特集3　現場の運用管理にすぐに役立つ!

Oracle 性能遅延解消のための特選Tips集

Oracleデータベースにおいてボトルネックになりやすい処理とされる索引スキャンや全表スキャンの物理読み取り、SQL解析などによる遅延現象は、性能改善(チューニング)作業で最も頻繁に遭遇する場面となる。本誌2009年9月号特集3「エキスパートが明かすOracle性能改善Tips」に続く本稿では、特にそのような分野での性能改善Tipsと、最新のOracle 11gにおける推奨機能を中心に紹介する。多くの現場の運用担当者の性能問題に関連する具体的な悩みどころとして、物理読み取りの最小化、最適な実行計画の維持、統計情報収集の安定運用などが挙げられる。断片的な解説ながら、本特集を通してそれらの悩みが少しでも解消されれば幸いである。

韓国エクセム　**趙 東郁** CHO,Donuku
日本エクセム株式会社　**金 圭福** KIM,Gyuboku

Tips 1 索引スキャンのボトルネックを解消

性能問題が起きているシステムの集中時間帯で、SQLごとの負荷度を測定してみると面白い結果が出る。システム全体の負荷の中で、上位100のSQLに90%以上、上位10のSQLに50%以上(90%以上のケースも多々ある)の負荷が集中していることがほとんどなのである。簡単に言えば、性能問題の詳細まで調べ尽くさなくても、純粋に上位SQLまたは索引などその周辺のチューニングで期待以上の改善効果を得られるわけだ。

その中ではデザインどおりに索引スキャンを行なっているため一見最適なパフォーマンスを出しているように見えるが、意外とボトルネックとなっているSQLが多い。このような場合、検討すべき改善案の1つがフルカラム索引(SELECT文で関わっている全カラムで構成された索引。公式用語ではない)だ。

通常、索引スキャンでは「①索引で該当ブロックを探索→②条件に合致する索引データを順次読み取る→③データごとに該当の表ブロックを読み取る」を行なうが、表ブロックはディスクの別領域に格納されているため、③の処理(ランダムアクセス)が最もコストを要する部分となる。フルカラム索引はこの部分をなくすことで改善効果をもたらす(**図1**)。

LIST1は、フルカラム索引の効果を端的に表わしている。100万件のデータを持つ表から、C1の条件で1001件のデータ(C1、C2カラム)を抽出している処理だが、「C1」の索引から「C1+C2」の索引に変更したことで、処理時間の改善とともに98.2%の物理読み取りが改善されたことが分かる。このようなフルカラム索引は、対象表のデータ件数が多くて抽出カラム数が少ない上位SQLの改善で特に効果的である。

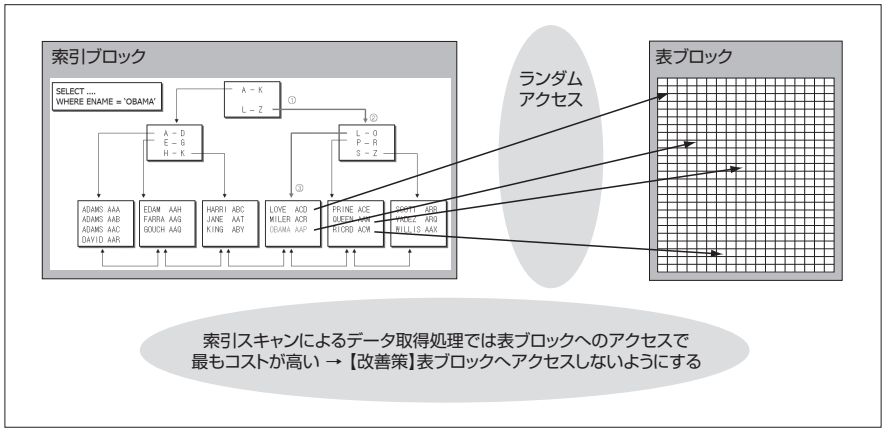

図1：索引スキャンのボトルネック

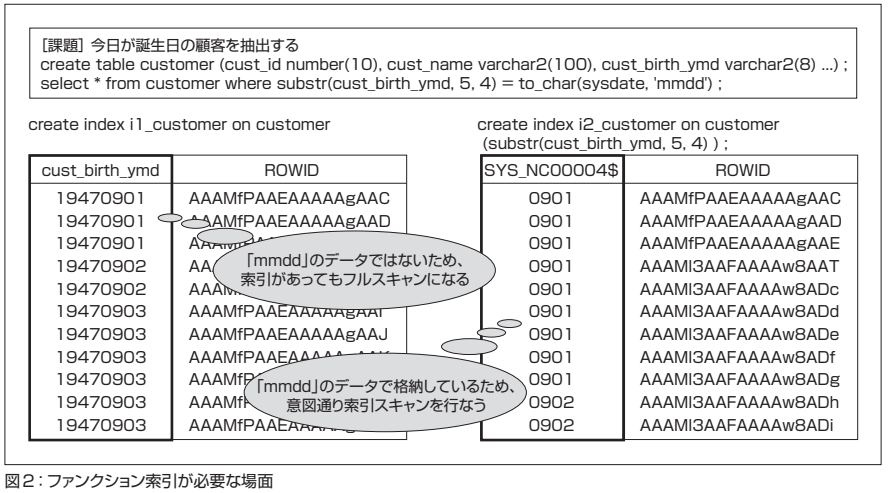

図2：ファンクション索引が必要な場面

Tips 2 ファンクション索引のフル活用

索引が存在してもSQL文で索引構成カラムが変更された場合は索引が活用できないことは

もはや常識だが、その回避方法の1つがファンクション索引だということは意外と知られていない。例えば、**図2**のように誕生日で検索を行なう場合、カラムのみを索引にすると索引は使われないが、条件文をそのまま索引にすると狙いどおりの索引スキャンを実行する。

このように、ファンクションや式に基づいて計算された値を架空のカラムとして物理的に格納しているファンクション索引は、古くはOracle8iより導入されているにもかかわらず、まだその価値が十分に認知されていないようだ。そこで、その活用例を紹介する。特に誤った論理設計を補強する場面で使うと、その効果が大きい。

① NULLデータを抽出

NULLデータは索引ブロックに格納されないため、フルスキャンでしか抽出できない。しかし**LIST2**のように該当するデータ件数が少ないと、やはりフルスキャンは無駄が多い。そもそも設計当初からこのような検索のパターンが予想されている場合は、該当カラムにNULLの代わりにダミーのデータを入れておくべきである。しかし、かなりの運用年数が経っているシステムでは、データの整合性の問題でNULLデータを入れ替えるのは容易でないことが多い。そこで、NULLデータ変換式NVLまでをファンクション索引として定義することで目的のデータを最小限の索引スキャンで抽出できるようになる(LIST2)。

② 結合キーの索引アクセス

論理設計ミスにより結合する表のキーの定義の整合性がとれていないと、開発フェーズでさまざまな不都合が生じる。**LIST3**はその1つで、SELECT文の結合キーで片方を変換せざるを得なくなり、結合処理時に片方への索引スキャンができなくなっているケースだ(LIST3では「t1→t2」のアクセス時に索引が使えない)。

例では「t1.ymd='20090807'」というデータの絞り込みが良さそうな条件があるため、なんとか使わせたいところだが、「t1→t2」のアクセスでは結合カラムの索引を使えないことをオプティマイザが分かっているため、結局「t2」表のフルスキャンからはじめて上記条件はフィルタとして使われた。

ここで、「yyyy || mm || dd」で構成するファンクション索引を追加すると、「t1→t2」のアクセスで索引スキャンが可能になるため、想定どおり「t1.ymd = '20090807'」の条件をデータの絞り込みとして使ってくれるようになる。このように、誤った論理設計も後で補強することが可能である。

