複雑で難解な内部動作も見通せる

OWIによるOracleの性能改善と障害対策

特集2

ITサービスの性能（パフォーマンス）問題の解決を考える場合、一般的には応答時間に基づく性能分析が常識になっている。しかしOracleデータベースの内部解析については、まだその考え方が定着していないのが現状である。その原因はいろいろ考えられるが、基本的には、未だにOracleがブラックボックスだと見られている部分が大きいのではないだろうか。本特集では、Oracleの性能問題の診断／分析においてすでに主流になりつつある“時間ベースの解析手法”を提供するOWI（Oracle Wait Interface）について詳しく解説する。

株式会社サンブリッジアンシス

川向一郎 KAWAMUKAI,Ichiro

金 圭福 KIM,Gyubok

PART 1 Oracleの時間ベース指標を分析し性能改善につなげるOWIの基礎

統計情報からAP処理のボトルネックを読み取る

本番環境でDBシステムの性能問題に遭遇したとき、現場ではCPUやメモリの使用状況、I/O状況、ネットワーク負荷を確認するのが一般的だ。これらを把握することで、トラブル発生時点でのシステム全体のおおよその状況が分かり、ある程度の推測が可能となる。

ところで、性能問題におけるボトルネックは、その50%以上がDBおよびアプリケーションで発生していると言われており、特に大容量かつ大量ユーザーがいる昨今のシステムではその傾向がより強くなっている。そのため、性能トラブル時にはDBのシステム環境の概要情報はもちろん、DB内部の性能統計情報を参照することが必要になる。収集済みのデータがない場合は、再実行もしくは再現テストを行なってSTATSPACKやSQLトレースを取得するのが現場の実情であろう。

ここからは、現場でよく発生する性能問題のパターンをシンプルにしたテストシナリオ（環境：Windows 2003 Server、Oracle 10.2.0.3）から、性能分析でよく使われるSTATSPACKレポートとトレースのサマリ（見やすくするため一部編集）でよく確認するOracle Wait Interface（以下、OWI。詳細は後述）の時間統計情報を、どのように読み取れば良いか述べていく。

まずは、これから示すレポートを基にOracle内部の処理が遅くなったかどうか考えてみよう。性能低下が発生している場合、そのボトルネック箇所、原因および改善ポイントも併せて考えてほしい。なお、以降では考え方のヒントを示すこととし、詳しい解説と改善策は本パートの後半部で行なう。

ケース① ライブラリキャッシュ（LIST1）

CPU使用時間より1.5倍程度の待ちが発生している。特にトップ1（最長）の待機として「latch: library cache」イベントが挙がっていることから、ライブラリキャッシュでのSQL解析処理もしくはオブジェクト参照でボトルネックが発生していると考えられる。

ケース② データファイルの読み込み（LIST2）

時間ベースで見ると、ほとんどの処理時間がイベントによる待ちで所要されている。特に、「db file sequential read」イベントによる待機がほとんどを占めている状況から、ディスクに対するシングルブロックの読み込み処理での競合がボトルネックになっていると考えられる。

データ処理においてディスクI/Oは避けられないため、I/O関連イベントの発生そのものが性能低下現象だと見なす必要はないが、今回の結果のように待機時間がほかの処理時間に比べて顕著に長い場合、その発生原因について診断／分析する必要がある。

ケース③ バッファキャッシュの検索処理（LIST3）

CPU使用時間の5倍以上が待ち状態になっている。特に「latch: cache buffers chains」イベントによる待機がほとんどを占めていることから、バッファキャッシュに対する検索処理でボトルネックが発生していると考えられる。

ケース④ バッファブロックに対する競合（LIST4）

上位の待機時間の合計がCPU使用時間

```
Top 5 Timed Events                                    ライブラリキャッシュで
~~~~~~~~~~~~~~~~~~                                   ボトルネック！
                                                                        Avg   %Total
                                                                       wait    Call
Event                                            Waits    Time (s)     (ms)    Time
------------------------------------------- ---------- ----------- ------ ------
CPU time                                                      1,055            34.7
latch: library cache                              7,278         750    103    24.7
latch: library cache lock                         4,194         465    111    15.3
job scheduler coordinator slave wait                 23         371  16124    12.2
cursor: pin S wait on X                           2,019         206    102     6.8
```

LIST1 ケース①の現象

```
Top 5 Timed Events                                    データファイルの読み込み処理で
~~~~~~~~~~~~~~~~~~                                   最大のボトルネック！
                                                                        Avg   %Total
                                                                       wait    Call
Event                                            Waits    Time (s)     (ms)    Time
------------------------------------------- ---------- ----------- ------ ------
db file sequential read                          15,077         182     12    90.8
CPU time                                                         13            6.4
control file sequential read                        366           2      5     1.0
log file parallel write                              71           1     14      .5
control file parallel write                          99           1      9      .4
```

LIST2 ケース②の現象

OWIによるOracleの性能改善と障害対策

を18倍以上上回っている。特に「buffer busy waits」と「log file sync」イベントによる待機がほとんどを占めている状況から、同一ブロックに対する競合とREDOログファイルへの書き込み処理がボトルネックになっていると考えられる。まず診断のポイントをリストアップし、改善効果を予測してみよう。

```
Top 5 Timed Events                                                    Avg %Total
~~~~~~~~~~~~~~~~~~                                                   wait   Call
Event                                            Waits    Time (s)   (ms)   Time
---------------------------------------- ------------ ----------- ------ ------
latch: cache buffers chains                    10,587       2,091    197   67.2
CPU time                                                      460           14.8
latch free                                      1,994         322    161   10.4
job scheduler coordinator slave wait               11         179  16307    5.8
read by other session                          13,009          35      3    1.1
```

バッファキャッシュの検索処理でボトルネック！

LIST3 ケース③の現象

```
Top 5 Timed Events                                                    Avg %Total
~~~~~~~~~~~~~~~~~~                                                   wait   Call
Event                                            Waits    Time (s)   (ms)   Time
---------------------------------------- ------------ ----------- ------ ------
buffer busy waits                              43,921       1,727     39   44.8
log file sync                                  43,096       1,107     26   28.7
log file switch (checkpoint incomplete)           355         283    797    7.3
CPU time                                                      175           4.5
job scheduler coordinator slave wait                8         128  16010    3.3
```

バッファブロックに対する競合！

LIST4 ケース④の現象

```
Top 5 Timed Events                                                    Avg %Total
~~~~~~~~~~~~~~~~~~                                                   wait   Call
Event                                            Waits    Time (s)   (ms)   Time
---------------------------------------- ------------ ----------- ------ ------
enq: TX - row lock contention                   1,784       4,699   2634   93.0
PL/SQL lock timer                               4,658         319     68    6.3
CPU time                                                       22            .4
log file switch completion                          6           5    814    .1
db file sequential read                           375           3      8    .1
```

行ロック待ちがトップ1！

LIST5 ケース⑤の現象

```
Top 5 Timed Events                                                    Avg %Total
~~~~~~~~~~~~~~~~~~                                                   wait   Call
Event                                            Waits    Time (s)   (ms)   Time
---------------------------------------- ------------ ----------- ------ ------
log buffer space                               12,143       2,773    228   52.7
buffer busy waits                               4,101         737    180   14.0
enq: HW - contention                            2,405         429    178    8.2
log file switch completion                      1,147         422    368    8.0
log file parallel write                         1,143         160    140    3.0
```

REDOログバッファでの領域競合！

LIST6 ケース⑥の現象

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse     1149      0.28      38.39          0          0          0           0
Execute 3641058   729.93   12967.91         82  124576169   11813911     1822467
Fetch       387     0.01       0.00          0        228          0         330
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total   3642594   730.23   13006.31         82  124576397   11813911     1822797

Elapsed times inclu                                    nts:
  Event waited on                                    Times   Max. Wait  Total Waited
  ---------------------------------------- ----------  ---------- ------------
  log file sync                                      4163        5.35       486.24
  latch: enqueue hash chains                         2737        0.57       292.84
  latch free                                         1539        0.99       283.74
  latch: cache buffers lru chain                     3046        0.57       279.13
  log file switch completion                          397        1.05       219.50
  latch: cache buffers chains                        1974        0.84       210.64
  buffer busy waits                                  2679        1.32       165.21
```

REDOログファイルの同期化処理でボトルネック！

LIST7 ケース⑦の現象

ケース⑤ 行ロック待ち（LIST5）

CPUをほとんど使わず、DB内部でロック待ち状態が多く発生している。ロックの中でも「enq：TX － row lock contention」イベントによる待機現象から、データの更新／追加処理での競合によるボトルネックになっていると考えられる。

ケース⑥ REDOログバッファでの領域競合（LIST6）

CPU使用時間が上位に入れないほど、待機イベントにほとんどの処理時間が使われている。特に、「log buffer space」と「buffer busy waits」イベントによる待機がほとんどを占めている状況から、REDOログバッファへの記録処理と同一ブロックに対する競合でボトルネックになっていると考えられる。まず、診断のポイントをリストアップし、改善効果を予測してみよう。

ケース⑦ REDOログファイルの同期化処理（LIST7）

CPU使用時間より2倍以上の待ちが発生している。特にトップ1の待機として「log file sync」イベントが挙がっていることから、REDOログファイルとの同期化処理でボトルネックが発生していると考えられる。まず、当該の遅延を解消してから、状況の変化に応じて次の手を考えると良いだろう。

ケース⑧ シーケンスに対するロック待ち（LIST8）

上位の待機時間の合計がCPU使用時間を3倍以上上回っている。特に「enq: SQ － contention」イベントによる遅延が目立っていることから、シーケンス取得処理のロックによる遅延現象だと考えられる。そのほかの待機現象は見られないが、トッ

プ1のイベントによって隠れている可能性もあるため、トップ1の遅延現象がなくなったときに表出することも考えられる。

ケース⑨ DBリンク経由のデータの転送（LIST9）

時間ベースにてほとんどの処理時間が「SQL*Net message from dblink」イベントによる待ちで所要されているため、DBリンクを経由してデータの転送処理でボトルネックになっていると考えられる。そのほかの待機現象は見られないが、トップ1のイベントによって隠れている可能性もあるため、トップ1の遅延現象がなくなったときに表出することも考えられる。

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse   151721      2.70       5.77          0          8          0           0
Execute 3151954   397.14    1870.02          1     284020     613423      153627
Fetch   3000795   163.75    2062.03         43      10227     168477     3001986
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total   6304470   563.59    3937.83         44     294255     781900     3155613

  Event waited on                             Times   Max. Wait  Total Waited
  ------------------------------------------  Waited  ----------  ------------
  enq: SQ - contention                        348423        1.63       1848.59
  log file switch completion                       8        0.99          2.48
  latch: library cache                           896        0.06          1.61
  enq: TX - row lock contention                   32        0.26          1.54
  latch free                                    1112        0.02          1.54
```

シーケンスに対するロック待ち！

LIST8 ケース⑧の現象

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse      131      0.64       9.46          0         40         10           0
Execute  30132     58.43    1464.62          0       7020        468         262
Fetch    30062     63.81     637.83          0         67          0       30050
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total    60325    122.89    2111.92          0       7127        478       30312

Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ------------------------------------------  Waited  ----------  ------------
  SQL*Net message from dblink                 60082        1.00       1004.46
  single-task message                             9        1.17          6.14
  SQL*Net message to dblink                   60082        0.00          0.24
```

DBリンク経由のデータ転送の負荷！

LIST9 ケース⑨の現象

Oracleの診断／分析メソッドとしてのOWI

所要時間および応答時間は、DBだけではなく、アプリケーションでもネットワーク階層でも、性能の計測／検証／診断／分析を実施するときには最も重要な指標になる。ここまで、いくつかの事例をSTATSPACKレポートとトレースファイルのサマリで確認してきたが、同じく診断／分析のキーとなっているのは時間ベース情報である。

これまでは、それほど重要視されていなかったかもしれないが、Oracleは常に時間に基づいた統計情報をOWIという仕組みで提供し、性能診断／分析の手がかりを与えている。ここからはOWIをより理解し、より効率良く活用することを目指して、改めてOWIのコンセプトから紹介していこう。

OWIとは

Oracleは、ユーザーからの作業を処理する過程で必要なリソースを獲得できない場合、そのリソースがリリースされるまで待ち状態に入る（図1）。例えば、先述のケース⑤のように、特定データを更新するためには、該当レコードに対する行ロック「enq: TX － row lock contention」を獲得するまで待機しなければならない。この例では「4699」秒待機して、「22」秒で実質の更新処理を行なったことになる。

このように、どのプロセスがどのリソースに対してどれくらい待ち状態だったのか、実質の処理にはどれくらい所要されたのかなどの統計データを記録して、後で確認できる手法を提供する一連の機能とインターフェイス、そして診断／分析方法を

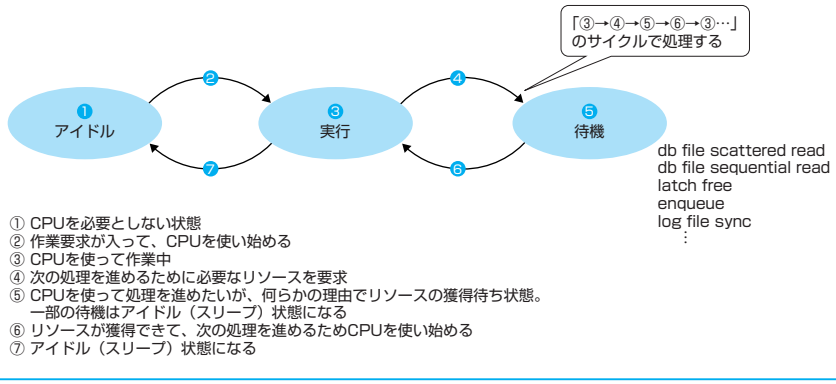


図1 プロセスの状態とOWI

特集2

OWIによるOracleの性能改善と障害対策

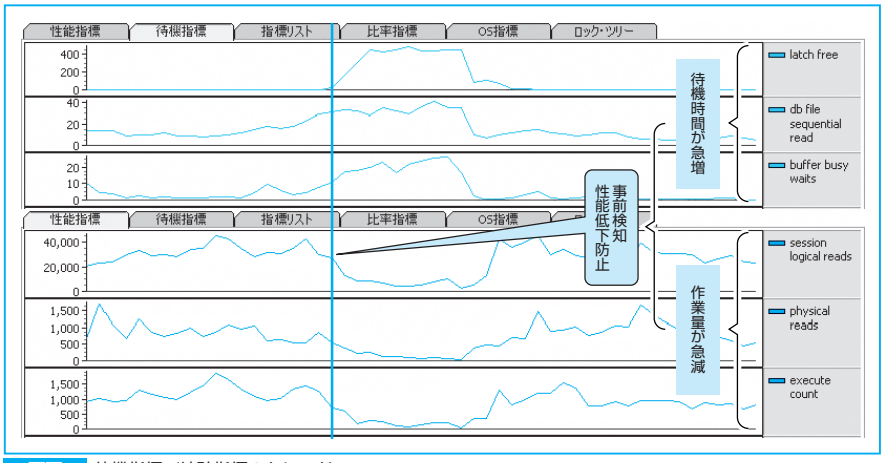
ひっくるめてOWI（Oracle Wait Interface）と呼んでいる。また、リソースを獲得するまで待っている状態を「Wait Event（イベントを待機する）」と言う。

OWIは時間に基づいた診断／分析を提供し、CPU処理時間または待機時間を最小限にチューニングするためのヒントを提示する。そのチューニング結果は、次の式のようにユーザーが感じる処理時間（応答時間）の改善にそのままつながる。

処理時間（応答時間）＝
サービス時間（CPU使用時間）＋待機時間

