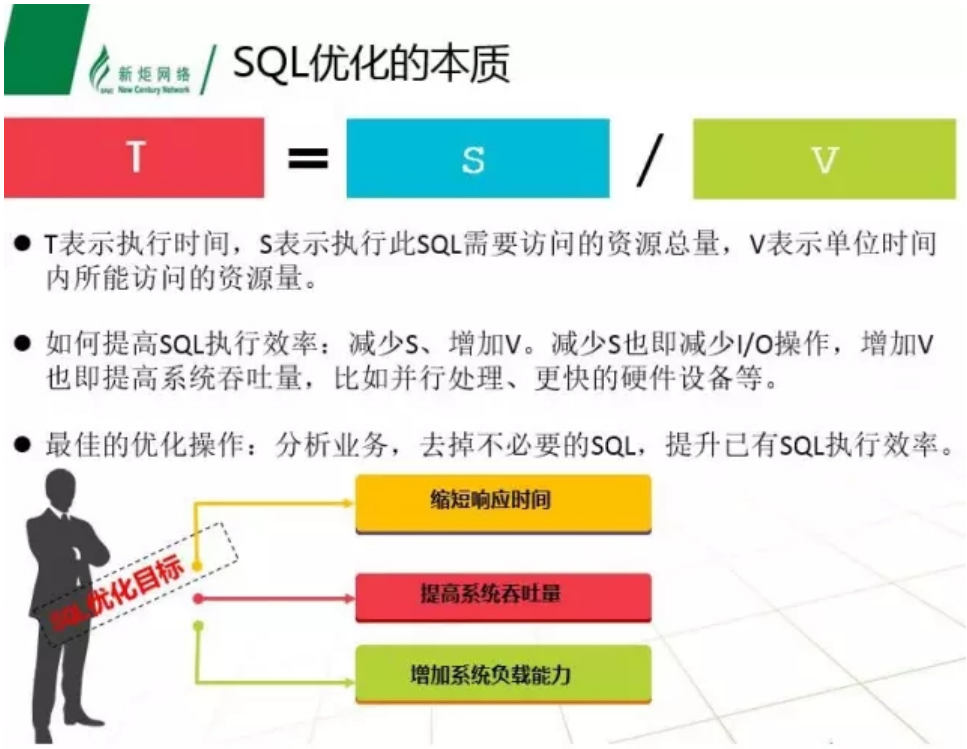
【重磅干货】看了此文，Oracle SQL优化文章不必再看！

dbaplus.cn/news-10-80-1.html

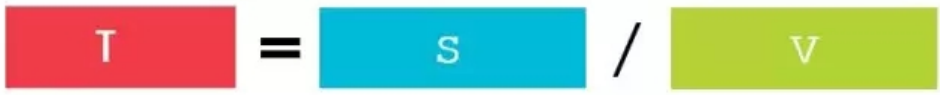
目录

—

SQL优化的本质



一般来说，SQL优化是让SQL运行得更快，使SQL更快的方式有很多，比如提高索引的使用效率，或者并行查询。可以看到里面的公式：



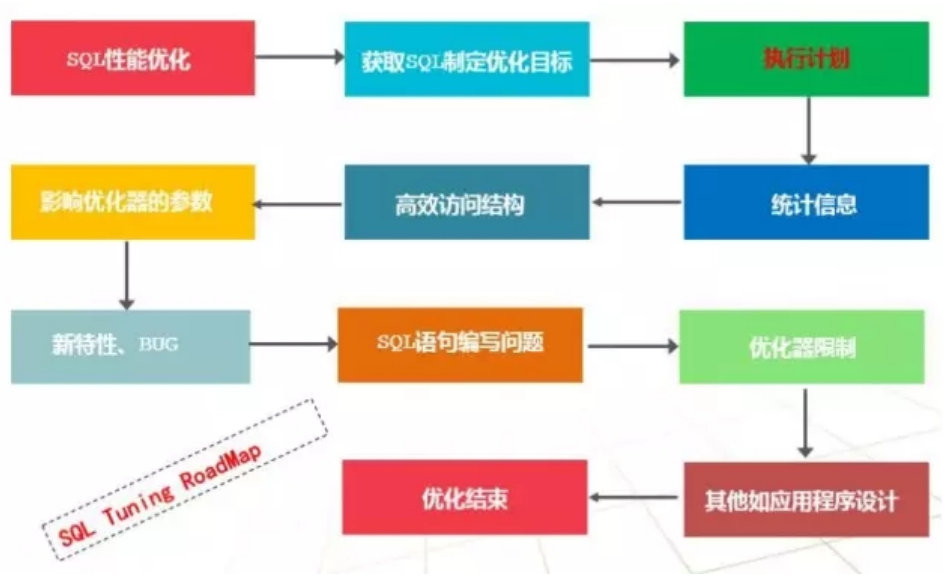
执行效率或者一般说的执行时间，是和完成一次SQL所需要访问的资源总量（S）成正比以及单位时间内能够访问的资源量（V）成反比，S越大，效率越低，V越大效率越高。 比如通过并行查询，则可以提升单位时间内访问的资源量。

当然，这仅仅是从执行时间上考虑，SQL优化肯定不仅仅是执行时间降低，应该是资源使用与执行时间降低之间寻求一种平衡，否则，盲目并行，可能提升不了效率，反而让系统资源消耗殆尽。

最终来说，SQL优化的本质就是：1、缩短响应时间；2、提升系统吞吐量；3、提升系统负载能力。要使用多种手段，在提升系统吞吐量和增加系统负载能力，提高单个SQL效率之间寻求一种平衡。就是要尽量减少一条SQL需要访问的资源总量，比如走索引更好，那么不要使用全表扫描。

二
SQL优化Road Map

一条SQL的优化路线图如下所示：



具体操作步骤：

2.1 制定SQL优化目标

获取待优化SQL、制定优化目标：从AWR、ASH、ORA工具等主动发现有问题的SQL、用户报告有性能问题DBA介入等，通过对SQL的执行情况进行了解，先初步制定SQL的优化目标。

