

# SQL性能监控和优化



----孙其成

# 目录



**1** 性能监控

**2** 执行计划

**3** 选择率

**4** 优化方法论

**5** 优化案例

# 性能监控（一）

## □ 数据库运行优化过程中DBA的困惑

性能突变

系统一直都在运行，SQL的性能有没有发生突发性的变化

多变系统

最新版本已经在测试环境中进行测试，如何从中那迅速找出新增那部分的SQL

X因素

数据库里面SQL众多，如何快速分析和定位最影响性能的X因素

### DBA的困扰



问题定位

影响性能的SQL里面，究竟出现了什么问题导致性能变差

# 性能监控（二）

主要针对新版本中引入SQL的性能状态分析，及时发现新版本中潜在的性能隐患。



上线SQL  
审查

重点分析数据库使用的核心SQL性能及其性能变化趋势，作出精确的分析及预警。



核心SQL  
管理

# 性能监控（三）

## □ 性能监控

SQL性能性能监控与指标上的性能监控同样重要，往往更能反映数据库系统由于突变致的性能问题。

一般当数据库系统告警CPU使用率高时，往往看到大量相同SQL的等待事件如 latch: buffer cache chains、read by other session、buffer busy、db file scattered read等，处理方式是对引起这些事件对应的SQL语句进行优化，问题可能出在统计信息、索引、程序并发量等方面。

优化完成后，分析是什么导致突然出现性能问题，需要跟踪SQL的来源，是系统运行很久的语句，还是程序变动产生的语句，如果是前者需要找出突变原因，后者的话，在变动前有没有对语句进行全面的评审。

上线评审、核心语句管理两项任务可以很好的诠释。

做好上线前语句评审，能够将性能隐患扼杀在摇篮中；对核心语句进行基线管理，可以很好监控性能变化。

# 性能监控（四）

## □ 上线SQL评审

对于需要新上线SQL语句，如果想要做到评审，需要开发商能够配合，将新上线语句梳理并发给评审人员，进行性能评审。

由于大量Ad Hoc的存在，无法做到全面，需要在测试环境中部署SQL采集脚本，对比SQL基线查找TOP SQL。

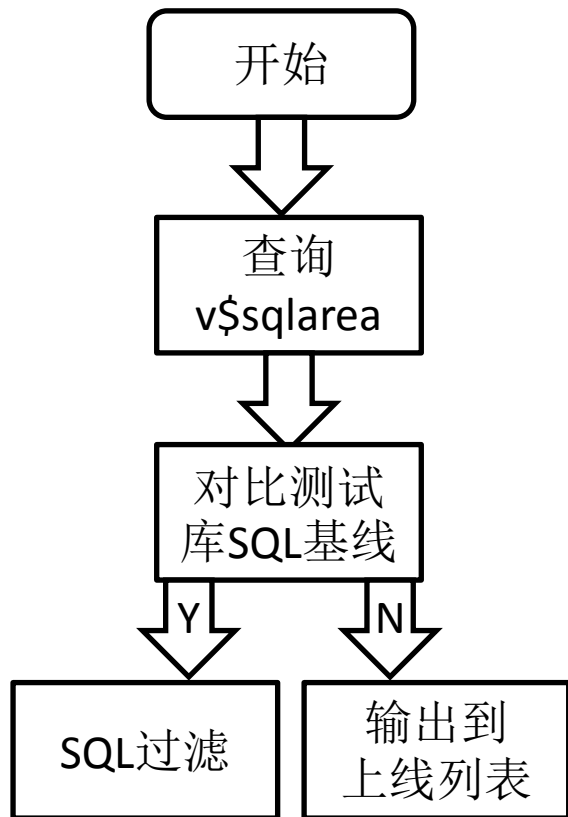
## □ 核心SQL管理

在现网生产数据库系统中，由于某些原因导致执行计划变化，且执行效率上变差的情况，发生情况还是比较多见，大多数变差原因是由于统计信息问题或索引问题导致，然而能够及早的发现问题并解决，显的尤为重要。

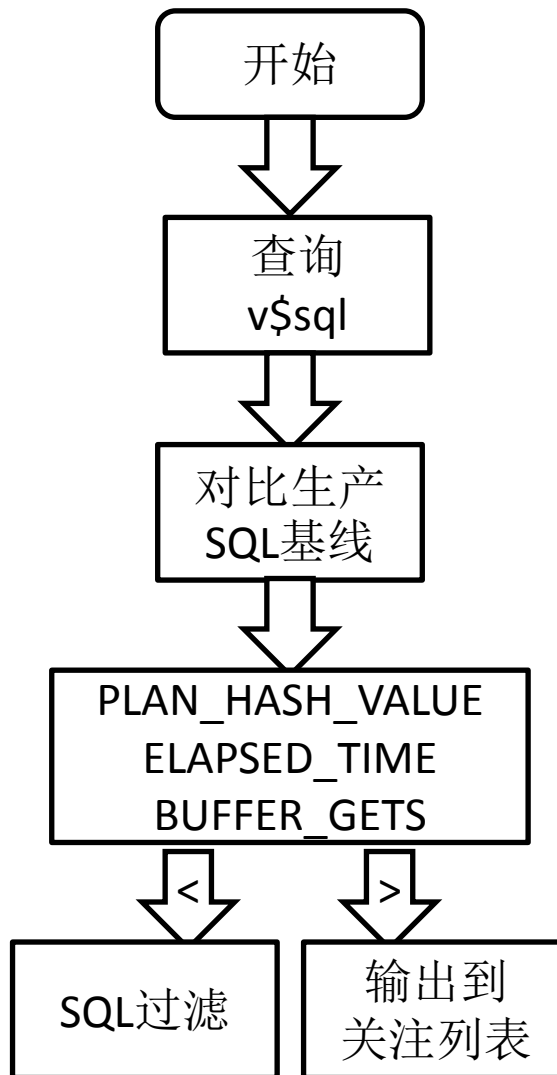
为监控变化情况，首先需要SQL基线，编写采集SQL基线的定时刷新程序，将SQL语句相关的执行时间、逻辑读等信息一并采集，通过临时表存储v\$sql信息，与基线表进行关联，PLAN\_HASH\_VALUE不同的SQL\_ID，且执行时间和逻辑读较高的我们认为是发生突变的语句，就应该引起关注。

