特3集 成功事例に学ぶ統計指標の活用とチューニングの実践

ボトルネックを見極めて最大限の効果を狙うDB性能改善テクニック

日本エクセム株式会社
金 圭福
KIM,Gyu-Bok

費用、時間、人、スペースなど、コストをかけるほどシステムは良くなっていく可能性が高い。DBチューニングは、既存のシステムリソースを最大限に活用できるようにする作業である一方で、諸コストを最小限に押さえられる最適ラインを決める作業でもある。そのため、やみくもにハードウェアを増設したり、いつまでもアプリケーションの見直し作業を行なってコストや時間を費やすのは好ましくない。ITシステムの総合コストを最小化する取り組みの1つとして、本稿ではOracleデータベースの物理読み取りとその解消法について、事例をまじえて紹介する。

接続エラーの性能問題

携帯電話やPC向けの音楽配信サイトを運営しているA社より、システムトラブルの話があった。ユーザーの利用が集中する20:00 ～ 22:00の時間帯に接続エラーが頻発しているというものだった。A社のビジネスの性格上、年末年始にはアクセスの急増が予想されるため、その時期でも耐えられるよう改善したいというニーズもあった。また、A社ではDBサーバーなどのハードウェアリソースの増設の提案を検討中で、その判断を裏付ける最終段階に入っていた。

A社のシステムは、携帯電話やPCからのリクエストに対して、クラスタ構成のWebアプリケーションサーバー（Apache Tomcat）とDBサーバー（Oracle9i R2 SE）でサービスを提供するというもので、WebアプリケーションサーバーとDBサーバーの間でコネクションプール注1を運用していた（**図1**）。各接続要求はロードバランサによってコネクションプールの接続数を基に最小接続数のプールに割り当てる仕組みになっていたが、この割り当て処理で接続エラーが発生していた。

今回のトラブルは、すでに最大値まで達しているコネクションプールに対して新たなコネクションを割り当てようとしたため発生したものであった。そのため、ロードバランサまたはコネクションプールの設定の問題とDB性能の問題で原因追跡のレイヤが考えられたが、本稿では後者について効果的に処理時間（接続時間→同時接続数）を短縮する方法を検討した。

注1：一度使い終わった接続を切断せずに保持しておいて、クライアントから新たな接続要求を受けたときに再利用する機能。WebアプリケーションのようにDBに対する接続や切断が頻繁に繰り返される場合、DB接続によるオーバーヘッドを軽減する効果がある。

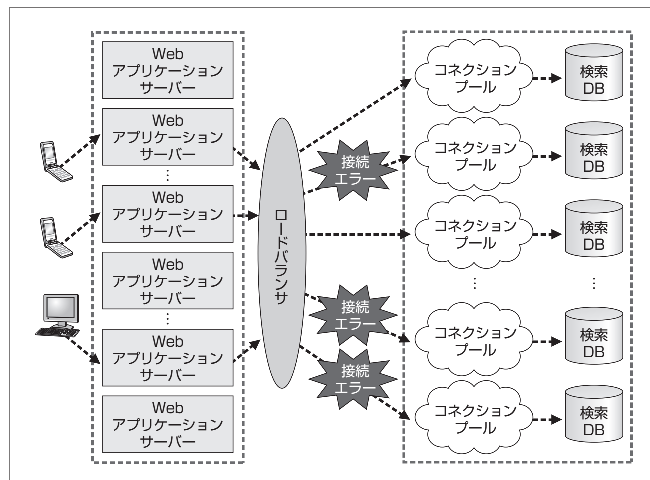

図1：検索システムの構成とトラブルのイメージ

診断と分析の概要

性能改善を行なうためには、まず処理のボトルネック（処理時間が長い箇所）を明確にして、その度合いを測定する必要がある。次に、その部分に対する詳細調査を行なったうえで、適用可能な改善案をリストアップする。その効果を検証してから運用条件を考慮し、運用環境への適用を戦略的に行なう必要がある（**図2**）。

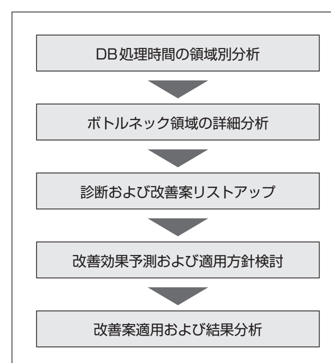

図2：DB性能改善の作業フロー

今回の事例となる検索システムの運用状況を表わした24時間トレンド（**図3**）から、以下の点で「20:00 ～ 22:00」区間がボトルネックになっていることが確認された。

- 通常10秒以下のDB全体滞留時間（total wait time）が、当該時間帯で平均「20秒」前後、最大「50秒」以上まで増加している
- CPU使用率（CPU）が当該時間帯で「100%」に達しても処理し続けている
- CPU使用時間（CPU used by this session）とDB接続数

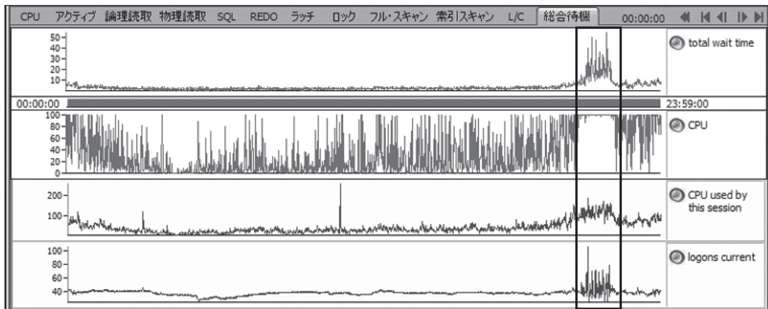

図3：統計指標における24時間トレンド（性能改善前）

表1：性能改善前（9/12）
「20:00～22:00」時間帯のDB処理時間統計

タイム領域	時間（秒）	DB処理時間に対する割合	要改善
DB処理時間[A＝B＋C]	103,016.30	100.0%	
CPU処理時間[B]	7,550.30	7.3%	
合計待機時間[C ≒ D～I]	95,466.00	92.7%	○
待機領域 ： SQL解析[D]	4,368.40	4.2%	△
待機領域 ： 物理読み取り[E]	89,481.80	86.9%	○
待機領域 ： 論理読み取り[F]	619.60	0.6%	
待機領域 ： ロック[G]	221.80	0.2%	
待機領域 ： REDOログ[H]	344.90	0.3%	
待機領域 ： 制御ファイル[I]	427.50	0.4%	

