SQL Optimizer und SQL Performance

Marco Mischke
Robotron Datenbank Software GmbH
Dresden

Schlüsselworte
SQL, Optimizer, Explain Plan, SQL Trace

Einleitung

Dieser Vortrag beschäftigt sich mit grundlegenden Eigenschaften des Oracle Query Optimizers. Es wird eine Übersicht über mögliche Zugriffspfade und die Ermittlung sowie die Interpretation von Ausführungsplänen gegeben. Dabei werden verschiedene Methoden zur Ermittlung von Ausführungsplänen gegeben sowie deren Vor- und Nachteile diskutiert. Weiterhin werden Methoden und Strategien zur Ermittlung von Performance Engpässen in SQL Abfragen beleuchtet. Dabei werden häufige Fehler im Design von Anwendung und Datenmodell aufgezeigt und Möglichkeiten zur positiven Beeinflussung der Ausführung von SQL Abfragen erläutert. Dabei werden u.a. Themen wie Function Based Indexes, Extended Column Statistics und Adaptive Cursor Sharing erörtert.
Dabei liegt der Schwerpunkt nicht auf Vollständigkeit. Es soll vielmehr ein erster Einblick in die grundlegende Funktionsweise gegeben werden.

Der Optimizer

Die Oracle Datenbank versucht SQL Statements möglichst effizient auszuführen. Die optimale Ausführung zu ermitteln ist Aufgabe des Optimizers. Dieser bezieht sich dabei auf eine Vielzahl von Randbedingungen. Dazu zählen Initialisierungsparameter genauso wie die Statistiken der einzelnen Segmente und Informationen zur Leistung der Serverhardware und des I/O Systems. Die Segmentstatistiken sollen dem Optimizer eine Möglichkeit geben die Art, Verteilung, Menge und Zusammenhänge der Daten abschätzen zu können. Um optimale Ausführungen zu erzielen vergleicht der Optimizer verschiedene Möglichkeiten die Daten aus den Tabellen zu ermitteln. Dazu zählen die Zugriffspfade auf die Tabellen sowie die verschiedenen Join-Methoden um die Inhalte mehrerer Tabellen zu verknüpfen.

Zugriffspfade

Um die Daten aus den einzelnen beteiligten Tabellen zu ermitteln, stehen dem Optimizer verschiedene Varianten zu Verfügung. Diese werden im Folgenden erläutert.

Singleblock I/O

ROWID	Der Datensatz wird direkt über die physikalische Adresse gelesen. Hierfür ist nur ein einziger I/O nötig und ist damit die effektivste Zugriffsmöglichkeit.
Cluster Hash Key	Der Datensatz befindet sich ein einer Hashcluster Tabelle. Die physikalische Adresse wird direkt aus dem Hashkey des Datensatzes ermittelt. Auch hier ist nur ein einziger I/O nötig.
Index Unique Scan	Der Datensatz wird aus einem Index mittels eines eindeutigen Schlüsselwertes ermittelt. Der Index liefert direkt die ROWID des gefordertem Datensatzes.
Index Range Scan	Aus einem Index werden mittels eines nicht eindeutigen Wertes oder eines Wertebereiches die Adressen der in Frage kommenden Datensätze ermittelt. Diese werden dann entsprechend gelesen und ggf. gefiltert.

Index Skip Scan	Die Adressen der Datensätze werden aus einem mehrspaltigen Index ermittelt. Dabei enthält die Abfrage nicht alle im Index enthaltenen Spalten so dass Werte praktisch übersprungen werden müssen.
Index Full Scan	Der Index wird komplett gelesen um die Adressen der Datensätze zu ermitteln. Dies ist effizient für Sortiervorgänge da Indizes per Definition sortiert sind.

Multiblock I/O

Index Fast Full Scan	Der Index wird komplett gelesen um die Adressen der Datensätze zu ermitteln. Diese werden dann ggf. gefiltert.
Full Table Scan	Der Zugriff auf die Tabellen erfolgt direkt über die Tabelle. Diese wird komplett gelesen. Im Nachgang werden ggf. Filterkriterien angewendet.