```
Instance Efficiency Percentages
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            Buffer Nowait %:   99.45          Redo NoWait %:   99.64
              Buffer Hit   %:   99.77      In-memory Sort %:  100.00
              Library Hit  %:   98.81          Soft Parse %:   91.64
         Execute to Parse %:   97.33            Latch Hit %:   99.90
Parse CPU to Parse Elapsd %:   38.68        Non-Parse CPU:    96.56
```

LIST10 STATSPACKレポートの「インスタンス効率」（例）



画面 待機指標／統計指標のトレンド

OWI活用のメリット

皆さんのシステムにおいて、性能問題はどのように発生するだろうか。「ユーザーからのクレームがあった」「バッチ処理が制限時間を越えている」「ある画面で1分が過ぎても結果が返ってこない」といった基準は、性能問題の認識において1つの重要なパロメータになる。性能問題は適切なタイミングで正確に認識することから、正しい改善案が生まれる。

LIST10を見てほしい。バッファキャッシュの平均共有率（Buffer Hit）が99.77％とか、メモリソート率（In-memory Sort）が100％といった情報から、この区間は性能問題がなかったと判断できるだろうか。実はこのデータはケース⑥で使われたデータの一部である。比率情報は参考程度の指標にはなるが、正確な性能診断の役には立たない。

OWIはケース⑥の解説のように性能問題を時間ベースでの認識だけではなく、イベント名でどの部分での問題なのかを直感的に提供する。そのため、各待機を解消することで、どれくらいの改善効果が見込まれるかもその場で分かるというわけだ。また、イベントが発生する仕組みを理解する（パート2にて解説）ことで、チューニングポイントのアイデアも提示される。つまり、OWIは性能問題において「認識 → 診断／分析 → チューニング」手法を、正確かつ直感的に提供するのである。

予兆監視

OWIは予兆監視運用をサポートする。**画面**はシステムレベルの性能指標とイベントを1分間隔で取得してその推移をグラフ化した画面である。何らかの傾向が読み取

COLUMN 待機イベントの背景

待機イベント（Wait Event）はOracle7から実装された。この時期は、ちょうどOracleがエンタープライズ環境で普及し始めた時期と重なる。大容量データの環境で同時接続数およびトランザクション数が急増したため、それまではさほど問題とならなかったリソースに対する競合がDB内部で急増することになった。そこで待ち時間を計るタイマーとして待機イベントが内部コードとして追加され始めたと考えられる。このタイマーは、Oracleのバージョンアップや機能拡張とともに、100個前後で始まって、Oracle 10gで800個（10.2.0.3で876個）を超えている（**図A**）。その傾向にはenqueやlatchの細分化などイベントの精度を高める方向性もあり、Oracleがより詳しく性能問題をレポートしていることを意味する。

さて、次期バージョンの11gではどれくらい増えるのだろうか。日本ではまだ待機イベントに関する資料が普及していないこともあって、多くのエンジニアが有効に活用できてない状況であるが、Oracleの進化とともにその重要性がますます高まることは間違いないだろう。

図A Oracleのバージョンアップによる待機イベント数の増加

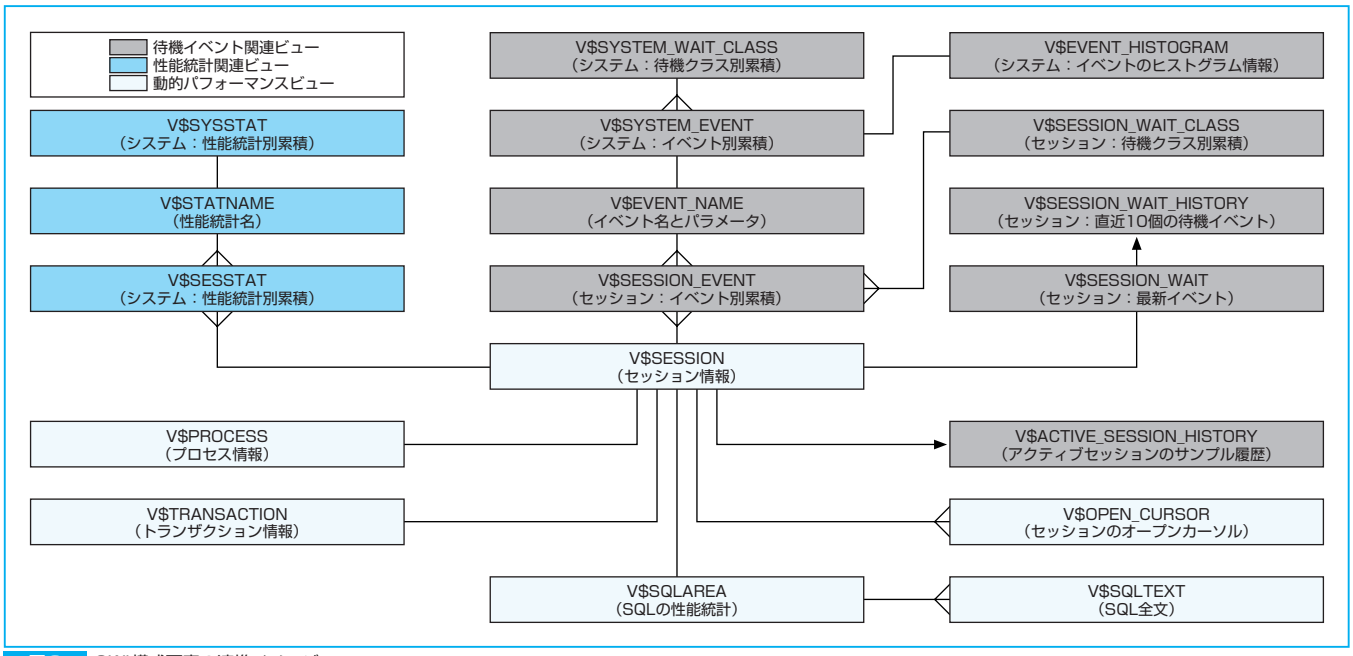


図2 OWI構成要素の連携イメージ

れるだろうか。

トレンドから、待機指標（イベント：latch free、db file sequential read、buffer busy waits）の待機時間がタテ線から急増し始める反面、性能指標（論理読み取り、物理読み取り、SQLの実行回数）が急減し始めることが分かる。当該グラフの推移は30分間での動きで、待機／性能指標の増減をあらかじめ検知できていれば急激な性能低下は事前に防ぐことができる。

上記の推移からも分かると思うが、そもそも性能問題は突然起きる現象ではない。1～2分前にその兆候を表わすものもあれば、数ヶ月間をかけて徐々に兆候として表われる現象もある。その兆候は確実にOWIの指標として表現されるため、事前にその特徴を把握しておけば、性能改善はもちろんトラブルの回避と潜在リスクの解消にも活用できるようになる。このような運用方法を予兆監視運用と言う。

OWIを提供するツール

Oracleはさまざまなインターフェイスでを提供している。前半部で紹介したテスト結果で見てきたが、STATSPACKレポートとSQLトレースファイルでイベントの統計情報を参照できる。前者は一定期間の運用状況のサマリを簡単に読み取れるため、システム全体の状況を把握するのに適している。後者はSQLやカーソル単位でイベントの統計を収集するため細かい分析が必要なときに最適である。

また、最も重要なインターフェイスとして、動的パフォーマンスビュー、待機イベント関連ビュー、性能統計関連ビューが提供されている（図2）。各種ビューを適切に連結して参照することで、システムレベルからセッションレベル、SQLレベルまでのサマリ情報から詳細情報まで、幅広く正確な統計情報を簡単に確認できる。特にOracle 10gでは、履歴情報を格納するビューが追加され、より手軽な分析が可能になっている。

最も理想的なOWIの活用方法は、各種ビューを定期的に参照し、その情報を保存することで、運用状況の推移分析や特定タイミングでの運用状況分析を行なうことで

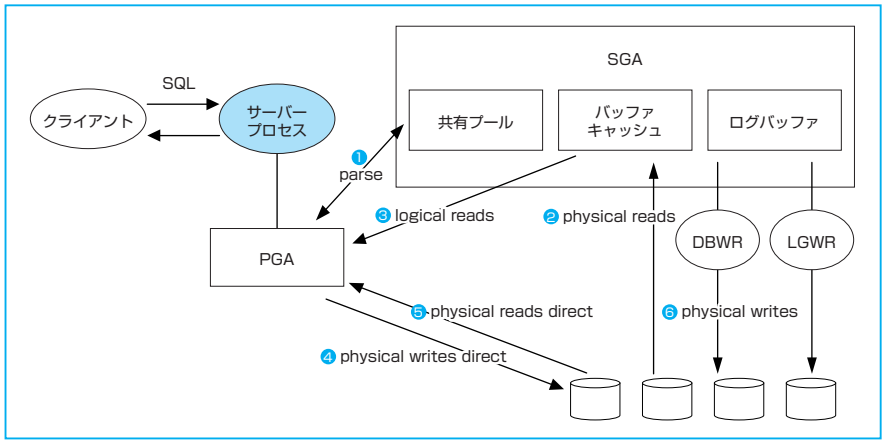


図3 Oracleアーキテクチャと統計情報

特集2

OWIによるOracleの性能改善と障害対策

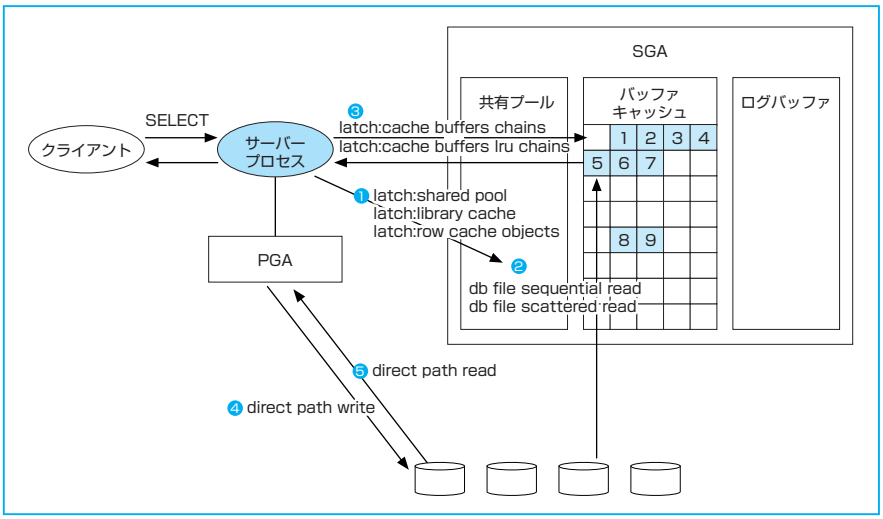


図4 SELECT処理時の待機イベント

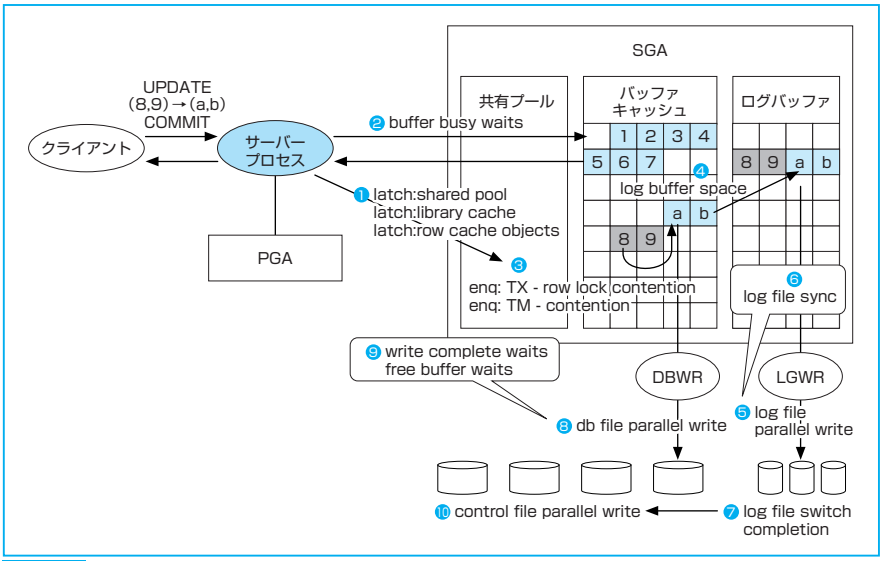


図5 UPDATE時の待機イベント

OWIを活用した性能改善

ここからは、前半部で紹介したシナリオ別のボトルネックに対する具体的な改善策とそれぞれの適用結果を説明していく。統計情報の簡単な解析とチューニング前後の統計情報を見比べることで、OWIの効果を知っていただければと思う。

ケース① の改善ポイント

まず、大量のSQL解析処理がDB内部でどのようなボトルネックで表われるかを確認する。通常、バインド変数を使ったSQLの解析処理はシステム負荷に影響を与えないとされているが、それは事実の半分だけである。もちろん新しいSQLの解析処理ほどの負荷ではないが、ライブラリキャッシュ内部に格納されている既存のオブジェクトを検索する処理は欠かせないので、SQL解析処理の数そのものがデータベース全体の性能に影響を与えることがある。

特に同時処理が多くなると、解析処理に使われるリソース（詳細はパート2のコラム「Oracleの同期化メカニズム、ラッチとロック」を参照）は限定されるため、一部のプロセスは「latch: library cache」イベントによる待ち状態になる。今回はソフト解析（詳細はパート2のコラム「SQL解析処理のフロー」を参照）も行なわないように「SESSION_CACHED_CURSORS」を設定することで、「2847 → 1734」秒（39.1％短縮）の応答時間の性能改善を実現した

ある。しかし、SQLで頻繁に情報を取得する処理が逆にDBに負荷を与えてしまったり、保存されたデータの加工の難しさや手間のため、ほとんど利用されていないのが現状である。

OracleアーキテクチャとOWI

OWIはOracleアーキテクチャと密接な関係にあり、OWIそのものがOracleの内部構造をそのまま表わしていると言っても過言ではない。

図3は統計情報がDB内部のどの部分で発生しているのかを表わしている。図4は「SELECT」処理時に、図5は「UPDATE」処理時に、DB内部で発生するイベントの流れを簡単なイメージにしたものだ。各領域とOWIの詳細についてはパート2で詳しく説明する。

各指標はOracleの各リソースと連携して、プロセスによるリソースの使用状況を数値化しているので、各性能／待機指標が発生する仕組みと領域を理解することでOWIのサポートをより受けやすくなる。まさしくOracleの内部動作を理解するツールとしてOWIを活用できるのである。

(LIST11)。

ケース② の改善ポイント

プロセスからディスクI/Oのリクエストがあったとき、対象のデータを含むブロックがSGAにロードされるまで当該プロセスは待機状態になるが、このときロードされるブロックの単位が1ブロックの場合は「db file sequential read」、複数ブロックの場合は「db file scattered read」イベントが発生する。

このテストでは、索引を経由してアクセスするデータの分散状況によってディスクI/O処理時間が急激に変化することを待機イベントに基づいて検証したシミュレーションになる。当然のことながらアクセスするデータブロック数が多いほど性能が悪くなり、少ないほど性能が良く、処理時間が短くなる。

今回は、索引キーの順番に合わせて表データを入れ直した結果、「199 → 13」秒(93.5%短縮)に処理時間が激減した(LIST12)。実際にアクセスするデータブロックが減ることで無駄なCPU処理時間がなくなったことも確認しよう。

ケース③ の改善ポイント

この改善策では、まず広い範囲に対する検索で発生するボトルネック現象を再現した。このような処理が認識されないまま実行されると、どのようなボトルネックが発生するかを確認できる。サーバープロセスからデータの要求(SELECT)があると、Oracleは該当ブロックがメモリにロードされているか確認するが、データの整合性とその検索処理を効率良く行なうため「cache buffers chains」ラッチという仕組みを使う。

検索範囲が広いとその検索にかかる時間も増えて、同じ検索処理を要求するほかのプロセスは待機状態になる。このときの待ち状態を「latch: cache buffers chains」イベントと言う。今回は、データ参照処理のSQLに合わせて、索引を最適化(項目の追加)することで、「3087 → 13」秒(99.6%短縮)と劇的な性能改善を実現した(LIST13)。これはアクセスするブロックの範囲を最適化することによって、CPU処理とメモリブロックに対する競合が大幅に減少したためである。

ケース④ の改善ポイント

Oracleはデータブロック単位でI/Oを行なっているため、複数のセッションが同時に異なるデータを変更しても、そのデータが物理的に同じブロックにある場合は、該当するメモリブロックを保護するため排他制御処理を行なう。そのとき、バッファロック[注]に対する競合が発生するが、バッファロックを獲得できなかったプロセスはロックが解除されるまで「buffer busy waits」イベントで待機する。

当イベントはどのシステムでもある程度は発生する現象なので発生そのものを問題視する必要はないが、トップ5に入るときには注意が必要である。通常、「UPDATE」と「UPDATE」、「INSERT」と「UPDATE」によって発生するが、「SELECT」と「UPDATE」によっても発生する。当テストでは、同時に複数のセッションが同じブロックにある異なるデータを更新している状況であった。

このような遅延を回避するため、アプリケーションの特性を考慮し、設計時に同じブロックのデータに対して追加/更新処理を最小化することが基本となる。一般的にパーティション化、FREELISTSの調整もしくはASSM適用などで同時性の多いブロックを分散することが推奨されるが、運用中の本番環境ではなかなか現実的ではない部分もあって、解消策に悩まされた苦い経験がある方もいることと思う。

この現象が上位に入る場合は、該当するセグメントを特定することから始める必要

注:メモリブロックに対する同時アクセスを制御するため共有/排他制御を行なうメカニズム。

