Oracle 性能管理Tips

プロが教える現場ワザ、第3弾!

特別企画

索引&SQL編

索引とSQLに言及せずに性能改善作業を語ることはできないに等しい。本稿では、基礎知識として索引アクセスによる実行計画のパターンを紹介する。また、中上級者向けのテクニックとして、SQLテキストを変更できない場合のチューニング手法と、知っておくべき11gの推奨機能、自動化作業の事例として簡単に作れるスクリプトツールを紹介する。

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

同様のテーマ／タイトルで3回目[注1]の寄稿となるが、今回のTipsの領域も大きくは変わらない。初期化パラメータ、統計情報、実行計画、OWIなどの話題は最終的にオプティマイザの動作を理解することに帰結する。なぜなら、Oracleユーザーとしての未熟さから発生する大半のトラブルとともに、たまに発生するオプティマイザの間違った判断を正すのが、Oracleの性能管理業務に携わるエンジニアの宿命だからだ。本記事の内容もその認識から始まっている。

索引と実行計画

実行計画と関連して、よくある質問の1つが次のようなものだ。

> 対象のデータを取り出すためには、索引経由のアクセスが効果的なのか、それとも表のフルスキャンが効果的なのか。

とても素朴な疑問だが、単純化しすぎたので次の質問がより適切かと思う。

> 対象のデータを取り出すにあたり、索引を経由する場合はどのようなアクセス方法が効果的なのか、それとも表のフルスキャンが効果的なのか。

なぜ適切かと言えば、一言で索引スキャンと表現しても、実際にはさまざまなアクセスパターンがあるからだ。本セクションではよく見られる索引（Bツリー索引）経由のアクセスパターンを取り上げて、改めて基礎的な観点に基づいてそれぞれの特徴や活用場面を考えてみる。

INDEX RANGE SCAN

「索引を経由する」と言えばこのアクセスパターンをイメージするほど、最もよく使われるオペレーションである。INDEX RANGE SCANでは**図1**のようにデータを取り出す。

①「c1 >= 5」条件を満たすキーを検索するため索引のルートブロックを探索する（論理読取数が「1」増加する）
②索引のブランチブロックを探索する（論理読取数が「1」増加する）
③「c1 >= 5」条件を満たす最初のリーフブロックを訪問して条件に該当するキー「5、6」を取り出して戻す（論理読取数が「1」増加する）
④続きのキー「7」を取り出すため、次のリーフブロックを訪問してキー「7、8、9」を取り出して戻す（論理読取数が「1」増加する）
⑤ブロックの探索が終わるまで「④」の処理を続ける

このようにブロック単位で索引キーを読み取るので、索引全体をスキャンするとリーフブロック数と論理読取数はほぼ一致すると予想される。しかし実際の動作は少し異なる。LIST1からその理由を説明する。

検証①で「リーフブロック数=19」の表に対してINDEX RANGE SCANを行なったが、その論理読取数は686（>>19）となっている。これはクライアント（ここではSQL*Plus）側でデータを取り出す際に配列サイズに分けて複数回フェッチを行

注1：以前の2回は、2009年9月号特集3「エキスパートが明かすOracle性能改善Tips」と、2009年12月号特集3「Oracle性能遅延改善のための特選Tips集」。

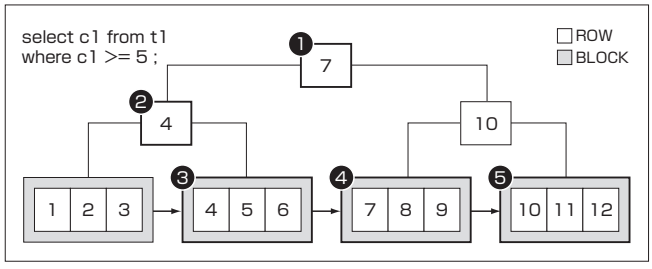


図1：INDEX RANGE SCAN

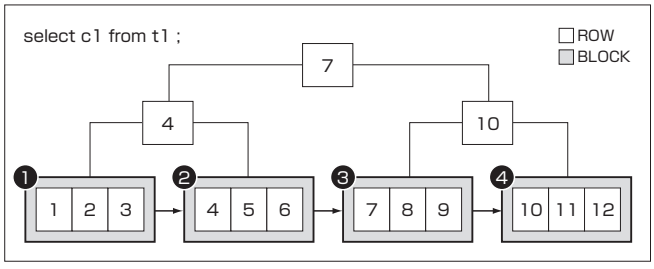
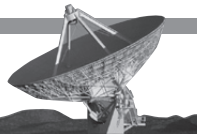


図2：INDEX FULL SCAN



LIST1：INDEX RANGE SCAN

```
SQL> create table t1 (c1 int, c2 int) ;
SQL> insert into t1 select level, level from dual connect by level <= 10000 ;
SQL> create index i1_t1 on t1 (c1) ;
→ dba_indexes.blevel:1、dba_indexes.leaf_blocks:19、dba_indexes.distinct_keys :10000

SQL> set arraysize 15
→ SQL*Plus のデフォルト配列サイズ

SQL> select /*+ gather_plan_statistics index (t1) */ c1 from t1 where c1 >= 5 ;
9996 行が選択されました。
-----------------------------------------------------------------------------------------
| Id  | Operation         | Name  | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------
|*  1 | INDEX RANGE SCAN | I1_T1 |      1 |   9997 |   9996 |00:00:00.03 |     686 |
-----------------------------------------------------------------------------------------
→ 検証① ルートブロック(1) + リーフブロック(19) + フェッチ回数(データ件数 / 配列サイズ =9996/15)
→ フェッチ回数ごとに追加で論理読取が発生

SQL> select /*+ gather_plan_statistics index (t1) */ count (*) from t1 where c1 >= 5 ;
1 行が選択されました。
-----------------------------------------------------------------------------------------
| Id  | Operation          | Name  | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------
|   1 | SORT AGGREGATE     |       |      1 |        |      1 |00:00:00.01 |      20 |
|*  2 | INDEX RANGE SCAN  | I1_T1 |      1 |   9997 |   9996 |00:00:00.03 |      20 |
-----------------------------------------------------------------------------------------
→ 検証② ルートブロック(1) + リーフブロック(19)
→ フェッチ回数1回

SQL> set arraysize 1000
SQL> select /*+ gather_plan_statistics index (t1) */ c1 from t1 where c1 >= 5 ;
-----------------------------------------------------------------------------------------
| Id  | Operation         | Name  | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------
|*  1 | INDEX RANGE SCAN | I1_T1 |      1 |   9997 |   9996 |00:00:00.04 |      30 |
-----------------------------------------------------------------------------------------
→ 検証③ ルートブロック(1) + リーフブロック(19) + フェッチ回数(データ件数 / 配列サイズ =9996/1000)
→ フェッチ回数ごとに追加で論理読取が発生

SQL> set arraysize 15
SQL> delete from t1 where c1 >= 1 ;
10000 行が削除されました。

SQL> select /*+ gather_plan_statistics index (t1) */ count (*) from t1 where c1 >= 5 ;
1 行が選択されました。
-----------------------------------------------------------------------------------------
| Id  | Operation          | Name  | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------
|   1 | SORT AGGREGATE     |       |      1 |        |      1 |00:00:00.01 |      20 |
|*  2 | INDEX RANGE SCAN  | I1_T1 |      1 |   9997 |      0 |00:00:00.01 |      20 |
-----------------------------------------------------------------------------------------
→ 検証④ ルートブロック(1) + リーフブロック(19)
→ 削除されてデータがないリーフブロックもアクセスした

SQL> alter index i1_t1 coalesce;
SQL> select /*+ gather_plan_statistics index (t1) */ count (*) from t1 where c1 >= 5 ;
レコードが選択されませんでした。
-----------------------------------------------------------------------------------------
| Id  | Operation          | Name  | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------
|   1 | SORT AGGREGATE     |       |      1 |        |      1 |00:00:00.01 |       2 |
|*  2 | INDEX RANGE SCAN  | I1_T1 |      1 |   9997 |      0 |00:00:00.01 |       2 |
-----------------------------------------------------------------------------------------
→ 検証⑤ 空きの領域がなくなって論理読取が最適化された
```

