarz

29. April 2013

Inhaltsverzeichnis

1 Statistiken

1.1 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;
```

1.2 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
```

2 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 ------------------------------------------------------------------------------
```

3 Versuche ohne Index

3.1 Projektion

3.1.1 * 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 ------------------------------------------------------------------------------
```

3.1.2 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 ------------------------------------------------------------------------------
```

3.1.3 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 ------------------------------------------------------------------------------
```

3.2 Selektion

3.2.1 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)
```

3.2.2 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)
```

3.2.3 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')
```

3.2.4 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')
```

3.2.5 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)
```

3.2.6 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')
```

3.3 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;
```

4 Versuche mit Index

4.1 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
```

4.2 Projektion

4.2.1 DISTINCT o_clerk FROM

Im Gegensatz zum Output ohne Index (siehe Abschnitt 3.1.3 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  ------------------------------------------------------------------------------------
```

4.2.2 * FROM

Führen wir dasselbe Query wie in Abschnitt 3.1.1 auf Seite 3 aus, sehen wir einen gegenüber der Variante ohne Index unveränderten Ausführungsplan.

4.3 Selektion

4.3.1 Exact Point

Das Exact Point Query profitiert in enormem Ausmass vom Index auf *O_ORDERKEY* (Kosten sinken von 6602 auf 4, vgl. Abschnitt 3.2.1 auf Seite 3). Zunächst wird im Index der Eintrag mit dem entsprechenden Wert von *O_ORDERKEY* 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)
```

Verwenden wir den Hint *FULL(orders)* im Statement, erhalten wir den selben Ausführungsplan wie in Abschnitt 3.2.1 auf Seite 3. Dann wird der Zugriff auf die gesuchte Zeile nicht über den Index vorgenommen.

4.3.2 Partial Point, OR

Das Partial Point Query mit der OR Bedingung profitiert vom Index auf *O_ORDERKEY* und auf *O_CLERK* (Kosten sinken von 6629 auf 336, vgl. Abschnitt 3.2.2 auf Seite 4). Für beide Bedingungen wird im Index der Eintrag mit dem entsprechenden Wert gesucht. Die gefundenen Einträge werden nun mit dem Operator OR verknüpft. Anschliessend wird mit der ermittelten ROWID ein Zugriff auf die Tabelle durchgeführt mit (*TABLE ACCESS BY INDEX ROWID*), da die komplette Zeile und nicht nur der Index ausgegeben werden soll.

Informationen zu den *BITMAP CONVERSION*s und *BITMAP OR* haben wir unter `http://gerardnico.com/wiki/database/oracle/bitmap` gefunden. Ein Bitmap-Index (der in diesem Query temporär erzeugt wird) bietet sich an, falls die Spalte eine geringe Anzahl unterschiedlicher Werte aufweist oder wir uns nur für wenige Werte interessieren. Letzteres ist nach den Zugriffen auf die einzelnen Indices (Ids 5 und 7 im Ausführungsplan) der Fall.

Für jede Zeile der Tabelle wird dabei in einem Bit gespeichert, ob der Wert in der gefragten Spalte (*o_orderkey* oder *o_clerk*) dem jeweiligen Kriterium entspricht. Diese beiden Bitsequenzen können danach OR-verknüpft werden und man erhält eine Bitsequenz, die genau für jene RowIDs ein *true* enthält, die danach im Resultat erscheinen sollen.

```
1  ------------------------------------------------------------------------------------------
2  | Id  | Operation                      | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT               |                |  1501 |  162K|   336   (0)| 00:00:05 |
5  |   1 |  TABLE ACCESS BY INDEX ROWID   | ORDERS         |  1501 |  162K|   336   (0)| 00:00:05 |
6  |   2 |   BITMAP CONVERSION TO ROWIDS  |                |       |      |            |          |
7  |   3 |    BITMAP OR                   |                |       |      |            |          |
8  |   4 |     BITMAP CONVERSION FROM ROWIDS|               |       |      |            |          |
9  |*  5 |      INDEX RANGE SCAN          | O_CLERK_IX     |       |      |     8   (0)| 00:00:01 |
10 |   6 |     BITMAP CONVERSION FROM ROWIDS|               |       |      |            |          |
11 |*  7 |      INDEX RANGE SCAN          | O_ORDERKEY_IX  |       |      |     3   (0)| 00:00:01 |
12 ------------------------------------------------------------------------------------------
13
14 Predicate Information (identified by operation id):
15 ---------------------------------------------------
16
17    5 - access("O_CLERK"='Clerk#000000286')
18    7 - access("O_ORDERKEY"=44444)
```

4.3.3 Partial Point, AND

Auch das Partial Point Query mit der AND Bedingung profitiert enorm vom Index auf *O_ORDERKEY* (Kosten sinken von 6611 auf 4, vgl. Abschnitt 3.2.3 auf Seite 4). Im Gegensatz zum Query mit OR muss hier die zweite Bedingung nur dann überprüft werden, wenn die erste Bedingung zugrifft. Daher kann direkt nach dem ersten *INDEX RANGE SCAN* der Zugriff auf die Tabelle gemacht werden. Anschliessend muss nur noch gefiltert werden (der Index *O_CLERK* wird hier nicht verwendet). Nur eine Zeile erfüllt die erste Bedingung, wodurch die Kosten so niedrig ausfallen.

```
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    1 - filter("O_CLERK"='Clerk#000000286')
13    2 - access("O_ORDERKEY"=44444)
```

