SQL TUNING

Modul dbarc FS20

Autoren

- Tobias Bossert
- Jilin Elavathingal

1. Einleitung

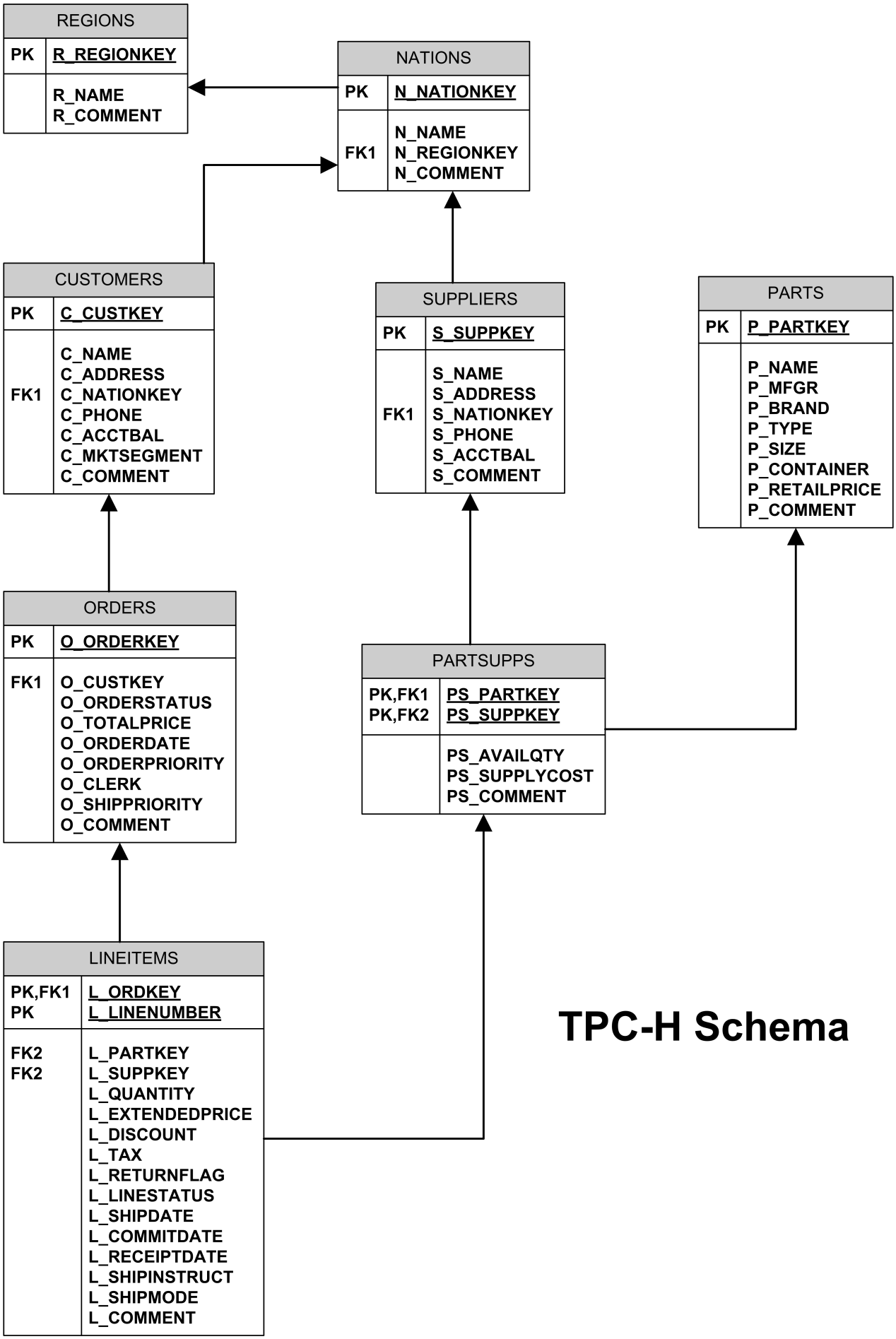
Dieses Dokument ist im Rahmen des Moduls dbarc entstanden und geht auf die Fragestellungen der Übung 8 ein. Konkret befasst sich diese mit dem SQL-Tuning. Für eine bessere Übersicht sind die SQL-Befehle in Konsolenschrift und der Syntax zusätzlich in Grossbuchstaben beschrieben, ggf. werden diese mit weiterführenden Befehlen ergänzt.

2. Vorbereitung

2.1 Manual Links

- Abbildung 4-2: Optimizer Components
- Kapitel 8: Optimizer Access Paths, Introduction to Access Paths
- Kapitel 6: Explaining and Displaying Execution Plans
- Kapitel 19: Influencing the Optimizer, Influencing the Optimizer with Hints

2.2 TPC-H Schema



2.3 Einrichten der Datenbasis

```
DROP TABLE regions;
DROP TABLE nations;
DROP TABLE parts;
DROP TABLE customers;
DROP TABLE suppliers;
DROP TABLE orders;
DROP TABLE partsupps;
DROP TABLE lineitems;

CREATE TABLE regions
AS SELECT *
FROM dbarc00.regions;

CREATE TABLE nations
AS SELECT *
FROM dbarc00.nations;

CREATE TABLE parts
AS SELECT *
FROM dbarc00.parts;

CREATE TABLE customers
AS SELECT *
FROM dbarc00.customers;

CREATE TABLE suppliers
AS SELECT *
FROM dbarc00.suppliers;

CREATE TABLE orders
AS SELECT *
FROM dbarc00.orders;

CREATE TABLE partsupps
AS SELECT *
FROM dbarc00.partsupps;

CREATE TABLE lineitems
AS SELECT *
FROM dbarc00.lineitems;
```

2.4 Statistiken erheben

Anzahl Zeilen **(a)** und Anzahl Blocks **(c)**

```
SELECT table_name, blocks, num_rows FROM all_tables WHERE owner='DBARC01';
```

	TABLE_NAME	BLOCKS	NUM_ROWS
1	CUSTOMERS	3494	150000
2	LINEITEMS	109221	6001215
3	NATIONS	4	25
4	ORDERS	24284	1500000
5	PARTS	3858	200000
6	PARTSUPPS	16651	800000
7	REGIONS	4	5
8	SUPPLIERS	220	10000

Anzahl Extents **(d)** und Anzahl Bytes **(b)**

```
SELECT segment_name, extents, bytes FROM DBA_SEGMENTS WHERE owner='DBARC01';
```

	SEGMENT_NAME ▲ 1	EXTENTS	BYTES
1	CUSTOMERS	43	29360128
2	LINEITEMS	186	902823936
3	NATIONS	1	65536
4	ORDERS	95	201326592
5	PARTS	46	32505856
6	PARTSUPPS	88	142606336
7	REGIONS	1	65536
8	SUPPLIERS	17	2097152

3. Ausführungsplan

```
EXPLAIN PLAN FOR
    SELECT * FROM ORDERS;
SELECT plan_table_output FROM TABLE(DBMS_XPLAN.DISPLAY('plan_table',null,'typical'));
```

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        |  1500K|   158M|  6599   (1)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| ORDERS |  1500K|   158M|  6599   (1)| 00:00:01 |
-------------------------------------------------------------------------------
```

Die hier dargestellte Tabelle zeigt einen Ausführungsplan welcher mit dem oberhalb angegebenen Query generiert wurde. Dabei werden die Kosten von unten nach oben aufsummiert. Sie werden berechnet aus Disk I/O, CPU Zeit und Hauptspeicher verbrauch.

In den folgenden Aufgaben verzichten wir aus Gründen der Übersichtlichkeit auf das wiederholte Ausschreiben von `EXPLAIN PLAN FOR` und `SELECT plan_table_output FROM TABLE(DBMS_XPLAN.DISPLAY('plan_table',null,'typical'));`.

4. Versuche ohne Index

4.1 Projektionen

```
SELECT * FROM ORDERS;
```

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        |  1500K|   158M|  6599   (1)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| ORDERS |  1500K|   158M|  6599   (1)| 00:00:01 |
-------------------------------------------------------------------------------
```

Bemerkung: Es wird ein Full-Table-Scan benötigt da das `SELECT`-Statement sämtliche Daten der Tabelle abfragt.

```
SELECT o_clerk FROM ORDERS;
```

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        |  1500K|    22M|  6597   (1)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| ORDERS |  1500K|    22M|  6597   (1)| 00:00:01 |
-------------------------------------------------------------------------------
```

Bemerkung: Hier ist interessant zu vermerken, dass derselbe Ausführungsplan verwendet wird wie wenn alle Spalten ausgewählt werden. Der einzige Unterschied ist der Memory-Footprint welcher von 158MB auf 22MB sinkt. Ferner sind auch die Kosten minimal tiefer. Der tiefere Memory-Footprint kommt davon, dass nur noch eine Spalte ausgewählt wird.

