DBARC Lab09SQL Tuning

short line

Florian Siffer  
Simon Wächter  
Lukas Willin  
13.05.2018

# Inhaltsverzeichnis

[**Inhaltsverzeichnis**](#_30j0zll) **1**

[**Statistiken**](#_1fob9te) **4**

[Vorgehensweise](#_3znysh7) 4

[Ergebnisse](#_2et92p0) 4

[**Versuche ohne Index**](#_3dy6vkm) **6**

[Vorgehensweise](#_1t3h5sf) 6

[Projektion](#_4d34og8) 6

[SELECT \*](#_2s8eyo1) 6

[Abfrage](#_17dp8vu) 6

[Plan](#_3rdcrjn) 6

[Beobachtungen](#_26in1rg) 6

[SELECT Attribute](#_lnxbz9) 6

[Abfrage](#_35nkun2) 6

[Plan](#_1ksv4uv) 7

[Beobachtungen](#_44sinio) 7

[SELECT DISTINCT](#_2jxsxqh) 7

[Abfrage](#_z337ya) 7

[Plan](#_3j2qqm3) 7

[Beobachtungen](#_1y810tw) 7

[Selektion](#_4i7ojhp) 8

[Exact Point Query](#_2xcytpi) 8

[Abfrage](#_1ci93xb) 8

[Plan](#_3whwml4) 8

[Beobachtungen](#_2bn6wsx) 8

[Partial Point Query](#_qsh70q) 8

[Abfragen](#_3as4poj) 8

[Pläne](#_1pxezwc) 9

[Beobachtungen](#_49x2ik5) 10

[Range Query](#_2p2csry) 10

[Abfragen](#_147n2zr) 10

[Pläne](#_23ckvvd) 11

[Beobachtungen](#_ihv636) 12

[Partial Range Query](#_32hioqz) 12

[Abfrage](#_1hmsyys) 12

[Plan](#_41mghml) 12

[Beobachtungen](#_2grqrue) 13

[Join](#_vx1227) 13

[Versuchsreihe](#_3fwokq0) 13

[Abfrage 1](#_1v1yuxt) 13

[Plan 1](#_4f1mdlm) 13

[Beobachtungen 1](#_2u6wntf) 13

[Abfrage 2](#_19c6y18) 14

[Plan 2](#_3tbugp1) 14

[Beobachtungen 2](#_28h4qwu) 14

[Beobachtungen](#_nmf14n) 14

[**Versuche mit Index**](#_1mrcu09) **15**

[Erstellung](#_46r0co2) 15

[Projektion](#_2lwamvv) 15

[Abfrage 1](#_111kx3o) 15

[Plan 1](#_3l18frh) 15

[Beobachtungen 1](#_206ipza) 15

[Selektion](#_4k668n3) 15

[Exact Point Query](#_2zbgiuw) 15

[Abfrage 1](#_1egqt2p) 15

[Plan 1](#_3ygebqi) 15

[Beobachtungen 1](#_2dlolyb) 16

[Full Table Scan](#_sqyw64) 16

[Abfrage 2](#_3cqmetx) 16

[Plan 2](#_1rvwp1q) 16

[Beobachtungen 2](#_4bvk7pj) 16

[Partial Point Query](#_2r0uhxc) 17

[Abfrage 3](#_1664s55) 17

[Plan 3](#_3q5sasy) 17

[Beobachtungen 3](#_25b2l0r) 17

[Abfrage 4](#_kgcv8k) 17

[Plan 4](#_34g0dwd) 17

[Beobachtungen 4](#_1jlao46) 18

[Abfrage 5](#_43ky6rz) 18

[Plan 5](#_2iq8gzs) 18

[Beobachtungen 5](#_xvir7l) 18

[Range Query](#_3hv69ve) 18

[Abfrage 6](#_1x0gk37) 18

[Plan 6](#_4h042r0) 19

[Beobachtungen 6](#_2w5ecyt) 19

[Abfrage-Plan 7](#_1baon6m) 19

[Beobachtungen Range Query](#_3vac5uf) 19

[Partial Range Query](#_2afmg28) 19

[Abfrage 7](#_pkwqa1) 19

[Plan 7](#_39kk8xu) 19

[Beobachtungen 7](#_1opuj5n) 20

[Join](#_48pi1tg) 20

[Abfrage 1](#_2nusc19) 20

[Plan 1](#_1302m92) 20

[Beobachtungen 1](#_3mzq4wv) 21

[Abfrage 2](#_2250f4o) 21

[Plan 2](#_haapch) 21

[Beobachtungen 2](#_319y80a) 21

[Abfrage 3](#_1gf8i83) 21

[Plan 3](#_40ew0vw) 21

[Beobachtungen 3](#_2fk6b3p) 22

[Abfrage 4](#_upglbi) 22

[Plan 4](#_3ep43zb) 22

[Beobachtungen 4](#_1tuee74) 22

[Beobachtungen](#_4du1wux) 22

[**Ausführungsplan Quiz**](#_184mhaj) **24**

[Ausgangszustand](#_3s49zyc) 24

[Abfrage](#_279ka65) 24

[Plan](#_meukdy) 24

[Beobachtungen](#_36ei31r) 25

[Bestes Ergebnis](#_1ljsd9k) 25

[Vorgehen](#_45jfvxd) 25

[Indexe & Abfrage](#_2koq656) 25

[Neuer Plan](#_zu0gcz) 26

[Entfernte Indexe & Funktionen](#_3jtnz0s) 27

[**Deep Left Join**](#_1yyy98l) **28**

[Vorinformation](#_4iylrwe) 28

[Statements ohne Indizes](#_2y3w247) 29

[Statements mit Indizes](#_1d96cc0) 32

[Beobachtungen](#_3x8tuzt) 35

[**Eigene SQL Anfragen**](#_2ce457m) **36**

[Beispiel 1](#_rjefff) 36

[Ausgangslage](#_3bj1y38) 36

[Anpassung](#_1qoc8b1) 37

[Boebachtung](#_4anzqyu) 38

[Beispiel 2](#_2pta16n) 39

[Ausgangslage](#_14ykbeg) 39

[Anpassung](#_3oy7u29) 40

[Boebachtung](#_243i4a2) 41

[Fazit](#_j8sehv) 41

# Statistiken

## Vorgehensweise

Folgender Befehl ergibt die Anzahl Blöcke, Bytes und Extents pro Tabelle:

|  |
| --- |
| SELECT \* FROM user\_segments; |

Die Anzahl Zeilen kann jeweils mit folgendem Befehl gefunden werden, wobei ‘CUSTOMERS’ jeweils mit dem Tabellennamen ersetzt werden muss:

|  |
| --- |
| *SELECT COUNT(\*) FROM CUSTOMERS;* |

## Ergebnisse

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Anz. Zeilen | Anz. Bytes | Anz. Blöcke | Anz. Extents |
| CUSTOMERS | 150000 | 29360128 | 3584 | 43 |
| LINEITEMS | 6001215 | 897581056 | 109568 | 178 |
| NATIONS | 25 | 65536 | 8 | 1 |
| ORDERS | 1500000 | 201326592 | 24576 | 95 |
| PARTS | 200000 | 32505856 | 3968 | 46 |
| PARTSUPPS | 800000 | 137363456 | 16768 | 88 |
| REGIONS | 5 | 65536 | 8 | 1 |
| SUPPLIERS | 10000 | 2097152 | 256 | 17 |

### 

# Versuche ohne Index

## Vorgehensweise

Für jede der folgenden Abfragen wird jeweils mit folgenden Befehlen der Ausführungsplan generiert und angezeigt:

|  |
| --- |
| EXPLAIN PLAN FOR SELECT \* FROM parts; SELECT plan\_table\_output FROM TABLE(DBMS\_XPLAN.DISPLAY()); |

Wobei jeweils der Teil nach “EXPLAIN PLAN FOR” mit der jeweiligen Abfrage ersetzt wird.

## Projektion

### SELECT \*

#### Abfrage

|  |
| --- |
| SELECT \* FROM orders; |

#### Plan

|  |
| --- |
| Plan hash value: 1275100350  --------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | --------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 1500K| 158M| 6612 (1)| 00:00:01 | | 1 | TABLE ACCESS FULL| ORDERS | 1500K| 158M| 6612 (1)| 00:00:01 | --------------------------------------------------------------------------- |

#### Beobachtungen

Hier ist zu sehen, dass die gesamte Tabelle gelesen (TABLE ACCESS FULL) und ausgegeben wird. Und zwar ohne Filter - deshalb ist auch die Anzahl Rows (1500k) und der benötigte Speicherplatz (158M Bytes) entsprechend gross. Selbst mit einem Filter wären die CPU Kosten (6612) des ersten Schrittes gleich gross (6612).

### SELECT Attribute

#### Abfrage

|  |
| --- |
| SELECT o\_clerk FROM orders; |

#### Plan

|  |
| --- |
| Plan hash value: 1275100350 --------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | --------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 1500K| 22M| 6609 (1)| 00:00:01 | | 1 | TABLE ACCESS FULL| ORDERS | 1500K| 22M| 6609 (1)| 00:00:01 | --------------------------------------------------------------------------- |

#### Beobachtungen

Weiterhin muss die gesamte Tabelle gelesen werden - daher wird die gleiche Anzahl Zeilen abgefragt. Dahe sind auch die Kosten beinahe identisch. Da jedoch viel weniger Daten ausgegeben werden (nur eine Spalte anstatt alle) ist natürlich der benötigte Speicherplatz (Bytes) viel kleiner.

