Datenbank-Architektur für Fortgeschrittene

Ausarbeitung 1: Anfrageverarbeitung

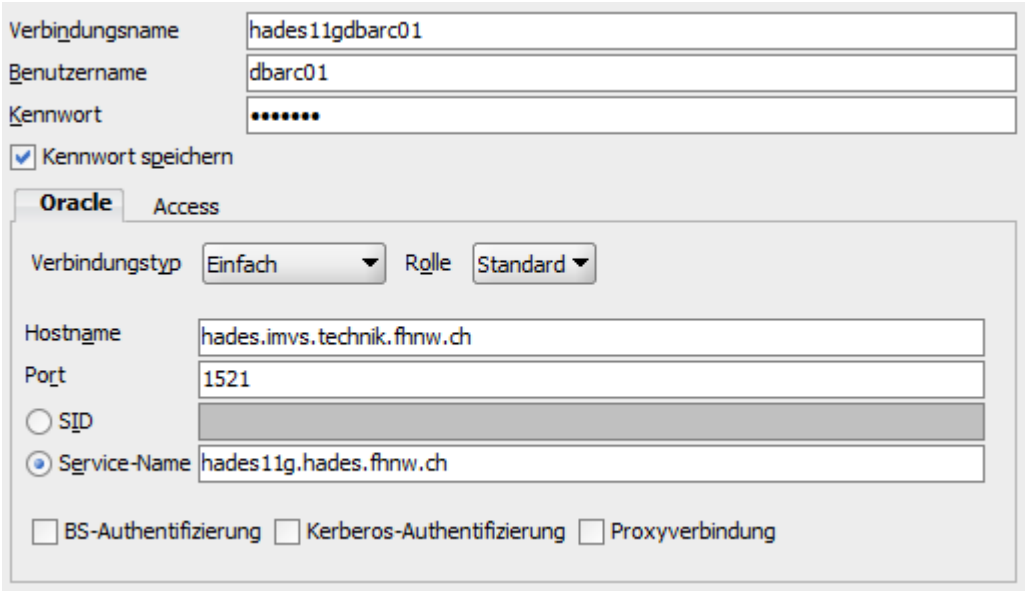
Thomas Baumann / Egemen Kaba

03. Mai 2013

Inhaltsverzeichnis

1 Einleitung

Die Ausarbeitung haben wir mit folgender Datenbankverbindung ausgeführt.



2 Vorbereitung

2.1 Einrichten Datenbasis

Die Datenbank haben wir mit folgenden Querys eingerichtet.

SQL-Query

```
1  CREATE TABLE regions
2  AS SELECT *
3    FROM dbarc00.regions;
4
5  CREATE TABLE nations
6  AS SELECT *
7    FROM dbarc00.nations;
8
9  CREATE TABLE parts
10 AS SELECT *
11   FROM dbarc00.parts;
12
13 CREATE TABLE customers
14 AS SELECT *
15   FROM dbarc00.customers;
16
17 CREATE TABLE suppliers
18 AS SELECT *
19   FROM dbarc00.suppliers;
20
21 CREATE TABLE orders
22 AS SELECT *
23   FROM dbarc00.orders;
24
25 CREATE TABLE partsupps
26 AS SELECT *
27   FROM dbarc00.partsupps;
28
29 CREATE TABLE lineitems
30 AS SELECT *
31   FROM dbarc00.lineitems;
```

2.2 Tabellenstatistik

SQL-Query

```
1 SELECT segment_name, bytes, blocks, extents
2 FROM user_segments;
3
4 SELECT table_name, num_rows
5 FROM user_tab_statistics;
```

Folgende Tabellenstatistik haben wir mit den oben genannten Querys erhoben.

Tabelle	Anzahl Zeilen	Grösse in Bytes	Anzahl Blöcke	Anzahl Extents
CUSTOMERS	150'000	29'360'128	3'584	43
LINEITEMS	6'001'215	897'581'056	109'568	178
NATIONS	25	65'536	8	1
ORDERS	1'500'000	201'326'592	24'576	95
PARTS	200'000	32'505'856	3'968	46
PARTSUPPS	800'000	142'606'336	17'408	88
REGIONS	5	65'536	8	1
SUPPLIERS	10'000	2'097'152	256	17

Je nachdem, wo man die Werte ausliest, erhält man andere Werte für die Grösse und Anzahl Blöcke.

3 Ausführungsplan

SQL-Query

```
1 EXPLAIN PLAN FOR
2 SELECT *
3 FROM parts;
4
5 SELECT plan_table_output
6 FROM TABLE(DBMS_XPLAN.DISPLAY('plan_table',null,'serial'));
```

Ausführungsplan

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |       |   190K|    26M|  1051   (1)| 00:00:13 |
|   1 |  TABLE ACCESS FULL| PARTS |   190K|    26M|  1051   (1)| 00:00:13 |
-------------------------------------------------------------------------------
```

Die Tabelle zeigt die einzelnen Schritte des Ausführungsplanes, welche der Optimizer erstellt hat, mit den jeweilig zurückgegebenen Anzahl Zeilen, deren Grösse, die Kosten und Zeit für die Teilschritte. Man kann sich den Ausführungsplan als Baum vorstellen. Die Einrückungen stellen die Knotentiefe dar. Die Kosten für einen Elternknoten werden aus der Summe der Kosten der Kindknoten plus die eigenen Kosten berechnet.

Für die nächsten Aufgaben verwenden wir das obenstehende Statement. Wir haben jeweils das Query ausgetauscht um den Ausführungsplan zu erhalten.

4 Versuche ohne Index

4.1 Projektion

SQL-Query

```
1 SELECT *
2 FROM orders;
```

Ausführungsplan

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         |  1579K|   209M|  6612   (1)| 00:01:20 |
|   1 |  TABLE ACCESS FULL| ORDERS  |  1579K|   209M|  6612   (1)| 00:01:20 |
-------------------------------------------------------------------------------
```

Reflexion
Da alle Zeilen und Spalten der Tabelle ausgelesen werden müssen, wird hier ein Full Table Scan durchgeführt. Da alle Spalten verwendet werden, muss keine Projektion vorgenommen werden.

SQL-Query

```
1 SELECT o_clerk
2 FROM orders;
```

Ausführungsplan

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         |  1579K|    25M|  6608   (1)| 00:01:20 |
|   1 |  TABLE ACCESS FULL| ORDERS  |  1579K|    25M|  6608   (1)| 00:01:20 |
-------------------------------------------------------------------------------
```

Reflexion
Es werden wiederum alle Zeile, jedoch nicht alle Spalten ausgelesen. Exakt läuft es so ab, dass zuerst die ganze Tabelle gelesen wird und danach die nicht benötigten Spalten herausgefiltert werden. Das verringert die Anzahl Daten, die gespeichert werden müssen, drastisch von 209MB auf 25MB.

SQL-Query

