

Datenbank-Architektur für Fortgeschrittene

Ausarbeitung 1: Anfrageverarbeitung

Thomas Baumann / Egemen Kaba

01.05.2013

Inhaltsverzeichnis

1	Einleitung	1
1.1	Sonderzächön	1
2	Vorbereitung	1
2.1	Einrichten Datenbasis	1
3	Ausführungsplan	1
4	Versuche ohne Index	2
4.1	Projektion	2
4.2	Selektion	3
4.3	Join	5

1 Einleitung

1.1 Sonderzächön

sind mägö cöäl! Hallo "Maxünd "Moritz"!

2 Vorbereitung

2.1 Einrichten Datenbasis

```
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;
```

3 Ausführungsplan

```
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'));
```

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 Zeilen, deren Grösse und die Kosten für die Teilschritte. Die Kosten sind immer aufsummiert von unten nach oben. Sie werden berechnet aus Disk I/O Zugriffen, CPU Belastung und Hauptspeicherverbrauch. Der CPU Verbrauch wird immer gerundet, womit wir die Differenzen nicht genau sehen. 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

Statement

```
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.

Statement

```
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.

Statement

```
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

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

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)
```

Es wird nur ein Tupel selektioniert, jedoch ist die Spalte „id“ nicht indexiert, somit weiss die Datenbank nicht, wo genau sich die Datensätze auf der Disk befinden. Da die „id“ nicht eindeutig ist, kann beim ersten gefundenen Datensatz nicht abgebrochen werden. Daraus resultiert wieder ein Full Table Scan. Die Kosten sind dank der Bedingung ein wenig kleiner, da viel weniger Hauptspeicher benötigt wird.

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

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')
```

Die Kosten sind hier durch den zusätzlichen Vergleich, welcher eine höhere CPU Belastung verursacht, minimal gestiegen.

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

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')

Die CPU-Kosten sind hier wieder kleiner, da aufgrund der AND-Verknüpfung die zweite Bedingung nur überprüft werden muss, wenn die erste Bedingung erfüllt ist.

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

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')

Wenn man das Statement anschaut sollte die CPU Belastung höher sein, da eine zusätzliche Operation pro Tupel notwendig ist ($o_{orderkey} * 2$). Die Kosten sind jedoch genau gleich wie beim vorherigen Statement. Wir vermuten, dass dies optimiert wird zu:

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

Dadurch würde die Operation nur einmal ausgeführt werden.

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

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)

TODO CHECK Die Kosten sind kleiner als oben, obwohl laut Ausführungsplan die CPU Belastung gleich ist und bei beiden ein Full Table Scan durchgeführt werden muss. Der Hauptspeicher Verbrauch ist sogar grösser, woraus höhere Kosten resultieren müssten. Diese Differenz kommt durch die Rundung der CPU Angabe. Wir haben mit folgendem Statement den CPU Verbrauch genauer angezeigt: `SELECT CPU_COST FROM PLAN_TABLE`; Dadurch haben wir festgestellt, dass der CPU Verbrauch tatsächlich kleiner ist als beim obigen Statement. Dies kommt daher, weil nur die Spalte „id“ verglichen werden muss anstatt „id“ und „provider“.

```
1 SELECT *
2 FROM orders
3 WHERE o_orderkey BETWEEN 44444 AND 55555
```

```
4 AND o_clerk BETWEEN 'Clerk#000000130' AND 'Clerk#000000139';
```

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')
```

TODO

4.3 Join

Statement

```
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		267	80901	7555 (1)	00:01:31
* 1	HASH JOIN		267	80901	7555 (1)	00:01:31
* 2	TABLE ACCESS FULL	ORDERS	267	37113	6603 (1)	00:01:20
3	TABLE ACCESS FULL	CUSTOMERS	131K	20M	951 (1)	00:00:12

Varianten

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

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

```
1 SELECT *
2 FROM customers, orders
3 WHERE o_custkey = c_custkey
4 AND 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 gejoinet.