# 性能监控（五）

## □ 上线SQL评审



## □ 核心SQL管理



# 目录



**1** 性能监控

**2** 执行计划

**3** 选择率

**4** 优化方法论

**5** 优化案例



# 执行计划（一）

## □ 查看执行计划

### **EXPLAIN PLAN 方式**

SQL不是真实执行，忽略绑定变量，对执行计划的估算。

### **SQL\_TRACE 和10046**

真实执行结果，需要使用TKPROF工具方便查看。

### **AUTOTRACE**

内部使用EXPLAIN PLAN，即不是真实执行。

### **DBMS\_XPLAN.DISPLAY\_CURSOR 和 V\$SQL\_PLAN**

查询共享池中的真实执行结果。

### **DBMS\_XPLAN.DISPLAY\_AWR**

查询AWR基表WRH\$存储的真实执行结果。

## **工具类方法**

# 执行计划（二）

## □ 查看执行计划

推荐使用查看真实执行计划的方式，系统包DBMS\_XPLAN；

### 查看指定SQLID对应的执行计划

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR('sqlid',null,'ADVANCED  
ALLSTATS LAST PEEKED_BINDS'));
```

### 查看当前SESSION最后执行语句的执行计划

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(null,null,'ADVANCED  
ALLSTATS LAST PEEKED_BINDS'));
```

### 查看指定SQLID历史执行计划

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_AWR('sqlid',format=>'ALL');
```

# 执行计划 (三)

## □ 解读执行计划

执行计划顺序按照“最右最上，父子迭代”原则。

| Id   | Operation                         | Name                           |
|------|-----------------------------------|--------------------------------|
| 0    | SELECT STATEMENT                  |                                |
| 1    | SORT AGGREGATE                    |                                |
| 2    | VIEW                              |                                |
| 3    | UNION-ALL                         |                                |
| * 4  | FILTER                            |                                |
| 5    | NESTED LOOPS SEMI                 |                                |
| * 6  | HASH JOIN                         |                                |
| 7    | PARTITION RANGE SINGLE            |                                |
| * 8  | TABLE ACCESS BY LOCAL INDEX ROWID | CM_CU_GRP                      |
| * 9  | INDEX RANGE SCAN                  | IDX_CM_GRP_CRT_DATE            |
| 10   | PARTITION RANGE SINGLE            |                                |
| * 11 | TABLE ACCESS FULL                 | GROUP_CUSTOMER                 |
| 12   | PARTITION RANGE SINGLE            |                                |
| * 13 | TABLE ACCESS BY LOCAL INDEX ROWID | CUSTOMER                       |
| * 14 | INDEX UNIQUE SCAN                 | PK_CM_CU_CUSTOMER              |
| * 15 | FILTER                            |                                |
| 16   | NESTED LOOPS SEMI                 |                                |
| 17   | PARTITION RANGE SINGLE            |                                |
| * 18 | TABLE ACCESS BY LOCAL INDEX ROWID | MM_SALE_APPROVE                |
| * 19 | INDEX RANGE SCAN                  | IDX_MM_SALE_APPROVE_CREATEDATE |
| 20   | PARTITION RANGE SINGLE            |                                |
| * 21 | TABLE ACCESS BY LOCAL INDEX ROWID | CUSTOMER                       |
| * 22 | INDEX UNIQUE SCAN                 | PK_CM_CU_CUSTOMER              |

0

1

2

3

4

1  
5

5

1  
6

6

1  
2

1  
7

2  
0

7

1  
0

1  
3

1  
8

2  
1

8

1  
1

1  
4

1  
9

2  
2

9

# 目录

1 性能监控

2 执行计划

3 选择率

4 优化方法论

5 优化案例



# 选择率（一）

## □ 结果集的评估行数

下面执行计划中列rows的值是如何得到，针对各种谓词选择条件方式（<、>、= etc），CBO是如何计算每一步的结果集的基数，从而根据基数大小来判断下一步选择哪种表连接方式，针对单表。

| Id  | Operation                   | Name                     | Rows |
|-----|-----------------------------|--------------------------|------|
| 0   | SELECT STATEMENT            |                          | 1    |
| 1   | SORT UNIQUE                 |                          | 1    |
| 2   | NESTED LOOPS ANTI           |                          | 1    |
| 3   | NESTED LOOPS                |                          | 1    |
| * 4 | TABLE ACCESS BY INDEX ROWID | T_OM_WORKLIST            | 1    |
| * 5 | INDEX RANGE SCAN            | IDX_T_OM_WT_ACNUM_NTTYPE | 1    |
| * 6 | TABLE ACCESS BY INDEX ROWID | SA_DB_DICTITEM           | 1    |
| * 7 | INDEX UNIQUE SCAN           | PK_SA_DB_DICTITEM        | 1    |
| * 8 | TABLE ACCESS BY INDEX ROWID | SA_DB_DICTITEM           | 2    |
| * 9 | INDEX UNIQUE SCAN           | PK_SA_DB_DICTITEM        | 1    |

# 选择率（二）

首先说一下什么是选择率(selectivity)，以及选择率的作用及重要性。

介绍一个词cardinality(基数)，我的理解是结果集的行数，在看我们的执行计划时，任何一步操作都会返回一个结果集，而这个结果集的行数，称作基数。基数在执行计划中非常重要，CBO会根据基数来判断是选择索引还是进行全表扫描，是进行HASH JOIN还是进行NEST LOOP，是以及表的执行先后顺序，以至于影响整个执行计划。

基数怎么重要，它是怎么计算来的？是的，通过selectivity选择率得到，只要准确计算出选择率，那么基数只要通过selectivity \* (number of rows) 得到。

# 选择率（三）

## □ 表访问选择率系列之等值条件

构造一个行数为1000表sun1, 且object\_name平均NDV为10, 收集统计信息;

```
update sun1 set object_name='sun1' WHERE object_id > 702 AND object_id <=802;
```