```
1 SELECT DISTINCT o_clerk
2 FROM orders;
```

Ausführungsplan

```
-------------------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         |  1579K|    25M|       | 15333   (1)| 00:03:04 |
|   1 |  HASH UNIQUE       |         |  1579K|    25M|    36M| 15333   (1)| 00:03:04 |
|   2 |   TABLE ACCESS FULL| ORDERS  |  1579K|    25M|       |  6608   (1)| 00:01:20 |
-------------------------------------------------------------------------------------
```

Reflexion
Wie beim vorherigen Befehl werden alle Zeilen einer bestimmten Spalte ausgelesen, was wiederum einen Zugriff auf die gesamte Tabelle nötig macht. Zusätzlich zu diesem Aufwand müssen alle doppelten Einträge herausgefiltert werden, was die massiv angestiegenen CPU-Kosten bei Operation 1 erklärt. Für die Speicherung von Zwischenresultaten wird dabei temporärer Speicher beansprucht.

4.2 Selektion

SQL-Query

```
1 SELECT *
2 FROM orders
3 WHERE o_orderkey = 44444;
```

Ausführungsplan

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |        |   267 | 37113 |  6603   (1)| 00:01:20 |
|*  1 |  TABLE ACCESS FULL| ORDERS |   267 | 37113 |  6603   (1)| 00:01:20 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("O_ORDERKEY"=44444)
```

Reflexion
Es soll nur ein Tupel ausgewählt werden (Spalte ist Primary Key), da jedoch kein index besteht, ist nicht bekannt, welches der Eintrag ist und zusätzlich kann nach dem ersten Fund nicht abgebrochen werden. Deshalb muss wieder ein Full Table Scan durchgeführt werden. Durch die Bedingung wird viel weniger Hauptspeicher benötigt.

SQL-Query

```
1 SELECT *
2 FROM orders
3 WHERE o_orderkey = 44444 OR o_clerk = 'Clerk#000000286';
```

Ausführungsplan

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |        |   267 | 37113 |  6631   (1)| 00:01:20 |
|*  1 |  TABLE ACCESS FULL| ORDERS |   267 | 37113 |  6631   (1)| 00:01:20 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("O_ORDERKEY"=44444 OR "O_CLERK"='Clerk#000000286')
```

Reflexion
Im Vergleich zum vorherigen Query sind nur die Kosten minimal gestiegen, dies ist auf die zweite Bedingung zurück zu führen.

SQL-Query

```
1 SELECT *
2 FROM orders
3 WHERE o_orderkey = 44444 AND o_clerk = 'Clerk#000000286';
```

Ausführungsplan

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         |   158 | 21962 |  6612   (1)| 00:01:20 |
|*  1 |  TABLE ACCESS FULL| ORDERS  |   158 | 21962 |  6612   (1)| 00:01:20 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("O_ORDERKEY"=44444 AND "O_CLERK"='Clerk#000000286')
```

Reflexion
Durch die AND-verknüpfung muss die zweite Bedingung nur überprüft werden, wenn die Erste erfüllt ist. So sind alle Werte, ausser der Zeit gesunken.

SQL-Query

```
1 SELECT *
2 FROM orders
3 WHERE o_orderkey*2 = 44444 AND o_clerk = 'Clerk#000000286';
```

Ausführungsplan

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         |   158 | 21962 |  6616   (1)| 00:01:20 |
|*  1 |  TABLE ACCESS FULL| ORDERS  |   158 | 21962 |  6616   (1)| 00:01:20 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("O_ORDERKEY"*2=44444 AND "O_CLERK"='Clerk#000000286')
```

Reflexion
Die Werte sind alle genau gleich, wie beim vorherigen Query, obwohl eigentlich eine zusätzliche Operation pro Zeile notwendig ist ($o\_orderkey * 2$). Wir vermuten, dass dieses Query vor der Abfrage optimiert wird, besser gesagt, die Berechnung wird vereinfacht, so muss nur eine Operation ausgeführt werden:

```
1 o_orderkey = 22222 AND o_clerk = 'Clerk#000000286'
```

SQL-Query

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

Ausführungsplan

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        |   267 | 37113 |  6603   (1)| 00:01:20 |
|*  1 |  TABLE ACCESS FULL| ORDERS |   267 | 37113 |  6603   (1)| 00:01:20 |
-------------------------------------------------------------------------------

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

Reflexion

Die Selektion

```
1 column BETWEEN x AND y
```

wird durch folgendes

```
1 column >= x AND column <= y
```

ersetzt. Im Vergleich zum Vorherigen, sind die Anzahl Zeilen und die Speicherbenutzung gestiegen, dies weil mehr Tupel selektiert werden. Hingegen sind die Kosten minimal gesunken, da die beiden Vergleiche auf der gleichen Spalte vorgenommen werden.

SQL-Query

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

Ausführungsplan

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        |    10 |  1390 |  6613   (1)| 00:01:20 |
|*  1 |  TABLE ACCESS FULL| ORDERS |    10 |  1390 |  6613   (1)| 00:01:20 |
-------------------------------------------------------------------------------

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

Reflexion

Auch hier wurden die BETWEEN-Statements mit grösser gleich und kleiner gleich ersetzt. Die Resultierende Anzahl Tupel und der benötigte Speicher sind nochmals gesunken. Wohin gegen die Kosten gestiegen sind, da bis zu vier Vergleiche notwendig sind.

4.3 Join

SQL-Query

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

Ausführungsplan

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |    25 |  6750 |  7555   (1)| 00:01:31 |
|*  1 |  HASH JOIN         |           |    25 |  6750 |  7555   (1)| 00:01:31 |
|*  2 |   TABLE ACCESS FULL| ORDERS    |    25 |  2775 |  6602   (1)| 00:01:20 |
|   3 |   TABLE ACCESS FULL| CUSTOMERS |  150K|    22M|   951   (1)| 00:00:12 |
-------------------------------------------------------------------------------

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

Varianten

SQL-Query

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

Ausführungsplan

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |    25 |  6750 |  7556   (1)| 00:01:31 |
|*  1 |  HASH JOIN         |           |    25 |  6750 |  7556   (1)| 00:01:31 |
|*  2 |   TABLE ACCESS FULL| ORDERS    |    25 |  2775 |  6604   (1)| 00:01:20 |
|   3 |   TABLE ACCESS FULL| CUSTOMERS |  150K|    22M|   951   (1)| 00:00:12 |
-------------------------------------------------------------------------------

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

SQL-Query

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

Ausführungsplan

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |    25 |  6750 |  7555   (1)| 00:01:31 |
|*  1 |  HASH JOIN         |           |    25 |  6750 |  7555   (1)| 00:01:31 |
|*  2 |   TABLE ACCESS FULL| ORDERS    |    25 |  2775 |  6602   (1)| 00:01:20 |
|   3 |   TABLE ACCESS FULL| CUSTOMERS |  150K|    22M|   951   (1)| 00:00:12 |
-------------------------------------------------------------------------------

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

SQL-Query

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

Ausführungsplan

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |    25 |  6750 |  7555   (1)| 00:01:31 |
|*  1 |  HASH JOIN         |           |    25 |  6750 |  7555   (1)| 00:01:31 |
|*  2 |   TABLE ACCESS FULL| ORDERS    |    25 |  2775 |  6602   (1)| 00:01:20 |
|   3 |   TABLE ACCESS FULL| CUSTOMERS |  150K|    22M|   951   (1)| 00:00:12 |
-------------------------------------------------------------------------------

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

Reflexion

Der Optimizer wählt bei allen Varianten automatisch die Perfomanteste aus. Somit kann man sagen, dass die Rheinfolge in der WHERE Klausel keine Rolle spielt.

Es wird immer ein Full Table Scan auf die Tabelle orders mit der Selektion id=2 vorgenommen, somit sind nur die benötigten Datensätze im Resultat. Das Resultat wird mit einem Hash Join mit der customers Tabelle vereinigt, welche mit einem Full Table Scan gelesen wurde. Da auf der Tabelle orders eine Selektion vorgenommen wird, wird diese Tabelle immer auf die Linke Seite des Joins genommen.

5 Versuche mit Index

Mit folgenden Befehlen wurden die Indizes erstellt.

SQL-Query

