特集3

現場の運用管理にすぐに役立つ

エキスパートが明かす Oracle 性能改善Tips

韓国エクセム
趙 東郁 CHO, DongWook
日本エクセム株式会社
金 圭福 KIM, Gyubok

バージョンアップごとにOracleの機能も一段と進化していく中で、逆にエンジニアがそのスピードに追い付けなくなってきている。本特集では、現場の技術者に向けて、数多くあるOracleの機能の中からOracleの性能管理の観点であまり知られてはいないが効果的な使い方と、注意して認識しておくべきポイントを紹介する。また、本特集を基にOracle性能管理の検証法や考え方についても参考にしていただきたい。

筆者が常々思っていることだが、Oracleには「あまり知られていないが、ぜひ活用してほしい」という優れた機能がたくさんある。OTNをはじめ、世界中のブロガー達はネット上でOracleの機能活用においてより良い方法の発掘作業を活発に行なっているが、今回はあえてオフラインでその一部を紹介したいと考えている。

本稿では、韓国の性能管理分野でその活動が高く評価されているブログ(Oracle ACE)から、管理者に役立つTipsをピックアップしてまとめた(追加のTipsは「http://www.ex-em.co.jp/exemlabo.html」を参照)。現場では今までのやり方である程度通常の運用を賄えていると思うが、本特集を参考により効率的な方法を積極的に取り入れていただければ幸いだ。

Tips1 IN句に変数を無制限に指定する

SQL文法でIN句に1000以上の値を指定するとエラーになる(**LIST1**)。8iまでは最大256個、9iからは最大1000個まで指定できる。「1000個を超える値を指定することがあるの?」と聞きたいところだと思うが、何が起きるか分からないのが世の中の常だ。現実世界では想定外のことが起こり得る。そのときはどうすべきか?

まず思いつくのは、ダミーカラムを使ってマルチカラムの条件を指定することで、1000個の制約をなくす方法だ(**LIST2**解消Tips①)。簡単で良い反面、とても長いSQLで何か違和感が漂う。グローバル一時表を使って、あらかじめ変数をため込む方法もある(LIST2解消Tips②)。

また、パイプライン表ファンクションを活用すると、より洗練された形で解消することもできる(LIST2解消Tips③)。このように、1つの課題に対してさまざまな解決策を用意しておけば、状況に合った最適なソリューションを見つけることができるだろう。

LIST1 : IN句付きSQL文のエラー

```
drop table t1 purge;
create table t1(c1 int, c2 int) ;

insert into t1 select level, level from dual connect by level <= 10000 ;

-- 指定するとエラーになる1000を超える値を設定するSQLを作る
var v_sql clob;
begin
    :v_sql := 'select count(*) from t1 where c1 in (';
    for idx in 1 .. 1100 loop
    :v_sql := :v_sql || idx || ', ';
    end loop;
    :v_sql := :v_sql || ' 1101);';
end;
/

set long 100000
print v_sql

--「print v_sql」の結果:IN句に1000を超える値を指定するとエラーになる
select count(*) from t1 where c1 in (1, 2, 3, 4, 5, ..., 1101);
ORA-01795: リストに指定できる式の最大数は1000です
```

Tips2 バインドミスマッチの怖さ

LIST3のようなJavaコードがある。長さが1、50、150、2500の文字列に対して同じSQLにバインド変数化して実行する。この場合、共有プールにはいくつのSQL文が載っているのか? エンジニアの常識では当然1つになるべきだが、驚くことに4つプールされている。

```
SQL> select version_count
from v$sqlarea where sql_text like
 'INSERT INTO t(name) VALUES(:1)%'
==> 4
```

どうしてこのような結果になるのだろうか。これは、バインド変数のタイプや長さによって共有されないバインドミスマッチ現象のためだ。特にVARCHAR2タイプのバインド変数でよく発生する。内部で32、128、2000、4000の範囲でその長さを切り上げて使っているからだ。そのため、LIST3のSQLはすべて異なるSQLとして認識され

LIST2：IN句付きSQL文のエラーの解消

```
-- 解消Tips①：ダミーカラムでマルチ条件を指定
select count(*) from t1 where (1, c1) in ((1, 1), (1, 2), ..., (1, 1101));

-- 解消Tips②：グローバル一時表を使用
create global temporary table gtt1(c1 int);

insert into gtt1
select level from dual connect by level <= 1101 ;

select count(*) from t1 where c1 in (select c1 from gtt1) ;

-- 解消Tips③：パイプライン表関数を利用
create or replace type type1 as table of int;
/

create or replace function func1
return type1
pipelined
is
begin
    for idx in 1 .. 1101 loop
        pipe row(idx);
    end loop;
    return;
end;
/

select count(*) from t1 where c1 in (select * from table(func1)) ;
```

LIST3：バインドミスマッチのJavaコード

```
PreparedStatement stmt = con.prepareStatement(""INSERT INTO t(name) VALUES(?)"");
stmt.setString(1, ""a""); // Length = 1
stmt.executeUpdate();
stmt.setString(1, ""aaaaaaaa...a""); // Length = 50
stmt.executeUpdate();
stmt.setString(1, ""aaaaaa.................a""); // Length = 150
stmt.executeUpdate();
stmt.setString(1, ""aaaaaa............................aaaaa""); // Length = 2500
stmt.executeUpdate();
```

LIST4：長さの違いによるバインドミスマッチの解消

```
-- 変数に考えられる最大長さまでスペースを加える
PreparedStatement stmt = con.prepareStatement(""INSERT INTO t(name) VALUES
(RTRIM(?))""); // RTRIMを追加
stmt.setString(1, ""a     ...    ""); // 変数の最大長(4000bytes)までスペースを付ける
stmt.executeUpdate();
```

LIST5：表関数の使い方

```
-- オブジェクトを作成
create or replace type obj_type1 as object (
          c1 int,
          c2 int
);
/

-- コレクションを宣言
create or replace type obj_tbl_type1 as table of obj_type1;
/

-- パイプライン・関数を作成
create or replace function func1(p1 int, p2 int, p3 int)
return obj_tbl_type1
pipelined
is
          v_obj obj_type1;
begin
          for idx in 1 .. p3 loop
                    v_obj := obj_type1(p1+idx, p2+idx);
                    pipe row(v_obj);
          end loop;
end;
/

-- 使い方①：表関数でデータを単純抽出
select * from table(func1(1, 1, 10)) ;
        C1          C2
---------- ----------
         2           2
         3           3
         ⋮
        11          11

--「1…100」のデータを持つ表を作成
drop table t1 purge;
create table t1(c1) as select level from dual connect
by level <= 100 ;

-- 使い方②：通常の表と結合して表関数のデータを抽出
select * from t1, table(func1(t1.c1, t1.c1, 10)) ;
        C1          C1          C2
---------- ---------- ----------
         1           2           2
         1           3           3
         ⋮
       100         110         110
```