### SELECT DISTINCT

#### Abfrage

|  |
| --- |
| SELECT DISTINCT o\_clerk FROM orders; |

#### Plan

|  |
| --- |
| Plan hash value: 3394267636 --------------------------------------------------------------------------- |Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | --------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 1000 | 16000 | 6682 (2)| 00:00:01 | | 1 | HASH UNIQUE | | 1000 | 16000 | 6682 (2)| 00:00:01 | | 2 | TABLE ACCESS FULL| ORDERS | 1500K| 22M| 6609 (1)| 00:00:01 | --------------------------------------------------------------------------- |

#### Beobachtungen

Die erste Schritt (ID 2) entspricht der letzten Abfrage in allen Werten. Dann kommen im nächsten Schritt (ID 1) zusätzliche CPU Kosten dazu, denn Ergebnisse werden mit einem Hash verfahren auf Einzigartige gefiltert. Logischerweise nehmen in diesem Schritt die Anzahl Zeilen und somit der Speicherplatz (Bytes) ab.

## **Selektion**

### Exact Point Query

#### Abfrage

SELECT \* FROM orders WHERE o\_orderkey = 44480;

#### Plan

|  |
| --- |
| Plan hash value: 1275100350 ---------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ---------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 1 | 111 | 6604 (1)| 00:00:01 | |\* 1 | TABLE ACCESS FULL| ORDERS | 1 | 111 | 6604 (1)| 00:00:01 | ----------------------------------------------------------------------------   Predicate Information (identified by operation id): ---------------------------------------------------  1 - filter("O\_ORDERKEY"=44480) |

#### Beobachtungen

Da auf eine einzelne Zeile gefiltert wird, ist sowohl die Anzahl Rows (1) und der benötigte Speicherplatz (111 Bytes) von Anfang an sehr klein. Da aber die gesamte Tabelle durchsucht werden muss (ID 1), ist der CPU Aufwand dafür relativ gross (6604).

### Partial Point Query

Hier werden direkt drei Abfragen und Pläne miteinander verglichen.

#### Abfragen

|  |
| --- |
| SELECT \* FROM orders WHERE o\_custkey = 97303 OR o\_clerk = 'Clerk#000000860'; |

|  |
| --- |
| SELECT \* FROM orders WHERE o\_custkey = 97303 AND o\_clerk = 'Clerk#000000860'; |

|  |
| --- |
| SELECT \* FROM orders WHERE o\_custkey\*2 = 194606 AND o\_clerk = 'Clerk#000000286'; |

#### Pläne

|  |
| --- |
| Plan hash value: 1275100350  ---------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ---------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 1515 | 164K| 6632 (1)| 00:00:01 | |\* 1 | TABLE ACCESS FULL| ORDERS | 1515 | 164K| 6632 (1)| 00:00:01 | ----------------------------------------------------------------------------   Predicate Information (identified by operation id): ---------------------------------------------------   1 - filter("O\_CLERK"='Clerk#000000860' OR "O\_CUSTKEY"=97303) |

|  |
| --- |
| Plan hash value: 1275100350 ---------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ---------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 15 | 1665 | 6613 (1)| 00:00:01 | |\* 1 | TABLE ACCESS FULL| ORDERS | 15 | 1665 | 6613 (1)| 00:00:01 | ----------------------------------------------------------------------------   Predicate Information (identified by operation id): ---------------------------------------------------  1 - filter("O\_CUSTKEY"=97303 AND "O\_CLERK"='Clerk#000000860')   Note -----  - dynamic statistics used: dynamic sampling (level=2)  - 1 Sql Plan Directive used for this statement |

|  |
| --- |
| Plan hash value: 1275100350 ---------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ---------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 15 | 1665 | 6617 (1)| 00:00:01 | |\* 1 | TABLE ACCESS FULL| ORDERS | 15 | 1665 | 6617 (1)| 00:00:01 | ----------------------------------------------------------------------------   Predicate Information (identified by operation id): ---------------------------------------------------  1 - filter("O\_CUSTKEY"\*2=194606 AND "O\_CLERK"='Clerk#000000286') |

#### Beobachtungen

Auf den ersten Blick scheinen diese Abfragen nahezu identisch: für jede Anfrage muss ein TABLE ACCESS FULL mit einem Filter durchgeführt werden, was mit den entsprechenden Kosten bezügliche Speicherplatz und CPU verbunden ist.

Die Unterschiede liegen hier in den Filtern:

* Die erste Abfrage arbeitet mit einer OR Verknüpfung. Dies erklärt sowohl die grösse Anzahl an Rows (denn auf die OR Anfrage treffen natürlich in diesem Fall mehr Zeilen zu als auf die anderen AND Abfragen) und somit auch die grössere Menge an Speicherplatz die benötigt wird, als auch die höheren CPU Kosten (denn für jede Zeile müssen jeweils beide Bedingungen geprüft werden, im Unterschied zu den AND Abfragen, bei welchen die Zeile erledigt ist, wenn die erste Bedingungen negativ war).
* Wenn wir nun die letzten beiden Anfragen vergleichen erkennen wird identische Row und Speicherplatz Kennzahlen. Der einzige Unterschied liegt in den CPU Kosten. Wir erkennen, dass die letzte Anfrage nochmals leicht höhere CPU Kosten hat, als die Zweite. Dies ist mit der ersten Bedingung der letzten Abfrage zu erklären: und zwar muss für jede Zeile der O\_CUSTKEY mit 2 multipliziert werden. Es wäre besser den O\_CUSTKEY direkt zu verwenden und stattdessen die Zahl auf der anderen Seite des Vergleichs durch 2 zu teilen.

### Range Query

#### Abfragen

Hier werden erneut direkt mehrere Abfragen und Pläne miteinander verglichen.

|  |
| --- |
| SELECT \* FROM orders WHERE o\_orderkey BETWEEN 111111 AND 222222; |

|  |
| --- |
| SELECT \* FROM orders WHERE o\_orderkey BETWEEN 1 AND 2; |

|  |
| --- |
| SELECT \* FROM orders WHERE o\_orderkey BETWEEN 1 AND 999999; |

#### 

#### Pläne

|  |
| --- |
| Plan hash value: 1275100350 ---------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ---------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 27780 | 3011K| 6604 (1)| 00:00:01 | |\* 1 | TABLE ACCESS FULL| ORDERS | 27780 | 3011K| 6604 (1)| 00:00:01 | ----------------------------------------------------------------------------   Predicate Information (identified by operation id): ---------------------------------------------------   1 - filter("O\_ORDERKEY"<=222222 AND "O\_ORDERKEY">=111111) |

|  |
| --- |
| Plan hash value: 1275100350  ---------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ---------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 1 | 111 | 6604 (1)| 00:00:01 | |\* 1 | TABLE ACCESS FULL| ORDERS | 1 | 111 | 6604 (1)| 00:00:01 | ----------------------------------------------------------------------------   Predicate Information (identified by operation id): ---------------------------------------------------   1 - filter("O\_ORDERKEY"<=2 AND "O\_ORDERKEY">=1) |

|  |
| --- |
| Plan hash value: 1275100350  ---------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ---------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 250K| 26M| 6606 (1)| 00:00:01 | |\* 1 | TABLE ACCESS FULL| ORDERS | 250K| 26M| 6606 (1)| 00:00:01 | ----------------------------------------------------------------------------   Predicate Information (identified by operation id): ---------------------------------------------------   1 - filter("O\_ORDERKEY"<=999999 AND "O\_ORDERKEY">=1) |

#### Beobachtungen

Erneut haben wir eine Anzahl von sehr ähnlichen Plänen, denn jeder muss einen TABLE ACCESS FULL mit Filter ausführen, was direkt die hohen CPU Kosten (~6606) erklärt. Der Speicherplatz und die benötigte Anzahl Rows skaliert sehr schöne mit der Filter-Range Grösse: logischerweise, je mehr Daten getroffen werden vom Filter, desto mehr Zeilen und Speicherplatz wird benötigt.

Die sehr kleinen Unterschiede in den CPU Kosten sind hier wohl mit dem Umstand zu erklären, dass mehr Treffer zu minimal mehr Arbeit führen (die Treffer müssen zum späteren Versenden präpariert werden) und dass bei der letzten Abfrage für viele Zeilen mehr beide Bedingungen geprüft werden müssen im Vergleich zu den ersten beiden.

Abschliessend kann also gesagt werden, dass die Grösse der Range einem verschwindenden Effekt auf die CPU Kosten hat und wie zu erwarten, einen grossen Effekt auf den benötigten Speicherplatz und die Anzahl Rows hat.

Spannend ist auch zu sehen, dass die “kleine als”-Bedingung im Filter jeweils zuerst kommt. Das ist wohl bewusst so - eine mögliche Erklärung wäre, dass so mehr Zeilen direkt mit der ersten Bedingung ausgefiltert werden, denn nach oben ist die Domäne noch sehr gross (es gibt noch sehr viele Mögliche grössere Zahlen).