LIST2：INDEX FULL SCAN

```
SQL> create table t1(c1 int, c2 int, c3 char(100));
SQL> insert into t1 select level, level, 'a' from dual connect by level <= 1000000 ;
SQL> create index i1_t1 on t1(c1);

SQL> explain plan for select /*+ index(t1) */ c1 from t1 ;
-------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |   997K|  4872K|  3595   (2)| 00:00:44 |
|   1 | TABLE ACCESS FULL | T1   |   997K|  4872K|  3595   (2)| 00:00:44 |
-------------------------------------------------------------------------------
→ 検証① 表に対してフルスキャンでアクセス。なぜ?

SQL> alter table t1 modify c1 not null ;
SQL> explain plan for select /*+ index(t1) */ c1 from t1 ;
-------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |   997K|  4872K|  2265   (2)| 00:00:28 |
|   1 | INDEX FULL SCAN   | I1_T1|   997K|  4872K|  2265   (2)| 00:00:28 |
-------------------------------------------------------------------------------
→ 検証② 索引構成カラムは「NULL」を含まないことで、索引に対するフルスキャンでアクセス

SQL> explain plan for select /*+ index(t1) */ max(c1) from t1 ;
-----------------------------------------------------------------------------------------
| Id  | Operation                  | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |       |     1 |     5 |     3   (0)| 00:00:01 |
|   1 | SORT AGGREGATE             |       |     1 |     5 |            |          |
|   2 | INDEX FULL SCAN (MIN/MAX)  | I1_T1 |   997K|  4872K|     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
→ 検証③ 最大値 / 最小値を求める際に、「INDEX FULL SCAN」が活躍する

SQL> explain plan for select /*+ index(t1) */ min(c1), max(c1) from t1 ;
-------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |     5 |  2265   (2)| 00:00:28 |
|   1 | SORT AGGREGATE    |      |     1 |     5 |            |          |
|   2 | INDEX FULL SCAN   | I1_T1|   997K|  4872K|  2265   (2)| 00:00:28 |
-------------------------------------------------------------------------------
→ 検証④ 最大値と最小値を同時に求める際の「INDEX FULL SCAN」は逆効果

SQL> explain plan for select sum(min_c1), sum(max_c1) from (
  2             select /*+ index(t1) */ min(c1) min_c1, 0 max_c1 from t1 union all
  3             select /*+ index(t1) */ 0 min_c1, max(c1) max_c1 from t1) ;
-------------------------------------------------------------------------------------------
| Id  | Operation                     | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |       |     1 |    26 |     6   (0)| 00:00:01 |
|   1 | SORT AGGREGATE                |       |     1 |    26 |            |          |
|   2 | VIEW                          |       |     2 |    52 |     6   (0)| 00:00:01 |
|   3 | UNION-ALL                     |       |       |       |            |          |
|   4 | SORT AGGREGATE                |       |     1 |     5 |            |          |
|   5 | INDEX FULL SCAN (MIN/MAX)     | I1_T1 |   997K|  4872K|     3   (0)| 00:00:01 |
|   6 | SORT AGGREGATE                |       |     1 |     5 |            |          |
|   7 | INDEX FULL SCAN (MIN/MAX)     | I1_T1 |   997K|  4872K|     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
→ 検証⑤ 最大値と最小値を別々に抽出することで、「INDEX FULL SCAN」を適切に活用できる
```

なうので、その分追加で訪問するブロック数が増加したためだ。SQL*Plusのデフォルト配列サイズは15なので、ここでは「ルートブロック(1)+リーフブロック(19)+フェッチ回数(データ件数／配列サイズ=9996/15)」で論理読取数と一致する。

検証②で1件を取り出すSQLでは追加のフェッチは発生しないので、「ルートブロック(1)+リーフブロック(19)→ 20ブロック」が論理読取数となった。

検証③で配列サイズを1000に変更すると、追加フェッチが「データ件数／配列サイズ=9996/1000≒10)」回発生して論理読取数30に減少した。

このようにフェッチ回数によって作業量は変わることを認識しておこう。

また、検証④のようにデータを削除されても索引アクセスでは実際にリーフブロックを訪問するので、無駄なコストが発生する。このような場合は検証⑤のように空きの領域をなくして論理読取を最適化しておく必要がある。

INDEX FULL SCAN

INDEX FULL SCANはユニークなオペレーションで、ヒントで制御することができない。INDEX RANGE SCANが狭い範囲のデータ検索を目的としている反面、INDEX FULL SCANの目的(索引経由で広い範囲のデータ検索)は少し曖昧だが、特殊な状況では効率良く動作する。動作メカニズムは、最初(または最後)のリーフブロックから探索を開始して最後まで読み取ることを除いて、INDEX RANGE SCANとまったく同じだ(**図2**)。

では、**LIST2**よりINDEX FULL SCANになれる条件と、どのようなケースにおいて効果的かを確認する。

検証①のSQLは抽出項目が索引構成カラムに含まれるため索引経由の実行計画になりそうだが、TABLE ACCESS FULLとなった。なぜだろう。その原因はカラム定義にあった。索引はNULL値を含まないため、NULL値を含む可能性があるカラムに対する検索は索引経由ではできない。該当カラムに「NOT NULL」を指定すると検証②のようにオプティマイザは安心してINDEX FULL SCANを行なうようになった。

検証③のように最大値／最小値を求めるケースは、同オペレーションが活躍する場面となる。

ソートされている索引ブロックで必要なブロック（最後のリーフブロック）のみにアクセスして結果を取り出している。1つ注意してもらいたいのは、検証④のように最大値と最小値を同時に求める際のINDEX FULL SCANは逆に効率が悪くなるので、検証⑤のように最大値と最小値を別々に抽出することで最適化できる。

INDEX FAST FULL SCAN

INDEX FAST FULL SCANは表に対するフルスキャンの索引バージョンで、索引セグメントの最初から最高水位標（HWM）まで読み取りながら、リーフブロックのキーを取り出すオペレーションだ。索引キーをすべて読み取るオペレーションで最も効率が良いため、索引の再構築（REBUILD）などOracleの内部処理でも使われている（**LIST3**の検証①を参照）。

図3のように物理的な順番でマルチブロック読み取りを行なうため、ほかの索引オペレーションより性能が良い。さらに、パラレル処理がサポートされるため性能改善効果は期待できる。しかし、その反面結果セットのソートが保証されない弱点がある。よって、索引の全ブロックにアクセスが必要で、かつソートが不要な場合は、index_ffsヒントでINDEX FAST FULL SCANに導くことが最適となる（LIST3の検証②）。さらに性能改善が必要な場合は、parallel_indexヒントを追加してパラレルで実行することを検討すべきである（LIST3の検証③）。

LIST3：INDEX FAST FULL SCAN

```
SQL> create table t1(c1 int, c2 int, c3 char(100)) ;
SQL> insert into t1 select level, level, 'a' from dual connect by level <= 1000000 ;
SQL> create index i1_t1 on t1(c1) ;

SQL> explain plan for alter index i1_t1 rebuild;
-------------------------------------------------------------------------------
| Id  | Operation             | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | ALTER INDEX STATEMENT |       |  1000K|  4885K|  3595   (2)| 00:00:44 |
|   1 | INDEX BUILD NON UNIQUE | I1_T1 |       |       |            |          |
|   2 | SORT CREATE INDEX     |       |  1000K|  4885K|            |          |
|   3 | INDEX FAST FULL SCAN  | I1_T1 |       |       |            |          |
-------------------------------------------------------------------------------
→ 検証① Oracleの内部処理でも効率が良いため「INDEX FAST FULL SCAN」を活用

SQL> explain plan for select /*+ index(t1) */ c1 from t1 where c1 > 0 ;
-------------------------------------------------------------------------
| Id  | Operation        | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT |       |  1009K|  4930K|  2265   (2)| 00:00:28 |
|*  1 | INDEX RANGE SCAN | I1_T1 |  1009K|  4930K|  2265   (2)| 00:00:28 |
-------------------------------------------------------------------------

SQL> explain plan for select /*+ index_ffs(t1) */ c1 from t1 where c1 > 0 ;
-------------------------------------------------------------------------------
| Id  | Operation            | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |       |  1000K|  4885K|   521   (7)| 00:00:07 |
|*  1 | INDEX FAST FULL SCAN | I1_T1 |  1000K|  4885K|   521   (7)| 00:00:07 |
-------------------------------------------------------------------------------
→ 検証② 索引の全ブロックにアクセスが必要で、ソートの必要がない場合は「INDEX FAST FULL SCAN」が最適

SQL> explain plan for select /*+ index_ffs(t1) parallel_index(t1 i1_t1) */ c1 from t1 where c1 > 0 ;
-------------------------------------------------------------------------------------------------------
| Id  | Operation               | Name     | Rows  | Bytes | Cost (%CPU)| Time     |    TQ  |IN-OUT| PQ Distrib |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |          |  1000K|  4885K|   521   (7)| 00:00:07 |        |      |            |
|   1 | PX COORDINATOR          |          |       |       |            |          |        |      |            |
|   2 | PX SEND QC (RANDOM)     | :TQ10000 |  1000K|  4885K|   521   (7)| 00:00:07 |  Q1,00 | P->S | QC (RAND)  |
|   3 | PX BLOCK ITERATOR       |          |  1000K|  4885K|   521   (7)| 00:00:07 |  Q1,00 | PCWC |            |
|*  4 | INDEX FAST FULL SCAN    | I1_T1    |  1000K|  4885K|   521   (7)| 00:00:07 |  Q1,00 | PCWP |            |
-------------------------------------------------------------------------------------------------------
→ 検証③ さらに性能改善のため、索引データ処理をパラレルで実行可能
```

INDEX SKIP SCAN

Oracle8iまでの環境でWHERE句のカラムが索引を使えるための基本要件は、索引を構成する先行カラムがWHERE句で使用されることだ。9i以降はINDEX SKIP SCANオペレーションの登場で「先行カラム」という制限が克服された。

図4の条件の「c2」は索引の先行カラムではないが、リーフブロックをスキップしながら探索することになった。そのおかげで索引経由でデータ参照できる場面が多くなったが、気をつけないといけない場面もある。

LIST4の検証①で検索範囲による作業量の変化を観察した結果、検索範囲に比例して作業量およびコストが増える傾向が確認された。また検証②のように先行カラムの値ごとにSQLを分割してINDEX RANGE SCANオペレーションに導くと性能改善効果が見られた。この点から検索範囲が広いほど同オペレーションの効率が悪くなることが分かる。なお、検証③のように先行カラムの値の種類（DISTINCT）を増やすと、同じ検索でも急激に作業量が増えることが確認された。