てしまう。

もしかすると、「4個ではあまり問題にはならないのでは?」と思う読者もいるかもしれないが、次のようなSQLを見てからでも問題にならないと思うだろうか。

```
INSERT INTO t(a、b、c,……,z) VALUES(?,?、?、...、?)
```

これは最悪のケースだが、「4*4*……* 4＝4^{24}＝281,474,976,710,656」個のSQLを共有プールに載せることを考えるとぞっとするだろう。バインド変数を使っているにもかかわらず「V $SQLAREA.VERSION_COUNT >= 2」あるいは「V $SQL_SHARED_CURSOR.BIND_MISMATCH = 'Y'」が多い場合はこのような現象を疑ってみよう。もし同じ現象と判断されたら、考えられる最長（通常4000バイト）のバインド変数のSQLを使うと、その後はどのようなバインド値が使われても再活用される（**LIST4**）。

Tips3 表関数の活用

次のような課題がある。

> 共有プールにキャッシュされているSQLの中で、論理読み取りが高い順でランタイムの実行計画を出力したい

この課題をクリアするためにプログラムを組んだり、さまざまな方法があると思うが、ここでは表関数を活用してSQLテキストベースで課題のデータを抽出してみる。

まず、簡単な使い方を確認しよう。**LIST5**のように、オブジェクトとコレクションを使ったパイプライン関数の結果を抽出すれば、関数の結果値が表のデータのように参照できるようになる（LIST5使い方①）。

COLUMN

「RTFM」と「BAAG」ーOracleユーザーのマナー

IT業界では、広く知られた業界用語がいくつもある。ここでは、その中で最近流行っている2つの用語を紹介しよう。

●RTFM
（Read The Fucking Manual）
「マニュアルをきちんと読みなさい」

オンラインフォーラムでの質問の80%以上がマニュアルに記載されている内容であることからできた言葉だ。オラクルが提供するマニュアルを見れば素直にうなずくしかないほど1つ1つが優れた教育教材で、我々が知るべき知識の80%以上を提供している。

●BAAG
（Battle Against Any Guesswork）
「どんな推測も排除しなさい」

これは、一瞬見てその意味が推測できない用語だろう。この言葉は、エキスパートの1人が推測による誤った性能診断が出回っている現実を改善するために提案した用語で、公式サイト（http://www.battleagainstanyguess.com）もある。エンジニアとして覚えておくべき姿勢であろう。

LIST6：上位SQLの実行計画を出力

```
select plan_table_output
from   (select *
        from   (select s.sql_id, s.child_number
                from v$sql s
                where exists(select 1 from v$sql_plan p where p.plan_hash_value = s.plan_hash_value)
                order by s.buffer_gets desc)
         where rownum <= 10
       ) s,
       table(dbms_xplan.display_cursor(s.sql_id, s.child_number, 'allstats last'))
;

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------------------
SQL_ID  4umgnah7kcw0a, child number 0
-------------------------------------
SELECT m.promo_name, p.prod_name, s.sum_quantity_sold, s.sum_amount_sold, s.top_rank FROM (
… 中略 …
p.prod_id AND s.promo_id = m.promo_id ORDER BY s.promo_id, s.prod_id, s.top_rank DESC

-------------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name            | E-Rows |  OMem |  1Mem | Used-Mem |
-------------------------------------------------------------------------------------------------------
|   1 |  SORT ORDER BY                           |                 |    204 |  9216 |  9216 | 8192  (0)|
|*  2 |   HASH JOIN                              |                 |    204 |   752K|   752K|  316K (0)|
|   3 |    NESTED LOOPS                          |                 |    204 |       |       |          |
… 中略 …
|  13 |    TABLE ACCESS FULL                     | PROMOTIONS      |    503 |       |       |          |
-------------------------------------------------------------------------------------------------------

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

   2 - access(""S"".""PROMO_ID""=""M"".""PROMO_ID"")
… 中略 …
  10 - access(""TIME_ID"">=TO_DATE('2000-07-01 00:00:00', 'yyyy-mm-dd hh24:mi:ss') AND
              ""TIME_ID""<=TO_DATE('2000-12-31 00:00:00', 'yyyy-mm-dd hh24:mi:ss'))
  12 - access(""S"".""PROD_ID""=""P"".""PROD_ID"")
```