### Partial Range Query

#### Abfrage

|  |
| --- |
| SELECT \* FROM orders WHERE o\_orderkey BETWEEN 44444 AND 55555 AND o\_clerk BETWEEN 'Clerk#000000130' AND 'Clerk#000000139'; |

#### Plan

|  |
| --- |
| Plan hash value: 1275100350  ---------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ---------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 6 | 666 | 6613 (1)| 00:00:01 | |\* 1 | TABLE ACCESS FULL| ORDERS | 6 | 666 | 6613 (1)| 00:00:01 | ----------------------------------------------------------------------------   Predicate Information (identified by operation id): ---------------------------------------------------   1 - filter("O\_ORDERKEY"<=55555 AND "O\_CLERK"<='Clerk#000000139' AND   "O\_ORDERKEY">=44444 AND "O\_CLERK">='Clerk#000000130') |

#### Beobachtungen

Grundsätzlich ist diese Anfrage erneut sehr ähnlich wie bisherige: ein TABLE ACCESS FULL durchforstet die gesamte Tabelle mit einem Filter und sorgt damit für entsprechend hohe CPU Kosten (6613). Die Anzahl Treffer führt dann logischerweise zu der Anzahl Rows (6) und dem benötigten Speicherplatz (666 Bytes).

Spannend ist hier, dass im Vergleich zu vorherigen Anfragen wir nun einen relativ komplexen Filter haben - dieser aber nur zu einem verschwindend kleinen Zuwachs an CPU Kosten führt. Wohl können die meisten Zeilen direkt mit der ersten Bedingung entfernt werden - und selbst wenn mehrere Bedingungen geprüft werden müssen ‘kostet’ dies nur sehr wenig.

## Join

### Versuchsreihe

#### Abfrage 1

|  |
| --- |
| SELECT \* FROM orders, customers WHERE o\_custkey = c\_custkey AND o\_orderkey < 100; |

#### Plan 1

|  |
| --- |
| Plan hash value: 3766683618 ------------------------------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ------------------------------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 25 | 6750 | 54 (0)| 00:00:01 | | 1 | NESTED LOOPS | | 25 | 6750 | 54 (0)| 00:00:01 | | 2 | NESTED LOOPS | | 25 | 6750 | 54 (0)| 00:00:01 | | 3 | TABLE ACCESS BY INDEX ROWID BATCHED | ORDERS | 25 | 2775 | 4 (0)| 00:00:01 | |\* 4 | INDEX RANGE SCAN | O\_ORDERKEY\_IX | 25 | | 3 (0)| 00:00:01 | |\* 5 | INDEX RANGE SCAN | C\_CUSTKEY\_IX | 1 | | 1 (0)| 00:00:01 | | 6 | TABLE ACCESS BY INDEX ROWID | CUSTOMERS | 1 | 159 | 2 (0)| 00:00:01 | ------------------------------------------------------------------------------------------------------- Predicate Information (identified by operation id): ---------------------------------------------------  4 - access("O\_ORDERKEY"<100)  5 - access("O\_CUSTKEY"="C\_CUSTKEY") Note -----  - this is an adaptive plan |

#### Beobachtungen 1

Als erstes werden beide Tabellen geöffnet und auf beiden ein Index Range Scan ausgeführt. Dabei bleibt auf Customers 1 Zeile übrig. Dasselbe für Orders. Hier bleiben 25 Zeilen übrig.

Der Vergleich von O\_ORDERKEY\_IX mit C\_CUSTKEY\_IX und der Join geschehen in 2 Nested Loops, die beide zwar keinen zusätzlichen Speicher verbrauchen, dafür aber etwas “mehr” CPU.

#### Abfrage 2

|  |
| --- |
| SELECT \* FROM orders INNER JOIN customers ON o\_custkey = c\_custkey WHERE o\_orderkey < 100; |

#### Plan 2

|  |
| --- |
| Plan hash value: 3766683618 ------------------------------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ------------------------------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 25 | 6750 | 54 (0)| 00:00:01 | | 1 | NESTED LOOPS | | 25 | 6750 | 54 (0)| 00:00:01 | | 2 | NESTED LOOPS | | 25 | 6750 | 54 (0)| 00:00:01 | | 3 | TABLE ACCESS BY INDEX ROWID BATCHED | ORDERS | 25 | 2775 | 4 (0)| 00:00:01 | |\* 4 | INDEX RANGE SCAN | O\_ORDERKEY\_IX | 25 | | 3 (0)| 00:00:01 | |\* 5 | INDEX RANGE SCAN | C\_CUSTKEY\_IX | 1 | | 1 (0)| 00:00:01 | | 6 | TABLE ACCESS BY INDEX ROWID | CUSTOMERS | 1 | 159 | 2 (0)| 00:00:01 | ------------------------------------------------------------------------------------------------------- Predicate Information (identified by operation id): ---------------------------------------------------  4 - access("ORDERS"."O\_ORDERKEY"<100)  5 - access("O\_CUSTKEY"="C\_CUSTKEY") Note -----  - this is an adaptive plan |

#### Beobachtungen 2

Es wird hier derselbe Plan mit Hash 3766683618 verwendet. Ausführung und Ressourcenverbrauch bleiben gleich.

### Beobachtungen

Egal wie die Abfrage verändert wird. Solange das Resultat in der Struktur dasselbe ist, verwendet die DB den selben Ausführungsplan. Das heisst auch kleinere Änderungen wie filter("ORDERS"."O\_ORDERKEY"<120) verändern den Ausführungsplan bis auf den Ressourcenverbrauch nicht.

# 

# Versuche mit Index

## Erstellung

|  |
| --- |
| OrdersTable O\_ORDERKEY\_IX Index im Verhältnis zur Tabelle ziemlich gross. Verhältnis (1:5.6) 201,33M bytes 36M bytes   O\_CLERK\_IX Index im Verhältnis zur Tabelle sehr gross. Verhältnis (1:4.5)  45M bytes  O\_CUSTKEY\_IX Index Im Verhältnis zur Tabelle ziemlich gross. Verhältnis (1:5.6)  36M bytes |

## Projektion

#### Abfrage 1

|  |
| --- |
| SELECT DISTINCT o\_clerk FROM orders; -- Executes in ~0.18 seconds |

#### Plan 1

|  |
| --- |
| Plan hash value: 943631156   ------------------------------------------------------------------------------------ | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ------------------------------------------------------------------------------------ | 0 | SELECT STATEMENT | | 1000 | 16000 | 1555 (6)| 00:00:01 | | 1 | HASH UNIQUE | | 1000 | 16000 | 1555 (6)| 00:00:01 | | 2 | INDEX FAST FULL SCAN | O\_CLERK\_IX | 1500K| 22M| 1482 (1)| 00:00:01 | ------------------------------------------------------------------------------------ |

#### Beobachtungen 1

Durch den Index entstehen hohe CPU Kosten (6) für das Select Statement und das erstellen des Hashs. Der Full Table Scan ist dagegen sehr schnell benötigt aber sehr viel Speicher.

Dadurch ist aber der DISTINCT (über die gesamte Tabelle) extrem schnell und benötigt wenig CPU und baut hierbei den Speicher schnell ab (von 22M auf 16K).

## Selektion

### Exact Point Query

#### Abfrage 1

|  |
| --- |
| SELECT \* FROM orders WHERE o\_orderkey = 44480; -- ~0.02 seconds |

#### Plan 1

|  |
| --- |
| Plan hash value: 149606076 ----------------------------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ----------------------------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 1 | 111 | 4 (0)| 00:00:01 | | 1 | TABLE ACCESS BY INDEX ROWID BATCHED | ORDERS | 1 | 111 | 4 (0)| 00:00:01 | |\* 2 | INDEX RANGE SCAN | O\_ORDERKEY\_IX | 1 | | 3 (0)| 00:00:01 | ----------------------------------------------------------------------------------------------------- Predicate Information (identified by operation id): ---------------------------------------------------  2 - access("O\_ORDERKEY"=44480) |

#### Beobachtungen 1

Da hier sehr wenige Zeilen ausgewählt werden ist der Speicherverbrauch sehr gering. Auch der CPU Bedarf ist marginal, da keine NESTED LOOPs oder FAST FULL SCANs nötig sind. Zuerst wird hier das Select Statement ausgewertet, danach der Table geöffnet und nach dem Index durchsucht.

### Full Table Scan

#### Abfrage 2

|  |
| --- |
| SELECT /\*+ FULL(orders)\*/ \* FROM orders WHERE o\_orderkey = 44480; -- ~0.125 seconds |

#### Plan 2

|  |
| --- |
| Plan hash value: 1275100350 ---------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ---------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 1 | 111 | 6604 (1)| 00:00:01 | |\* 1 | TABLE ACCESS FULL| ORDERS | 1 | 111 | 6604 (1)| 00:00:01 | ---------------------------------------------------------------------------- Predicate Information (identified by operation id): ---------------------------------------------------  1 - filter("O\_ORDERKEY"=44480) |

#### Beobachtungen 2

Zwar entfällt durch den FULL TABLE SCAN der BATCHED TABLE ACCESS BY INDEX, verbraucht aber im Vergleich das Vielfache an CPU während der Ganze Table zuerst gescannt wird. Der Speicherverbrauch bleibt dabei aber gleich.