```
SQL> SELECT count(distinct object_name) FROM sun1 ;
```

```
COUNT(DISTINCTOBJECT_NAME)
```

```
-----
```

```
3
```

```
SELECT * FROM sun1 WHERE object_name='sun1';
```

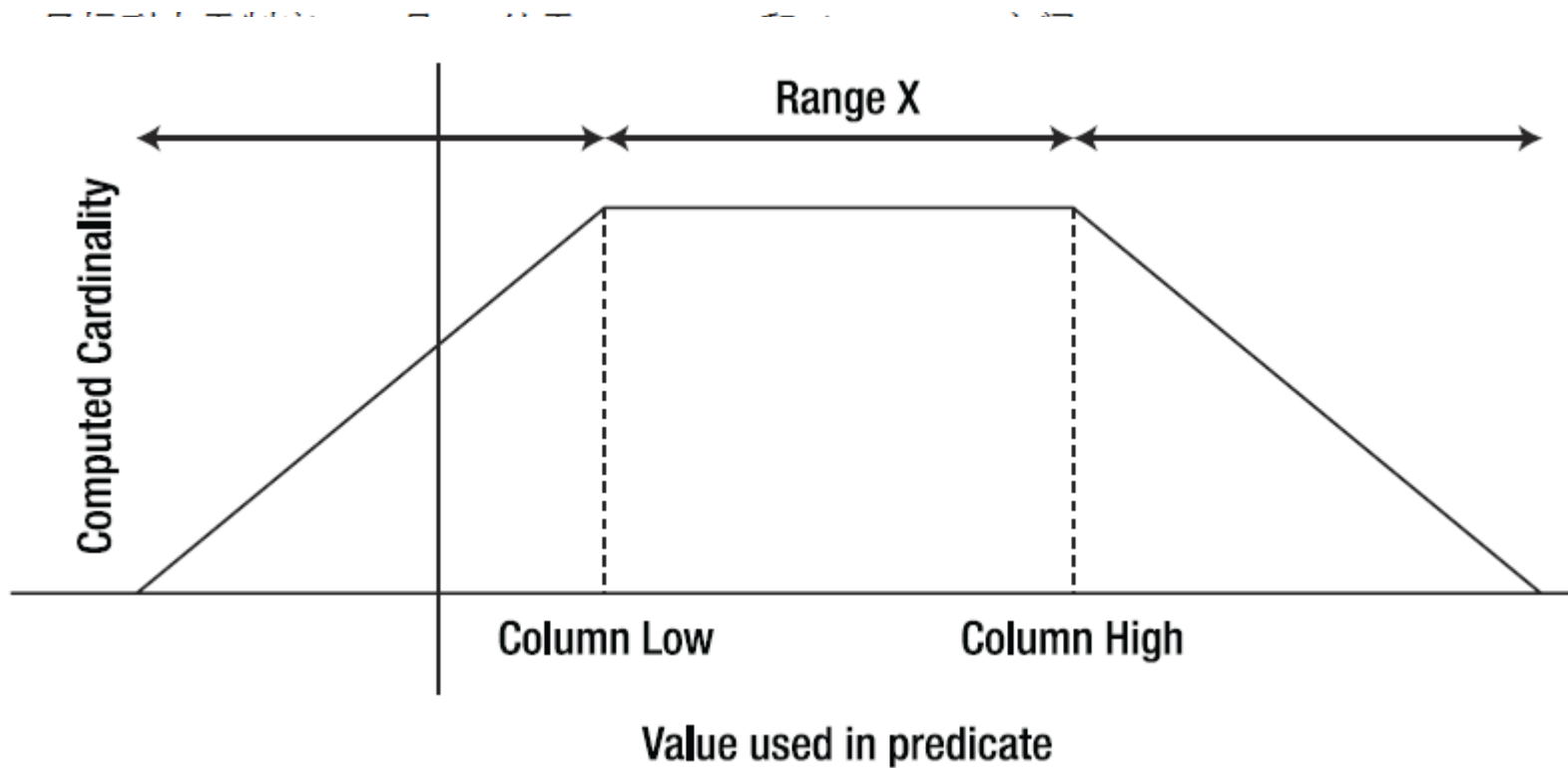
```
-----  
| Id | Operation | Name | Starts | E-Rows |  
-----  
| 0 | SELECT STATEMENT | | 1 | |  
|* 1 | TABLE ACCESS FULL| SUN1 | 1 | 333 |  
-----
```

如果在object\_name上有个单列索引的话, 就会选择索引因为基数小于一定比例, 比例是多少呢? 答案是30%

```
selnull=(row_nums-nul_nums)/row_nums  
selectivity=1/distict_value*Selnull=1/10*1=0.1  
cardinality=selectivity*row_nums=0.1 * 1000=100
```

# 选择率（四）

## □ 表访问选择率系列之范围条件



DATA\_OBJECT\_ID between 25 and 30

30

Bounded, closed, closed



# 选择率（五）

## □ 表访问选择率系列之列表list条件

| 谓语句条件  | 基数 (card) |                          |
|--|-----------|--------------------------|
| <code>data_object_id in (25)</code>          | 1         | ouch                     |
| <code>data_object_id in (4,4)</code>         | 100       | good                     |
| <code>data_object_id in (3,25)</code>        | 100       | ouch,but consistent      |
| <code>data_object_id in (3,15)</code>        | 173       | ouch,but consistent      |
| <code>data_object_id in (3,25,26)</code>     | 101       | ouch,but consistent      |
| <code>data_object_id in (3,25,25,26)</code>  | 101       | ouch,but consistent      |
| <code>data_object_id in (3,25,null)</code>   | 100       | ouch,ouch                |
| <code>data_object_id in (:b1,:b2,:b3)</code> | 300/300   | ouch,ouch,but consistent |

```
data_object_id in (3,25,26) --à100
```

判断值是否有效，等于等值条件

```
selectivity=1/distict_value=1/12=1/12
```

```
cardinality=selectivity*row_nums=1/12* 1200=100
```

```
data_object_id in (3,25,25,26) --à100
```

判断值是否有效，等于等值条件

```
selectivity=1/distict_value=1/12=1/12
```

```
cardinality=selectivity*row_nums=1/12* 1200=100
```

```
data_object_id in (3,25,null) --à100
```

判断值是否有效，等于等值条件

```
selectivity=1/distict_value=1/12=1/12
```

```
cardinality=selectivity*row_nums=1/12* 1200=100
```

```
data_object_id in (:b1,:b2,:b3) --à300/300
```

对于不适用窥探情况，不考虑重复值

```
selectivity=1/distict_value + 1/distict_value +1/distict_value =3/12
```

```
cardinality=selectivity*row_nums=3/1200* 1200 =300
```

# 选择率（六）

## □ 表访问选择率系列之多个谓语句条件

```
SELECTIVITY(predicate1) AND SELECTIVITY(predicate2)
    = SELECTIVITY(predicate1) * SELECTIVITY(predicate2)
SELECTIVITY(predicate1) OR SELECTIVITY(predicate2)
    = SELECTIVITY(predicate1) + SELECTIVITY(predicate2)
    - SELECTIVITY(predicate1) * SELECTIVITY(predicate2)
SELECTIVITY(NOT predicate1)
    = 1- SELECTIVITY(predicate1)
```