2.2 检查执行计划

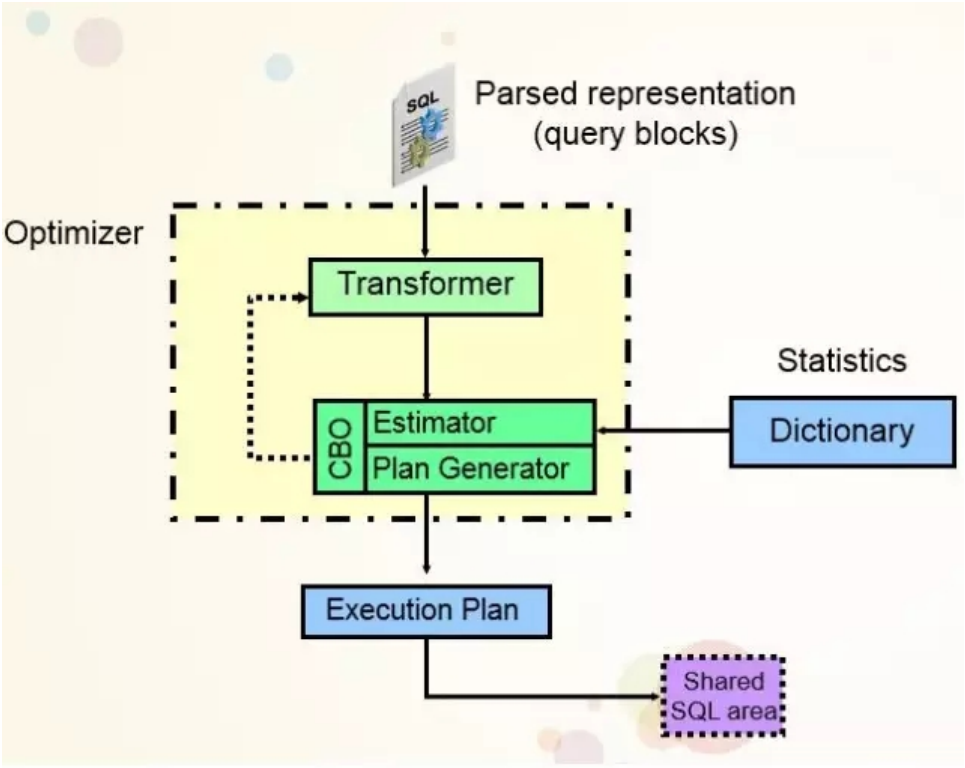
explain工具、sql*plus autotrace、dbms_xplan、10046、10053、awrsqrpt.sql等。 执行计划是我们进行SQL优化的核心内容，无计划，不优化。看执行计划有一些技巧，也有很多方式，各种方式之间是有区别的。

2.3 检查统计信息

ORACLE使用DBMS_STATS包对统计信息进行管理，涉及系统统计信息、表、列、索引、分区等对象的统计信息，统计信息是SQL能够使用正确执行计划的保证。我们知道，ORACLE CBO优化器是利用统计信息来判断正确的执行路径，JOIN方式的，因此，准确的统计信息是产生正确执行计划的首要条件。

可以从这个图看出，一条SQL产生执行计划需要经过哪些步骤，在我看来：1、正确的查询转换；2、准确的统计信息，是产生正确执行计划的重要保证。当然，还有BUG，或优化器限制等也会导致SQL效率低下，无法产生正确的执行计划。

如图所示：



2.4 检查高效访问结构

重要的访问结构，诸如索引、分区等能够快速提高SQL执行效率。表存储的数据本身，如碎片过多、数据倾斜严重、数据存储离散度大，也会影响效率。

2.5 检查影响优化器的参数

optimizer_mode、optimizer_index_cost_adj、optimizer_dynamic sampling、_optimizer_mjc_enabled、_optimizer_cost_based_transformation、hash_join_enable等对SQL执行计划影响较大。比如有时候我们通过禁用 _optimizer_mjc_enabled 参数，让执行计划不要使用笛卡尔积来提升效率，因为这个参数开启有很多问题，所以一般生产库都要求禁用。

还有什么能够影响执行计划呢？对，new features,每个版本的new features，引入的目的都是好的，但是实际使用中，可能触发BUG。比如11g的ACS(自适应游标共享）、automatic serial direct path（自动串行直接路径读）、extended statistics、SQL query result cache等。有的新特性会导致问题，所以需要谨慎使用。

比如11g adaptive cursor sharing,自适应游标共享，它的引入是为了解决使用绑定变量与数据倾斜值，要产生多样性执行计划。因为绑定变量是为了共享执行计划，但是数据倾斜了，有的值要求走索引，有的值要求走全表，这样与使用绑定变量就产生了矛盾。以前是通过cursor_sharing=similar这样的设置可以解决，但是有很多BUG，会产生version count过高的问题，或者我们对不同的值（如果值很少），可以写多条SQL来解决，这都不是好的方案，11g acs引入就是为了解决这些问题，让这些东西交给oracle来做。但是事与愿违，以后你们遇到执行计划一会变一下，有快有慢，首先可以检查acs有没有关闭。

alter system set “_optimizer_extended_cursor_sharing_rel”=’NONE';

2.6 SQL语句编写问题

SQL语句结构复杂、使用了不合理的语法，比如UNION代替UNION ALL都可能导致性能低下。 并不是说ORACLE优化器很强大了，我们就可以随便写SQL了，那是不正确的。SQL是一门编程语言，它能够执行的快，是有一些普遍的规则的，遵循这种编程语言特性，简化语句，才能写出好的程序。SQL语句编写出了问题，我们就需要改写，就需要调整业务，改涉及等。

2.7 SQL优化器限制导致的执行计划差

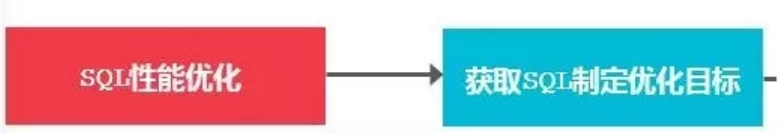
这个很重要，统计信息准确，SQL也不复杂，索引也有。。。都满足，为什么我的SQL还是差，那么得考虑优化器限制因素了。这里说1点常见的执行计划限制，当semi join与or连用的时候（也就是exists(subquery) or ...或者in (subquery) or...，如果执行计划中因为OR导致有FILTER操作符，就得注意了，可能慢的因素就和OR有关。这时候我们得改写SQL，当然改写为UNION或UNION ALL了。

OK，以上全部检查完毕，我的系统还是很差，功能还是很慢，或者已经无法从SQL本身进行调整提升性能了，那咋办？优化设计，这是终极方法。有些东西不优化设计是无法解决的，比如业务高峰期跑了一堆SQL，CPU已经很吃紧，又不给增加，突然上线一个耗资源的业务，其他SQL已无法调整。那只能优化设计，比如有些耗资源的业务可以换时间段执行等。

以上几点，是我们进行优化需要考虑的地方，可以逐步检查。当然，80%到90%的纯SQL性能调整，我们通过建立索引，收集正确统计信息，改写避免优化器限制，已经能够解决了。

三
SQL优化案例

看第一个获取待优化的SQL.....如果主动优化，一般从AWR、ASH等里面找到性能差的SQL，然后优化之。



看一个案例，占CPU 72%的SQL来自于同一模块，第一行是存储过程，通过下面绿色框住的SQL与第一行比较，主要通过EXECUTION，基本判断下面的绿色框住的SQL就是那个存储过程中的。也可以和业务确认下，OK，这些SQL的执行频次很高，因为是营销业务，如果要优化，就得搞定这些SQL。