LIST7：良いクラスタ化係数vs悪いクラスタ化係数

```
drop table t_cf purge;
create table t_cf(c1 int, c2 int);
create index t_cf_i1 on t_cf(c1);
create index t_cf_i2 on t_cf(c2);

--「c1」カラム：表ブロックのデータ順と同じデータが入る、「c2」カラム ： ランダム順のデータが入る
insert into t_cf
select rownum, lvl
from   (
         select level lvl
         from dual connect by level <= 10000
         order by dbms_random.random
       ) ;
commit;

exec dbms_stats.gather_table_stats(user, 't_cf', method_opt=>'for all columns size 1', cascade=>true);

select index_name, blevel, leaf_blocks, distinct_keys, clustering_factor from dba_ind_statistics ⇒
where table_name = 'T_CF' ;
INDEX_NAME                       BLEVEL LEAF_BLOCKS DISTINCT_KEYS CLUSTERING_FACTOR
------------------------------ ---------- ----------- ------------ -----------------
T_CF_I1                               1          19        10000                18 -> 良いクラスタ化係数
T_CF_I2                               1          32        10000              9425 -> 悪いクラスタ化係数

-- c1、c2カラムのデータ分布は同じ
select count(*),
       sum(case when c1 between 1 and 100 then 1 else 0 end) c1_cnt,
       sum(case when c2 between 1 and 100 then 1 else 0 end) c2_cnt
from   t_cf  ;
  COUNT(*)     C1_CNT     C2_CNT
---------- ---------- ----------
     10000        100        100

explain plan for select /*+ good cf index(t_cf) */ *  from t_cf where c1 between 1 and 100;
select * from table(dbms_xplan.display);
-- 良いクラスタ化係数の場合、表ブロックへのアクセスのコストは低い
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |   100 |   700 |      3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID | T_CF    |   100 |   700 |      3   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | T_CF_I1 |   100 |       |      2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

explain plan for select /*+ bad cf index(t_cf) */ * from t_cf where c2 between 1 and 100;
select * from table(dbms_xplan.display);
-- 良いクラスタ化係数の場合、表ブロックへのアクセスのコストは高い
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |   100 |   700 |     97   (0)| 00:00:02 |
|   1 |  TABLE ACCESS BY INDEX ROWID | T_CF    |   100 |   700 |     97   (0)| 00:00:02 |
|*  2 |   INDEX RANGE SCAN           | T_CF_I2 |   100 |       |      2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

※誌面の都合により⇒で折り返し。以下同

また、表のデータであるため通常の表と結合してデータを参照することもできる（LIST5使い方②）。この特性を活用すると、v $sqlと結合して上位SQLの実行計画を出力することで、いとも簡単に課題をクリアできるのだ（**LIST6**）。

Tips4 クラスタ化係数の良し悪し

クラスタ化係数は、索引スキャンを行なうか表スキャンを行なうかの判断において最も重要なファクターの1つだ。クラスタ化係数が良いときと悪いときの性能の違いをサンプルで紹介する。**LIST7**の例を見てみよう。実行計画のコストの違いの原因は何だろうか。

スクリプトを見ると分かるが、データの分布とSELECT文の検索範囲は同じだ。索引の統計情報を確認すると、コストの違いはクラスタ化係数による現象と分かる。データの絞り込みが良い索引でこのように索引から表ブロックへのアクセスコストが急増する場合は、クラスタ化係数の影響を確認して該当索引の順番に合わせて表データを入れ替える案も検討してみよう。もし統計情報の収集ができない場合は、**LIST8**のように手動で確認することもできる。

Tips5 索引スキャンが予想外の動きをする

たまにOracleがエンジニアの常識とは反する動きをすることがある。単純に言うとそれほど知られていない内部動作のメカニズムか不具合による現象だが、ほと

LIST8：クラスタ化係数の手動算出

```
-------------------------------------------------------------------
-- @usage:
--       @cf index_name  sample_percent
--       @cf t1_n1 10
-------------------------------------------------------------------

define __IND_NAME = &1
define __SAMPLE = &2

set serveroutput on

declare
  v_cursor      sys_refcursor;
  v_cols        varchar2(4000);
  v_tbl         varchar2(4000);
  v_sample      varchar2(4000);
  v_tmp         varchar2(4000);
  v_fno         number;
  v_bno         number;
  v_prev_fno    number;
  v_prev_bno    number;
  v_cf          number := 0;
  v_acf         number := 0;
begin

  open v_cursor for
      'select column_name ' ||
      'from user_ind_columns ' ||
      'where index_name = upper(''&__IND_NAME'') ' ||
      'order by column_position';
  loop
    fetch v_cursor into v_tmp;
    exit when v_cursor%notfound;
    v_cols := v_cols||', ' || v_tmp;
  end loop;

  close v_cursor;

  v_cols := substr(v_cols, 2);
  dbms_output.put_line('Columns = ' || v_cols);

  select table_name into v_tbl
  from user_indexes
  where index_name = upper('&__IND_NAME')
  ;

  dbms_output.put_line('Table = ' || v_tbl);

  select decode(&__SAMPLE,100,' ',' sample(&__SAMPLE) ') into v_sample
  from dual
  ;

  open v_cursor for
      'select /*+ full(' || v_tbl || ') */ ' ||
      '       dbms_rowid.rowid_block_number(rowid) ' ||
      '      ,dbms_rowid.rowid_relative_fno(rowid) ' ||
      'from ' || v_tbl || v_sample ||
      'order by ' || v_cols
      ;
  loop
    fetch v_cursor into v_bno, v_fno;
    exit when v_cursor%notfound;
    if(v_prev_fno <> v_fno or v_prev_bno <> v_bno) then
      v_cf := v_cf + 1;
    end if;
    v_prev_fno := v_fno;
    v_prev_bno := v_bno;
  end loop;

  close v_cursor;

  v_cf := v_cf + 1;
  v_acf := trunc(v_cf * 100 / &__SAMPLE);

  dbms_output.put_line('Caculated Clustering Factor = ' || v_cf);
  dbms_output.put_line('Adjusted Clusetring Factor = ' || v_acf);

end;
/

set serveroutput off

※使用例
@cf.sql
define __IND_NAME = &1
1に値を入力してください: t_cf_i1
18:20:12 SQL> define __SAMPLE = &2
2に値を入力してください: 100

Columns =  C1
Table = T_CF
Caculated Clustering Factor = 18
Adjusted Clusetring Factor = 18
```

んどの場合は前者だ。**LIST9**でその一例を紹介しよう。100万件を持つ表から1件のみを残して他のすべてを削除した状況で、索引スキャンでデータを抽出するとどうなるか。次の2つの推測が可能だ。

(1) 1,000,000以下の値は1件のみ存在するため、3ブロックのアクセスで処理される
(2) 1件のみ存在するが、Oracleは各ブロックに何の値が入っているか分からないため、データの削除前と同じく結局1,000,000以下のすべてのブロックをアクセスして処理するしかない

当然(1)のように実装されていると思いがちだが、実際には(2)のような動きをする。実行計画の統計「Buffers=2002」から推論できるが、索引のブランチブロックはデータ削除前と同じく1,000,000以下のリーフブロックに関する情報を持っていて、直接各ブロックを覗いてみないと値の有無が判断できないためである。幸いこのようなケースは滅多になく、索引の断片化を解消する作業(DROP/CREATE、COALESCE、REBUILD、SHRINK)でなくすことができる。

LIST9：索引の効率が悪くなった