③ 複数表のカラムを索引化

親子関係の表を結合してデータを抽出する際、効果的なデータ絞り込み条件で索引が作成されている場合は、当然その索引を使うデータ処理が最も効果的である。逆に**LIST4**のように、親表の条件「m.c2 between '20090707' and '20090807'」も、子表の条件「d.c4='ab'」もデータの絞り込みが良くない場合は、両方とも索引アクセスは行なわずハッシュ結合にするほうが効果的となる。

仮に、親子両方の条件を合わせた結合条件「d.c4+m.c2」のデータの絞り込みが良い場合はどうなるだろうか。通常は複数表のカラムを同時に索引にすることはできないため論外かもしれないが、LIST4のようにファンクション索引を構成すると話は変わる。最初から「d.c4+m.c2」のファンクション索引を使ってデータを絞り込むこ

LIST1：フルカラム索引の効果

```
SQL> create table t1(c1 number(10), c2 varchar2(5), c3 char(1000), c4 char(1000));
SQL> insert /*+ append */ into t1
  2  select level+1, to_char(mod(abs(dbms_random.random),100000),'FM00000'), 'a', 'b'
  3  from dual
  4  connect by level <= 1000000;

SQL> create index i1_t1 on t1 (c1) ;
SQL> select c1, c2 from t1 where c1 between 10000 and 11000;
経過: 00:00:00.25
-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |       |  1004 | 11044 |   340   (0)| 00:00:05 |
|   1 |  TABLE ACCESS BY INDEX ROWID | T1    |  1004 | 11044 |   340   (0)| 00:00:05 |
|*  2 |   INDEX RANGE SCAN           | I1_T1 |  1004 |       |     5   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
統計
----------------------------------------------------------
        407  consistent gets
        341  physical reads

SQL> drop index i1_t1;
SQL> create index i1_t1 on t1 (c1, c2) ;
SQL> select c1, c2 from t1 where c1 between 10000 and 11000;
経過: 00:00:00.03
-----------------------------------------------------------------------------------
| Id  | Operation        | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT |       |   997 | 10967 |     6   (0)| 00:00:01 |
|*  1 |  INDEX RANGE SCAN| I1_T1 |   997 | 10967 |     6   (0)| 00:00:01 |
-----------------------------------------------------------------------------------
統計
----------------------------------------------------------
         73  consistent gets
          6  physical reads
→「physical reads」が「341→6」に改善された
```

LIST2：ファンクション索引でNULLデータを抽出

```
SQL> create table t1(c1 number(10), c2 varchar2(100), c3 varchar2(8));
SQL> insert /*+ append */ into t1
  2  select level+1, rpad('a',mod(level,10)+1,'b'),
 to_char(sysdate + mod(dbms_random.random, 1000),'yyyymmdd')
  3  from dual
  4  connect by level <= 1000000;
SQL> insert /*+ append */ into t1
  2  select level+1, rpad('a',mod(level,10)+1,'b'), null
  3  from dual
  4  connect by level <= 100;
→ 100件のNULLデータを追加

SQL> create index i1_t1 on t1 (c3) ;
SQL> select * from t1 where c3 is null ;
-----------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    40 |  2840 |   833   (6)| 00:00:10 |
|*  1 |  TABLE ACCESS FULL | T1   |    40 |  2840 |   833   (6)| 00:00:10 |
-----------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("C3" IS NULL)
→ NULLデータは索引に格納されないため、フルスキャンのフィルタ条件として使われた

SQL> create index i2_t1 on t1 (nvl(c3,'99991231')) ;
SQL> select * from t1 where nvl(c3,'99991231') = '99991231' ;
-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |       |    40 |  2840 |   792   (1)| 00:00:10 |
|   1 |  TABLE ACCESS BY INDEX ROWID | T1    |    40 |  2840 |   792   (1)| 00:00:10 |
|*  2 |   INDEX RANGE SCAN           | I2_T1 |  4531 |       |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access(NVL("C3",'99991231')='99991231')
→ 変換されたNULLデータをファンクション索引に格納したため、索引スキャンのアクセス条件として使われた
```

LIST3：ファンクション索引で結合キーの索引アクセスを支援

```
SQL> create table t1(ymd varchar2(8), c1 varchar2(1000));
SQL> create table t2(yyyy varchar2(8), mm varchar2(2), dd varchar2(2), c3 varchar2(100));

SQL> insert /*+ append */ into t1
  2  select to_char(sysdate + mod(dbms_random.random,
1000),'yyyymmdd'), rpad('a',mod(level,1000)+1,'b')
  3  from dual
  4  connect by level <= 1000000;

SQL> insert /*+ append */ into t2
  2  select to_char(sysdate + mod(dbms_random.random, 1000),'yyyy'),
  3  to_char(sysdate + mod(dbms_random.random, 1000),'mm'),
  4  to_char(sysdate + mod(dbms_random.random, 1000),'dd'),
  5  rpad('a',mod(level,10)+1,'b')
  6  from dual
  7  connect by level <= 100000;

SQL> create index i1_t1 on t1 (ymd) ;
SQL> create index i1_t2 on t2 (yyyy) ;
SQL> create index i2_t2 on t2 (mm) ;
SQL> create index i3_t2 on t2 (dd) ;

SQL> select * from t1, t2
  2  where t1.ymd = t2.yyyy || t2.mm || t2.dd
  3  and   t1.ymd = '20090807' ;
→ 論理設計ミスで結合カラムの定義の整合性がとれていないため、片方を変換せざるを得ない

-------------------------------------------------------------------------------
| Id  | Operation                     | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |       |    12 |  6924 |   270   (4)| 00:00:04 |
|   1 |  TABLE ACCESS BY INDEX ROWID  | T1    |     1 |   513 |     4   (0)| 00:00:01 |
|   2 |   NESTED LOOPS                |       |    12 |  6924 |   270   (4)| 00:00:04 |
|*  3 |    TABLE ACCESS FULL          | T2    |    47 |  3008 |    81  (10)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN           | I1_T1 |     1 |       |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   3 - filter("T2"."YYYY"||"T2"."MM"||"T2"."DD"='20090807')
   4 - access("T1"."YMD"='20090807')
       filter("T1"."YMD"="T2"."YYYY"||"T2"."MM"||"T2"."DD")
→ 「t1.ymd = '20090807'」条件をデータの絞り込みで使えなかった
→ 結合キーで「t2」の索引を使えず「t1→t2」のアクセスは出来ないため、「t2→t1」のアクセスになってしまった

SQL> create index i4_t2 on t2 (yyyy || mm || dd ) ;
SQL> select * from t1, t2
  2  where t1.ymd = t2.yyyy || t2.mm || t2.dd
  3  and   t1.ymd = '20090807' ;
-------------------------------------------------------------------------------
| Id  | Operation                     | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |       |    53 | 30581 |   262   (1)| 00:00:04 |
|   1 |  TABLE ACCESS BY INDEX ROWID  | T1    |     1 |   513 |     4   (0)| 00:00:01 |
|   2 |   NESTED LOOPS                |       |    53 | 30581 |   262   (1)| 00:00:04 |
|   3 |    TABLE ACCESS BY INDEX ROWID| T2    |    47 |  3008 |    72   (0)| 00:00:01 |
|*  4 |     INDEX RANGE SCAN          | I4_T2 |   379 |       |     1   (0)| 00:00:01 |
|*  5 |    INDEX RANGE SCAN           | I1_T1 |     1 |       |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access("YYYY"||"MM"||"DD"='20090807')
   5 - access("T1"."YMD"='20090807')
       filter("T1"."YMD"="YYYY"||"MM"||"DD")
→ 「t1.ymd = '20090807'」条件をデータの絞り込みで使えた
→ 結合キーで索引を使えるため「t1→t2」のアクセスも「t2→t1」のアクセスもできる
```