这些SQL，单条SQL的buffer gets也就1000多点，效率还是很高的，但是因为执行的太过于频繁，所以资源消耗极大，因此，得检查下，能不能更优呢？

以第1条SQL：58q183atbusat为例：

```
SELECT B.ACT_ID,

B.ACT_NAME,

B.TASK_ID,

B.MKT_DICTION,

B.CUST_GROUP_ID,

NVL(B.ATTEST_FLAG, 'N'),

NVL(B.DOUWIN_FLAG, 'N'),

B.CHN_DESC,

NVL(B.SIGN_FLAG, 'N'),

B.MAX_EXECUTE_NUM

FROM (SELECT DISTINCT (ACT_ID)

FROM MK_RULECHN_REL

WHERE CHN_STATUS = '04'

AND CHN_TYPE = :B1) A,

TABLE(CAST(:B2 AS TYPE_MK_ACTIONINFO_TABLE)) B

WHERE A.ACT_ID = B.ACT_ID
```

SQL其实很简单，一个查询构建的A表，一个TABLE函数构建的B表关联..... 不知道大家对这个TABLE函数熟悉不熟悉？也就是将一个集合转成表，是PL/SQL里的东西

那个collection部分就是TABLE函数，下面的表走了全表扫描：

```
Plan hash value: 918180822
-------------------------------------------------------------------------------------------
| Id  | Operation                          | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |               |       |       |   839 (100)|          |
|*  1 |  HASH JOIN                         |               |  5784 | 75192 |   839   (5)| 00:00:05 |
|   2 |   COLLECTION ITERATOR PICKLER FETCH|               |       |       |            |          |
|   3 |   VIEW                             |               | 12361 |  132K |   818   (5)| 00:00:05 |
|   4 |    HASH UNIQUE                     |               | 12361 |  301K |   818   (5)| 00:00:05 |
|*  5 |     TABLE ACCESS FULL              | MK_RULECHN_REL | 21104 |  515K|   814   (4)| 00:00:05 |
-------------------------------------------------------------------------------------------

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

   1 - access("A"."ACT_ID"=SYS_OP_ATG(VALUE(KOKBF$),1,2,2))
   5 - filter(("CHN_TYPE"=:B1 AND "CHN_STATUS"='04'))
```

按步骤检查，发现不了问题，但是知道，可能是因为HASH JOIN导致全表扫描的问题，是否走NESTED LOOPS+INDEX更好，很显然，要检查TABLE函数大概返回多少行。

经过确认，最多也就返回200-300行，最终结果集也是几百行而已。

那么猜测，问题就在于TABLE函数，走了HASH JOIN，上面的执行计划，TABLE函数部分，ROWS为空。

来单独检查一把：返回8168行，返回8000多行，足以导致走HASH JOIN了....而事实，我们至多返回200-300行：

```
SQL> explain plan for
  2  SELECT *
  3  FROM TABLE(CAST(:B2 AS TYPE_MK_ACTIONINFO_TABLE));
Explained.
Elapsed: 00:00:00.01

SQL> select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 1692170009
--------------------------------------------------------------------------------
| Id  | Operation                          | Name | Rows  | Bytes | Cost (%CPU)
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |      |  8168 | 16336 |    14   (0)
|   1 |  COLLECTION ITERATOR PICKLER FETCH |      |       |       |
--------------------------------------------------------------------------------
```

所以每个步骤返回的行，是JOIN方式选择的重要因素，可以谷歌一把，TABLE函数返回8168就是个固定值，block_size=8K的时候就是这么大，可以说，这是ORACLE的一个限制。

只要你用了TABLE函数，就偏向于走HASH JOIN了

http://www.oracle-developer.net/display.php?id=427 有兴趣的可以看这个链接的内容。

解决方式很多了，也就是要走NESTED LOOPS+index, 既然8168很大，那么我们就让优化器知道TABLE函数返回的行少点，才百行左右。

以下些都可以，当然也可以使用hint:use_nl等

CARDINALITY hint (9i+) undocumented;

OPT_ESTIMATE hint (10g+) undocumented;

DYNAMIC_SAMPLING hint (11.1.0.7+);

Extensible Optimiser (10g+).

因为SQL的SELECT部分只访问B，全部来自于TABLE函数，所以改写为子查询就可以了，使用子查询，自然distinct也就没有必要了，因为是semi join(半连接）。

最终改写使用cardinality hint让优化器知道B返回的行只有100行，你给我走NESTED LOOPS+INDEX，然后解决。

原来的sql：

修改后的sql：

```
SELECT B.ACT_ID,
    B.ACT_NAME,
    B.TASK_ID,
    B.MKT_DICTION,
    B.CUST_GROUP_ID,
    NVL(B.ATTEST_FLAG, 'N'),
    NVL(B.DOUWIN_FLAG, 'N'),
    B.CHN_DESC,
    NVL(B.SIGN_FLAG, 'N'),
    B.MAX_EXECUTE_NUM
 FROM (SELECT DISTINCT (ACT_ID)
     FROM MK_RULECHN_REL
    WHERE CHN_STATUS = '04'
      AND CHN_TYPE = :B1) A,
    TABLE(CAST(:B2 AS TYPE_MK_ACTIONINFO_TABLE)) B
WHERE A.ACT_ID = B.ACT_ID
```

```
SELECT/*+cardinality(b 100)*/ B.ACT_ID, B.ACT_NAME, B.TASK_ID, B.MKT_DICTION, B.CUST_GROUP_ID,NVL(B.ATTEST_FLAG,'N'), NVL(B.DOUWIN_FLAG,'N'), B.CHN_DESC, NVL(B.SIGN_FLAG,'N'),B.MAX_EXECUTE_NUM
FROM TABLE(CAST( B2 AS TYPE_MK_ACTIONINFO_TABLE)) B
 WHERE EXISTS(SELECT 1 FROM MK_RULECHN_REL A
  WHERE A.CHN_STATUS = '04'
  AND A.CHN_TYPE = B1
  AND A.ACT_ID = B.ACT_ID
  );
```

效率提升几十倍：

```
lan hash value: 884413475

-----------------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name              | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------------
|   1 | NESTED LOOPS SEMI                   |                   |      1 |      1 |     75 |00:00:00.01 |     227 |
|   2 |  COLLECTION ITERATOR PICKLER FETCH  |                   |      1 |        |     75 |00:00:00.01 |       0 |
|*  3 |  TABLE ACCESS BY INDEX ROWID        | MK_RULECHN_REL    |     75 |      1 |     75 |00:00:00.01 |     227 |
|*  4 |   INDEX RANGE SCAN                  | PK_MK_RULECHN_REL |     75 |      2 |     75 |00:00:00.01 |     152 |
-----------------------------------------------------------------------------------------------------------------

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

   3 - filter("A"."CHN_STATUS"='04')
   4 - access("A"."ACT_ID"=SYS_OP_ATG(VALUE(KOKBF$),1,2,2) AND "A"."CHN_TYPE"=:B2)
       filter("A"."CHN_TYPE"=:B2)
```


一个占72%的应用，我们提升几十倍后，那对系统性能明显是极好的。最终，在执行次数增加50%的情况下，w4sd08pa主机CPU使用率由原来的高峰期平均47%的使用率降低为23%。

这个问题能够解决有两个方面：

1、猜测并测试优化器的限制（table函数固定返回行8168)；2、实际返回的行200-300。两者缺一不可。如果实际返回的行就是几千上万，那么，单纯通过优化SQL，也是无法取得良好效果的。