```
SQL> SELECT count(*) FROM sun.rangetab WHERE DATA_OBJECT_ID != 12 ;
```

| Id  | Operation         | Name     | Rows | Bytes | Cost (%CPU) | Time     |
|-----|-------------------|----------|------|-------|-------------|----------|
| 0   | SELECT STATEMENT  |          | 1    | 3     | 7 (0)       | 00:00:01 |
| 1   | SORT AGGREGATE    |          | 1    | 3     |             |          |
| * 2 | TABLE ACCESS FULL | RANGETAB | 1100 | 3300  | 7 (0)       | 00:00:01 |

```
CARDINALITY= SELECTIVITY * ROW_NUMES
              = (1-1/12)*1200
              = 1100
```

# 选择率（七）

## □ 表访问选择率系列之表连接

```
select t1.v1, t2.v1
from t1, t2
```

```
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=57 Card=2000 Bytes=68000)
1      0      HASH JOIN (Cost=57 Card=2000 Bytes=68000)
2      1      TABLE ACCESS (FULL) OF 'T2' (Cost=28 Card=200 Bytes=3400)
3      2      TABLE ACCESS (FULL) OF 'T1' (Cost=28 Card=400 Bytes=6800)
```

```
Join Selectivity =
((num_rows(t1) - num_nulls(t1.join1)) / num_rows(t1)) *
((num_rows(t2) - num_nulls(t2.join2)) / num_rows(t2)) /
greater(num_distinct(t1.join1), num_distinct(t2.join2))

Join Cardinality =
Join Selectivity *
filtered cardinality(t1) * filtered cardinality(t2)
```

```
Join Selectivity =
(10,000 - 0) / 10,000) *
(10,000 - 0) / 10,000) /
greater(30, 40) = 1/40
Join Cardinality = 1/40 * (400 * 200) = 2000
```

# 目录

1 性能监控

2 执行计划

3 选择率

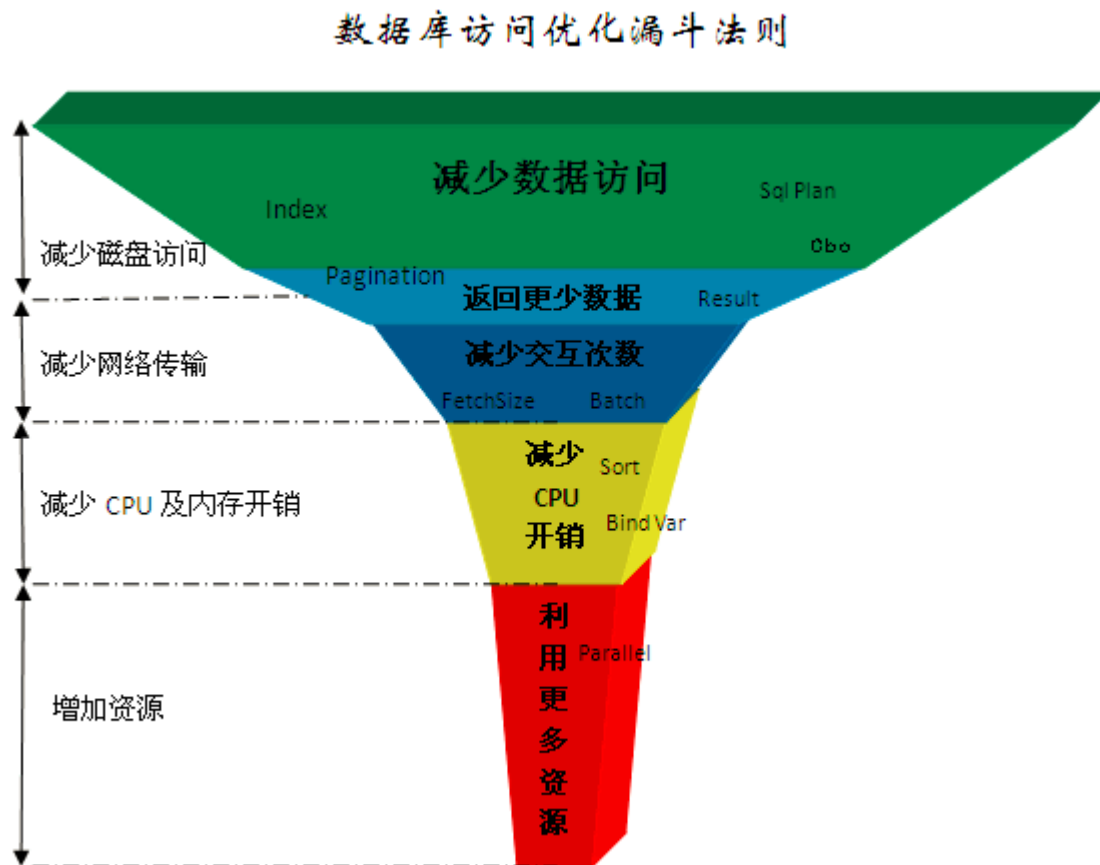
4 优化方法论

5 优化案例



# 优化方法论（一）

所谓“方法论”，即“怎么做”，对于优化来讲，要有一个总体优化路线，在路线的引领下提升性能。



# 优化方法论（二）

入口论，即执行计划中的第一步，我们在解读执行计划时，需要按照递进关系找出执行的顺利，往往会先找到入口。

当你遇到一个高消耗的SQL语句，不管是单表还是多表，如何看到SQL语句就能猜出优化的执行计划？

不管SQL语句多复杂，总有一个入口的步骤，然而返回的结果集基数决定着之后执行的表的访问方式和连接方式，需要掌握NL、HJ、MJ这三种连接方式的特性以及索引方式。

```
VAR CD VARCHAR2(32)
EXEC :CD := '0'