とで、より効率的な検索が可能になる。

ただし、このようなユーザー定義関数によるファンクション索引は索引作成後のデータの変更が反映されないため、更新処理が発生しない読み取り専用の表あるいは一定期間更新されない場合にのみ活用できるので注意が必要だ。

このほかにもファンクション索引にはいくつか固有の制約があるので、主に以下の点は認識しておこう。

- コストベースオプティマイザ環境でのみ使われる
- ファンクション索引作成後は統計情報を再収集する必要がある
- 値が変化するSYSDATE、ROWNUMなどは指定できない
- 集計ファンクションを含めることはできない
- ユーザー定義ファンクションはDETERMINISTICを指定する必要がある
- 「QUERY_REWRITE_ENABLED=TRUE」（10g以降のデフォルト値はTRUE、それ以前はFALSE）の環境で使用可

ファンクション索引と統計情報

10g以降の環境では、索引が作成されると統計情報が自動的に収集される。これはファンクション索引でも同じだが、通常の索引と違って仮想カラムが追加されるため明示的に統計情報を再収集する必要がある。LIST5でその理由と影響が明確になっている。

ファンクション索引の作成後に索引の統計情報は自動収集されているが、追加された仮想カラムには統計情報は自動収集されていない。夜間の統計情報収集ジョブで収集されれば良いのだが、その前にファンクション索引を使うSQLが発行されると例のように予測件数（1000件）と実測件数（0件）は大きく異なってくる。統計情報を再収集するとこの現象はなくなる。

フルスキャンの効率化

表に対するフルスキャンは、対象表に対して

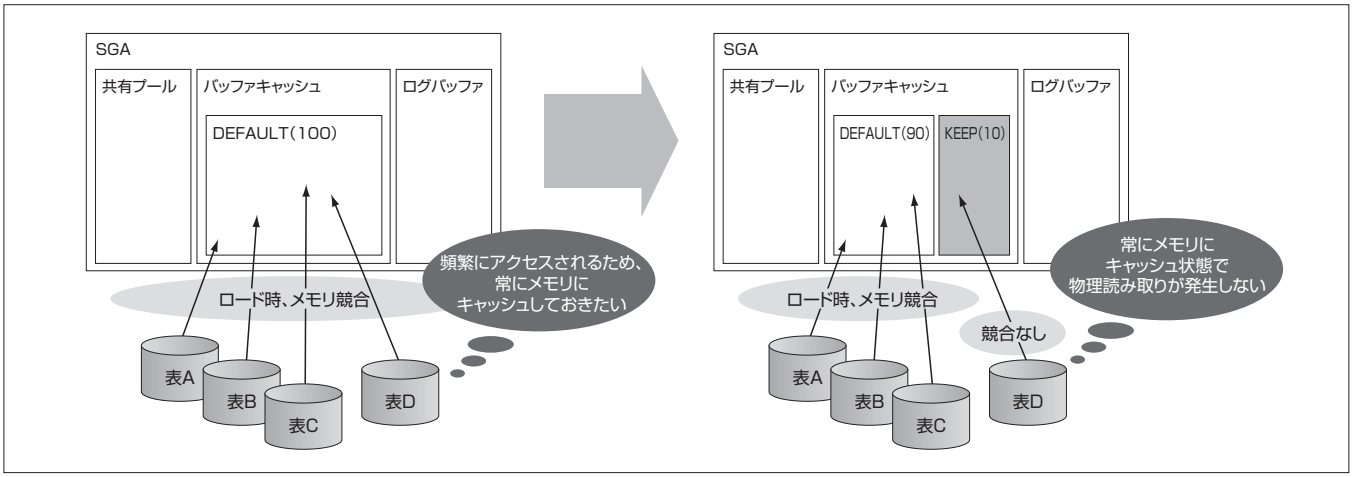

図3：KEEPバッファプールの効果

広い範囲の物理読み取りが発生するだけではなく、ほかのデータをキャッシュアウトさせるため、性能問題において上位SQLでもよくリストアップされるケースが多く、クリティカルパスになりがちだ。そのため、索引構成を見直してフルスキャンそのものを発生させないことが一般的な対処だが、検索条件がないなどやむを得ない場合はどのように改善すべきだろうか。

まずバッファキャッシュ（メモリ）を増強することが考えられるが、今回はリソースの追加ができない状況でマルチバッファプール機能の活用による改善を考える。当機能はOracle8から提供されて、バッファキャッシュを参照頻度ごとに分けることでメモリに対する競合を最小限にするために使われている。

例えば、表の全データが頻繁にアクセスされるため常にメモリにキャッシュしておかないといけない場合は、**図3**のように既存のバッファキャッシュの一部をKEEPバッファプールとして割り当てることで当該表に対する物理読み取りが発生しないようにする。その効果を**LIST6、7**の検証例からも確認できる。テストは以下のシナリオで行なわれた。

①バッファキャッシュより小さい表「T_SMALL」と大きい表「T_BIG」を作る
②「T_BIG」と「T_SMALL」に対して数回フルスキャンを行なう
③最後に「T_SMALL」に対するフルスキャンを行ない、その時の物理読み取りの統計を確認する
④KEEPバッファプールの設定あり/なしの各状況で「①〜③」の作業結果を観察する

テスト結果から、KEEPバッファプールの設定がなくデフォルトバッファのみで運用した場合（LIST6）は「T_SMALL」表に対する物理読み取りが激しく発生していることが分かる。一方、KEEPバッファプールが適切に設定されている場合（LIST7）は、「T_SMALL」表に対するフルスキャンで物理読み取りがまったく発生しなくなっている。