```
1 CREATE INDEX o_orderkey_ix ON orders(o_orderkey);
2 CREATE INDEX o_clerk_ix ON orders(o_clerk);
```

Mit folgendem Befehl wurden die Statistiken für die Indizes erhoben.

SQL-Query

```
1 SELECT segment_name, bytes
2 FROM user_segments;
```

Index	Grösse in Bytes	Tabellen Grösse in Bytes	Anteil von Index an Tabelle
O_ORDERKEY_IX	30'408'704	201'326'592	15.10%
O_CLERK_IX	48'234'496	201'326'592	23.96%

5.1 Projektion

SQL-Query

```
1 SELECT DISTINCT o_clerk
2 FROM orders;
```

Ausführungsplan

```
-------------------------------------------------------------------------------
| Id  | Operation             | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |            |  1000 | 16000 |  1622   (5)| 00:00:20 |
|   1 |  HASH UNIQUE          |            |  1000 | 16000 |  1622   (5)| 00:00:20 |
|   2 |   INDEX FAST FULL SCAN| O_CLERK_IX |  1500K|    22M|  1553   (1)| 00:00:19 |
-------------------------------------------------------------------------------
```

Reflexion
Statt dem Full Table Scan wird jetzt ein Index Range Scan angewendet. Dadurch können die benötigten Tupel wesentlich schneller gefunden werden. Im Vergleich ohne Index werden nur minimal weniger Tupel ausgelesen, jedoch viel schneller. Hingegen der Hash Unique wird massiv schneller durchgearbeitet und liefert auch weniger Tupel zurück. Bei beiden Projektionen beanspruchen keine Kosten.

5.2 Selektion

SQL-Query

```
1 SELECT *
2 FROM orders
3 WHERE o_orderkey = 44444;
```

Ausführungsplan

```
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |     1 |   111 |     4   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| 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"=44444)
```

Reflexion
Hier wird selektiv mittels eines Index Range Scans gesucht. Es liefert die Position auf der Disk mittels der ROWID. Anhand dieser ROWID wird wiederum direkt auf die Tabelle zugegriffen.

SQL-Query

```
1 SELECT /*+ FULL(orders) */ *
2 FROM orders
3 WHERE o_orderkey = 44444;
```

Ausführungsplan

```
-----------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         |     1 |   111 |  6602   (1)| 00:01:20 |
|*  1 |  TABLE ACCESS FULL| ORDERS  |     1 |   111 |  6602   (1)| 00:01:20 |
-----------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("O_ORDERKEY"=44444)
```

Reflexion
Der Hint erzwingt einen Full Table Scan, was zu einen massiven Anstieg des Ressourcenverbrauchs führt. Der Index wird dabei nicht benutzt.

SQL-Query

```
1 SELECT *
2 FROM orders
3 WHERE o_orderkey = 44444 OR o_clerk = 'Clerk#000000286';
```

Ausführungsplan

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                      | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |               |  1501 |   162K|   336   (0)| 00:00:05 |
|   1 |  TABLE ACCESS BY INDEX ROWID   | ORDERS        |  1501 |   162K|   336   (0)| 00:00:05 |
|   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_ORDERKEY_IX |       |       |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   5 - access("O_CLERK"='Clerk#000000286')
   7 - access("O_ORDERKEY"=44444)
```

Reflexion
Durch den Zugriff auf Indizes wird auch hier direkt auf die benötigten Tupel zugegriffen. Statt einer werden zwei Indizes verwendet, da die OR-Verknüpfung den Zugriff auf zwei indexierte Spalten verlangt.

SQL-Query

```
1 SELECT *
2 FROM orders
3 WHERE o_orderkey = 44444 AND o_clerk = 'Clerk#000000286';
```

Ausführungsplan

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                      | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |               |     1 |   111 |     4   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| 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):
---------------------------------------------------
   1 - filter("O_CLERK"='Clerk#000000286')
   2 - access("O_ORDERKEY"=44444)
```

Reflexion
Da ein Index auf o_orderkey besteht, werden direkt nur die benötigten Tupel ausgelesen. Was so gut wie keine Kosten verursacht und Zeit benötigt.

SQL-Query

```
1 SELECT *
2 FROM orders
3 WHERE o_orderkey*2 = 44444 AND o_clerk = 'Clerk#000000286';
```

Ausführungsplan

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |    15 |  1665 |  1470   (1)| 00:00:18 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| ORDERS      |    15 |  1665 |  1470   (1)| 00:00:18 |
|*  2 |   INDEX RANGE SCAN           | O_CLERK_IX  |  1500 |       |     8   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("O_ORDERKEY"*2=44444)
   2 - access("O_CLERK"='Clerk#000000286')
```

Reflexion

Im Vergleich zum vorherigen Ausführungsplan sind die beiden Selektionen vertauscht. Dies kommt zustande, da o_orderkey*2 erst nachher berechnet wird.

SQL-Query

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

Ausführungsplan

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               | 27780 |  3011K|   932   (1)| 00:00:12 |
|   1 |  TABLE ACCESS BY INDEX ROWID| ORDERS        | 27780 |  3011K|   932   (1)| 00:00:12 |
|*  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)
```

Reflexion

Der Optimizer wandelt den BETWEEN-Befehl in zwei mathematische Operationen um.
Hier wird der Index Range Scan ausgeführt, weil der Range klein genug gewählt wurde, dass sich die Anzahl IO-Zugriffe noch lohnt.

SQL-Query

```
1 SELECT *
2 FROM orders
3 WHERE o_orderkey BETWEEN 111111 AND 222222123;
```

Ausführungsplan

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         |  1472K|   155M|  6617   (1)| 00:01:20 |
|*  1 |  TABLE ACCESS FULL| ORDERS  |  1472K|   155M|  6617   (1)| 00:01:20 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("O_ORDERKEY">=111111 AND "O_ORDERKEY"<=222222123)
```

Reflexion
Hier wird nun ein Full Table Scan ausgeführt, da der Range zu gross ist. Ein Index Range Scan würde zu viele IO-Zugriffe verursachen.

SQL-Query

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

Ausführungsplan

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

Reflexion
Hier wurden die Between-Klausel auch in zwei mathematische Funktionen umgewandelt.
Durch die Indizes werden Bitmaps verwendet, was zu ein schneller Ausführung führt.

5.3 Join

SQL-Query

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

Ausführungsplan

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

Reflexion
Auf die Tabellen wird mit einem Full Table Scan zugegriffen. In der Hash Join Funktion werden die beiden Tabellen vereint, wobei die kleinere Tabelle auf der linken Seite ist. Dabei wird vom Wert der ersten Tabelle ein Hash erzeugt und der zugehörige Wert wird in der grossen Tabelle gesucht.