```
Top 5 Timed Events                                                    Avg %Total
~~~~~~~~~~~~~~~~~~                                                   wait   Call
Event                                            Waits    Time (s)    (ms)   Time
---------------------------------------------- ------------ ----------- ------ ------
CPU time                                                        1,006          52.3
cursor: pin S wait on X                            2,815          389    138   20.2
cursor: pin S                                      3,944          177     45    9.2
job scheduler coordinator slave wait                   6           98  16274    5.1
os thread startup                                     76           64    847    3.3
```

SQL解析処理の回数を最小化して解消

LIST11 ケース①の改善結果

```
Top 5 Timed Events                                                    Avg %Total
~~~~~~~~~~~~~~~~~~                                                   wait   Call
Event                                            Waits    Time (s)    (ms)   Time
---------------------------------------------- ------------ ----------- ------ ------
CPU time                                                            7          43.7
db file sequential read                              665           6      9   41.2
control file sequential read                         344           0      1    3.0
db file scattered read                                71           0      6    2.6
control file parallel write                           59           0      7    2.6
```

データの入れ直しでディスクI/O待ちを最小化

LIST12 ケース②の改善結果

```
Top 5 Timed Events                                                    Avg %Total
~~~~~~~~~~~~~~~~~~                                                   wait   Call
Event                                            Waits    Time (s)    (ms)   Time
---------------------------------------------- ------------ ----------- ------ ------
CPU time                                                            8          63.9
enq: TX - row lock contention                         95           2     20   14.7
db file sequential read                              138           1      6    6.1
os thread startup                                     30           1     21    4.8
control file sequential read                         321           1      2    4.3
```

索引構成の変更でアクセス範囲を最適化

LIST13 ケース③の改善結果

がある。当テストでは、ハッシュパーティションを適用することで、「3420 → 1530」秒（55.3％短縮）の応答時間の性能改善を実現した（LIST14）が、REDOログファイルでの競合解消がまだ課題として残っている。

ケース⑤の改善ポイント

特に同時ユーザーが多いシステムでよく見られる現象で、アプリケーションのトランザクション制御が正しく行なわれていない場合に発生する行ロック待ち状態を再現した。コミットなしでデータに対するロックを長時間持ち続ける処理でよく発生するが、アプリケーションごとにデータ更新対象の順番が異なる場合も多々あり、運用面およびトランザクション制御の修正で対処するべきだろう。

このテストでは、データ更新後コミットを行なうように修正したことで、行ロック待ち状態が発生するものの、「5048 → 401」秒（92.1％短縮）の応答時間の性能改善を実現した（LIST15）。

ケース⑥の改善ポイント

ここでは、複数のセッションが同時にDMLを発行した際、REDOログバッファにかかる負荷をシミュレーションしてみた。通常は、コミット時の3秒ごとに、REDOログバッファの1/3以上あるいは1MB以上データが溜まったときに、LGWRによってREDOログバッファの内容がREDOログに記録される。

しかし、短時間でコミットなしの大量のREDOエントリが発生する場合、LGWRが状況を検知し、書き出す前にREDOログバッファはいっぱいになる。これはディスクI/Oよりメモリ処理の速度がはるかに上回っているからで、REDOログバッファがいっぱいになるとその次のREDOエントリは入れなくなり、プロセスは待機状態になる。このときの待ち状態を「log buffer space」イベントと言う。

パーティション化によるデータ分散でボトルネック解消

Top 5 Timed Events Event	Waits	Time (s)	Avg wait (ms)	%Total Call Time
log file sync	[illegible]	790	26	42.1
log file switch (checkpoint incomplete)	309	282	913	15.0
buffer busy waits	2,571	197	76	10.5
CPU time		135		7.2
enq: TX - row lock contention	2,654	126	47	6.7

LIST14 ケース④の改善結果

トランザクション制御の見直しで解消

Top 5 Timed Events Event	Waits	Time (s)	Avg wait (ms)	%Total Call Time
PL/SQL lock timer	4,614	318	69	77.8
enq: TX - row lock contention	2,418	48	20	11.7
job scheduler coordinator slave wait	1	16	16000	3.9
CPU time		12		2.9
latch: cache buffers chains	165	7	41	1.7

LIST15 ケース⑤の改善結果

REDOログバッファのサイズの変更で解消

Top 5 Timed Events Event	Waits	Time (s)	Avg wait (ms)	%Total Call Time
enq: HW - contention	3,973	2,184	550	52.5
log file switch (checkpoint incomplete)	580	505	871	12.1
buffer busy waits	4,7[illegible]	464	98	11.2
log file switch completion	470	248	527	6.0
log buffer space	210	135	643	3.2

LIST16 ケース⑥の改善結果

このテストでは、REDOログバッファのサイズを大きくすることで、「4521 → 3536」秒（21.8％短縮）の応答時間の性能改善を実現した（LIST16）。実質トップ1だった「log buffer space」イベントがほとんど（95％以上）なくなったにもかかわらず、全体の所要時間が期待するほど縮小されていないのは、最初のボトルネックによって表出していなかった、潜在的なボトルネックが表面に出たためである。「enq: HW − contention」がHWM（ハイウォーターマーク：高水位標）の移動の際に発生するイベントであることをヒントにして、続けて診断／分析を行なうべきであろう。

ケース⑦の改善ポイント

このテストは、頻繁なコミット処理が処理全体の応答時間にどのような影響を与えるかを確認するためのシミュレーションになる。サーバープロセスでコミットおよびロールバックが実行されると、そのデータを保証するためLGWRがその時点までのREDOエントリをREDOログファイルに書き出す。

このとき、サーバープロセスはLGWRの処理が完了するまで「log file sync」による待ち状態になる。このような場合は、グループコミット（例えば1000件ごと）などでその回数を減らすことで、当該待機は解消される。今回は、「2667.5 → 1347.0」秒（49.5％短縮）の改善を図れた（LIST17）。また、それほどクリティカルではないが、ほかの待機イベントに対しても引き続き次の改善策を検討したほうが良いだろう。

ケース⑧の改善ポイント

SQロックは低い値の「CACHE」属性を持ったシーケンスを同時に大量に発行したときに発生する。シーケンスは低負荷でユニークな値を取得できることを目的にしているため、一定区間の値をメモリにキャッシュしておく。ただし、ディクショナリの

更新が行なわれる際、複数のセッションで1つのセッションだけがシーケンスプールを再度キャッシュできることが保証されているため、その際はほかのセッションは「enq: SQ – contention」イベントで待機することになる。

例えば、デフォルトで作成されたシーケンスは「20」のCACHE属性を持つため、「20」回発行されると再度ディクショナリを更新し、次の20個のシーケンスをメモリにキャッシュしなければならない。そのため、トランザクションが同時かつ大量に発生する表のキーとしてシーケンスを採用する場合は、CACHE属性の値が小さすぎるとボトルネックの原因になるので、十分大きく設定する必要がある。

当テストでは、シーケンスの「CACHE」属性を「20→10000」に変更することで、「2419→531」秒（78.1%短縮）の応答時間の性能改善を実現した（LIST18）。また、シーケンスでのボトルネックが解消されて、「latch: library cache」などの待機が増え、上位にランクされたが、CPU使用時間に比べて気にするほどの現象ではないと考えられる。

ケース⑨の改善ポイント

このテストは、リモートサーバーとの連携作業が多い状況でネットワークの負荷が高まったときに現われる現象を、待機イベントに基づいて検証したシミュレーションになる。今回のようにDBリンク経由で大量にデータを参照／追加／更新作業を行なうと、ローカルサーバーでは「SQL*Net message from dblink」での待機になるなど、ネットワークのボトルネックは「SQL*Net…」イベントに現われる。今回はリモート表をローカルに定期的にリフレッシュすることで、「1133.7→92.6」秒（91.8%短縮）の処理時間の性能改善が実現され（LIST19）、CPU使用時間を含む統計情報が安定的な数値を表わすようになる。

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse     1185      0.03       0.24          0          0          0           0
Execute 1819095    554.15    8243.49       1858  141324571    3392489     1802240
Fetch      466      0.00       0.01          0        391          0         367
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total   1820746    554.18    8243.74       1858  141324962    3392489     1802607

Elapsed times include waiting on following ev
  Event waited on                          Times   Max. Wait  Total Waited
  ---------------------------------------  Waited  ----------  ------------
  latch: cache buffers chains                1483        67         189.80
  latch: cache buffers lru chain             1822        0.7        188.13
  latch free                                 1333        0.7        144.34
  log file switch completion                  161        1.07        97.06
  buffer busy waits                          3024        1.16        89.87
  latch: undo global data                     346        0.45        39.29
  log file sync                               206        0.63        44.28
```

グループコミットで解消

LIST17 ケース⑦の改善結果

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse     1741      0.26       2.41          0          2          0           0
Execute 2681741    320.85    1745.00          0      42022       5273        3520
Fetch   2680403    107.28     465.12          0          7        274     2680403
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total   5363885    428.40    2212.54          0      42031       5547     2683923

Elapsed times include waiting on following e
  Event waited on                          Times   Max. Wait  Total Waited
  ---------------------------------------  Waited  ----------  ------------
  latch: library cache                       1347        1.31        43.32
  latch free                                  960        0.13        27.14
  enq: SQ - contention                        895        0.15        16.11
  cursor: pin S                               712        0.15         8.13
  latch: library cache pin                    222        0.12         5.58
  enq: TX - row lock contention                23        0.43         1.13
  buffer busy waits                             4        0.52         0.98
```