フルスキャン処理がボトルネックになっていてほかの手が打てない場合は、ぜひ検討してほしいものだ。

LIST4：ファンクション索引で複数表のカラムを索引化

```
SQL> create table t_master(c1 number(10), c2 varchar2(8));
SQL> create table t_detail(c1 number(10), c3 number(10), c4 varchar2(100));

SQL> insert /*+ append */ into t_master
  2  select mod(level,1000000)+1, to_char(sysdate + mod(dbms_random.random, 3000),'yyyymmdd')
  3  from dual
  4  connect by level <= 1000000;

SQL> insert /*+ append */ into t_detail
  2  select mod(level,1000000)+1, mod(level,10)+1, rpad('a',mod(level,50)+1,'b')
  3  from dual
  4  connect by level <= 10000000;

SQL> create index i1_master on t_master (c1) ;
SQL> create index i2_master on t_master (c2) ;
SQL> create index i1_detail on t_detail (c1, c3) ;
SQL> create index i2_detail on t_detail (c4) ;

SQL> select *
  2  from   t_master m, t_detail d
  3  where  m.c1 = d.c1
  4  and    m.c2 between '20090707' and '20090807'
  5  and    d.c4 = 'ab' ;
-----------------------------------------------------------------------------------
| Id  | Operation          | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |          |  1057 | 49679 | 13155   (5)| 00:02:38 |
|*  1 |  HASH JOIN         |          |  1057 | 49679 | 13155   (5)| 00:02:38 |
|*  2 |   TABLE ACCESS FULL| T_MASTER |   966 | 12558 |   632   (8)| 00:00:08 |
|*  3 |   TABLE ACCESS FULL| T_DETAIL |  200K|  6654K| 12519   (4)| 00:02:31 |
-----------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("M"."C1"="D"."C1")
   2 - filter("M"."C2"<='20090807' AND "M"."C2">='20090707')
   3 - filter("D"."C4"='ab')
→ 索引があるにもかかわらず、「m.c2」、「d.c4」カラムの条件はフィルタとして使われた
→ どちらの索引もデータの絞り込み効果が良くないと判断された

SQL> create or replace function get_c2 (p_k1 in number)
  2  return varchar2 deterministic is
  3    v_c2 varchar2(100) ;
  4  begin
  5    select c2 into v_c2
  6    from   t_master
  7    where  c1 = p_k1 ;
  8    return v_c2 ;
  9  end get_c2 ;
 10  /
→「m.c2」（結合カラム）から「m.c2」のデータを取得する関数を作成する

SQL> create index i3_detail on t_detail (c4, get_c2(c1)) ;
→「d.c4 + m.c2」で異なる表のカラムの値をひとつの索引に格納する

SQL> select *
  2  from   t_master m, t_detail d
  3  where  m.c1 = d.c1
  4  and    get_c2(d.c1) between '20090707' and '20090807'
  5  and    d.c4 = 'ab' ;
-----------------------------------------------------------------------------------
| Id  | Operation                     | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |           |   501 | 23547 |  1549   (4)| 00:00:19 |
|*  1 |  HASH JOIN                    |           |   501 | 23547 |  1549   (4)| 00:00:19 |
|   2 |   TABLE ACCESS BY INDEX ROWID | T_DETAIL  |   501 | 17034 |   912   (1)| 00:00:11 |
|*  3 |    INDEX RANGE SCAN           | I3_DETAIL |   902 |       |     9   (0)| 00:00:01 |
|   4 |   TABLE ACCESS FULL           | T_MASTER  |  1002K|    12M|   620   (6)| 00:00:08 |
-----------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("M"."C1"="D"."C1")
   3 - access("D"."C4"='ab' AND "SH"."GET_C2"("C1")>='20090707' AND
              "SH"."GET_C2"("C1")<='20090807')
→「d.c4 + m.c2」のファンクション索引に対するアクセスから始めてデータを絞り込んだ
→「d.c4 + m.c2」結合条件のデータの絞り込み効果が良いと判断された
```

LIST5：ファンクション索引作成後の、統計情報収集の必要性

```
SQL> select * from v$version;
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Prod

SQL> create table t1 as select level as c1, level as c2 from dual connect by level <= 100000 ;
SQL> exec dbms_stats.gather_table_stats(user, 't1', cascade=>true, no_invalidate=>false);

SQL> create index t1_n1 on t1(c1);
SQL> create index t1_n2 on t1(c2+1);
SQL> select index_name, last_analyzed from user_indexes where table_name = 'T1';
INDEX_NAME                     LAST_ANA
------------------------------ --------
T1_N1                          09-09-13
T1_N2                          09-09-13
→ 索引に対する統計情報は自動収集される

SQL> select column_name, last_analyzed from user_tab_cols where table_name = 'T1';
COLUMN_NAME                    LAST_ANA
------------------------------ --------
C1                             09-09-13
C2                             09-09-13
SYS_NC00003$
→ 追加された仮想カラムの統計情報は自動収集されない

SQL> select /*+ gather_plan_statistics */ * from t1 where c2+1 = 1;
レコードが選択されませんでした。
SQL> select * from table(dbms_xplan.display_cursor(null, null, 'allstats last'));
-------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name  | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-------------------------------------------------------------------------------------------------------
|   1 |  TABLE ACCESS BY INDEX ROWID| T1    |      1 |   1000 |      0 |00:00:00.08 |       2 |      4 |
|*  2 |   INDEX RANGE SCAN          | T1_N2 |      1 |    400 |      0 |00:00:00.08 |       2 |      4 |
-------------------------------------------------------------------------------------------------------
→ カラムに統計情報がない場合、ファンクション索引を使う予測件数（1000件）と実測件数（0件）は大きく異なる

SQL> exec dbms_stats.gather_table_stats(user, 't1', cascade=>true, no_invalidate=>false);
SQL> select column_name, last_analyzed from user_tab_cols where table_name = 'T1';
COLUMN_NAME                    LAST_ANA
------------------------------ --------
C1                             09-09-13
C2                             09-09-13
SYS_NC00003$                   09-09-13

SQL> select /*+ gather_plan_statistics */ * from t1 where c2+1 = 1;
SQL> select * from table(dbms_xplan.display_cursor(null, null, 'allstats last'));
-------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name  | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------------
|   1 |  TABLE ACCESS BY INDEX ROWID| T1    |      1 |      1 |      0 |00:00:00.01 |       2 |
|*  2 |   INDEX RANGE SCAN          | T1_N2 |      1 |      1 |      0 |00:00:00.01 |       2 |
-------------------------------------------------------------------------------------------------
→ カラムに統計情報がある場合、ファンクション索引を使う予測件数（0件）と実測件数（0件）が一致する
→ ファンクション索引の作成後に、統計情報を明示的に再収集する必要がある
```

SQL解析処理の遅延現象の解消

データベース処理で、I/Oの次にボトルネックになりやすい部分がSQL解析処理だ。SQL解析処理では「①SQLテキストに対する文法／権限などのチェック→②共有プールでの存在チェック→③メモリ獲得および実行計画作成→④実行計画の格納」が行なわれる（**図4**）。

ここで新しいSQLに対しては最もコストのかかる③と④の処理が追加で実行されるが、このような一連の解析処理を「ハードパース」と言う。そのため、SQLは共有できるようにバインド化することが推奨されているが、すでに運用中のシステムでリテラルSQLの問題が発見されても、全SQLを修正することはなかなかできないのが実情だ。

このような状況でリテラルSQLを自動で共有させるために、CURSOR_SHARINGパラメータを「EXACT」（デフォルト）から「FORCE」か「SIMILAR」に変更することが検討される。しかし、予測と異なって「SIMILAR」設定ではSQLが共有されない場合がある（**図5**）。以下の条件では共有されないため、激しいハードパースを原因とする性能遅延現象が発生する可能性があるので注意が必要だ。

- 等価条件「＝」ではないSQL（例：＜、＞、＞＝、＜＝、LIKE）
- 条件カラムにヒストグラムの統計情報が存在する場合

LIST8でその検証結果を確認すると、「CURSOR_SHARING=SIMILAR、カラムのヒストグラム統計あり」では、リテラルSQLが共有されずハードパースされた。また「CURSOR_SH

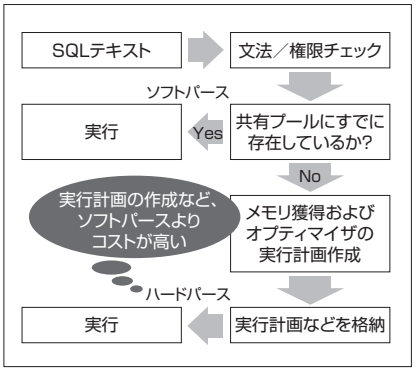

図4：ハードパース

LIST6：KEEPバッファプールの適用前

```
SQL> alter system set db_cache_size = 100M ;
SQL> alter system set db_keep_cache_size = 0M ;

SQL> create table t_small(c1 char(1000));
SQL> create table t_big(c1 char(1000));

SQL> insert /*+ append */ into t_small select 'a' from dual connect by level <= 50000;
SQL> insert /*+ append */ into t_big select 'b' from dual connect by level <= 120000;
SQL> select segment_name, bytes/(1024 * 1024)
  2  from dba_segments where lower(segment_name) in ('t_small', 't_big') ;
SEGMENT_NAME         BYTES/(1024*1024)
-------------------- -----------------
T_BIG                              136
T_SMALL                             57
→「T_BIG」、「T_SMALL」のサイズ（193MB）がバッファキャッシュのサイズ（100MB）を超えている

SQL> alter system flush buffer_cache;
SQL> select count(*) from t_big;
SQL> select count(*) from t_small;
SQL> select count(*) from t_big;
SQL> select count(*) from t_small;

SQL> set autotrace traceonly
SQL> select count(*) from t_small;
-----------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         |     1 |  1601   (1)| 00:00:20 |
|   1 |  SORT AGGREGATE    |         |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T_SMALL | 50864 |  1601   (1)| 00:00:20 |
-----------------------------------------------------------------------------
統計
----------------------------------------------------------
       7153  consistent gets
       7141  physical reads
→「T_SMALL」表に対するフルスキャンで物理読み取りが多発している
```