```
drop table index_test purge;
create table index_test(id int);

create index index_test_idx on index_test(id);

insert into index_test select level from dual connect by level <= 1000000 ;
→ 1000000行が作成されました

delete from index_test where id > 1;
commit ;

select count(*) from index_test;
→ 1件

alter session set statistics_level = ALL ;
select /*+ index(index_test index_test_idx) */ * from index_test where id < 1000000;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
-----------------------------------------------------------------------------------------------------
| Id  | Operation         | Name           | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------
|*  1 |  INDEX RANGE SCAN| INDEX_TEST_IDX |      1 |      1 |      1 |00:00:00.01 |    2002 |
-----------------------------------------------------------------------------------------------------
→ 1件のデータ抽出で「2002」ブロックを読み取っている

alter index index_test_idx shrink space ;
select /*+ index(index_test index_test_idx) */ * from index_test where id < 1000000;
"select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));"
-----------------------------------------------------------------------------------------------------
| Id  | Operation         | Name           | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------
|*  1 |  INDEX RANGE SCAN| INDEX_TEST_IDX |      1 |      1 |      1 |00:00:00.01 |       3 |
-----------------------------------------------------------------------------------------------------
→ 読み取りのブロック数が「3」に減った
```

Tips6 索引ヒントがより使いやすくなった

従来の索引ヒントは、次のように指定する。

```
select /*+ index(t_index t_index_idx1) */
count(*)
from t_index
where c1 > 0 and c2 > 0;
```

ただし、この方法では索引名が変わると該当索引を使わなくなる可能性があるため、10gより次の文法が追加された。

```
select /*+ index(t_index t_index(c1)) */
count(*)
from t_index
where c1 > 0 and c2 > 0;
```

すなわち、索引名の代わりに構成カラムを指

定することでヒントの意味がよりクリアになり、索引名の変更時の影響を回避できるようになった。

Tips7 複雑なSQLの実行計画を簡単に解析

数ページに渡る複雑なSQLは、見ているだけで目が眩みそうになる。さらにその実行計画を解析するとなるともはや勇気が必要になるくらいの努力が必要だが、10gから追加された「QB_NAME」ヒントを使えば、このような些細な悩みはなくなることだろう。

LIST10①では、どの実行計画がSQLのどの行に当たるかは簡単に把握できない。そこでSQLにQB_NAMEヒントでブロック名を付け加える（LIST10②）とその名前が実行計画の下に表示され、どの実行計画がSQLテキストのどの部分と連結しているかが直感的に見えてくる。また、別のヒントでブロック名を別名として使う場合にも便利だ。

LIST10：複雑なSQLの実行計画

```
-- ①複雑なSQL
explain plan for
select t1.id, t1.name, t2.name, t3.name, t5.name,
       (select count(*) from t1 s where s.id = t1.id) as id1_1
from   t1, t2, t3, t5,
       ( select t4.id, t5.name
         from t4, t5
         where t4.id = t5.id and t5.name like '%c%'
       ) x
where  t1.id = t2.id
and    t2.id in (select id from t3 where name like '%b%')
and    t2.id = x.id
and    t3.id = t1.id
and    t5.id = t1.id;

select * from table(dbms_xplan.display(null,null));
-------------------------------------------------------------------------------
| Id  | Operation                    | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |     2 |   806 |   425  (10)| 00:00:06 |
|   1 |  SORT AGGREGATE              |      |     1 |    13 |            |          |
| *  2 |   TABLE ACCESS FULL          | T1   |   966 | 12558 |    59   (7)| 00:00:01 |
| *  3 |  HASH JOIN                   |      |     2 |   806 |   425  (10)| 00:00:06 |
| *  4 |   HASH JOIN                  |      |     2 |   676 |   364  (10)| 00:00:05 |
| *  5 |    HASH JOIN                 |      |     2 |   546 |   304  (10)| 00:00:04 |
| *  6 |     HASH JOIN                |      |     2 |   416 |   243  (10)| 00:00:03 |
| *  7 |      HASH JOIN               |      |     3 |   429 |   181   (9)| 00:00:03 |
| *  8 |       HASH JOIN RIGHT SEMI   |      |     3 |   390 |   121  (10)| 00:00:02 |
| *  9 |        TABLE ACCESS FULL     | T3   |     3 |   195 |    60   (9)| 00:00:01 |
|  10 |        TABLE ACCESS FULL     | T2   |  109K |  6962K|    59   (7)| 00:00:01 |
|  11 |       TABLE ACCESS FULL      | T4   |  103K |  1318K|    58   (6)| 00:00:01 |
| * 12 |      TABLE ACCESS FULL       | T5   | 82313 |  5224K|    60   (9)| 00:00:01 |
|  13 |     TABLE ACCESS FULL        | T1   | 96647 |  6134K|    58   (6)| 00:00:01 |
|  14 |    TABLE ACCESS FULL         | T5   |  102K |  6532K|    58   (6)| 00:00:01 |
|  15 |   TABLE ACCESS FULL          | T3   |  106K |  6736K|    58   (6)| 00:00:01 |
-------------------------------------------------------------------------------

-- ②QB_NAMEヒント付き複雑なSQL
alter session set statistics_level = all;

explain plan for
select /*+ qb_name(main) */ t1.id, t1.name, t2.name, t3.name, t5.name,
       (select /*+ qb_name(scalar) */ count(*) from t1 s where s.id = t1.id) as id1_1
from   t1, t2, t3, t5,
       ( select /*+ qb_name(inline) */ t4.id, t5.name
         from t4, t5
         where t4.id = t5.id and t5.name like '%c%'
       ) x
where  t1.id = t2.id
and    t2.id in (select /*+ qb_name(subquery) */ id from t3 where name like '%b%')
and    t2.id = x.id
and    t3.id = t1.id
and    t5.id = t1.id;

select * from table(dbms_xplan.display(null,null, 'ALL'));

…①と同じ実行計画…

-- 実行計画のラインごとに「qb_name」ヒントで指定した名前が表示され、実行計画と直感的にマッチングする
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------
   1 - SCALAR
   2 - SCALAR       / S@SCALAR
   3 - SEL$EA1A1EE6
   9 - SEL$EA1A1EE6 / T3@SUBQUERY
  10 - SEL$EA1A1EE6 / T2@MAIN
  11 - SEL$EA1A1EE6 / T4@INLINE
  12 - SEL$EA1A1EE6 / T5@INLINE
  13 - SEL$EA1A1EE6 / T1@MAIN
  14 - SEL$EA1A1EE6 / T5@MAIN
  15 - SEL$EA1A1EE6 / T3@MAIN