上記の結果から「①INDEX SKIP SCANは検索範囲が狭い、②先行カラムの値の種類

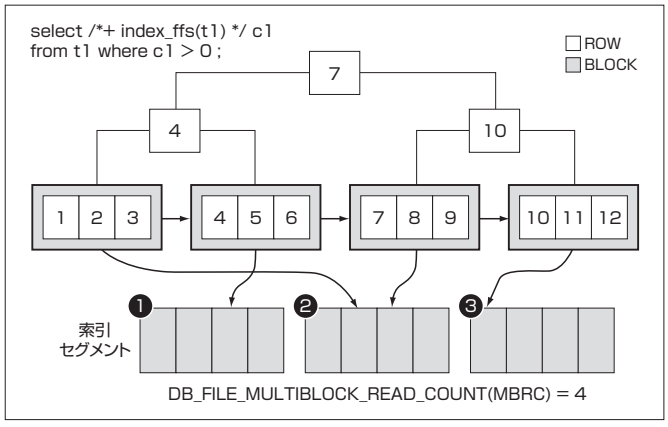


図3：INDEX FAST FULL SCAN

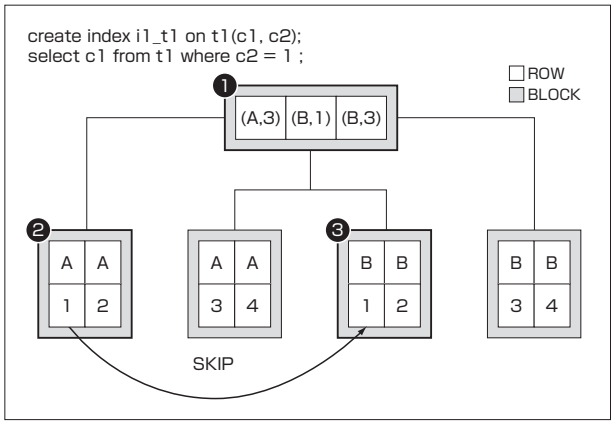


図4：INDEX SKIP SCAN

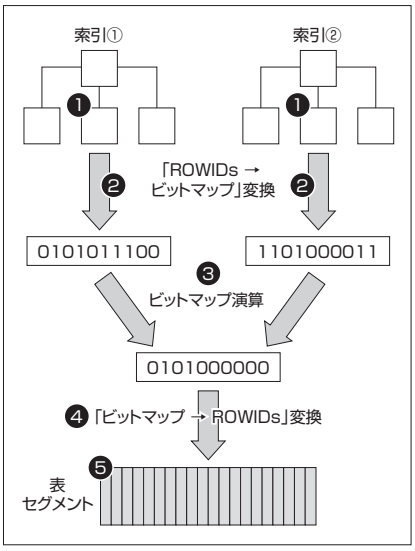


図5：Bツリー索引のビットマップ変換

（DISTINCT）が多くない」という状況でより効果的だということが考えられる。

Bツリー索引のビットマップ変換

カラムc1、c2が条件指定されていて、両方のカラムにはそれぞれ索引が付いている場合、両方の索引を同時に使うことができれば最も効果的な処理になる。このようなオペレーションを可能にしたのがビットマップ索引だが、激しいDMLブロッキング問題[注2]によりOLTP環境では事実上使わないようにしている。そこでDMLブロッキング問題を解消してビットマップ索引のメリットをそのまま活かしたのがBツリー索引のビットマップ変換のオペレーションだ。このオペレーションは次のように動作する（図5）。

①各索引から索引キーとROWIDを読み取る
②ROWIDをビットマップに変換する
③各索引の「②」の結果からビットマップ演算を行なう
④「③」の結果をROWIDに変換する
⑤表ブロックから対象のデータを取り出す

このように複数の索引を同時にアクセスできるメリットがある反面、ビットマップ演算にメモリやCPUが使われるため追加で発生する負荷も考慮する必要がある。

また、LIST5から効果的な場面を確認してみよう。検証①では2つの索引を経由してアクセスする表ブロックがほとんど異なるため、索引の組み合わせによるデータの絞り込みが良く、表ブロックへのアクセスが最小化された。逆に検証②のように両方の索引を経由してアクセスする表ブロックを同じくした場合は、個別索引のデータの選択度（絞り込み率1000件）は同じものの、組み合わせ

LIST4：INDEX SKIP SCAN

```
SQL> create table t1(c1 varchar2(2), c2 int, c3 char(100));
SQL> insert into t1 select 'A', level, 'a' from dual connect by level <= 5000 ;
SQL> insert into t1 select 'B', level, 'b' from dual connect by level <= 5000 ;
SQL> create index i1_t1 on t1(c1, c2);

SQL> select /*+ gather_plan_statistics index_ss(t1) */ * from t1 where [ 検索条件① ] ;
-----------------------------------------
| Id  | Operation                  | Name  |
-----------------------------------------
|   1 |  TABLE ACCESS BY INDEX ROWID| T1    |
|*  2 |   INDEX SKIP SCAN          | I1_T1 |
-----------------------------------------
```

検索条件①	コスト	論理読取	データ件数
c2 between 1 and 1	4	**8**	2
c2 between 1 and 10	22	**12**	20
c2 between 1 and 100	205	**51**	200
c2 between 1 and 1000	2,036	**441**	2,000

→ **検証①** 検索範囲に比例して作業量およびコストが増える

```
SQL> select /*+ gather_plan_statistics index(t1) */ * from t1 where c1 = 'A' and c2 between 1 and 1000
  2  union all
  3  select /*+ gather_plan_statistics index(t1) */ * from t1 where c1 = 'B' and c2 between 1 and 1000 ;
-----------------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name  | Starts | E-Rows | Cost (%CPU)| A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------
|   1 |  UNION-ALL                   |       |      1 |        |            |   2000 |00:00:00.02 |     306 |
|   2 |  TABLE ACCESS BY INDEX ROWID | T1    |      1 |   1000 |     20  (0)|   1000 |00:00:00.01 |     153 |
|*  3 |   INDEX RANGE SCAN           | I1_T1 |      1 |   1000 |      4  (0)|   1000 |00:00:00.01 |      71 |
|   4 |  TABLE ACCESS BY INDEX ROWID | T1    |      1 |   1000 |     20  (0)|   1000 |00:00:00.01 |     153 |
|*  5 |   INDEX RANGE SCAN           | I1_T1 |      1 |   1000 |      4  (0)|   1000 |00:00:00.01 |      71 |
-----------------------------------------------------------------------------------------------------------
```

→ **検証②**「INDEX RANGE SCAN」に導くことで性能改善効果を出せる

```
SQL> truncate table t1 ;
SQL> insert into t1 select s.value, rownum, 'c'
  2  from   (select to_char(mod(level,100), 'FM00') as value from dual connect by level <= 100) s,
  3         (select level from dual connect by level <= 50) t ;
SQL> insert into t1 select s.value, rownum, 'c'
  2  from   (select to_char(mod(level,100), 'FM00') as value from dual connect by level <= 100) s,
  3         (select level from dual connect by level <= 50) t ;
```

実行SQL	アクセス方法	コスト	論理読取	データ件数
select /*+ index_ss(t1) */ * from t1 where c2 between 1 and 100 ;	INDEX SKIP SCAN	302	**263**	100
select /*+ index(t1) */ * from t1 where c1 between '00' and '99' and c2 between 1 and 100 ;	INDEX RANGE SCAN	237	**249**	100

→ **検証③** 先行カラムの「Distinct Count」を高める（2 → 100）と作業量が「51 → 263」に増加した

LIST5：Bツリー索引のビットマップ変換

```
SQL> create table t1(c1 int, c2 int, c3 char(100));
SQL> create index t1_i1 on t1(c1);
SQL> create index t1_i2 on t1(c2);
SQL> insert into t1 select mod(level,10)+1, 10-mod(level,10)-1, 'a' from dual connect by level <= 10000 ;
→「t1_i1」索引と「t1_i2」索引を経由してアクセスする表ブロックはほとんど異なる

SQL> select /*+ gather_plan_statistics index_combine(t1 t1(c1) t1(c2)) */ c3 from t1 where c1 = 1 and c2 = 1 ;
-----------------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name  | Starts | E-Rows | Cost (%CPU)| A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------
|   1 |  TABLE ACCESS BY INDEX ROWID | T1    |      1 |    100 |     24  (5)|      0 |00:00:00.01 |       7 |
|   2 |  BITMAP CONVERSION TO ROWIDS |       |      1 |        |            |      0 |00:00:00.01 |       7 |
|   3 |  BITMAP AND                  |       |      1 |        |            |      0 |00:00:00.01 |       7 |
|   4 |  BITMAP CONVERSION FROM ROWIDS |     |      1 |        |            |      1 |00:00:00.01 |       3 |
|*  5 |   INDEX RANGE SCAN           | T1_I1 |      1 |        |      2  (0)|   1000 |00:00:00.01 |       3 |
|   6 |  BITMAP CONVERSION FROM ROWIDS |     |      1 |        |            |      1 |00:00:00.01 |       4 |
|*  7 |   INDEX RANGE SCAN           | T1_I2 |      1 |        |      2  (0)|   1000 |00:00:00.01 |       4 |
-----------------------------------------------------------------------------------------------------------
```