LIST7：KEEPバッファプールの適用後

```
SQL> alter system set db_cache_size = 40M ;
SQL> alter system set db_keep_cache_size = 60M ;

SQL> drop table t_small purge;
SQL> drop table t_big purge;
SQL> create table t_small(c1 char(1000)) storage (buffer_pool keep);
→「T_SMALL」表の「buffer_pool」属性をKEEPバッファプールに設定する
SQL> create table t_big(c1 char(1000));

SQL> insert /*+ append */ into t_small select 'a' from dual connect by level <= 50000;
SQL> insert /*+ append */ into t_big select 'b' from dual connect by level <= 120000;
SQL> select segment_name, bytes/(1024 * 1024)
  2  from dba_segments where lower(segment_name) in ('t_small', 't_big') ;
SEGMENT_NAME         BYTES/(1024*1024)
-------------------- -----------------
T_BIG                              136
T_SMALL                             57
→「T_SMALL」表の全データ（57MB）はKEEPバッファプール（60MB）にキャッシュされるサイズ

SQL> select count(*) from t_big;
SQL> select count(*) from t_small;
SQL> select count(*) from t_big;
SQL> select count(*) from t_small;

SQL> set autotrace traceonly
SQL> select count(*) from t_small;
-----------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         |     1 |  1601   (1)| 00:00:20 |
|   1 |  SORT AGGREGATE    |         |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T_SMALL | 50475 |  1601   (1)| 00:00:20 |
-----------------------------------------------------------------------------
統計
----------------------------------------------------------
       7153  consistent gets
          0  physical reads
→「T_SMALL」表に対するフルスキャンで物理読み取りが発生しない
```

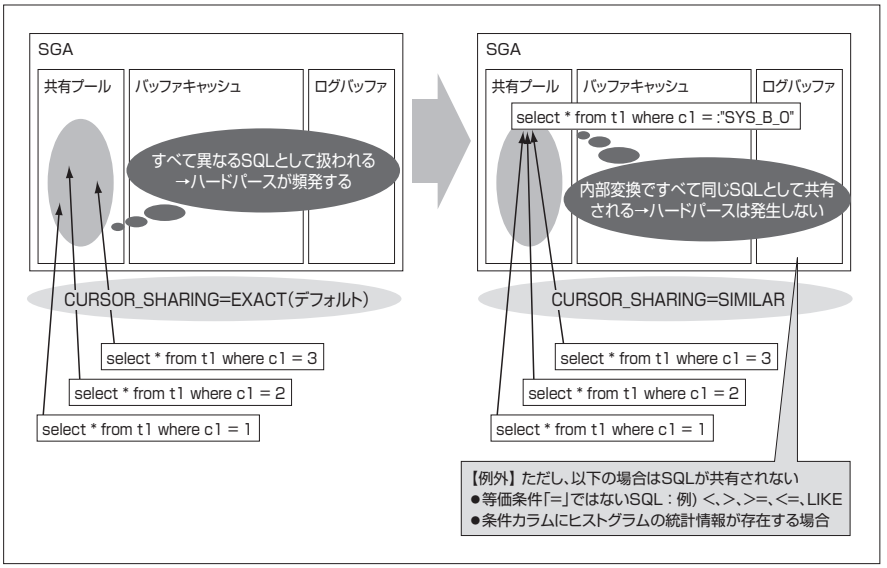

図5：「CURSOR_SHARING=SIMILAR」設定効果

ARING＝FORCE」ではリテラルSQLは自動的に共有された。「CURSOR_SHARING＝SIMILAR、カラムのヒストグラム統計なし」の状況では「＝」のSQLのみ共有され、「＞」のSQLは共有されなかった。

このように「CURSOR_SHARING＝SIMILAR」設定下の環境でSQL解析の処理時間が増えていく現象がある場合は、上記のような例外をチェックしてみよう。

GRANTがカーソルを無効にする

DDL（Data Definition Language）を発行すると、共有プールの関連カーソルがINVALIDされることはよく知られているが、この動作は意外にGRANTも同じである。GRANTを行なうと、関連するオブジェクトに依存する共有カーソルがINVALIDされ、ハードパースが急増する可能性がある。

LIST9で簡単に確認しよう。表、シノニム、ファンクションを参照するSQLを実行した後、表、シノニムを対象にGRANTを実行すると、全SQLは共有プールからなくなる。

統計情報収集とハードパース問題

SQLの共有カーソルをINVALIDにしてハードパースを起こす主な原因には、関連オブジェクトに対するDDL以外にも統計情報の再収集が挙げられる。ANALYZEコマンドで統計情報を収集すると、関連共有カーソルがINVALIDになってその直後に集中的なハードパースが発生する。このような現象は9i環境でDBMS_STATSを使った場合も同様だ。ただし、「no_invalidate＝TRUE」（デフォルトはFALSE）の引数設定の場合は上記現象は発生しない。

しかし、これには悩ましいジレンマがある。「TRUE」の設定ではハードパース問題が発生しない反面、意図的にまたは自然に共有カーソルがキャッシュアウトされない限り新しい統計情報がSQLの実行計画に反映されないのである。かたや「FALSE」の場合はハードパース問題が発生する。その解決策として、10g（～11gも）