扫描文末二维码，关注DBA+社群微信公众号（dbaplus），可下载DBA+社群技术沙龙、OOW大会、2015GOPS、DCon2015等技术盛典PPT。

四
SQL执行计划获取

执行计划就是SQL调优的核心，上面的SQL也是通过看到执行计划走HASH JOIN可能有问题出发的。

那么首先要搞定2个问题：

1、如何获取我要的执行计划（准确的计划）；

2、怎么看懂并找出执行计划里的问题。

4.1 如何获取准确的执行计划

获取SQL执行计划的方式：

EXPLAIN PLAN

估算

忽略绑定变量

非执行

SQL_TRACE

真实计划，需要用TKPROF工具解析

可以获得绑定变量值

EVENT 10053

真实计划

研究执行计划产生的原因

AUTOTRACE

内部使用EXPLAIN PLAN

DBMS_XPLAN

dbms_xplan.display_cursor

dbms_xplan.display_awr

真实计划

OTHERS

如awrsqrpt、sqlt、pl/sql、sql developer、toad等

大家一般怎么获取执行计划？我一般用的较多的是dbms_xplan.display_cursor，优点很明显：1、获取的是真实执行的计划；2、多种参数。还可以获取绑定变量的值方便验证。

10053是检查优化器行为的，实在搞不懂为什么走那个计划可以看看，用得较少。

10046可以检查一些等待事件的内容，也可以获取绑定变量，一般用得也比较少。

set autotrace traceonly或者explain,他们的执行计划是同一来源，记住，都来自plan_table，是估算的，可能不是真实执行的计划，可能是不准的。

所以，你看得不对劲了，就得质疑它的准确性，autotrace traceonly的好处是可以看到一致性读，物理读，返回行等，这是真实的。因为可以用一致性读，物理读来验证优化效果

其他的，比如awrsqrpt等都可以获取执行计划，不过我很少用，特别是plsq developer这种工具，F5看计划，我几乎是不用的，他也是plan table里的估算计划。如果很长，那无法分析。

建议大家看真实的计划，说一点，我经常通过alter session set statistics_level=all或者gather_plan_statistics hint，然后执行sql,然后通过

select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));来看实际执行的信息

好处很明显，能够看到执行计划每步的E-ROWS（估算的行）,A-ROWS（真实的行）,STARTS,BUFFER GETS，A-TIME（真实的执行时间）等信息。。。我们通过对比估算的与真实的差距，可以判断哪些表统计信息可能有问题，执行计划是不是走错了，省的我们自己根据谓词去计算这步导致返回多少行。

注意一点，如果一SQL执行很长时间，通过上面的方式来看计划，我们是可以终止的，比如执行2小时执行不玩的SQL，一般我没有耐心，最多5分钟，我就终止。终止完，通过display_cursor也是可以看出执行信息的。

比如某个步骤执行100万次，我这条SQL才能执行完，要3小时才可以，我5分钟执行了100次，我终止了SQL我要看的就是一个比例情况，可以通过这个比例来判断，哪个步骤耗的时间最长，哪里大概有问题，然后解决。

优化器很多限制的，比如刚才的TABLE函数固定返回8168，或者算法限制.....很多不准的，如果算法算出来的与真实差别很大，那可能就会导致问题。统计信息有时候也无法收集准确的，比如直方图，就有很多问题，所以12c的直方图多了几种....之前只有等高和等频直方图。

刚才的set statistics_level直接写会输出结果，我们可以让他不输出结果：

1、sql内容放到文件中，前面加上set termout off （这样可以对输出结果不输出）

2、然后display_cursor文件中

```
set termout on
select *    from table(dbms_xplan.display_cursor(null,null,'allstats last'));

--sqlplus 中要首先 set serveroutput off
dingjun123@ORCL&gt; set serveroutput off;
dingjun123@ORCL&gt; set timing on;
dingjun123@ORCL&gt; set line 200;
dingjun123@ORCL&gt; set pagesize 999;
dingjun123@ORCL&gt; alter session set statistics_level=all;

Session altered.

Elapsed: 00:00:00.00
dingjun123@ORCL&gt; @test
dingjun123@ORCL&gt; @display_cursor

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------------------
--------------------

SQL_ID d4a3yq3r7j8x2, child number 0
-------------------------------------
SELECT a.cust_id, a.cust_first_name, (SELECT COUNT(*) FROM sh.sales
b WHERE a.cust_id=b.cust_id ) cnt, (SELECT MAX(amount_sold) FROM
sh.sales b WHERE a.cust_id=b.cust_id ) max_amount FROM sh.customers a

Plan hash value: 3553006969

------------------------------------------------------------------------------------------------------------------------
| Id | Operation | Name | Starts | E-Rows | A-Rows | A-Time | Buffers |
------------------------------------------------------------------------------------------------------------------------
| 0 | SELECT STATEMENT | | 1 | | 55500 |00:00:00.03 | 5059 |
| 1 | SORT AGGREGATE | | 55500 | 1 | 55500 |00:00:03.40 | 2451K|
| 2 | PARTITION RANGE ALL | | 55500 | 130 | 918K|00:00:03.33 | 2451K|
| 3 | BITMAP CONVERSION TO ROWIDS | | 1554K| 130 | 918K|00:00:03.02 | 2451K|
|* 4 | BITMAP INDEX SINGLE VALUE | SALES_CUST_BIX | 1554K| | 35808 |00:00:02.53 |
2451K|
| 5 | SORT AGGREGATE | | 55500 | 1 | 55500 |00:00:04.70 | 3063K|
| 6 | PARTITION RANGE ALL | | 55
```

用这种东西看执行计划，有时候很方便找出问题,否则我们自己得手动根据每个步骤对应的谓词，自己写SQL去计算真实返回的行，然后再来比较，用这个，ORACLE全帮我们干好了。