Join-Methoden

Neben den Zugriffsmethoden für einzelne Tabellen ist entscheidend wie die Inhalte mehrerer Tabellen verknüpft werden können. Dazu kennt der Optimizer verschiedene Methoden von denen im Folgenden die wesentlichen erläutert werden.

Nested Loop	Der Nested Loop ist eine geschachtelte Schleife. Für alle Datensätze der äußeren Tabelle (driving site) wird die innere Tabelle abgefragt. Der Join kann auf diese Art unvollständig ausgeführt werden. Das bedeutet, der Join liefert schnell die ersten Ergebnisse.
Hash Join	Beim Hash Join werden werden die Spalten aus der Join Bedingung mittels eines Hash Algorithmus verknüpft. Dies geschieht in einen einzigen Durchgang. Das bedeutet, dass nach Beendigung des Hash Joins die komplette Ergebnismenge vorliegt, er kann nicht teilweise durchgeführt werden. Hash Joins sind effizient um schnell alle Datensätze eines Joins zu ermitteln.
Sort Merge Join	Mit Sort Merge Joins werden die jeweiligen Spalten der Join Kriterien sortiert und dann zusammengefügt. Dies kommt zur Anwendung wenn der Join nicht nur aus Gleichheitsoperatoren besteht sondern auch <, >, >=, <= o.ä. beinhaltet.

Suche nach dem Plan

Nachdem nun einige der möglichen Ausführungsmethoden umrissen sind, stellt sich die Frage wie der Ausführungsplan eines SQL Statements nun tatsächlich aussieht. Um solch einen Ausführungsplan zu ermitteln gibt es verschiedene Möglichkeiten die nun beschrieben werden sollen.

EXPLAIN PLAN FOR

Diese Möglichkeit gibt es bereits seit vielen Oracle Versionen. Mittels dieses Befehls wird das nachfolgende SQL Statement an den Optimizer übergeben, der dann einen Ausführungsplan zurückgibt. Bei dieser Methode verfügt der Optimizer aber nicht über Informationen über aktuelle Werte von Bind Variablen und benutzt die Einstellungen der aktuellen Session. Das bedeutet der ermittelte Ausführungsplan kann sich vom Plan bei der tatsächlichen Ausführung signifikant unterscheiden. Man erhält also nur eine Idee wie die Abfrage vielleicht ausgeführt werden könnte. Viel mehr lässt sich damit aber nicht anfangen.

Package DBMS_XPLAN
Der tatsächlich verwendete Ausführungsplan von SQL Statements findet sich im Shared Pool. Dabei kann es durchaus mehrere verschiedene Pläne zu einem Statement geben, je nach Umgebung, Werten von Binds usw. Diese Varianten werden als *Child* bezeichnet. Mittels des Package DBMS_XPLAN ist es möglich den tatsächlichen Ausführungsplan aus dem Shared Pool zu ermitteln. Der Aufruf von DBMS_XPLAN.DISPLAY zeigt den Plan des zuletzt ausgeführten SQL Statements. Mittels DBMS_XPLAN.DISPLAY_CURSOR ist es möglich unter Angabe der SQL-ID und der Child Nummer den Plan eines bestimmten SQL Statements zu extrahieren sofern dieser sich noch im Shared Pool befindet. Der Ausführungsplan ist abrufbar sobald die Ausführung des Statements begonnen wurde, es muss dazu nicht komplett ausgeführt worden sein.

SQL*Plus, set autotrace traceonly
Die Einstellung von *set autotrace traceonly* im SQL*Plus zeigt nach der Ausführung des SQL den zugehörigen Ausführungsplan an. Dazu wird im Hintergrund das EXPLAIN PLAN FOR benutzt, es gelten daher die gleichen Randbedingungen wie bereits oben erläutert.