LIST8：「CURSOR_SHARING=SIMILAR」設定効果

```
SQL> create table t_cursor_sharing(c1 int, c2 varchar2(100));
SQL> create index i1_cursor_sharing on t_cursor_sharing(c1);
SQL> insert into t_cursor_sharing
  2  select level, rpad('A',mod(level,100),'qfqwg') from dual connect by level <= 10000 ;
SQL> insert into t_cursor_sharing
  2  select 10000, rpad('B',mod(level,100),'qfqwg') from dual connect by level <= 10000 ;

 -- ケース①：CURSOR_SHARING=SIMILAR、カラムのヒストグラム統計あり
SQL> alter system set cursor_sharing = similar ;
SQL> exec dbms_stats.set_param ('method_opt','for all columns size auto');
SQL> exec dbms_stats.gather_table_stats ('sh', 't_cursor_sharing');
SQL> select table_name, column_name, last_analyzed, histogram
  2  from   dba_tab_cols
  3  where  lower( table_name ) = 't_cursor_sharing' and lower( column_name ) = 'c1' ;
TABLE_NAME           COLUMN_NAM LAST_ANA HISTOGRAM
-------------------- ---------- -------- ---------------
T_CURSOR_SHARING     C1         09-09-14 HEIGHT BALANCED

SQL> alter system flush shared_pool ;
 --「select count(*) from t_cursor_sharing where c1 = 1..1000」を実施
 --「select count(*) from t_cursor_sharing where c1 > 1..1000」を実施

SQL> select sql_text, version_count from v$sqlarea
  2  where sql_text like 'select count(*) from t_cursor_sharing where c1%' ;
SQL_TEXT                                                         VERSION_COUNT
---------------------------------------------------------------- -------------
select count(*) from t_cursor_sharing where c1 > :"SYS_B_0"              1000
select count(*) from t_cursor_sharing where c1 = :"SYS_B_0"              1000
→「CURSOR_SHARING=SIMILAR、カラムのヒストグラム統計あり」では、全SQLがハードパースされた

 -- ケース②：CURSOR_SHARING=FORCE、カラムのヒストグラム統計あり
SQL> alter system set cursor_sharing = force ;
SQL> exec dbms_stats.set_param ('method_opt','for all columns size auto');
SQL> exec dbms_stats.gather_table_stats ('sh', 't_cursor_sharing');
SQL> select table_name, column_name, last_analyzed, histogram
  2  from   dba_tab_cols
  3  where  lower( table_name ) = 't_cursor_sharing' and lower( column_name ) = 'c1' ;
TABLE_NAME           COLUMN_NAM LAST_ANA HISTOGRAM
-------------------- ---------- -------- ---------------
T_CURSOR_SHARING     C1         09-09-14 HEIGHT BALANCED

SQL> alter system flush shared_pool ;
 --「select count(*) from t_cursor_sharing where c1 = 1..1000」を実施
 --「select count(*) from t_cursor_sharing where c1 > 1..1000」を実施

SQL> select sql_text, version_count from v$sqlarea
  2  where sql_text like 'select count(*) from t_cursor_sharing where c1%' ;
SQL_TEXT                                                         VERSION_COUNT
---------------------------------------------------------------- -------------
select count(*) from t_cursor_sharing where c1 > :"SYS_B_0"                1
select count(*) from t_cursor_sharing where c1 = :"SYS_B_0"                1
→「CURSOR_SHARING=FORCE、カラムのヒストグラム統計あり」では、全SQLは共有された

 -- ケース③：CURSOR_SHARING=SIMILAR、カラムのヒストグラム統計なし
SQL> alter system set cursor_sharing = similar ;
SQL> exec dbms_stats.set_param ('method_opt','for all columns size 1');
SQL> exec dbms_stats.gather_table_stats ('sh', 't_cursor_sharing');
SQL> select table_name, column_name, last_analyzed, histogram
  2  from   dba_tab_cols
  3  where  lower( table_name ) = 't_cursor_sharing' and lower( column_name ) = 'c1' ;
TABLE_NAME           COLUMN_NAM LAST_ANA HISTOGRAM
-------------------- ---------- -------- ---------------
T_CURSOR_SHARING     C1         09-09-14 NONE

SQL> alter system flush shared_pool ;
 --「select count(*) from t_cursor_sharing where c1 = 1..1000」を実施
 --「select count(*) from t_cursor_sharing where c1 > 1..1000」を実施

SQL> select sql_text, version_count from v$sqlarea
  2  where sql_text like 'select count(*) from t_cursor_sharing where c1%' ;
SQL_TEXT                                                         VERSION_COUNT
---------------------------------------------------------------- -------------
select count(*) from t_cursor_sharing where c1 > :"SYS_B_0"              1000
select count(*) from t_cursor_sharing where c1 = :"SYS_B_0"                1
→「CURSOR_SHARING=SIMILAR、カラムのヒストグラム統計なし」では、「＝」SQLのみ共有された
```

LIST9：GRANTがカーソルを無効にする

```
「SQL-B」以降


SQL> create table t1(c1 int, c2 int);
SQL> create or replace public synonym syn1 for t1 ;
SQL> create or replace function func1(v1 int)
  2  return number is
  3    v_value int;
  4  begin
  5    select max(c2) into v_value from t1 where c1 = v1 ;
  6    return v_value + 1;
  7  end;
  8  /

SQL> select /*+ grant_test1 */ c1, c2 from t1 ;
SQL> select /*+ grant_test2 */ c1, func1(c1) from t1 ;
SQL> select /*+ grant_test3 */ c1, c2 from syn1 ;
SQL> with s1 as (select count(*) as case1 from v$sql where
sql_text like 'select /*+ grant_test1 */%'),
  2  s2 as (select count(*) as case2 from v$sql where
sql_text like 'select /*+ grant_test2 */%'),
  3  s3 as (select count(*) as case3 from v$sql where
sql_text like 'select /*+ grant_test3 */%')
  4  select case1, case2, case3 from s1, s2, s3 ;  --  以降「SQL-A」
     CASE1      CASE2      CASE3
---------- ---------- ----------
         1          1          1

SQL> grant select on t1 to scott;
--「SQL-A」実行結果 → 関連SQLはすべてINVALIDされた
     CASE1      CASE2      CASE3
---------- ---------- ----------
         0          0          0

--「SQL-B」実行
SQL> grant select on syn1 to scott;
--「SQL-A」実行結果 → 関連SQLはすべてINVALIDされた
     CASE1      CASE2      CASE3
---------- ---------- ----------
         0          0          0
```

からは「AUTO_INVALIDATE」の新たなオプションが追加された。一定期間（デフォルト5時間）に分散して各共有カーソルをそれぞれINVALIDにすることで、瞬間的なハードパースの急増現象はこれ以上悩みではなくなった（**図6**）。

Tips 8 バインドSQLと最適実行計画

リテラルSQLが原因となる弊害の対策としてSQLのバインド化が推奨されるが、バインドSQLにも副作用があった。**図7**を参考にしよう。データの分布から「:b1 =1」の場合は表のフルスキャン、「:b1＝99」の場合は索引スキャンが有利だが、8iまではバインド変数の値が見えなかったため、すべての値に対して索引スキャンを行なっていた。9iになってから初回のみバインド変数の値を参照するようになったが、カラムのヒストグラム統計が自動作成されずデータの分布が分からなかったため、同じく索引スキャンになっていた。

10g環境では、カラムのヒストグラム統計が通常自動的に作成されるため、バインド変数の値に基づいて実行計画を作成するが、2回目からは既存の実行計画を利用する仕組みになった。例えば、1回目に「:b1＝1」で実行されると、2回目以降は「:b1=99」の値でもすべてフルスキャンで実行されてしまう。逆パターンも同じく、1回目に「:b1＝99」で実行されると次からは「:b1＝1」の場合も索引スキャンで実行されてしまう動作となる。しかし、結果的に初回実行時の値に左右されるため、より不安定な動きをしてしまうようになった。

LIST10：11gの「Adaptive Cursor Sharing」機能

```
SQL> create table t1(id int, name char(10));
SQL> create index i1_t1 on t1(id);
SQL> insert into t1 select 1, 'name' from all_objects where rownum <= 100000;
SQL> insert into t1 values(99, 'name');

SQL> exec :id := 1;
SQL> select count(name) from t1 where id = :id;
-------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |       |       |    69 (100)|          |
|   1 |  SORT AGGREGATE    |      |     1 |    14 |            |          |
|*  2 |   TABLE ACCESS FULL| T1   | 65609 |   896K|    69   (2)| 00:00:01 |
-------------------------------------------------------------------------

-- 「exec :id := 1」で数回実行

SQL> exec :id := 99;
SQL> select count(name) from t1 where id = :id;
-------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |       |       |    69 (100)|          |
|   1 |  SORT AGGREGATE    |      |     1 |    14 |            |          |
|*  2 |   TABLE ACCESS FULL| T1   | 65609 |   896K|    69   (2)| 00:00:01 |
-------------------------------------------------------------------------
→ 初期は適切な実行計画ではない、索引スキャンがより有利な実行計画

-- 「exec :id := 99」で数回実行
-- 「exec :id := 1」で数回実行

SQL> exec :id := 99;
SQL> select count(name) from t1 where id = :id;
-----------------------------------------------------------------------------------
| Id  | Operation                    | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |       |       |       |     2 (100)|          |
|   1 |  SORT AGGREGATE              |       |     1 |    14 |            |          |
|   2 |   TABLE ACCESS BY INDEX ROWID| T1    |     1 |    14 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN          | I1_T1 |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------
→ 異なる値のSQLが繰り返し実行されてから、やっと適切な実行計画になった

SQL> exec :id := 1;
SQL> select count(name) from t1 where id = :id;
-------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |       |       |    69 (100)|          |
|   1 |  SORT AGGREGATE    |      |     1 |    14 |            |          |
|*  2 |   TABLE ACCESS FULL| T1   | 65609 |   896K|    69   (2)| 00:00:01 |
-------------------------------------------------------------------------

SQL> exec :id := 10;
SQL> select count(name) from t1 where id = :id;
-----------------------------------------------------------------------------------
| Id  | Operation                    | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |       |       |       |     2 (100)|          |
|   1 |  SORT AGGREGATE              |       |     1 |    14 |            |          |
|   2 |   TABLE ACCESS BY INDEX ROWID| T1    |     1 |    14 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN          | I1_T1 |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------
→ 他の値のみも適切な実行計画になる
```

