Datenbank-Architektur für Fortgeschrittene

Ausarbeitung 1: Anfrageverarbeitung

Thomas Baumann / Egemen Kaba

01.05.2013

Inhaltsverzeichnis

1 Einleitung

1.1 Sonderzächön

sind mägö cöäl! Hallo "Maxünd "Moritz"!

2 Vorbereitung

2.1 Einrichten Datenbasis

SQL-Query

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

CREATE TABLE nations
AS SELECT *
  FROM dbarc00.nations;

CREATE TABLE parts
AS SELECT *
  FROM dbarc00.parts;

CREATE TABLE customers
AS SELECT *
  FROM dbarc00.customers;

CREATE TABLE suppliers
AS SELECT *
  FROM dbarc00.suppliers;

CREATE TABLE orders
AS SELECT *
  FROM dbarc00.orders;

CREATE TABLE partsupps
AS SELECT *
  FROM dbarc00.partsupps;

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

3 Ausführungsplan

SQL-Query

```
EXPLAIN PLAN FOR
SELECT *
FROM parts;

SELECT plan_table_output
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 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 Hauptspeicher verbrauch. 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

SQL-Query

```sql
SELECT *
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

```sql
SELECT o_clerk
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

```sql
SELECT DISTINCT o_clerk
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

```sql
SELECT *
FROM orders
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

```sql
SELECT *
FROM orders
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

```sql
SELECT *
FROM orders
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

```sql
SELECT *
FROM orders
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:

```sql
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

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

SQL-Query

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

SQL-Query

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

SQL-Query

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

Reflexion

Der Optimizer wählt bei allen Varianten automatisch die Perfomanteste 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 Tuppel vorab herausgefiltert werden. Dies hat den Vorteil, dass weniger Speicher und CPU für weitere Operationen benötigt werden. Danach werden die restlichen Tuppel der Tabelle orders mit der zweiten Bedingung herausgefiltert. Wenn nun die minimalst mögliche Anzahl Tuppel erreicht worden sind, wird auf die Tabelle customers zugegriffen und mit der Tabelle orders gejoint.