SELECT A.STAFFID,
       B.WORKNO,
       A.STAFFNAME,
       E.DUTYNAME,
       D.ORGACODE AS CITYCODE,
       D.ORGANAME AS CITYNAME,
       B.CCID,
       B.VDNID
FROM ICDPUB.T_UCP_STAFFBASICINFO A,
     ICDPUB.T_UCP_STAFFWORKNO     B,
     ICDPUB.T_UCP_ORGAINFO         C,
     ICDPUB.T_UCP_CITYINFO         D,
     ICDPUB.T_UCP_DUTYINFO         E
WHERE A.STAFFID = B.STAFFID
      AND A.ORGAINFID = C.ORGAINFID
      AND C.ORGACODE LIKE D.ORGACODE || '%'
      AND A.DUTYID = E.DUTYID(+)
      AND B.CCID = :CD;
```

# 目录

1 性能监控

2 执行计划

3 选择率

4 优化方法论

5 优化案例



# 优化案例（一）

先来看一下下面两个SQL，这里只讨论SQL本身。

```
SELECT *
  from subs_product a
 where a.prodid = 'C03'
       and a.region = '533'
       and a.subsid in (select t.subsid
                        from subscriber t
                        where region = 533
                           and subtype = 'Person'
                           and active = '1'
                           and t.status = 'US10'
                           and prodid = 'pp.eo.nw'
                           and exists (select 1
                                       from subs_product b
                                       where t.subsid = b.subsid
                                          and b.region = 533)
                                       and rownum < 1001);
```

```
SELECT *
  from subs_product a
 where a.prodid = 'C03'
       and a.region = '533'
       and a.subsid in (select t.subsid
                        from subscriber t
                        where region = 533
                           and subtype = 'Person'
                           and active = '1'
                           and t.status = 'US10'
                           and prodid = 'pp.eo.nw'
                           and exists (select 1
                                       from subs_product b
                                       where t.subsid = b.subsid
                                          and b.region = 533
                                          and b.prodid = 'C03')
                                       and rownum < 1001);
```

两个SQL，区别  
在与多一个条件  
b.prodid = 'C03'。

执行效率相同  
嘛？



# 优化案例（一）

```

and exists (select 1
            from tbc.s.subs_product b
            where t.subsid = b.subsid
              and b.region = 533
              and b.prodId = 'C03')
and rownum < 1001)

```

| call    | count | cpu   | elapsed | disk   | query  | current | rows |
|---------|-------|-------|---------|--------|--------|---------|------|
| Parse   | 1     | 0.02  | 0.03    | 0      | 0      | 0       | 0    |
| Execute | 1     | 0.00  | 0.00    | 0      | 0      | 0       | 0    |
| Fetch   | 1     | 19.95 | 181.52  | 784503 | 879153 | 9       | 0    |
| total   | 3     | 19.97 | 181.56  | 784503 | 879153 | 9       | 0    |

Misses in library cache during parse: 1  
 Optimizer mode: ALL\_ROWS  
 Parsing user id: SYS  
 Number of plan statistics captured: 1

| Rows (1st) | Rows (avg) | Rows (max) | Row Source Operation  |
|------------|------------|------------|---|
| 0          | 0          | 0          | NESTED LOOPS (cr=879153 pr=784503 pw=0 time=181529197 us)   |
| 0          | 0          | 0          | NESTED LOOPS (cr=879153 pr=784503 pw=0 time=181529191 us cost=216487 size=10584 card=72)  |
| 0          | 0          | 0          | VIEW VW_NS0_1 (cr=879153 pr=784503 pw=0 time=181529186 us cost=215982 size=936 card=72)   |
| 0          | 0          | 0          | COUNT STOPKEY (cr=879153 pr=784503 pw=0 time=181529182 us)  |
| 0          | 0          | 0          | NESTED LOOPS SEMI (cr=879153 pr=784503 pw=0 time=181529156 us cost=215982 size=5184 card=72)  |
| 11295      | 11295      | 11295      | PARTITION RANGE SINGLE PARTITION: 4 4 (cr=785493 pr=784503 pw=0 time=179527187 us cost=215478 size=3312 card=72)                            |
| 11295      | 11295      | 11295      | TABLE ACCESS FULL SUBSCRIBER PARTITION: 4 4 (cr=785493 pr=784503 pw=0 time=179518245 us cost=215478 size=3312 card=72)                      |
| 0          | 0          | 0          | PARTITION RANGE SINGLE PARTITION: 4 4 (cr=93660 pr=0 pw=0 time=1973872 us cost=7 size=3215654 card=123679)                                  |
| 0          | 0          | 0          | PARTITION HASH ITERATOR PARTITION: KEY KEY (cr=93660 pr=0 pw=0 time=1943050 us cost=7 size=3215654 card=123679)                             |
| 0          | 0          | 0          | TABLE ACCESS BY LOCAL INDEX ROWID SUBS_PRODUCT PARTITION: KEY KEY (cr=93660 pr=0 pw=0 time=1906434 us cost=7 size=3215654 card=123679)      |
| 431212     | 431212     | 431212     | INDEX RANGE SCAN IDX_CM_SUBS_PRODUCT_SUBSID PARTITION: KEY KEY (cr=35942 pr=0 pw=0 time=636320 us cost=6 size=0 card=4)(object id 20412717) |
| 0          | 0          | 0          | PARTITION RANGE SINGLE PARTITION: 4 4 (cr=0 pr=0 pw=0 time=0 us cost=6 size=0 card=4)   |
| 0          | 0          | 0          | PARTITION HASH ITERATOR PARTITION: KEY KEY (cr=0 pr=0 pw=0 time=0 us cost=6 size=0 card=4)  |
| 0          | 0          | 0          | INDEX RANGE SCAN IDX_CM_SUBS_PRODUCT_SUBSID PARTITION: KEY KEY (cr=0 pr=0 pw=0 time=0 us cost=6 size=0 card=4)(object id 20412717)          |
| 0          | 0          | 0          | TABLE ACCESS BY LOCAL INDEX ROWID SUBS_PRODUCT PARTITION: 1 1 (cr=0 pr=0 pw=0 time=0 us cost=7 size=134 card=1)                             |

Elapsed times include waiting on following events:

| Event waited on             | Times Waited | Max. Wait | Total Waited |
|-----------------------------|--------------|-----------|--------------|
| SQL*Net message to client   | 1            | 0.00      | 0.00         |
| Disk file operations I/O    | 31           | 0.00      | 0.00         |
| db file scattered read      | 15864        | 0.14      | 166.60       |
| db file sequential read     | 59           | 0.00      | 0.12         |
| SQL*Net message from client | 1            | 0.00      | 0.00         |

\*\*\*\*\*

# 优化案例（一）

通过查看上面两个执行计划，访问路径不同在这一步，  
filter(“B” . “PRODID” = ‘C03’ AND “B” . “REGION” =533)。同样是全表扫描，为什么产生这么大的差距？

发现慢的SQL，产生了大量的物理读，问题出在哪里？

问题出在rownum < 1001，如果去掉这个条件，再次执行两个SQL，执行差别不大。

第一个SQL扫描了较少的块，就找到1000行满足条件的数据，于是终止扫描。

第二个SQL 增加条件 and b.prodId = ‘C03’ 后需要扫描更多的块才能获得满足条件的1000行，实际上扫描了所有的块也没有找到1000行满足条件的数据，扫描更多的块，花费更多的时间。

这也是我们在平时优化时，明明执行计划相同，为什么效率就变低了的原因之一。所以优化需要综合考虑各种因素。

优化思路： 改写SQL，创建组合索引，尽量减少扫描的块数。

# 优化案例（二）