SQL Trace, tkprof
Oracle bietet die Möglichkeit mit DBMS_MONITOR.TRACE_ENABLE vielfältige Informationen zur Ausführung des SQL in eine Trace Datei zu schreiben. Sofern das SQL während des Tracings geparst wurde, findet sich der entstandene Ausführungsplan in der Trace Datei wieder. Wurde bereits ein passender Ausführungsplan im Shared Pool gefunden so wird dieser zwar verwendet aber nicht mit in der Trace Datei vermerkt. Nach dem Trace Lauf kann das Tool *tkprof* die Trace Datei auswerten und ermittelt auch die Ausführungspläne. Tkprof verfügt über einen zusätzlichen Parameter EXPLAIN=user/pwd, dieser führt nur ein EXPLAIN PLAN FOR zu den gefunden SQL Statements aus, unterliegt also den oben schon erwähnten Einschränkungen und ist somit nicht zu empfehlen.

Abschließend noch einmal eine kurze Gegenüberstellung der erläuterten Möglichkeiten:

Methode	**Realistisch?**	**Ausführung des SQL nötig?**
EXPLAIN PLAN FOR …	Nein	Nein
DBMS_XPLAN	Ja	Ja
Autotrace	Nein	Ja
DBMS_MONITOR	Ja	Ja

Warum ist mein Plan langsam
Wenn ein SQL Statement nicht die gewünschte Performance liefert, so ist dafür oft ein ungünstiger Ausführungsplan die Ursache. Das bedeutet der Optimizer hat sich verschätzt. Es gilt also zu ermitteln an welchen Punkten sich die Schätzung des Optimizers von der tatsächlichen Ausführung unterscheidet. Dazu gibt es die Möglichkeit erweiterte Statistiken zur Ausführung des SQLs zu sammeln. Für ein einzelnes Statement lässt sich das mittels des Hints /*+GATHER_PLAN_STATISTICS*/ erreichen oder durch den Parameter *statistics_level=all* auf Sessionebene oder für das gesamte System aktivieren. Die damit ermittelten Statistiken lassen sich per DBMS_XPLAN oder auch mit SQL Trace und tkprof auswerten. Es kann dann die vom Optimizer geschätzte Datenmenge mit der tatsächlich bei der Ausführung entstandenen Menge verglichen werden. Dies ermöglicht Rückschlüsse um die Ursache der Fehlschätzung zu finden.
Bei der Auswertung mittels SQL Trace und tkprof wird die tatsächliche Datenmenge in der ersten Spalte (Rows 1st) mit der geschätzen Datenmenge (card=<Zahl>) verglichen. Hier ein Beispiel:

```
SQL> exec dbms_monitor.session_trace_enable(waits=>true, binds=>true,
                                            plan_stat=>'ALL_EXECUTIONS')

PL/SQL-Prozedur erfolgreich abgeschlossen.

SQL> SELECT prod_category, AVG(amount_sold)
  2  FROM sales s, products p
  3  WHERE p.prod_id = s.prod_id
  4  GROUP BY prod_category;

Abgelaufen: 00:00:01.95

SQL> exec dbms_monitor.session_trace_disable;

PL/SQL-Prozedur erfolgreich abgeschlossen.

tkprof marco_ora_12330.trc marco_ora_12330.trc.tkp sys=no


Rows (1st) Row Source Operation
---------- ---------------------------------------------------
         5 HASH GROUP BY (cr=1641 pr=1095 pw=1095 time=1830994 us cost=8
                          size=150 card=5)
    918843  HASH JOIN    (cr=1641 pr=1095 pw=1095 time=3068320 us cost=7
                          size=300 card=10)
    918843   PARTITION RANGE ALL PARTITION: 1 28
                         (cr=1635 pr=0 pw=0 time=5077500 us cost=4
                          size=90 card=10)
    918843    TABLE ACCESS FULL SALES PARTITION: 1 28
                         (cr=1635 pr=0 pw=0 time=1980264 us cost=4
                          size=90 card=10)
        72   VIEW  index$_join$_002
                         (cr=6 pr=0 pw=0 time=4386 us cost=3  size=1512
                          card=72)
        72    HASH JOIN  (cr=6 pr=0 pw=0 time=4094 us)
        72     INDEX FAST FULL SCAN PRODUCTS_PK
                         (cr=3 pr=0 pw=0 time=187 us cost=1 size=1512
                          card=72)
        72     INDEX FAST FULL SCAN PRODUCTS_PROD_CAT_IX
                         (cr=3 pr=0 pw=0 time=454 us  cost=1 size=1512
                          card=72)
```