```
SELECT DISTINCT o_clerk FROM ORDERS;
```

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |        |  1000 | 16000 |  6640   (1)| 00:00:01 |
|   1 |  HASH UNIQUE       |        |  1000 | 16000 |  6640   (1)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| ORDERS |  1500K|    22M|  6597   (1)| 00:00:01 |
-------------------------------------------------------------------------------
```

Bemerkung: Hier starten wir wieder mit einem Full-Access-Scan über die Tabelle `orders`. Durch `HASH UNIQUE` reduziert sich die Anzahl Zeilen auf 1000. Dies bedeutet, dass es genau 1000 unique `o_clerk` gibt. Auch hier trägt der Full-Access-Scan den grössten Teil der *Cost* bei. Das zusätzliche Filtern von Duplikaten hat die leicht höheren Kosten zur Folge als dem vorherigen Query.

4.2 Selektion

4.2.1 Exact point query

```
SELECT * FROM orders WHERE o_orderkey = 44480;
```

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        |     1 |   111 |  6594   (1)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| ORDERS |     1 |   111 |  6594   (1)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("O_ORDERKEY"=44480)
```

Bemerkung: Wie bei den Projektionen auch, muss hier jeweils ein Full-Access-Scan gemacht werden um anschliessend die Filter Anwenden zu können. Da die Spalte `o_clear`knicht indiziert ist, wird beim ersten Match nicht abgebrochen. Allerdings sparen wir bei den Kosten, da eine Bedingung vorhanden ist.

Das ist bei jedem der Beispiele in diesem Kapitel der Fall.

4.2.2 Partial point query (OR)

```
SELECT * FROM orders WHERE o_custkey = 97303 OR o_clerk = 'Clerk#000000860';

-------------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        |  1515 |   164K|  6611   (1)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| ORDERS |  1515 |   164K|  6611   (1)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("O_CLERK"='Clerk#000000860' OR "O_CUSTKEY"=97303)
```

Bemerkung: Als wesentlichen Unterschied zum *exact point query* kann hier die Grösse in Bytes genannt werden. Was auch Sinn ergibt, da durch die `OR` Operation die Schnittmenge vergrössert wird. Auch die Kosten sind etwas höher, was durch das zusätzliche `OR` erklärbar ist.

4.2.3 Partial point query (AND)

```
SELECT * FROM orders WHERE o_custkey = 97303 AND o_clerk = 'Clerk#000000860';

-------------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        |     1 |   111 |  6599   (1)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| ORDERS |     1 |   111 |  6599   (1)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("O_CUSTKEY"=97303 AND "O_CLERK"='Clerk#000000860')
```

Bemerkung: Hier haben wir eigentlich wieder ein *exact point query* (Sichtbar an den Zeilen, welche jeweils 1 sind), jedoch diesmal mit 2 Bedingungen. Allerdings haben wir hier leicht höhere Kosten als in 4.2.1, da der Filter komplexer ist. Ferner bemerken wir einen Kostenersparnis gegenüber 4.2.2, da die AND-Verknüpfung die nachfolgende Bedingung nur überprüft, wenn die vorhergehende erfüllt wurde.

4.2.4 Partial point query (with sum)

```
SELECT * FROM orders WHERE o_custkey*2 = 194606 AND o_clerk = 'Clerk#000000286';

-------------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        |    15 |  1665 |  6601   (1)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| ORDERS |    15 |  1665 |  6601   (1)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("O_CUSTKEY"*2=194606 AND "O_CLERK"='Clerk#000000286')
```

Bemerkung: Hier handelt es wieder um eine AND-Verknüpfung, allerdings ist eine zusätzliche Operation im Statement ersichtlich. Die Kosten sind aber nicht auffällig gestiegen, was bedeutet die Operation wurde nur einmal angewendet, sprich sie wurde zurückgerechnet und in das folgende Statement überführt:

```
SELECT * FROM orders WHERE o_custkey = 97303 AND o_clerk = 'Clerk#000000286';
```

4.2.5 Range query

```
SELECT * FROM orders WHERE o_orderkey BETWEEN 111111 AND 222222;
```

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        | 27780 |  3011K|  6594   (1)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| ORDERS | 27780 |  3011K|  6594   (1)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("O_ORDERKEY"<=222222 AND "O_ORDERKEY">=111111)
```

Bemerkung: Die Range hat sehr wohl einen Effekt - allerdings nur auf den Speicher! Dies ist zumindest so lange der Fall wie keine Indizes bestehen. Siehe dazu auch 5.2.5.

Zur Überprüfung wurden verschiedene Intervallgrössen getestet:

```
-- larger range
SELECT * FROM orders WHERE o_orderkey BETWEEN 0 AND 222222;
```

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        | 55556 |  6022K|  6594   (1)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| ORDERS | 55556 |  6022K|  6594   (1)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("O_ORDERKEY"<=222222 AND "O_ORDERKEY">=0)
```

```
-- smaller range
SELECT * FROM orders WHERE o_orderkey BETWEEN 222220 AND 222222;
```

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        |     3 |   333 |  6594   (1)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| ORDERS |     3 |   333 |  6594   (1)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("O_ORDERKEY"<=222222 AND "O_ORDERKEY">=222220)
```

Bemerkung: Es lässt sich gut Erkennen, dass die Kosten unabhängig von der Range gleich bleiben ,da alle Zeilen traversiert werden. Beim Speicherbedarf stellen wir fest, dass diese abhängig von der Range zunimmt. Was auch Sinn ergibt da mehr/weniger Zeilen ausgewählt werden müssen.

4.2.6 Partial range query

```
SELECT * FROM orders WHERE o_orderkey BETWEEN 44444 AND 55555
AND o_clerk BETWEEN 'Clerk#000000130' AND 'Clerk#000000139';
```

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        |     6 |   666 |  6599   (1)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| ORDERS |     6 |   666 |  6599   (1)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("O_ORDERKEY"<=55555 AND "O_CLERK"<='Clerk#000000139' AND
              "O_ORDERKEY">=44444 AND "O_CLERK">='Clerk#000000130')
```

Bemerkung: Die leichte Zunahme bei den Kosten lässt sich durch die zusätzlichen Filter erklären.

4.3 Join

4.3.1 Natural join

```
SELECT * FROM orders, customers WHERE o_custkey = c_custkey;
```

```
-----------------------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |  1500K|   386M|       | 17493   (1)| 00:00:01 |
|*  1 |  HASH JOIN         |           |  1500K|   386M|    24M| 17493   (1)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| CUSTOMERS |   150K|    22M|       |   950   (1)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| ORDERS    |  1500K|   158M|       |  6599   (1)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("O_CUSTKEY"="C_CUSTKEY")
```

Bemerkung: Für diesen Join müssen beide Tabellen komplett geladen werden und da keine Filterung stattfindet, ist der `HASH JOIN` auch entsprechend teuer.

4.3.2 Mit zusätzlicher Selektion

```
SELECT * FROM orders, customers WHERE o_custkey = c_custkey AND o_orderkey < 100;
```

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |    25 |  6750 |  7544   (1)| 00:00:01 |
|*  1 |  HASH JOIN         |           |    25 |  6750 |  7544   (1)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| ORDERS    |    25 |  2775 |  6594   (1)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| CUSTOMERS |   150K|    22M|   950   (1)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("O_CUSTKEY"="C_CUSTKEY")
   2 - filter("O_ORDERKEY"<100)
```

Spielen Varianten der Formulierung eine Rolle?

```
SELECT * FROM orders INNER JOIN customers ON o_custkey = c_custkey WHERE o_orderkey < 100;
```

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |    25 |  6750 |  7544   (1)| 00:00:01 |
|*  1 |  HASH JOIN         |           |    25 |  6750 |  7544   (1)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| ORDERS    |    25 |  2775 |  6594   (1)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| CUSTOMERS |   150K|    22M|   950   (1)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("O_CUSTKEY"="C_CUSTKEY")
   2 - filter("ORDERS"."O_ORDERKEY"<100)
```

Bemerkung: Beim Natural-Join werden alle Zeilen im Rahmen des Full-Access-Scans abgearbeitet, dadurch entstehen erheblich hohe Kosten und Speicherbedarf. Bei den gefilterten Joins sehen wir keinen Unterschied betreffend dem Ausführungsplan - was darauf schliessen lässt, dass der Optimizer den Plan (zumindest in diesem Fall) unabhängig von der Formulierung erstellt. Ausser man definiert Hints, dann fallen gewisse Optimierungen weg.

5. Versuche mit Index

Indizes erstellen:

```
CREATE INDEX o_orderkey_ix ON orders(o_orderkey);
CREATE INDEX o_clerk_ix ON orders(o_clerk);
CREATE INDEX o_custkey_ix ON orders(o_custkey);
```

Wie gross sind die Indizes in Bytes?