4.2 看懂执行计划执行顺序

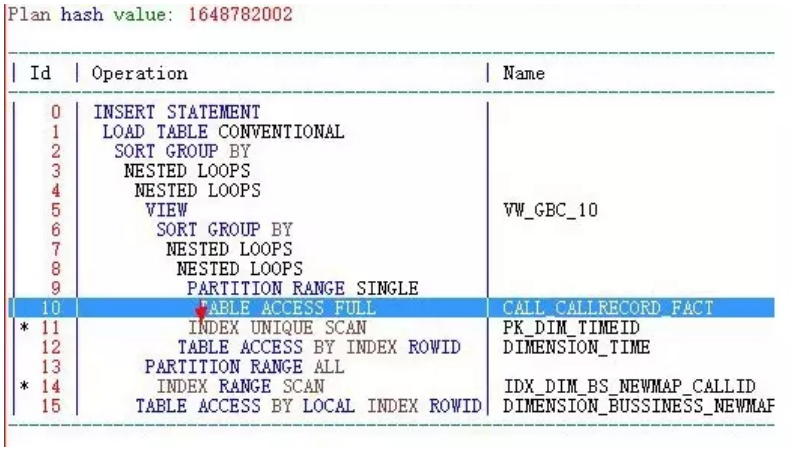
一般怎么看执行计划呢？

COPY到UE里去。

```
Plan hash value: 1648782002

| Id  | Operation                          | Name                      | Rows |
|   0 | INSERT STATEMENT                   |                           |      |
|   1 |  LOAD TABLE CONVENTIONAL           |                           |      |
|   2 |   SORT GROUP BY                    |                           |    1 |
|   3 |    NESTED LOOPS                    |                           |      |
|   4 |     NESTED LOOPS                   |                           |    1 |
|   5 |      VIEW                          | VW_GBC_10                 |    1 |
|   6 |       SORT GROUP BY                |                           |    1 |
|   7 |        NESTED LOOPS                |                           |      |
|   8 |         NESTED LOOPS               |                           |    1 |
|   9 |          PARTITION RANGE SINGLE    |                           |    1 |
|  10 |           TABLE ACCESS FULL        | CALL_CALLRECORD_FACT      |    1 |
| * 11 |          INDEX UNIQUE SCAN         | PK_DIM_TIMEID             |    1 |
|  12 |         TABLE ACCESS BY INDEX ROWID| DIMENSION_TIME            |    1 |
|  13 |       PARTITION RANGE ALL          |                           |    3 |
| * 14 |        INDEX RANGE SCAN            | IDX_DIM_BS_NEWMAP_CALLID  |    3 |
|  15 |      TABLE ACCESS BY LOCAL INDEX ROWID| DIMENSION_BUSSINESS_NEWMAP | 1 |

Predicate Information (identified by operation id):

  11 - access("CF"."TIMEID"="DT"."TIMEID")
  14 - access("ITEM_1"="DBNM"."CALLID")
```

用光标大法，找到入口，最先执行的，光标定位ID=0的，然后一直缩进向下，如果被挡住了，那么这部分就是入口了。

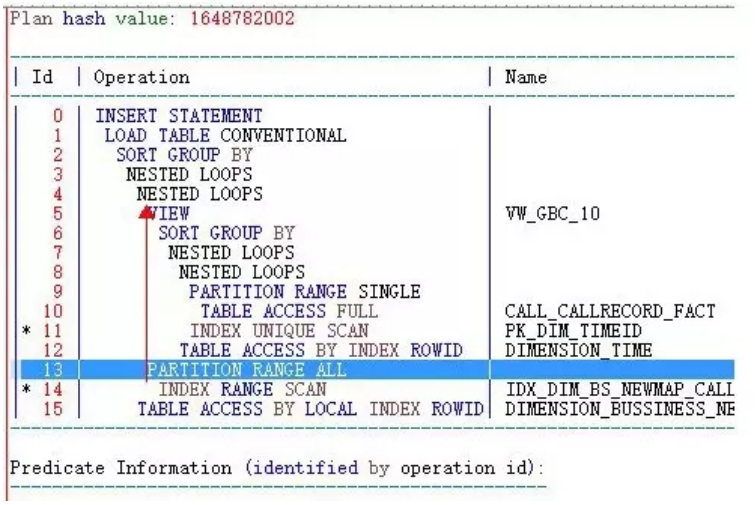
比如ID=10的继续索引，就被ID=11的挡住了，所以第10步就是入口。



```
Plan hash value: 1648782002

| Id  | Operation                          | Name
|   0 | INSERT STATEMENT                   |
|   1 |  LOAD TABLE CONVENTIONAL           |
|   2 |   SORT GROUP BY                    |
|   3 |    NESTED LOOPS                    |
|   4 |     NESTED LOOPS                   |
|   5 |      VIEW                          | VW_GBC_10
|   6 |       SORT GROUP BY                |
|   7 |        NESTED LOOPS                |
|   8 |         NESTED LOOPS               |
|   9 |          PARTITION RANGE SINGLE    |
|  10 |           TABLE ACCESS FULL        | CALL_CALLRECORD_FACT
| * 11 |          INDEX UNIQUE SCAN         | PK_DIM_TIMEID
|  12 |         TABLE ACCESS BY INDEX ROWID | DIMENSION_TIME
|  13 |       PARTITION RANGE ALL          |
| * 14 |        INDEX RANGE SCAN            | IDX_DIM_BS_NEWMAP_CALLID
|  15 |      TABLE ACCESS BY LOCAL INDEX ROWID | DIMENSION_BUSSINESS_NEWMAP
```

找到入口后，反向光标来，利用平行级别的最上最先执行，最右最先执行原则，来看父操作与子操作的关系，移动光标即可。

比如这里的第13步，我只需要定位光标在PARTITION这个P前面，然后向上移动，立马就知道，它的驱动表是ID=5的VIEW，因为他们是对齐的。



```
Plan hash value: 1648782002

| Id  | Operation                          | Name
|   0 | INSERT STATEMENT                   |
|   1 |  LOAD TABLE CONVENTIONAL           |
|   2 |   SORT GROUP BY                    |
|   3 |    NESTED LOOPS                    |
|   4 |     NESTED LOOPS                   |
|   5 |      VIEW                          | VW_GBC_10
|   6 |       SORT GROUP BY                |
|   7 |        NESTED LOOPS                |
|   8 |         NESTED LOOPS               |
|   9 |          PARTITION RANGE SINGLE    |
|  10 |           TABLE ACCESS FULL        | CALL_CALLRECORD_FACT
| * 11 |          INDEX UNIQUE SCAN         | PK_DIM_TIMEID
|  12 |         TABLE ACCESS BY INDEX ROWID | DIMENSION_TIME
|  13 |       PARTITION RANGE ALL          |
| * 14 |        INDEX RANGE SCAN            | IDX_DIM_BS_NEWMAP_CALL
|  15 |      TABLE ACCESS BY LOCAL INDEX ROWID | DIMENSION_BUSSINESS_NE

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

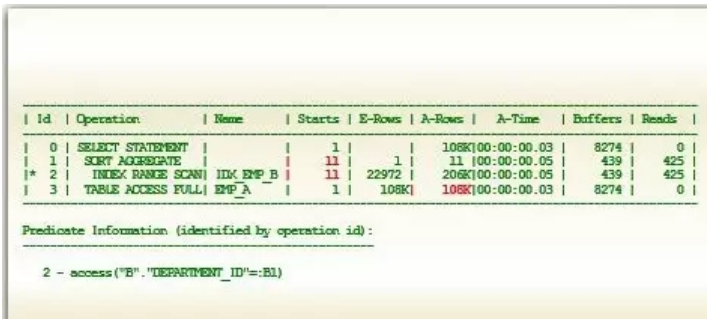
然后看看之间的JOIN关系是不是有问题，返回的行估算等。

执行计划最右最上最先执行规则，有个例外，大家知道不？？就是通过以上规则，是不正确的。

（标量子查询）

```
SELECT a.employee_id,

a.department_id,

(SELECT COUNT(*) FROM emp_b b

WHERE a.department_id=b.department_id

) cnt

FROM emp_a a;
```

比如这个ID=2的在前面，但是它事实上是被ID=3的驱动的，也就是被emp_a驱动的，这违背了一般的执行计划顺序规则，平时注意点就行了，标量子查询谓词里会出现绑定变量，比如这里的：B1，因为每次带一个值去驱动子查询。