Das selbe Beispiel mit DBMS_XPLAN stellt die betreffenden Informationen direkt in den Spalten E-Rows (estimated rows) und A-Rows (actual rows) gegenüber. Die Spalte „Name“ mit den Bezeichnern der Tabellen und Indexe wurde aus Platzgründen weggelassen:

```
SQL> alter session set statistics_level=all;

Session wurde geändert.

SQL> SELECT prod_category, AVG(amount_sold)
  2  FROM sales s, products p
  3  WHERE p.prod_id = s.prod_id
  4  GROUP BY prod_category;

Abgelaufen: 00:00:15.22
```

```
SQL> select * from table(dbms_xplan.display_cursor(format=>'ALLSTATS LAST'));

-------------------------------------------------------------------------------
| Id  | Operation                | Starts | E-Rows | A-Rows |   A-Time   |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |      1 |        |      5 |00:00:15.11 |
|   1 |  HASH GROUP BY           |      1 |      5 |      5 |00:00:15.11 |
|*  2 |   HASH JOIN              |      1 |     10 |    918K|00:00:13.21 |
|   3 |    PARTITION RANGE ALL   |      1 |     10 |    918K|00:00:05.21 |
|   4 |     TABLE ACCESS FULL    |     28 |     10 |    918K|00:00:02.18 |
|   5 |    VIEW                  |      1 |     72 |     72 |00:00:00.01 |
|*  6 |     HASH JOIN            |      1 |        |     72 |00:00:00.01 |
|   7 |      INDEX FAST FULL SCAN|      1 |     72 |     72 |00:00:00.01 |
|   8 |      INDEX FAST FULL SCAN|      1 |     72 |     72 |00:00:00.01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("P"."PROD_ID"="S"."PROD_ID")
   6 - access(ROWID=ROWID)
```

Zudem erkennt man in der Ausgabe von DBMS_XPLAN in der Spalte A-Time sehr gut die tatsächliche Ausführungszeit der einzelnen Schritte, kann so also ermitteln wo die Zeit bei der Ausführung verloren geht.

Was geht besser – Index und NULL

Abschließend sollen noch einige häufige Fallstricke und zugehörige Lösungsansätze aufgezeigt werden. Z.B. werden NULL Werte in Spalten nicht indiziert, es sei denn der Index umfasst mehrere Spalten und eine nachfolgende Spalte ist als NOT NULL definiert. Nur dann werden die NULL Werte der vorgelagerten Spalten mit im Index aufgenommen. Ein

```
create index ix_mgr on emp (mgr);

select * from emp where mgr is null;
```

führt zu

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    40 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMP  |     1 |    40 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

Wird der Index mit einer nachfolgenden NOT NULL Spalte angelegt, das kann auch eine Konstante sein, ergibt sich folgendes Bild:

```
create index ix_mgr on emp (mgr, 0);

select * from emp where mgr is null;
```

was zu diesem Plan mit Benutzung des Index führt:

```
-------------------------------------------------------------------------------------
| Id  | Operation                   | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |        |     1 |    40 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP    |     1 |    40 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | IX_MGR |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("MGR" IS NULL)
```