### Partial Point Query

#### Abfrage 3

|  |
| --- |
| SELECT \* FROM orders WHERE o\_custkey = 97303 OR o\_clerk = 'Clerk#000000860' -- ~0.02 |

#### Plan 3

|  |
| --- |
| Plan hash value: 3504738899 ---------------------------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ---------------------------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 1457 | 157K| 339 (0)| 00:00:01 | | 1 | TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS | 1457 | 157K| 339 (0)| 00:00:01 | | 2 | BITMAP CONVERSION TO ROWIDS | | | | | | | 3 | BITMAP OR | | | | | | | 4 | BITMAP CONVERSION FROM ROWIDS | | | | | | |\* 5 | INDEX RANGE SCAN | O\_CLERK\_IX | | | 8 (0)| 00:00:01 | | 6 | BITMAP CONVERSION FROM ROWIDS | | | | | | |\* 7 | INDEX RANGE SCAN | O\_CUSTKEY\_IX | | | 3 (0)| 00:00:01 | ---------------------------------------------------------------------------------------------------- Predicate Information (identified by operation id): ---------------------------------------------------  5 - access("O\_CLERK"='Clerk#000000860')  7 - access("O\_CUSTKEY"=97303) Note -----  - dynamic statistics used: dynamic sampling (level=2)  - 1 Sql Plan Directive used for this statement |

#### Beobachtungen 3

Bei Verwendung mehrerer Access Filter wird eine Bitmap erstellt für die beiden Argumente “O\_CUSTKEY” und “O\_CLERK”. Anhand dieser Bitmap wird einerseit nach O\_CUSTKEY und O\_CLERK gefiltert.

Dies hat (im Verhältnis zum INDEX RANGE SCAN) höhere initale Kosten für Speicher und CPU zur Folge, die sich aber beim Filtern wieder auszahlen, da dann kaum CPU und noch weniger Speicher aufgewendet werden muss.

#### Abfrage 4

|  |
| --- |
| SELECT \* FROM orders WHERE o\_custkey = 97303 AND o\_clerk = 'Clerk#000000860' -- ~0.02 |

#### Plan 4

|  |
| --- |
| Plan hash value: 2748057999 ---------------------------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ---------------------------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 8 | 888 | 11 (0)| 00:00:01 | | 1 | TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS | 8 | 888 | 11 (0)| 00:00:01 | | 2 | BITMAP CONVERSION TO ROWIDS | | | | | | | 3 | BITMAP AND | | | | | | | 4 | BITMAP CONVERSION FROM ROWIDS | | | | | | |\* 5 | INDEX RANGE SCAN | O\_CUSTKEY\_IX | 15 | | 3 (0)| 00:00:01 | | 6 | BITMAP CONVERSION FROM ROWIDS | | | | | | |\* 7 | INDEX RANGE SCAN | O\_CLERK\_IX | 15 | | 8 (0)| 00:00:01 | ---------------------------------------------------------------------------------------------------- Predicate Information (identified by operation id): ---------------------------------------------------  5 - access("O\_CUSTKEY"=97303)  7 - access("O\_CLERK"='Clerk#000000860') Note -----  - dynamic statistics used: dynamic sampling (level=2)  - 1 Sql Plan Directive used for this statement |

#### Beobachtungen 4

Der Ausführungsplan entspricht in etwa dem Vorherigen. Durch das AND allerdings wird der Outcome trotz gleicher Struktur verändert wodurch sich der Hash des Ausführungsplans ändert. Die stark einschränkende AND Operation schlägt sich auch im Kostenbereich bei der Erstellung der Bitmaps nieder.

Interessanterweise sind die Vorteile beim INDEX RANGE SCAN nicht zu spüren. Mit zusammengerechnet 11 CPU-Punkten entspricht dies genau dem Verbrauch der vorherigen Abfrage mit OR Operation.

#### Abfrage 5

|  |
| --- |
| SELECT \* FROM orders WHERE o\_custkey \* 2 = 194606 AND o\_clerk = 'Clerk#000000286'; -- ~0.02 |

#### Plan 5

|  |
| --- |
| Plan hash value: 1793913688 -------------------------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | -------------------------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 15 | 1665 | 1404 (1)| 00:00:01 | |\* 1 | TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS | 15 | 1665 | 1404 (1)| 00:00:01 | |\* 2 | INDEX RANGE SCAN | O\_CLERK\_IX | 1500 | | 8 (0)| 00:00:01 | -------------------------------------------------------------------------------------------------- Predicate Information (identified by operation id): ---------------------------------------------------  1 - filter("O\_CUSTKEY"\*2=194606)  2 - access("O\_CLERK"='Clerk#000000286') |

#### Beobachtungen 5

Interessanterweise wird hier im Gegensatz zur vorherigen Abfrage keine Bitmap verwendet. Dies liegt vermutlich an der Multiplikation von O\_CUSTKEY. Hierdurch ist der Wert errechnet und nicht mehr statisch und daher nicht weiter automatisch optimiert.

Dadurch fallen durchwegs höhere Kosten an bei TABLE ACCESS und INDEX RANGE SCAN.

### Range Query

#### Abfrage 6

|  |
| --- |
| SELECT \* FROM orders WHERE o\_orderkey BETWEEN 111111 AND 222222; -- ~0.025 |

#### Plan 6

|  |
| --- |
| Plan hash value: 149606076 ----------------------------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ----------------------------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 27780 | 3011K| 951 (1)| 00:00:01 | | 1 | TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS | 27780 | 3011K| 951 (1)| 00:00:01 | |\* 2 | INDEX RANGE SCAN | O\_ORDERKEY\_IX | 27780 | | 68 (0)| 00:00:01 | ----------------------------------------------------------------------------------------------------- Predicate Information (identified by operation id): ---------------------------------------------------  2 - access("O\_ORDERKEY">=111111 AND "O\_ORDERKEY"<=222222) |

#### Beobachtungen 6

Es scheint hier das selbe zu geschehen wie in der vorherigen Abfrage kann nicht mehr automatisch optimiert werden, da ein zusammengesetzter bzw. Errechneter Wert in der Abfrage vorkommt -> "O\_ORDERKEY">=111111 AND "O\_ORDERKEY"<=222222. Dies hat wiederum einen durchwegs erhöhten CPU und Speicherverbrauch zur Folge.

#### Abfrage-Plan 7

|  |
| --- |
| Plan hash value: 149606076 ----------------------------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ----------------------------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 55558 | 6022K| 1898 (1)| 00:00:01 | | 1 | TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS | 55558 | 6022K| 1898 (1)| 00:00:01 | |\* 2 | INDEX RANGE SCAN | O\_ORDERKEY\_IX | 55558 | | 134 (1)| 00:00:01 | ----------------------------------------------------------------------------------------------------- Predicate Information (identified by operation id): ---------------------------------------------------  2 - access("O\_ORDERKEY">=111111 AND "O\_ORDERKEY"<=333333) |

#### Beobachtungen Range Query

Der Plan bleibt derselbe. Allerdings verändern sich Ressourcenverbrauch (Speicher & CPU) in etwa linear zur Menge der Abgefragten Zeilen und Spalten.

### Partial Range Query

#### Abfrage 7

|  |
| --- |
| SELECT \* FROM orders WHERE o\_orderkey BETWEEN 44444 AND 55555 AND o\_clerk BETWEEN 'Clerk#000000130' AND 'Clerk#000000139'; -- ~0.025 |

#### Plan 7

|  |
| --- |
| Plan hash value: 2771568121 ----------------------------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ----------------------------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 30 | 3330 | 26 (12)| 00:00:01 | | 1 | TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS | 30 | 3330 | 26 (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 | | 13 (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') Note -----  - dynamic statistics used: dynamic sampling (level=2)  - 1 Sql Plan Directive used for this statement |

#### Beobachtungen 7

Trotz der viel aufwendigeren Abfrage, ist es SQL möglich diese Frage mit Indices und Bitmaps zu zu optimieren. Dies da die statischen Felder mit Keywords verglichen werden (x BETWEEN a AND b). Zwar wird dies von der Engine wieder umformuliert zu einem grösser/kleiner/gleich Vergleich allerdings weiss sie nun, dass der Boolean aus statischen Werten entnommen wird, die sie selbst kennt.

Auch hier sind CPU- und Speicherauslastung signifikant kleiner.

## Join

#### Abfrage 1

|  |
| --- |
| SELECT \* FROM orders, customers WHERE o\_custkey = c\_custkey; -- ~0.28 |

#### Plan 1

|  |
| --- |
| Plan hash value: 3042513348 ---------------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes |TempSpc| Cost (%CPU)| Time | ---------------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 1500K| 386M| | 17517 (1)| 00:00:01 | |\* 1 | HASH JOIN | | 1500K| 386M| 24M| 17517 (1)| 00:00:01 | | 2 | TABLE ACCESS FULL| CUSTOMERS | 150K| 22M| | 952 (1)| 00:00:01 | | 3 | TABLE ACCESS FULL| ORDERS | 1500K| 158M| | 6612 (1)| 00:00:01 | ---------------------------------------------------------------------------------------- Predicate Information (identified by operation id): ---------------------------------------------------  1 - access("O\_CUSTKEY"="C\_CUSTKEY") Note -----  - this is an adaptive plan |

#### Beobachtungen 1

Auch hier ist es der SQL Engine nicht möglich zu Optimieren, da der Join in der WHERE-Klausel von einer Funktion durchgeführt wird. Dadurch sind 2 Full Table Access ohne Optimierung nötig. Dies hat wie so oft zur Folge, dass unverhältnismässig viel CPU und Speicher benötigt wird.