→ **検証①**「t1_i1」&「t1_i2」の索引組み合わせによるデータの絞り込みが良く、表ブロックへのアクセスが最小化された（0 ブロック）

```
SQL> truncate table t1 ;
SQL> insert into t1 select mod(level,10)+1, mod(level,10)+1, 'a' from dual connect by level <= 10000 ;
→「t1_i1」索引と「t1_i2」索引を経由してアクセスする表ブロックは同じ

SQL> select /*+ gather_plan_statistics index_combine(t1 t1(c1) t1(c2)) */ c3 from t1 where c1 = 1 and c2 = 1 ;
-----------------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name  | Starts | E-Rows | Cost (%CPU)| A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------
|   1 |  TABLE ACCESS BY INDEX ROWID | T1    |      1 |    100 |     26  (4)|   1000 |00:00:00.01 |     222 |
|   2 |  BITMAP CONVERSION TO ROWIDS |       |      1 |        |            |   1000 |00:00:00.01 |       8 |
|   3 |  BITMAP AND                  |       |      1 |        |            |      1 |00:00:00.01 |       8 |
|   4 |  BITMAP CONVERSION FROM ROWIDS |     |      1 |        |            |      1 |00:00:00.01 |       4 |
|*  5 |   INDEX RANGE SCAN           | T1_I1 |      1 |        |      3  (0)|   1000 |00:00:00.01 |       4 |
|   6 |  BITMAP CONVERSION FROM ROWIDS |     |      1 |        |            |      1 |00:00:00.01 |       4 |
|*  7 |   INDEX RANGE SCAN           | T1_I2 |      1 |        |      3  (0)|   1000 |00:00:00.01 |       4 |
-----------------------------------------------------------------------------------------------------------
```

→ **検証②** 個別索引のデータの選択度（絞り込み率、1000 件）は同じものの、組み合わせの効率が悪く、表ブロックへのアクセスが多い（216 ブロック =220-8）

注2：ビットマップの構成カラムにデータの変更が発生する場合、該当レコードだけではなくブロック単位でロックがかかるという現象。

の絞り込み率が悪く、表ブロックへのアクセスが多くなっている。この結果から同オペレーションが常に有利なわけではなく、索引の組み合わせの絞り込みが良いほど効果的ということが分かる。

SQLチューニング

ERPなどのパッケージ製品を使う環境では、ヒントを含めSQLを変更できない場合が多い。このような状況でSQLチューニングができるだろうか。もちろん、できる。逆にSQLの見直し以外のチューニング手法がすべてそこに該当する。具体的に並べてみると、次のような方法がある。

①索引の見直し（追加、削除、カラム構成変更）
②マテリアライズドビューを作成
③パーティション化
④プランスタビリティ（ストアドアウトライン）を適用（8.1.5.0以降）
⑤SQLプロファイルを適用（10g以降）
⑥SQL計画ベースラインを適用（11g以降）
⑦統計情報を変更：統計情報の再収集、サンプルサイズの見直し、ヒストグラム統計の作成/削除、DBMS_STATS.SET_XXX_STATSなど
⑧初期化パラメータの調整：オプティマイザの動作に影響を与える
⑨運用調整：実行時間帯、対象データ件数など
⑩ハードウェアの増強：CPU、メモリなど

最もよく使われるのが①「索引の見直し」だということは言うまでもない。場面によるが通常、筆者はこの中で作業の効果（↑）、影響範囲（狭）、リスク（↓）、コスト（↓）の面から「①、②、③＞④、⑤、⑥＞⑦＞⑨＞⑧＞⑩」の順で検討することを推奨している。

本セクションでは、リスクが少なく効果的でありながらそれほど使われていない⑤と⑥の手法について見ていきたい。

LIST6：SQLプロファイルを活用したSQLチューニング

```
SQL> create table t1(c1 int, c2 int);
SQL> insert into t1 select level, level from dual connect by level <= 1000 ;
SQL> create index t1_i1 on t1(c1);
SQL> create index t1_i2 on t1(c2);

SQL> explain plan for select /*+ full(t1) */ * from t1 where c1 = 1 or c2 = 2 ;
-------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     2 |    14 |     3   (0)| 00:00:01 |
|*  1 | TABLE ACCESS FULL  | T1   |     2 |    14 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------
→ 検証① 効率が良くない実行計画を確認する：SQL-A

SQL> explain plan for select /*+ use_concat */ * from t1 where c1 = 1 or c2 = 2 ;
-----------------------------------------------------------------------------------
| 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_I2 |     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_I1 |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------
→ 検証② 効率が良い実行計画を確認する：SQL-B

SQL> select * from table(dbms_xplan.display(null, null, '+outline'));
Outline Data
-------------
  /*+
      BEGIN_OUTLINE_DATA
      INDEX(@"SEL$1_2" "T1"@"SEL$1_2" ("T1"."C1"))
      INDEX(@"SEL$1_1" "T1"@"SEL$1" ("T1"."C2"))
      OUTLINE(@"SEL$1")
      OUTLINE_LEAF(@"SEL$1_2")
      USE_CONCAT(@"SEL$1" 8)
            …
      END_OUTLINE_DATA
  */
→ 検証③ SQL-B の実行計画のアウトラインデータ（ヒント集）を確認する

SQL> begin
  2    dbms_sqltune.import_sql_profile(
3             name => 'index_prof',
4             sql_text => 'select /*+ full(t1) */ * from t1 where c1 = 1 or c2 = 2',
5             profile => sqlprof_attr('USE_CONCAT(@"SEL$1" 8)',
6                                     'INDEX(@"SEL$1_2" "T1"@"SEL$1_2" ("T1"."C1"))',
7                                     'INDEX(@"SEL$1_1" "T1"@"SEL$1" ("T1"."C2"))')
  8    );
  9  end;
 10  /
→ 検証④ SQL-A に対して SQL-B のヒントに基づく SQL プロファイルを適用する

SQL> explain plan for select /*+ full(t1) */ * from t1 where c1 = 1 or c2 = 2 ;
-----------------------------------------------------------------------------------
| 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_I2 |     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_I1 |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------
...
Note
-----
   - SQL profile "index_prof" used for this statement
→ 検証⑤ SQL-A の実行結果から SQL プロファイルの適用を確認する
```

SQLプロファイル

SQLプロファイルとは簡単に言うと「クエリの実行計画を制御するためのヒント[注3]の集まり」だ。本来は自動SQLチューニングを目的に導入された機能だが、ここでは手動で制御してSQLチューニングにつなげる例を紹介する。

LIST6では、同じく特定表のデータを取り出すSQLを2つ作成して、一方の「SQL-A」の効率は悪く、他方「SQL-B」の効率は良いと想定している。以下、作業手順となる。

検証① 効率が良くない実行計画を確認する：SQL-A → TABLE ACCESS FULL
検証② 効率が良い実行計画を確認する：SQL-B → INDEX RANGE SCAN
検証③ SQL-Bの実行計画のアウトラインデータ（ヒント集）から実行パスのキーとなるヒントを確認する
検証④ SQL-Aに対してSQL-Bのヒントに基づくSQLプロファイルを適用する
検証⑤ SQL-Aの実行結果からSQLプロファイルが適用されていることを確認する（「V$SQL.SQL_PROFILE」からも同様の結果が確認できる）

どうだろうか。改善後の実行計画と、そのポイ

注3：オプティマイザに補助の統計を提供するヒント（例：FIRST_ROWS）。

ントが分かれば適用そのものは意外と簡単に行なえる。

SQL計画の管理

次に、SQL計画の管理機能を活用して、SQLテキストの修正を行なわずSQL性能を改善する手法を確認する。この機能は、SQLの実行計画の履歴を記録および評価し、より効率が良いと判断される実行計画を採用する仕組みをとっていて、SQLの安定稼動、すなわちSQLの性能劣化を事前に防ぐ目的で導入された。

ここでは、計画ベースラインに格納されている実行計画に、効率の良い既知の実行計画をロードすることでSQLの改善を図ってみる。

図6でそのイメージを見てみよう。「OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES=TRUE」設定後にSQL-Aを2回実行すると、計画ベースラインに実行計画が自動ロードされる。そこでキャッシュされているSQL-Bの実行計画を手動でロードすることで、SQL-AはSQL-Bの実行計画で動くようになる。

同様の作業手順を**LIST7**の検証サンプルから確認しよう。

検証① 改善対象のSQLの実行計画を確認する：SQL-A → TABLE ACCESS FULL
検証② SQL-Aを繰り返し実行して、SQL計画ベースラインに追加する
検証③ SQL-A計画ベースライン（SQL_HANDLE、PLAN_NAME）を確認する
検証④ 改善後のSQL情報（SQL_ID、planh ash value）を確認する：SQL-B
検証⑤ SQL-Bの実行計画を共有プールからSQL計画ベースラインに手動でロードする
検証⑥ 既存のSQL-Aの計画ベースラインを削除する
検証⑦ 改善後のSQL実行計画と適用された計画ベースラインを確認する（「V$SQL.SQL_PLAN_BASELINE」からも同様の結果が確認できる）