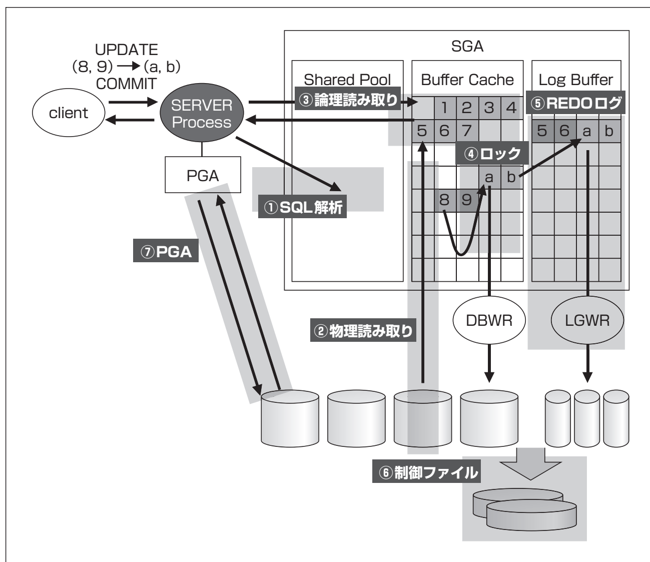

図4：ボトルネックが発生しやすい領域

表2：「20:00～22:00（性能改善前）」時間帯の上位待機イベント

待機イベント	待機時間	比率
db file sequential read	**89,318.70**	**88.15%**
SQL*Net message from dblink	5,908.45	5.83%
latch free	4,366.41	4.31%
buffer busy waits	614.17	0.61%
control file parallel write	426.73	0.42%
log file parallel write	292.79	0.29%
enqueue	219.23	0.22%
db file parallel read	105.69	0.10%
db file scattered read	46.09	**0.05%**
log file sync	27.21	0.03%
合計	101,325.47	100.00%

(logons current) が当該時間帯で2倍以上増加している

一般的にDB内部処理でボトルネックになりやすい領域を簡単にまとめると、**図4**のようになる。この中で、ほとんどのシステムで悩まされていて最も処理時間を要する箇所が「②物理読み取り」だ。

ほかのアプリケーションと同じく、DBMSもSELECTなどユーザーからの作業依頼をメモリ上で処理する。しかし、膨大なユーザーデータを同時にメモリ（「バッファキャッシュ」と言う）上に乗せられないため、頻繁に使われるデータはできるだけバッファキャッシュに保持され、そうではないデータは早めに追い出される仕組みになっている。ここで、要求されたデータをバッファキャッシュから参照するI/Oを論理読み取りと言い、要求されてデータがバッファキャッシュに存在しないためディスクからバッファキャッシュにコピーするI/Oを物理読み取りと言う。このような物理読み取りは、OSとのやり取りやハードディスクの回転といったDB外部の処理が発生させるため、最も重いDB処理とされている。

ボトルネック時間帯のDB処理時間をまとめた**表1**からも分かると思うが、物理読み取りを解消すれば、86.9%の処理時間を削減できる。もちろん、ディスク基盤のDBMSで物理読み取りを完全になくすことはできないが、例えばその半分にできればDB全体処理時間において大きな改善が期待できる。

また、**表2**の上位待機イベントリストで、「db file sequential read[注2]」(88.15%) が「db file scattered read[注3]」(0.05%) よりかなり高いことから、索引スキャンによるデータ読み取りの改善効果が最も大きいことが分かった。

さらに、処理時間および物理読み取りのブロック数が多い上位SQLを対象にその実行統計を集計すると、上位5つのSQLが全SQL処理の70%以上を占めていることが確認された（**表3**）。このデータからも上位5つのSQLの個別チューニングを行なうことで最大70.02%の処理時間の改善が見込まれることが分かる。

上記の診断結果より複数の改善案がリストアップされたが、その中でシステム運用に対するリスクが少なく、かつ改善効果が大きい上位5つのSQLに的を絞って索引の見直しを行なった。その結果、処理時間が大幅に改善され、接続エラーは発生しなくなった。

このように、あるタイミングのDB稼動状況の概要と詳細を事前に確認できると、どの部分を集中的に改善すべきか、どのくらい改善されるかがあ

注2：ディスクからのマルチデータブロック読み取りが完了するまでの待機時間。通常は表に対するフルスキャンが多い場合は高くなる。

注3：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。通常は索引を経由したデータ読み取りが多い場合に高くなる。

る程度は見えてくるので、より効率的な性能改善作業につながるのである。

表3：「9/12 20:00～22:00」時間帯の上位SQL

区分		実行時間（秒）	CPU時間（秒）	待機時間（秒）	論理読み取り（ブロック）	物理読み取り（ブロック）	実行回数（回）
全体		111,072.20	9,794.30	101,277.90	889,383,615	6,288,401	119,843
SQL1	値	22,224.70	2,610.95	19,613.75	242,338,858	1,094,865	27,797
	割合	20.01%	26.66%	19.37%	27.25%	17.41%	23.19%
SQL2	値	14,509.85	2,606.00	11,903.85	231,984,822	635,326	24,881
	割合	13.06%	26.61%	11.75%	26.08%	10.10%	20.76%
SQL3	値	22,193.05	1,278.75	20,914.30	143,863,260	1,730,082	3,928
	割合	19.98%	13.06%	20.65%	16.18%	27.51%	3.28%
SQL4	値	8,254.80	314.60	7,940.20	9,567,032	567,845	7,809
	割合	7.43%	3.21%	7.84%	1.08%	9.03%	6.52%
SQL5	値	10,591.80	348.85	10,242.95	9,720,280	809,052	4,435
	割合	9.54%	3.56%	10.11%	1.09%	12.87%	3.70%
SQL1～5合計	値	77,774.20	7,159.15	70,615.05	637,474,252.00	4,837,170.00	68,850.00
	割合	70.02%	73.10%	69.72%	71.68%	76.92%	57.45%

稼動ログを収集する

エンドユーザーからシステムの性能についてのクレームが来たとき、通常はOSツールなどでシステムの稼動状況を確認する。しかし、この手の話の大半は過去の現象だということが問題だ。そのため、ある程度の経験値が蓄積された現場では、問題となった、あるいはトラブルになりそうな部分（処理）に対して常に稼動ログを収集している。