#### Abfrage 2

|  |
| --- |
| SELECT \* FROM orders, customers WHERE o\_custkey = c\_custkey AND o\_orderkey < 100; -- ~0.1 |

#### Plan 2

|  |
| --- |
| Plan hash value: 3766683618 ------------------------------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ------------------------------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 25 | 6750 | 54 (0)| 00:00:01 | | 1 | NESTED LOOPS | | 25 | 6750 | 54 (0)| 00:00:01 | | 2 | NESTED LOOPS | | 25 | 6750 | 54 (0)| 00:00:01 | | 3 | TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS | 25 | 2775 | 4 (0)| 00:00:01 | |\* 4 | INDEX RANGE SCAN | O\_ORDERKEY\_IX | 25 | | 3 (0)| 00:00:01 | |\* 5 | INDEX RANGE SCAN | C\_CUSTKEY\_IX | 1 | | 1 (0)| 00:00:01 | | 6 | TABLE ACCESS BY INDEX ROWID | CUSTOMERS | 1 | 159 | 2 (0)| 00:00:01 | ------------------------------------------------------------------------------------------------------- Predicate Information (identified by operation id): ---------------------------------------------------  4 - access("O\_ORDERKEY"<100)  5 - access("O\_CUSTKEY"="C\_CUSTKEY") Note -----  - this is an adaptive plan |

#### Beobachtungen 2

Zwar ist die Abfrage weiterhin nicht besonders gut optimiert. Berücksichtigt aber initial bereits O\_ORDERKEY < 100 wodurch der Join Bereich stark eingeschränkt wird und somit auch der Aufwand. Trotzdem sind 2 NESTED LOOPs nötig.

#### Abfrage 3

|  |
| --- |
| CREATE INDEX c\_custkey\_ix ON customers(c\_custkey); SELECT \* FROM orders, customers WHERE o\_custkey = c\_custkey; -- ~0.35 |

#### Plan 3

|  |
| --- |
| Plan hash value: 3042513348 ---------------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes |TempSpc| Cost (%CPU)| Time | ---------------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 1500K| 386M| | 17517 (1)| 00:00:01 | |\* 1 | HASH JOIN | | 1500K| 386M| 24M| 17517 (1)| 00:00:01 | | 2 | TABLE ACCESS FULL| CUSTOMERS | 150K| 22M| | 952 (1)| 00:00:01 | | 3 | TABLE ACCESS FULL| ORDERS | 1500K| 158M| | 6612 (1)| 00:00:01 | ---------------------------------------------------------------------------------------- Predicate Information (identified by operation id): ---------------------------------------------------  1 - access("O\_CUSTKEY"="C\_CUSTKEY") Note -----  - this is an adaptive plan |

#### Beobachtungen 3

Die Problemstellung des SELECT-Statements ist hier die Gleiche wird aber durch die Erstellung des Index nicht beschleunigt. Speicher- sowie CPU-Auslastung steigen.

#### Abfrage 4

|  |
| --- |
| SELECT /\*+ USE\_NL \*/ \* FROM orders JOIN customers ON o\_custkey = c\_custkey WHERE o\_orderkey < 100; |

#### Plan 4

|  |
| --- |
| Plan hash value: 3766683618 ------------------------------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ------------------------------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 25 | 6750 | 54 (0)| 00:00:01 | | 1 | NESTED LOOPS | | 25 | 6750 | 54 (0)| 00:00:01 | | 2 | NESTED LOOPS | | 25 | 6750 | 54 (0)| 00:00:01 | | 3 | TABLE ACCESS BY INDEX ROWID BATCHED | ORDERS | 25 | 2775 | 4 (0)| 00:00:01 | |\* 4 | INDEX RANGE SCAN | O\_ORDERKEY\_IX | 25 | | 3 (0)| 00:00:01 | |\* 5 | INDEX RANGE SCAN | C\_CUSTKEY\_IX | 1 | | 1 (0)| 00:00:01 | | 6 | TABLE ACCESS BY INDEX ROWID | CUSTOMERS | 1 | 159 | 2 (0)| 00:00:01 | ------------------------------------------------------------------------------------------------------- Predicate Information (identified by operation id): ---------------------------------------------------  4 - access("ORDERS"."O\_ORDERKEY"<100)  5 - access("O\_CUSTKEY"="C\_CUSTKEY") Note -----  - this is an adaptive plan |

#### Beobachtungen 4

Der Plan ist identisch mit Abfrage 2

### Beobachtungen

Es gibt verschiedenste Methoden um die Abfrage zu beschleunigen und zu optimieren. Die einfachste hierbei ist, die Abfragen so gestalten, dass die SQL Engine das Statement auf statische Werte analysieren kann. Sobald funktionen verwendet werden, die es der Engine verunmöglichen die Quellwerte zu analysieren muss in Kauf genommen werden, dass die komplette Tabelle gescannt werden muss.

Um dieser Situation etwas abzuhelfen, kann im Voraus ein entsprechender Index/Indices erstellt werden. Diese benötigen zwar Speicherplatz verringern aber die Laufzeit meist um ein Vielfaches, wodurch der TradeOff gerechtfertigt ist.

# 

# Ausführungsplan Quiz

## Ausgangszustand

Zuerst wollen wir die Abfrage und deren Plan ohne optimierungen Anschauen.

### Abfrage

|  |
| --- |
| SELECT COUNT(\*) FROM parts, partsupps, lineitems WHERE p\_partkey=ps\_partkey  AND ps\_partkey=l\_partkey  AND ps\_suppkey=l\_suppkey  AND ( (ps\_partkey = 5 AND p\_type = 'MEDIUM ANODIZED BRASS')  OR (ps\_partkey = 5 AND p\_type = 'MEDIUM BRUSHED COPPER') ); |

### Plan

|  |
| --- |
| Plan hash value: 1497992900  ---------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ---------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 1 | 45 | 35602 (2)| 00:00:02 | | 1 | SORT AGGREGATE | | 1 | 45 | | | |\* 2 | HASH JOIN | | 4 | 180 | 35602 (2)| 00:00:02 | |\* 3 | HASH JOIN | | 4 | 144 | 5889 (6)| 00:00:01 | |\* 4 | TABLE ACCESS FULL| PARTSUPPS | 4 | 36 | 4525 (1)| 00:00:01 | |\* 5 | TABLE ACCESS FULL| PARTS | 2667 | 72009 | 1052 (1)| 00:00:01 | | 6 | TABLE ACCESS FULL | LINEITEMS | 6001K| 51M| 29682 (1)| 00:00:02 | ----------------------------------------------------------------------------------   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') |

### Beobachtungen

Sofort fällt ins Auge, dass die Filterbedingungen wohl noch vereinfacht werden können - denn diese dynamischen und verschachtelten Bedingungen verringern nicht nur die menschliche Lehsbarkeit sondern machen es wohl auch schwieriger für den Optimizer eine möglichst schnelle Abfrage zu gestalten.

Als zweites fällt auf, dass ein grossteil der CPU Kosten von den initialen (und die meisten vom Ersten) TABLE ACCESS FULL stammt - die HASH JOIN Operationen tragen nur sehr wenig bei, genauso wie das SORT AGGREGATE.

Dass drei TABLE ACCESS FULL gemacht werden müssen, macht Sinn, denn wir Fragen nach Daten von 3 Tabellen.

## Bestes Ergebnis

### Vorgehen

Zuerst habe ich alle berührten Spalten in der Abfrage Indexe erstellt. Danach habe ich das Query umgeschrieben, so dass es sowohl für Menschen leserlicher geworden ist als auch dem Optimizer mehr Freiraum gibt. Ich habe ausserdem eine Funktion erstellt, welche deterministisch 1 zurückgibt wenn 'MEDIUM ANODIZED BRASS' oder 'MEDIUM BRUSHED COPPER' in die Funktion hineingegeben wird, und 0 in allen anderen Fällen. Darauf habe ich dann einen weiteren Function Index erstellt - wobei ich sowohl eine Bitmap Index als auch einen normalen versucht habe.

Danach habe ich einige Pläne verglichen und musste feststellen, dass die Indexe bezüglich der String-Vergleiche keinen Unterschied machen. Auch mit der Reihenfolge der Bedingungen konnte ich keine Unterschiede mehr herbeiführen. Zuletzt habe ich noch alle Indexe und Funktionen wieder entfernt welche nicht gebraucht werden in der besten Abfrage - dies ist wichtig um eine Datenbank in einem möglichst sauberen und performanten Zustand zu halten.