4.3.4 Partial Point, AND mit Funktion

Auch das Partial Point Query mit der AND Bedingung und der Funktion profitiert vom Index auf *O_CLERK* (Kosten sinken von 6615 auf 1464, vgl. Abschnitt 3.2.4 auf Seite 4). Im Gegesatz zum Query mit AND wird hier der Zugriff auf den Index *O_CLERK* gemacht und anschliessend gefiltert. Da 1500 Zeilen die Bedingung O_CLERK='Clerk#000000286' erfüllen sind die Kosten wesentlich höher als beim Partial Point mit AND.

```
1  ------------------------------------------------------------------------------------------
2  | Id  | Operation                     | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT              |              |    15 |  1665 |  1464   (1)| 00:00:18 |
5  |*  1 |  TABLE ACCESS BY INDEX ROWID| ORDERS       |    15 |  1665 |  1464   (1)| 00:00:18 |
6  |*  2 |   INDEX RANGE SCAN            | O_CLERK_IX |  1500 |       |     8   (0)| 00:00:01 |
7  ------------------------------------------------------------------------------------------
8
9  Predicate Information (identified by operation id):
10 ---------------------------------------------------
11
12    1 - filter("O_ORDERKEY"*2=44444)
13    2 - access("O_CLERK"='Clerk#000000286')
```

4.3.5 Range Query

Das Range Query profitiert vom Index auf *O_ORDERKEY* (Kosten sinken von 6603 auf 932, vgl. Abschnitt 3.2.5 auf Seite 5). Auch hier wird ein *INDEX RANGE SCAN* durchgeführt und anschliessend für alle Treffer ein *TABLE ACCESS BY INDEX ROWID* gemacht.

```
1  ------------------------------------------------------------------------------------------
2  | Id  | Operation                     | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT              |                | 27780 |  3011K|   932   (1)| 00:00:12 |
5  |   1 |  TABLE ACCESS BY INDEX ROWID| ORDERS         | 27780 |  3011K|   932   (1)| 00:00:12 |
6  |*  2 |   INDEX RANGE SCAN            | O_ORDERKEY_IX | 27780 |       |    68   (0)| 00:00:01 |
7  ------------------------------------------------------------------------------------------
8
9  Predicate Information (identified by operation id):
10 ---------------------------------------------------
11
12    2 - access("O_ORDERKEY">=111111 AND "O_ORDERKEY"<=222222)
```

Die Intervallgrösse spielt hier eine Rolle, da mit dem Index nur bei den zutreffenden Zeilen ein Zugriff auf die Tabelle durchgeführt wird. Wenn das Intervall vergrössert wird und dadurch mehr Zeilen gefunden werden, werden automatisch mehr Zugriffe auf die Tabelle gemacht, was die Kosten erhöht.
Wenn die Range genügend gross ist, wird der Index nicht mehr verwendet, sondern es wird direkt ein *TABLE ACCESS FULL* gemacht, da mit dem INDEX kein besseres Ergebnis erzielt werden kann, wenn jede Zeile im Index ein treffer ist.

```
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)
```

4.3.6 Partial Range Query

Das Partial Range Query kann wiederum vom Index auf *O_ORDERKEY* und auf *O_CLERK* profitieren (Kosten sinken von 6611 auf 27, vgl. Abschnitt 3.2.6 auf Seite 5). Zuerst wird für jede Bedingung ein *INDEX RANGE SCAN* durchgeführt, welche anschliessend mit der Bedingung AND verknüpft werden. Erst dann wird ein *TABLE ACCESS BY INDEX ROWID* durchgeführt.

Informationen bzgl. *BITMAP* haben wir bereits in Abschnitt 4.3.2 auf Seite 7 angeführt.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                   |     6 |   666 |    27  (12)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID       | ORDERS            |     6 |   666 |    27  (12)| 00:00:01 |
|   2 |   BITMAP CONVERSION TO ROWIDS      |                   |       |       |            |          |
|   3 |    BITMAP AND                      |                   |       |       |            |          |
|   4 |     BITMAP CONVERSION FROM ROWIDS  |                   |       |       |            |          |
|   5 |      SORT ORDER BY                 |                   |       |       |            |          |
|*  6 |       INDEX RANGE SCAN             | O_ORDERKEY_IX     |  2780 |       |     9   (0)| 00:00:01 |
|   7 |     BITMAP CONVERSION FROM ROWIDS  |                   |       |       |            |          |
|   8 |      SORT ORDER BY                 |                   |       |       |            |          |
|*  9 |       INDEX RANGE SCAN             | O_CLERK_IX        |  2780 |       |    14   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   6 - access("O_ORDERKEY">=44444 AND "O_ORDERKEY"<=55555)
   9 - access("O_CLERK">='Clerk#000000130' AND "O_CLERK"<='Clerk#000000139')
```

4.4 Join

4.4.1 Join

Das gegebene Query mit dem normalen Join führt zu Kosten von 17514. Trotz Index wird ein *TABLE ACCESS FULL* gemacht. Dies aus dem Grund, dass keine Bedingung gesetzt ist und somit für jede Zeile ein Join gemacht wird. Die Kosten des *HASH JOIN* belaufen sich auf 9953.

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation           | Name        | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |             |  1500K|   386M|       | 17514   (1)| 00:03:31 |
|*  1 |  HASH JOIN          |             |  1500K|   386M|   24M| 17514   (1)| 00:03:31 |
|   2 |   TABLE ACCESS FULL | CUSTOMERS   |   150K|    22M|       |   951   (1)| 00:00:12 |
|   3 |   TABLE ACCESS FULL | ORDERS      |  1500K|   158M|       |  6610   (1)| 00:01:20 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("O_CUSTKEY"="C_CUSTKEY")
```