本事例では、稼動中のデータベースの性能問題に対して下記を含む稼動ログを収集した。

- 性能統計指標
- 待機イベント指標
- SQL実行統計
- SQLテキスト
- CPU使用率

診断／分析の概要でも説明したが、システム全般の稼動状況から性能診断対象の時間帯を絞ってDB処理時間の改善度合いを明確にするため、性能統計指標、待機イベント指標、CPU使用率のデータを利用した（図3、表1、表2参照）。

実際の作業では、筆者が所属する日本エクセムの「MaxGauge」というツールを使ったが、**LIST1**の「gather_instance_stats.sql」と「gather_instance_waits.sql」のスクリプトを利用すれば、同様のデータ収集および診断が可能である。

このスクリプトは、性能統計指標と待機イベント指標の統計値を定期的（サンプルでは1分間隔）に収集して「収集時刻、指標、値」項目を出力するもので、その差分が各時刻間で発生した各指標の統計値を示している。この差分を時系列でつないだトレンドが図3で、各指標の差分を合計して逆ソートしたものが表2である。**表4**でタイム領域を構成する主要詳細指標をまとめているが、その詳細指標の差分を各タイム領域単

```
<gather_instance_stats.sql>
set serveroutput on
declare
    fp   utl_file.file_type;
begin

    while ( 1 = 1 ) loop

        fp := utl_file.fopen('d:¥temp','instance_stats.csv','a'); -- 初期化パラメータ「utl_file_dir」の指定場所

        for rec in (
            SELECT TO_CHAR( SYSDATE, 'yyyy/mm/dd hh24:mi:ss' ) logging_time ,
                name ,
                value
            FROM   v$sysstat
            WHERE name = 'CPU used by this session'          -- その他の性能統計指標も収集する場合は、この条件を外す

        ) loop

            utl_file.put_line( fp , '="' || rec.logging_time || '",' || rec.name  || ',' || rec.value );

        end loop;

        utl_file.fclose(fp);

        dbms_lock.sleep (60); -- データ収集頻度：「1回/1分」推奨

    end loop;

exception

    when others
    then

        dbms_output.put_line('file output error' || to_char(sysdate, 'yyyy/mm/dd hh24:mi:ss') || ⇒
sqlcode || ',' || sqlerrm) ;
end;
/
```

※ 出力データの編集サンプル

収集時刻	指標	値(0.01秒)	差分(0.01秒)
2009/01/10 12:04:58	CPU used by this session	4,667	978
2009/01/10 12:05:58	CPU used by this session	5,604	937
2009/01/10 12:06:58	CPU used by this session	6,430	826

```
<gather_instance_waits.sql>
set serveroutput on
declare
    fp   utl_file.file_type;
begin

    while ( 1 = 1 ) loop

        fp := utl_file.fopen('d:¥temp','instance_waits.csv','a'); -- 初期化パラメータ「utl_file_dir」の指定場所

        for rec in (
            SELECT TO_CHAR( SYSDATE, 'yyyy/mm/dd hh24:mi:ss' ) logging_time ,
                   event ,
                   time_waited -- 単位：0.0.1秒
            FROM   v$system_event
            WHERE  event not in (
                         'ASM background timer',             -- アイドルイベントなど性能に影響を与えない待機指標
                      ：
                         'watchdog main loop'
                     )
        ) loop

            utl_file.put_line( fp , '="' || rec.logging_time || '",' || rec.event  || ',' || ⇒
rec.time_waited );

        end loop;

        utl_file.fclose(fp);

        dbms_lock.sleep (60); -- データ収集頻度：「1回/1分」推奨

    end loop;

exception
    when others
    then
        dbms_output.put_line('file output error' || to_char(sysdate, 'yyyy/mm/dd hh24:mi:ss') || ⇒
sqlcode || ',' || sqlerrm) ;
end;
/
```

※ 出力データの編集サンプル

収集時刻	指標	値(0.01秒)	差分(0.01秒)
2009/01/10 12:04:58	db file sequential read	18307	254
2009/01/10 12:05:58	db file sequential read	18329	22
2009/01/10 12:06:58	db file sequential read	18515	186

※誌面の都合により⇒で折り返し。以下同

LIST1 ：DB統計指標収集のスクリプト

表4：各タイム領域を構成する主要指標

タイム領域	指標名
CPU処理時間	CPU used by this session
SQL解析	latch: library cache
	latch: shared pool
	latch: library cache
	latch: library cache lock
	latch: library cache pin
	library cache pin
	library cache lock
	latch: row cache objects
物理読み取り	db file sequential read
	db file parallel read
	read by other session
	db file scattered read
論理読み取り	buffer busy waits
	latch: cache buffers chains
	latch: cache buffers lru chain
ロック	enq%
REDOログ	log buffer space
	log file parallel write
	log file switch (archiving needed)
	log file switch (checkpoint incomplete)
	log file switch (clearing log file)
	log file switch (private strand flush incomplete)
	log file switch completion
	log file sync
制御ファイル	control file parallel write
	control file sequential read
	control file single write
PGA	direct path read
	direct path write
	direct path read temp
	direct path write temp

位で合計すると表1になる。

また、SQL単位の負荷率を算出するため、**LIST2**の「gather_sql_stats.sql」でSQLの実行統計を、「gather_sql_texts.sql」でSQLテキストを定期的に収集する。SQL実行統計値もSQLがロードされてからの累積値なので、一定期間の実行統計は各タイミングの差分で算出する。この結果を「hash_value、address」ごとに合計して逆ソートし、全体合計に対する各SQLの比率を出すことで表3を作り上げる。最終的に改善を行なうSQLに対してはSQLテキストを参照する。

ここでは簡単ながらサンプルスクリプトを紹介したが、STATSPACK[注4]レポートの「Top 5 Timed Events」「Wait Events」「SQL ordered by