### Indexe & Abfrage

|  |
| --- |
| CREATE INDEX PS\_PARTKEY ON partsupps(ps\_partkey); CREATE INDEX P\_PARTKEY ON parts(p\_partkey); CREATE INDEX L\_PARTKEY ON lineitems(l\_partkey); CREATE INDEX L\_SUPPKEY ON lineitems(l\_suppkey); |

|  |
| --- |
| SELECT COUNT(\*) FROM parts, partsupps, lineitems WHERE p\_partkey = 5  AND ps\_partkey = 5  AND l\_partkey = 5  AND ps\_suppkey=l\_suppkey  AND (p\_type = 'MEDIUM ANODIZED BRASS'  OR p\_type = 'MEDIUM BRUSHED COPPER'); |

### Neuer Plan

|  |
| --- |
| Plan hash value: 4259255744 ---------------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ---------------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 1 | 45 | 21 (10)| 00:00:01 | | 1 | SORT AGGREGATE | | 1 | 45 | | | | 2 | NESTED LOOPS | | 1 | 45 | 21 (10)| 00:00:01 | | 3 | MERGE JOIN CARTESIAN | | 1 | 36 | 5 (0)| 00:00:01 | |\* 4 | TABLE ACCESS BY  INDEX ROWID | PARTS | 1 | 27 | 2 (0)| 00:00:01 | |\* 5 | INDEX RANGE SCAN | P\_PARTKEY | 1 | | 1 (0)| 00:00:01 | | 6 | BUFFER SORT | | 4 | 36 | 3 (0)| 00:00:01 | | 7 | TABLE ACCESS BY  INDEX ROWID BATCHED| PARTSUPPS | 4 | 36 | 3 (0)| 00:00:01 | |\* 8 | INDEX RANGE SCAN | PS\_PARTKEY | 4 | | 2 (0)| 00:00:01 | | 9 | BITMAP CONVERSION COUNT| | 1 | 9 | 21 (10)| 00:00:01 | | 10 | BITMAP AND | | | | | | | 11 | BITMAP CONVERSION  FROM ROWIDS | | | | | | |\* 12 | INDEX RANGE SCAN | L\_PARTKEY | 30 | | 2 (0)| 00:00:01 | | 13 | BITMAP CONVERSION  FROM ROWIDS | | | | | | |\* 14 | INDEX RANGE SCAN | L\_SUPPKEY | 30 | | 2 (0)| 00:00:01 | ----------------------------------------------------------------------------------------   Predicate Information (identified by operation id): ---------------------------------------------------    4 - filter("P\_TYPE"='MEDIUM ANODIZED BRASS' OR "P\_TYPE"='MEDIUM BRUSHED COPPER')  5 - access("P\_PARTKEY"=5)  8 - access("PS\_PARTKEY"=5) |

Dass wir mit 4 Indexen und einer kleinen Veränderung der Abfrage eine beinahe 1’700-fache Beschleunigung der Abfrage (und eine 1’700’000-fache Reduktion der maximalen Speicherbelastung) erreicht haben ist beachtlich.

### Entfernte Indexe & Funktionen

Hier zeige ich nicht alle Abfragen die ich versucht haben, denn das sind einige. Jedoch möchte ich die wieder gelöschten Funktionen und Indexe die ich versucht habe dokumentieren.

|  |
| --- |
| create or replace FUNCTION IS\_MAB  (  STR IN VARCHAR2  ) RETURN number DETERMINISTIC AS  BEGIN  if STR = 'MEDIUM ANODIZED BRASS' then  return 1;  else   return 0;  end if; END IS\_MAB; |

|  |
| --- |
| create or replace FUNCTION IS\_MBC  (  STR IN VARCHAR2  ) RETURN number DETERMINISTIC AS  BEGIN  if STR = 'MEDIUM BRUSHED COPPER' then  return 1;  else   return 0;  end if; END IS\_MBC; |

|  |
| --- |
| create or replace FUNCTION IS\_MAB\_OR\_MBC  (  STR IN VARCHAR2  ) RETURN number DETERMINISTIC AS  BEGIN  if STR = 'MEDIUM ANODIZED BRASS' OR STR = 'MEDIUM BRUSHED COPPER' then  return 1;  else   return 0;  end if; END IS\_MAB\_OR\_MBC; |

|  |
| --- |
| CREATE INDEX PS\_SUPPKEY ON partsupps(ps\_suppkey); CREATE INDEX P\_TYPE ON parts(p\_type); CREATE INDEX P\_TYPE\_ISMAB on parts(is\_mab(p\_type)); CREATE INDEX P\_TYPE\_ISMBC on parts(is\_mbc(p\_type)); CREATE INDEX P\_TYPE\_ISMAB\_OR\_MBC on parts(IS\_MAB\_OR\_MBC(p\_type)); CREATE BITMAP INDEX P\_TYPE\_ISMAB\_OR\_MBC\_BITMAP on parts(IS\_MAB\_OR\_MBC(p\_type)); |

Zu guter letzt zeige ich noch die die Abfrage ausgesehn hätte mit obigem Function-Index:

|  |
| --- |
| SELECT COUNT(\*) FROM parts, partsupps, lineitems WHERE p\_partkey = 5  AND l\_partkey = 5  AND ps\_suppkey=l\_suppkey  AND ps\_partkey = 5  AND IS\_MAB\_OR\_MBC(p\_type) = 1; |

# Deep Left Join

## Vorinformation

Generell neigt Oracle SQL immer zu einem Left Deep Join und sehr selten zu einem Right Deep Join. Was aber gilt ist, dass ein Bushy Join in der Regel nie verwendet wird und künstlich erzwungen werden muss. Als Ressourcen werden folgende Links verwendet:

* <http://dbakevlar.com/2014/05/right-deep-left-deep-and-bushy-joins-in-sql/>
* <http://www.oaktable.net/content/right-deep-left-deep-and-bushy-joins>
* <https://tonyhasler.wordpress.com/2008/12/27/bushy-joins/>

Um nicht mit anderen Gruppenmitgliedern und deren Indizes in Konflikt zu geraten, werden die benötigten Tabellen kopiert:

|  |
| --- |
| CREATE TABLE PARTS0 AS (SELECT \* FROM PARTS); CREATE TABLE PARTSUPPS0 AS (SELECT \* FROM PARTSUPPS); CREATE TABLE LINEITEMS0 AS (SELECT \* FROM LINEITEMS); CREATE TABLE ORDERS0 AS (SELECT \* FROM ORDERS); |

## Statements ohne Indizes

Ein reguläres Statement sieht wie folgt aus:

|  |
| --- |
| EXPLAIN PLAN FOR SELECT \* FROM PARTS0, PARTSUPPS0, LINEITEMS0, ORDERS0 WHERE  L\_ORDERKEY = O\_ORDERKEY AND  L\_PARTKEY = PS\_PARTKEY AND  L\_SUPPKEY = PS\_SUPPKEY AND  PS\_PARTKEY = P\_PARTKEY;   SELECT plan\_table\_output FROM TABLE(DBMS\_XPLAN.DISPLAY()); |

Dieses Statement generiert folgende Kosten:

|  |
| --- |
| Plan hash value: 897656887   ------------------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes |TempSpc| Cost (%CPU)| Time | ------------------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 803K| 391M| | 122K (1)| 00:00:05 | |\* 1 | HASH JOIN | | 803K| 391M| 27M| 122K (1)| 00:00:05 | | 2 | TABLE ACCESS FULL | PARTS0 | 200K| 25M| | 1052 (1)| 00:00:01 | |\* 3 | HASH JOIN | | 811K| 293M| 175M| 105K (1)| 00:00:05 | | 4 | TABLE ACCESS FULL | ORDERS0 | 1500K| 158M| | 6612 (1)| 00:00:01 | |\* 5 | HASH JOIN | | 800K| 204M| 118M| 79094 (1)| 00:00:04 |  PLAN\_TABLE\_OUTPUT  ------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------ | 6 | TABLE ACCESS FULL| PARTSUPPS0 | 800K| 109M| | 4528 (1)| 00:00:01 | | 7 | TABLE ACCESS FULL| LINEITEMS0 | 6001K| 715M| | 29763 (1)| 00:00:02 | -------------------------------------------------------------------------------------------   Predicate Information (identified by operation id): ---------------------------------------------------    1 - access("PS\_PARTKEY"="P\_PARTKEY")  3 - access("L\_ORDERKEY"="O\_ORDERKEY")  5 - access("L\_PARTKEY"="PS\_PARTKEY" AND "L\_SUPPKEY"="PS\_SUPPKEY")  21 Zeilen gewählt. |

Ein Statement mit erzwungenen Bushy Tree sieht wie folgt aus (Hinweis: Ich habe recht lange versucht ein Bushy Tree Statement zu bauen und bin gescheitert. Um weiterarbeiten zu können, habe ich auf das Statement von Christian Glatthard & Filip Schramka zurückgegriffen):

|  |
| --- |
| EXPLAIN PLAN FOR WITH LINE\_ORDERS AS(SELECT /\*+ leading(LINEITEMS0 ORDERS0) no\_swap\_join\_inputs(ORDERS0) use\_hash(ORDERS0) no\_merge \*/ \* FROM LINEITEMS0, ORDERS0 WHERE L\_ORDERKEY = O\_ORDERKEY),  SUPPS\_PARTS AS(SELECT /\*+ leading(PARTSUPPS0 PARTS0) no\_swap\_join\_inputs(PARTS0) use\_hash(PARTS0) no\_merge \*/ \* FROM PARTSUPPS0,PARTS0 WHERE PS\_PARTKEY = P\_PARTKEY)  SELECT /\*+ leading(line\_orders supps\_parts) use\_hash(supp\_parts) no\_swap\_join\_inputs(supp\_parts) \*/ \* FROM LINE\_ORDERS, SUPPS\_PARTS  WHERE L\_PARTKEY = PS\_PARTKEY AND L\_SUPPKEY = PS\_SUPPKEY;  SELECT plan\_table\_output FROM TABLE(DBMS\_XPLAN.DISPLAY()); |