```
SELECT segment_name,bytes FROM user_segments WHERE segment_name IN('O_ORDERKEY_IX','O_CLERK_IX','O_CUSTKEY_IX
```

	SEGMENT_NAME	BYTES
1	ORDERS	201326592
2	O_CLERK_IX	48234496
3	O_CUSTKEY_IX	28311552
4	O_ORDERKEY_IX	30408704

Bemerkung: Man sieht, dass die Grösse der Indizes von der Tabelle abhängt. Generell verhalten sie sich in etwa proportional zu der Tabellengrösse.

5.1 Projektion

```
SELECT DISTINCT o_clerk FROM ORDERS;
```

```
-------------------------------------------------------------------------------------
| Id  | Operation             | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |            |  1000 | 16000 |  1585   (4)| 00:00:01 |
|   1 |  HASH UNIQUE          |            |  1000 | 16000 |  1585   (4)| 00:00:01 |
|   2 |   INDEX FAST FULL SCAN| O_CLERK_IX |  1500K|    22M|  1542   (1)| 00:00:01 |
-------------------------------------------------------------------------------------
```

Bemerkung: Beim Vergleichen mit 4.1 sehen wir, dass kein Full-Access-Scan mehr verwendet wird, sondern nun der erstellte Index verwendet wird. Dies verdeutlicht der Eintrag mit der Id 2.

5.2 Selektion

5.2.1 Exact point query

```
SELECT * FROM orders WHERE o_orderkey = 44480;
```

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |               |     1 |   111 |     4   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS        |     1 |   111 |     4   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | O_ORDERKEY_IX |     1 |       |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------

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

   2 - access("O_ORDERKEY"=44480)
```

Bemerkung: Es wird ein Index-Range-Scan ausgeführt, welcher die ROWIDs mit den Speicherinformationen auf der Disk enthält. Diese wurden beim Indexieren erstellt, die Suche greift nur auf die Indexinformationn zu und nicht auf die eigentlichen Daten. Somit können wir die Kosten um ein Vielfaches senken.

Mit erzwungenem *full table scan*:

```
SELECT /*+ FULL(orders) */ *FROM orders WHERE o_orderkey = 44480;
```

```
----------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        |     1 |   111 |  6594   (1)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| ORDERS |     1 |   111 |  6594   (1)| 00:00:01 |
----------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("O_ORDERKEY"=44480)
```

Bemerkung: Hier wird mit einem Hint die Optimierung, in diesem Fall das Verwenden des Indexes, umgangen und ein Full-Access-Scan wird forciert. Die Kosten sind um 1600-faches höher.

5.2.2 Partial point query (OR)

```
SELECT * FROM orders WHERE o_custkey = 97303 OR o_clerk = 'Clerk#000000860';
```

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |              |  1515 |   164K|   339   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS       |  1515 |   164K|   339   (0)| 00:00:01 |
|   2 |   BITMAP CONVERSION TO ROWIDS       |              |       |       |            |          |
|   3 |    BITMAP OR                        |              |       |       |            |          |
|   4 |     BITMAP CONVERSION FROM ROWIDS   |              |       |       |            |          |
|*  5 |      INDEX RANGE SCAN               | O_CLERK_IX   |       |       |     8   (0)| 00:00:01 |
|   6 |     BITMAP CONVERSION FROM ROWIDS   |              |       |       |            |          |
|*  7 |      INDEX RANGE SCAN               | O_CUSTKEY_IX |       |       |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   5 - access("O_CLERK"='Clerk#000000860')
   7 - access("O_CUSTKEY"=97303)
```

Bemerkung: Wir sehen wieder, dass die Indizes verwendet werden. Diesmal werden gleich zwei Index-Range-Scans ausgeführt, welche mit einem Bitmap-Conversion verglichen werden. Dies ist erheblich effizienter und wird durch den Optimizer angewendet, danach kann wieder anhand der ROWID auf den Diskspeicher zugegriffen werden.

5.2.3 Partial point query (AND)

```
SELECT * FROM orders WHERE o_custkey = 97303 AND o_clerk = 'Clerk#000000860';
```

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |              |     1 |   111 |    11   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS       |     1 |   111 |    11   (0)| 00:00:01 |
|   2 |   BITMAP CONVERSION TO ROWIDS       |              |       |       |            |          |
|   3 |    BITMAP AND                       |              |       |       |            |          |
|   4 |     BITMAP CONVERSION FROM ROWIDS   |              |       |       |            |          |
|*  5 |      INDEX RANGE SCAN               | O_CUSTKEY_IX |    15 |       |     3   (0)| 00:00:01 |
|   6 |     BITMAP CONVERSION FROM ROWIDS   |              |       |       |            |          |
|*  7 |      INDEX RANGE SCAN               | O_CLERK_IX   |    15 |       |     8   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   5 - access("O_CUSTKEY"=97303)
   7 - access("O_CLERK"='Clerk#000000860')
```

Bemerkung: Gleich wie bei 5.2.2, ausser dass die AND-Verknüpfung wie schon in 4.2.3 den Ergebnisbereich noch mehr einschränkt und weniger Speicher benötigt (was am Ende viel weniger Disk-Access zur Folge hat). Ferner wird die Reihenfolge der Scans vertauscht, eine Optimierung.

5.2.4 Partial point query (with product)

```
SELECT * FROM orders WHERE o_custkey*2 = 194606 AND o_clerk = 'Clerk#000000286';
```

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |            |    15 |  1665 |  1463   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS     |    15 |  1665 |  1463   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | O_CLERK_IX |  1500 |       |     8   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("O_CUSTKEY"*2=194606)
   2 - access("O_CLERK"='Clerk#000000286')
```

Bemerkung: Hier erfolgt der Scan ebenfalls über den Index, allerdings generiert die zusätzliche Multiplikation höhere Kosten. Diese sind immerhin 6-mal (vgl. 4.2.4) geringer Dank dem erstellten Index.

5.2.5 Range query

```
SELECT * FROM orders WHERE o_orderkey BETWEEN 111111 AND 222222;
```

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |               | 27780 |  3011K|   952   (1)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS        | 27780 |  3011K|   952   (1)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | O_ORDERKEY_IX | 27780 |       |    68   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("O_ORDERKEY">=111111 AND "O_ORDERKEY"<=222222)
```

Bemerkung: Man erkennt, auch beim Range-Query wird ein Index-Range-Scan ausgeführt, allerdings durch den Index wieder etwa um Faktor 6 Kostenersparnis. Wieder untersuchen wir ob die Grösse des Ranges einen Einfluss hat:

```
-- select larger range
SELECT * FROM orders WHERE o_orderkey BETWEEN 0 AND 222222;
```

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |               | 27780 |  3011K|   952   (1)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS        | 27780 |  3011K|   952   (1)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | O_ORDERKEY_IX | 27780 |       |    68   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("O_ORDERKEY">=111111 AND "O_ORDERKEY"<=222222)
```

```
-- select smaller range
SELECT * FROM orders WHERE o_orderkey BETWEEN 222220 AND 222222;
```

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |               |     3 |   333 |     4   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS        |     3 |   333 |     4   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | O_ORDERKEY_IX |     3 |       |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("O_ORDERKEY">=222220 AND "O_ORDERKEY"<=222222)
```

Bemerkung: Es lässt sich gut erkennen, dass nun auch die Kosten signifikant tiefer sind (im Gegensatz zu vorher 4.2.5). Der Index hilft hier bei beiden Fällen den Disk-Access auf das nötige zu reduzieren.