-- ③別のヒントでQB_NAME使用
explain plan for
select /*+ qb_name(main) no_unnest(@subquery) */ t1.id, t1.name, t2.name, t3.name, t5.name,
       (select /*+ qb_name(scalar) */ count(*) from t1 s where s.id = t1.id) as id1_1
from   t1, t2, t3, t5,
       ( select /*+ qb_name(inline) */ t4.id, t5.name
         from t4, t5
         where t4.id = t5.id and t5.name like '%c%'
       ) x
where  t1.id = t2.id
and    t2.id in (select /*+ qb_name(subquery) */ id from t3 where name like '%b%')
and    t2.id = x.id
and    t3.id = t1.id
and    t5.id = t1.id;
```

Tips8 実行計画の予測値と結果値を手軽に確認

実行計画を参照すると、オプティマイザの予測と実際の実行結果が確認できる。SQLチューニングの目的の1つは「オプティマイザが正しい判断をするように導く」、すなわち実行計画の予測値と実行結果値をできるだけ近づけることだ。

従来のやり方で両方の情報を確認するには、「Explain Plan」の予測値とトレースの実行結果を比較する必要があり、少し面倒だった（**LIST11**①②）。しかし、10gからは「GATHER_PLAN_STATISTICS」ヒントが追加され、簡単に参照できるようになった（LIST11③）。そのおかげで、今まで手動で比較してきた手間が省け、より効果的にSQLチューニングを実行できるようになったというわけだ。

Tips9 「動的サンプリング」による最強チューニング

統計情報がない表に対して実行計画を生成するため、「動的サンプリング」を行なうことはよく知られている。では、統計情報が最新化されている場合は動的サンプリングが必要ないのか？その場合も動的サンプリングは必要だ。より正確に表現すれば、動的サンプリングは「最高のチューニング手段の1つ」なのである。

LIST12の表は統計情報が最新化されている

ため、デフォルト状態では動的サンプリングは行なわれない。何も条件がない場合はオプティマイザの予測はほぼ実際の実行結果と一致する(①)。

だが、ここにLIKE条件を1つ追加すると、予測件数(162K)と実件数(1990K)が大きくずれてしまう(②)。OracleはLIKE条件が追加されると予測件数を5%に見積もってしまうが、これは固定値で5%程度のデータがLIKE条件に合致するという仮定に基づいているためで、予測というプロセスから必然的に発生する誤差の1つだ。

LIKE条件を増やしていくと、事態はより悪化する。LIKE条件が3個になると予測そのものの意味がなくなるほどだ(④)。このような問題点は、結合する表を追加するとその悪さの影響が明らかになる。表をフルスキャンしながら1990K回の「NESTED LOOPS」結合(ランダムアクセス)を行なってしまうのだ(⑤)。これは恐ろしい。予測件数が「407」件だったので「NESTED LOOPS」結合が有利と判断したからだ。このような大量データの結合には「HASH JOIN」がより適している。

では、この場合の解消策はどうすべきなのか? ハッシュ結合をガイドする「USE_HASH」ヒントを付けるのは正解ではない。LIKE条件によって「NESTED LOOPS」がより有利な場合もあるからだ。ここでは動的サンプリングが最適なソリューションだ。既存の統計情報から正確な予測ができない場合は、SQLが解析される際に動的サンプリングで比較的正確な情報が得られる。「dynamic_sampling」ヒントを使って動的サンプリングを行なうと、実件数に近い予測ができてハッシュ結合で実行される(⑥)。

Tips10 SQL文レベルでオプティマイザパラメータを変更

SQL文のレベルでオプティマイザパラメータを変更できるか? たまにこのような制御が必要な場面がある。従来は「alter session set……」を該当SQLの前後で設定することで実装できたが、10g R2からは「OPT_PARAM」ヒントを追加するSQLの変更で同様の制御ができるようになった。

LIST11:実行計画の参照方法

```
explain plan for
select count(*) from t_plan where c1 = 'many2';
select * from table(dbms_xplan.display());
-- ①Explain Plan:予測に基づく
-----------------------------------------------------------------------------
| Id  | Operation          | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |        |     1 |     6 |     1   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE    |        |     1 |     6 |            |          |
|*  2 |   INDEX RANGE SCAN | I_PLAN |     1 |     6 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------
→ アクセス予測件数:1件

alter session set events '10046 trace name context forever, level 12';
select /*+ gather_plan_statistics */ count(*) from t_plan where c1 = 'many2';
alter session set events '10046 trace name context off';
-- ②トレース:実際の実行結果
Rows     Row Source Operation
-------  ---------------------------------------------------
      1  SORT AGGREGATE (cr=28 pr=0 pw=0 time=2427 us)
  10000   INDEX RANGE SCAN I_PLAN (cr=28 pr=0 pw=0 time=30024 us)(object id 54166)
→ 実際のアクセス件数:1000件

select /*+ gather_plan_statistics */ count(*) from t_plan where c1 = 'many2';
select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));
-- ③gather_plan_statisticsヒント付きでの実行計画
-------------------------------------------------------------------------------------------
| Id  | Operation          | Name   | Starts | E-Rows | A-Rows |   A-Time    | Buffers |
-------------------------------------------------------------------------------------------
|   1 |  SORT AGGREGATE    |        |      1 |      1 |      1 |00:00:00.01 |      38 |
|*  2 |   INDEX RANGE SCAN | I_PLAN |      1 |      1 |  10000 |00:00:00.03 |      38 |
-------------------------------------------------------------------------------------------
→ アクセス予測件数:1件(Starts * E-Rows)、実際のアクセス件数:1000件(A-Rows)

※ Start :オペレーションの実行回数
   E-Rows :アクセスされると予測した件数
   A-Rows :SQL実行中に実際にアクセスした件数
```

LIST13:OPT_PARAMヒント

```
explain plan for
select * from t1 where c1 = :b1 or c2 = :b2 ;
select * from table(dbms_xplan.display());
-----------------------------------------------------------------------------
| Id  | Operation                     | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |       |     2 |    14 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID  | T1    |     2 |    14 |     2   (0)| 00:00:01 |
|   2 |   BITMAP CONVERSION TO ROWIDS |       |       |       |            |          |
|   3 |    BITMAP OR                  |       |       |       |            |          |
|   4 |     BITMAP CONVERSION FROM ROWIDS|    |       |       |            |          |
|*  5 |      INDEX RANGE SCAN         | T1_N1 |       |       |     1   (0)| 00:00:01 |
|   6 |     BITMAP CONVERSION FROM ROWIDS|    |       |       |            |          |
|*  7 |      INDEX RANGE SCAN         | T1_N2 |       |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------
→ B*ツリー索引のビットマップ変換でアクセスしている