注4：パフォーマンスデータを取得したスナップショット間の差分に基づくパフォーマンス診断ツール。Oracle8.1.6から使用可能。

```
<gather_sql_stats.sql>

set serveroutput on
declare
    fp   utl_file.file_type;
begin

    while ( 1 = 1 ) loop

        fp := utl_file.fopen('d:¥temp','sql_stats.csv','a'); -- 初期化パラメータ「utl_file_dir」の指定場所

        for rec in (
            SELECT TO_CHAR(SYSDATE, 'yyyy/mm/dd hh24:mi:ss') logging_time ,
                   hash_value ,
                   address ,
                   SUM(elapsed_time) elapsed_time ,                       -- 単位：1/1000000秒
                   SUM(cpu_time) cpu_time ,                               -- 単位：1/1000000秒
                   SUM(disk_reads) disk_reads ,                           -- 単位：読取り回数→ブロック
                   SUM(buffer_gets) buffer_gets ,                         -- 単位：読取り回数→ブロック
                   SUM(executions) executions                             -- 単位：回
            FROM   v$sql
            WHERE  parsing_schema_id NOT IN (
                       SELECT user_id
                       FROM   dba_users
                       WHERE  username IN (
                                'BI','CTXSYS','DBSNMP','DMSYS','EXFSYS','HR','IX',
                                'MDSYS','OE','OLAPSYS','ORDPLUGINS','ORDSYS','OUTLN','PERFSTAT',
                                'PM','PUBLIC','SCOTT','SH','SI_INFORMTN_SCHEMA',
                                'SYSMAN','SYSTEM','SYS','TSMSYS','WMSYS','XDB','ODM'
                              )
                   )
            AND    elapsed_time >= 10000
            GROUP BY hash_value, address
        ) loop

            utl_file.put_line( fp , '="' ||
                rec.logging_time    || '",="' ||
                rec.hash_value                   || '",="' ||
                rec.address                      || '",' ||
                rec.elapsed_time    || ',' ||
                rec.cpu_time                     || ',' ||
                rec.disk_reads                   || ',' ||
                rec.buffer_gets                  || ',' ||
                rec.executions
            );

        end loop;

        utl_file.fclose(fp);

        dbms_lock.sleep (600); -- データ収集頻度：「1回/10分」推奨

    end loop;

exception
    when others
    then
        dbms_output.put_line('file output error' || to_char(sysdate, 'yyyy/mm/dd hh24:mi:ss') || ⇒
sqlcode || ',' || sqlerrm) ;
end;
/
```

※ 出力データの編集サンプル：SQL実行統計の収集結果と10分間の統計値（差分）

logging_time	hash_value	address	elapsed_time (1/1000000秒)	disk_reads (ブロック)	elapsed_time (差分:秒)	disk_reads (差分:ブロック)
2009/01/10 12:13:58	304208677	2AAE4744	4643534	0	4.450902	0
2009/01/10 12:23:58	304208677	2AAE4744	8760676	0	4.117142	0
2009/01/10 12:33:59	304208677	2AAE4744	13197174	0	4.436498	0

```
<gather_sql_texts.sql>
set serveroutput on
declare
    fp   utl_file.file_type;
begin

    while ( 1 = 1 ) loop

        fp := utl_file.fopen('d:¥temp','sql_texts.csv','a'); -- 初期化パラメータ「utl_file_dir」の指定場所

        for rec in (
            SELECT TO_CHAR(SYSDATE, 'yyyy/mm/dd hh24:mi:ss') logging_time ,
                   address, hash_value, sql_text
            FROM   v$sqltext
            WHERE  (hash_value, address) IN (
                      SELECT hash_value, address
                      FROM   v$sql
                      WHERE  parsing_schema_id NOT IN (
                                 SELECT user_id
                                 FROM   dba_users
                                 WHERE  username IN (
                                          'BI','CTXSYS','DBSNMP','DMSYS','EXFSYS','HR','IX',
                                          'MDSYS','OE','OLAPSYS','ORDPLUGINS','ORDSYS','OUTLN','PERFSTAT',
                                          'PM','PUBLIC','SCOTT','SH','SI_INFORMTN_SCHEMA',
                                          'SYSMAN','SYSTEM','SYS','TSMSYS','WMSYS','XDB','ODM'
                                        )
                             )
                      AND    elapsed_time >= 10000
                   )
            ORDER BY hash_value, address, piece
        ) loop

            utl_file.put_line( fp , '="' ||
                rec.logging_time    || '",="' ||
                rec.hash_value                   || '",="' ||
                rec.address                      || '","' ||
                rec.sql_text        || '"'
            );

        end loop;

        utl_file.fclose(fp);

        dbms_lock.sleep (600 * 6); -- データ収集頻度：「1回/1時間」推奨

    end loop;

exception
    when others
    then
        dbms_output.put_line('file output error' || to_char(sysdate, 'yyyy/mm/dd hh24:mi:ss') || ⇒
sqlcode || ',' || sqlerrm) ;
end;
/
```

※ 出力データの編集サンプル：SQL実行統計の収集結果と10分間の統計値（差分）

logging_time	hash_value	address	SQL文
2009/1/10 12:03	722095927	24FE78C0	SELECT m.channel_desc, s.cust_name, s.sum_quantity_sold, s.sum_a
2009/1/10 12:03	722095927	24FE78C0	mount_sold, s.top_rank FROM (SELECT s.channel_id, c.cust_firs
2009/1/10 12:03	722095927	24FE78C0	t_name || ' ' || c.cust_last_name cust_name, SUM(s.quantity_sold

LIST2：SQL関連情報収集のスクリプト

…」セクションからもほぼ同様の診断が可能である。しかし、STATSPACKは一定期間の性能全般を評価するには最適なツールだが、次の場合の性能分析の目的には沿えない傾向がある。

- ログ収集の負荷が懸念される場合
- 短い間隔のログが必要なとき
- 任意時間帯の分析が必要なとき
- 時系列の情報が必須な場合
- リテラルSQLが多い場合
- 収集対象SQLの条件をユーザー定義する場合

物理読み取り改善への一般的な対処

ディスク性能が高速化の傾向にあっても、ディスク基盤データベースの性能問題のほとんどはI/Oに起因している。本事例が検索システムということもあって、物理読み取りでDB処理時間の86.9%を費やしていた（表1）。物理読み取りのボトルネックは、一般的に以下の待機イベントの統計値でよく表われる。

db file scattered read
db file sequential read
db file parallel read注5
read by other session注6

本セクションでは、上記指標が上位にラックされたDBシステムに対する一般的な改善対策と本事例での考え方を紹介する。

（1）索引の見直し

データのI/Oをより効率的に行なえるよう、DBMSはさまざまな物理的仕組みを提供しているが、その代表的なものが索引だ。例えば、**図5**のように1000ブロックを占めている表から索引なしで「OBAMA」のデータを読み取るには、最大1000ブロックを確認する必要がある。しかし、名前の索引が作成されていれば、4ブロックの読み取りで同様の作業ができる。

このように、索引を適切に活用すればI/Oの性能問題は一段と対処しやすくなる。検索処理で条件カラムに索引がないため、「db file scattered read」が高くなるケースもあるが、索引は機能するものの効果的でないという場合、「db file sequential read」が急増するケースもよくある。