```
| Id  | Operation          | Name      | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
|   0 | SELECT STATEMENT   |           |      1 |        |   106K|00:00:00.03 |    8274 |      0 |
|   1 |  SORT AGGREGATE    |           |     11 |      1 |    11 |00:00:00.05 |     439 |    425 |
|*  2 |   INDEX RANGE SCAN | IDX_EMP_B |     11 |  22972 |   206K|00:00:00.05 |     439 |    425 |
|   3 | TABLE ACCESS FULL  | EMP_A     |      1 |   106K|   106K|00:00:00.03 |    8274 |      0 |

Predicate Information (identified by operation id):

   2 - access("B"."DEPARTMENT_ID"=:B1)
```

搞清楚执行计划怎么干，那么看执行计划看啥？

1、看JOIN的方式

2、看表的访问方式，走全表，走索引

3、看有没有一些经常影响性能的操作，比如FILTER

4、看cardinality(rows)与真实的差距

不要太过于关注COST，COST是估算的，大不一定就慢，小不一定就快……当然比如COST很小，rows返回的都是很小的，很慢。那么，我们可能得考虑统计信息是不是过旧问题。

统计信息很重要，就说一个例子：

走了索引，COST很小，一切都很完美，但是AWR现实占80%的资源。一般啥情况？单纯从SQL上看，也就是这执行计划估计不对，自己测一下，很慢。也就是COST很小，ROWS很小，走索引，很完美的计划是错误的，那么很显然，基本就是统计信息导致的了。

实际第4步走sendtime索引，应该返回1689393行，但是执行计划估算返回1行，统计信息不准确，再次检查统计信息收集日期是5月前的。

SQL> SELECT COUNT(1) FROM MSP.T_MS_MEDIA_TASK WHERE SENDTIME >=TRUNC(SYSDATE,'dd') AND MONTHDAY = TO_CHAR(SYSDATE,'mmdd') ;

? COUNT(1)

? ?1689393

收集统计信息，for all columns size repeat 保持原有直方图信息

?exec DBMS_STATS.GATHER_TABLE_STATS(ownname=>'MSP',tabname=>'T_MS_MEDIA_TASK',estimate_percent=>10,method_opt=>'for all columns size repeat', no_invalidate=>false,cascade=>true,degree => 10);

返回168万行，但是现有统计信息却让cbo认为是1行，这差别也太大了。

method_opt=>'for all columns size repeat', 这里说下，更新统计信息，最好使用for all columns size repeat...

repeat的好处是啥，比如列有直方图，会给你保留，列没有统计信息会按照for all columns size 1收集。。。其他原来怎么收就怎么收。

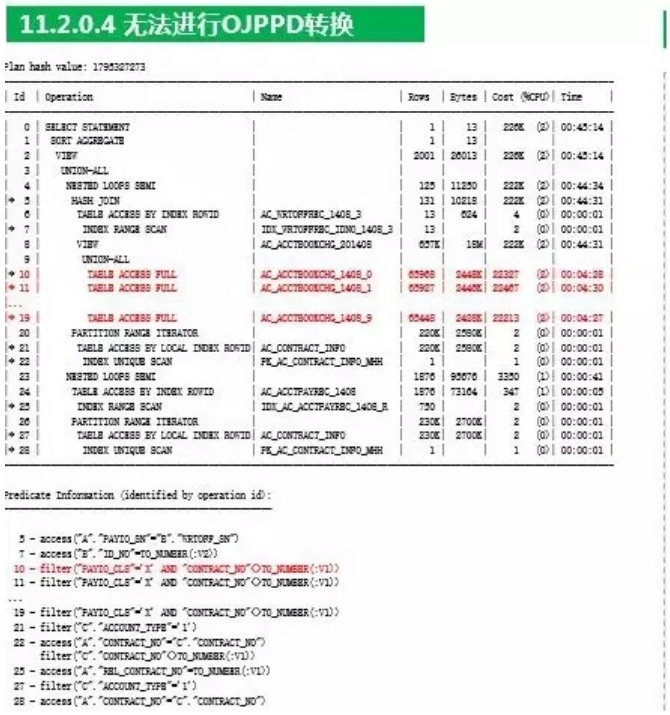
你用一个for all columns size 1或size skewonly,或者不写（auto)都可能改变原有统计信息的收集方式，都有可能影响SQL的执行效率。

高效访问结构让SQL更快，这个不说了，主要是建索引。如何建索引也是一个很复杂的问题，说一点，一般复合索引，等值查询条件频率高的，作为前导列较好。因为直接访问可能效率比>,<...等高，后者访问了还需要过滤。

下面看下影响优化器的参数导致的性能问题。

这是10g执行计划，一个视图是UNION ALL做的，全部走索引：

但是11.2.0.4全表扫描了。



10g视图有谓词推荐，也就是查询转换里的一种OJPPD=OLD JOIN PUSH PREDICATE

升级到11.2.0.4，视图里的10张表都变成FULL SCAN。

连接谓词(A.“PAYIO_SN”=“B”.“WRTOFF_SN”)未推入到视图中。

执行时间从0.01s到4s,buffer gets从212到99w。

很显然，我要检查，统计信息没有问题，然后怎么干？？看在11g里做优化器降级如何。

在11.2.0.4中使用optimizer_features_enable分别测试10.2.0.4和11.2.0.3均可谓词推入到视图中走索引。那么问题就出现在11.2.0.4了，因为11.2.0.3都是可以的。说明11.2.0.4对视图谓词推入算法有了改变。很多优化器的东西，oracle都有参数控制的，除了参数，还有各补对应的fix control。那么先检查补丁相关的

from v$system_fix_control WHERE sql_feature LIKE ‘%JPPD%’

查到了，各种开启关闭，没有用。最后看10053，分析10053，详细参看是否是BUG导致，还是优化器改进问题，参数设置问题：

```
**************************************
PARAMETERS USED BY THE OPTIMIZER
********************************
  *************************************
  PARAMETERS WITH ALTERED VALUES
  *********************************
Compilation Environment Dump
o o o
_optimizer_cost_based_transformation  = off
_optimizer_mjc_enabled                = false
_optimizer_squ_bottomup               = false
o o o
Bug Fix Control Environment
    fix  8560951 = enabled *
o o o

************************
Query transformations (QT)
************************
CBQT bypassed for query block SEL$1 (#0): Disabled by parameter.
CBQT: Validity checks failed for 2a92tkckh82vs.
CSE: Considering common sub-expression elimination in query block SEL$1 (#0)
o o o
************************
Predicate Move-Around (PM)
************************
o o o
OJPPD:     OJPPD bypassed: View semijoined to table.
JPPD:     JPPD bypassed: View not on right-side of outer-join.
FPD:   Following are pushed to where clause of query block SEL$12 (#0)
```

10053看到默认参数被关了，检查下，大概和查询转换的两个参数:

_optimizer_cost_based_transformation

_optimizer_squ_bottomup

都被关了，当然10.2.0.4和11.2.0.3被关了也是可以的。