Mit einem Hint kann für das gleiche Query erzwungen werden, dass ein *NESTED LOOP* verwendet werden soll. Das Query sieht nun wie folgt aus:

```
SELECT /*+ USE_NL(customers orders) */*
FROM orders, customers
WHERE o_custkey = c_custkey;
```

Die Kosten für das Query mit dem *NESTED LOOP* sind wesentlich höher als mit dem *HASH JOIN*. Die kompletten Kosten belaufen sich auf 991000000, wobei die beiden *TABLE ACCESS FULL* beinahe gleich sind wie im normalen Join mit *HASH JOIN*. Der *NESTED LOOP* sollte hier auf keinen Fall verwendet werden.

```
-----------------------------------------------------------------------------------------------
| Id  | Operation           | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |             |  1500K|   386M|   991M  (1)|999:59:59 |
|   1 |  NESTED LOOPS       |             |  1500K|   386M|   991M  (1)|999:59:59 |
|   2 |   TABLE ACCESS FULL | CUSTOMERS   |   150K|    22M|   951   (1)| 00:00:12 |
|*  3 |   TABLE ACCESS FULL | ORDERS      |    10 |  1110 |  6608   (1)| 00:01:20 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("O_CUSTKEY"="C_CUSTKEY")
```

Ebenfalls mit einem Hint erzwingen wir den *MERGE JOIN*. Das Query sieht wie folgt aus:

```
SELECT /*+ USE_MERGE(customers orders) */*
FROM orders, customers
WHERE o_custkey = c_custkey;
```

Die Kosten für das Query mit dem *MERGE JOIN* sind klar tiefer als beim *NESTED LOOP* aber immer noch ca. 3 mal grösser als beim *HASH JOIN*. Sie belaufen sich auf 50568. Auch hier wird ein *TABLE ACCESS FULL* gemacht, auf welchen jeweils ein *SORT JOIN* ausgeführt wird. Der *MERGE JOIN* wird anschliessend auf die beiden *SORT JOIN* gemacht.

```
1  ------------------------------------------------------------------------------------------
2  | Id  | Operation           | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT    |           |  1500K|   386M|       | 50568   (1)| 00:10:07 |
5  |   1 |  MERGE JOIN         |           |  1500K|   386M|       | 50568   (1)| 00:10:07 |
6  |   2 |   SORT JOIN         |           |   150K|    22M|    52M|  6202   (1)| 00:01:15 |
7  |   3 |    TABLE ACCESS FULL| CUSTOMERS |   150K|    22M|       |   951   (1)| 00:00:12 |
8  |*  4 |   SORT JOIN         |           |  1500K|   158M|   390M| 44366   (1)| 00:08:53 |
9  |   5 |    TABLE ACCESS FULL| ORDERS    |  1500K|   158M|       |  6610   (1)| 00:01:20 |
10 ------------------------------------------------------------------------------------------
11
12 Predicate Information (identified by operation id):
13 ---------------------------------------------------
14
15    4 - access("O_CUSTKEY"="C_CUSTKEY")
16        filter("O_CUSTKEY"="C_CUSTKEY")
```

Fazit: Das Query mit dem *HASH JOIN* erreicht die geringsten Kosten.

4.4.2 Join, AND

Das Query Join mit AND Bedingung kann den Index *O_ORDERKEY* verwenden und kann daher die Kosten auf 957 reduzieren. Die Anzahl Zeilen bei der Tabelle ORDERS wird von 1500000 auf 25 reduziert, dies dank der AND Bedingung. Bei der Tabelle CUSTOMERS wird immer noch ein *TABLE ACCESS FULL* gemacht. Die Kosten des *HASH JOIN* belaufen sich auf 2.

```
1  ------------------------------------------------------------------------------------------
2  | Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT             |               |    25 |  6750 |   957   (1)| 00:00:12 |
5  |*  1 |  HASH JOIN                   |               |    25 |  6750 |   957   (1)| 00:00:12 |
6  |   2 |   TABLE ACCESS BY INDEX ROWID| ORDERS        |    25 |  2775 |     4   (0)| 00:00:01 |
7  |*  3 |    INDEX RANGE SCAN          | O_ORDERKEY_IX |    25 |       |     3   (0)| 00:00:01 |
8  |   4 |   TABLE ACCESS FULL          | CUSTOMERS     |  150K|    22M|   951   (1)| 00:00:12 |
9  ------------------------------------------------------------------------------------------
10
11 Predicate Information (identified by operation id):
12 ---------------------------------------------------
13
14    1 - access("O_CUSTKEY"="C_CUSTKEY")
15    3 - access("O_ORDERKEY"<100)
```

Mit einem Hint kann für das gleiche Query erzwungen werden, dass ein *NESTED LOOP* verwendet werden soll. Das Query sieht nun wie folgt aus:

```
1  SELECT /*+ USE_NL(customers orders) */*
2  FROM orders, customers
3  WHERE o_custkey = c_custkey
4  AND o_orderkey < 100;
```