このような場合、気持ちとしては全索引の使用状況を把握してから現行のDB運用パターンに合わせて全索引の設計を見直したいところだが、余りにも莫大な費用と時間がかかるため現実的ではない。本稿の目的でもあるが、最小限のコストで最大限の効果を出すためには、表3のような上位SQLに集中して、最も効率の良い索引を作ることが現実的にはベストな対処となる。

ただし、索引の追加／変更／削除は、関連表にアクセスしているSQLの実行計画を意図しないパスに変えてしまうことがあるので、本番適用前には十分な検証が必要だ。

SQLチューニングと連携した改善案の1つとして**LIST3**を参考にしてほしい。サンプルSQLは表3の「SQL1」に当たる。SQL1の平均処理時間は「0.79954」秒と一見軽く見えるが、検索システムにおける処理時間と物理読み取り数でTOP1なので、どんなコストを投入しても改善しておきたいところだ。また、表のデータ件数、索引の構成から「s_msc_sch.索引②、s_prd_sch.索引②、s_prd.索引①」を活用してデータを読み取れることが分かる。

ここではSELECT項目が少ない（msc_idのみ）こととデータ件数が多いことを考慮して、表の領域までアクセスを行なわずに索引の領域だけのデータ処理で結果を出せるよう、関連する全カラムを索引として構成した。その結果、SQL1の平均処理時間は「0.09886」秒に縮まった。

これはメモリにロードするブロック数を最小限に抑えたことによる効果である。特に索引のみでデータ処理を行なうため、「s_prd」の索引に「NVL(rbt_sts_pub_dc,'1')」を含むファンクション索引注7を適用するコストをかけたことにも注目しよう。

（2）SQLチューニング

負荷が高く性能にとってクリティカルなSQLに対しては、実行パス（実行計画）の妥当性を検証してアクセス範囲を減らす方向でチューニングを行なう必要がある。実行時間が長いSQLは個々のSQLの負荷が高い場合もあるが、個々のSQLの負荷は低いのに実行回数が多いSQLも現われる。特に、リテラルSQL注8は集計してみないとその怖さがなかなか見えてこないので、さらに注意が必要だ。

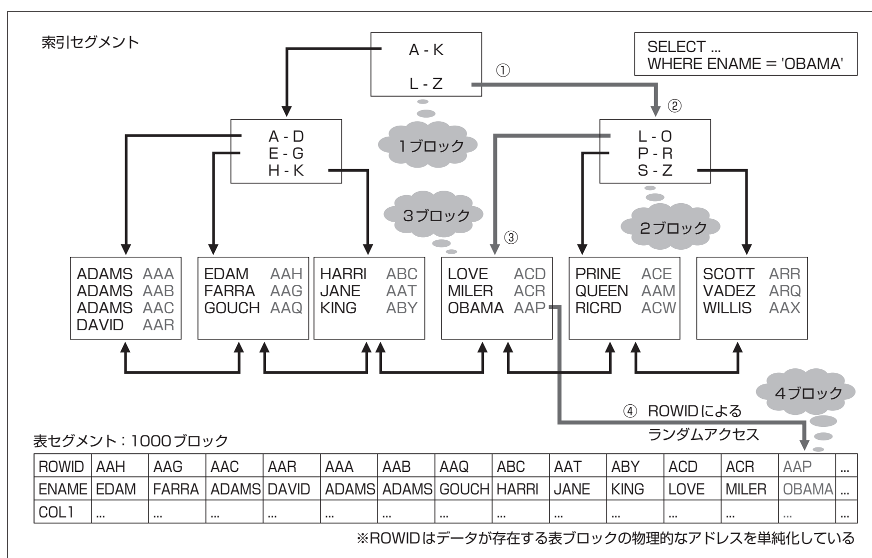

図5：B*ツリー索引によるデータアクセスのイメージ

注5：1つ以上のデータファイルから連続していないシングルブロックを同時に読み取る場合に増加する。
注6：ディスクの特定ブロックが同時に複数プロセスによって読み込み対象になった場合、1セッションのみ読み込み処理を行なう。ほかのセッションは対象のブロックがメモリにロードされるまで待機する。同じブロックに対する読み取り競合の場合に増加する。
注7：ファンクションや式の結果値を物理的に格納している索引。

(3) バッファプールの拡張

OSで使われていない物理メモリやSGAの空きメモリがある場合、その一部をバッファキャッシュに追加で割り当てることは、伝統的ながら効率の良い方法だ。OSの物理メモリを割り当てる場合は「sga_max_size」と「db_cache_size」の拡張設定を、SGA内部でメモリ調整を行なう場合は「sga_target= sga_max_size」を推奨する。「sga_target」を設定できない場合は、「v $sgastat」でSGAの利用統計を確認してから「db_cache_size」の拡張設定を検討する。

(4) パーティション化

過去データのように、めったにアクセスされないデータと頻繁にアクセスされるデータが同じ表に格納されていると、データ管理面だけでなく、メモリにロードされて物理読み取りでも無駄が発生する。このような場合は、アクセス頻度に基づいてセグメントを分けることで、読み取り効率の最適化を図れる。DBMS提供のパーティション機能を適用するか、その機能を使えない場合は明示的に表の名前で分けてデータを格納することで実現することも可能だ。

(5) データの再編成

運用時間の経過とともにデータの変更、削除および追加が重なって、データを格納している領域の断片化が進んだためデータの格納の効率が悪くなると、その分だけ物理読み取りの回数も増えてくる。特に索引セグメントは表セグメントより断片化の傾向が激しいので、データの増減パターンに合わせて定期的に再編成[注9]を行なう必要がある。