SQL-Query

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

Ausführungsplan

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |               |    25 |  6750 |   957   (1)| 00:00:12 |
|*  1 |  HASH JOIN                    |               |    25 |  6750 |   957   (1)| 00:00:12 |
|   2 |   TABLE ACCESS BY INDEX ROWID| 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|   951   (1)| 00:00:12 |
-----------------------------------------------------------------------------------------------

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

Reflexion
Im Vergleich zum vorherigen Query wird eine zusätzliche Selektion verwendet. Dies führt dazu, dass die Tabelle orders zuerst mit einem Index Range Scan auf die gewünschten Tupel durchsucht wird. Dadurch ist diese Tabelle kleiner als die andere und rückt auf die Linke Seite der Hash Join Funktion.

Es wird ein neuer Index eingefügt:

SQL-Query

```
1 CREATE INDEX c_custkey_ix ON customer(c_custkey);
```

SQL-Query

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

Ausführungsplan

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

Reflexion
Auf die Tabelle wird ein Full Table Scan aufgeführt, da es aus Sicht des Optimizers nicht lohnt einen Index Range Scan durchzuführen. Zudem müsste sowieso jeder Tupel auf die Bedingung überprüft werden.

Im Folgenden wird mittels Hint angegeben, dass ein Nested Loop angewendet werden soll.
Zudem wird exemplarisch das Basisbeispiel der vorherigen Übungen von Kapitel 5.3 verwendet.

SQL-Query

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

Ausführungsplan

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |              |  1500K|   386M|  3007K  (1)| 10:01:34 |
|   1 |  NESTED LOOPS                |              |       |       |            |          |
|   2 |   NESTED LOOPS               |              |  1500K|   386M|  3007K  (1)| 10:01:34 |
|   3 |    TABLE ACCESS FULL         | ORDERS       |  1500K|   158M|  6610   (1)| 00:01:20 |
|*  4 |    INDEX RANGE SCAN          | C_CUSTKEY_IX |     1 |       |     1   (0)| 00:00:01 |
|   5 |   TABLE ACCESS BY INDEX ROWID| CUSTOMERS    |     1 |   159 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

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

Reflexion
Der Ausführungsplan zeigt auf, dass hier die Anwendung von Nested Loops Ressourcentechnisch unsinnig ist. Der Grund liegt darin, dass hier über jedes einzelne Tupel iteriert werden muss, was nicht nötig wäre.

Im Folgenden wird mittels Hint angegeben, dass kein Hash Join angewendet werden soll.
Zudem wird exemplarisch das Basisbeispiel der vorherigen Übung von Kapitel 5.3 verwendet.

SQL-Query

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

Ausführungsplan

```
-------------------------------------------------------------------------------------------
| Id  | Operation           | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           |  1500K|   386M|       | 50568   (1)| 00:10:07 |
|   1 |  MERGE JOIN         |           |  1500K|   386M|       | 50568   (1)| 00:10:07 |
|   2 |   SORT JOIN         |           |   150K|    22M|    52M|  6202   (1)| 00:01:15 |
|   3 |    TABLE ACCESS FULL| CUSTOMERS |   150K|    22M|       |   951   (1)| 00:00:12 |
|*  4 |   SORT JOIN         |           |  1500K|   158M|   390M| 44366   (1)| 00:08:53 |
|   5 |    TABLE ACCESS FULL| ORDERS    |  1500K|   158M|       |  6610   (1)| 00:01:20 |
-------------------------------------------------------------------------------------------

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

Reflexion

Wenn der Hash Join nicht verwendet werden darf, wird hier ein Merge Join durchgeführt. Auch hier sind die Kosten höher als beim Hash Join. Der Grund dafür ist, dass zum einten die Tabellen für den Merge zuvor sortiert werden müssen.

6 Quiz

SQL-Query

```
1 SELECT count(*)
2 FROM parts, partsupps, lineitems
3 WHERE p_partkey=ps_partkey
4 AND ps_partkey=l_partkey
5 AND ps_suppkey=l_suppkey
6 AND ( (ps_partkey = 5 AND p_type = 'MEDIUM ANODIZED BRASS')
7 OR (ps_partkey = 5 AND p_type = 'MEDIUM BRUSHED COPPER') );
```

Ausführungsplan

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

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("PS_PARTKEY"="L_PARTKEY" AND "PS_SUPPKEY"="L_SUPPKEY")
   3 - access("P_PARTKEY"="PS_PARTKEY")
       filter("PS_PARTKEY"=5 AND "P_TYPE"='MEDIUM ANODIZED BRASS' OR
              "PS_PARTKEY"=5 AND "P_TYPE"='MEDIUM BRUSHED COPPER')
   4 - filter("PS_PARTKEY"=5)
   5 - filter("P_TYPE"='MEDIUM ANODIZED BRASS' OR "P_TYPE"='MEDIUM BRUSHED COPPER')
```

Wie aus dem Ausführungsplan ohne Indizes hervorgeht, werden 35'577 Kosten verursacht. Diese werden hauptsächlich durch einen Full Table Scan auf LINEITEMS verursacht.

Wir haben danach verschieden Indizes eingeführt und sind schliesslich auf folgendes Endergebnis gekommen:

SQL-Query

```
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 ps_suppkey_ix ON partsupps(ps_suppkey);
5 CREATE INDEX l_suppkey_ix ON lineitems(l_suppkey);
6 CREATE INDEX p_type_ix ON parts(p_type);
```

Ausführungsplan

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                |     1 |    45 |    52   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE                    |                |     1 |    45 |            |          |
|   2 |   NESTED LOOPS                     |                |     4 |   180 |    52   (0)| 00:00:01 |
|   3 |    NESTED LOOPS                    |                |     4 |   144 |    12   (0)| 00:00:01 |
|   4 |     TABLE ACCESS BY INDEX ROWID    | PARTSUPPS      |     4 |    36 |     4   (0)| 00:00:01 |
|*  5 |      INDEX RANGE SCAN              | PS_PARTKEY_IX  |     4 |       |     3   (0)| 00:00:01 |
|*  6 |     TABLE ACCESS BY INDEX ROWID    | PARTS          |     1 |    27 |     2   (0)| 00:00:01 |
|*  7 |      INDEX RANGE SCAN              | P_PARTKEY_IX   |     1 |       |     1   (0)| 00:00:01 |
|   8 |    BITMAP CONVERSION COUNT         |                |     1 |     9 |    52   (0)| 00:00:01 |
|   9 |     BITMAP AND                     |                |       |       |            |          |
|  10 |      BITMAP CONVERSION FROM ROWIDS |                |       |       |            |          |
|* 11 |       INDEX RANGE SCAN             | L_PARTKEY_IX   |    30 |       |     2   (0)| 00:00:01 |
|  12 |      BITMAP CONVERSION FROM ROWIDS |                |       |       |            |          |
|* 13 |       INDEX RANGE SCAN             | L_SUPPKEY_IX   |    30 |       |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   5 - access("PS_PARTKEY"=5)
   6 - filter(("P_TYPE"='MEDIUM ANODIZED BRASS' OR "P_TYPE"='MEDIUM BRUSHED COPPER') AND
              ("PS_PARTKEY"=5 AND "P_TYPE"='MEDIUM ANODIZED BRASS' OR "PS_PARTKEY"=5 AND
              "P_TYPE"='MEDIUM BRUSHED COPPER'))
   7 - access("P_PARTKEY"="PS_PARTKEY")
  11 - access("PS_PARTKEY"="L_PARTKEY")
  13 - access("PS_SUPPKEY"="L_SUPPKEY")
```

Durch die Erstellung der Indizes werden keine Full Table Scans, sondern Index Range Scans durchgeführt. Als Hilfskonstruktion werden Bitmaps und Nested Loops verwendet. Dies führt zu Gesamtkosten von 52, was zu einer Kostenersparnis von Faktor 685 führt.

7 Deep Left Join

Verwendetes Statement, um ein initiales Deep Left Join zu erzeugen:

SQL-Query