explain plan for
select /*+ opt_param('_b_tree_bitmap_plans', 'false') */ * from t1 where c1 = :b1 or c2 = :b2 ;
select * from table(dbms_xplan.display());
-----------------------------------------------------------------------------
| Id  | Operation                     | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |       |     2 |    14 |     4   (0)| 00:00:01 |
|   1 |  CONCATENATION                |       |       |       |            |          |
|   2 |   TABLE ACCESS BY INDEX ROWID | T1    |     1 |     7 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN           | T1_N2 |     1 |       |     1   (0)| 00:00:01 |
|*  4 |   TABLE ACCESS BY INDEX ROWID | T1    |     1 |     7 |     2   (0)| 00:00:01 |
|*  5 |    INDEX RANGE SCAN           | T1_N1 |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------
→ B*ツリー索引スキャンで行なっている
```

例えば、データの分布などでLIST13のようにビットマップ変換によるアクセスが効果的ではないと判断した場合は、オプティマイザがビットマッププランを検討させないようにできる。

Tips11 アウトラインヒントで細やかなチューニングを

LIST14の「検証環境」で「c1 = :b1 or c2 = :b2」の条件でデータを抽出する際に、「c1」は索引スキャン、「c2」はフルスキャンのほうが最も効率が良いと判断した場合、どのようなSQLを作成すれば良いか。

まず「SQL①」のように別々のSQLを作成し、「UNION ALL」でマージする方法が考えられる。分かりやすくてシンプルで良い。ここではヒントで制御する手順を紹介する。

①「SQL②」で索引スキャンに誘導するヒント「index(t1(c1))」を使って、索引スキャン時

LIST12：動的サンプリングの効果

```
drop table t_dynamic purge;
create table t_dynamic(id int, c1 varchar2(100));

insert into t_dynamic
select rownum, object_name||'_$@' || lvl
from   user_objects,
      (select level as lvl from dual connect by level <= 10000 order by dbms_random.random) ;
→ 3,260,000行が作成されました。

create index t_dynamic_idx on t_dynamic(id);
exec dbms_stats.gather_table_stats(user, 't_dynamic', cascade=>true, no_invalidate=>false);

-- ①条件がない場合
select /*+ gather_plan_statistics */ count(*) from t_dynamic ;
→ 3,260,000件抽出

select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));
-------------------------------------------------------------------------------------
| Id  | Operation          | Name      | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------
|   1 |  SORT AGGREGATE    |           |      1 |      1 |      1 |00:00:00.44 |   14188 |
|   2 |   TABLE ACCESS FULL| T_DYNAMIC |      1 |   3257K|   3260K|00:00:09.78 |   14188 |
-------------------------------------------------------------------------------------
→ 予測件数(3257K)と実件数(3260K)がほぼ一致する、いい状態だ。

-- ②LIKE条件が1個
select /*+ gather_plan_statistics */ count(*) from t_dynamic
where c1 like '%T%' ;
→ 1,990,000件抽出

select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));
-------------------------------------------------------------------------------------
| Id  | Operation          | Name      | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------
|   1 |  SORT AGGREGATE    |           |      1 |      1 |      1 |00:00:02.65 |   14188 |
|*  2 |   TABLE ACCESS FULL| T_DYNAMIC |      1 |    162K|   1990K|00:00:07.96 |   14188 |
-------------------------------------------------------------------------------------
→ 予測件数(162K)と実件数(1990K)が大きくずれる

-- ③LIKE条件が2個
select /*+ gather_plan_statistics */ count(*) from t_dynamic
where c1 like '%T%' and c1 like '%_%' ;
→ 1,990,000件抽出

select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));
-------------------------------------------------------------------------------------
| Id  | Operation          | Name      | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------
|   1 |  SORT AGGREGATE    |           |      1 |      1 |      1 |00:00:03.28 |   14188 |
|*  2 |   TABLE ACCESS FULL| T_DYNAMIC |      1 |    8143|   1990K|00:00:09.95 |   14188 |
-------------------------------------------------------------------------------------
→ 予測件数(8143)と実件数(1990K)の差がより大きくなる

-- ④LIKE条件が3個
select /*+ gather_plan_statistics */ count(*) from t_dynamic
where c1 like '%T%' and c1 like '%_%' and c1 like '%$%' ;
→ 1,990,000件抽出

select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));
-------------------------------------------------------------------------------------
| Id  | Operation          | Name      | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------
|   1 |  SORT AGGREGATE    |           |      1 |      1 |      1 |00:00:04.84 |   14188 |
|*  2 |   TABLE ACCESS FULL| T_DYNAMIC |      1 |     407|   1990K|00:00:11.94 |   14188 |
-------------------------------------------------------------------------------------
→ 予測件数(407)と実件数(1990K)、予測の意味がなくなる

-- ⑤LIKE条件が3個 + 結合
select /*+ gather_plan_statistics */ count(*)
from t_dynamic t1, t_dynamic t2
where t1.id = t2.id and t1.c1 like '%T%' and t1.c1 like '%_%' and t1.c1 like '%$%' ;
→ 1,990,000件抽出

select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));
-----------------------------------------------------------------------------------------------
| Id  | Operation          | Name          | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-----------------------------------------------------------------------------------------------
|   1 |  SORT AGGREGATE    |               |      1 |      1 |      1 |00:00:46.13 |     3998K|   2286 |
|   2 |   NESTED LOOPS     |               |      1 |    407 |   1990K|00:01:01.72 |     3998K|   2286 |
|*  3 |    TABLE ACCESS FULL| T_DYNAMIC     |      1 |    407 |   1990K|00:00:11.94 |    14188 |      0 |
|*  4 |    INDEX RANGE SCAN | T_DYNAMIC_IDX |   1990K|      1 |   1990K|00:00:29.10 |     3984K|   2286 |
-----------------------------------------------------------------------------------------------
→ 予測件数(407)が少ないため「NESTED LOOPS」でアクセスする、1990K回のランダムアクセスが発生する