5.2.6 Partial range query

```
SELECT * FROM orders WHERE o_orderkey BETWEEN 44444 AND 55555
AND o_clerk BETWEEN 'Clerk#000000130' AND 'Clerk#000000139';
```

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                            | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                     |               |     6 |   666 |    26   (8)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS        |     6 |   666 |    26   (8)| 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')
```

Bemerkung: Ebenso wie im vorherigen Beispiel (4.2.6) wird aus der Bedingung mit dem kleineren Subset die ROWIDs generiert, dies erfolgt mit Hilfe vom Index-Range-Scan. Danach werden die Daten abgefragt und gefiltert, wir stellen hier ein bemerkenswertes Kostenersparnis fest. (ca. Faktor 250)

5.3 Join

5.3.1 Natural join

```
SELECT * FROM orders, customers WHERE o_custkey = c_custkey;
```

```
-------------------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |  1500K|   386M|       | 17493   (1)| 00:00:01 |
|*  1 |  HASH JOIN         |           |  1500K|   386M|    24M| 17493   (1)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| CUSTOMERS |   150K|    22M|       |   950   (1)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| ORDERS    |  1500K|   158M|       |  6599   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("O_CUSTKEY"="C_CUSTKEY")
```

Bemerkung: Hier erfolgt trotz Index ein Full-Access-Scan analog zu 4.3.1.

5.3.2 Mit zusätzlicher Selektion

```
SELECT * FROM orders, customers WHERE o_custkey = c_custkey AND o_orderkey < 100;
```

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                            | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                     |               |    25 |  6750 |   955   (1)| 00:00:01 |
|*  1 |  HASH JOIN                           |               |    25 |  6750 |   955   (1)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS        |    25 |  2775 |     4   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN                  | O_ORDERKEY_IX |    25 |       |     3   (0)| 00:00:01 |
|   4 |   TABLE ACCESS FULL                  | CUSTOMERS     |   150K|    22M|   950   (1)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("O_CUSTKEY"="C_CUSTKEY")
   3 - access("O_ORDERKEY"<100)
```

Bemerkung: Da in diesem Statement eine Selektion vorgenommen wird, wird diese mit einem Index-Range-Scan für die linke Tabelle (Orders) durchgeführt. (Kostenersparnis ca. um Faktor 8)

Mit Zusätzlichem Index

```
CREATE INDEX c_custkey_ix ON customers(c_custkey);
SELECT * FROM orders, customers WHERE o_custkey = c_custkey;
```

```
-------------------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |  1500K|   386M|       | 17493   (1)| 00:00:01 |
|*  1 |  HASH JOIN         |           |  1500K|   386M|    24M| 17493   (1)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| CUSTOMERS |   150K|    22M|       |   950   (1)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| ORDERS    |  1500K|   158M|       |  6599   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("O_CUSTKEY"="C_CUSTKEY")
```

Bemerkung: Trotz erstelltem Index für die Tabelle `customers` findet kein Index-Range-Scan statt, sondern ein Full-Access-Scan, es wird also auf alle Daten zugegriffen. Die Indexe werden ignoriert da eventuell alle Daten für den Join benötigt werden.

Erzwingen eines Nested-Loop-Joins

Zitat Oracle Doc:
`The value TYPICAL displays only the hints that are not used in the final plan, whereas the value ALL displays both used and unused hints.` Aus Gründen der Übersichtlichkeit verzichten wir in den folgenden Ausgaben auf den `ALL` report. Das die Hints benutzt wurden, ist jeweils an den Ausführungsplänen ersichtlich.

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

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |              |  1500K|   386M|  1812K  (1)| 00:01:11 |
|   1 |  NESTED LOOPS                |              |  1500K|   386M|  1812K  (1)| 00:01:11 |
|   2 |   NESTED LOOPS               |              |  2250K|   386M|  1812K  (1)| 00:01:11 |
|   3 |    TABLE ACCESS FULL         | CUSTOMERS    |   150K|    22M|   950   (1)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN          | O_CUSTKEY_IX |    15 |       |     2   (0)| 00:00:01 |
|   5 |   TABLE ACCESS BY INDEX ROWID| ORDERS       |    10 |  1110 |    17   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access("O_CUSTKEY"="C_CUSTKEY")

Hint Report (identified by operation id / Query Block Name / Object Alias):
Total hints for statement: 1 (U - Unused (1))
---------------------------------------------------------------------------
   3 -  SEL$1 / CUSTOMERS@SEL$1
         U -  use_nl(ORDERS, CUSTOMERS)
*/
```

Bemerkung: Durch die Vorgabe eines NL-Joins werden alle Datensätze verglichen, die Kosten für diese Abfrage sind sehr hoch.

Erzwingen eines anderen als den Hash-Join

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

```
-------------------------------------------------------------------------------------------
| Id  | Operation           | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           |  1500K|   386M|       | 50515   (1)| 00:00:02 |
|   1 |  MERGE JOIN         |           |  1500K|   386M|       | 50515   (1)| 00:00:02 |
|   2 |   SORT JOIN         |           |   150K|    22M|    52M|  6197   (1)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| CUSTOMERS |   150K|    22M|       |   950   (1)| 00:00:01 |
|*  4 |   SORT JOIN         |           |  1500K|   158M|   390M| 44318   (1)| 00:00:02 |
|   5 |    TABLE ACCESS FULL| ORDERS    |  1500K|   158M|       |  6599   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------

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

   4 - access("O_CUSTKEY"="C_CUSTKEY")
       filter("O_CUSTKEY"="C_CUSTKEY")
```

Bemerkung: Durch die Vorgabe, dass der Hash-Join nicht verwendet werden darf, wird ein Merge-Join verwendet. Da dieses Join eine sortierte Tabelle benötigt, wird sie kurzer Hand sortiert und mit einem Full-Access-Scan ausgelesen. Dies lässt die Abfrage sehr teuer und ineffizient werden.

6. Quiz

Benchmark-Query Folgend der Ausführungsplan ohne jegliche Indizes oder Umschreibungen:

```
SELECT COUNT(*)
FROM parts, partsupps, lineitems
WHERE p_partkey=ps_partkey
AND ps_partkey=l_partkey
AND ps_suppkey=l_suppkey
AND ( (ps_suppkey = 2444 AND p_type = 'MEDIUM ANODIZED BRASS')
OR (ps_suppkey = 2444 AND p_type = 'MEDIUM BRUSHED COPPER') );
```

```
-------------------------------------------------------------------------------
| Id  | Operation            | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |           |     1 |    45 | 35220   (1)| 00:00:02 |
|   1 |  SORT AGGREGATE      |           |     1 |    45 |            |          |
|*  2 |   HASH JOIN          |           |    80 |  3600 | 35220   (1)| 00:00:02 |
|*  3 |    HASH JOIN         |           |    80 |  1440 | 34170   (1)| 00:00:02 |
|*  4 |     TABLE ACCESS FULL| PARTSUPPS |    80 |   720 |  4520   (1)| 00:00:01 |
|*  5 |     TABLE ACCESS FULL| LINEITEMS |   600 |  5400 | 29650   (1)| 00:00:02 |
|*  6 |    TABLE ACCESS FULL | PARTS     |  2667 | 72009 |  1050   (1)| 00:00:01 |
-------------------------------------------------------------------------------

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

   2 - access("P_PARTKEY"="PS_PARTKEY")
   3 - access("PS_PARTKEY"="L_PARTKEY" AND "PS_SUPPKEY"="L_SUPPKEY")
   4 - filter("PS_SUPPKEY"=2444)
   5 - filter("L_SUPPKEY"=2444)
   6 - filter("P_TYPE"='MEDIUM ANODIZED BRASS' OR "P_TYPE"='MEDIUM
              BRUSHED COPPER')
```

6.1 Ausgangslage

Die Abfrage verwendet drei Tabellen `parts`, `partsupps` und `lineitems`, auf allen Tabellen erfolgt ein Full-Access-Scan, das ist auf höchster Ebene ineffizient. Des Weiteren sind keine Indizes auf den Tabellen definiert oder werden zumindest nicht für diese Abfrage verwendet.

6.2 Lösungsansatz

Wir möchten mit Hilfe von Indizes auf die `PARTKKEY`- und `SUPPKEY`-Spalten die Abfrage optimieren. Zudem erstellen wir noch einen Bitmap Index auf `p_type`. Damit dieser allerdings benutzer werden kann, müssen noch zwei Subquerys hinzugefügt werden.