```
1 SELECT *
2 FROM orders, lineitems, partsupps, parts
3 WHERE orders.o_orderkey = lineitems.l_orderkey
4 AND lineitems.l_suppkey = partsupps.ps_suppkey
5 AND partsupps.ps_partkey = parts.p_partkey;
```

Ausführungsplan

```
-------------------------------------------------------------------------------------------
| Id  | Operation             | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |           |   482M|   229G|       |  9176K  (1)| 30:35:13 |
|*  1 |  HASH JOIN            |           |   482M|   229G|    27M|  9176K  (1)| 30:35:13 |
|   2 |   TABLE ACCESS FULL   | PARTS     |   200K|    25M|       |  1051   (1)| 00:00:13 |
|*  3 |   HASH JOIN           |           |   486M|   171G|   118M|   168K  (2)| 00:33:39 |
|   4 |    TABLE ACCESS FULL  | PARTSUPPS |   800K|   109M|       |  4526   (1)| 00:00:55 |
|*  5 |    HASH JOIN          |           |  6086K|  1369M|   175M| 84027   (1)| 00:16:49 |
|   6 |     TABLE ACCESS FULL | ORDERS    |  1500K|   158M|       |  6610   (1)| 00:01:20 |
|   7 |     TABLE ACCESS FULL | LINEITEMS |  6001K|   715M|       | 29752   (1)| 00:05:58 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("PARTSUPPS"."PS_PARTKEY"="PARTS"."P_PARTKEY")
   3 - access("LINEITEMS"."L_SUPPKEY"="PARTSUPPS"."PS_SUPPKEY")
   5 - access("ORDERS"."O_ORDERKEY"="LINEITEMS"."L_ORDERKEY")
```

Reflexion
Im ersten Statement sieht man gut, dass ein Deep Left Join erzeugt wird.
Die Kosten sind dementsprechend extrem hoch. Diese Kosten werden vor allem durch Joins von Tabellen mit bereits gejointen Tabellen verursacht.

Modifiziertes Statement, um einen Bushy Tree zu erzeugen:

SQL-Query

```
SELECT  *
FROM (
  SELECT /*+ no_merge */ *
  FROM orders, lineitems
  WHERE orders.o_orderkey = lineitems.l_orderkey
), (
  SELECT /*+ no_merge */ *
  FROM partsupps, parts
  WHERE partsupps.ps_partkey = parts.p_partkey
)
WHERE l_suppkey = ps_suppkey;
```

Ausführungsplan

```
-------------------------------------------------------------------------------------------
| Id  | Operation            | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |           |   482M|   286G|       |   211K  (2)| 00:42:23 |
|*  1 |  HASH JOIN           |           |   482M|   286G|   234M|   211K  (2)| 00:42:23 |
|   2 |   VIEW               |           |   792K|   225M|       | 12812   (1)| 00:02:34 |
|*  3 |    HASH JOIN         |           |   792K|   207M|    27M| 12812   (1)| 00:02:34 |
|   4 |     TABLE ACCESS FULL| PARTS     |   200K|    25M|       |  1051   (1)| 00:00:13 |
|   5 |     TABLE ACCESS FULL| PARTSUPPS |   800K|   109M|       |  4526   (1)| 00:00:55 |
|   6 |   VIEW               |           |  6086K|  1967M|       | 84027   (1)| 00:16:49 |
|*  7 |    HASH JOIN         |           |  6086K|  1369M|   175M| 84027   (1)| 00:16:49 |
|   8 |     TABLE ACCESS FULL| ORDERS    |  1500K|   158M|       |  6610   (1)| 00:01:20 |
|   9 |     TABLE ACCESS FULL| LINEITEMS |  6001K|   715M|       | 29752   (1)| 00:05:58 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("L_SUPPKEY"="PS_SUPPKEY")
   3 - access("PARTSUPPS"."PS_PARTKEY"="PARTS"."P_PARTKEY")
   7 - access("ORDERS"."O_ORDERKEY"="LINEITEMS"."L_ORDERKEY")
```

Reflexion

Im zweiten Statement wird nun durch Umformulierung und Einfügen von Hints explizit angegeben, in welcher Reihenfolge die Tabellen gejoined und dass diese nicht gemerged werden sollen. Im Ausführungsplan sieht man sehr gut, dass zuerst zwei Views mit jeweils zwei gejointen Tabellen erstellt werden. Anschliessend werden diese beiden Views gejoint, was schlussendlich zu einem Bushy-Tree führt. Ebenfalls sind die Kosten massiv gesunken, fast um einem Faktor von 40.

Folgende Indizes wurden nun erstellt, um die Anfragen schneller durchlaufen zu lassen:

SQL-Query

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

Das Ergebnis bei Left Deep Join:

Ausführungsplan

```
-------------------------------------------------------------------------------------------
| Id  | Operation            | Name       | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |            |   482M|   229G|       |  9176K  (1)| 30:35:13 |
|*  1 |  HASH JOIN           |            |   482M|   229G|    27M|  9176K  (1)| 30:35:13 |
|   2 |   TABLE ACCESS FULL  | PARTS      |   200K|    25M|       |  1051   (1)| 00:00:13 |
|*  3 |   HASH JOIN          |            |   486M|   171G|   118M|   168K  (2)| 00:33:39 |
|   4 |    TABLE ACCESS FULL | PARTSUPPS  |   800K|   109M|       |  4526   (1)| 00:00:55 |
|*  5 |    HASH JOIN         |            |  6086K|  1369M|   175M| 84027   (1)| 00:16:49 |
|   6 |     TABLE ACCESS FULL| ORDERS     |  1500K|   158M|       |  6610   (1)| 00:01:20 |
|   7 |     TABLE ACCESS FULL| LINEITEMS  |  6001K|   715M|       | 29752   (1)| 00:05:58 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("PARTSUPPS"."PS_PARTKEY"="PARTS"."P_PARTKEY")
   3 - access("LINEITEMS"."L_SUPPKEY"="PARTSUPPS"."PS_SUPPKEY")
   5 - access("ORDERS"."O_ORDERKEY"="LINEITEMS"."L_ORDERKEY")
```

Das Ergebnis bei Bushy Tree:

Ausführungsplan