Auch hier kann der Index *O_ORDERKEY* verwendet werden. Im Vergleich mit dem ersten Query ohne Bedingung sind die Kosten wesentlich kleiner (von 991000000 auf 23747). Verglichen mit dem *HASH JOIN* mit Bedingung sind die Kosten aber immer noch viel zu hoch und sollte daher nicht verwendet werden.

```
1  ------------------------------------------------------------------------------------------
2  | Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT             |               |    25 |  6750 | 23747   (1)| 00:04:45 |
5  |   1 |  NESTED LOOPS                |               |    25 |  6750 | 23747   (1)| 00:04:45 |
6  |   2 |   TABLE ACCESS BY INDEX ROWID| ORDERS        |    25 |  2775 |     4   (0)| 00:00:01 |
7  |*  3 |    INDEX RANGE SCAN          | O_ORDERKEY_IX |    25 |       |     3   (0)| 00:00:01 |
8  |*  4 |   TABLE ACCESS FULL          | CUSTOMERS     |     1 |   159 |   950   (1)| 00:00:12 |
9  ------------------------------------------------------------------------------------------
10
11 Predicate Information (identified by operation id):
12 ---------------------------------------------------
```

```
13
14      3 - access("O_ORDERKEY"<100)
15      4 - filter("O_CUSTKEY"="C_CUSTKEY")
```

Ebenfalls mit einem Hint erzwingen wir den *MERGE JOIN*. Das Query sieht wie folgt aus:

```
1  SELECT /*+ USE_MERGE(customers orders) */*
2  FROM orders, customers
3  WHERE o_custkey = c_custkey
4  AND o_orderkey < 100;
```

Der Merge Join kann ebenfalls vom Index *O_ORDERKEY* profitieren und bringt seine Kosten von 50568 ohne Bedingung auf 6207 mit AND Bedingung. Vergliche mit dem *NESTED LOOP* sind die Kosten beinahe um den Faktor 4 kleiner. Andererseits sind die Kosten gegenüber dem *HASH JOIN* um den Faktor 6 höher. Die Zeile mit Id 3 in der folgenden Ausgabe wurde durch uns aus Darstellungsgründen gekürzt.

```
1  ------------------------------------------------------------------------------------------
2  | Id  | Operation                | Name            | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT         |                 |    25 |  6750 |       |  6207   (1)| 00:01:15 |
5  |   1 |  MERGE JOIN              |                 |    25 |  6750 |       |  6207   (1)| 00:01:15 |
6  |   2 |   SORT JOIN              |                 |    25 |  2775 |       |     5  (20)| 00:00:01 |
7  |   3 |    TABLE ACCESS BY I R| ORDERS          |    25 |  2775 |       |     4   (0)| 00:00:01 |
8  |*  4 |     INDEX RANGE SCAN     | O_ORDERKEY_IX   |    25 |       |       |     3   (0)| 00:00:01 |
9  |*  5 |   SORT JOIN              |                 |  150K|    22M|    52M|  6202   (1)| 00:01:15 |
10 |   6 |    TABLE ACCESS FULL     | CUSTOMERS       |  150K|    22M|       |   951   (1)| 00:00:12 |
11 ------------------------------------------------------------------------------------------
12
13 Predicate Information (identified by operation id):
14 ---------------------------------------------------
15
16    4 - access("O_ORDERKEY"<100)
17    5 - access("O_CUSTKEY"="C_CUSTKEY")
18        filter("O_CUSTKEY"="C_CUSTKEY")
```

Fazit: Das Query mit dem *HASH JOIN* erreicht die geringsten Kosten.

4.4.3 Join, INDEX

Als erstes haben wir den neuen Index erstellt. (Ab Zeile 3 Output)

```
1  CREATE INDEX c_custkey_ix ON customers(c_custkey);
2
3  index C_CUSTKEY_IX erstellt.
```

Das Query ergibt genau denselben Ausführungsplan wie mit nur einem Index. Obwohl für beide Tabellen Indices existieren, wird jeweils ein *TABLE ACCESS FULL* durchgeführt.

```
1  ------------------------------------------------------------------------------------------
2  | Id  | Operation            | Name       | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT     |            |  1500K|   386M|       | 17514   (1)| 00:03:31 |
5  |*  1 |  HASH JOIN           |            |  1500K|   386M|   24M| 17514   (1)| 00:03:31 |
6  |   2 |   TABLE ACCESS FULL| CUSTOMERS  |   150K|    22M|       |   951   (1)| 00:00:12 |
7  |   3 |   TABLE ACCESS FULL| ORDERS     |  1500K|   158M|       |  6610   (1)| 00:01:20 |
8  ------------------------------------------------------------------------------------------
9
10 Predicate Information (identified by operation id):
11 ---------------------------------------------------
12
13    1 - access("O_CUSTKEY"="C_CUSTKEY")
```

Mit einem Hint kann für das gleiche Query erzwungen werden, dass ein *NESTED LOOP* verwendet werden soll. Das Query sieht nun wie folgt aus:

```
1  SELECT /*+ USE_NL(customers orders) */*
2  FROM orders, customers
3  WHERE o_custkey = c_custkey;
```