```
CREATE INDEX li_pk_ix on LINEITEMS(L_PARTKEY);
CREATE INDEX li_sk_ix on LINEITEMS(L_SUPPKEY);
CREATE INDEX p_pk_ix ON PARTS(p_partkey);
CREATE INDEX ps_pk_ix ON PARTSUPPS(ps_partkey);
CREATE INDEX ps_sk_ix ON PARTSUPPS(ps_suppkey);
CREATE INDEX p_ty_ix ON PARTS(p_type);
SELECT COUNT(*) FROM parts,partsupps,lineitems
WHERE p_partkey = ps_partkey
  AND ps_partkey = l_partkey
  AND ps_suppkey = l_suppkey
  AND ((ps_suppkey = 2444 AND p_type = 'MEDIUM ANODIZED BRASS') OR
       (ps_suppkey = 2444 AND p_type = 'MEDIUM BRUSHED COPPER'))
  AND ps_suppkey IN (SELECT ps_suppkey FROM partsupps WHERE p_partkey = 2444)
  AND p_type IN (SELECT p_type FROM parts WHERE p_type = 'MEDIUM ANODIZED BRASS' OR p_type = 'MEDIUM BRUSHED
```

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |           |     1 |    71 |    16   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE                          |           |     1 |    71 |            |          |
|   2 |   NESTED LOOPS SEMI                      |           |     1 |    71 |    16   (0)| 00:00:01 |
|   3 |    NESTED LOOPS                          |           |     1 |    49 |    14   (0)| 00:00:01 |
|   4 |     NESTED LOOPS                         |           |     1 |    22 |    12   (0)| 00:00:01 |
|   5 |      MERGE JOIN CARTESIAN                |           |     1 |    13 |     9   (0)| 00:00:01 |
|   6 |       BITMAP CONVERSION TO ROWIDS        |           |     1 |     9 |     7   (0)| 00:00:01 |
|   7 |        BITMAP AND                        |           |       |       |            |          |
|   8 |         BITMAP CONVERSION FROM ROWIDS    |           |       |       |            |          |
|*  9 |          INDEX RANGE SCAN                | LI_PK_IX  |    30 |       |     3   (0)| 00:00:01 |
|  10 |         BITMAP CONVERSION FROM ROWIDS    |           |       |       |            |          |
|* 11 |          INDEX RANGE SCAN                | LI_SK_IX  |    30 |       |     4   (0)| 00:00:01 |
|  12 |       BUFFER SORT                        |           |    80 |   320 |     2   (0)| 00:00:01 |
|  13 |        SORT UNIQUE                       |           |    80 |   320 |     2   (0)| 00:00:01 |
|* 14 |         INDEX RANGE SCAN                 | PS_SK_IX  |    80 |   320 |     2   (0)| 00:00:01 |
|* 15 |      TABLE ACCESS BY INDEX ROWID BATCHED | PARTSUPPS |     1 |     9 |     3   (0)| 00:00:01 |
|* 16 |       INDEX RANGE SCAN                   | PS_PK_IX  |     4 |       |     2   (0)| 00:00:01 |
|* 17 |     TABLE ACCESS BY INDEX ROWID BATCHED  | PARTS     |     1 |    27 |     2   (0)| 00:00:01 |
|* 18 |      INDEX RANGE SCAN                    | P_PK_IX   |     1 |       |     1   (0)| 00:00:01 |
|* 19 |    INDEX RANGE SCAN                      | P_TY_IX   |    41 |   902 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

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

   9 - access("L_PARTKEY"=2444)
  11 - access("L_SUPPKEY"=2444)
  14 - access("PS_SUPPKEY"=2444)
  15 - filter("PS_SUPPKEY"=2444 AND "PS_SUPPKEY"="PS_SUPPKEY")
  16 - access("PS_PARTKEY"=2444)
```

```
  17 - filter("P_TYPE"='MEDIUM ANODIZED BRASS' OR "P_TYPE"='MEDIUM BRUSHED COPPER')
  18 - access("P_PARTKEY"=2444)
  19 - access("P_TYPE"="P_TYPE")
       filter("P_TYPE"='MEDIUM ANODIZED BRASS' OR "P_TYPE"='MEDIUM BRUSHED COPPER')
```

6.3 Erkenntnis

Mit den Indizes und den zusätzliche Subqueries konnten wir die Abfragekosten erheblich reduzieren, fast um Faktor 2200!. Es werden nur doch die Indizes durchsucht und kein Full-Access-Scan mehr durchgeführt, diese Massnahmen entlasten zusätzlich den Speicherbedarf.

Mit dem Erstellen von geschickt gewählten Indizes lassen sich Tabellen effizienter durchsuchen, damit wird die Performance drastisch verbessert. Zudem lässt sich durch den gezielten Einsatz von Subqueries das kartesische Produkt für den Join erheblich reduzieren. Dieser Vorgang wird in den Oracle Docs als `Star Transformation` beschrieben.

7. Deep Left Join?

7.1 Vorbereitung

Als erstes erstellen wir einen Query welche die Tabellen `orders`, `lineitems`, `partsupps` und `parts` umfasst:

```
SELECT partsupps.ps_suppkey, lineitems.l_suppkey
FROM parts,partsupps,lineitems,orders
WHERE p_partkey = ps_partkey
  AND l_orderkey = o_orderkey
  AND ps_suppkey = l_suppkey;
```

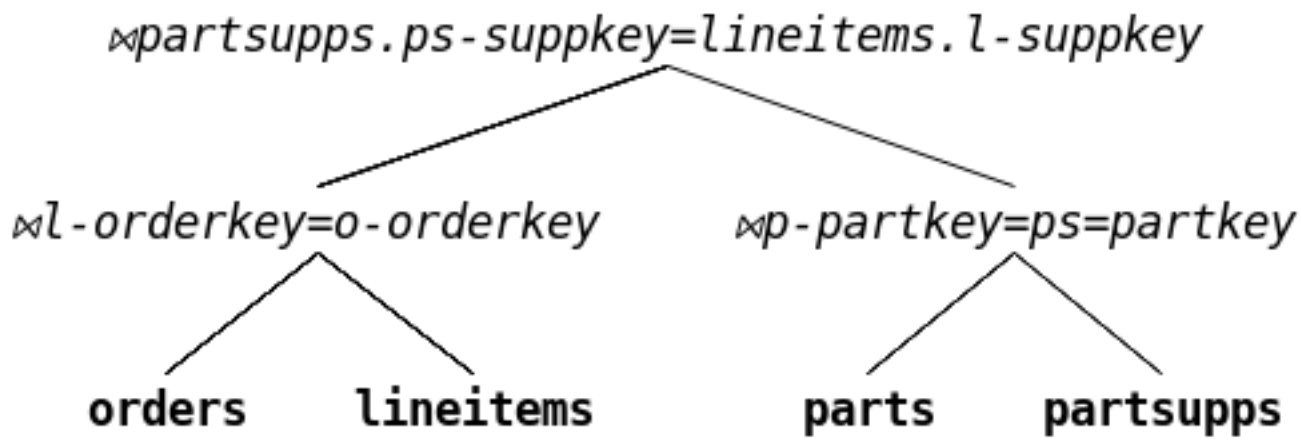
Ohne unser Eingreifen resultiert diese Query in folgendem Ausführungsplan (welcher einen `Deep Left Tree` darstellt):

```
-------------------------------------------------------------------------------------------
| Id  | Operation            | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |           |   482M|    13G|       |   864K  (1)| 00:00:34 |
|*  1 |  HASH JOIN           |           |   482M|    13G|    25M|   864K  (1)| 00:00:34 |
|   2 |   TABLE ACCESS FULL  | ORDERS    |  1500K|  8789K|       |  6591   (1)| 00:00:01 |
|*  3 |   HASH JOIN          |           |   475M|    10G|    19M| 44867   (4)| 00:00:02 |
|*  4 |    HASH JOIN         |           |   792K|    10M|  3328K|  6530   (1)| 00:00:01 |
|   5 |     TABLE ACCESS FULL| PARTS     |   200K|   976K|       |  1049   (1)| 00:00:01 |
|   6 |     TABLE ACCESS FULL| PARTSUPPS |   800K|  7031K|       |  4519   (1)| 00:00:01 |
|   7 |    TABLE ACCESS FULL | LINEITEMS |  6001K|    57M|       | 29641   (1)| 00:00:02 |
-------------------------------------------------------------------------------------------

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

   1 - access("L_ORDERKEY"="O_ORDERKEY")
   3 - access("PS_SUPPKEY"="L_SUPPKEY")
   4 - access("P_PARTKEY"="PS_PARTKEY")
```