```
-------------------------------------------------------------------------------------------
| Id  | Operation            | Name       | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |            |   482M|   286G|       |   211K  (2)| 00:42:23 |
|*  1 |  HASH JOIN           |            |   482M|   286G|   234M|   211K  (2)| 00:42:23 |
|   2 |   VIEW               |            |   792K|   225M|       | 12812   (1)| 00:02:34 |
|*  3 |    HASH JOIN         |            |   792K|   207M|    27M| 12812   (1)| 00:02:34 |
|   4 |     TABLE ACCESS FULL| PARTS      |   200K|    25M|       |  1051   (1)| 00:00:13 |
|   5 |     TABLE ACCESS FULL| PARTSUPPS  |   800K|   109M|       |  4526   (1)| 00:00:55 |
|   6 |   VIEW               |            |  6086K|  1967M|       | 84027   (1)| 00:16:49 |
|*  7 |    HASH JOIN         |            |  6086K|  1369M|   175M| 84027   (1)| 00:16:49 |
|   8 |     TABLE ACCESS FULL| ORDERS     |  1500K|   158M|       |  6610   (1)| 00:01:20 |
|   9 |     TABLE ACCESS FULL| LINEITEMS  |  6001K|   715M|       | 29752   (1)| 00:05:58 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("L_SUPPKEY"="PS_SUPPKEY")
   3 - access("PARTSUPPS"."PS_PARTKEY"="PARTS"."P_PARTKEY")
   7 - access("ORDERS"."O_ORDERKEY"="LINEITEMS"."L_ORDERKEY")
```

Reflexion
Der Optimizer hatte sowohl bei Left Deep Join, als auch beim Bushy Tree keine Indizes verwendet. Somit konnte auch kein Performance-Zuwachs feststellt werden.

Mit den folgenden Statements wurde versucht, einen Fast Full Index-Scan zu erzwingen. Jedoch wurde das vom Optimizer ebenfalls ignoriert.

Left Deep Join mit Hint

SQL-Query

```
SELECT /*+
          INDEX_FFS(orders O_ORDERKEY_IX)
          INDEX_FFS(lineitems L_ORDERKEY_IX)
          INDEX_FFS(lineitems L_SUPPKEY_IX)
          INDEX_FFS(partsupps PS_PARTKEY_IX)
          INDEX_FFS(partsupps PS_SUPPKEY_IX)
          INDEX_FFS(parts P_PARTKEY_IX)*/ *
FROM orders, lineitems, partsupps, parts
WHERE orders.o_orderkey = lineitems.l_orderkey
AND lineitems.l_suppkey = partsupps.ps_suppkey
AND partsupps.ps_partkey = parts.p_partkey;
```

Bushy Tree mit Hint

SQL-Query

```
SELECT *
FROM
(
SELECT /*+ no_merge INDEX_FFS(order O_ORDERKEY_IX) INDEX_FFS(lineitems L_ORDERKEY_IX) */ *
FROM orders, lineitems
WHERE orders.o_orderkey = lineitems.l_orderkey
)
,
(
SELECT /*+ no_merge INDEX_FFS(partsupps PS_PARTKEY_IX) INDEX_FFS(parts P_PARTKEY_IX) */ *
FROM partsupps, parts
WHERE partsupps.ps_partkey = parts.p_partkey
)
WHERE l_suppkey = ps_suppkey;
```

Folgend wurden einige Versuche unternommen, um Statements zu bilden, damit der Optimizer Indizes wieder zulässt. Dabei wurden die Spalten geändert, über die gejoint werden:

SQL-Query

```
1 SELECT *
2 FROM orders, lineitems, partsupps, parts
3 WHERE orders.o_orderkey = lineitems.l_orderkey
4 AND lineitems.l_orderkey = partsupps.ps_suppkey
5 AND partsupps.ps_suppkey = parts.p_partkey;
```

Mit Index

Ausführungsplan

```
-------------------------------------------------------------------------------------------
| Id  | Operation             | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |           |  3292K|  1604M|       |   170K  (1)| 00:34:12 |
|*  1 |  HASH JOIN            |           |  3292K|  1604M|   175M|   170K  (1)| 00:34:12 |
|   2 |   TABLE ACCESS FULL   | ORDERS    |  1500K|   158M|       |  6610   (1)| 00:01:20 |
|*  3 |   HASH JOIN           |           |  3246K|  1238M|   218M| 92355   (1)| 00:18:29 |
|*  4 |    HASH JOIN          |           |   800K|   209M|    27M| 12812   (1)| 00:02:34 |
|   5 |     TABLE ACCESS FULL| PARTS     |   200K|    25M|       |  1051   (1)| 00:00:13 |
|   6 |     TABLE ACCESS FULL| PARTSUPPS |   800K|   109M|       |  4526   (1)| 00:00:55 |
|   7 |    TABLE ACCESS FULL | LINEITEMS |  6001K|   715M|       | 29752   (1)| 00:05:58 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("ORDERS"."O_ORDERKEY"="LINEITEMS"."L_ORDERKEY")
   3 - access("LINEITEMS"."L_ORDERKEY"="PARTSUPPS"."PS_SUPPKEY")
   4 - access("PARTSUPPS"."PS_SUPPKEY"="PARTS"."P_PARTKEY")
```

Ohne Index

Ausführungsplan

```
-------------------------------------------------------------------------------------------
| Id  | Operation             | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |           |  3292K|  1604M|       |   170K  (1)| 00:34:12 |
|*  1 |  HASH JOIN            |           |  3292K|  1604M|   175M|   170K  (1)| 00:34:12 |
|   2 |   TABLE ACCESS FULL   | ORDERS    |  1500K|   158M|       |  6610   (1)| 00:01:20 |
|*  3 |   HASH JOIN           |           |  3246K|  1238M|   218M| 92355   (1)| 00:18:29 |
|*  4 |    HASH JOIN          |           |   800K|   209M|    27M| 12812   (1)| 00:02:34 |
|   5 |     TABLE ACCESS FULL| PARTS     |   200K|    25M|       |  1051   (1)| 00:00:13 |
|   6 |     TABLE ACCESS FULL| PARTSUPPS |   800K|   109M|       |  4526   (1)| 00:00:55 |
|   7 |    TABLE ACCESS FULL | LINEITEMS |  6001K|   715M|       | 29752   (1)| 00:05:58 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("ORDERS"."O_ORDERKEY"="LINEITEMS"."L_ORDERKEY")
   3 - access("LINEITEMS"."L_ORDERKEY"="PARTSUPPS"."PS_SUPPKEY")
   4 - access("PARTSUPPS"."PS_SUPPKEY"="PARTS"."P_PARTKEY")
```

SQL-Query

```
SELECT *
FROM
(
SELECT /*+ no_merge */ *
FROM orders, lineitems
WHERE orders.o_orderkey = lineitems.l_orderkey
)
,
(
SELECT /*+ no_merge */ *
FROM partsupps, parts
WHERE partsupps.ps_suppkey = parts.p_partkey
)
WHERE lineitems.l_orderkey = partsupps.ps_suppkey;
```

Mit Index

Ausführungsplan