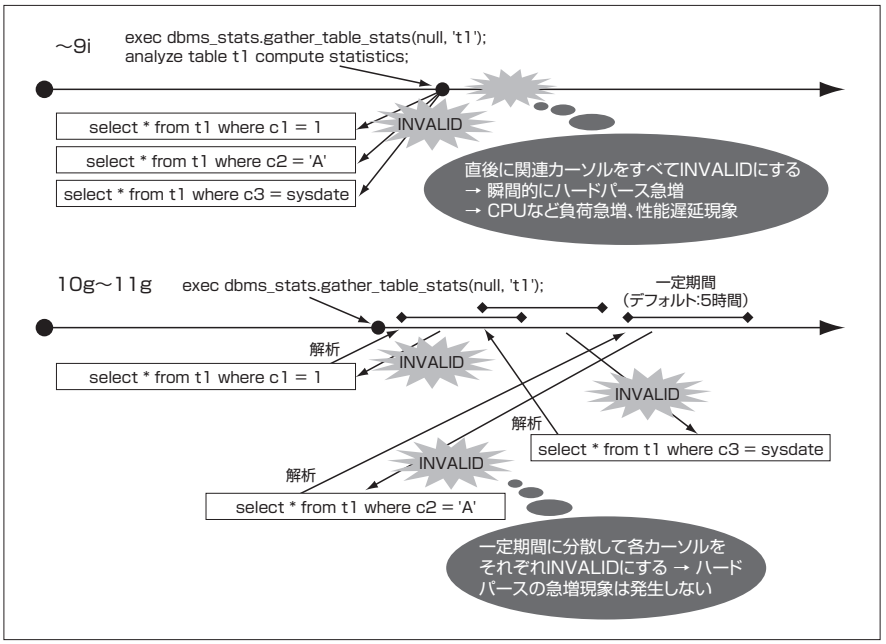

図6：統計情報収集とSQL共有カーソル

このような長年の努力の歴史があるからこそ、11gのAdaptive Cursor Sharing機能[注]は喜ばれる改善の1つとして評価すべきだと考える。**LIST10**で簡単にその検証結果を確認しよう。バインド変数の値をフルスキャンが有利な「1」と索引スキャンが有利な「99」をスイッチしながら複数回実行すると、最初は適切な実行計画ではなかったケースも最終的には適切な実行計画に変わっていくことが確認できる。このような傾向の動きはほかの値でも同じだ。

このように改善された機能も頼もしいが、改善機能を使えない環境（<=10gなど）で、例のようなデータ分布の認識のうえ偏ったデータの検索を行なう場合は、あえてリテラルSQL（例：「select * from t1 where c1 = 1」）を活用するこ

注：オプティマイザがバインドSQLのバインド変数の値を参考にして柔軟にカーソルを共有する11gの新機能。

とも古典的ながら秘策とも言えるものだ。

Tips 9 11gの複数カラムの統計

実行計画を作成するオプティマイザが、より正確な予測をするために克服しなければならない課題がいくつかある。その1つが、複数カラムの相関性に関するものだ。

LIST11の場合、国と年を持つ表と「country='Japan' and city='Okinawa'」という条件がある。データから2つのカラムが密接な関係にあることは分かるが、今まではオプティマイザがその事実を知るはずがなかった。しかし、11gで複数カラムに関する統計を作成することが可能となった。その恩恵のありがたみはLIST11のケースで劇的に表われている。

「国+都市」のヒストグラム統計がなかった時には、全体データ「20002」件の中で「国+都市」が7種類のことから「20002 * 1/7 ≒ 2857」件と予測したため、正確な予測ができず、フルスキャンを選んだ。複数カラムのヒストグラム統計ができてからは、「1」件と正確に予測し、索引スキャンを行なうようになったのだ。

このようにカラム間の相関性が原因で非効率的な実行計画になった場合は、オプティマイザに正確な情報を提供することを心掛けよう。

LIST11：11gの複数列の統計

```
SQL> create table t_ext_stat(country varchar2(20), city varchar2(10), code char(5));

SQL> insert into t_ext_stat
  2  select 'Korea', decode(mod(rownum,2),0, 'Seoul', 'Busan'), rpad(rownum,5,' ')
  3  from all_objects where rownum <= 10000;
SQL> insert into t_ext_stat
  2  select 'Korea', 'Jeju', '11111' from dual;
SQL> insert into t_ext_stat
  2  select 'Japan', decode(mod(rownum,3),0,'Tokyo', 1, 'Osaka', 'Kyoto'), rpad(rownum,5,' ')
  3  from all_objects where rownum <= 10000;
SQL> insert into t_ext_stat
  2  select 'Japan', 'Okinawa', '11111' from dual;

SQL> create index t_ext_stat_idx on t_ext_stat(country, city);

SQL> exec dbms_stats.gather_table_stats(user,'t_ext_stat');
SQL> select * from t_ext_stat where country = 'Japan' and city = 'Okinawa';
-------------------------------------------------------------------------------
| Id  | Operation          | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |             |  2857 | 51426 |    19   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL | T_EXT_STAT  |  2857 | 51426 |    19   (0)| 00:00:01 |
-------------------------------------------------------------------------------
→「国+都市」のヒストグラム統計がなかったため正確な予測ができず、フルスキャンを行なった
→ 全体データ「20002」件の中で「国+都市」が7種類なので「20002 * 1/7 ≒ 2857」件と予測した

SQL> exec dbms_stats.gather_table_stats(user,'t_ext_stat', -
> method_opt=>'for columns size skewonly (country, city)' );
→「国+都市」の相関性に関するヒストグラム統計を作成する

SQL> select * from t_ext_stat where country = 'Japan' and city = 'Okinawa';
-------------------------------------------------------------------------------------------
| Id  | Operation                     | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                 |     1 |    19 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID  | T_EXT_STAT      |     1 |    19 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN            | T_EXT_STAT_IDX  |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
→「国+都市」のヒストグラム統計があったため、正確な予測ができて、索引スキャンを行なった
→ ヒストグラム統計により「1」件と予測
```