Für den Join vollen wir den Optimizer dazu bringen folgenden `Bushy Tree` zu bilden:



Um das zu erreichen muss der Query umgeschrieben werden. Zum einen müssen zwei Subquerys für die Tabellen `orders` und `lineitems` sowie für `parts` und `partsupps` erstellt werden. Zum anderen muss der Optimizer dazu gebracht werden, diese beiden Querys separat zu Joinen. Dies erreichen wir mit den beiden Hints `LEADING(t1, t2)` und `NO_MERGE`. Folgend also der umgeschriebene Query:

```
SELECT ps.*, l.*
FROM (SELECT /*+ LEADING(parts,partsupps) NO_MERGE */ partsupps.ps_suppkey ps_sk
      FROM parts,partsupps
      WHERE p_partkey = ps_partkey) ps,
     (SELECT /*+ LEADING(orders,lineitems) NO_MERGE */ lineitems.l_suppkey l_sk
      FROM lineitems,orders
      WHERE l_orderkey = o_orderkey) l
WHERE ps.ps_sk = l.l_sk;
```

7.2 BT-Join ohne Indizes

```
-------------------------------------------------------------------------------------------
| Id  | Operation            | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |           |   482M|    11G|       | 59955   (3)| 00:00:03 |
|*  1 |  HASH JOIN           |           |   482M|    11G|    18M| 59955   (3)| 00:00:03 |
|   2 |   VIEW               |           |   792K|     9M|       |  6530   (1)| 00:00:01 |
|*  3 |    HASH JOIN         |           |   792K|    10M|  3328K|  6530   (1)| 00:00:01 |
|   4 |     TABLE ACCESS FULL| PARTS     |   200K|   976K|       |  1049   (1)| 00:00:01 |
|   5 |     TABLE ACCESS FULL| PARTSUPPS |   800K|  7031K|       |  4519   (1)| 00:00:01 |
|   6 |   VIEW               |           |  6086K|    75M|       | 43794   (1)| 00:00:02 |
|*  7 |    HASH JOIN         |           |  6086K|    92M|    25M| 43794   (1)| 00:00:02 |
|   8 |     TABLE ACCESS FULL| ORDERS    |  1500K|  8789K|       |  6591   (1)| 00:00:01 |
|   9 |     TABLE ACCESS FULL| LINEITEMS |  6001K|    57M|       | 29641   (1)| 00:00:02 |
-------------------------------------------------------------------------------------------

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

   1 - access("PS"."PS_SK"="L"."L_SK")
   3 - access("P_PARTKEY"="PS_PARTKEY")
   7 - access("L_ORDERKEY"="O_ORDERKEY")
```

Dieses Beispiel zeigt, dass der Optimizer eben nicht immer den optimalen Ausführungsplan berechnet. Der `Bushy Tree` reduziert sowohl den Memoryfootprint (11G vs. 13G) wie auch die Kosten (59961 vs. 864K). Da wir jetzt volle Kontrolle über den Query haben, müssen wir uns auch Überlegen in welcher Reihenfolgen die Joins stattfinden sollen. Drehen wir beispielsweise bei (7) die Reihenfolge mithilfe von `LEADING(lineitems, orders)` um, erhöht sich der Memory-Footprint um 100M (auf die Kosten hat es allerdings in diesem Fall kaum einen Einfluss). Grundsätzlich wollen wir immer die kleine Tabelle mit der grösseren Joinen.

7.3 BT-Join mit Indizes

Um die Ausführung weiter zu beschleunigen erstellen wir noch Indizes auf den betroffenen Spalten:

```
CREATE INDEX p_partkey_ix ON parts(p_partkey);
CREATE INDEX ps_partkey_ix ON partsupps(ps_partkey);
CREATE INDEX o_orderkey_ix ON orders(o_orderkey);
CREATE INDEX l_oderkey_ix ON lineitems(l_orderkey);
CREATE INDEX ps_suppkey_ix ON partsupps(ps_suppkey);
CREATE INDEX l_suppkey_ix ON lineitems(l_suppkey);
```

```
-------------------------------------------------------------------------------------------------
| Id  | Operation               | Name          | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |               |   482M|    11G|       | 53399   (4)| 00:00:03 |
|*  1 |  HASH JOIN              |               |   482M|    11G|    18M| 53399   (4)| 00:00:03 |
|   2 |   VIEW                  |               |   792K|     9M|       |  5604   (1)| 00:00:01 |
|*  3 |    HASH JOIN            |               |   792K|    10M|  3328K|  5604   (1)| 00:00:01 |
|   4 |     INDEX FAST FULL SCAN| P_PARTKEY_IX  |   200K|   976K|       |   123   (1)| 00:00:01 |
|   5 |     TABLE ACCESS FULL   | PARTSUPPS     |   800K|  7031K|       |  4519   (1)| 00:00:01 |
|   6 |   VIEW                  |               |  6086K|    75M|       | 38164   (1)| 00:00:02 |
|*  7 |    HASH JOIN            |               |  6086K|    92M|    25M| 38164   (1)| 00:00:02 |
|   8 |     INDEX FAST FULL SCAN| O_ORDERKEY_IX |  1500K|  8789K|       |   961   (1)| 00:00:01 |
|   9 |     TABLE ACCESS FULL   | LINEITEMS     |  6001K|    57M|       | 29641   (1)| 00:00:02 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("PS"."PS_SK"="L"."L_SK")
   3 - access("P_PARTKEY"="PS_PARTKEY")
   7 - access("L_ORDERKEY"="O_ORDERKEY")
```

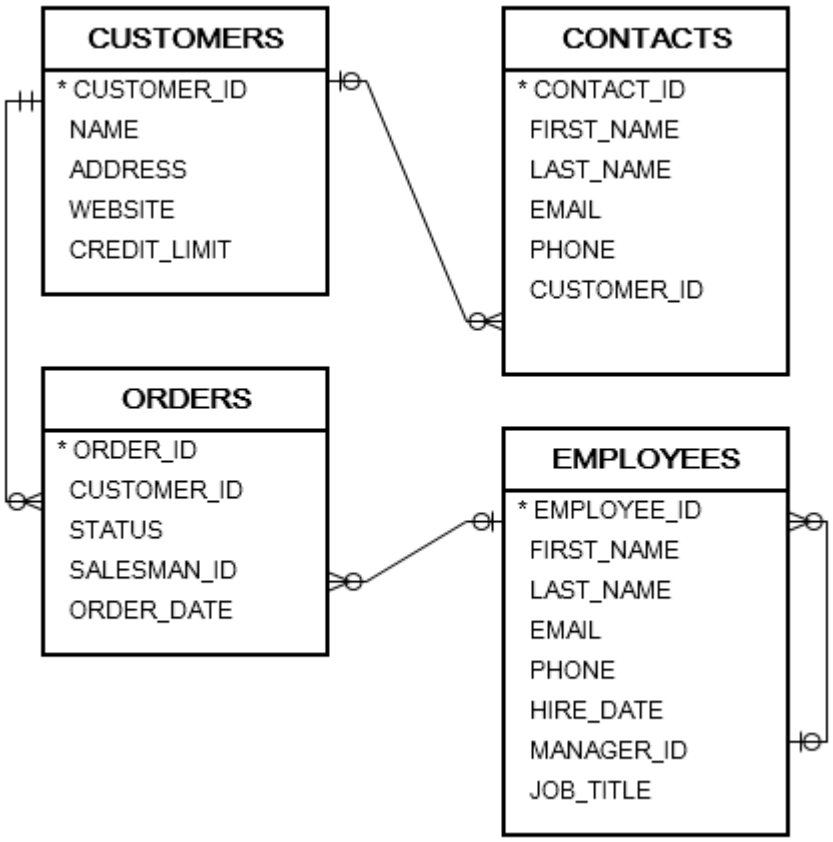
Interessanterweise, bringt uns bei diesem Query ein Index nicht einen allzu grossen Vorteil. Mann müsste wahrscheinlich noch weitere Subquerys für `partsupps` und `lineitems` definieren damit auch alle Indizes benützt werden können. Siehe dazu Star Tranformation