Das Query mit dem *NESTED LOOP* kann vom zweiten Index profitieren. Er für einen *TABLE ACCESS FULL* auf die Tabelle ORDERS durch und greift anschliessend über den Index *C_CUSTKEY* auf die Tabelle CUSTOMERS zu. Die Kosten für den *NESTED LOOP* liegen ca. bei 3000390.

```
1  ------------------------------------------------------------------------------------------
2  | Id  | Operation                     | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT              |               |  1500K|   386M|  3007K  (1)| 10:01:34 |
5  |   1 |  NESTED LOOPS                 |               |       |       |            |          |
6  |   2 |   NESTED LOOPS                |               |  1500K|   386M|  3007K  (1)| 10:01:34 |
7  |   3 |    TABLE ACCESS FULL          | ORDERS        |  1500K|   158M|  6610   (1)| 00:01:20 |
8  |*  4 |    INDEX RANGE SCAN           | C_CUSTKEY_IX  |     1 |       |     1   (0)| 00:00:01 |
9  |   5 |   TABLE ACCESS BY INDEX ROWID| CUSTOMERS     |     1 |   159 |     2   (0)| 00:00:01 |
10 ------------------------------------------------------------------------------------------
11
12 Predicate Information (identified by operation id):
13 ---------------------------------------------------
14
15    4 - access("O_CUSTKEY"="C_CUSTKEY")
```

Ebenfalls mit einem Hint erzwingen wir den *MERGE JOIN*. Das Query sieht wie folgt aus:

```
1  SELECT /*+ USE_MERGE(customers orders) */*
2  FROM orders, customers
3  WHERE o_custkey = c_custkey;
```

Auch diese Query ergibt genau denselben Ausführungsplan wie mit nur einem Index. Obwohl für beide Tabellen Indices existieren, wird jeweils ein *TABLE ACCESS FULL* durchgeführt.

```
1  ------------------------------------------------------------------------------------------
2  | Id  | Operation             | Name        | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT      |             |  1500K|   386M|       | 50568   (1)| 00:10:07 |
5  |   1 |  MERGE JOIN           |             |  1500K|   386M|       | 50568   (1)| 00:10:07 |
6  |   2 |   SORT JOIN           |             |   150K|    22M|   52M|  6202   (1)| 00:01:15 |
7  |   3 |    TABLE ACCESS FULL  | CUSTOMERS   |   150K|    22M|       |   951   (1)| 00:00:12 |
8  |*  4 |   SORT JOIN           |             |  1500K|   158M|   390M| 44366   (1)| 00:08:53 |
9  |   5 |    TABLE ACCESS FULL  | ORDERS      |  1500K|   158M|       |  6610   (1)| 00:01:20 |
10 ------------------------------------------------------------------------------------------
11
12 Predicate Information (identified by operation id):
13 ---------------------------------------------------
14
15    4 - access("O_CUSTKEY"="C_CUSTKEY")
16        filter("O_CUSTKEY"="C_CUSTKEY")
```

5 Quiz

Ohne Optimierung werden von jeder Tabelle sämtliche Zeilen gelesen. Dies hat enorme Kosten zur Folge.

Die Operation TABLE ACCESS FULL kostet auf der Tabelle LINEITEMS 29675, auf der Tabelle PARTS 1052 und auf der Tabelle PARTSUPPS 4525.

Der erste HASH JOIN auf PARTSUPPS und PARTS kostet 295. Der zweite HASH JOIN auf LINEITEMS und den ersten HASH JOIN kostet 30. Der HASH JOIN ist sehr CPU intensiv.

Ebenfalls ist erkennbar, dass die Abfrage durch die grosse Datenmenge relativ lange dauert. Da es sich um ein COUNT handelt, wird die Anzahl Zeilen am Ende auf 1 reduziert.

```
1  ------------------------------------------------------------------------------
2  | Id  | Operation             | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT      |             |     1 |    45 | 35577   (2)| 00:07:07 |
5  |   1 |  SORT AGGREGATE       |             |     1 |    45 |            |          |
6  |*  2 |   HASH JOIN           |             |     4 |   180 | 35577   (2)| 00:07:07 |
7  |*  3 |    HASH JOIN          |             |     4 |   144 |  5872   (6)| 00:01:11 |
8  |*  4 |     TABLE ACCESS FULL | PARTSUPPS   |     4 |    36 |  4525   (1)| 00:00:55 |
9  |*  5 |     TABLE ACCESS FULL | PARTS       |  2667 | 72009 |  1052   (1)| 00:00:13 |
10 |   6 |    TABLE ACCESS FULL  | LINEITEMS   |  6001K|   51M| 29675   (1)| 00:05:57 |
```

```
11 ---------------------------------------------------------------------------
12
13 Predicate Information (identified by operation id):
14 ---------------------------------------------------
15
16    2 - access("PS_PARTKEY"="L_PARTKEY" AND "PS_SUPPKEY"="L_SUPPKEY")
17    3 - access("P_PARTKEY"="PS_PARTKEY")
18        filter("PS_PARTKEY"=5 AND "P_TYPE"='MEDIUM ANODIZED BRASS' OR
19               "PS_PARTKEY"=5 AND "P_TYPE"='MEDIUM BRUSHED COPPER')
20    4 - filter("PS_PARTKEY"=5)
21    5 - filter("P_TYPE"='MEDIUM ANODIZED BRASS' OR "P_TYPE"='MEDIUM
22               BRUSHED COPPER')
```