シーケンスのメモリキャッシュ量の調整で改善

LIST18 ケース⑧の改善結果

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse       99      1.09      13.85          0          0          0           0
Execute  27099     22.96     207.14        137    2159177        281         232
Fetch    27027     48.23     234.80        137    3024000          0       27027
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total    54225     72.29     455.80        274    5183177        281       27259

Elapsed times include waiti                   :
  Event waited on                          Times   Max. Wait  Total Waited
  ---------------------------------------  Waited  ----------  ------------
  latch: cache buffers chains                 388        0.13        12.33
  cursor: pin S wait on X                     530        0.02         5.72
  latch free                                   40        0.10         1.17
  read by other session                       619        0.08         1.02
  db file sequential read                      28        0.00         0.03
  db file scattered read                       27        0.01         0.04
```

リモート表をローカルに周期コピーで待ち時間の改善

LIST19 ケース⑨の改善結果

＊　　＊　　＊

以上、パート1では、いくつかの例からOWIの診断／分析のパターンを見てきたが、いかがだっただろうか。やはり慣れない部分もあったかもしれないと思う。新しいコミュケーション術を身に付けるには、その仕組みの理解と若干の試行錯誤から得られる生の知識および経験が必要になる。その仕組みについては、パート2で領域ごとに分けてコンセプトとフローを説明しているので、引き続き参考にしていただきたい。最後まで読み終わってから、再びパート1のシナリオ別のケースに戻ってみると、統計データがより鮮明に見えてくることだろう。

DBM

SPECIAL FEATURE 2

PART 2

パフォーマンス診断と分析に活かす効果的なOWI活用の実践

OWIで見るOracleの内部動作のポイント

アプリケーションプログラムの開発では、トラブル時にデータの変更内容や各コンポーネントの実行履歴を追跡するため、何らかのトレースを残すことが一般化されている。同じくOracleデータベースの安定運用を目指すにあたって、稼動履歴を記録することは最初にやるべき作業である。より詳細な稼動履歴を収集／分析することで、システムに対する理解度を一層高めることになり、今まで隠れていた部分も見えてくる。パート2では、稼動履歴の見方に役立つ、Oracleの内部動作のポイントを説明する。

OWI分析用の履歴データ

例えば、高速道路のある場所で10分おきに写真を撮っているCCTV（監視システム）があって、そこで事故が起きたらどうなるだろうか。警察官は、当事者からの説明を元に事故結果と事故発生タイミングの前後のCCTV写真で裏付けをとって、事故の原因を判断するだろう。さて、そこでもしCCTVの写真の間隔が10分ではなく1秒だったらどうなるだろうか。写真だけで事故原因のほとんどが説明できるはずだ。場合によっては、当事者による説明をも覆すことになるかもしれない。

DBシステムの運用においても同じことが言える。システムの稼動データの収集が重要だということはよく言われるが、実際に必要な場面に遭遇しないとなかなか実践できないものである。しかし、図1のように突然処理時間が遅くなった場合でも、稼動履歴データの有無によって対応がまったく異なる。稼動履歴データがあれば、その客観的な数値から誰もが理解できる判断や説明が可能になるのである。

図2は、あるタイミングから応答時間が遅くなったシステムの稼動履歴データだが、何がトリガーとなって性能が低下しているのかお分かりだろうか。データから事実を並べてみると、次のようになる。

- 通常80程度の接続数「logons current」が400まで増加した
- OSの空きメモリ「Free Memory（MB）」がほとんどなくなった
- I/O関連CPUの利用率が高くなった
- SQL解析処理での滞留「latch: library cache」（参照：「共有プールとOWI」セクション）が最大40秒以上増加した
- 通常20程度のアクティブセッション「active sessions」が100を超えた

滞留の原因はSQL解析処理にあるが、接続数とOSの空きメモリが深く関わっているように見える。実際の診断／分析で、やはり接続数の急増がトリガーになってOSメモリが不足し、SQL解析処理を行なう共有プールのページング現象が発生したことを確認できた。

このように、稼動状況の履歴データはトラブルのトリガーの診断／分析に大いに役立つ。データが細かくなるほど、その正確性が高まることは言うまでもない。また、そのトレンドを分析することで、事象のパターンや傾向が分かるようになり、潜在リスクの把握や事例の蓄積という面でも活用可能になる。さらに、運用状況をそのまま数値化するので、ほかの日の運用状況との客観的な比較も可能になる。

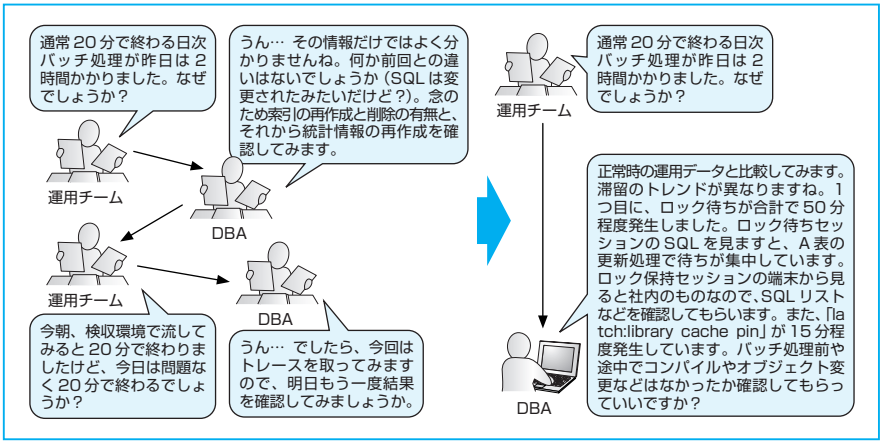


図1 稼動履歴データの有無の違い

OWI観点での運用履歴データ収集

パート1でも少しコメントしたが、OWI観点の運用履歴データを収集／記録する方法をいくつか紹介しよう。まず、最も詳細

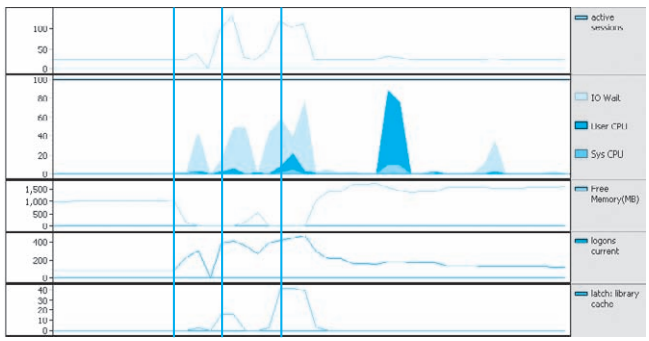


図2 稼動履歴データの例

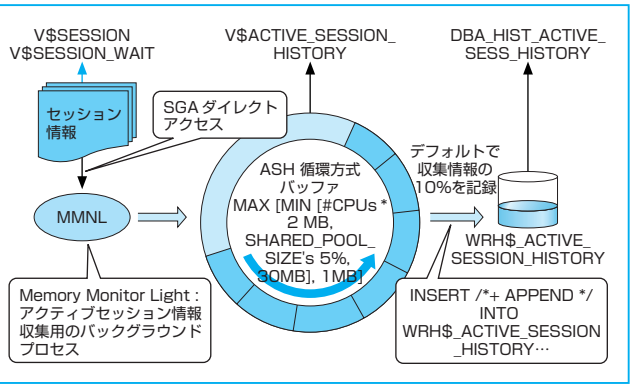


図3 Oracle内部のアクティブセッション履歴データの収集／記録のイメージ

なデータを収集するには「event 10046[注1]」によるトレースをとる。これによって、プロセス（セッション）単位でSQLごとの統計情報／イベントの詳細を確認できる。しかし、情報解析にかかる手間とデータベースに対する高い負荷のためすべてのセッションを対象にするのは現実的でなく、常時情報を取得するには向いていない。

最も一般的に使われているSTATSPACKを使ったデータ収集は、実装の手軽さと、よくまとまったレポートのおかげでシステム全般の情報が一目で分かるようになり、タイムベースの稼動指標や高負荷のSQLなど、効率の良い解析ができるという強みを持っている。その反面、セッションレベルの状況とあるタイミングのシステム状況の把握ができない、データの取得／格納時にある程度の負荷はやむを得ないため取得間隔を短くできない、といったところが弱点となる。

次によく使われる方法として、定期的なSQL発行によるOWI関連データ（パート1の図3を参照）の収集がある。こちらは、収集対象を自由自在に定義できるメリットがある反面、同じくシステムへの負荷と取得／記録間隔の間でジレンマが生じる。また、データベースハングなどシステムが致命的な状況になった場合に、接続さえできなくなる可能性がある。

注1：通常のSQLトレース情報に、バインド変数、待機イベントの詳細情報を追加で記録する。

最後に最も理想的で、Oracle 10gでも採用されている方法として、SGAの直接読み取り（SGAダイレクトアクセス）を使う手法がある（図3）。この方法には、システムへの負荷を最小限に抑えながら細かいデータの収集ができるというメリットがある。ただ、その実装が難しく、実装方法によって部分的なデータしか取れない可能性がある。

このように各収集方法にはそれぞれ長所と短所があるので、システム運用のニーズと要件に合わせて実装する必要がある。筆者の個人的な意見であるが、どの方法かを選ぶにあたっては、システム／セッション／SQLレベル情報の有無と連携分析のしやすさ、データの細かさ（収集間隔）、トレンド化できるか（しやすいか）、システムへの低負荷、といった基準を設けておくと良いと思う。

アクティブセッションとOWI

Oracle内部で、アクティブセッションを中心に稼動履歴を記録するのはなぜだろうか。今まで認識していなかった人もいると思うが、アクティブセッションはマルチユーザーシステムの安定状況の良し悪しを判断する最適の指標で、統計指標（STATS）、待機イベント（滞留状況）、OS指標（CPUやメモリの使用状況）をすべて反映している。

「アクティブ」ということは、現在CPUを使っているか、またはリソースを使うため待っている状態（アイドルイベント待ちは除外）で、「SELECT * from V$SESSION WHERE STATUS = 'ACTIVE'」のSQLで参照できる。場合によっては、バックグランドプロセスは含めずに使うこともある（ACTIVE SESSIONS = BACKGROUND PROCESSES + WAIT PROCESSES + ACTIVE USER PROCESSES）。

特に図4のように、アクティブセッションは全体の流れの中で滞留現象が発生すると、流出が遅れたりできなくなったりして、短時間である領域に溜まっていく。アクティブセッション数の増減は、アクティブセッションの流入／流出状況と滞留の発生量と密接に関係している（図5）。

システムの性能を維持するためには、アクティブセッションの流出数が流入数を超えないようにする待機イベントをモニタリングする必要がある。つまり、アクティブセッション数が増加し始めるところに注意するということだ。アクティブセッション数はCPU使用率からも影響を受けるが、「CPU = 100%」でなければその影響はほ

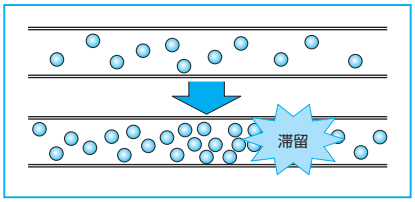


図4 アクティブセッションと滞留

特集2

OWIによるOracleの性能改善と障害対策

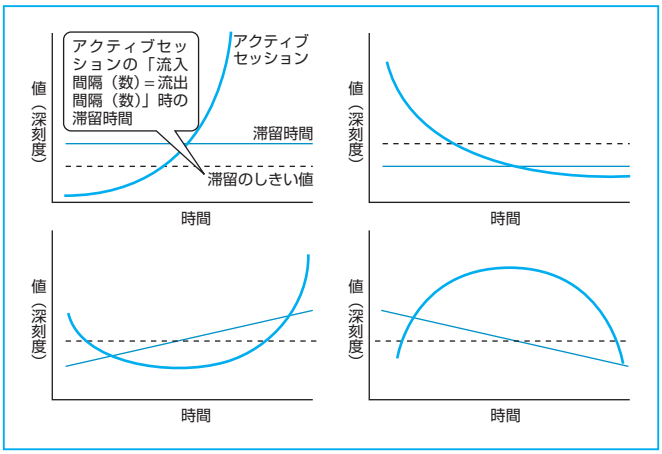


図5 アクティブセッションと滞留の相関関係

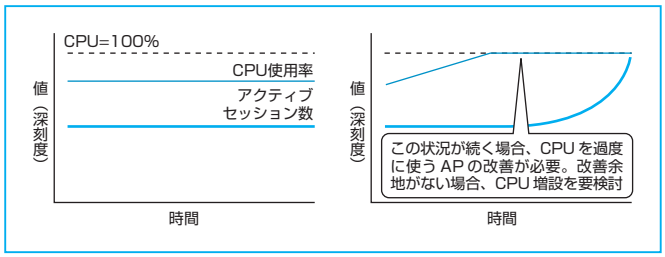


図6 アクティブセッションとCPUの相関関係

とんどないと考えて構わない（図6）。

リアルタイムモニタリングおよび稼動履歴の診断／分析でアクティブセッション数が急激に増え始めたときは、その原因を突き止めるためのアクションを起こそう。最初に確認するべき部分は、同タイミングの待機イベントと統計指標の推移との連携分析である。類似の動きを表わすイベントがそのトリガーであり、統計指標がその結果になる可能性が高くなる。

次に同時点のセッションの詳細と実作業（SQL）を確認する。指標の推移を説明できるポイントになるか、逆にもっと確認すべき情報が浮上することもある。

次に、同時点のセッション間の関係を分析する。マルチユーザー環境では、システムそのものの性能よりプロセス間の競合（例えばロック制御や同一ブロックの変更、同じオブジェクトの取り合いなど）が性能低下の主な原因になる。併せて、トップSQL[注2]、正常時でのトレンド比較、特定区間の集中分析などを行なうと、より確実な原因追求が可能になる。

Oracleアーキテクチャにおける OWI