8. Eigene SQL Anfragen

8.1 Versuch 1

8.1.1 Vorbereitung

Wir erstellen neue Tabellen `OWN_CONT`, `OWN_CUST`, `OWN_EMP` und `OWN_ORD` und füllen diese mit entsprechenden Beispieldaten ab. Diese beschreiben jeweils Kontaktangaben von Kunden, ihre Bestellungen und den zuständigen Mitarbeiter und dessen Vorgesetzten. Im folgenden Schema lässt sich die Relation verdeutlichen:




```
-- customer
CREATE TABLE OWN_CUST
(
    customer_id NUMBER
        GENERATED BY DEFAULT AS IDENTITY START WITH 1
        PRIMARY KEY,
    name         VARCHAR2( 255 ) NOT NULL,
    address      VARCHAR2( 255 )         ,
    website      VARCHAR2( 255 )         ,
    credit_limit NUMBER( 8, 2 )
);
-- contacts
CREATE TABLE OWN_CONT
(
    contact_id NUMBER
        GENERATED BY DEFAULT AS IDENTITY START WITH 1
        PRIMARY KEY,
    first_name  VARCHAR2( 255 ) NOT NULL,
    last_name   VARCHAR2( 255 ) NOT NULL,
    email       VARCHAR2( 255 ) NOT NULL,
    phone       VARCHAR2( 20 )          ,
    customer_id NUMBER                  ,
```

```
    CONSTRAINT fk_contacts_customers
        FOREIGN KEY( customer_id )
            REFERENCES OWN_CUST( customer_id )
                ON DELETE CASCADE
);
-- employees
CREATE TABLE OWN_EMP
(
    employee_id NUMBER
        GENERATED BY DEFAULT AS IDENTITY START WITH 1
        PRIMARY KEY,
    first_name VARCHAR( 255 ) NOT NULL,
    last_name  VARCHAR( 255 ) NOT NULL,
    email      VARCHAR( 255 ) NOT NULL,
    phone      VARCHAR( 50 ) NOT NULL ,
    hire_date  DATE NOT NULL          ,
    manager_id NUMBER( 12, 0 )        , -- fk
    job_title  VARCHAR( 255 ) NOT NULL,
    CONSTRAINT fk_emp_manager
        FOREIGN KEY( manager_id )
            REFERENCES OWN_EMP( employee_id )
                ON DELETE CASCADE
);
-- orders
CREATE TABLE OWN_ORD
(
    order_id NUMBER
        GENERATED BY DEFAULT AS IDENTITY START WITH 1
        PRIMARY KEY,
    customer_id NUMBER( 6, 0 ) NOT NULL, -- fk
    status      VARCHAR( 20 ) NOT NULL ,
    salesman_id NUMBER( 6, 0 )         , -- fk
    order_date  DATE NOT NULL          ,
    CONSTRAINT fk_orders_customers
        FOREIGN KEY( customer_id )
            REFERENCES OWN_CUST( customer_id )
                ON DELETE CASCADE,
    CONSTRAINT fk_orders_employees
        FOREIGN KEY( salesman_id )
            REFERENCES OWN_EMP( employee_id )
                ON DELETE SET NULL
);
```

Das Einfügen von Beispieldaten wurde hier bewusst weggelassen, da dieser Teil nicht weiter von Bedeutung ist. Mehr Infos unter https://www.oracletutorial.com/getting-started/oracle-sample-database

8.1.2 Ausgangslage

Wir möchten nun eine Abfrage über alle Bestellungen mit dem Status 'Pending' oder 'Cancelled' machen, welche aus dem Jahr 2016 stammen. Ebenso sollen die Kontaktinformationen und der zuständige Verkäufer abgefragt werden.

```
SELECT * FROM OWN_ORD, OWN_EMP
WHERE OWN_ORD.SALESMAN_ID = OWN_EMP.EMPLOYEE_ID AND
    (OWN_ORD.STATUS = 'Pending' OR OWN_ORD.STATUS = 'Cancelled') AND
      OWN_EMP.JOB_TITLE = 'Sales Representative' AND
    OWN_ORD.ORDER_DATE BETWEEN to_date('1-JAN-16','DD-MON-RR') AND to_date('31-DEZ-16','DD-MON-RR')
ORDER BY OWN_ORD.ORDER_DATE;
```

```
-------------------------------------------------------------------------------
| Id  | Operation            | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |         |     6 |  3828 |     7  (15)| 00:00:01 |
|   1 |  SORT ORDER BY       |         |     6 |  3828 |     7  (15)| 00:00:01 |
|*  2 |   FILTER             |         |       |       |            |          |
|*  3 |    HASH JOIN         |         |     6 |  3828 |     6   (0)| 00:00:01 |
|*  4 |     TABLE ACCESS FULL| OWN_ORD |     9 |   540 |     3   (0)| 00:00:01 |
|*  5 |     TABLE ACCESS FULL| OWN_EMP |    30 | 17340 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   2 - filter(TO_DATE('31-DEZ-16','DD-MON-RR')>=TO_DATE('1-JAN-16','DD-M
              ON-RR'))
   3 - access("OWN_ORD"."SALESMAN_ID"="OWN_EMP"."EMPLOYEE_ID")
   4 - filter(("OWN_ORD"."STATUS"='Cancelled' OR
              "OWN_ORD"."STATUS"='Pending') AND "OWN_ORD"."ORDER_DATE">=TO_DATE('1-JAN
              -16','DD-MON-RR') AND "OWN_ORD"."ORDER_DATE"<=TO_DATE('31-DEZ-16','DD-MO
              N-RR'))
   5 - filter("OWN_EMP"."JOB_TITLE"='Sales Representative')

Note
-----
   - dynamic statistics used: dynamic sampling (level=2)
   - this is an adaptive plan
```

Bemerkung: Alle Tabellen werden mit einem Full-Access-Scan verarbeitet, das ist sehr ineffizient. Bei grösseren Datenmengen macht sich dies sehr schnell bemerkbar.

8.1.3 Lösungsansatz

Wie schon in den vorherigen Aufgabenstellungen können wir mit Indizes die Abfrage optimieren. Um die Ausführung weiter zu beschleunigen, erstellen wir also Indizes auf die betroffenen Spalten:

```
CREATE INDEX oo_status_ix ON OWN_ORD(STATUS);
CREATE INDEX oo_order_date_ix ON OWN_ORD(ORDER_DATE);
CREATE INDEX oe_job_title_ix ON OWN_EMP(JOB_TITLE);
```