これについては、**表5**のように格納率が高い順から再編成対象を判断すれば良く、表3の上位SQLがアクセスしている表と引き合わせることで、その対象がより分かりやすくなる。ただ、「PCT_FREE」によって再編成後は逆にブロック数が増えることがあるので、「格納率 ＞ 150%」と「PCT_FREE <= 10」の表とその索引を対象にしてほしい。

```
<改善前（現行）：索引とSQL>

表：s_msc_sch                          10,000,000件
    索引①：(pk) msc_id + sit_id + dev_cd + sub_no
    索引②：typ_art_id + dev_cd + sit_id
    索引③：art_id + dev_cd + sit_id

表：s_prd_sch                          15,000,000件
    索引①：(pk) prd_id + sit_id + dev_cd + sub_no
    索引②：msc_id + dev_cd + sit_id

表：s_prd                              500,000件
    索引①：(pk) prd_id

表：s_art_msc                          100件
    索引①：(pk) art_id + msc_id

SELECT sms.msc_id
FROM   s_msc_sch sms ,
       s_prd_sch sps ,
       s_prd sp
WHERE  sms.typ_art_id = :1
AND    sms.dev_cd = :2
AND    sms.sit_id = :3
AND    sms.art_sts_pub = '1'
AND    sms.prv_flg = '0'
AND    sms.msc_id = sps.msc_id
AND    sms.dev_cd = sps.dev_cd
AND    sps.sit_id IN ( 1 , 2 , 10 , 11 , 12 )
AND    sps.prd_id = sp.prd_id
AND    sps.sts_pub = '1'
AND ( sp.rbt_sts_pub_dc = '1' OR sp.rbt_sts_pub_dc IS NULL )
AND    sps.srt_date <= :4
AND    sps.end_date >= :5
AND    NOT EXISTS (
                   SELECT 1
                   FROM   s_art_msc sam
                   WHERE  sam.typ_art_id = sms.typ_art_id
                   AND    sam.msc_id = sms.msc_id
       )
GROUP  BY sms.msc_id
;

<改善案：索引とSQL>
表：s_msc_sch
    索引②「：typ_art_id + dev_cd + sit_id + art_sts_pub + prv_flg + msc_id

表：s_prd_sch
    索引②：msc_id + dev_cd + sit_id + sts_pub + prd_id + srt_date + end_date + data_type

表：s_prd
    索引②：prd_id + NVL(rbt_sts_pub_dc,'1')

SELECT sms.msc_id
FROM   s_msc_sch sms ,
       s_prd_sch sps ,
       s_prd sp
WHERE  sms.typ_art_id = :1
AND    sms.dev_cd = :2
AND    sms.sit_id = :3
AND    sms.art_sts_pub = '1'
AND    sms.prv_flg = '0'
AND    sms.msc_id = sps.msc_id
AND    sms.dev_cd = sps.dev_cd
AND    sps.sit_id IN ( 1 , 2 , 10 , 11 , 12 )
AND    sps.prd_id = sp.prd_id
AND    sps.sts_pub = '1'
AND    NVL(sp.rbt_sts_pub_dc, '1') = '1'
AND    sps.srt_date <= :4
AND    sps.end_date >= :5
AND    NOT EXISTS (
                   SELECT 1
                   FROM   s_art_msc sam
                   WHERE  sam.typ_art_id = sms.typ_art_id
                   AND    sam.msc_id = sms.msc_id
       )
GROUP  BY sms.msc_id
;
```

LIST3 ： 性能改善対象上位SQL1と改善案

表5 ： 表の統計データ

表	PCT_FREE	データ件数 (A)	平均長さ (B)	ブロック数 (C)	格納率 C / (A * B / 8K)
S_CAT_MSC	10	400,000	30	4151	283.37%
S_MSC	20	2,000,000	300	171241	233.80%
S_MSC_SCH	10	10,000,000	150	351104	191.75%
S_MSC_FWD	10	500,000	60	6012	164.17%
S_PRD	10	500,000	150	15000	163.84%
S_PKG_FWD	10	600,000	70	8001	156.06%
S_PRD_SCH	20	13,000,000	165	401511	153.34%
S_PKG	30	50,000	300	2802	153.03%

※データは「select table_name, pct_free, num_rows, avg_row_len, blocks from dba_tables」で抽出する

(6) PCTFREEの見直し

表の作成時、初期に追加されたデータが更新によって長くなることを考慮して、「PCTFREE」

注8：変数の部分のみが異なるSQL（下の例を参照）。1つのSQLでは軽いため目立たないが、実際にはシステムにとって致命的となる場合が多いため、実行統計を類似SQL単位でまとめて評価する必要がある。
例）　select * from emp where empno = 1029;
　　　select * from emp where empno = 2839;

注9：領域の無駄をなくすためにデータを入れ直す作業。

に余裕分を持たせて設定する。デフォルトでは「10」%の伸び率までカバーするが、データ更新の特性を考慮してその値を調整する必要がある。1レコードが長くなる可能性が低いほど、「PCTFREE」を小さく設定することで物理読み取りのブロック数を削減する効果がある。よって、表5で「PCTFREE＞10」である表は、その値の妥当性を検討する必要がある。

(7) クラスタファクターの効率化

「クラスタファクター」は、索引の値をすべて読み取るときに関連して読み取れる表ブロックが切り替わる回数で、dba_indexes.clustering_factorで確認できる。

値が少ないほどその索引による物理読み取り回数が少なくなる傾向があるため（**図6**）、表の中で最も頻繁に使われて物理読み取りのボトルネックになっている索引が特定された場合、その索引の構成カラム順に表のデータの並べ替え作業を行なうことを推奨する。例えば、LIST3のSQL1で「索引②」がその対象の場合、次の作業を行なう。

```
create table s_msc_sch_work  as select * ⇒
from s_msc_sch ;

truncate table s_msc_sch ;

insert into s_msc_sch
select * from s_msc_sch_work
order by typ_art_id, dev_cd, sit_id, art_⇒
sts_pub, prv_flg, msc_id ;
```

(8) マルチバッファプールの活用

バッファキャッシュを全表検索で多く使う場合、「db_keep_cache_size」や「db_recycle_cache_size」の適切な設定で、バッファキャッシュをより効率的に使うことができる。セグメントごとのアクセス頻度を確認し、どのバッファプールを使うかを事前に定義する必要がある。例えば、めったに使わない表の全データには、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がない場合はRECYCLEバッファプールを使用する。逆に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読み取りを減らしたい場合は、KEEPバッファプールの使用を推奨する。次のように、KEEPバッファプールに指定する。

```
alter table ［table_name］ storage (buffer_pool keep) ;
alter index ［index_name］ storage (buffer_pool keep) ;
```

ただし、各キャッシュのサイズを見積もる場合、バッファのイメージコピーも考慮する必要がある。また、セグメントと各キャッシュのバランスが悪いと、逆に物理読み取りが増えることがあるので十分な検証が必要だ。

(9) メモリの追加

上記の改善策を行なったうえで、それでも物理読み取りのボトルネックが解消されない場合は、作業量に対して絶対的にメモリが足りないということなので、物理メモリの追加と「(3) バッファプールの拡張」の再検討を推奨する。逆にアプリケーションが最適化されていない状態でメモリ追加を先行すると、大量の無駄な物理読み取りは消えないので、同じ現象が解消されずに問題の先送りになることが多い。