Dieses Statement generiert folgende Kosten:

|  |
| --- |
| Plan hash value: 3290369918   ------------------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes |TempSpc| Cost (%CPU)| Time | ------------------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 803K| 488M| | 209K (1)| 00:00:09 | |\* 1 | HASH JOIN | | 803K| 488M| 2037M| 209K (1)| 00:00:09 | | 2 | VIEW | | 6086K| 1967M| | 84057 (1)| 00:00:04 | |\* 3 | HASH JOIN | | 6086K| 1369M| 784M| 84057 (1)| 00:00:04 | | 4 | TABLE ACCESS FULL| LINEITEMS0 | 6001K| 715M| | 29763 (1)| 00:00:02 | | 5 | TABLE ACCESS FULL| ORDERS0 | 1500K| 158M| | 6612 (1)| 00:00:01 |  PLAN\_TABLE\_OUTPUT  ------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------ | 6 | VIEW | | 792K| 225M| | 12816 (1)| 00:00:01 | |\* 7 | HASH JOIN | | 792K| 207M| 118M| 12816 (1)| 00:00:01 | | 8 | TABLE ACCESS FULL| PARTSUPPS0 | 800K| 109M| | 4528 (1)| 00:00:01 | | 9 | TABLE ACCESS FULL| PARTS0 | 200K| 25M| | 1052 (1)| 00:00:01 | -------------------------------------------------------------------------------------------   Predicate Information (identified by operation id): ---------------------------------------------------    1 - access("L\_PARTKEY"="PS\_PARTKEY" AND "L\_SUPPKEY"="PS\_SUPPKEY")  3 - access("L\_ORDERKEY"="O\_ORDERKEY")  PLAN\_TABLE\_OUTPUT  ------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------  7 - access("PS\_PARTKEY"="P\_PARTKEY")  23 Zeilen gewählt. |

Enorm eindrücklich, wenn nicht sogar schon beängstigend, sind die x-fach höhere Cost sowie die Byte-Positionen mit 1967 M und 1369 M des Bushy Statements. Werden mehrere solche Bushy Queries parallel ausgeführt, wird ein ressourcenstarker Datenbankserver verwendet.

Um die Statements zu verbessern, setzen wir Indizes:

|  |
| --- |
| CREATE INDEX ORDER0\_P\_PARTKEY\_INDEX ON PARTS0(P\_PARTKEY); CREATE INDEX ORDER0\_PS\_PARTKEY\_INDEX ON PARTSUPPS0(PS\_PARTKEY); CREATE INDEX ORDER0\_PS\_SUPPKEY\_INDEX ON PARTSUPPS0(PS\_SUPPKEY); CREATE INDEX ORDER0\_L\_PARTKEY\_INDEX on LINEITEMS0(L\_PARTKEY); CREATE INDEX ORDER0\_L\_SUPPKEY\_INDEX on LINEITEMS0(L\_SUPPKEY); CREATE INDEX ORDER0\_L\_ORDERKEY\_INDEX ON LINEITEMS0(L\_ORDERKEY); CREATE INDEX ORDER0\_O\_ORDERKEY\_INDEX ON ORDERS0(O\_ORDERKEY); |

## Statements mit Indizes

Lassen wir nun das reguläre Statement laufen, generieren wir folgende Kosten:

|  |
| --- |
| Plan hash value: 897656887   ------------------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes |TempSpc| Cost (%CPU)| Time | ------------------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 803K| 391M| | 122K (1)| 00:00:05 | |\* 1 | HASH JOIN | | 803K| 391M| 27M| 122K (1)| 00:00:05 | | 2 | TABLE ACCESS FULL | PARTS0 | 200K| 25M| | 1052 (1)| 00:00:01 | |\* 3 | HASH JOIN | | 811K| 293M| 175M| 105K (1)| 00:00:05 | | 4 | TABLE ACCESS FULL | ORDERS0 | 1500K| 158M| | 6612 (1)| 00:00:01 | |\* 5 | HASH JOIN | | 800K| 204M| 118M| 79094 (1)| 00:00:04 |  PLAN\_TABLE\_OUTPUT  ------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------ | 6 | TABLE ACCESS FULL| PARTSUPPS0 | 800K| 109M| | 4528 (1)| 00:00:01 | | 7 | TABLE ACCESS FULL| LINEITEMS0 | 6001K| 715M| | 29763 (1)| 00:00:02 | -------------------------------------------------------------------------------------------   Predicate Information (identified by operation id): ---------------------------------------------------    1 - access("PS\_PARTKEY"="P\_PARTKEY")  3 - access("L\_ORDERKEY"="O\_ORDERKEY")  5 - access("L\_PARTKEY"="PS\_PARTKEY" AND "L\_SUPPKEY"="PS\_SUPPKEY")    PLAN\_TABLE\_OUTPUT  ------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------ Note -----  - this is an adaptive plan  25 Zeilen gewählt. |

Das Bushy Tree Statement generiert folgende Kosten:

|  |
| --- |
| Plan hash value: 3290369918   ------------------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes |TempSpc| Cost (%CPU)| Time | ------------------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 803K| 488M| | 209K (1)| 00:00:09 | |\* 1 | HASH JOIN | | 803K| 488M| 2037M| 209K (1)| 00:00:09 | | 2 | VIEW | | 6086K| 1967M| | 84057 (1)| 00:00:04 | |\* 3 | HASH JOIN | | 6086K| 1369M| 784M| 84057 (1)| 00:00:04 | | 4 | TABLE ACCESS FULL| LINEITEMS0 | 6001K| 715M| | 29763 (1)| 00:00:02 | | 5 | TABLE ACCESS FULL| ORDERS0 | 1500K| 158M| | 6612 (1)| 00:00:01 |  PLAN\_TABLE\_OUTPUT  ------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------ | 6 | VIEW | | 792K| 225M| | 12816 (1)| 00:00:01 | |\* 7 | HASH JOIN | | 792K| 207M| 118M| 12816 (1)| 00:00:01 | | 8 | TABLE ACCESS FULL| PARTSUPPS0 | 800K| 109M| | 4528 (1)| 00:00:01 | | 9 | TABLE ACCESS FULL| PARTS0 | 200K| 25M| | 1052 (1)| 00:00:01 | -------------------------------------------------------------------------------------------   Predicate Information (identified by operation id): ---------------------------------------------------    1 - access("L\_PARTKEY"="PS\_PARTKEY" AND "L\_SUPPKEY"="PS\_SUPPKEY")  3 - access("L\_ORDERKEY"="O\_ORDERKEY")  PLAN\_TABLE\_OUTPUT  ------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------  7 - access("PS\_PARTKEY"="P\_PARTKEY")  23 Zeilen gewählt. |

## Beobachtungen

* Eigentlich sollten die Statements mit Indizes einiges schneller sein als die ohne. Im obigen Output ist das aber nicht der Fall und mir ist momentan nicht ersichtlich, warum dies der Fall ist. Schauen wir das 1. Beispiel in unseren eigenen SQL Anfragen an, sollten alle ID Vergleiche mir Indizes beschleunigbar sein.
* Das reguläre Left-Deep Statement ist viel ressourcenschonender als der Bushy Tree
* Durch die beiden Views können im Bushy Tree die Resultate nicht wie im Left Deep nach der Baumverarbeitung verworfen werden und müssen gespeichert werden). Besonders erwähnenswert sind der Speicheraufwand von 1967M und 1396M !):

Meiner Meinung nach macht ein Bushy Tree überhaupt nur Sinn, wenn die grössten Datensätze zuerst (Sprich in der Hierarchy unten) abgearbeitet werden kann und so die zu weiterverarbeitende Datengrösse sinkt/noch klein ist. Allenfalls kann ein Bushy Tree im schlimmsten Fall zu einer quadratischen Komplexität führen (Das können andere Trees auch, aber der Bushy Tree kann schlechter alte Nodes verwerfen), welche mit jeder Verknüpfung weitergeführt wird und so enorm ressourcenintensiv wird (Achtung: Worst Case):

* m Rows aus der linken Node multipliziert mal n Rows aus der rechten Node → 100 x 100 = 10’000).
* Resultat m x n aus der linken Node multipliziert mal o Rows aus der rechten Node → 10’000 x 100 = 1’000’000)

Mir persönlich entzieht sich der wirkliche Use Cases eines Bushy Tree, da ein Left/Right Deep Tree die abgearbeiteten Einträge schneller wieder verwerfen kann und so viel CPU und Speicher freundlicher ist.