プロセス（アクティブセッション）とOracleリソース（メモリやディスクなど）の処理過程を数値化したものが統計指標と待機イベントである。より分かりやすい説明のため、共有プール、バッファキャッシュ、I/O、RAC環境に分けて、各領域でのプロセスの処理の流れと併せて各指標の具体的な意味を説明していく。

共有プールとOWI

Oracleで最も処理負荷が高い部分は、I/Oとメモリになる。メモリでのデータ制御において、検索や割り当て、同期化が効率良く行なわれるような仕組みを用意しているが、使い方によってその性能の良し悪しが極端に分かれる場合がある。

バッファキャッシュは業務データに対してブロック単位で制御されるが、共有プールでは特に一定の決まったサイズではなくオブジェクトに合わせて適切なサイズを割り当てることで制御される。さまざまなオブジェクトにメモリの割り当てが進むと残っているメモリのサイズがどんどん小さくなっていくため、共有プールではオブジェクトの共有とメモリの断片化の回避が重要なポイントになる。

図7を参考にして、共有プールでのメモリ関連制御の概要を見ていこう（番号はイベントの発生順ではない）。

❶ユーザーから依頼されたSQLに対してハッシュ値を算出して、該当するハッシュ値を管理しているバケット[注3]にアクセスするが、ここで該当バケットを管理しているラッチに競合が発生すると、「latch: library cache」イベントで待機する。続いてバケットで管理するチェーンで該当するSQLのハンドル（LCO[注4]に対するメタ情報およびポインタ）があるか確認

❷ライブラリキャッシュで実行対象のSQLが見つからないと、SQLの解析処理の結果を保存するため新しいメモリを割り当てるが、ここでプロセス間競合が発生すると、「latch: shared pool」イベントで待機する。図7では、14KBが必要な解析結果のため、16KBのメモリがLCOとして使われ、2KBの新しいフリーメモリ（断片化）が発生する

❸SQLの解析処理の間、共有もしくは排他モードで「latch: library cache lock」イベントで待機する。当イベントは解析結果の定義情報が変更されないように該当ハンドルを保護する

注2：合計所要時間、実行平均所要時間、CPU使用時間、待機時間、実行回数、物理読み取り、論理読み取りが大きい順。

注3：同一ハッシュ値の集合体。

注4：Libray Cache Object。SQLテキストなど実行に必要な情報の実体。

❹SQLの実行処理の間、実行計画などの実行情報が変更されないよう、共有もしくは排他モードで「latch: library cache pin」イベントで待機する。このイベントは表の変更などのDDL、プロシージャのコンパイル作業によって③の滞留現象と同時に急増するので、運用時間帯でのオブジェクトの変更には十分な注意が必要である

❺SQLの解析処理でオブジェクトの定義情報にアクセスする間、該当オブジェクトが変更されないよう、「latch: row cache objects」イベントで待機する

❻オブジェクトの定義を変更するとき競合が発生すると、「row cache lock」イベントで待機する

❼ディクショナリからメモリにシーケンスを読み込む際に競合が発生すると、「enq: SQ ー contention」イベントで待機する

パート1のケース①は、上記❶の処理でボトルネックになった現象になる。CPU数より大きく、かつ最小の素数を「library cache」ラッチとして使うため、同時解析処理で最初に競合が発生しやすい部分である。ハンドル（異なるSQL）の数が多いほどSQLの検索時間が長くなるため、ライブラリキャッシュの断片化を最小化する（SQLの共有化）ことが重要なチューニングのポイントになる。❷の「shared pool」ラッチの競合でも同じことが言える。パート1のケース⑧は、上記❼の発生イメージを参照してほしい。

バッファキャッシュとOWI

Oracleは物理的なディスクI/Oを最小限にするため、よく使われるデータブロックをメモリに保持しようとする。そこで、メモリの検索や割り当て、リリースなどで発

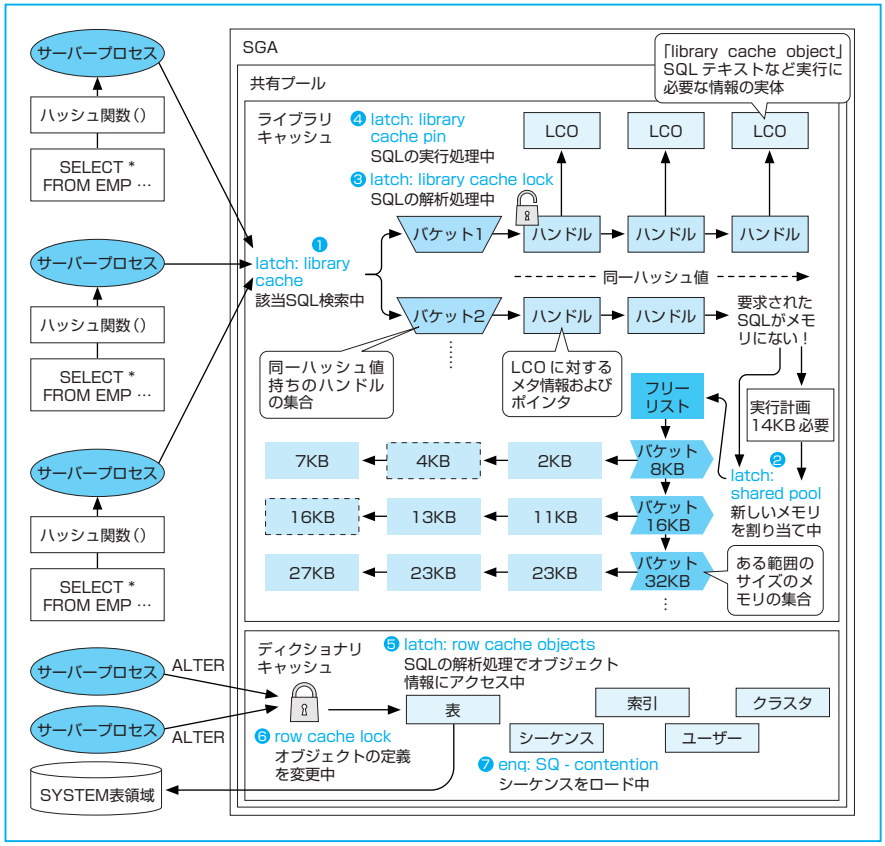


図7 共有プールでのOWIイメージ

COLUMN

Oracleの同期化メカニズム～ラッチとロック

Oracleでは大量の同時アクセスユーザーによる同時作業処理は当然とされているが、その実現となると、そう容易なことではない。数個～数千個のプロセスが同時に同じリソースにアクセスするとき、Oracleはその同時要求を正確に処理するため、ラッチとロックという同期化メカニズムを使っている。この2つは、一貫性を保つため、Oracleのリソースに対して排他制御を行なうという点では共通しているが、まったく別のオブジェクトである。

ラッチは負荷を最小限に抑え、早く動作することを目的に単純なコマンドを使って実装されている。実際に保護するリソースはSGAで、SGAにアクセスするプロセスは必ずアクセスしようとする領域を管轄するラッチを獲得した後にだけアクセスが許可される。また、Oracleはラッチを獲得する順序を保証していない。順序を保証するためには複雑な同期化アルゴリズムが必要となるので、ラッチ本来の目的が果たせなくなるからである。

一方、ロックが保護するリソースは、表領域やテーブル、トランザクションなどデータベース全体と言える。プロセスは、ロックの獲得に失敗するとキュー（queue）で管理されて、要求した順にロックを獲得する。ラッチと違い、ロックではデッドロックが発生する可能性が常にある。ロック獲得で待機しているプロセスは、一定時間経過するとデッドロックが発生しているかどうか検査する。もしデッドロックが確認できると、実行中のトランザクションはロールバックされ、アラートログファイルにデッドロック情報が記録される。また、ロック自体もSGAの共有プールに存在する一種のメモリ構造体なのでラッチによって保護されている。

このように、ラッチとロックでは実装方法や獲得方法、保護するリソースなどさまざまな点で違いがある。ある作業を実行するためには、ラッチとロックを獲得しなければならないが、これ自体は性能問題とは無関係である。しかし、数十GBに達するバッファキャッシュや数十TBに達するストレージといった、ラッチやロックを利用して管理しなければならないリソース量と、数千にものぼる同時アクセスユーザーによって、ラッチとロックを獲得する過程で競合が発生する確率も高くなる。DB管理者として、このような競合により発生する性能問題を解決するには、ラッチとロックの動作方式だけでなく、具体的なリソースレベルでどのような種類のラッチとロックを使うのかという詳細な知識が必要になる。

特集2

OWIによるOracleの性能改善と障害対策

生するオーバーヘッドを効率良く制御するとともにデータの整合性を保つため、「バケット→チェーン[注5]→バッファヘッダー[注6]」のハッシュチェーン構造を使っているが、その仕組みがOWIとしてそのまま現われる。その制御のイメージを、図8を基に説明していく。

❶サーバープロセスがリクエストするデータブロックと、ブロッククラス[注7]に基づくハッシュ値を使って該当する「バケット」を決定し、そのバケットを保護しているCBC（Cache Buffers Chains）ラッチを、読み取り作業の場合は共有モードで、変更作業の場合は排他モードで獲得する。ここで競合が発生すると、「latch: cache buffers chains」イベントで待機する。バケットで管理するチェインで該当するバッファヘッダー（BH）があるか確認する

❷バッファキャッシュに該当ブロックが存在しない場合、ディスクから読み込み、キャッシュされることになるので、空き領域（フリーバッファ）を確保するためCBLC（Cache Buffers Lru Chain）ラッチを排他モードで獲得する。ここで競合が発生すると、「latch: cache buffers lru chain」イベントで待機する

❸フリーバッファが見つからないと、サーバープロセスはDBWRに変更済み（ダーティ）バッファをディスクに書き出して、フリーバッファを確保するようにリクエストする。ここでDBWRによってフリーバッファが確保されるまでサーバープロセスは「free buffer waits」イベントで待機する

❹ディスクからデータブロックを読み込むとき、サーバープロセスは該当バッファに対して排他モードでバッファロックを獲得する。このとき、他プロセスが同じブロックを同時に読み込もうとする場合、後続のサーバープロセスは該当ブロックがロードされるまで「read by other session」イベントで待機する

❺続きの作業を実行するため、バッファキャッシュの該当ブロックに対して共有あるいは排他モードでバッファロックを獲得。ここで競合が発生すると、「buffer busy waits」イベントで待機する

❻DBWRによってディスクに記録中のバッファに対してバッファロックを獲得しようとすると、「write complete waits」イベントで待機する

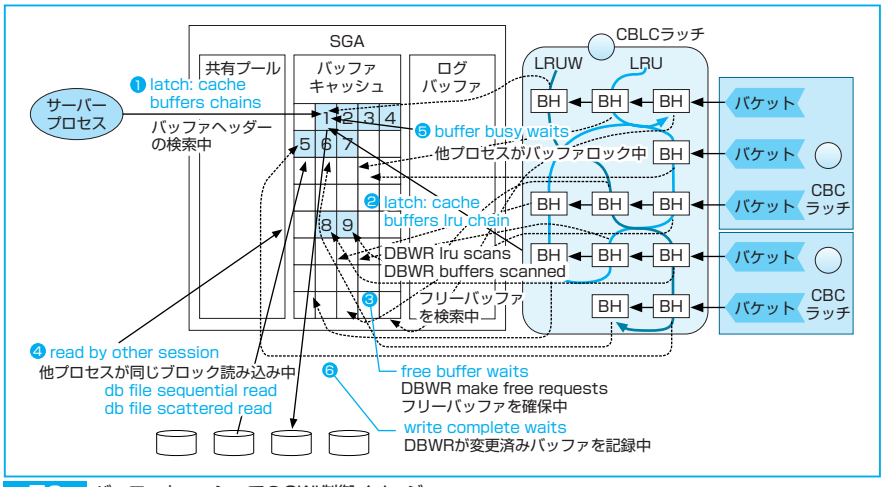


図8 バッファキャッシュでのOWI制御イメージ

パート1のケース③は、上記❶の処理でボトルネックになった現象である。特定CBCラッチにアクセスが集中することは当該ラッチが管理しているバッファヘッダーへのアクセスが多いことを意味する。多くのプロセスが特定ブロックに集中的にアクセスする場合（ホットブロック現象）と、広い範囲の検索処理が、一般的な原因として考えられる。

注5：同じハッシュ値を持つオブジェクトのつながり。
注6：バッファへのポインタとメタ情報を持つ。
注7：ブロックの種類。例）1：データブロック、2：ソートブロック……。

COLUMN

SQL解析処理のフロー

SQLが実行される際、はじめに解析（Parse）が行なわれる。この解析処理では、Oracle内部でどのようなことが起きているのか、順に見ていこう。

① SQL文の実行要求があると、Oracleは権限／文法チェックなどを行なった後、ライブラリキャッシュに同じSQL文が存在するか確認する
② 同じSQL文が存在しない場合は、まず共有プール内にSQL文をキャッシュするための領域を確保。この領域はチャンクと呼ばれ、フリーチャンク（未使用チャンク）は、フリーリストによって管理されている
③ もし最適なサイズのフリーチャンクが存在しないと、より大きなフリーチャンクを探し、ここから必要な大きさを切り取った後、残りのチャンクは再度フリーチャンクとしてフリーリストに登録される
④ すべてのフリーリストを検索しても適切な大きさのフリーチャンクが見つからないと、LRUリストを検索し、再利用可能なチャンクを探す
⑤ LRUリストを検索しても適切な大きさのチャンクを確保できなければ、Shared Pool内の未使用メモリ空間を割り当てる
⑥ ここまでの処理がすべて失敗すると、ORA-4031エラーが発生する
⑦ 適切なフリーチャンクが見つかると、実行計画（Ececution Plan）を生成

この後は、実行（Execute）、フェッチ（Fetch）と続く。①で同じSQL文が見つかった場合は、すぐにSQLが実行される。このケースをソフトパース（Soft Parsing）と呼び、「②〜⑦」までの処理をハードパース（Hard Parsing）と呼ぶ。ソフトパースに比べ、ハードパースの場合はさまざまな作業が発生し、Oracleに負荷がかかっていることが想像できるだろうか。この負荷を減らすためには、バインド変数を利用するなどしてSQLを共有することが重要になる。

テストでは検索処理でアクセスされるブロック数を最適化して、それを保護しているラッチへのアクセスを最小化したことになる。つまり広範囲の検索処理は、検索および抽出など作業量と特定リソースへの競合の両方の面で悪影響を与える。ホットブロックは、そもそもそのブロックにアクセスしないか該当ブロックのデータを適切に分散して解消する（**コラム**「データの分散効果」参照）。

また、パート1のケース④では上記❺の処理のボトルネック現象が現われていた。明示的に行ロックではなくても、バッファキャッシュではバッファロックになる。具体的な原因を調査するため、イベントのパラメータ[注8]（P3）から競合が発生しているブロックのクラスを確認するとともに、実行SQLを確認することで競合のタイプを把握することが重要である。