Nun führen wir erneut die Abfrage aus [8.1.2] durch:

```
SELECT * FROM OWN_ORD, OWN_EMP
WHERE OWN_ORD.SALESMAN_ID = OWN_EMP.EMPLOYEE_ID AND
    (OWN_ORD.STATUS = 'Pending' OR OWN_ORD.STATUS = 'Cancelled') AND
      OWN_EMP.JOB_TITLE = 'Sales Representative' AND
    OWN_ORD.ORDER_DATE BETWEEN to_date('1-JAN-16','DD-MON-RR') AND to_date('31-DEZ-16','DD-MON-RR')
ORDER BY OWN_ORD.ORDER_DATE;
```

```
-------------------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |                   |     6 |  3828 |     5  (20)| 00:00:01 |
|   1 |  SORT ORDER BY                           |                   |     6 |  3828 |     5  (20)| 00:00:01 |
|*  2 |   FILTER                                 |                   |       |       |            |          |
|*  3 |    HASH JOIN                             |                   |     6 |  3828 |     4   (0)| 00:00:01 |
|*  4 |     TABLE ACCESS BY INDEX ROWID BATCHED| OWN_ORD           |     9 |   540 |     2   (0)| 00:00:01 |
|*  5 |      INDEX RANGE SCAN                    | OO_ORDER_DATE_IX  |    50 |       |     1   (0)| 00:00:01 |
|   6 |     TABLE ACCESS BY INDEX ROWID BATCHED| OWN_EMP           |    30 | 17340 |     2   (0)| 00:00:01 |
|*  7 |      INDEX RANGE SCAN                    | OE_JOB_TITLE_IX   |    30 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------------

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

   2 - filter(TO_DATE('31-DEZ-16','DD-MON-RR')>=TO_DATE('1-JAN-16','DD-MON-RR'))
   3 - access("OWN_ORD"."SALESMAN_ID"="OWN_EMP"."EMPLOYEE_ID")
   4 - filter("OWN_ORD"."STATUS"='Cancelled' OR "OWN_ORD"."STATUS"='Pending')
   5 - access("OWN_ORD"."ORDER_DATE">=TO_DATE('1-JAN-16','DD-MON-RR') AND
              "OWN_ORD"."ORDER_DATE"<=TO_DATE('31-DEZ-16','DD-MON-RR'))
   7 - access("OWN_EMP"."JOB_TITLE"='Sales Representative')

Note
-----
   - dynamic statistics used: dynamic sampling (level=2)
```

Bemerkung: Die zuvor erstellten Indizes werden verwendet und die Abfrage erfolgt mit weniger Kostenaufwand.

8.1.4 Erkenntnis

Wir konnten die Abfrage bereits für diese kleine Datenmenge optimieren. Allerdings haben wir zu wenig Testdaten, um die Effizienz mit Faktor 100 zu belegen. Deshalb greifen wir im Versuch 2 8.2 auf eine bestehende Tabelle mit grösseren Datenmengen zurück.

8.2 Versuch 2

8.2.1 Vorbereitung

Wir erstellen uns eine Kopie aus den vorhandenen Tabellen `ORDERS` und `CUSTOMERS`, diese benennen wir jeweils mit dem Prefix `TRY_`:

```
CREATE TABLE TRY_CUSTOMERS AS SELECT * FROM CUSTOMERS;
-- Table created.
CREATE TABLE TRY_ORDERS AS SELECT * FROM ORDERS;
-- Table created.
```

8.2.2 Ausgangslage

Wir möchten nun eine Abfrage über alle Bestellungen mit dem Status 'P' oder 'O' machen, welche von 1994 bis 1995 stammen. Ebenso soll der zugehörige Kunde mit dem Guthaben von 100 Einheiten abgefragt werden.

```
SELECT * FROM TRY_CUSTOMERS,TRY_ORDERS
WHERE TRY_ORDERS.O_CUSTKEY = TRY_CUSTOMERS.C_CUSTKEY
  AND (TRY_ORDERS.O_ORDERSTATUS = 'P' OR TRY_ORDERS.O_ORDERSTATUS = 'O')
  AND TRY_ORDERS.O_ORDERDATE BETWEEN to_date('1-JAN-94', 'DD-MON-RR')
  AND to_date('31-DEZ-95', 'DD-MON-RR')
  AND TRY_CUSTOMERS.C_ACCTBAL = 100;
```

```
-------------------------------------------------------------------------------------
| Id  | Operation           | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |               |     1 |   270 |  7570   (1)| 00:00:01 |
|*  1 |  FILTER             |               |       |       |            |          |
|*  2 |   HASH JOIN         |               |     1 |   270 |  7570   (1)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL| TRY_CUSTOMERS |     1 |   159 |   950   (1)| 00:00:01 |
|*  4 |    TABLE ACCESS FULL| TRY_ORDERS    | 26834 |  2908K|  6619   (1)| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(TO_DATE('31-DEZ-95','DD-MON-RR')>=TO_DATE('1-JAN-94','DD-MON-RR
              '))
   2 - access("TRY_ORDERS"."O_CUSTKEY"="TRY_CUSTOMERS"."C_CUSTKEY")
   3 - filter("TRY_CUSTOMERS"."C_ACCTBAL"=100)
   4 - filter("TRY_ORDERS"."O_ORDERDATE"<=TO_DATE('31-DEZ-95','DD-MON-RR')
              AND ("TRY_ORDERS"."O_ORDERSTATUS"='O' OR "TRY_ORDERS"."O_ORDERSTATUS"='P')
              AND "TRY_ORDERS"."O_ORDERDATE">=TO_DATE('1-JAN-94','DD-MON-RR'))
```

Bemerkung: Wie zu erwarten, erfolgt die Abfrage über die Full-Access-Scans auf die beiden Tabellen. Und dies obwohl am Ende nur eine einzige Zeile benötigt wird. Nun haben wir deutlich höhere Kosten als im Versuch 1 8.1, dies ist primär der grösseren Datenmenge geschuldet.

8.2.3 Lösungsansatz

Auch hier werden wir mit Indizes die Abfrage optimieren. Wir erstellen die Indizes auf die folgenden Spalten:

```
CREATE INDEX to_ck_ix ON TRY_ORDERS(O_CUSTKEY);
CREATE INDEX tc_ac_ix ON TRY_CUSTOMERS(C_ACCTBAL);
```

Nun führen wir erneut die Abfrage aus 8.2.2 durch:

```
SELECT * FROM TRY_CUSTOMERS,TRY_ORDERS
WHERE TRY_ORDERS.O_CUSTKEY = TRY_CUSTOMERS.C_CUSTKEY
  AND (TRY_ORDERS.O_ORDERSTATUS = 'P' OR TRY_ORDERS.O_ORDERSTATUS = 'O')
  AND TRY_ORDERS.O_ORDERDATE BETWEEN to_date('1-JAN-94', 'DD-MON-RR')
  AND to_date('31-DEZ-95', 'DD-MON-RR')
  AND TRY_CUSTOMERS.C_ACCTBAL = 100;
```

```
-----------------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |               |     1 |   270 |    20   (0)| 00:00:01 |
|*  1 |  FILTER                                  |               |       |       |            |          |
|   2 |   NESTED LOOPS                           |               |     1 |   270 |    20   (0)| 00:00:01 |
|   3 |    NESTED LOOPS                          |               |    15 |   270 |    20   (0)| 00:00:01 |
|   4 |     TABLE ACCESS BY INDEX ROWID BATCHED| TRY_CUSTOMERS |     1 |   159 |     3   (0)| 00:00:01 |
|*  5 |      INDEX RANGE SCAN                    | TC_AB_IX      |     1 |       |     1   (0)| 00:00:01 |
|*  6 |     INDEX RANGE SCAN                     | TO_CK_IX      |    15 |       |     2   (0)| 00:00:01 |
|*  7 |    TABLE ACCESS BY INDEX ROWID           | TRY_ORDERS    |     1 |   111 |    17   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(TO_DATE('31-DEZ-95','DD-MON-RR')>=TO_DATE('1-JAN-94','DD-MON-RR'))
   5 - access("TRY_CUSTOMERS"."C_ACCTBAL"=100)
   6 - access("TRY_ORDERS"."O_CUSTKEY"="TRY_CUSTOMERS"."C_CUSTKEY")
   7 - filter("TRY_ORDERS"."O_ORDERDATE"<=TO_DATE('31-DEZ-95','DD-MON-RR') AND
              ("TRY_ORDERS"."O_ORDERSTATUS"='O' OR "TRY_ORDERS"."O_ORDERSTATUS"='P') AND
              "TRY_ORDERS"."O_ORDERDATE">=TO_DATE('1-JAN-94','DD-MON-RR'))
```

Bemerkung: Die zuvor erstellten Indizes werden verwendet und die Abfrage erfolgt mit deutlich weniger Kostenaufwand.

8.2.4 Erkenntnis

Durch die erheblich grössere Datenmenge konnten wir feststellen, dass die Optimierung mind. Faktor 370 mit sich bringt. Es ist durchaus sinnvoll seine Tabellen und Datenstrukturen so zu designen, dass sie schon von Anfang an mit grösseren Datenmengen effizient umgehen kann. Indizes sind eine von vielen Faktoren, um die Datenbankarchitektur effizient zu gestalten.