-- ⑥「dynamic_sampling」ヒント
select /*+ gather_plan_statistics dynamic_sampling(t1 2) */ count(*)
from t_dynamic t1, t_dynamic t2
where t1.id = t2.id and t1.c1 like '%T%' and t1.c1 like '%_%' and t1.c1 like '%$%' ;
→ 1,990,000件抽出

select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));
-----------------------------------------------------------------------------------------------------------------
| Id  | Operation              | Name          | Starts | E-Rows | A-Rows |   A-Time   | Buffers |  OMem |  1Mem | Used-Mem |
-----------------------------------------------------------------------------------------------------------------
|   1 |  SORT AGGREGATE        |               |      1 |      1 |      1 |00:00:07.84 |   21801 |       |       |          |
|*  2 |   HASH JOIN            |               |      1 |   2010K|   1990K|00:00:35.73 |   21801 |   63M |  5385K|  103M (0)|
|   3 |    INDEX FAST FULL SCAN| T_DYNAMIC_IDX |      1 |   3257K|   3260K|00:00:09.78 |    7613 |       |       |          |
|*  4 |    TABLE ACCESS FULL   | T_DYNAMIC     |      1 |   2010K|   1990K|00:00:09.95 |   14188 |       |       |          |
-----------------------------------------------------------------------------------------------------------------
→ 予測件数(2010K)と実件数(1990K)がほぼ一致、「HASH JOIN」でアクセスする
```

に利用されるアウトラインヒントを確認する。「dbms_xplan.display」の「outline」オプションを使うと、オプティマイザが実行計画を生成する際に必要なヒントリストを「Outline Data」セクションで確認できる

② 「SQL③」でフルスキャンに誘導するヒント「full（t1）」を使って、フルスキャン時に利用されるアウトラインヒントを確認する

③ ①②で確認したアウトラインヒントを元SQLにヒントとして追加する

このようにアウトラインヒントを直接参照し制御することで、より細かいチューニングができるようになった。

Tips12 dbms_xplanを活用しよう

9iで「dbms_xplan」パッケージが導入されてから、実行計画の使い方に一大革命が起きた。単純に実行計画を推測するレベルをはるかに超え、今やSQLトレースの代わりに、またはSQLトレースを補強するツールとして進化したのである。本特集でもここまでのサンプルで多く使ってきたので、もうその便利さに気付いているとは思うが、ここで改めてその出力結果から確認ポイントをまとめてみる。

❶述語で内部動作を確認

LIST15のSQL性能はベストな状況なのか。3070ブロックの論理読み取りを5.67秒で実行していることを見ると、少ない論理読み取りの割には実行時間が長いようだが、その理由は述語の情報より読み取れそうだ。「NAME」カラムを索引スキャンでアクセスしているが（「access（"NAME"）.」）、同様のカラムでフィルタリングが行なわれている（「filter」）。アクセスされたレコード100万件（A-Rows=1000K）ごとにTRIM関数が繰り返し実行されている。このように、SQLトレースで解析できないことも各実行計画のオペレーションごとにオプティマイザの詳細な動作を調べると見えてくる。

LIST14：アウトラインヒント

```
-- 検証環境
drop table t1 purge;
create table t1(c1 int, c2 int);
create index t1_n1 on t1(c1);
create index t1_n2 on t1(c2);

insert into t1 select level, level from dual connect by level <= 10000 ;
commit ;

exec dbms_stats.gather_table_stats(user, 't1', cascade=>true, no_invalidate=>false);

-- SQL①：「UNION ALL」で同じ表に対するフルスキャンと索引スキャンを行う
explain plan for
select /*+ full(t1) */ * from t1 where c2 = :b2
union all
select /*+ index(t1(c1)) */ * from t1 where c1 = :b1 ;
select * from table(dbms_xplan.display());
-----------------------------------------------------------------------------------------
| Id  | Operation                     | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |       |     2 |    14 |     9  (34)| 00:00:01 |
|   1 |  UNION-ALL                    |       |       |       |            |          |
|*  2 |   TABLE ACCESS FULL           | T1    |     1 |     7 |     7  (15)| 00:00:01 |
|   3 |   TABLE ACCESS BY INDEX ROWID | T1    |     1 |     7 |     2   (0)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN           | T1_N1 |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

-- SQL②：索引スキャン時のヒントを確認する
explain plan for
select /*+ use_concat index(t1(c1)) */ * from t1 where c1 = :b1 or c2 = :b2 ;
select * from table(dbms_xplan.display(null,null,'outline'));"
-----------------------------------------------------------------------------------------
| Id  | Operation                     | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |       |     2 |    14 |     4   (0)| 00:00:01 |
|   1 |  CONCATENATION                |       |       |       |            |          |
|   2 |   TABLE ACCESS BY INDEX ROWID | T1    |     1 |     7 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN           | T1_N2 |     1 |       |     1   (0)| 00:00:01 |
|*  4 |   TABLE ACCESS BY INDEX ROWID | T1    |     1 |     7 |     2   (0)| 00:00:01 |
|*  5 |    INDEX RANGE SCAN           | T1_N1 |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

Outline Data
-------------
  /*+
      BEGIN_OUTLINE_DATA
      INDEX(@""SEL$1_2"" ""T1""@""SEL$1_2"" (""T1"".""C1""))
      INDEX(@""SEL$1_1"" ""T1""@""SEL$1"" (""T1"".""C2""))
      …
      END_OUTLINE_DATA
  */
→「INDEX(@""SEL$1_2"" ""T1""@""SEL$1_2"" (""T1"".""C1""))」が「c1」に対する索引スキャンのクエリーブロック

-- SQL③：フルスキャン時のヒントを確認する
explain plan for
select /*+ use_concat full(t1) */ * from t1 where c1 = :b1 or c2 = :b2 ;
select * from table(dbms_xplan.display(null,null,'outline'));
→「outline」オプションでオプティマイザが実行計画を生成する際に必要なヒントリストを表示する
-----------------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |     2 |    14 |    13   (8)| 00:00:01 |
|   1 |  CONCATENATION      |      |       |       |            |          |
|*  2 |   TABLE ACCESS FULL | T1   |     1 |     7 |     7  (15)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL | T1   |     1 |     7 |     7  (15)| 00:00:01 |
-----------------------------------------------------------------------------------

Outline Data
-------------
  /*+
      BEGIN_OUTLINE_DATA
      FULL(@""SEL$1_2"" ""T1""@""SEL$1_2"")
      FULL(@""SEL$1_1"" ""T1""@""SEL$1"")
      :
      END_OUTLINE_DATA
  */
→ SQL②の索引スキャンブロックを除くと、「FULL(@""SEL$1_1"" ""T1""@""SEL$1"")」が「c2」に対するフルスキャンのクエ⇒リーブロック、