例えば、「INSERT/INSERT」の競合はセグメントの低いフリーリスト設定に起因することが多く、ASSM適用もしくは「freelists、 freelist groups」の調整が必要になる。「UPDATE/UPDATE」は更新対象のブロックの競合になるため、ジョブ運用の改善、処理ロジックの変更、集中ブロックの分散（**コラム**「データの分散効果」参照）などを考慮しなければならない。

注8：イベントが発生しているリソースなどの詳細を表現。「v$event_name」で各イベントのパラメータ内容が確認できる。例）db file scattered read：P1＝file#、P2＝block#、P3＝blocks

I/OとOWI

Oracleのすべての機能は、最終的にいかに効率良くデータをディスクに格納し、いかに効率良く読み取るかに関するものなので、数多くの性能問題がI/Oと深く関連している。I/Oと関連するOracleの性能問題を診断／分析するためには、Oracleプロセスが次のような複数のレイヤを経由してI/O作業を行なっていることを理解する必要がある。

A. アプリケーションレイヤ：SELECT / INSERT / UPDATE / DELETE / TRUNCATE / DROP など
B. Oracleメモリレイヤ：バッファキャッシュ、PGA など
C. Oracleセグメントレイヤ：データファイル、一時ファイル、表領域、セグメント など
D. OSデバイスレイヤ：非同期I/O、ダイレクトI/O、ローデバイス、RAID など

I/Oの性能問題の原因を把握し改善案を探るには、アーキテクチャと効率面から考えて「A→B→C→D」の順で診断／分析を行なうので、I/O関連の各イベントも同様に見ていく。次に関連イベントの概要（発生順ではない）を説明するので、**図9**をイメージしながら読み進めてほしい。

COLUMN

ハードパースのリスク

共有プールを効率的に運用するため必ず確認しなければならないのが「ハードパース」の発生状況である。ライブラリキャッシュに解析結果が保存されていない新しいSQLに対して行なう解析処理をハードパースと言うが（**コラム**「SQL解析処理のフロー」参照）、次の3つの観点でパフォーマンスに悪影響を与える。

① 実行計画を作成する際、CPUに余計な負荷を与える。例えば、各々2つの索引を持つ3つの表に対するSQLで可能な実行計画を単純に算出しても、1458種類「3!（表のアクセス順序）＊27（スキャン：各表に対して表／索引1／索引2の3パターン）＊9（ジョイン方法：NESTED、HASH、SORT MERGE）」の実行パスのコストを算出しなければならない
② すべてのSQLが異なるSQLとして認識されライブラリキャッシュの検索対象になるので、解析時間が長くなる（「library cache」ラッチの保持時間の増加）
③ 各SQLのサイズに合わせたメモリに格納されるので、共有プールの断片化が進むため、フリーメモリの検索時間が長くなる（「shared pool」ラッチの保持時間の増加）。この現象は「ORA-4031」エラー発生につながりかねないので、より重大な問題になる

このようなハードパースを改善するため、CURSOR SHARINGの使用、SQLのバインド化処理、静的SQLの使用が考えられるが、**表A**に示す簡単なテスト結果からそれぞれの効果を確認してほしい。「SQLのバインド化処理、静的SQL」で「parse count（total）」と各タイム統計が激減したことに注目しよう。

環境構成と測定結果	リテラルSQL	CURSOR SHARING	バインドSQL	静的SQL
SQL文	execute immediate 'select count(*) from dual where dummy = to_char(''ii'')' into v_cnt;	execute immediate 'select count(*) from dual where dummy = to_char(''ii'')' into v_cnt;	execute immediate 'select count(*) from dual where dummy = to_char (:b1)' into v_cnt using ii;	select count(*) into v_cnt from dual where dummy = to_char(ii);
CURSOR_SHARING	EXACT	FORCE	EXACT	EXACT
CPU time(s)	67	60	29	28
latch: library cache(ms)	11,260	77	0	0
latch: shared pool (ms)	333	0	0	0
parse count(total)	202,883	202,396	2,613	2,557
parse count(hard)	20,566	322	347	341
parse time elapsed(0.01s)	18,099	1,548	142	258
parse time cpu(0.01s)	1,541	610	88	70

表A ハードパースの改善効果

※ テスト方法：10セッション同時に、2万回のループ処理の結果をSTATSPACKで統計情報として取得

特集2

OWIによるOracleの性能改善と障害対策

❶サーバープロセスが物理I/O コールをしてから、メモリに存在しないデータブロックをバッファキャッシュにロードするまで待機する。索引スキャンのようなシングルブロックの場合は「db file sequential read」が、全表検索のような複数ブロックの場合は「db file scattered read」イベントが発生する。通常、物理読み取りという処理で発生するが、データの転送元はディスクだけではなくファイルキャッシュやストレージキャッシュになることもある（データブロックのロード処理で行なわれるメモリの同期化制御は「バッファキャッシュとOWI」のセクションを参照のこと）

❷バッファキャッシュを使わず、永続表領域よりデータブロックを読み込む際に「direct path read」イベントで待機する。バッファキャッシュの同期化などメモリ制御でボトルネックになっている場合に有効な改善案になる

❸ダイレクトロード、パラレルDMLやパラレルCTAS（Create Table As Select…）などバッファキャッシュを経由せず、対象のデータファイルに記録する際に「direct path write」イベントで待機する。同イベントが滞留の主原因になる場合は、I/Oシステムの性能を確認する必要がある

❹バッファキャッシュで変更済みになっているメモリブロックはDBWRによってディスクに記録されるが、DBWRはI/Oコールをしてから記録が完了するまで「db file parallel write」イベントで待機する

❺設定されているソート用のメモリサイズより大きいソート作業が発生した場合、一時表領域にソートセグメントを作成して一時的にデータを保存するが、このときI/Oコールが戻るまで「direct path write temp」イベントで待機する

❻❺で一時保存したソートセグメントを読み込む際に、「direct path read temp」イベントで待機する

❼LGWRはREDOログバッファの内容をREDOログファイルに記録するため、必要なI/Oコールをしてから完了するまで「log file parallel write」イベントで待機する

❽サーバープロセスでコミットおよびロールバックが行なわれるとトランザクションの結果を保証するためLGWRはその時点までのREDOレコードをREDOログファイルに記録する。このときサーバープロセスはLGWRの記録が完了されるまで「log file sync」イベントで待機する（併せてパート1のケース⑦を参照）

❾サーバープロセスがREDOログバッファにREDOエントリを記録するとき、空き領域が見つからないとLGWRにREDOログファイルへの記録を要請する。このときREDOログファイルがいっぱいになって次のファイルへの切り替えが発生すると、サーバープロセスは切り替え作業が終わるまで「log file switch completion」イベントで待機する

❿REDOログファイルのスイッチ、チェックポイントなど制御ファイルの情報の変更が必要な処理で競合が発生する場合、該当プロセスは更新が完了されるまで「control file parallel write」イベントで待機する

主なイベントの発生の仕組みからボトルネック箇所と場面が理解できたと思う。個々の現象にはその状況に合わせた診断／分析が必要になるが、参考までに階層ごとに推奨される改善ポイントの概要をまとめてみよう。

アプリケーションレイヤ

不要なI/Oを最小化することを最終目標として、アプリケーション（SQLを含む）を効率良く作成する。非効率的なアプリケーションを修正せずシステムをチューニングすることはほとんど不可能で、効果は期待できない。Bツリー索引、パラレルクエリ、パラレルDML、Nologgingオプ

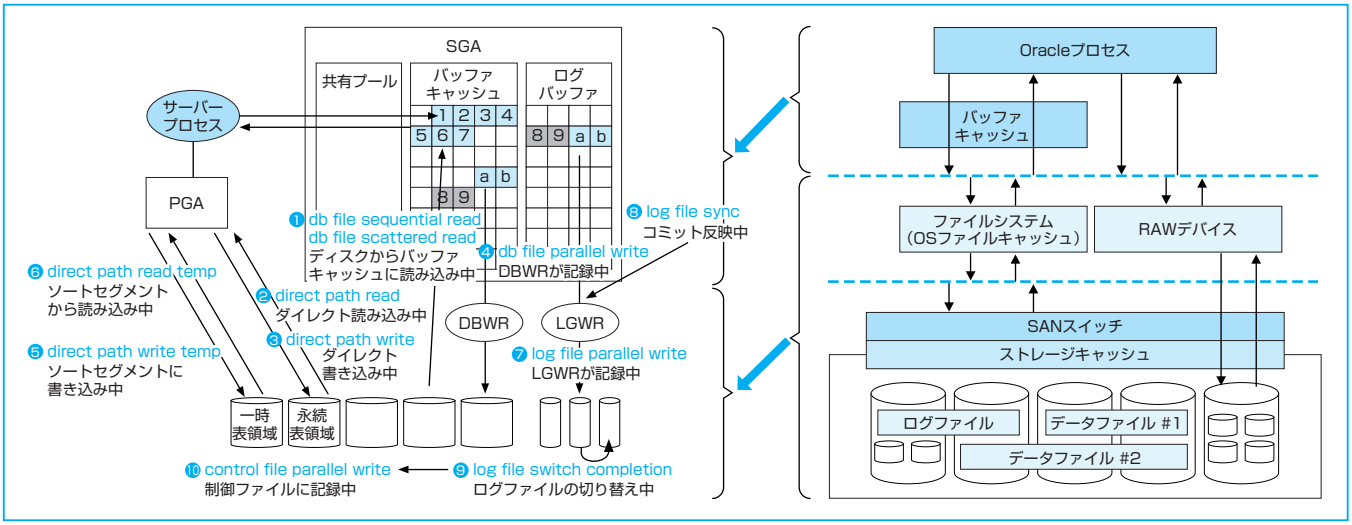


図9 I/OでのOWIイメージ

ション、ダイレクトロード、ダイレクト読み取り、分析（Analytic）関数など、I/Oを効率的にさせるさまざまなOracle機能を使って、無駄なI/O負荷を減らすことが可能。またデータの性質によって、クラスタや索引構成表[注9]、パーティション、ビットマップ索引を適切に使うこともアプリケーションの側面から考慮すべきである。

Oracleメモリレイヤ

頻繁に使われるブロックをメモリにキャッシュしておいて、ディスクI/Oを最小限にする。バッファキャッシュを効率良く使うために、次のような機能の適用を検討する。

多重バッファプール

頻繁に使うブロックとめったに使われないブロックを分けてメモリにロードすることによって、バッファキャッシュの効率化を図る。システムで常時使われるオブジェクトには「デフォルトバッファプール」を、比較的よく使われる小さいオブジェクトには「キープバッファプール」を、めったに使われない大きいオブジェクトには「リサイクルバッファプール」を設定することで、メモリの無駄使いを防ぐことができる。特定セグメントを完全に常駐させるため、「キープバッファプール」のサイズを決める際には、CRブロックが占める領域も考慮する必要がある。

ブロックサイズの変更

各セグメントの特性に合わせたブロックサイズを適用することで、性能改善効果を期待できる。例えば、レコードのサイズが大きく、全件検索が多い表には大きいブロックサイズを適用することが望まれる。

ダイレクトパスI/O

メモリにデータをロードする必要がない大容量のデータ処理を行なう場合は、バッファキャッシュを経由しないダイレクトパスI/Oを使う。SGA領域を経由しないた

注9：索引のキーを基準に表のデータを格納したセグメントで、索引を経由するROWID検索のオーバーヘッドがない。

COLUMN

データの分散効果

同じ処理を行なっても、セグメントに格納されているデータの分散状態によって発生する滞留は異なる。データの集中が原因で発生する性能問題は、通常該当データを分散することで解消する。ここではデータの分散方法（**LISTA**）による効果を測定してみた（**表B**）。

「PCTFREE調整」では、データが完全に分散されてブロック競合（buffer busy waits）は解消されたものの、ラッチ（latch: cache buffers chains）による滞留が増加した。「ハッシュパーティション化」では、ブロック数を増やさないでデータだけの分散になったため、ラッチ待機を増加させずに、ブロック競合が多少解消されている。「アプリケーションの特性を考慮したデータ分散」では、更新処理に合わせてデータを完全に分散したため、ラッチ待機とブロック競合が最小化された。しかし、更新パターンが変わった場合、再対処が必要になる可能性がある。「小さいブロックサイズ（2KB）」適用では、ロック競合が多少解消されたものの、ラッチ待機も多少増えているとともにフルスキャンに対する性能低下現象の不安要素も新しく出てくる。

このテストではアプリケーションの特性に合わせた対処が最も効果的であったが、どんな状況でも常に完全な対応方法はなく、問題の原因に合わせた対処をとる必要があると考えられる。