Um die Abfrage zu optimieren, haben wir folgende Indices erstellt (ab Zeile 6 Output):

```
1 CREATE INDEX p_partkey_ix ON parts (p_partkey) ;
2 CREATE INDEX ps_partkey_ix ON partsupps (ps_partkey) ;
3 CREATE INDEX l_partkey_ix ON lineitems (l_partkey) ;
4 CREATE INDEX l_suppkey_ix ON lineitems (l_suppkey) ;
5
6 index P_PARTKEY_IX erstellt.
7 index PS_PARTKEY_IX erstellt.
8 index L_PARTKEY_IX erstellt.
9 index L_SUPPKEY_IX erstellt.
```

Wir haben versucht weitere Indices zu erstellen, welche jedoch keine weitere Auswirkungen auf die Kosten hatten, weshalb wir sie weggelassen haben:

```
1 CREATE INDEX ps_suppkey_ix ON partsupps (ps_suppkey) ;
2 CREATE INDEX p_type_ix ON parts (p_type) ;
```

Nach dem Erstellen der Indices sind die Kosten enorm gesunken, von vorher 35577 auf neu 52.

Durch die erstellten Indices werden keine TABLE ACCESS FULL Operationen mehr ausgeführt, sondern INDEX RANGE SCAN Operationen, welche nur minime Kosten verursachen.

Anstelle des HASH JOIN werden neu NESTED LOOPS verwendet.

Weiter haben wir mithilfe verschiedener Hints versucht die Kosten zu senken (*LEADING(...)*, *USE_HASH(...)*, *USE_MERGE(...)*, ...) jedoch verwendet Oracle hier automatisch den NESTED LOOP welcher in diesem Fall sehr effizient ist.

```
1  ---------------------------------------------------------------------------------------------------
2  | Id  | Operation                      | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
3  ---------------------------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT               |               |     1 |    45 |    52   (0)| 00:00:01 |
5  |   1 |  SORT AGGREGATE                |               |     1 |    45 |            |          |
6  |   2 |   NESTED LOOPS                 |               |     4 |   180 |    52   (0)| 00:00:01 |
7  |   3 |    NESTED LOOPS                |               |     4 |   144 |    12   (0)| 00:00:01 |
8  |   4 |     TABLE ACCESS BY INDEX ROWID| PARTSUPPS     |     4 |    36 |     4   (0)| 00:00:01 |
9  |*  5 |      INDEX RANGE SCAN          | PS_PARTKEY_IX |     4 |       |     3   (0)| 00:00:01 |
10 |*  6 |     TABLE ACCESS BY INDEX ROWID| PARTS         |     1 |    27 |     2   (0)| 00:00:01 |
11 |*  7 |      INDEX RANGE SCAN          | P_PARTKEY_IX  |     1 |       |     1   (0)| 00:00:01 |
12 |   8 |    BITMAP CONVERSION COUNT     |               |     1 |     9 |    52   (0)| 00:00:01 |
13 |   9 |     BITMAP AND                 |               |       |       |            |          |
14 |  10 |      BITMAP CONVERSION FROM ROWIDS|            |       |       |            |          |
15 |* 11 |       INDEX RANGE SCAN         | L_PARTKEY_IX  |    30 |       |     2   (0)| 00:00:01 |
16 |  12 |      BITMAP CONVERSION FROM ROWIDS|            |       |       |            |          |
17 |* 13 |       INDEX RANGE SCAN         | L_SUPPKEY_IX  |    30 |       |     2   (0)| 00:00:01 |
18 ---------------------------------------------------------------------------------------------------
19
20 Predicate Information (identified by operation id):
21 ---------------------------------------------------
22
23    5 - access("PS_PARTKEY"=5)
24    6 - filter(("P_TYPE"='MEDIUM ANODIZED BRASS' OR "P_TYPE"='MEDIUM BRUSHED COPPER') AND
25               ("PS_PARTKEY"=5 AND "P_TYPE"='MEDIUM ANODIZED BRASS' OR "PS_PARTKEY"=5 AND
26               "P_TYPE"='MEDIUM BRUSHED COPPER'))
27    7 - access("P_PARTKEY"="PS_PARTKEY")
28   11 - access("PS_PARTKEY"="L_PARTKEY")
29   13 - access("PS_SUPPKEY"="L_SUPPKEY")
```

6 Deep Left Join?

Beim Bushy Tree geht es darum, die Nodes gleichmässig zu verteilen. Eine gute Visualisierung, sowie ein Vergleich des Left Deep, Right Deep und Bushy Join ist unter `http://www.oaktable.net/content/right-deep-left-deep-and-bushy-joins` ersichtlich.

Wir haben versucht einen Bushy Tree zu bilden, indem wir ausschliesslich Hints verwenden (*LEADING(...)*, *USE_HASH(...)*, *NO_MERGE*, ...). Dies ist jedoch nicht möglich.

Bei unseren Recherchen sind wir auf den Link `https://sites.google.com/site/embtdbo/Home/sql-hint-documentation` gestossen, in welchem beschrieben ist, dass ein Bushy Tree zwingend Subselects benötigt. Wir zitieren:

> What Jonathan means by BUSHY is that Oracle does not execute the join of 2 or more tables and then join that result to 2 other tables (or more) to achieve that affect you have that with Common Table Expressions (CTE) known as Subquery Factoring in Oracle (ie the 'with' clause) Oracle only joins one table table at a time to the current result set. In order to make Oracle join the results sets from table A and B to the result set from tables C and D, you have to use subqueries and the NO_MERGE hint.