SQLプロファイルより確認すべき情報が少し多いところもあるが、複数の手段を持つことは各場面での選択肢が増えることなのでより柔軟な

LIST7：SQL計画の管理を活用したSQLチューニング

```
SQL> explain plan for select /*+ full(t1) */ * from t1 where c1 = 1 or c2 = 2 ;
SQL> select * from table(dbms_xplan.display);
-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     2 |    14 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL | T1   |     2 |    14 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
→ 検証① 改善対象の SQL の実行計画を確認：SQL-A

SQL> alter session set optimizer_capture_sql_plan_baselines = true ;
SQL> select /*+ full(t1) */ * from t1 where c1 = 1 or c2 = 2 ;
SQL> select /*+ full(t1) */ * from t1 where c1 = 1 or c2 = 2 ;
SQL> alter session set optimizer_capture_sql_plan_baselines = false ;
→ 検証② 改善対象の SQL を繰り返し実行して、SQL 計画ベースラインに追加する

SQL> select sql_handle, plan_name from dba_sql_plan_baselines where sql_text like 'select /*+ full(t1) */%' ;
SQL_HANDLE                                                    PLAN_NAME
------------------------------------------------------------ ------------------------------------------------
SYS_SQL_a76bb82a57c2f683                                      SYS_SQL_PLAN_57c2f683dbd90e8e
→ 検証③ 改善対象の SQL 計画ベースライン（SQL_HANDLE、PLAN_NAME）を確認

SQL> select /*+ use_concat */ * from t1 where c1 = 1 or c2 = 2 ;
SQL> select * from table(dbms_xplan.display_cursor);
SQL_ID  5k7qdjv52u110, child number 0
Plan hash value: 3562366018
-------------------------------------------------------------------------------------
| Id  | Operation                     | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |       |       |       |     4 (100)|          |
|   1 |  CONCATENATION                |       |       |       |            |          |
|   2 |   TABLE ACCESS BY INDEX ROWID | T1    |     1 |     7 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN           | T1_I2 |     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_I1 |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------
→ 検証④ 改善後の SQL 情報（SQL_ID 、PLAN HASH VALUE）を確認：SQL-B

SQL> variable r_num number ;
SQL> exec :r_num := dbms_spm.load_plans_from_cursor_cache ( -
>                   sql_id          => '5k7qdjv52u110', -
>                   plan_hash_value => 3562366018,      -
>                   sql_handle      => 'SYS_SQL_a76bb82a57c2f683' -
>                 ) ;
→ 検証⑤ SQL-B の実行計画を SQL 計画ベースラインにロードする

SQL> exec :r_num := dbms_spm.drop_sql_plan_baseline ( -
>                   sql_handle      => 'SYS_SQL_a76bb82a57c2f683', -
>                   plan_name       => 'SYS_SQL_PLAN_57c2f683dbd90e8e' -
>                 ) ;
→ 検証⑥ 既存の SQL-A の計画ベースラインを削除する

SQL> explain plan for select /*+ full(t1) */ * from t1 where c1 = 1 or c2 = 2 ;
SQL> 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_I2 |     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_I1 |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------
Note
-----
   - SQL plan baseline "SYS_SQL_PLAN_57c2f683e782b2e4" used for this statement
→ 検証⑦ 改善後の SQL 実行計画と適用された計画ベースラインを確認する
```

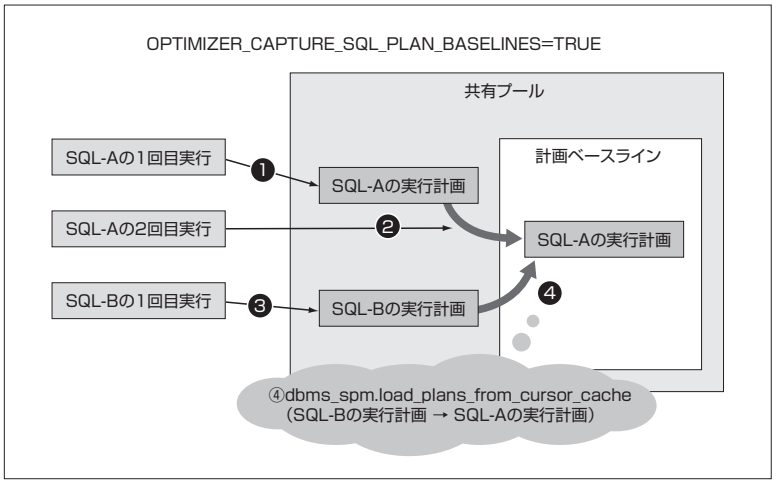


図6：SQL計画の管理を活用したSQLチューニング

対策ができることは間違いないだろう。

結果キャッシュ

Oracleのバージョンアップの歴史は、メモリ管理の自動化とともにキャッシュ機能を強化してきた歴史でもあった。11gもまた「結果キャッシュ」という注目すべき機能を備えている。この機能は繰り返し実行される参照処理の応答時間の改善を主な目的としているので、当該場面での性能遅延で困っているという方はぜひ活用を検討したいところだ。

主な内部動作は次のとおりだ(**図7**)。

①同じ結果セットを求める同一SQLが複数実行される
②結果キャッシュで対象のオブジェクトを検索するが、初回実行ではキャッシュされていないため、物理読取と論理読取を行なう
③処理の結果を結果キャッシュに保存(キャッシュ)する
④結果キャッシュから結果を取り出す。2回目以降は結果セットがキャッシュされているので、物理読取および論理読取が発生しない → 応答時間が早くなる

LIST8:結果キャッシュの性能改善効果

```
SQL> create table t1(c1 int, c2 char(100));
SQL> insert into t1 select level, rpad('a',mod(level,100),'b') from dual connect by level <= 100000 ;
SQL> create index t1_i1 on t1(c1);

SQL> alter system set result_cache_max_size = 15M ;
SQL> select /*+ gather_plan_statistics result_cache */ * from t1 ;
-----------------------------------------------------------------------------------------------------------------------
| Id  | Operation          | Name                       | Starts | E-Rows | Cost (%CPU)| A-Rows |   A-Time    | Buffers | Reads  |
-----------------------------------------------------------------------------------------------------------------------
|   1 |  RESULT CACHE      | 92wub75s24c3agd643x31gkt5n |      1 |        |            |   100K|00:00:00.64 |    8120 |   1573 |
|   2 |  TABLE ACCESS FULL | T1                         |      1 |   100K|   444   (1)|   100K|00:00:00.24 |    8120 |   1573 |
-----------------------------------------------------------------------------------------------------------------------
→ 検証① 1回目の実行で物理読取(Reads=1573 ブロック)、論理読取(Buffers=8120 ブロック)が発生した

SQL> select name, status, scan_count, row_count from v$result_cache_objects where type = 'Result' ;
NAME                                                                STATUS              SCAN_COUNT  ROW_COUNT
------------------------------------------------------------------- --------- ---------- ----------
select /*+ gather_plan_statistics result_cache */ * from t1         Published                    0     100000
→ クエリの結果が結果キャッシュに保存され、結果を使用可能(STATUS = Published)となっている

SQL> select * from v$sgastat where lower(name) like '%result%' ;
POOL                    NAME                                            BYTES
----------------------- ---------------------------------------------- ----------
shared pool             Result Cache: State Objs                          2852
shared pool             Result Cache                                   11022128
shared pool             Result Cache: Memory Mgr                           128
shared pool             Result Cache: Bloom Fltr                          2048
shared pool             Result Cache: Cache Mgr                            112

SQL> select /*+ gather_plan_statistics result_cache */ * from t1 ;
-----------------------------------------------------------------------------------------------------------
| Id  | Operation          | Name                       | Starts | E-Rows | Cost (%CPU)| A-Rows |   A-Time    |
-----------------------------------------------------------------------------------------------------------
|   1 |  RESULT CACHE      | 92wub75s24c3agd643x31gkt5n |      1 |        |            |   100K|00:00:00.10 |
|   2 |  TABLE ACCESS FULL | T1                         |      0 |   100K|   444   (1)|      0 |00:00:00.01 |
-----------------------------------------------------------------------------------------------------------
→ 検証② 2回目の実行で、物理読取、論理読取が発生しなくなった(Reads=0、Buffers=0)

SQL> select name, status, scan_count, row_count from v$result_cache_objects where type = 'Result' ;
NAME                                                                STATUS              SCAN_COUNT  ROW_COUNT
------------------------------------------------------------------- --------- ---------- ----------
select /*+ gather_plan_statistics result_cache */ * from t1         Published                    1     100000
→ 結果キャッシュが使用された(SCAN_COUNT=1)
```