```
-- 表作成（標準ブロックサイズ=8K）
create table t_blockcontention_test ( id number, name char(700) );

-- データ作成
begin
 for idx in 1 .. 100 loop
  insert into t_blockcontention_test(id, name) values(idx, ' ');
 end loop;
end;
/

-- データ更新
create or replace package body test is
 procedure blockcontention(idx in number) is
 begin
  for ii in 1 .. 1000 loop
   for jj in 1 .. 10  loop
    update t_blockcontention_test set name = ' ' where id = 10 * (jj - 1) + idx ;
    commit;
   end loop;
  end loop;
 end;
end test;
/

-- ジョブ（10セッション）登録
for idx in 1 .. 10 loop
 dbms_job.submit(job=>job_no, what=>'test.blockcontention('||idx||');');
end loop;
```

LISTA テストスクリプト

テストケース（100件のデータを10セッションで10件ずつに分けて更新）	ブロック数	所要時間	CPU時間	buffer busy waits（time／回数）	latch: cache buffers chains（time／回数）	備考
デフォルトで表作成	10	167	39	49／9402	4／824	上記LIST A
PCTFREE調整	100	108	47	1／665	20／772	「PCTFREE=10」で、1件/ブロック
ハッシュパーティション	12	99	32	8／1248	1／116	"partition by hash（id）partitions 5;"
アプリケーションの特性を考慮したデータ分散	10	80	30	0／6	1／12	partition by list（id）
小さいブロックサイズ（2KB）	50	152	43	14／2994	5／741	"alter system set db_2k_cache_size = ... ; create tablespace ... blocksize 2K ... ;"

表B データ分散効果測定

め、メモリの割り当てや検索、メモリ共有のための同期化作業がなくなり、その分パフォーマンスが良くなる。パラレルクエリやパラレルDMLなどは永続表領域に対して、ソート作業は一時表領域に対してダイレクトパスI/O作業を行なう。ただ、CPUとメモリ使用の増加と変更済みのメモリブロックに対するチェックポイント作業の増加によって、逆に性能低下現象が発生することもあるので、十分に注意する必要がある。

Oracleセグメントレイヤ

セグメントへの領域割り当て作業を行なう際に発生するSTロックは、LMT（Locally Managed Tablespace）とASSM（Automatic Segment Space）を適用することでエクステントおよびセグメントに関する性能問題はほとんど解消される。また、大容量の表はパーティション化することで、検索処理の効率化を図れる。さらにパート1のケース②、③のように、索引とデータ構成の調整でアクセス範囲を最小化することも考えられるだろう。

OSデバイスレイヤ

Oracleは、I/O作業の性能向上のためにできるだけ非同期I/Oを使う傾向にある。また制御ファイルやREDOログファイルは、リカバリが可能な最小限の数を持つことが性能改善につながる。このほか、ファイルを物理的に分散させて、ディスク間の競合（データファイル間、REDOログとアーカイブログ間など）を避けることも望まれるが、I/O構成の変更は高いコストを要するため、できるだけ最終手段として考えるべきである。

RAC環境におけるOWI

1つのデータベースを複数のインスタンスで管理するためには、どのような技術が必要だろうか。プログラマの観点から見ると、そのキーワードになるのは間違いなく「データの同期化」になる。シングルインスタンス環境の中でもデータの整合性のため、エンキューとラッチ（コラム「Oracleの同期化メカニズム、ラッチとロック」参照）を使っているが、マルチインスタンス

COLUMN

マルチバッファプールの効果

バッファキャッシュのサイズが小さすぎると、物理I/Oの増加で性能低下が発生する。特にバッファキャッシュを多く使う全表検索でその影響が深刻になるが、その場合マルチバッファプールを適用することでバッファキャッシュをより効率的に使うことができる。データの特性とアクセス頻度を考慮し、どのバッファプールを使うか事前に定義する。例えば、めったに使わない表の全データを、一度の全表検索でバッファキャッシュに保存しておく必要はない。

表Cのテスト結果を見ると、アクセス頻度が低い表「t_multiple_buffer_cache_2」のデータを「RECYCLE」バッファで処理することで、よくアクセスされる表「t_multiple_buffer_cache_1」での「db file scattered read」待機が24.5％（3.78秒）減っている。面白いのは、アクセス頻度が低い表で発生する「db file scattered read」待機にも26.4％（4.25秒）の減少効果があることだ。周りの状況によってその効果は少し異なるが、マルチバッファプールの使用は、①アクセス頻度が高いデータをメモリに常駐させることで物理I/Oを減らす、②アクセス頻度が低いデータに割り当てられたメモリはできるだけ早めに再利用することでメモリの無駄を最小化する、③各バッファプールごとに別途用意されている「latch: cache buffers lru chain」を使うことでラッチに対する競合を減らす、といった面で性能改善に大いに役立つ。

全表検索の性能改善にはI/O単位「DB_FILE_MULTIBLOCK_READ_COUNT」の調整や、大きいブロックのサイズの適用も効果的なので、併せて検討することを推奨する。

環境構成と測定結果	マルチバッファプールを未使用	マルチバッファプールを使用
表	「t_multiple_buffer_cache_1」はよくアクセスされるが、「t_multiple_buffer_cache_2～5」へのアクセス頻度は低い	
	「t_multiple_buffer_cache_1～5」各表のブロックサイズの合計 ≒ 16MB	
	t_multiple_buffer_cache_1 BUFFER_POOL DEFAULT t_multiple_buffer_cache_2 BUFFER_POOL DEFAULT t_multiple_buffer_cache_3 BUFFER_POOL DEFAULT t_multiple_buffer_cache_4 BUFFER_POOL DEFAULT t_multiple_buffer_cache_5 BUFFER_POOL DEFAULT	t_multiple_buffer_cache_1 BUFFER_POOL DEFAULT t_multiple_buffer_cache_2 BUFFER_POOL RECYCLE t_multiple_buffer_cache_3 BUFFER_POOL RECYCLE t_multiple_buffer_cache_4 BUFFER_POOL RECYCLE t_multiple_buffer_cache_5 BUFFER_POOL RECYCLE
DB_CACHE_SIZE	32MB	16MB
DB_RECYCLE_CACHE_SIZE	0	16MB
DB_KEEP_CACHE_SIZE	0	0
セッション	「セッション1～5」は「t_multiple_buffer_cache_1～5」を同時に各々フルスキャンする	
CPU time(インスタンス)	2.00	2.00
db file scattered read(インスタンス)	80.00	60.00
db file sequential read(インスタンス)	8.00	5.00
CPU time(セッション1)	0.34	0.26
db file scattered read(セッション1)	15.40	11.62
db file sequential read(セッション1)	1.69	0.04
CPU time(セッション2)	0.35	0.17
db file scattered read(セッション2)	16.12	11.87
db file sequential read(セッション2)	1.08	0.10

表C マルチバッファプールの効果測定

※ インスタンスレベルの結果はSTATSPACKから、セッションレベルの結果はトレースファイルのサマリーから抽出した値

の構成にはインスタンス間の整合性の基本要件が加わる。

表1からも分かると思うが、インスタンス間の同期化を実現するため、シングルインスタンスでは比較的シンプルだったイベントがRAC（マルチインスタンス）環境では倍以上に増えている。それほど制御の複雑性が増したことを示している。

図10を基に、RAC環境での読み取り処理をシンプル化した流れで追ってみる。

❶ローカルノードのプロセスAが特定ブロックをCRモード（一貫性読み込みモードで、変更中のブロックに対して過去のイメージを作成する）で読み取りを始める。該当ブロックがローカルキャッシュに存在する場合、ローカルキャッシュから読み取る。存在しない場合は、該当ブロックのオーナーシップを持つマスターノードにブロックの転送を要求し、「gc cr request」イベントで待機する

❷ブロックの要求を受けたマスターノードは、ブロックの最新イメージを持っているノードを確認する。どのノードもまだキャッシュを持っていない場合は、ローカルノードにその情報を伝え、該当ブロックに対するディスクI/Oの読み込み権限を与える。ここでローカルノードでは「gc cr grant 2-way」イベントが発生する。マスターノードがブロックの最新イメージを持っている場合は、そのイメージをローカルノードに転送（「gc cr block 2-way」イベント発生）。ほかのノードがブロックの最新イメージを持っている場合は、そのホルダーノードに転送要求を伝達する

❸ホルダーノードは該当ブロックをローカルノードに転送。ここでローカルノードでは「gc cr block 3-way」イベントが発生する

実際の処理ではブロックの要請モード、最新ブロックの状態、ネットワークやプロセス状態によってローカルノードで発生するイベントは異なるが、イベント名だけで、同期化制御の流れやローカルプロセス

シングルインスタンス	RAC
db file sequential read	gc cr request, gc current request, gc cr grant 2-way, gc cr block 2-way, gc cr block 3-way, gc current grant 2-way, gc current block 2-way, gc current block 3-way, gc cr block lost, gc current block lost, ⋮
buffer busy waits read by other session	gc buffer busy, gc cr block busy, gc cr grant busy, gc current block busy, gc current grant busy, gc current split, ⋮

表1 RAC環境でのイベントの拡張例

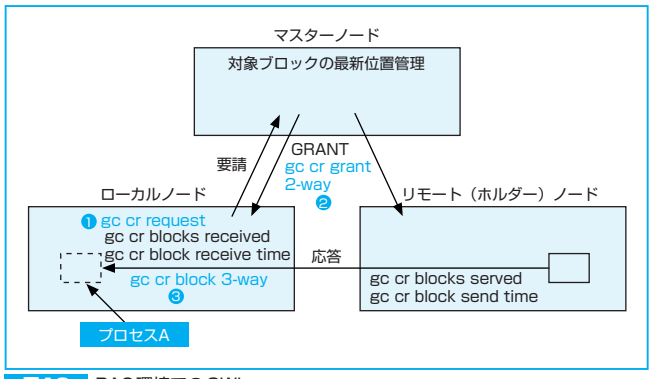


図10 RAC環境でのOWI

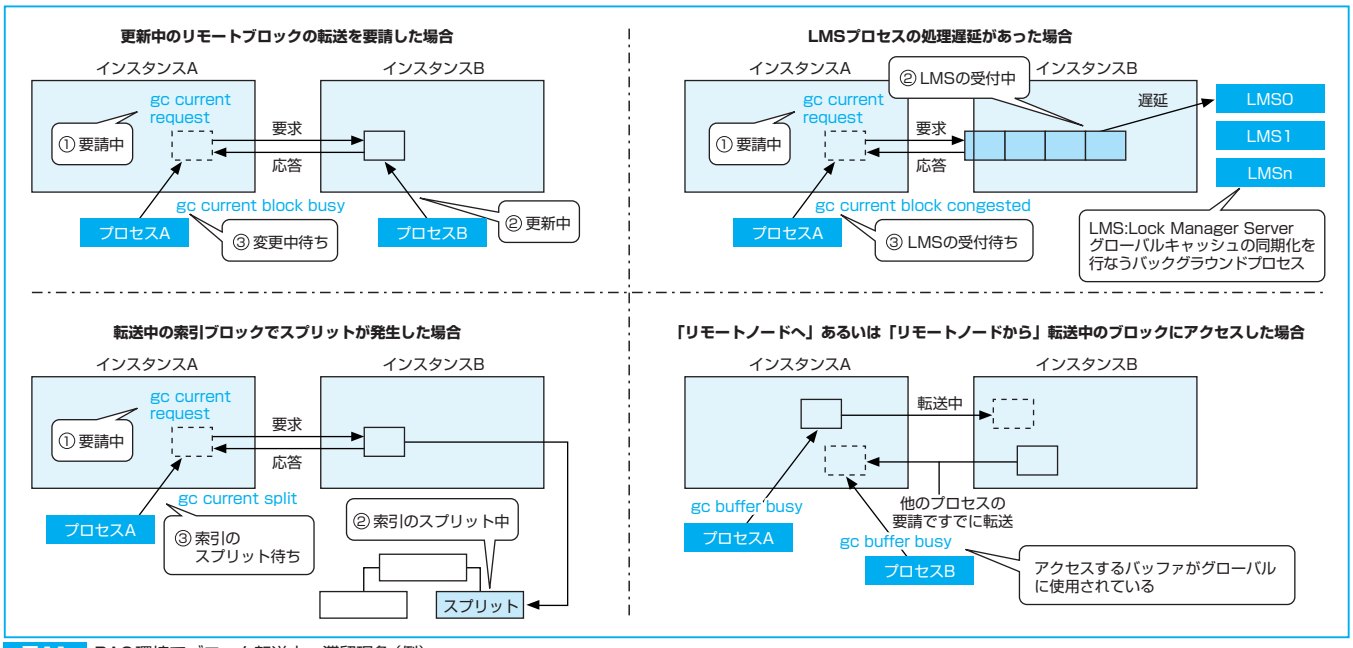


図11 RAC環境でブロック転送中、滞留現象（例）

区分	指標名	説明
統計指標	gc cr blocks received	ローカルノードがリモートノードからCRブロックイメージを受信した回数
	gc cr block receive time	ローカルノードがCRブロックを要請してからリモートノードから転送してもらうまでの所要時間（cs） CRブロックの平均受信時間（ms）= 10 * gc cr block receive time / gc cr blocks received
	gc cr blocks served	ホルダーノードがローカルノードに要求されたCRブロックイメージを転送した回数
	gc cr block build time	ホルダーノードのLMSがCRブロックイメージのコピー処理にかかった時間
	gc cr block flush time	CRブロックの転送要求に対して、ホルダーノードがカレントイメージを転送する際、カレントイメージをREDOログファイルへ記録する処理にかかった時間
	gc cr block send time	ホルダーノードがローカルノードに要求されたCRブロックイメージを転送する処理にかかったネットワーク時間 CRブロックの転送時間 = gc cr block build time + gc cr block flush time + gc cr block send time CRブロックの平均転送時間（ms）= 10 * CRブロックの転送時間 / gc cr blocks served
	gc current blocks received	ローカルノードがリモートノードからカレントブロックイメージを受信した回数
	gc current block receive time	ローカルノードがカレントブロックを要請してからリモートノードから転送してもらうまでの所要時間（cs） カレントブロックの平均受信時間（ms）= 10 * gc current block receive time / gc current blocks received
	gc current blocks served	ホルダーノードがローカルノードに要求されたカレントブロックイメージを転送した回数
	gc current block pin time	ホルダーノードのLMSがカレントブロックをピンする処理にかかった時間
	gc current block flush time	カレントブロックの転送要求に対して、ホルダーノードがカレントイメージをREDOログファイルへ記録する処理にかかった時間
	gc current block send time	ホルダーノードがローカルノードに要求されたカレントブロックイメージを転送する処理にかかったネットワーク時間 カレントブロックの転送時間 = gc current block pin time + gc current block flush time + gc current block send time カレントブロックの平均転送時間（ms）= 10 * カレントブロックの転送時間 / gc current blocks served
	global enqueue get time	ロックをグローバルに獲得する処理にかかった時間
イベント指標	gc cr request	ローカルキャッシュに存在しない単一ブロックを、リモートノードにCRモードで要求中