-- SQL④：「c1」は索引スキャン、「c2」はフルスキャンのヒントを指定する
explain plan for
select /*+ use_concat FULL(@""SEL$1_1"" ""T1""@""SEL$1"") INDEX(@""SEL$1_2"" ""T1""@"⇒
"SEL$1_2"" (""T1"".""C1"")) */ *
from t1 where c1 = :b1 or c2 = :b2 ;
select * from table(dbms_xplan.display(null,null,'outline'));
-----------------------------------------------------------------------------------------
| Id  | Operation                     | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |       |     2 |    14 |     9  (12)| 00:00:01 |
|   1 |  CONCATENATION                |       |       |       |            |          |
|*  2 |   TABLE ACCESS FULL           | T1    |     1 |     7 |     7  (15)| 00:00:01 |
|*  3 |   TABLE ACCESS BY INDEX ROWID | T1    |     1 |     7 |     2   (0)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN           | T1_N1 |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

LIST15：dbms_xplanで述語を確認

```
-- 検証環境
select /*+ gather_plan_statistics */ count(*) from t_const where name = '1234567890';
select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));
-----------------------------------------------------------------------------------------------------
| Id  | Operation          | Name     | Starts | E-Rows | A-Rows |   A-Time    | Buffers |
-----------------------------------------------------------------------------------------------------
|   1 |  SORT AGGREGATE    |          |      1 |      1 |      1 |00:00:05.67 |    3070 |
|*  2 |   INDEX RANGE SCAN | I2_CONST |      1 |   9495 |   1000K|00:00:03.00 |    3070 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access(""NAME""='1234567890')
       filter(TRIM(""NAME"")='1234567890')
```

LIST16：dbms_xplanでバインド変数を確認

```
var v_prod_id number ;
exec :v_prod_id := 125;
select count(*) from sales where prod_id = :v_prod_id ;

select plan_table_output
from   (select s.sql_id, s.child_number
         from v$sql s
        where sql_text like 'select count(*) from sales where prod_id =%'
       ) s,
       table(dbms_xplan.display_cursor(s.sql_id, s.child_number, 'allstats last +peeked_binds')) ;
-------------------------------------------------------------------------------------------------------------
| Id  | Operation                      | Name           | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------------------------
|   1 |  SORT AGGREGATE                |                |      1 |      1 |      1 |00:00:00.01 |     116 |
|   2 |   PARTITION RANGE ALL          |                |      1 |  15318 |     16 |00:00:00.01 |     116 |
|   3 |    BITMAP CONVERSION COUNT     |                |     28 |  15318 |     16 |00:00:00.01 |     116 |
|*  4 |     BITMAP INDEX FAST FULL SCAN| SALES_PROD_BIX |     28 |        |     16 |00:00:00.01 |     116 |
-------------------------------------------------------------------------------------------------------------

Peeked Binds (identified by position):
--------------------------------------
   1 - (NUMBER): 125

Predicate Information (identified by operation id):
---------------------------------------------------
   4 - filter(""PROD_ID""=:V_PROD_ID)
```

LIST17：dbms_shared_pool.purgeの使用例

```
SQL> select count(*) from sales where amount_sold > 1000 ;
  COUNT(*)
----------
     32640

SQL> select sql_id, address, hash_value
  2  from v$sql
  3  where sql_text like 'select count(*) from sales where amount_sold%';
SQL_ID                  ADDRESS  HASH_VALUE
----------------------- -------- ----------
1dua3nw528u01           29968BE4  170158081

SQL> exec sys.dbms_shared_pool.purge('29968BE4,170158081', 'C');
PL/SQLプロシージャが正常に完了しました。

SQL> select sql_id, address, hash_value
  2  from v$sql
  3  where sql_text like 'select count(*) from sales where amount_sold%';
レコードが選択されませんでした。
```

❷予測と実測の統計

LIST12と同様に、LIST15ももう1箇所改善すべき部分がある。予測件数（9495件）が実測件数（1000K件）と大きくずれていることから、実行計画が最適でない可能性が高いと考えられる。制約があると、オプティマイザは99%のデータがフィルタリングされると予測する。予測件数が本来の1%に達しているためだ。ほかにも、各オペレーションの論理読み取り（Buffers）、物理読み取り（Reads）、物理書き込み（Writes）、ハッシュおよびソート処理のメモリ使用統計が分かる。

❸ランタイムで実行状況を確認

性能異常現象関連でよく聞かれる質問がある。

「検証環境では早いクエリが、本番運用ではなぜ遅いのか?」

さまざまな原因が考えられるが、最も可能性が高いのはテスト時の実行計画とランタイム時の実行計画が異なるからだ。では、本番運用時の実行計画をどのように確認すれば良いのか。実行済みSQLの実行計画や実行統計を確認するには、dbms_xplanパッケージが最も適している。**LIST16**のように行なえば、実行計画、実行計画生成時のバインド変数、実行統計を簡単に参照できる。

Tips13 特定のSQLをハードパースさせる

頻繁ではないが、チューニング作業で特定のSQLをハードパースさせないといけない場面がある。共有プールをフラッシュしたり、関連オブジェクトの定義を変更したり、統計情報を再収集することも可能だが、いずれの方法もシステム全体へのリスクが大きい。

このような状況に適している方法が10.2.0.4パッチセットから追加された「dbms_shared_pool.purge」機能だ。**LIST17**に簡単な使い方を示したので、確認してほしい。

＊　　＊　　＊

今回紹介したさまざまな機能の中で、みなさんが初めて耳にするものもあったと思うが、いずれもエンジニアを幸せにしてくれるTipsばかりだったのではないだろうか。機会があれば、今度は初期化パラメータと統計情報を中心に性能管理者が熟知しておくべきOracleの内部動作を紹介したいと思う。 **DBM**

※本記事は著者のブログ記事の一部を翻訳、再構成してまとめたものです。

【注意】
本記事の検証結果は環境、バージョンごとに異なる可能性があるので、内容の理解と十分な検証のうえ自己責任で適用を実施してください。

趙 東郁（ちょどんうく）
自称Oracle Performance Storyteller、韓国エクセム所属。Oracleデータベースの性能関連エキスパート（Oracle ACE）として、著作、トレーニングをはじめ、ブログとASK EXEMを通じてオンライン／オフラインで知識共有活動を旺盛に行なっている。
http://dioncho.wordpress.com（English）
http://ukja.tistory.com（Korean）

金 圭福（きむぎゅうぼく）
日本エクセム（www.ex-em.co.jp）所属。AP開発、DBAの経験を経て、現在Oracleデータベースのトラブルシューティングおよびパフォーマンス改善コンサルティングを行なっている。