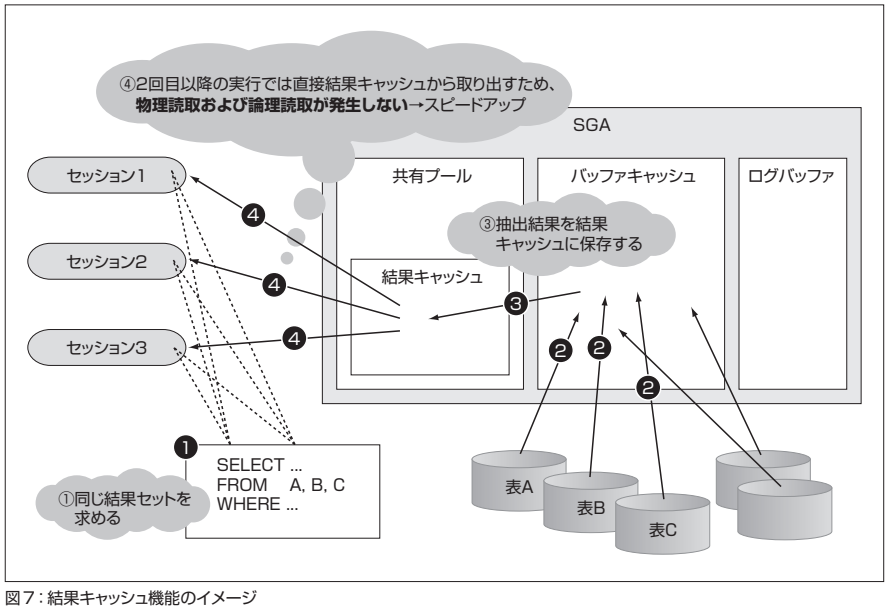


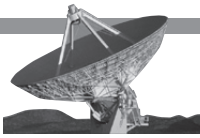
図7:結果キャッシュ機能のイメージ

その効果を**LIST8**で確認しよう。ある表の全件データを取り出している処理を行なうと、初回実行時(検証①)に1573ブロックの物理読取、8120ブロックの論理読取が発生したが、2回目の実行(検証②)では物理読取、論理読取がすべて発生しなくなった。実行計画から、両方ともRESULT CACHEオペレーションを行なっていることが分かる。そのおかげで、実行時間(A-Time)でも大幅な改善があった。内部ビュー(V$RESULT_CACHE_OBJECTS)から、使用可能(STATUS=Published)状態で結果セットがキャッシュされていることが確認できる。2回目の処理では結果キャッシュが使われたことも確認できる(SCAN_COUNT=1)。

同リストでは「RESULT_CACHE」ヒントを利用して特定クエリに適用しているが、セッションあるいはシステムレベルで「RESULT_CACHE_MODE=FORCE」パラメータを設定することで同様の結果が得られる。

ただし、結果キャッシュオブジェクトを共有できるクエリの書き方には注意が必要だ。**LIST9**で、キャッシュ済みのSQLテキストに変更を加えて共有できるかどうかを確認した。最初の大文字化した検証②で共有されたことが分かる。

引き続き、全角スペースの追加(検証③)、実行計画に影響を与えないヒント文字列の追加(検証④)でも共有が確認された。その半面、選択カラムに表の名前を付けた場合(検証⑤)は、同じ結果セットを返すと予測できるのにもかかわらず既存の結果キャッシュオブジェクトを使えず、新しいオブジェクトを生成した。また実行計画に影響するヒントが加わったSQL(検証⑥)も共有できないことが分かる。



それでは、バインドSQLのほうはどうだろうか。予測どおり、バインド変数の値が同じ場合はきちんと共有され、バインド変数の値が異なる場合は新しい結果キャッシュオブジェクトを生成し、共有することはない(LIST10)。

上記の結果より、特定SQLが結果キャッシュオブジェクトを共有できるかどうかは予想し難い面もあるが、SQLテキストおよび変数値、実行計画が同じ場合は共有できると言えるだろう。例えば、SQLテキストが同じでも初期化パラメータが異なるため実行計画が変わる場合は共有できないので改めて認識しておこう。

結果キャッシュサイズ設定などの注意点

この機能の適用において、もう1つ気をつけてもらいたいのが結果キャッシュサイズ「RESULT_CACHE_MAX_SIZE」の設定に関連するものだ。結果キャッシュはバッファキャッシュと同じく、空き領域がなくなった場合は古いオブジェクトからキャッシュアウトして新しいオブジェクトを保存する仕組みをとっている反面、結果セットのサイズがキャッシュのサイズを超えると結果セット全体がキャッシュされない点では異なる動作をする。

LIST11でその動作を具体的に確認する。事前にセグメントサイズが13MBの表を2つ作成しておいた。10MB(<13MB)で結果キャッシュを設定した場合(検証①)、クエリの結果(T1)を保存できず、結果は使用できなくなってしまう(INVALID)。15MB(>13MB)で設定した場合(検証②)はクエリの結果(T1)を保存できて、使用可能(PUBLISHED)となった。

次に同じ設定(15M<13M*2)で「T2」表に対するクエリを行なった結果(検証③)、「T1」「T2」両方の抽出結果を保存できず、古いオブジェクト(T1)はキャッシュアウトされて直近の結果(T2)だけがロードされる。続いて、30MB(>13M*2)で設定した場合(検証④)は、両方のクエリ結果をロードできることが分かる。

上記の結果から、結果キャッシュのサイズを設定する際にはキャッシュ対象のクエリの結果セットのサイズと変数値の種類の数を考慮しないといけないことが分かる。

このほかに、次の事項も覚えておいてもらいたい。

LIST9:結果キャッシュを利用できるSQL

```
SQL> exec dbms_result_cache.flush ;
SQL> alter system set result_cache_max_size = 15M ;

SQL> select /*+ gather_plan_statistics result_cache */ * from t1 ;
SQL> select name, status, scan_count, row_count from v$result_cache_objects where type = 'Result' ;
NAME                                                                             STATUS              SCAN_COUNT  ROW_COUNT
-------------------------------------------------------------------------------- ------------------- ---------- ----------
select /*+ gather_plan_statistics result_cache */ * from t1                      Published                    0     100000
→ 検証① 1回目の実行の結果がキャッシュされている

SQL> SELECT /*+ GATHER_PLAN_STATISTICS RESULT_CACHE */ * FROM T1 ;
SQL> select name, status, scan_count, row_count from v$result_cache_objects where type = 'Result' ;
NAME                                                                             STATUS              SCAN_COUNT  ROW_COUNT
-------------------------------------------------------------------------------- ------------------- ---------- ----------
select /*+ gather_plan_statistics result_cache */ * from t1                      Published                    1     100000
→ 検証② すべて大文字化されたSQLも結果キャッシュを使った(SCAN_COUNT=1)

SQL> select /*+ gather_plan_statistics result_cache */ *   from t1 ;
SQL> select name, status, scan_count, row_count from v$result_cache_objects where type = 'Result' ;
NAME                                                                             STATUS              SCAN_COUNT  ROW_COUNT
-------------------------------------------------------------------------------- ------------------- ---------- ----------
select /*+ gather_plan_statistics result_cache */ * from t1                      Published                    2     100000
→ 検証③ 全角のスペース(*　from)が加わったSQLも結果キャッシュを使った(SCAN_COUNT=2)

SQL> select /*+ gather_plan_statistics result_cache [this is comment] */ * from t1 ;
SQL> select name, status, scan_count, row_count from v$result_cache_objects where type = 'Result' ;
NAME                                                                             STATUS              SCAN_COUNT  ROW_COUNT
-------------------------------------------------------------------------------- ------------------- ---------- ----------
select /*+ gather_plan_statistics result_cache */ * from t1                      Published                    3     100000
→ 検証④ 実行計画に影響を与えないヒント文字列[this is comment]が加わったSQLも結果キャッシュを使った(SCAN_COUNT=3)

SQL> select /*+ gather_plan_statistics result_cache */ t1.* from t1 ;
-------------------------------------------------------------------------------------------------------------------------
| Id  | Operation          | Name                       | Starts | E-Rows | Cost (%CPU)| A-Rows |   A-Time   | Buffers | Reads  |
-------------------------------------------------------------------------------------------------------------------------
|   1 |  RESULT CACHE      | 0m5qvbavmf1020xpsxkjmlsarf |      1 |        |            |   100K|00:00:00.63 |    8120 |   1568 |
|   2 |  TABLE ACCESS FULL | T1                         |      1 |   100K|   444   (1)|   100K|00:00:00.23 |    8120 |   1568 |
-------------------------------------------------------------------------------------------------------------------------
SQL> select name, status, scan_count, row_count from v$result_cache_objects where type = 'Result' ;
NAME                                                                             STATUS              SCAN_COUNT  ROW_COUNT
-------------------------------------------------------------------------------- ------------------- ---------- ----------
select /*+ gather_plan_statistics result_cache */ t1.* from t1                   Published                    0     100000
→ 検証⑤ 本文が変更されたSQLは結果キャッシュを使えなく新しいオブジェクトを生成した(SCAN_COUNT=0、0m5qvbavmf1020xpsxkjmlsarf)

SQL> select /*+ result_cache gather_plan_statistics full(t1) */ * from t1 where c1 > 0 ;
-------------------------------------------------------------------------------------------------------------------------
| Id  | Operation          | Name                       | Starts | E-Rows | Cost (%CPU)| A-Rows |   A-Time   | Buffers | Reads  |
-------------------------------------------------------------------------------------------------------------------------
|   1 |  RESULT CACHE      | 8g30t47jdzgy6a7dbvydt2rmbb |      1 |        |            |   100K|00:00:00.60 |    8120 |   1568 |
|*  2 |  TABLE ACCESS FULL | T1                         |      1 |   100K|   445   (1)|   100K|00:00:00.20 |    8120 |   1568 |
-------------------------------------------------------------------------------------------------------------------------
SQL> select name, status, scan_count, row_count from v$result_cache_objects where type = 'Result' ;
NAME                                                                             STATUS              SCAN_COUNT  ROW_COUNT
-------------------------------------------------------------------------------- ------------------- ---------- ----------
select /*+ result_cache gather_plan_statistics full(t1) */ * from t1 where c1 > 0 Published                    0     100000
→ 検証⑥ 実行計画に影響するヒント(full (t1))が加わったSQLは既存の結果キャッシュを使えなかった(SCAN_COUNT=0、8g30t47jdzgy6a7dbvydt2rmbb)
```