	gc current request	ローカルキャッシュに存在しない単一ブロックを、リモートノードにカレントモードで要求中
	gc cr grant 2-way	CRブロックの転送要求に対して、直接ディスクから読み込む権限を付与された結果
	gc cr block 2-way	CRブロックの転送要求に対して、要求先のサーバーから直接CRブロックイメージを受信した結果
	gc cr block 3-way	CRブロックの転送要求に対して、要求先のサーバー以外のホルダーサーバーを経由して、CRブロックイメージを受信した結果
	gc current grant 2-way	カレントブロックの転送要求に対して、直接ディスクから読み込む権限を付与された結果
	gc current block 2-way	カレントブロックの転送要求に対して、要求先のサーバーから直接カレントブロックイメージを受信した結果
	gc current block 3-way	カレントブロックの転送要求に対して、要求先のサーバー以外のホルダーサーバーを経由して、カレントブロックイメージを受信した結果
	gc cr multi block request	ローカルキャッシュに存在しないマルチブロックを、リモートノードにCRモードで要求中
	gc current multi block request	ローカルキャッシュに存在しないマルチブロックを、リモートノードにカレントモードで要求中
	gc buffer busy	処理対象のブロックがグローバルに使用中
	gc cr block congested	CRブロックの転送要求に対して、リモートノードのLMSプロセスの受付処理の遅延中
	gc cr block congested	カレントブロックの転送要求に対して、リモートノードのLMSプロセスの受付処理の遅延中

表2 RAC環境での主要指標

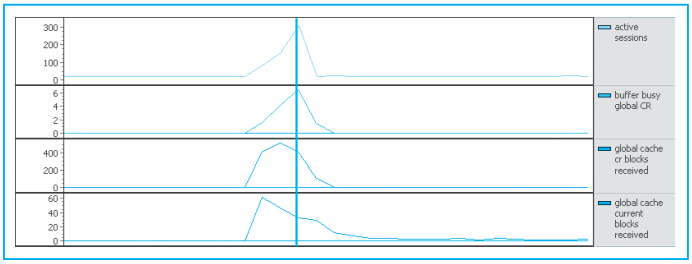


図12 RAC環境での性能低下例①

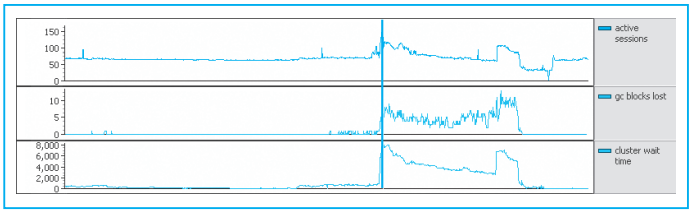


図13 RAC環境での性能低下例②

がどのような状況であるかが、直感的に分かるようになっている。主要な指標について図11と表2にまとめているので、参考にしてほしい。

RAC関連イベントの正確な動きに関する多くの情報がまだ公開されていないため、RACの性能問題の理解と改善にはほど遠い状況だが、一般的にシングルインスタンス環境でのチューニングポイント、特にバッファキャッシュの効率的な使い方と改善ポイント（SQLの検索範囲の最適化、ホットブロックの解消、マルチバッファプールの使用など）が、そのままRAC環境の性能改善につながるケースが多く見られる。そのため、RAC環境の診断／分析の際には、RAC関連イベントがシングルインスタンス環境のイベントと連動しているかを先に確認し、対応する必要がある。その後、RAC構成の特性に基づいて診断／分析を行なう。

RAC環境では、各インスタンス間で発生する競合を最小限に抑えること（アフィニティ）が重要なポイントになる。すなわち、特定インスタンスが特定リソースを独占して使えるようにすることで、インスタンス間の情報伝達を最小限にする。このような傾向はOracle内部で自動的に調整される部分もあるが、現状ではまだ人間の工

領域	指標名	説明
統計指標	CPU used by this session	CPU処理時間（10ミリ秒）
	active sessions	システムの安定度を示す代表目安で、性能指標と待機指標の両方を反映している
	session logical reads	メモリ（バッファキャッシュ）から読み取ったブロック数
	physical reads	ディスクから読み取ったブロック数
	logons current	データベースへの接続（セッション）数
	execute count	SQL文の実行回数（ユーザーコールおよび再帰コール）
	parse time elapsed	解析処理の所要時間（10ミリ秒）
	parse count（hard）	ハードパースの回数
共有プール	latch: shared pool	新しいSQL文（ハードパース）、PL/SQLプロシージャ、ファンクション、パッケージおよびトリガーの領域の割当、既存のchunkを解除または再利用に必要なラッチの獲得待ち
	latch: library cache	オブジェクトの変更、検索、ピン（pin）、ロック、ロード、または実行に必要なラッチの獲得待ち
	library cache lock	SQL解析処理中にメタ情報を保護
	library cache pin	SQL実行中に実行中のソース情報を保護
	latch: row cache objects	ディクショナリ情報をロード、参照、解除に必要なラッチの獲得待ち
	enq: SQ - contention	シーケンスをロード（キャッシュ）中にシーケンス情報を保護
バッファキャッシュ	latch: cache buffers chains	同時アクセス時のバッファブロック保護を目的に、バッファキャッシュへのブロック追加、削除、調査、読み取り、修正などに必要なラッチの獲得待ち
	latch: cache buffers lru chain	バッファキャッシュで新しいブロックをロードするため空き領域の確保に必要なラッチの獲得待ち
	buffer busy waits	同一のバッファを複数のセッションが変更を行なう場合、先に占有するセッション以外のセッションは待機
	read by other session	同一のバッファを複数のセッションが読み込みを行なう場合、先に占有するセッション以外のセッションは待機
	write complete waits	DBWRがディスクへ記録中のブロックを変更する場合、DBWRの記録完了まで待機
	free buffer waits	サーバープロセスが、バッファキャッシュから空きバッファを見つけられない場合、DBWRが変更済みのバッファの記録を完了し、フリーバッファを確保するまで待機
I／O	db file sequential read	ディスクからの単一データブロックの読み取りが実行されている間の待機時間で、通常索引を経由したデータのI/Oが多い場合、高くなる
	db file scattered read	ディスクからの複数データブロックの読み取りが実行されている間の待機時間で、通常表に対するフルスキャンが多い場合、高くなる
	direct path read	ダイレクト読取で、表領域からPGAにロードするまで待機
	direct path write	ダイレクト書き込み（PDML、CTASなど）で、PGAから表領域への書き込みが完了するまで待機
	direct path read temp	ソートセグメントを読み取るまで待機
	direct path write temp	ソートセグメントに書き込むまで待機
	db file parallel write	DBWRが、ダーティブロックをディスクへ記録するようI/Oコールをしてから完了するまで待機
	control file parallel write	ログスウィッチ、データファイル追加／削除などの場合、制御ファイルへの書き込み完了まで待機
	log file sync	ユーザープロセスのコミットによってLGWRがREDOエントリをディスクへ記録し終わるまで待機
	log file parallel write	LGWRが、REDOエントリをディスクへ記録するようI/Oコールをしてから完了するまで待機
	log buffer space	空きのログバッファが足りないためREDOエントリをコピーできない場合、LGWRがログバッファをディスクに書き下ろすまで待機
	log file switch completion	ログスウィッチの発生時、LGWRが現在ログファイルへの記録を完了し、新しいログファイルをオープンするまで待機
トランザクション	enq: TM - contention	オブジェクトに対するアクセスを同期化するためのロック制御
	enq: TX - allocate ITL entry	データブロック変更ためブロックヘッダーのITLの確保待ち
	enq: TX - index contention	索引ブロックの変更／追加作業を行なうとき、該当索引のスプリット完了待ち
	enq: TX - row lock contention	変更対象のレコードに対して他のトランザクションが完了するまで待ち
	enq: UL - contention	ユーザープロセスによる明示的なロック処理による待ち
セグメント	enq: HW - contention	複数のセッションから同時にHWM（最高水位標）の移動を行なうとき待機
	enq: ST - contention	ディクショナリ管理表領域の場合、複数のセッションから同時に領域拡張を行なうとき待機
	enq: TT - contention	表領域に対する同時DDLを行なうとき待機
	enq: US - contention	UNDOセグメントに対する同時DDLを行なうとき待機

表3 主な指標（イベント／統計）一覧

夫を超えることはない。例えば、セグメントの「FREELIST GROUPS」属性をノード数に合わせて設定することやASSMの採用などがある。また、シーケンスにキャッシュと共に、「NOORDER」属性を指定することも同じ観点から推奨される。**図12**では、不適切な「FREELIST GROUPS」の設定のため、インスタンス間でデータヘッダーブロックの転送が過度に行なわれた結果、性能低下現象が発生している。インターコネクトなどネットワーク設定による影響も少なくない。**図13**の性能低下現象は、ネットワーク装置の設定ミスにより発生した、ブロックの転送流失が原因である。

OWIの進化

Oracle9iから10gにバージョンアップされた際に、量と質の両面でOWIにも大きな変化があった。まず、量では400程度だったイベントが800以上に増えている。特に9iまでラッチ関連競合を「latch free」1つで現わしたものが「latch: ...」で29個の詳細イベントに、ロック関連競合を「enqueue」ですべて表現したものが「enq: ...」で209個の詳細イベントに分割されたため（10.2.0.3基準）、9iまではイベントのパラ

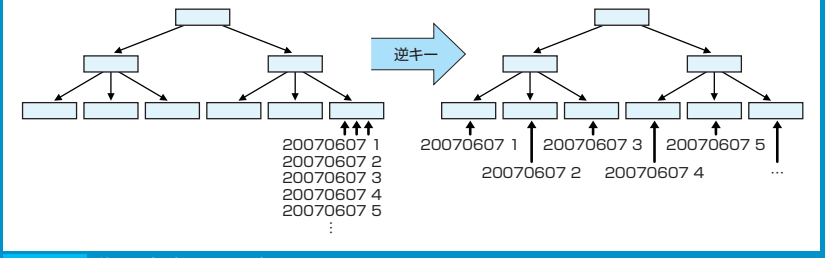
COLUMN

逆キー索引効果の落とし穴

シーケンスを含む索引は、データが追加される際に最後のブロックのみ集中的に使われるため、一般的に逆キー索引を構成することでアクセスの集中を避けるよう推奨されているが、これは一部だけの事実になる。**表D**のケース2のテスト結果から、索引ブロックへの集中による滞留「buffer busy waits、enq: TX - index contention」は一部解消されたが、これだけでは十分ではないことが分かる。残りの競合は新しいデータブロックの割り当て処理で起きているからである。「freelists」と自動セグメント領域管理（ASSM）を適用した結果からデータブロックでの滞留「enq: HW - contention、buffer busy waits 」が激減したことが確認できる。また、逆キー索引（**図D**）はクラスタ化係数を高めて索引スキャンの検索処理でデメリットがあるので、注意して使う必要がある。

テスト条件 / 結果	ケース1	ケース2	ケース3	ケース4
処理概要	「日付＋シーケンス」が主キーの表に対して、10セッションで同時にデータを追加する			
逆キー索引	×	○	×	×
freelists	1	1	10	N/A
ASSM適用	×	×	×	○
enq: TX - index contention	11	4	13	26
enq: HW - contention	50	57	10	9
buffer busy waits	81	57	21	46
CPU time	30	30	29	29
Wait Time	227	189	167	173
Elapsed Time	257	219	196	202
CLUSTERING_FACTOR	31,072	309,896	29,782	30,415

表D 逆キー索引



図D 逆キー索引のイメージ

メータで判断しなければならなかったことがイベント名で直感的に分かるようになった（ほかの主要指標と併せて**表3**を参照）。9iまでロックのタイプが多くて分かりにくかった部分も「V$LOCK_TYPE」の提供で一部解消された。

また、イベントのパラメータの詳細もより分かりやすくなっている。例えば、「buffer busy waits」のP3（「V$EVENT_NAME」のPARAMETER3）は原因コードからブロックのクラスに変更され、待ちリソースの種類がよりクリアになった。このほか、10gでは本格的なRAC環境の技術インフラ整備に伴ってRAC関連イベントの名前や分類、数の面でより精巧さが増した。

最後に最も歓迎すべき点は、「OWI分析用の履歴データ」セクションでも説明したが、Oracle内部でアクティブセッションを中心に履歴データを記録し始めたことで、OWIの活用の場が広がったことである。このように、より細かくより使いやすくなる方向でOWIは現在も進化中である。

おわりに

以上、OWIの活用法について説明してきたが、いかがだっただろうか。誌面の都合により、当初予定していた内容のすべては紹介できなかったが、OWIの活用イメージと実践方法をつかんでいただけたことと思う。具体的な事象によって、通常「良い」と言われている改善方法も逆に毒になるケースも多々あるため、本稿ではあえて各イベントの一般的なチューニングポイントを記載しなかったが、イベントが発生する仕組みを理解することで的確な診断／分析につなげられるはずだと考える。

OWIを構成している統計指標とイベントの個別の特徴を理解することでより正確な解析が可能になるが、また機会があれば改めてこの誌面で紹介できればと思う。今後、Oracleを使ったシステム開発や性能テスト、運用監視、トラブル解析／対応でOWI的な考え方が広がれば幸いである。

DBM

※当記事は次の資料を参考にし、テスト結果とともに再構成してまとめたものである(参考頻度順)。
- Advanced OWI in Oracle 10g
- Practical OWI Training Materials
- DBMS INTERNALS MAGAZINE
- http://support.oracle.co.jp/
- https://metalink.oracle.com/

※使用ツール（使用頻度順）
- Windows Server 2003 / Oracle 10.2.0.3）
- STATSPACK
- EVENT 10046 TRACE
- MaxGauge 2.5 for Oracle

川向一郎（かわむかいいちろう）
株式会社サンブリッジアンシスに所属。日本オラクルにて主にパートナー企業および通信業界担当のセールスコンサルタントとして勤務。現在パフォーマンス関連コンサルタント兼エンジニアリング部門のマネージャ。

金 圭福（きむぎゅうぼく）
株式会社サンブリッジアンシスに所属。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」の技術サポートおよびパフォーマンス関連コンサルティングを担当している。最近はフルマラソン完走という個人的な実績も上げている。