```
-------------------------------------------------------------------------------------------
| Id  | Operation             | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |           |  3292K|    65M|       | 54044   (1)| 00:10:49 |
|*  1 |  HASH JOIN            |           |  3292K|    65M|    25M| 54044   (1)| 00:10:49 |
|   2 |   TABLE ACCESS FULL   | ORDERS    |  1500K|  8789K|       |  6599   (1)| 00:01:20 |
|*  3 |   HASH JOIN           |           |  3246K|    46M|    16M| 41981   (1)| 00:08:24 |
|*  4 |    HASH JOIN          |           |   800K|  7031K|  3328K|  6351   (1)| 00:01:17 |
|   5 |     TABLE ACCESS FULL| PARTS     |   200K|   976K|       |  1050   (1)| 00:00:13 |
|   6 |     TABLE ACCESS FULL| PARTSUPPS |   800K|  3125K|       |  4523   (1)| 00:00:55 |
|   7 |    TABLE ACCESS FULL | LINEITEMS |  6001K|    34M|       | 29663   (1)| 00:05:56 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("ORDERS"."O_ORDERKEY"="LINEITEMS"."L_ORDERKEY")
   3 - access("LINEITEMS"."L_ORDERKEY"="PARTSUPPS"."PS_SUPPKEY")
   4 - access("PARTSUPPS"."PS_SUPPKEY"="PARTS"."P_PARTKEY")
```

Ohne Index

Ausführungsplan

```

-------------------------------------------------------------------------------------------
| Id  | Operation             | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |           |  3292K|    65M|       | 54044   (1)| 00:10:49 |
|*  1 |  HASH JOIN            |           |  3292K|    65M|    25M| 54044   (1)| 00:10:49 |
|   2 |   TABLE ACCESS FULL   | ORDERS    |  1500K|  8789K|       |  6599   (1)| 00:01:20 |
|*  3 |   HASH JOIN           |           |  3246K|    46M|    16M| 41981   (1)| 00:08:24 |
|*  4 |    HASH JOIN          |           |   800K|  7031K|  3328K|  6351   (1)| 00:01:17 |
|   5 |     TABLE ACCESS FULL| PARTS     |   200K|   976K|       |  1050   (1)| 00:00:13 |
|   6 |     TABLE ACCESS FULL| PARTSUPPS |   800K|  3125K|       |  4523   (1)| 00:00:55 |
|   7 |    TABLE ACCESS FULL | LINEITEMS |  6001K|    34M|       | 29663   (1)| 00:05:56 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("ORDERS"."O_ORDERKEY"="LINEITEMS"."L_ORDERKEY")
   3 - access("LINEITEMS"."L_ORDERKEY"="PARTSUPPS"."PS_SUPPKEY")
   4 - access("PARTSUPPS"."PS_SUPPKEY"="PARTS"."P_PARTKEY")
```

SQL-Query

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

Mit Index

Ausführungsplan

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                | Name          | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |               |  3292K|    65M|       | 17611   (2)| 00:03:32 |
|*  1 |  HASH JOIN               |               |  3292K|    65M|    25M| 17611   (2)| 00:03:32 |
|   2 |   INDEX FAST FULL SCAN   | O_ORDERKEY_IX |  1500K|  8789K|       |   965   (2)| 00:00:12 |
|*  3 |   HASH JOIN              |               |  3246K|    46M|    16M| 11181   (2)| 00:02:15 |
|*  4 |    HASH JOIN             |               |   800K|  7031K|  3328K|  1360   (2)| 00:00:17 |
|   5 |     INDEX FAST FULL SCAN| P_PARTKEY_IX  |   200K|   976K|       |   123   (1)| 00:00:02 |
|   6 |     INDEX FAST FULL SCAN| PS_SUPPKEY_IX |   800K|  3125K|       |   459   (2)| 00:00:06 |
|   7 |    INDEX FAST FULL SCAN | L_ORDERKEY_IX |  6001K|    34M|       |  3854   (2)| 00:00:47 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("ORDERS"."O_ORDERKEY"="LINEITEMS"."L_ORDERKEY")
   3 - access("LINEITEMS"."L_ORDERKEY"="PARTSUPPS"."PS_SUPPKEY")
   4 - access("PARTSUPPS"."PS_SUPPKEY"="PARTS"."P_PARTKEY")
```

Ohne Index

Ausführungsplan

```
-----------------------------------------------------------------------------------------------
| Id  | Operation              | Name       | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |            |  3292K|    65M|       | 54044   (1)| 00:10:49 |
|*  1 |  HASH JOIN             |            |  3292K|    65M|    25M| 54044   (1)| 00:10:49 |
|   2 |   TABLE ACCESS FULL    | ORDERS     |  1500K|  8789K|       |  6599   (1)| 00:01:20 |
|*  3 |   HASH JOIN            |            |  3246K|    46M|    16M| 41981   (1)| 00:08:24 |
|*  4 |    HASH JOIN           |            |   800K|  7031K|  3328K|  6351   (1)| 00:01:17 |
|   5 |     TABLE ACCESS FULL| PARTS      |   200K|   976K|       |  1050   (1)| 00:00:13 |
|   6 |     TABLE ACCESS FULL| PARTSUPPS  |   800K|  3125K|       |  4523   (1)| 00:00:55 |
|   7 |    TABLE ACCESS FULL   | LINEITEMS  |  6001K|    34M|       | 29663   (1)| 00:05:56 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("ORDERS"."O_ORDERKEY"="LINEITEMS"."L_ORDERKEY")
   3 - access("LINEITEMS"."L_ORDERKEY"="PARTSUPPS"."PS_SUPPKEY")
   4 - access("PARTSUPPS"."PS_SUPPKEY"="PARTS"."P_PARTKEY")
```

SQL-Query

```

SELECT o_orderkey, l_orderkey, ps_suppkey, p_partkey
FROM
(
SELECT /*+ no_merge */ o_orderkey, l_orderkey
FROM orders, lineitems
WHERE orders.o_orderkey = lineitems.l_orderkey
)
,
(
SELECT /*+ no_merge */ ps_suppkey, p_partkey
FROM partsupps, parts
WHERE partsupps.ps_suppkey = parts.p_partkey
)
WHERE lineitems.l_orderkey = partsupps.ps_suppkey;
```

Mit Index

Ausführungsplan

```
-------------------------------------------------------------------------------------------
| Id  | Operation            | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |           |  3292K|    65M|       | 54044   (1)| 00:10:49 |
|*  1 |  HASH JOIN           |           |  3292K|    65M|    25M| 54044   (1)| 00:10:49 |
|   2 |   TABLE ACCESS FULL  | ORDERS    |  1500K|  8789K|       |  6599   (1)| 00:01:20 |
|*  3 |   HASH JOIN          |           |  3246K|    46M|    16M| 41981   (1)| 00:08:24 |
|*  4 |    HASH JOIN         |           |   800K|  7031K|  3328K|  6351   (1)| 00:01:17 |
|   5 |     TABLE ACCESS FULL| PARTS     |   200K|   976K|       |  1050   (1)| 00:00:13 |
|   6 |     TABLE ACCESS FULL| PARTSUPPS |   800K|  3125K|       |  4523   (1)| 00:00:55 |
|   7 |    TABLE ACCESS FULL | LINEITEMS |  6001K|    34M|       | 29663   (1)| 00:05:56 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("ORDERS"."O_ORDERKEY"="LINEITEMS"."L_ORDERKEY")
   3 - access("LINEITEMS"."L_ORDERKEY"="PARTSUPPS"."PS_SUPPKEY")
   4 - access("PARTSUPPS"."PS_SUPPKEY"="PARTS"."P_PARTKEY")
```