```
var B2      VARCHAR2(32)
var B1      NUMBER
exec :B2    := 'ca2x5001'
exec :B1    := 533
SELECT COUNT(1)
  FROM TBCS.CM_CU_GROUPECUSTSERVCHANNEL T
 WHERE T.CUSTOMGR = :B2
       AND T.REGION = :B1
       AND EXISTS(SELECT 1 FROM TBCS.MM_WK_COMMONWORKLOG T1 WHERE
```

发现如上的不正确的执行计划，简单说明一下执行计划，首先通过T1表的DATE字段选择过滤出结果集，再将表T通过CUSTOMGR字段选择出结果集，再将两个查询的结果集通过NEST LOOP方式返回数据。正常情况下如果连接条件不是通过函数进行转换且CUST\_ID上有索引的话，会通过索引CUST\_ID进行NEST LOOP 返回结果。

那么我们怎么知道选择这种执行计划时不正确的呢？ 因为比之前正确的执行计划消耗更高。

| Id   | Operation                         | Name                        |
|------|-----------------------------------|-----------------------------|
| 0    | SELECT STATEMENT                  |                             |
| 1    | SORT AGGREGATE                    |                             |
| * 2  | FILTER                            |                             |
| 3    | NESTED LOOPS                      |                             |
| 4    | NESTED LOOPS                      |                             |
| 5    | SORT UNIQUE                       |                             |
| 6    | PARTITION RANGE ITERATOR          |                             |
| * 7  | TABLE ACCESS BY LOCAL INDEX ROWID | MM_WK_COMMONWORKLOG         |
| * 8  | INDEX RANGE SCAN                  | IDX_COMWORKLOG_SERVICEDATE  |
| 9    | PARTITION RANGE SINGLE            |                             |
| * 10 | INDEX RANGE SCAN                  | IDX_GROUPECUSTSERV_CUSTOMGR |
| * 11 | TABLE ACCESS BY LOCAL INDEX ROWID | CM_CU_GROUPECUSTSERVCHANNEL |

# 优化案例（二）

## □ 查看历史执行情况

| SNAP_TIME           | SQL_ID        | PLAN_HASH_VALUE | EXES | BFGETS     | ETIMES | ROWDS |
|---------------------|---------------|-----------------|------|------------|--------|-------|
| 2015-07-29 03:30:42 | 43kqxxv4nrv43 | 554499663       | 2561 | 17690.98   | .34    | 2578  |
| 2015-07-30 03:30:00 | 43kqxxv4nrv43 | 554499663       | 2554 | 17936.6    | .34    | 2567  |
| 2015-07-31 03:30:24 | 43kqxxv4nrv43 | 554499663       | 2545 | 18396.21   | .39    | 2551  |
| 2015-08-01 03:30:10 | 43kqxxv4nrv43 | 554499663       | 2510 | 18678.46   | .31    | 2523  |
| 2015-08-02 03:30:05 | 43kqxxv4nrv43 | 554499663       | 2517 | 18237.55   | .38    | 2523  |
| 2015-08-03 03:30:18 | 43kqxxv4nrv43 | 554499663       | 2512 | 17518.95   | .37    | 2524  |
| 2015-08-04 03:30:19 | 43kqxxv4nrv43 | 554499663       | 2507 | 17709.81   | .3     | 2514  |
| 2015-08-05 03:30:15 | 43kqxxv4nrv43 | 554499663       | 2512 | 18184.56   | .41    | 2515  |
| 2015-08-06 03:30:09 | 43kqxxv4nrv43 | 554499663       | 2508 | 17931.24   | .37    | 2515  |
| 2015-08-06 03:30:10 | 43kqxxv4nrv43 | 554499663       | 2508 | 17931.24   | .37    | 2515  |
| 2015-08-07 03:30:13 | 43kqxxv4nrv43 | 554499663       | 2505 | 18133.27   | .37    | 2514  |
| 2015-08-07 03:30:14 | 43kqxxv4nrv43 | 554499663       | 2505 | 18133.27   | .37    | 2514  |
| 2015-08-08 03:30:16 | 43kqxxv4nrv43 | 554499663       | 2505 | 18182.2    | .38    | 2510  |
| 2015-08-09 03:30:03 | 43kqxxv4nrv43 | 554499663       | 2504 | 17781.07   | .39    | 2512  |
| 2015-08-10 03:30:35 | 43kqxxv4nrv43 | 3217193188      | 230  | 2026330.68 | 24.36  | 227   |
| 2015-08-10 04:00:43 | 43kqxxv4nrv43 | 3217193188      | 173  | 3710687.76 | 41.15  | 172   |
| 2015-08-10 04:30:51 | 43kqxxv4nrv43 | 3217193188      | 211  | 3119577.36 | 33.39  | 211   |
| 2015-08-10 05:00:06 | 43kqxxv4nrv43 | 3217193188      | 112  | 6447776.46 | 61.86  | 112   |
| 2015-08-10 05:00:07 | 43kqxxv4nrv43 | 3217193188      | 112  | 6447776.46 | 61.86  | 112   |
| 2015-08-10 05:30:13 | 43kqxxv4nrv43 | 3217193188      | 196  | 3501607.68 | 36.33  | 196   |
| 2015-08-10 05:30:14 | 43kqxxv4nrv43 | 3217193188      | 196  | 3501607.68 | 36.33  | 196   |

# 优化案例（二）

通过上面的查询结果，我们可以看得出来在2015-8-10 3:30 的时候，出现了错误的选择。那么问题就来了，为什么会选择错误的执行计划，为什么在8月10号发生？我们可以推断次SQL是在每天3:30的时候进行定时的执行，且10号前执行效率很高，索引2500次的执行再很短时间内就完成了。但在10号后，由于错误执行计划的出现，导致SQL执行时间变长，2500次的执行不能很快完成，将战线拖到了半天，占用白天的资源。

那么为什么会发生这样的情况呢？我们通过10053来看一下：  
不正确的执行计划的10053 trace。

可以看到提示：**Using prorated density: 0.000001 of col #9 as selectvity of out-of-range/non-existent value pred**

这个提示说明出现了越界现象。当日期判断的范围超出了统计信息记录的最大值和最小值的范围，CBO无法计算选择率，得出错误的结果集，此时通过估计的值来进行选择，当评估的值与实际情况差别很大时，就会出现选择偏差。

问题为什么会发生在8月10日？

# 优化案例（二）

查看表上日期字段的统计值