Mithilfe den Subselects und den Hints *NO_MERGE* haben wir folgendes Query erstellt, welches einen Bushy Tree generiert:

```
1  SELECT COUNT(*) FROM
2    (SELECT /*+ no_merge */ ps_partkey, ps_suppkey
3      FROM parts, partsupps
4      WHERE p_partkey=ps_partkey)
5     tbl_a,
6    (SELECT /*+ no_merge */ l_partkey, l_suppkey
7      FROM lineitems, orders
8      WHERE l_orderkey=o_orderkey)
9    tbl_b
10 WHERE tbl_a.ps_partkey = tbl_b.l_partkey
11 AND tbl_a.ps_suppkey=tbl_b.l_suppkey;
```

Ohne Indices erhalten wir Kosten von 64246. Dabei wird für jede Tabelle ein *TABLE ACCESS FULL* gemacht. Der Ausführungsplan wiederspiegelt die Struktur welche unter `http://www.oaktable.net/content/right-deep-left-deep-and-bushy-joins` beschrieben ist. Es werden jeweils zwei Tabellen mit einem *HASH_JOIN* verknüpft und in einer View gespeichert. Die beiden Views werden wiederum mit einem *HASH_JOIN* verknüpft, was schlussendlich zum Bushy Tree führt.

```
1  ------------------------------------------------------------------------------------------
2  | Id  | Operation                | Name        | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
3  ------------------------------------------------------------------------------------------
4  |   0 | SELECT STATEMENT         |             |     1 |    52 |       | 64246   (1)| 00:12:51 |
5  |   1 |  SORT AGGREGATE          |             |     1 |    52 |       |            |          |
6  |*  2 |   HASH JOIN              |             |   803K|    39M|    28M| 64246   (1)| 00:12:51 |
7  |   3 |    VIEW                  |             |   792K|    19M|       |  6540   (1)| 00:01:19 |
8  |*  4 |     HASH JOIN            |             |   792K|    10M|  3328K|  6540   (1)| 00:01:19 |
9  |   5 |      TABLE ACCESS FULL   | PARTS       |   200K|   976K|       |  1050   (1)| 00:00:13 |
10 |   6 |      TABLE ACCESS FULL   | PARTSUPPS   |   800K|  7031K|       |  4523   (1)| 00:00:55 |
11 |   7 |    VIEW                  |             |  6086K|   150M|       | 45282   (1)| 00:09:04 |
12 |*  8 |     HASH JOIN            |             |  6086K|   121M|    25M| 45282   (1)| 00:09:04 |
13 |   9 |      TABLE ACCESS FULL   | ORDERS      |  1500K|  8789K|       |  6599   (1)| 00:01:20 |
14 |  10 |      TABLE ACCESS FULL   | LINEITEMS   |  6001K|    85M|       | 29675   (1)| 00:05:57 |
15 ------------------------------------------------------------------------------------------
16
17 Predicate Information (identified by operation id):
18 ---------------------------------------------------
19
20    2 - access("TBL_A"."PS_PARTKEY"="TBL_B"."L_PARTKEY" AND
21                "TBL_A"."PS_SUPPKEY"="TBL_B"."L_SUPPKEY")
22    4 - access("P_PARTKEY"="PS_PARTKEY")
23    8 - access("L_ORDERKEY"="O_ORDERKEY")
```

Bei Verwendung der Indices sinken die Kosten von 64246 auf 57686. Die Struktur des Ausführungsplans verändert sich nicht. Es wird lediglich ein *INDEX FAST FULL SCAN* anstelle des einten *TABLE ACCESS FULL* verwendet.

```
-------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name              | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                   |     1 |    52 |       | 57686   (1)| 00:11:33 |
|   1 |  SORT AGGREGATE               |                   |     1 |    52 |       |            |          |
|*  2 |   HASH JOIN                   |                   |   803K|    39M|    28M| 57686   (1)| 00:11:33 |
|   3 |    VIEW                       |                   |   792K|    19M|       |  5613   (1)| 00:01:08 |
|*  4 |     HASH JOIN                 |                   |   792K|    10M|  3328K|  5613   (1)| 00:01:08 |
|   5 |      INDEX FAST FULL SCAN     | P_PARTKEY_IX      |   200K|   976K|       |   123   (1)| 00:00:02 |
|   6 |      TABLE ACCESS FULL        | PARTSUPPS         |   800K|  7031K|       |  4523   (1)| 00:00:55 |
|   7 |    VIEW                       |                   |  6086K|   150M|       | 39649   (1)| 00:07:56 |
|*  8 |     HASH JOIN                 |                   |  6086K|   121M|    25M| 39649   (1)| 00:07:56 |
|   9 |      INDEX FAST FULL SCAN     | O_ORDERKEY_IX     |  1500K|  8789K|       |   965   (2)| 00:00:12 |
|  10 |      TABLE ACCESS FULL        | LINEITEMS         |  6001K|    85M|       | 29675   (1)| 00:05:57 |
-------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("TBL_A"."PS_PARTKEY"="TBL_B"."L_PARTKEY" AND
              "TBL_A"."PS_SUPPKEY"="TBL_B"."L_SUPPKEY")
   4 - access("P_PARTKEY"="PS_PARTKEY")
   8 - access("L_ORDERKEY"="O_ORDERKEY")
```

7 Eigene SQL-Anfragen

Wir haben ein Query erstellt, welches die Bestellungen zurückgibt, die einen Preis von 502742.76 haben und die Priorität 1 besitzen.

```
SELECT *
FROM orders
WHERE O_ORDERPRIORITY='1-URGENT'
AND O_TOTALPRICE = 502742.76;
```