Ohne Index

Ausführungsplan

```
-------------------------------------------------------------------------------------------
| Id  | Operation            | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |           |  3292K|    65M|       | 54044   (1)| 00:10:49 |
|*  1 |  HASH JOIN           |           |  3292K|    65M|    25M| 54044   (1)| 00:10:49 |
|   2 |   TABLE ACCESS FULL  | ORDERS    |  1500K|  8789K|       |  6599   (1)| 00:01:20 |
|*  3 |   HASH JOIN          |           |  3246K|    46M|    16M| 41981   (1)| 00:08:24 |
|*  4 |    HASH JOIN         |           |   800K|  7031K|  3328K|  6351   (1)| 00:01:17 |
|   5 |     TABLE ACCESS FULL| PARTS     |   200K|   976K|       |  1050   (1)| 00:00:13 |
|   6 |     TABLE ACCESS FULL| PARTSUPPS |   800K|  3125K|       |  4523   (1)| 00:00:55 |
|   7 |    TABLE ACCESS FULL | LINEITEMS |  6001K|    34M|       | 29663   (1)| 00:05:56 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("ORDERS"."O_ORDERKEY"="LINEITEMS"."L_ORDERKEY")
   3 - access("LINEITEMS"."L_ORDERKEY"="PARTSUPPS"."PS_SUPPKEY")
   4 - access("PARTSUPPS"."PS_SUPPKEY"="PARTS"."P_PARTKEY")
```

Reflexion

Bei Bushy Trees werden die Indizes ignoriert. Einzig bei einem Left Deep Join gelang es, durch Indizes einen Performance-Zuwachs zu erreichen.

Bei diesem Versuch wurde darauf geachtet, dass die Spalte, über die auf der rechten Seite gejoint wurde, beim nächsten Join auf der linken Seite verwendet wurde.

8 Eigene SQL-Anfrage

Wir haben zwei neue Tabellen gemäss folgenden Angaben erstellt.

SQL-Query

```
1 CREATE TABLE test1
2 AS SELECT *
3 FROM DBARC00.orders;
4
5 CREATE TABLE test2
6 AS SELECT *
7 FROM DBARC00.orders;
```

Nun haben wir eine eigene SQL-Anfrage mit diesen zwei Tabellen erstellt.

SQL-Query

```
1 SELECT *
2 FROM test1 t1, test2 t2
3 WHERE t1.o_orderkey = t2.o_orderkey
4 AND t1.o_orderkey BETWEEN 2345 AND 2543
5 ORDER BY t1.o_orderkey;
```

Ausführungsplan

```
-------------------------------------------------------------------------------
| Id  | Operation           | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |       |   266 | 73948 | 13200   (1)| 00:02:39 |
|   1 |  SORT ORDER BY      |       |   266 | 73948 | 13200   (1)| 00:02:39 |
|*  2 |   HASH JOIN         |       |   266 | 73948 | 13199   (1)| 00:02:39 |
|*  3 |    TABLE ACCESS FULL| TEST2 |   267 | 37113 |  6599   (1)| 00:01:20 |
|*  4 |    TABLE ACCESS FULL| TEST1 |   267 | 37113 |  6600   (1)| 00:01:20 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("T1"."O_ORDERKEY"="T2"."O_ORDERKEY")
   3 - filter("T2"."O_ORDERKEY">=2345 AND "T2"."O_ORDERKEY"<=2543)
   4 - filter("T1"."O_ORDERKEY">=2345 AND "T1"."O_ORDERKEY"<=2543)
```

Da auf den Tabellen kein Index besteht, muss ein Full Table Scan durchgeführt werden, was die hohen Kosten verursachen. Sowohl das joinen, sortieren wie auch das projizieren verursachen wenig kosten, da das Resultat nur auf wenige Anzahl Zeilen geschätzt wird.

Als Gegenmassnahme haben wir zwei Indizes erstellt, jeweils auf die Spalte, auf die wir selektieren.

SQL-Query

```
1 CREATE INDEX test1_oorderkey_idx ON test1(o_orderkey);
2 CREATE INDEX test2_oorderkey_idx ON test2(o_orderkey);
```

Ausführungsplan

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                     |    48 | 13344 |    11  (10)| 00:00:01 |
|   1 |  MERGE JOIN                   |                     |    48 | 13344 |    11  (10)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID | TEST1               |    48 |  6672 |     5   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN           | TEST1_OORDERKEY_IDX |    48 |       |     3   (0)| 00:00:01 |
|*  4 |   SORT JOIN                   |                     |    48 |  6672 |     6  (17)| 00:00:01 |
|   5 |    TABLE ACCESS BY INDEX ROWID| TEST2               |    48 |  6672 |     5   (0)| 00:00:01 |
|*  6 |     INDEX RANGE SCAN          | TEST2_OORDERKEY_IDX |    48 |       |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access("T1"."O_ORDERKEY">=2345 AND "T1"."O_ORDERKEY"<=2543)
   4 - access("T1"."O_ORDERKEY"="T2"."O_ORDERKEY")
       filter("T1"."O_ORDERKEY"="T2"."O_ORDERKEY")
   6 - access("T2"."O_ORDERKEY">=2345 AND "T2"."O_ORDERKEY"<=2543)
```

Durch die Indizes werden zwei Index Range Scan durchgeführt, welche viel weniger Kosten verursachen. Auch im Vergleich zur ersten Variante, wird nur eine Tabelle sortiert und nachher erst gejoint. Schlussendlich wird die Projektion ausgeführt.

Dies für zu einer Kostenersparnis vom Faktor 1200.

9 Reflexion

9.1 Allgemein

Über die ganze Ausarbeitung hinweg konnten wir uns mit SQL-Queries und Ausführungsplänen vertieft auseinandersetzen. Dabei haben wir gelernt, wie der Optimizer mithilfe von Hints und Indizes sowohl positiv als auch negativ beeinflusst werden kann.

9.2 Anmeldung

Wir hatten zuerst Probleme uns auf dem DB-Server anzumelden, da wir unser Passwort nicht wussten. So probierten wir die uns bekannten Passwörter aus, was schliesslich dazu führte, dass unser Account gesperrt wurde, da zu viele Anmeldeanfragen durchgeführt wurden. Da wir auch keinen SSH-Zugriff auf diesen Server hatten, beauftragten wir eine andere Gruppe unser Konto zu aktivieren.

SQL-Query

```
1 ALTER USER dbarc01 ACCOUNT UNLOCK;
```

Sie konnten dies jedoch nicht vornehmen, da unsere Accounts zu wenig Rechte haben. Schlussendlich haben wir dem Dozenten, Herr Wyss, geschrieben, dass er uns freischalten könnte. Damit wir trotzdem arbeiten konnten, bis wir eine Antwort vom Dozenten erhielten, durften wir den Account der Gruppe dbarc03 verwenden.

9.3 Indizes

Durch die Anwendung von Indizes konnten wir feststellen, dass die Performance stark verbessert werden konnte.