```
SQL> SELECT low_value,high_value FROM dba_tab_col_statistics WHERE table_name='MM_WK_COMMONWORKLOG'  
AND column_name ='SERVICEDATE' ;
```

| LOW_VALUE      | HIGH_VALUE     |
|----------------|----------------|
| 78670108011313 | 7873070A120F03 |

```
SQL> Set serverout on
```

```
declare  
temp date;  
SQL> 2 3 BEGIN  
4 dbms_stats.convert_raw_value('78670108011313',temp);  
5 dbms_output.put_line(temp);  
6 END;  
7 /
```

```
2003-01-08 00:18:18
```

```
PL/SQL procedure successfully completed.
```

```
SQL> Set serverout on
```

```
SQL> declare  
2 temp date;  
3 BEGIN  
4 dbms_stats.convert_raw_value('7873070A120F03',temp);  
5 dbms_output.put_line(temp);  
6 END;  
7 /
```

```
2015-07-10 17:14:02
```

```
PL/SQL procedure successfully completed.
```

我们看到记录的最大日志是2015-07-10，在与SQL语句SERVICEDATE >= (SYSDATE - 30)，执行计划出错的日期进行比较，可以发现，8月10号-30天正好超过了记录的最大日期值。那这就是变化发生在8月10日的原因。

改变条件SYSDATE-40，执行效率就提升了。

# 优化案例（三）

| SQL Statement               | Id   | Operation                   | Name                    | Rows  | B |
|-----------------------------|------|-----------------------------|-------------------------|-------|---|
| VAR CD VARCHAR2(32)         | 0    | SELECT STATEMENT            |                         | 13760 |   |
| EXEC :CD := '0'             | *    | HASH JOIN RIGHT OUTER       |                         | 13760 |   |
| SELECT A.STAFFID,           | 2    | TABLE ACCESS FULL           | T_UCP_DUTYINFO          | 8     |   |
| B.WORKNO,                   | *    | HASH JOIN                   |                         | 13760 |   |
| A.STAFFNAME,                | *    | HASH JOIN                   |                         | 137K  |   |
| E.DUTYNAME,                 | 5    | MERGE JOIN CARTESIAN        |                         | 137K  |   |
| D.ORGACODE AS CII           | 6    | TABLE ACCESS FULL           | T_UCP_CITYINFO          | 20    |   |
| D.ORGANAME AS CII           | 7    | BUFFER SORT                 |                         | 6881  |   |
| B.CCID,                     | *    | TABLE ACCESS FULL           | T_UCP_STAFFWORKNO       | 6881  |   |
| B.VDNID                     | 9    | TABLE ACCESS FULL           | T_UCP_STAFFBASICINFO    | 432K  |   |
| FROM ICDPUB.T_UCP_STAF      | 10   | TABLE ACCESS FULL           | T_UCP_ORGAINFO          | 960K  |   |
| ICDPUB.T_UCP_STAFFWORKNO    | B,   |                             |                         |       |   |
| ICDPUB.T_UCP_ORGAINFO       | Id   | Operation                   | Name                    | Rows  |   |
| ICDPUB.T_UCP_CITYINFO       | 0    | SELECT STATEMENT            |                         | 1     |   |
| ICDPUB.T_UCP_DUTYINFO       | 1    | NESTED LOOPS                |                         |       |   |
| WHERE A.STAFFID = B.STAFFID | 2    | NESTED LOOPS                |                         | 1     |   |
| AND A.ORGAINFO = C.ORGAINFO | 3    | NESTED LOOPS OUTER          |                         | 1     |   |
| AND C.ORGACODE LIKE I       | 4    | NESTED LOOPS                |                         | 1     |   |
| AND A.DUTYID = E.DUTYID     | 5    | MERGE JOIN CARTESIAN        |                         | 1     |   |
| AND B.CCID = :CD;           | 6    | TABLE ACCESS FULL           | T_UCP_CITYINFO          | 1     |   |
|                             | 7    | BUFFER SORT                 |                         | 6985  |   |
|                             | *    | TABLE ACCESS FULL           | T_UCP_STAFFWORKNO       | 6985  |   |
|                             | 9    | TABLE ACCESS BY INDEX ROWID | T_UCP_STAFFBASICINFO    | 1     |   |
|                             | * 10 | INDEX UNIQUE SCAN           | PK_T_UCP_STAFFBASICINFO | 1     |   |
|                             | 11   | TABLE ACCESS BY INDEX ROWID | T_UCP_DUTYINFO          | 1     |   |
|                             | * 12 | INDEX UNIQUE SCAN           | PK_T_UCP_DUTYINFO       | 1     |   |
|                             | * 13 | INDEX UNIQUE SCAN           | PK_T_UCP_ORGAINFO       | 1     |   |
|                             | * 14 | TABLE ACCESS BY INDEX ROWID | T_UCP_ORGAINFO          | 1     |   |

| 值      | 逻辑读     | 执行时间  |
|--------|---------|-------|
| 0 (NL) | 475     | 0.01s |
| 5 (NL) | 1568279 | 3.19s |
| 0 (HJ) | 10701   | 0.1s  |
| 5 (HJ) | 47202   | 1.09s |

# 优化案例（三）

首先，谓词CCID是否能够走索引，因为列上存在索引，而且绑定值为0时的可选择率也是相当好，

CCID COUNT(\*)

| ----- |       |
|-------|-------|
| 0     | 1     |
| 2     | 3893  |
| 1     | 8469  |
| 4     | 16650 |
| 5     | 18024 |