LIST10:結果キャッシュとバインド変数

```
SQL> exec dbms_result_cache.flush ;
SQL> variable b1 number ;
SQL> exec :b1 := 10000 ;
SQL> select /*+ result_cache */ * from t1 where c1 = :b1 ;
SQL> select name, status, scan_count, row_count, cache_key from v$result_cache_objects where type = 'Result' ;
NAME                                                  STATUS      SCAN_COUNT  ROW_COUNT CACHE_KEY
----------------------------------------------------- ----------- ---------- ---------- ----------------------------
select /*+ result_cache */ * from t1 where c1 = :b1   Published            0          1 84hwag8whmnz856r3xzazayss2

SQL> select /*+ result_cache */ * from t1 where c1 = :b1 ;
SQL> select name, status, scan_count, row_count, cache_key from v$result_cache_objects where type = 'Result' ;
NAME                                                  STATUS      SCAN_COUNT  ROW_COUNT CACHE_KEY
----------------------------------------------------- ----------- ---------- ---------- ----------------------------
select /*+ result_cache */ * from t1 where c1 = :b1   Published            1          1 84hwag8whmnz856r3xzazayss2
→ 検証① バインド変数の値が同じ場合、結果キャッシュを使った(SCAN_COUNT=1)

SQL> exec :b1 := 20000 ;
SQL> select /*+ result_cache */ * from t1 where c1 = :b1 ;
SQL> select name, status, scan_count, row_count, cache_key from v$result_cache_objects where type = 'Result' ;
NAME                                                  STATUS      SCAN_COUNT  ROW_COUNT CACHE_KEY
----------------------------------------------------- ----------- ---------- ---------- ----------------------------
select /*+ result_cache */ * from t1 where c1 = :b1   Published            0          1 70h0za2z24rwg0v2xfshlcs489
select /*+ result_cache */ * from t1 where c1 = :b1   Published            1          1 84hwag8whmnz856r3xzazayss2
→ 検証② バインド変数の値が異なる場合、新しい結果キャッシュオブジェクトを生成する(SCAN_COUNT=0)
```

- 結果セットが変更される可能性があるSQL、ファンクションには使用できない。例:ディクショナリ、または一時表を参照、シーケンス、SYSDATEを使用
- ヒントが優先される
- さまざまなクエリブロックで使える(インラインビュー、サブクエリ、UNION ALLなど)
- データが更新されると、該当表に関わる結果キャッシュオブジェクトはINVALIDになる(LIST11の検証⑤)

最後に本セクションのまとめとして、本機能の適用の際に参考にしてもらいたいガイドラインを挙げておく。

- キャッシュ対象は高負荷のSQLを中心に個別に選定する(RESULT_CACHE_MODE=MANUAL)
- 変更が少ない表を対象にする(変更が多いとロード処理の競合が激しくなる)
- 対象とするクエリの結果セットのサイズ、変数値の種類の数を考慮してキャッシュのサイジングを行なう

なお、本セクションではSQLの結果キャッシュを紹介したが、ファンクションの結果キャッシュとOCIクライアント結果キャッシュも同様のコンセプトで性能改善効果が期待できるので、併せて検討するのが良いだろう。

LIST11：結果キャッシュのサイジング

```
 -- 表「t1」と同じ設定(カラム、サイズ、データ)で表「t2」を作成しておく
SQL> select segment_name, bytes/(1024 * 1024) mb_bytes from dba_segments
  2  where lower(segment_name) in ('t1', 't2') order by 1;
SEGMENT_NAME                    MB_BYTES
------------------------------ ----------
T1                                     13
T2                                     13

SQL> exec dbms_result_cache.flush ;
SQL> alter system set result_cache_max_size = 10M ;
SQL> select /*+ result_cache */ * from t1 ;
SQL> select name, status, scan_count, row_count from v$result_cache_objects where type = 'Result' ;
NAME                                                     STATUS              SCAN_COUNT  ROW_COUNT
-------------------------------------------------------- ------------------ ---------- ----------
select /*+ result_cache */ * from t1                     Invalid                     0          0
→ 検証① 10MBの結果キャッシュでクエリの結果を保存できず、結果は使用できない(STATUS=Invalid)

SQL> alter system set result_cache_max_size = 15M ;
SQL> select /*+ result_cache */ * from t1 ;
SQL> select name, status, scan_count, row_count from v$result_cache_objects where type = 'Result' ;
NAME                                                     STATUS              SCAN_COUNT  ROW_COUNT
-------------------------------------------------------- ------------------ ---------- ----------
select /*+ result_cache */ * from t1                     Published                   0     100000
→ 検証② 15MBの結果キャッシュでクエリの結果を保存できて、使用可能(STATUS=Published)となった

SQL> select /*+ result_cache */ * from t2 ;
SQL> select name, status, scan_count, row_count from v$result_cache_objects where type = 'Result' ;
NAME                                                     STATUS              SCAN_COUNT  ROW_COUNT
-------------------------------------------------------- ------------------ ---------- ----------
select /*+ result_cache */ * from t2                     Published                   0     100000
→ 検証③ 15MBの結果キャッシュで、「t1」、「t2」両方の結果を保存できず、直近のSQLの結果のみがロードされた

SQL> alter system set result_cache_max_size = 30M ;
SQL> select /*+ result_cache */ * from t1 ;
SQL> select name, status, scan_count, row_count from v$result_cache_objects where type = 'Result' ;
NAME                                                     STATUS              SCAN_COUNT  ROW_COUNT
-------------------------------------------------------- ------------------ ---------- ----------
select /*+ result_cache */ * from t1                     Published                   0     100000
select /*+ result_cache */ * from t2                     Published                   0     100000
→ 検証④ 30MBの結果キャッシュで、「t1」、「t2」両方のクエリ結果をロードできた

SQL> update t1 set c2 = null where c1 = 1 ;
SQL> commit ;
SQL> select name, status, scan_count, row_count from v$result_cache_objects where type = 'Result' ;
NAME                                                     STATUS              SCAN_COUNT  ROW_COUNT
-------------------------------------------------------- ------------------ ---------- ----------
select /*+ result_cache */ * from t2                     Published                   0     100000
select /*+ result_cache */ * from t1                     Invalid                     0     100000
→ 検証⑤ データが更新された表に関わる結果キャッシュが使用できなくなった(STATUS=Invalid)
```

LIST12：V$SQL_HINT

```
SQL> desc v$sql_hint
 名前                            NULL?    型
 ------------------------------- -------- --------------------
 NAME                                     VARCHAR2(64)
 SQL_FEATURE                              VARCHAR2(64)
 CLASS                                    VARCHAR2(64)
 INVERSE                                  VARCHAR2(64)
 TARGET_LEVEL                             NUMBER
 PROPERTY                                 NUMBER
 VERSION                                  VARCHAR2(25)
 VERSION_OUTLINE                          VARCHAR2(25)

SQL> select name, inverse, version from v$sql_hint where version like '11%' ;

NAME                 INVERSE              VERSION
-------------------- -------------------- ----------
…
RESULT_CACHE         NO_RESULT_CACHE      11.1.0.6
NO_RESULT_CACHE      RESULT_CACHE         11.1.0.6
XML_DML_RWT_STMT                          11.1.0.6
PLACE_GROUP_BY       NO_PLACE_GROUP_BY    11.1.0.6
…
```

V$SQL_HINTビュー

11gから凄いビューが追加されている。V$SQL_HINTである。このビューを参照すると、SQLヒントをどのバージョンから使えるか、またその逆ヒントが確認できる(LIST12)。

例えば、結果キャッシュをさせたくないSQLには「NO_RESULT_CACHE」ヒントを指定できることが簡単に分かる。またヒントの正確な文字列が気になる時など、これからヒントを使う時にはマニュアルよりこのビューを参照することが多くなりそうだ。

拡張DESC

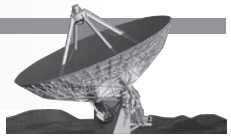
データベースのデータ量とともに管理対象のシステムが増えていく一方で、DBエンジニアの人数は限られているため、性能問題の診断や分析作業をできる限り効率化することが求められている。そのため繰り返し行なわれる作業はできるだけ自動化する必要があるので、ここで簡単に作れて、手軽に使えるスクリプトツールの例を紹介しておく。

DESC[RIBE]コマンドは、SQL*Plusで表の定義情報を参照するためによく使われる。個人的にはチューニングツールの範疇に入れても良いと思っている。性能問題を分析する際には、必ずと言って良いほどオブジェクトの定義を正確に把握する必要が生じるケースが多いからだ。

筆者の場合は、DESCコマンドの結果に、①表の統計情報、②索引情報(構成カラム、統計情報、パーティション関連情報など)、③表のDDL、④索引のDDLを合わせて確認するケースが多かったが、このような情報の参照機能を追加したDESCコマンドを個人的に「拡張DESC」と呼んでいる。