```
**************************↓
Query transformations (QT)↓
**************************↓
CBQT bypassed for query block SEL$1 (#0): Disabled by parameter.↓
CBQT: Validity checks failed for 2a92tkckh82vs.↓
CSE: Considering common sub-expression elimination in query block SEL$1 (#0)
ooo ↓
**************************↓
Predicate Move-Around (PM)↓
**************************↓
ooo ↓
OJPPD: OJPPD bypassed: View semijoined to table.↓
JPPD: JPPD bypassed: View not on right-side of outer-join.↓
FPD: Following are pushed to where clause of query block SEL$12 (#0)↵
```

还看到基于CBO的查询转换失败，因为参数被关了，OJPPD（10g那种方式）失效了……那当然走不了，JPPD是11g的，也失效了。

基本知道执行计划如何看，关注哪些就很有用了，不要太关注啥COST前面讲了11.2.0.3都可以，到11.2.0.4不行了，那可能有2种原因：1、算法改了；2、BUG。

当然基于正常的理解，视图谓词推荐，ORACLE是必须支持的，也是不存在问题的，所以肯定有正规的解决方式。先看第2个BUG，按理说，这种常见的东西，特别是这SQL不算复杂，ORACLE应该不会触发BUG，当然，查询转换是存在各种BUG的，11.2.0,4少了很多MOS中搜一下，比如这个JPPD，就有很多BUG，但是没有看到11.2.0.4对应的。

****

Predicate Move-Around (PM)

****

。。。

OJPPD: OJPPD bypassed: View semijoined to table.

JPPD: JPPD bypassed: View not on right-side of outer-join.

通过这个判断，10.2.0.4那种OJPPD，基于规则的查询转换不行了，也就是算法改变，因为cost_base_query_transformation参数关了，应该走OJPPD的。现在JPPD也走不了，因为参数被关了，这个是基于成本的查询转换才可以。

所以，这是由于算法更新导致的问题，要求必须按照ORACLE官方建议，恢复对应查询转换参数默认值：在基于COST的查询转换部分，只能走JPPD（和OJPPD类似），ORACLE建议设置CBQT参数，基于COST查询转换更准确。

开启COST查询转换，初始化优化器参数 _optimizer_cost_based_transformation设为默认值(linear)。CBQT参数有如下值：

"exhaustive", "iterative", "linear", "on", "off"。

另外通过测试得知，还需要设置_optimizer_squ_bottomup （enables unnesting of subquery in a bottom-up manner）

参数默认值true.

这个问题，但是发了SR，老外也不知道，然后我发现这2个参数恢复默认值可以，当然首先cbqt参数我认为肯定有关系，后面的squ_bottomup是测试出来的。。。后来告诉老外，老外也认可算法改变导致的问题。所以核心参数的默认值改变，是很危险的，可能影响全局，如果这两个参数不恢复，涉及数百条核心SQL就无法正常执行，也就是系统不具有可用性了。

最后说一下，经常碰到的一个优化器缺陷：

SELECT ELEMENT_TYPEA,

ELEMENT_IDA,

ELEMENT_TYPEB,

ELEMENT_IDB,

RELATION_TYPE,

EFF_RULE_ID,

EXP_RULE_ID,

CTRL_CODE,

EFF_DATE,

```
EXP_DATE,
GROUP_ID,
BASE_TIME_ TYPE,
POWER_RIGHT,
POSITIVE_TYPE,
BOTHWAY_FLAG
FROM DBPRODADM.pd_prc_rel a
WHERE EXISTS (SELECT 1
FROM DBPRODADM.pd_prc_dict b
WHERE a.element_ida = b.prod_prcid
AND b.prod_prc_type = '1')
AND a.exp_date > SYSDATE
AND (EXISTS (SELECT 1
FROM DBPRODADM.pd_prc_dict c
WHERE a.element_idb = c.prod_prcid
AND c.prod_prc_type = '1')
OR a.element_idb = 'X')
AND a.relation_type = '10'
```

当OR与semi join放在一起的时候，会触发无法进行subquery unnest的问题，也就是可能会产生FILTER,导致SQL非常缓慢，有的甚至几天，几十天也别想运行结束了。

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers
0	SELECT STATEMENT		1		621K	00:00:12.14	2222K
* 1	FILTER		1		621K	00:00:12.14	2222K
* 2	HASH JOIN RIGHT SEMI		1	1351K	1457K	00:00:03.62	62157
* 3	TABLE ACCESS FULL	PD_PRC_DICT	1	15707	26762	00:00:00.02	878
* 4	TABLE ACCESS FULL	PD_PRC_REL	1	1742K	1935K	00:00:01.22	62279
* 5	TABLE ACCESS BY INDEX ROWID	PD_PRC_DICT	928K	1	428K	00:00:05.83	2159K
* 6	INDEX UNIQUE SCAN	PK_PD_PRODPRC_DICT	928K	1	928K	00:00:03.03	1230K

Predicate Information (identified by operation id):

```
1 - filter(("A"."ELEMENT_IDB"='X' OR  IS NOT NULL))
2 - access("A"."ELEMENT_IDA"="B"."PROD_PRCID")
3 - filter("B"."PROD_PRC_TYPE"='1')
4 - filter(("A"."RELATION_TYPE"='10' AND "A"."EXP_DATE">SYSDATE@!))
5 - filter("C"."PROD_PRC_TYPE"='1')
6 - access("C"."PROD_PRCID"=:B1)
```

伴随着FILTER有多个子节点，执行计划中会出现绑定变量，父节点的行集会驱动子节点查询。FILTER与NESTED LOOPS类似，与NESTED LOOPS区别是FILTER内部会维护一HASH表，所以父节点1351K行，子节点实际执行928K次，如果JOIN条件，比如：B1这个值重复的特别多，那么第5,6步的STARTS就会少很多，那么FILTER还是有很大优势，否则对SQL性能可能是灾难！

第5、6步执行92万多次，那肯定慢了……问题就是有个FILTER……

FILTER类似循环，在无法unnest子查询中存在，类似标量子查询那种走法，谓词里也有绑定变量的东西。

他们唯一的好处就是内部构建HASH 表，如果匹配的重复值特别多，那么探测次数少，效率好，但是大部分时候，重复值不多，那么就是灾难了

对于这种优化器限制的，一般就是得改写了，因为SQL结构决定无法走高效的执行计划。。。因为我这里虽然走了所以，但是执行次数太多，如果执行次数少，到也无所谓。

改写后的sql：

```
SELECT ELEMENT_TYPEA,
ELEMENT_IDA,
ELEMENT_TYPEB,
ELEMENT_IDB,
RELATION_TYPE,
EFF_RULE_ID,
EXP_RULE_ID,
CTRL_CODE,
EFF_DATE,
EXP_DATE,
GROUP_ID,
BASE_TIME_ TYPE,
POWER_RIGHT,
POSITIVE_TYPE,
BOTHWAY_FLAG
FROM DBPRODADM.pd_prc_rel a
WHERE EXISTS (SELECT 1
FROM DBPRODADM.pd_prc_dict b
WHERE a.element_ida = b.prod_prcid
AND b.prod_prc_type = '1')
AND a.exp_date > SYSDATE
AND (EXISTS (SELECT 1
FROM DBPRODADM.pd_prc_dict c
WHERE a.element_idb = c.prod_prcid
AND c.prod_prc_type = '1')
OR a.element_idb = 'X')
AND a.relation_type = '10'
```

很显然，这里的条件是exists or ...那么改写得用UNION或UNION ALL了，为了避免有重复行，用UNION