```
SELECT * FROM (
SELECT snap_id,
dbms_sqltune.extract_bind(bind_data,1).value_string bind1
FROM dba_hist_sqlstat
WHERE sql_id='2njyghn86vc9' ORDER BY snap_id desc ) WHERE rownum < 50;
      SNAP_ID BIND1
```

| ----- |   |
|-------|---|
| 16859 | 0 |
| 16858 | 0 |
| 16858 | 5 |
| 16851 | 0 |
| 16850 | 0 |
| 16850 | 0 |
| 16843 | 0 |
| 16842 | 0 |
| 16842 | 0 |
| 16835 | 0 |
| 16834 | 0 |
| 16834 | 0 |
| 16827 | 0 |
| 16826 | 0 |
| 16826 | 0 |
| 16819 | 0 |
| 16818 | 0 |
| 16818 | 0 |
| 16811 | 0 |
| 16810 | 5 |



# 优化案例（三）

摆在面前的三条路，

- 1.继续按照当前的方式执行，0和5普遍消耗 稍高，但不会出现暴增
- 2.使用HINT路径的方式，当0时执行迅速，但5时消耗会百倍增加
- 3.从应用角度将0和5的值进行不同处理，选择各自最优方式执行

第一条路，当时是不做任何改变。

第二条路，需要伪造T\_UCP\_CITYINFO的统计信息都为0，等同于当表里没有记录的时候，收集统计信息，生产环境当然不能把表清空收集统计信息，因为我们只想统计信息改变。

```
CREATE TABLE T_UCP_CITYINFO2 AS SELECT * FROM T_UCP_CITYINFO WHERE 1=2;
```

此时导入没报错，但是表T\_UCP\_CITYINFO却没有统计信息，

需要修改统计信息表的信息

```
Update T_UCP_CITYINFO_STAT2 set c1= 'T_UCP_CITYINFO' ; commit;
```

第三条路，需要应用开发对0或5采用不同的处理方式，0时采用索引方式，5时采用HASH方式或者使用明文值，采集CCID列上直方图，CBO自己选择合适执行计划。

# 号外-上线评审

The screenshot displays the DATAYUN SQL Performance Management Platform interface. The top navigation bar includes the DATAYUN logo and the text "SQL 性能管理平台". The main header area shows the current database name "数据库名: hqq\_32\_11g", version "版本号: v1.0", type "类型: ORACLE 11G", SSID "SSID: ngboss", and IP "IP: 192.168.12.32".

The left sidebar contains a navigation menu with the following items: 上线前审核, 总览, 版本审核报表, SQL跟踪报表, 生产TopSQL, 采集管理, 即时审核, and 系统管理. The "生产TopSQL" item is highlighted with a red bracket.

The main content area is titled "上线前审核" and displays a table with the following columns: 所属用户, SQL\_ID, 执行次数(次), 平均CPU时间(毫秒)/平均耗时(毫秒)/次, 平均逻辑读/次, 平均物理读/次, 审核状态, and 操作. Below the table, there are tabs for "审核详情", "执行计划", "历史性能趋势", and "自动优化建议". The "自动优化建议" tab is active, showing a table of optimization suggestions.

| 类型 | 名称           | 描述   | 建议  |
|----|--------------|--|---|
| 表  | "HR"."SMALL" | The optimizer requires up-to-date statistics for the table in order to select a good execution plan. | -Consider collecting optimizer statistics for this table. execute dbms_stats.gather_table_stats(ownname => 'HR', tablename => |
| 索引 | "HR"."BIG"   | The optimizer requires up-to-date statistics for the table in order to select a good execution plan. | -Consider collecting optimizer statistics for this table. execute dbms_stats.gather_table_stats(ownname => 'HR', tablename => |
| 表  | "HR"."SMALL" | The optimizer requires up-to-date statistics for the table in order to select a good execution plan. | -Consider collecting optimizer statistics for this table. execute dbms_stats.gather_table_stats(ownname => 'HR', tablename => |

Below the table, there is a "深度建议" section with the following content:

FINDINGS SECTION (1 finding)

1- Index Finding (see explain plans section below)

The execution plan of this statement can be improved by creating one or more indices.

Recommendation (estimated benefit: 99.98%)

- Consider running the Access Advisor to improve the physical schema design or creating the recommended index.

```
create index HQ.IDX$$_34C80001 on HQ.HQ_TEST("A");
```

Rationale

Creating the recommended indices significantly improves the execution plan of this statement. However, it might be preferable to run "Access Advisor" using a representative SQL workload as opposed to a single statement. This will allow to get comprehensive index recommendations which takes into

# 号外-核心SQL管理

上线前审核

总览

版本审核报表

SQL跟踪报表

生产TopSQL

采集管理

即时审核

系统管理



PlanHashValue选择:  最新审核时间: 2016-06-28 16:15:01



[导出执行计划](#)

[全部展开](#) | [全部收起](#) | [显示更多信息](#)

| 操作                                  | 违反规则名称 | 对象              | 对象类型           |
|-------------------------------------|--------|-----------------|----------------|
| ▼ NESTED LOOPS OUTER                |        |                 |                |
| ▼ MERGE JOIN CARTESIAN              | 笛卡尔积   |                 |                |
| ▼ TABLE ACCESS BY INDEX ROWID       |        | PROFILE\$       | TABLE          |
| INDEX RANGE SCAN                    |        | I_PROFILE       | INDEX          |
| ▼ BUFFER SORT                       |        |                 |                |
| ▼ PARTITION RANGE ITERATOR          |        |                 |                |
| ▼ TABLE ACCESS BY LOCAL INDEX ROWID |        | WRH\$_ACTIVE... | TABLE          |
| INDEX RANGE SCAN                    |        | WRH\$_ACTIVE... | INDEX (UNIQUE) |
| INDEX UNIQUE SCAN                   |        | WRM\$_SNAPSH... | INDEX (UNIQUE) |
| INDEX UNIQUE SCAN                   |        | WRH\$_EVENT_... | INDEX (UNIQUE) |

# 号外-在线分析

  **SQL**  
性能管理平台

- 上线前审核
  - 总览
  - 版本审核报表
  - SQL跟踪报表
- 生产TopSQL
  - 总览
  - 审核报表
- 采集管理
- 即时审核**
- 系统管理

### 即时审核

数据库类型:  数据库:

SQL语句 审核完成,请查看审核项结果!

```
1  
2 DELETE  
3 FROM  
4   HQ.HQ_TEST  
5 WHERE  
6   A=1
```

● 全表扫描

执行计划

## Question & Answer



**If you have more questions later, feel free to ask**