

Datenbank-Architektur für Fortgeschrittene

Ausarbeitung 1: Anfrageverarbeitung

Thomas Baumann / Egemen Kaba

01.05.2013

Inhaltsverzeichnis

1	Einleitung	1
1.1	Sonderzächen	1
2	Vorbereitung	1
2.1	Einrichten Datenbasis	1
2.2	Tabellenstatistik	1
3	Ausführungsplan	2
4	Versuche ohne Index	2
4.1	Projektion	2
4.2	Selektion	3
4.3	Join	6
5	Versuch mit Index	8
5.1	Projektion	8
5.2	Selektion	8
5.3	Join	12
6	Quiz	12
7	Deep Left Join	12
8	Eigene SQL-Anfragen	13

1 Einleitung

1.1 Sonderzächön

sind mäggö cöäl! Hallo "Maxünd "Moritz"!

2 Vorbereitung

2.1 Einrichten Datenbasis

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

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.

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. Aus der Statistik geht hervor, dass diese Tabelle 1579K Zeilen umfasst, 209MB gross ist und ein Full Table Scan 6612 CPU-Indexpunkte beansprucht.

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. Zudem ist die benötigte CPU-Leistung minimal weniger, aufgrund der Reduktion des zu verwaltenden Speichers.

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 vorangehend alle doppelten Einträge herausgefiltert werden, was die massiv angestiegenen CPU-Kosten bei Id 0 und 1 erklärt. Die zusätzliche Operation mit der Id 1 dient dazu, die doppelten Werte herauszufiltern. 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, wo der Eintrag liegt 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 die 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 gesunken, dies ist aus unserer Sicht dadurch zu Begründen, da bei beiden Vergleichen, die gleiche Spalte verwendet wird.

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 wieder 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 Performanteste aus. Somit lautet die Antwort auf die

Frage: Nein. Im Folgenden werden einige Gründe aufgeführt: Es wird ein Fremdschlüssel von orders und der Primärschlüssel von customers verglichen. Dadurch können in beiden Tabellen nicht benötigte Tupel vorab herausgefiltert werden. Dies hat den Vorteil, dass weniger Speicher und CPU für weitere Operationen benötigt werden. Danach werden die restlichen Tupel der Tabelle orders mit der zweiten Bedingung herausgefiltert. Wenn nun die minimalst mögliche Anzahl Tupel erreicht worden sind, wird auf die Tabelle customers zugegriffen und mit der Tabelle orders gejoint.

5 Versuche mit Index

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 Access wird jetzt ein Index Range Scan angewendet. Dadurch können die benötigten Einträge wesentlich schneller gefunden werden. Konkret werden im Vergleich zum Versuch ohne Index über weniger Reihen iteriert, weniger Platz im Memory sowie kein temporärer Speicher mehr benötigt, weniger CPU beansprucht und das Query wird deutlich schneller abgearbeitet.

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 Scan 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 Access, 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

Bevor hier der Index Range Scan auf die Spalte orderkey angewendet wird, werden die Einträge nach der Spalte clerk gefiltert. Dies geschieht deswegen, weil ein Tuppel Kriterien in zwei Spalten erfüllen muss.

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

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 wurde nun ein Full Table Scan ausgeführt, weil der Range gross genug gewählt wurde, dass es wegen der vielen IO-Zugriffe nicht mehr lohnt.

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

5.3 Join

6 Quiz

7 Deep Left Join

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

SQL-Query

```
1 SELECT *
2 FROM orders, lineitems, partsupp, parts
3 WHERE orders.o_orderkey = lineitems.l_orderkey
4 AND lineitems.l_suppkey = partsupp.ps_suppkey
5 AND partsupp.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	PARTSUPP	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("PARTSUPP"."PS_PARTKEY"="PARTS"."P_PARTKEY")
- 3 - access("LINEITEMS"."L_SUPPKEY"="PARTSUPP"."PS_SUPPKEY")
- 5 - access("ORDERS"."O_ORDERKEY"="LINEITEMS"."L_ORDERKEY")

Modifiziertes Statement, um ein Bushy Tree zu erzeugen:

SQL-Query

```
1 SELECT *
2 FROM
3 (
4 SELECT /*+ no_merge */ *
5 FROM orders, lineitems
6 WHERE orders.o_orderkey = lineitems.l_orderkey
7 )
8 ,
9 (
10 SELECT /*+ no_merge */ *
11 FROM partsupp, parts
12 WHERE partsupp.ps_partkey = parts.p_partkey
13 )
14 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

8 Eigene SQL-Anfragen