Das Query kommt auf Kosten von 6610. Es werden weder Indices noch andere Optimierungen vorgenommen.

```
-----------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         |     1 |   111 |  6610   (1)| 00:01:20 |
|*  1 |  TABLE ACCESS FULL | ORDERS  |     1 |   111 |  6610   (1)| 00:01:20 |
-----------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("O_TOTALPRICE"=502742.76 AND "O_ORDERPRIORITY"='1-URGENT')
```

Um die Kosten zu senken haben wir einen Index erstellt für den Gesamtpreis:

```
CREATE INDEX o_totalprice_ix ON orders (O_TOTALPRICE) ;
```

Mit den Indices sinken die Kosten von 6610 auf 5. Die Kosten konnten also um einen Faktor von über 1000 gesenkt werden.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                   |     1 |   111 |     5   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS BY INDEX ROWID  | ORDERS            |     1 |   111 |     5   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN            | O_TOTALPRICE_IX   |     1 |       |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("O_ORDERPRIORITY"='1-URGENT')
   2 - access("O_TOTALPRICE"=502742.76)
```

Für ein weiteres Beispiel haben wir ein Query gewählt, welches alle Bestellungen eines bestimmten Kundes zurückgibt.

```
SELECT * FROM orders, customers
WHERE o_custkey = c_custkey
AND c_name = 'Customer#000124831';
```

Die Kosten für das Query betragen ohne Optimierung 7569.

```
-----------------------------------------------------------------------------------------
| Id  | Operation           | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           |    15 |  4050 |  7569   (1)| 00:01:31 |
|*  1 |  HASH JOIN          |           |    15 |  4050 |  7569   (1)| 00:01:31 |
|*  2 |   TABLE ACCESS FULL | CUSTOMERS |     1 |   159 |   951   (1)| 00:00:12 |
|   3 |   TABLE ACCESS FULL | ORDERS    |  1500K|   158M|  6610   (1)| 00:01:20 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("O_CUSTKEY"="C_CUSTKEY")
   2 - filter("C_NAME"='Customer#000124831')
```

Für die Optimierung haben wir wiederum Indices erstellt, da dies eine der besten und einfachsten Optimierungsmöglichkeiten ist:

```
CREATE INDEX o_custkey_ix ON orders(o_custkey) ;
CREATE INDEX c_name_ix ON customers(c_name) ;
```

Die Indices haben die Kosten von 7569 auf 21 gesenkt. Dies entspricht einem Faktor von ca. 360.

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |               |    15 |  4050 |    21   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                 |               |       |       |            |          |
|   2 |   NESTED LOOPS                |               |    15 |  4050 |    21   (0)| 00:00:01 |
|   3 |    TABLE ACCESS BY INDEX ROWID| CUSTOMERS     |     1 |   159 |     4   (0)| 00:00:01 |
|*  4 |     INDEX RANGE SCAN          | C_NAME_IX     |     1 |       |     3   (0)| 00:00:01 |
|*  5 |    INDEX RANGE SCAN           | O_CUSTKEY_IX  |    15 |       |     2   (0)| 00:00:01 |
|   6 |   TABLE ACCESS BY INDEX ROWID | ORDERS        |    15 |  1665 |    17   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("C_NAME"='Customer#000124831')
   5 - access("O_CUSTKEY"="C_CUSTKEY")
```

8 Reflexion

In erster Linie waren wir von den enormen Kostenunterschieden überrascht, die durch die Verwendung von Indices resp. den Verzicht darauf ausgelöst werden. Dass der Kostenunterschied und damit einhergehend die Ausführungszeit der Anfragen in derartigem Ausmass gesenkt werden kann, hätten wir uns nicht gedacht.

Interessant ist ebenfalls, dass der Optimizer keinesfalls immer Zugriffe über Indices verwendet sobald diese vorhanden wären. Bei Queries ohne Selektion oder mit grossen Intervallen bei Range-Searches verwendet Oracle trotz vorhandenen Indices volle Tabellenzugriffe (vgl. Abschnitt 4.3.5 auf Seite 9), da dadurch grössere Blöcke am Stück von der Disk gelesen werden können und die hohen Kosten für IO einigermassen in Zaum gehalten werden. Eine Ausnahme diesbezüglich stellt das Query im Abschnitt 4.2.1 auf Seite 6 dar, da in diesem fall die angefragten Daten komplett im Index gespeichert sind.

Indices belegen zwar zusätzlichen Speicherplatz, aber bei den heutigen Preisen für Arbeitsspeicher und Festplatten überwiegen die Geschwindigkeitsvorteile gegenüber dem Verzicht auf

Indices klar. Da die Festplatten (und auch die darauf gespeicherten Datenbanken) immer grösser werden, aber die Geschwindigkeit nicht im selben Ausmass zunimmt wie die Speicherdichte (von Solid-State-Drives einmal abgesehen), wird sich dieser Effekt in Zukunft noch verstärken.

Indices müssten gemäss unserem Verständnis beim Einfügen und Aktualisieren von Datensätzen gewisse Kosten verursachen. Wir haben versucht dies mittels erneutem Befüllen der Orders-Tabelle (einmal mit und einmal ohne Indices) nachzuvollziehen, allerdings zeigen die Ausführungspläne dabei auch bzgl. Kosten keine Unterschiede.