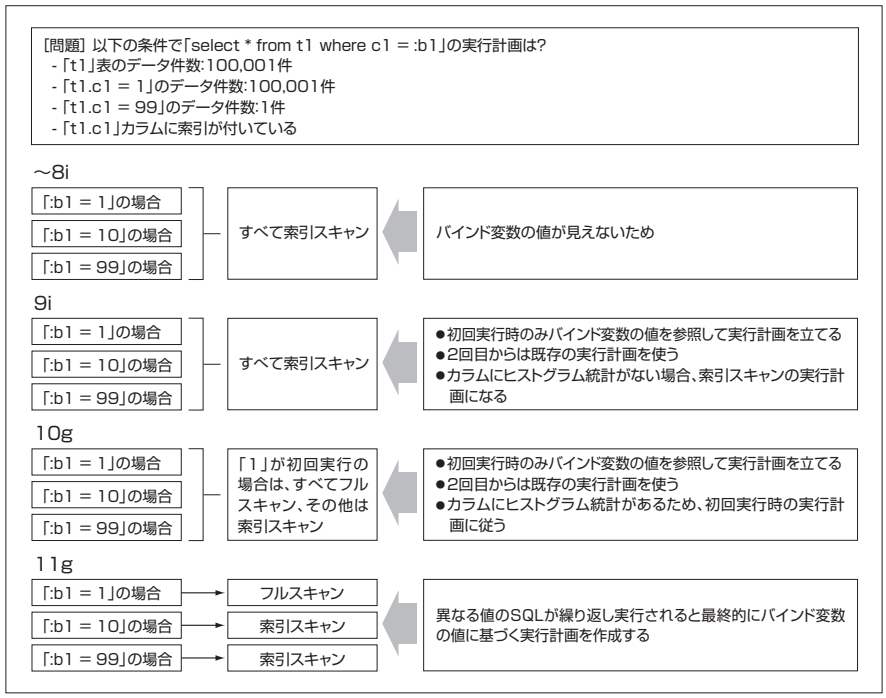

図7：バインドSQL問題の進化

Tips 10 索引を適用前に評価する

新しい索引を追加すると、関連表にアクセスするほかのSQLの実行計画に影響を与える可能性があるため、索引追加でもほかのユーザーの使用状況を考慮しながら慎重な評価を行なう必要があった。11gからは、新規索引の効果および影響を適用前に評価するINVISIBLE索引機能が追加されたため、ほかのユーザーの使用状況は気にせずに、比較的簡単に評価ができるようになった。使い方のポイントは次のとおりだ。

- 評価時には「INVISIBLE」属性で作る
- 評価時には、セッションレベルで「OPTIMIZER_USE_INVISIBLE_INDEXES」パラメータを「TRUE」にするか、あるいは個々のSQLに「USE_INVISIBLE_INDEXES」ヒントを追加することでオプティマイザに認識させる
- 評価が終わったら、追加する索引の属性を「VISIBLE」に変更する

LIST12からその実力を見ると、データ分布の良い「I1_INVISIBLE」索引は使われて、データ分布の悪い「I2_INVISIBLE」索引は使われずフルスキャンが行なわれたことから、索引に対する評価が正確にできることが分かる。

Tips 11 トレース取得作業の効率化

性能改善作業でトレースを取得する場面はしばしば出てくるので、トレース取得作業そのも

LIST12：索引を適用前に評価する

```
SQL> drop table t_invisible purge;
SQL> create table t_invisible(id int, id2 int);
SQL> insert into t_invisible
  2  select level, decode(mod(level,2),1,100000-level+1,level)
  3  from dual connect by level <= 100000 ;

SQL> create index i1_invisible on t_invisible(id) invisible ;
SQL> create index i2_invisible on t_invisible(id2) invisible ;
SQL> select index_name, clustering_factor from user_indexes where table_name = 'T_INVISIBLE' ;
INDEX_NAME                                                       CLUSTERING_FACTOR
---------------------------------------------------------------- -----------------
I2_INVISIBLE                                                             99994
I1_INVISIBLE                                                               204
→「I1_INVISIBLE」のクラスタ係数は良いが、「I2_INVISIBLE 」のクラスタ係数は悪い
→「I1_INVISIBLE」は索引スキャンに適切、「I2_INVISIBLE 」は索引スキャンに不適切

SQL> alter session set optimizer_use_invisible_indexes=true;
SQL> select * from t_invisible where id between 1 and 100;
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |   100 |   900 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID | T_INVISIBLE   |   100 |   900 |     3   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | I1_INVISIBLE  |   100 |       |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
→「I1_INVISIBLE」を経由する索引スキャンが行なわれた

SQL> select * from t_invisible where id2 between 1 and 100;
-------------------------------------------------------------------------------
| Id  | Operation          | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |             |   100 |   900 |    69   (2)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL | T_INVISIBLE |   100 |   900 |    69   (2)| 00:00:01 |
-------------------------------------------------------------------------------
→「I2_INVISIBLE」の索引スキャンの代わりに、フルスキャンが行なわれた

SQL> alter session set optimizer_use_invisible_indexes=false;
SQL> alter index i1_invisible visible;
SQL> drop index i2_invisible ;
```

LIST13：トレース取得作業の効率化

```
< trace_on.sql > -- 事前に要作成
alter session set events '&1 trace name context forever, level &2';

< trace_off.sql > -- 事前に要作成
alter session set events '&1 trace name context off';

< trace_file.sql > -- 事前に要作成
column trace_file_name new_value trace_file

-- フルパスでトレースファイル名を取得
select d.value||'/' ||p.value||'_ora_' ||s.spid||'.trc' as trace_file_name
from   (
        select value
        from   v$parameter
        where  name = 'instance_name'
       ) p ,
       (
        select value
        from   v$parameter
        where  name = 'user_dump_dest'
       ) d ,
       (
        select spid
        from   v$process
        where  addr = (
                       select paddr
                       from   v$session
                       where  sid = (
                                     select sid
                                     from   v$mystat
                                     where  rownum = 1
                                    )
                      )
       ) s ;

prompt &trace_file

< tkprof > -- 事前に要作成
host tkprof &trace_file &1

-- 上記ファイルの格納場所で以下のスクリプトをSQL*PLUSに貼り付ける
@trace_on 10046 12
select count(*) from t1 where c1 = 1; -- ここでトレースを取りたい処理を行なう
@trace_off
@trace_file
@tkprof tkprof.log
ed tkprof.log
```

のを簡素化することだけで全体作業がかなり効率化される。ツールも使えるが、**LIST13**のファイルを作成してから下段部分のスクリプトをSQL*PLUSコマンドに貼り付けるだけで、トレース結果がポップアップされるので意外と便利である。

まとめ

性能改善作業と言っても、具体的な作業イメージがピンと来ない部分もあるかもしれないが、運用環境でのボトルネックを見極めて最適なリソースや手段を動員して最終的にボトルネックを取り除く作業となる。本稿では、多くの現場で抱えているI/O遅延やSQL解析に関する問題を取り上げて、比較的低コストで大きい効果が期待できるTipsと、関連するOracleの内部動作の仕組みをいくつか紹介した。類似の性能問題で困っている場合はぜひ参考にしてほしい。

また今後、誌面が許されるのであれば、次回は索引に関して誤解されやすい仕組みと、ERPパッケージなどSQL文の修正ができない場合の改善方法について触れてみたいと思う。 **DBM**

※本記事のLISTは誌面の制約上、実検証ログから抜粋／編集したものです。

■ 注意事項

本記事の検証結果は環境や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。

■ 参考文献

- 「Optimizing Oracle Optimizer」趙東郁@EXEM, 2008
- 「データベースパフォーマンスアップの教科書」李華植, 2006
- Oracle Database Documentation Library
- http://support.oracle.co.jp/
- https://metalink.oracle.com/

本誌付録CD-ROMに「MaxGauge Version 3.1 日本語評価版」（日本エクセム）を収録しています。

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、トレーニングを始め、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、現在データベース監視／分析ツール「MaxGauge」の技術サポート、トラブル解析およびパフォーマンス改善コンサルティング、技術セミナーを行なっている。