簡単な例から、その手軽さを実感してほしい。複数の索引を持つ表T1を対象として、その定義情報をLIST13のような拡張DESCコマンドから確認してみる。

スクリプトは、単にUSER_TABLES、USER_INDEXES、USER_IND_COLUMNSをSEL



LIST13：拡張DESCスクリプト

```
< desc.sql >
-----------------------------------------------------------------
-- usage   : @desc table_name ddl_yn storage_yn
-- example : desc.sql T1 N N
-----------------------------------------------------------------
set echo off
set timing off
set feedback off
set verify off
set serveroutput on
set long 100000

set line 150
col index_name for a20
col column_name for a59
col B for 9
col LB for 999,999
col DK for 99,999,999,999
col CF for 999,999,999
col NR for 99,999,999,999

define __TAB_NAME = &1
define __DDL_YN = &2 ;
define __STORAGE_YN = &3 ;

set pages 0
set line 80
select rpad('■',(76-length('&__TAB_NAME'))/2,'■')||'  '||upper('&__TAB_NAME')
       ||'  '||rpad('■',(76-length('&__TAB_NAME'))/2,'■') from dual ;

 -- 表の DESC
desc &__TAB_NAME

set pages 1000
set line 150
 -- 表の統計情報、パーティション化有無
select tablespace_name, decode(logging,'YES','Y','NO','N','') "L",
       num_rows, blocks, chain_cnt, avg_row_len, cache, last_analyzed,
       decode(partitioned,'YES','Y','N') "P"
from   user_tables
where  table_name = upper('&__TAB_NAME') ;

 -- 索引の構成カラム、統計情報、パーティション化有無
select A.index_name, ltrim( sys_connect_by_path( column_name , ' + ' ) , ' + ' ) column_name,
       B.blevel "B", B.leaf_blocks "LB", B.distinct_keys "DK", B.clustering_factor "CF",
       B.last_analyzed, decode(B.uniqueness,'UNIQUE','U','') "U",
       decode(B.partitioned,'YES','Y','N') "P"
from   (
        select index_name, column_name || decode(descend,'DESC','(D)','') column_name,
               row_number() over ( partition by index_name order by column_position ) rn ,
               count(*) over ( partition by index_name ) cnt
        from   user_ind_columns
        where  table_name = upper('&__TAB_NAME')
       ) A,
       user_indexes B
where  level = cnt
and    A.index_name = B.index_name
start with rn = 1
connect by prior A.index_name = A.index_name
and    prior rn = rn - 1
order by 1 ;

set line 1000
begin
    if upper('&__DDL_YN') = 'Y' then
        if upper('&__STORAGE_YN') = 'Y' then
            dbms_metadata.set_transform_param(dbms_metadata.session_transform,'STORAGE',true);
        else
            dbms_metadata.set_transform_param(dbms_metadata.session_transform,'STORAGE',false);
        end if ;

             -- 表の DDL
        for rec in (
            select dbms_metadata.get_ddl('TABLE',table_name,username) || ' ;' DDL
            from   user_tables, user_users where  table_name = upper('&__TAB_NAME')
        ) loop
          dbms_output.put_line(rec.ddl);
        end loop;

             -- 索引の DDL
        for rec in (
            select dbms_metadata.get_ddl('INDEX',index_name,username) || ' ;' DDL
            from   user_indexes, user_users where  table_name = upper('&__TAB_NAME') ORDER BY index_name
        ) loop
          dbms_output.put_line(rec.ddl);
        end loop;
    end if ;
end ;
/
```

ECTして見やすいフォーマットで情報を取り出しているものだ。また、DBMS_METADATA.GET_DDLを組み合わせて活用することで、普段いろいろと手間をかけて参照している情報を意外と簡単に取り出せる。LIST14からその結果を確認すると、見たい情報が一目瞭然に表示されている。

便利な商用ツールもたくさん出ている中で、い

LIST14：拡張DESCの実行例

```
SQL> @DESC T1 Y N
■■■■■■■■■■■■■■■■■■■■■■  T1  ■■■■■■■■■■■■■■■■■■■■■■
 名前                                      NULL?    型
 ----------------------------------------- -------- ----------------------------
 C1                                                 NUMBER
 C2                                                 NUMBER(10,2)
 C3                                                 VARCHAR2(100)
 C4                                                 CHAR(100)
 C5                                                 DATE

TABLESPACE_NAME                L   NUM_ROWS     BLOCKS  CHAIN_CNT AVG_ROW_LEN CACHE      LAST_ANA P
------------------------------ - ---------- ---------- ---------- ----------- ---------- -------- -
USERS                          Y     999414      24377          0         169     N      10-01-18 N

INDEX_NAME      COLUMN_NAME                    B       LB              DK           CF LAST_ANA U P
--------------- ------------------------------ -- -------- -------------- ------------ -------- - -
I1_T1           C1                             2    2,226       1,000,000       24,400 10-01-18   N
I2_T1           C1 + C2                        2    2,783       1,000,000       24,400 10-01-18 U N
I3_T1           C3                             2    8,429             100      983,396 10-01-18   N
I4_T1           C5 + C3                        3    9,514           7,300      987,208 10-01-18   N
I5_T1           C1 + C2 + C5                   2    4,046       1,000,000       24,400 10-01-18   N

→ 以下はDDL表示を行なう場合：ddl_yn='Y', storage_yn='N'
  CREATE TABLE "MAXGAUGE"."T1"
   (    "C1" NUMBER,
        "C2" NUMBER(10,2),
        "C3" VARCHAR2(100),
        "C4" CHAR(100),
        "C5" DATE
   ) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 NOCOMPRESS LOGGING
  TABLESPACE "USERS"
  ;

  CREATE INDEX "MAXGAUGE"."I1_T1" ON "MAXGAUGE"."T1" ("C1")
  PCTFREE 10 INITRANS 2 MAXTRANS 255 COMPUTE STATISTICS
  TABLESPACE "USERS"
  ;
...
```

LIST16：アラートログの分析スクリプトの実行例

```
SQL> @alert_analyze 2009/01/01 2009/12/31 %
WHEN        ERROR_CODE   COUNT(*)
----------  ----------  ----------
2009/11/20 ORA-32320            5
2009/11/20 ORA-12012            5
2009/11/20 ORA-06512           25
2009/12/01 ORA-1652             2
```

まひとつ自分の好みに合うものが見つからなかった方には、このような自作スクリプトを作成してみることを推奨する。これによって、一層自分ならではのノウハウが蓄積できるだろう。また、改めて言っておきたいポイントは、オブジェクトの定義を正確に把握することはとても重要だということだ。

アラートログ分析例

Oracleが提供するデータの中で、障害解析で最も重要なものがアラートログだということに異論はないだろう。しかし、運用環境で生成されるほとんどのアラートログは巨大なため、検索しながら目で追うことには限界がある。そこで、テキストファイルの分析を自動化することでより手軽に必要な情報を得られる。

例えば、ORA-600、ORA-4031のような明示的なエラーの発生頻度や分布の確認にLIST15[注4]のスクリプトを使ってみよう。LIST16の例のように、特定日付の特定エラーの発生状況が一目で分かるように集計できる。

このスクリプトは、①アラートログを取り込む一時表を作成する処理、②ファイルの中身を配列で戻す関数、③正規表現[注5]を活用してアラートログから日付情報をピックアップする処理、④エラーを抽出して一時表に格納する処理、⑤集まったエラーをグルーピングして出力する処理で構成されている。

③では「20:38:51 2010」のようなパターンを持つ文字列を、④では最初の「:」の箇所を正規表現を使って検索しているが、このようなパターン検索を活用すると、特にトレースファイルなどテキストに対してさまざまなパターンの分析ができるようになるのでもう一度注目してほしい。

特定エラーだけでなく、キーとなる特定文字列を分析する際にもその力を発揮する。例えばトレースファイルで特定の待機イベントが、どのファイル、どのブロックで、どのような頻度で発生しているかを分析する際に、特に同じ作業で苦労した経験がある方の眼には魅力的に映ることだろう。

*　　*　　*

いま一度、日常の作業を振り返ってみて、繰り返し作業に多くの時間を取られているのであれば、一部でも自動化テクニックを取り入れる努力をすることがエンジニアの生活をより快適にしていく近道だと思う。

最後に、ここで紹介してきたTipsの1つ1つが完結された豆知識として、関連領域の知識を深めることに少しでも貢献できれば、また皆さんの現場業務の役に立つことができれば幸いである。

DBM

■注意

本記事のLISTは、Oracleのバージョン10.2.0.1.0、11.1.0.6.0環境で行なった検証結果で、誌面の制約上一部抜粋／編集したものです。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りますので、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
- Oracle Database Documentation Library
- http://support.oracle.co.jp/
- https://metalink.oracle.com/

本誌付録CD-ROMに、誌面に掲載できなかったLISTおよび全LISTの完全版と検証結果を収録しています。また、DBマガジンのWebサイト（http://www.shoeisha.com/mag/dbm/）からもダウンロードできます。

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）株式会社所属。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。

注4：誌面の都合により、LIST15は付録CD-ROMに収録しています。（編集部）
注5：正規表現の詳細は、マニュアル「Oracle Databaseアプリケーション開発者ガイド- 基礎編」を参照のこと。