ここまでで説明した一般的な対処を、**表6**で筆者なりにコストやリスク、改善効果を考慮して適用の推奨順番を付けてみた。各システムが置かれた状況によって費用やリスク、改善効果は異なってくるので参考程度にしてもらいたい。

索引とSQLチューニング手順

プログラムの処理結果は同じでも、そこにたどり着くまでのロジックは開発者の数だけあり、さまざまだ。DBMSではオプティマイザが作成する実行計画というものが、そのロジックの役割を担っている。すなわち、実行計画によって作業の処理量は大きく左右されるが、そのオプティマイザの判断に最も大きな影響を与えているのが索引とSQLそのものになる。

だから、数時間かかった処理を数分で終わるようにチューニングすることも珍しくはない。索引は頻繁に行なわれる検索パターンのデータを物理的パスとして提供している。SQL（特にSELECT文）は処理の目的（抽出カラム）だけではなく、その過程（FROM、WHERE）のガイドラインも定義しているので、物理読み取りがDB性能のボトルネックと判断された場合はSQL周辺の索引の見直しとSQL文のチューニングを最初に検討すべきだ。逆に処理量が最適化されていない

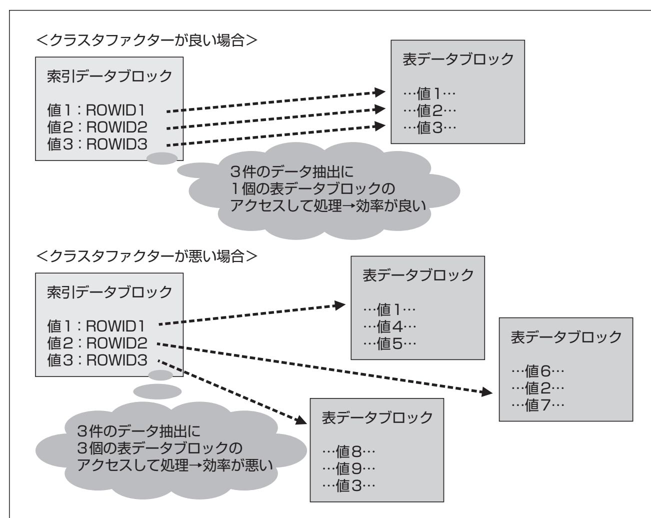

図6：クラスタファクター

表6：物理読み取りボトルネックの一般的な対策

改善案	コスト	リスク	改善効果	適用推奨順
索引の見直し	小	大	大	A
SQLチューニング	小	小	大	A
バッファプールの拡張	小	小	中	A
パーティション化	大	大	中	C
データの再編成	中	中	中	B
PCTFREEの見直し	中	中	小	C
クラスタファクターの効率化	小	中	小	B
マルチバッファプールの活用	中	大	小	D
メモリの増設	大	小	中	C

状態では、ほかの改善策もその力を発揮できない。この考え方を紹介するため、LIST3をサンプルにしてその手順を説明する。

(1) SQL図を作成する

複雑なSQLはもちろん、シンプルなSQLに関しても、その周辺情報を把握しないとどの部分をより効率良くできるかは見えてこない。**図7**のように、SQL図で次のようなSQL周辺情報をまとめると、SQLの解析が一段と分かりやすくなる。

① 抽出カラム

② 表名（アリアスを含む）

③ 表の結合条件

④ 表の単独条件

⑤ PKを含む索引と構成カラム

```
SELECT  index_name, column_name,
FROM   user_ind_columns
WHERE  table_name = ' S_MSC_SCH'
ORDER  BY index_name,
           column_position ;
```

⑥ データ件数

```
SSELECT COUNT (*) FROM s_msc_sch;
```

⑦ リレーションシップ（1:1 or 1:n or m:n）

結合条件でPKが含まれる表を「1 ： 実線」、そうではない表には「多 ： 実線にタコ足記号」を付ける。結合条件がどちらのPKも含まない場合、両側に「多」を付ける

(2) 表のアクセス順番を決定

同じ作業結果でも表のアクセス順によって作業コストが異なるため、最初にアクセスされるデータの絞り込み効果が良いほどコストは少なくなる。例えば、各表の単独条件によって「s_msc_sch ： 1,000件 」「s_prd_sch ： 500,000件 」「s_prd ： 490,000件 」「s_art_msc ： 100件 」件がヒットする場合、「s_msc_sch → s_prd_sch → s_prd → s_art_msc」の順にアクセスさせる。