```
select ELEMENT_TYPEA,ELEMENT_IDA,ELEMENT_TYPEB,ELEMENT_IDB,RELATION_TYPE
,EFF_RULE_ID,EXP_RULE_ID,CTRL_CODE,EFF_DATE,EXP_DATE,GROUP_ID,BASE_TIME_TYPE,
POWER_RIGHT,POSITIVE_TYPE,BOTHWAY_FLAG
from DBPRODADM.pd_prc_rel a
where exists
(select 1
from DBPRODADM.pd_prc_dict b
where a.element_ida = b.prod_prcid
and b.prod_prc_type = '1')
```

```
and a.exp_date > sysdate

and exists (select 1

from DBPRODADM.pd_prc_dict c

where a.element_idb = c.prod_prcid

and c.prod_prc_type = '1')

and a.relation_type = '10'

union

select ELEMENT_TYPEA,ELEMENT_IDA,ELEMENT_TYPEB,ELEMENT_IDB,RELATION_TYPE

,EFF_RULE_ID,EXP_RULE_ID,CTRL_CODE,EFF_DATE,EXP_DATE,GROUP_ID,BASE_TIME_TYPE,

POWER_RIGHT,POSITIVE_TYPE,BOTHWAY_FLAG

from DBPRODADM.pd_prc_rel a

where exists

(select 1

from DBPRODADM.pd_prc_dict b

where a.element_ida = b.prod_prcid

and b.prod_prc_type = '1')

and a.exp_date > sysdate

and a.element_idb = 'X'

and a.relation_type = '10';
```

两个分支都走HASH JOIN，starts全部为1，虽然全部是全表扫描，但是执行效率提升很明显，执行时间从12s到7s,gets从222w到4.5w之后，是否还有优化空间？

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers
0	SELECT STATEMENT		1		621K	00:00:07.54	45108
1	SORT UNIQUE		1	447K	621K	00:00:07.54	45108
2	UNION-ALL		1		621K	00:00:05.24	45108
* 3	HASH JOIN		1	436K	597K	00:00:04.18	22993
* 4	TABLE ACCESS FULL	PD_PRC_DICT	1	15707	26762	00:00:00.02	878
* 5	HASH JOIN		1	872K	952K	00:00:03.11	22115
* 6	TABLE ACCESS FULL	PD_PRC_DICT	1	15707	26762	00:00:00.02	878
* 7	TABLE ACCESS FULL	PD_PRC_REL	1	1742K	1838K	00:00:01.13	21237
* 8	HASH JOIN		1	10302	24602	00:00:00.30	22115
* 9	TABLE ACCESS FULL	PD_PRC_DICT	1	15707	26762	00:00:00.02	878
* 10	TABLE ACCESS FULL	PD_PRC_REL	1	20675	24803	00:00:00.22	21237

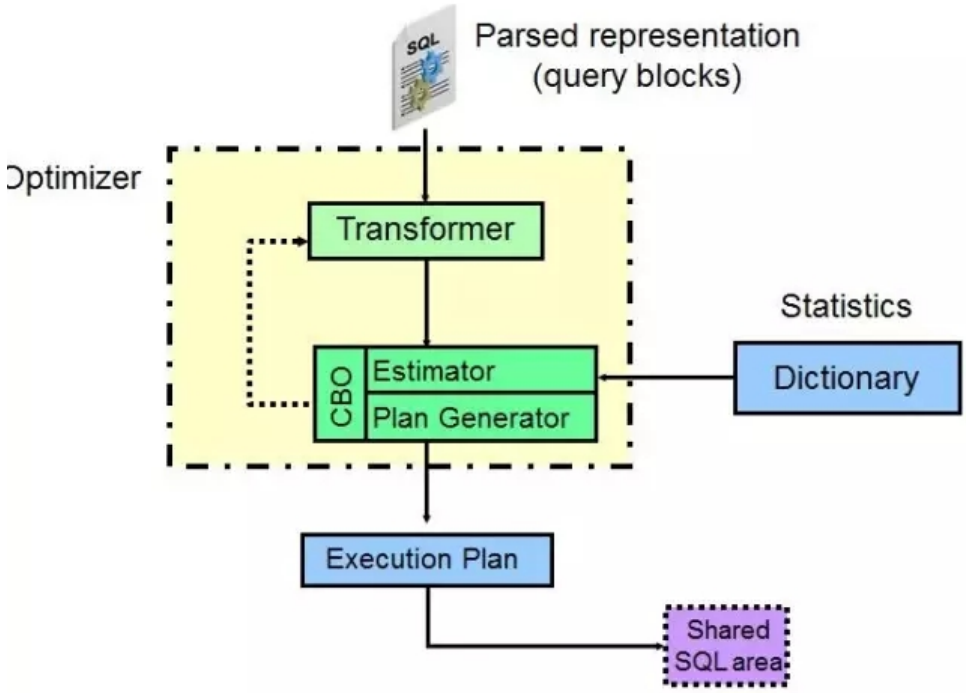
Predicate Information (identified by operation id):

```
   3 - access("A"."ELEMENT_IDA"="B"."PROD_PRCID")
   4 - filter("B"."PROD_PRC_TYPE"='1')
   5 - access("A"."ELEMENT_IDB"="C"."PROD_PRCID")
   6 - filter("C"."PROD_PRC_TYPE"='1')
   7 - filter(("A"."RELATION_TYPE"='10' AND "A"."EXP_DATE">SYSDATE@!))
   8 - access("A"."ELEMENT_IDA"="B"."PROD_PRCID")
   9 - filter("B"."PROD_PRC_TYPE"='1')
  10 - filter(("A"."ELEMENT_IDB"='X' AND "A"."RELATION_TYPE"='10' AND "A"."EXP_DATE">SYSDATE@!))
```

特别逻辑读少了很多。后续优化：

1)改写使用了UNION,是否能改成UNION ALL避免排序？

2)这么多全表扫描，是否能够让一些可以走索引？当然，这些是可以做到的，但是不是主要工作了。这个案例告诉我们，优化器是有很多限制的，不是万能的。



除了统计信息正确，良好的SQL结构，能够让SQL正确进行查询转换，正确的访问结构，如索引等……都是让SQL高效执行的前提条件。复杂！=低效，简单！=高效。让优化器理解，并且有合适的访问结构支持，才是王道！

简单的SQL不是快的保证，复杂的也不一定见得慢，高效的执行计划才是最重要的，索引优化SQL，最重要的就是让不好的执行计划变得好。

也就是从多个方面入手，最终达到我们的优化目标。