# Eigene SQL Anfragen

## Beispiel 1

### Ausgangslage

Das erste Beispiel bezieht sich auf die oben thematisierte m x n Multiplikation mit mehreren WHERE Kriterien. Ohne Indizes auf diese WHERE Kriterien müssen beide Tabellen komplett gescannt werden. Für die einfachere Verwendung kopieren wir die Orders-Tabelle:

|  |
| --- |
| CREATE TABLE ORDERS1 AS (SELECT \* FROM ORDERS); CREATE TABLE ORDERS2 AS (SELECT \* FROM ORDERS); |

Erstellen wir nun ein Query mit mehreren WHERE Klauseln, welche einen gleichen Key verfügen und einen Range spezifizieren:

|  |
| --- |
| EXPLAIN PLAN FOR SELECT \* FROM ORDERS1, ORDERS2 WHERE  ORDERS1.O\_ORDERKEY = ORDERS2.O\_ORDERKEY AND  ORDERS2.O\_ORDERKEY BETWEEN 19450 AND 19460;  SELECT plan\_table\_output FROM TABLE(DBMS\_XPLAN.DISPLAY()); |

Wie erwähnt müssen beide Tables komplett gescannt werden:

|  |
| --- |
| Plan hash value: 1730955651   ------------------------------------------------------------------------------ | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ------------------------------------------------------------------------------ | 0 | SELECT STATEMENT | | 4 | 888 | 13207 (1)| 00:00:01 | |\* 1 | HASH JOIN | | 4 | 888 | 13207 (1)| 00:00:01 | |\* 2 | TABLE ACCESS FULL| ORDERS1 | 5 | 555 | 6604 (1)| 00:00:01 | |\* 3 | TABLE ACCESS FULL| ORDERS2 | 5 | 555 | 6604 (1)| 00:00:01 | ------------------------------------------------------------------------------    PLAN\_TABLE\_OUTPUT  ------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------ Predicate Information (identified by operation id): ---------------------------------------------------    1 - access("ORDERS1"."O\_ORDERKEY"="ORDERS2"."O\_ORDERKEY")  2 - filter("ORDERS1"."O\_ORDERKEY"<=19460 AND   "ORDERS1"."O\_ORDERKEY">=19450)  3 - filter("ORDERS2"."O\_ORDERKEY"<=19460 AND   "ORDERS2"."O\_ORDERKEY">=19450)  19 Zeilen gewählt. |

### Anpassung

Um einen kompletten Scan zu vermeiden, setzen wir auf alle involvierten Columns der WHERE Klausel Indizes:

|  |
| --- |
| CREATE INDEX ORDERS1\_ORDERKEY\_INDEX on ORDERS1(O\_ORDERKEY); CREATE INDEX ORDERS2\_ORDERKEY\_INDEX on ORDERS2(O\_ORDERKEY); |

Lassen wir das Query erneut laufen, erhalten wir folgendes Resultat:

|  |
| --- |
| Plan hash value: 4127084313   --------------------------------------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | --------------------------------------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 4 | 888 | 8 (0)| 00:00:01 | |\* 1 | HASH JOIN | | 4 | 888 | 8 (0)| 00:00:01 | | 2 | TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS1 | 5 | 555 | 4 (0)| 00:00:01 | |\* 3 | INDEX RANGE SCAN | ORDERS1\_ORDERKEY\_INDEX | 5 | | 3 (0)| 00:00:01 | | 4 | TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS2 | 5 | 555 | 4 (0)| 00:00:01 | |\* 5 | INDEX RANGE SCAN | ORDERS2\_ORDERKEY\_INDEX | 5 | | 3 (0)| 00:00:01 |  PLAN\_TABLE\_OUTPUT  ------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------ ---------------------------------------------------------------------------------------------------------------   Predicate Information (identified by operation id): ---------------------------------------------------    1 - access("ORDERS1"."O\_ORDERKEY"="ORDERS2"."O\_ORDERKEY")  3 - access("ORDERS1"."O\_ORDERKEY">=19450 AND "ORDERS1"."O\_ORDERKEY"<=19460)  5 - access("ORDERS2"."O\_ORDERKEY">=19450 AND "ORDERS2"."O\_ORDERKEY"<=19460)   Note -----  PLAN\_TABLE\_OUTPUT  ------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------  - this is an adaptive plan  23 Zeilen gewählt. |

### Boebachtung

Mit dem Setzen eines Indizes auf die beiden Order-Keys erreichen wir die geforderte Cost-Reduktion um den Faktor 100. Grund dafür ist, dass kein kompletter Table Scan (TABLE ACCESS FULL) mehr stattfindet und nur noch ein Range-Scan (INDEX RANGE SCAN).

## Beispiel 2

### Ausgangslage

Beim Rumgoogeln für das erste Beispiel wurde mir sehr schnell klar, dass Funktionen wie GROUP BY, ORDER BY und DISTINCT ohne eigene Indizes temporäres Memory für die Funktionsumsetzung benötigen. Als gängige Lösung wurde ein sogenannter “Covering-Index” vorgeschlagen, welcher die Funktionswerte, sprich Resultate in Indizes zusammenfasst.

Ich möchte wieder die Orders Tabelle verwenden und dabei der gesamte Bestellwert eines bestimmten Clerks abfragen.

Zuerst erstellen wir eine neue Tabelle basierend auf Orders:

|  |
| --- |
| CREATE TABLE ORDERS3 AS (SELECT \* FROM ORDERS); |

Danach fragen wir die Bestellwerte eines Clerks ab:

|  |
| --- |
| EXPLAIN PLAN FOR SELECT SUM(O\_TOTALPRICE) FROM ORDERS3 WHERE O\_CLERK = 'Clerk#000000001';  SELECT plan\_table\_output FROM TABLE(DBMS\_XPLAN.DISPLAY()); |

Obwohl wir eigentlich nur den Gesamtwert der bestellten Waren für einen einzelnen Clerk abfragen wollen, wir die gesamte Tabelle gescannt:

|  |
| --- |
| Plan hash value: 1643448575   ------------------------------------------------------------------------------ | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ------------------------------------------------------------------------------ | 0 | SELECT STATEMENT | | 1 | 22 | 6629 (1)| 00:00:01 | | 1 | SORT AGGREGATE | | 1 | 22 | | | |\* 2 | TABLE ACCESS FULL| ORDERS3 | 1500 | 33000 | 6629 (1)| 00:00:01 | ------------------------------------------------------------------------------   Predicate Information (identified by operation id):  PLAN\_TABLE\_OUTPUT  ------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------ ---------------------------------------------------    2 - filter("O\_CLERK"='Clerk#000000001')  14 Zeilen gewählt. |

### Anpassung

In grossen Tabellen wäre es schön, wenn für jeden Clerk schon ein voraggregiertes SUM Indizes existieren würde und somit ein Scan der gesamten Tabelle verhindert werde könnte. Basierend auf der Theorie des “Covering-Index” erstellen wir also für jeden Clerk Indizes auf die Bestellungen:

|  |
| --- |
| CREATE INDEX ORDERS3\_TOTALPRICESUM\_INDEX ON ORDERS3(O\_CLERK, O\_TOTALPRICE); |

Lassen wir das Query erneut laufen, erhalten wir folgendes Resultat:

|  |
| --- |
| Plan hash value: 1280548289   ------------------------------------------------------------------------------------------------- | Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time | ------------------------------------------------------------------------------------------------- | 0 | SELECT STATEMENT | | 1 | 22 | 9 (0)| 00:00:01 | | 1 | SORT AGGREGATE | | 1 | 22 | | | |\* 2 | INDEX RANGE SCAN| ORDERS3\_TOTALPRICESUM\_INDEX | 1500 | 33000 | 9 (0)| 00:00:01 | -------------------------------------------------------------------------------------------------   Predicate Information (identified by operation id):  PLAN\_TABLE\_OUTPUT  ------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------ ---------------------------------------------------    2 - access("O\_CLERK"='Clerk#000000001')  14 Zeilen gewählt. |

### Boebachtung

Wie auch im 1. Beispiel können wir hier durch das Wegfallen eines gesamten Scan der Tabelle durch einen INDEX RANGE SCAN die Cost um mehr als den Faktor 100 senken.

## Fazit

In meiner Lehre habe ich einen Lehrling kennengelernt, welcher bei Coop mit MSSQL und Oracle-Datenbanken arbeitet. Ich als Java Entwickler und Hibernate/ORM Anwender lag mit ihm immer ihm stetigen Clinch betreffend Performance und verbosen Queries: Ich war der Meinung, mir würde das händische Schreiben von Queries nicht bezahlt werden (Unwirtschaftlich) und er der, dass die Query von Hibernate/ORM enorm unperformant sind und man stetig händisch Indizes setzen muss (Was seine gesamte Abteilung nerven würde). Ich habe seine Aussage immer etwas auf die leichte Schulter genommen.

Nach dem Verfassen und Testen dieser beiden doch recht einfachen Queries bin ich recht schockiert, wie stark sich die Performance durch das Setzen von Indizes verändern kann und die Coast sinkt. Wir haben hier schon mit mehr Daten als üblich gearbeitet, doch wird es sicher noch viel eindrücklicher, wenn die Anzahl der Rows einer produktiven Datenbank in die Millionen geht.