Was geht besser – Index und Texte

Ein weiterer häufiger Fehler bei der Indizierung von Textspalten ist die Schreibweise. Ist diese nicht bekannt, wird häufig mit *upper* oder *lower* gearbeitet um die gewünschten Ergebnisse zu finden. Das führt dann zu Full Table Scans:

```
create index ix_ename on emp (ename);

select * from emp where upper(ename)='SCOTT';
```

```
-----------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    40 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMP  |     1 |    40 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(UPPER("ENAME")='SCOTT')
```

Dies lässt sich einfach mit einem Function Based Index umgehen. Diese sind seit Oracle 9.2 Teil der Standard Edition und somit auch ohne Einschränkungen verfügbar:

```
create index ix_ename on emp ( upper(ename) );

select * from emp where upper(ename)='SCOTT';

-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |          |     1 |    40 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP      |     1 |    40 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | IX_ENAME |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access(UPPER("ENAME")='SCOTT')
```

Was geht besser – Spaltenzusammenhänge

Häufig bereiten auch die Zusammenhänge zwischen Spalten Probleme bei den Schätzungen. Der Optimizer kennt zwar die Wertverteilung in den einzelnen Spalten, nicht aber die Zusammenhänge. Er kann z.B. nicht wissen, dass im Department „Sales“ viele „Salesman“-Jobs vertreten sind, in allen anderen Departments aber nicht. Der Optimizer multipliziert einfach die einzelnen Häufigkeiten für seine Schätzung:

```
SQL> select * from emp where job='SALESMAN' and deptno=30;

------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    40 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMP  |     1 |    40 |     3   (0)| 00:00:01 |
------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("JOB"='SALESMAN' AND "DEPTNO"=30)
```

Diese Zusammenhänge kann man dem Optimizer mit erweiterten Statistiken bekannt machen, dann schätzt der Optimizer auch die korrekte Datenmenge:

```
SQL> exec dbms_stats.gather_table_stats(user, 'EMP', method_opt=>'for columns
(job,deptno)');

SQL> select * from emp where job='SALESMAN' and deptno=30;

------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     4 |   160 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMP  |     4 |   160 |     3   (0)| 00:00:01 |
------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("JOB"='SALESMAN' AND "DEPTNO"=30)
```

Was geht besser – Bind Variablen, Cursor Sharing

Um Ausführungspläne wiederverwendbar zu machen, müssen Bind Variablen in den SQL Statements benutzt werden. Vor Oracle 11 konnte der Optimizer zwar beim ersten Ausführen eines Statements den aktuellen Wert einer Bind Variable abfragen und daraus entsprechend einen passenden Ausführungsplan ableiten. Sind die Daten in der betreffenden Spalte nun ungleich verteilt, fällt die Entscheidung des Optimizers je nach Wert in die eine (Index) oder andere (Table Scan) Richtung aus. Danach steht der Ausführungsplan so im Cache und wird für alle weiteren Aufrufe unabhängig von der Belegung der Bind Variable wiederverwendet. Ab Oracle 11 erkennt der Optimizer solche Konstellationen in dem er bei der Ausführung per „cardinality feedback“ eine Rückmeldung über die verarbeitete Datenmenge erhält und generiert nach Bedarf neue Ausführungspläne für verschiedene Werte der Bind Variablen. Nachfolgend wird dieser als „adaptive cursor sharing“ bezeichnete Mechanismus an einem einfachen Beispiel gezeigt.

```
SQL> create table viel as select lpad('-', 2000) breit, 10 nummer from dual
  2> connect by level < 1000;

SQL> update viel set nummer =1 where rownum < 20;

SQL> commit;

SQL> create index ix_viel on viel(nummer);

SQL> exec dbms_stats.gather_table_stats(user, 'VIEL', cascade=>true,
method_opt=>'for all columns size 254');

SQL> exec :id := 1;

SQL> select /*TEST*/ * from viel where nummer = :id;

19 Zeilen ausgewählt.

SQL> select sql_id ,child_number,IS_BIND_AWARE, IS_BIND_SENSITIVE,
  2> executions, buffer_gets from v$sql where sql_id= :sql_id;

SQL_ID        CHILD_NUMBER I I   EXECUTIONS  BUFFER_GETS
------------- ------------ - - ------------ ------------
gnacnx2xb9pxj            0 N Y            1           13
```

```
SQL_ID  gnacnx2xb9pxj, child number 0
-------------------------------------
select /*TEST*/ * from viel where nummer = :id

Plan hash value: 1116875173

-----------------------------------------------------------------------------------------
| Id  | Operation                   | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |         |       |       |     8 (100)|          |
|   1 |  TABLE ACCESS BY INDEX ROWID| VIEL    |    19 | 38076 |     8   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | IX_VIEL |    19 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("NUMMER"=:ID)
```

Man sieht hier, dass nach einer Ausführung ein Cursor entstanden ist und der Optimizer das Vorhandensein von Bind Variablen berücksichtigt (is_bind_sensitive). Nun wird das Statement mit einem anderen Wert ausgeführt:

```
SQL> exec :id := 10:

SQL> select /*TEST*/ * from viel where nummer = :id;

980 Zeilen ausgewählt.


SQL> select sql_id ,child_number,IS_BIND_AWARE, IS_BIND_SENSITIVE,executions,
  2> buffer_gets from v$sql where sql_id= :sql_id;

SQL_ID        CHILD_NUMBER I I   EXECUTIONS  BUFFER_GETS
------------- ------------ - - ------------ ------------
gnacnx2xb9pxj            0 N Y            2          475
```

An dem Cursor hat sich erst einmal nichts geändert. So werden Ausreißer und andere Einzeleffekte ausgeschlossen. Erst bei der erneuten Ausführung reagiert der Optimizer da er erneut ein deutlich verändertes Verhalten des Statements feststellt:

```
SQL> select /*TEST*/ * from viel where nummer = :id;

980 Zeilen ausgewählt.

SQL> select sql_id ,child_number,IS_BIND_AWARE, IS_BIND_SENSITIVE,executions,
  2> buffer_gets from v$sql where sql_id= :sql_id;

SQL_ID        CHILD_NUMBER I I   EXECUTIONS  BUFFER_GETS
------------- ------------ - - ------------ ------------
gnacnx2xb9pxj            0 N Y            2          475
gnacnx2xb9pxj            1 Y Y            1          403
```

Es wird ein neuer Child Cursor erstellt und der Optimizer markiert diesen nun als „is_bind_aware“ da die verschiedenen Werte für die Bind Variable zu verschiedenen Antwortmengen führen und damit ein anderer Ausführungsplan in Betracht kommt.

```
SQL_ID  gnacnx2xb9pxj, child number 1
-------------------------------------
select /*TEST*/ * from viel where nummer = :id

Plan hash value: 2088219638

-------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |       |       |    96 (100)|          |
|*  1 |  TABLE ACCESS FULL| VIEL |   980 |  1917K|    96   (0)| 00:00:02 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("NUMMER"=:ID)
```

Nun wird der Optimizer für jede weitere Ausführung den passenden Ausführungsplan abhängig vom Wert der Bind Variablen wählen.

Fazit
Der Optimizer unterliegt einer stetigen Weiterentwicklung und kann immer bessere Ausführungspläne für wechselnde Anforderungen generieren. Das enthebt den Entwickler aber nicht von der Pflicht sich mit den Konzepten zu befassen und dem Optimizer entsprechend das nötige Handwerkszeug bereitzustellen.

Kontaktadresse:
Marco Mischke
Robotron Datenbank-Software GmbH
Stuttgarter Str. 29
D-01189 Dresden

Telefon: +49 (0) 351-25 85 9 2884
Fax: +49 (0) 351-25 85 9 3696
E-Mail marco.mischke@robotron.de
Internet: www.robotron.de