(3) アクセスコストの最小化

最初にアクセスされる表に対しては単独条件のカラムを、2番目以降にアクセスされる表に対しては結合条件と単独条件のカラムを候補にして、

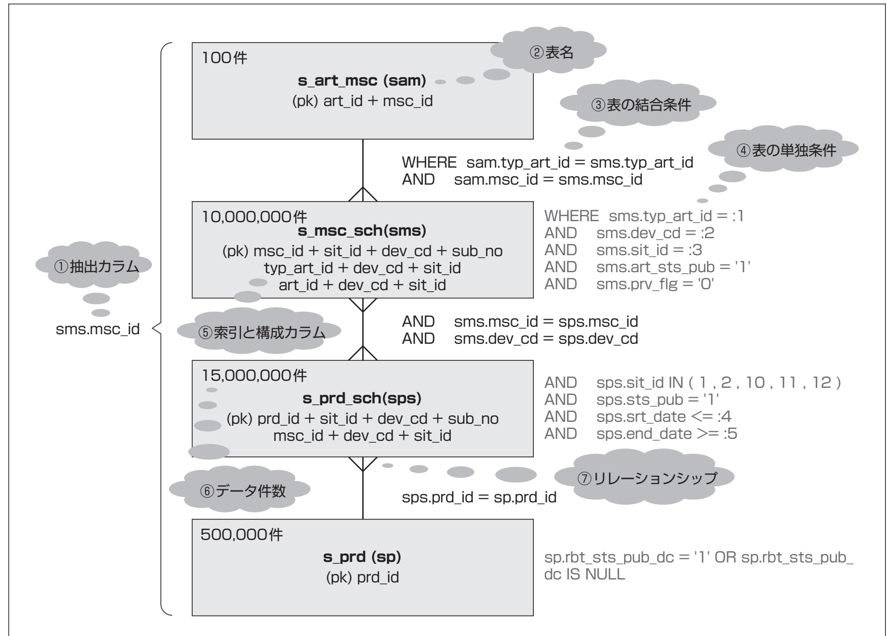

図7 ： SQL図のサンプル（上位SQL1）

```
×: SELECT e.ename FROM emp e WHERE SUBSTR(e.ename, 1, 1) = ‘M’ ;
○: SELECT e.ename FROM emp e WHERE e.ename LIKE ‘M%’ ;

×: SELECT e.ename FROM emp e WHERE TO_CHAR(e.hiredate,’ YYYYMMDD’ ) = ‘20061003’ ;
○: SELECT e.ename FROM emp e
     WHERE e.hiredate BETWEEN TO_DATE( ‘20061003’ ,’ YYYYMMDD’ ) AND TO_DATE( ‘20061003’ ,’ YYYYMMDD’ ) + 0.9999 ;

×: SELECT empno,ename FROM emp WHERE NVL(comm,0) > 10000;
○: SELECT empno,ename FROM emp WHERE comm > 10000;

×: SELECT empno FROM emp WHERE TRUNC(hiredate) >= TRUNC(consdate);
○: SELECT empno FROM emp WHERE hiredate >= TRUNC(consdate);
```

LIST4 ： 索引を使えないSQLと使えるSQLの例

最適の索引を作成する。

例えば、「s_msc_sch → s_prd_sch → s_prd → s_art_msc」順のアクセスの場合、「s_msc_sch」には「typ_art_id、dev_cd、sit_id、art_sts_pub、prv_flg」、「s_prd_sch」には「msc_id、dev_cd、sit_id、sts_pub、srt_date、end_date」が候補になる。表ブロックにアクセスする前に、索引ブロックでデータの絞り込みを行なうことが望ましいので、SQLの最大性能を出すためには候補の全カラムを索引として構成することがベストだ。また、索引を使えなくなるようなSQLがあれば修正しておこう（**LIST4**）。

ただし、ファンクション索引を使用する場合は、関数の形をそのまま維持する。例えば、索引はNULLデータを格納しないので、「sp.rbt_sts_pub_dc = '1' OR sp.rbt_sts_pub_dc IS NULL」は索引を使えないが、「NVL（sp.rbt_sts_pub_dc, '1'） = '1'」に修正することでファンクション索引を使えるようになる。このように特殊な場面で索引に合わせてSQLを修正することもある。

(4) 現行SQLの検証結果を確認

本番並みのボリュームのデータが入っている環境において、現行のSQLのトレースをとって実行時間、実行計画、「physical reads」統計値を確認する（**LIST5**）。単位SQLの正確な性能検証のためには、共有プールおよびバッファキャッシュの初期化（以下のSQLコマンドを参照）などの環境条件を整えることが重要だ。

```
alter system flush shared_pool ;
alter system flush buffer_cache ;
```

(5) 改善案SQLの検証結果を確認

同じ環境で索引改善案を適用して、改善SQLのトレース結果（**LIST6**）から同じ統計値を現行のSQLの結果と比較して改善率を求める。改善検証テストを実施する際には結果の誤差範囲を最小限にするため、抽出データ件数（バインド変数の値の範囲）やバッファキャッシュの初期

```
SQL> set autotrace traceonly
SQL>
SQL> -- LIST3の「改善前（現行）：索引とSQL」に変数を指定して実行

730行が選択されました。

経過: 00:00:08.89

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=28 Card=1 Bytes=102)
   1    0   SORT (GROUP BY) (Cost=28 Card=1 Bytes=102)
   2    1     HASH JOIN (ANTI) (Cost=19 Card=1 Bytes=102)
   3    2       NESTED LOOPS (Cost=10 Card=1 Bytes=90)
   4    3         NESTED LOOPS (Cost=8 Card=1 Bytes=82)
   5    4           TABLE ACCESS (BY INDEX ROWID) OF 'S_MSC_SCH' (Cost=5 Card=1 Bytes=27)
   6    5             INDEX (RANGE SCAN) OF 'S_MSC_SCH_IDX1'(NON-UNIQUE) (Cost=3 Card=2)
   7    4           INLIST ITERATOR
   8    7             TABLE ACCESS (BY INDEX ROWID) OF 'S_PRD_SCH' (Cost=3 Card=1 Bytes=55)
   9    8               INDEX (RANGE SCAN) OF 'S_PRD_SCH_IDX1' (NON-UNIQUE) (Cost=2 Card=1)
  10    3         TABLE ACCESS (BY INDEX ROWID) OF 'S_PRD' (Cost=2 Card=1 Bytes=8)
  11   10           INDEX (UNIQUE SCAN) OF 'S_PRD_PM' (UNIQUE) (Cost=1 Card=1)
  12    2       TABLE ACCESS (FULL) OF 'S_ART_MSC' (Cost=8 Card=181 Bytes=2172)


統計
----------------------------------------------------------
       1759  recursive calls
          0  db block gets
     135232  consistent gets
       2671  physical reads
        136  redo size
      11515  bytes sent via SQL*Net to client
       1031  bytes received via SQL*Net from client
         50  SQL*Net roundtrips to/from client
         25  sorts (memory)
          0  sorts (disk)
        730  rows processed
```

LIST5 ： 現行のSQL

```
SQL> set autotrace traceonly
SQL>
SQL> -- LIST3の「改善案：索引とSQL」に変数を指定して実行

730行が選択されました。

経過: 00:00:00.53

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=26 Card=1 Bytes=102)
   1    0   SORT (GROUP BY) (Cost=26 Card=1 Bytes=102)
   2    1     NESTED LOOPS (Cost=17 Card=1 Bytes=102)
   3    2       NESTED LOOPS (Cost=15 Card=1 Bytes=94)
   4    3         HASH JOIN (ANTI) (Cost=12 Card=1 Bytes=39)
   5    4           INDEX (RANGE SCAN) OF 'S_MSC_SCH_IDX3' (NON-UNIQUE) (Cost=3 Card=1 Bytes=27)
   6    4           TABLE ACCESS (FULL) OF 'S_ART_MSC' (Cost=8 Card=181 Bytes=2172)
   7    3         INLIST ITERATOR
   8    7           INDEX (RANGE SCAN) OF 'S_PRD_SCH_IDX2' (NON-UNIQUE) (Cost=3 Card=1 Bytes=55)
   9    2       INDEX (RANGE SCAN) OF 'S_PRD_PM_IDX1' (NON-UNIQUE) (Cost=2 Card=1 Bytes=8)

統計
----------------------------------------------------------
       2568  recursive calls
          0  db block gets
     152186  consistent gets
        685  physical reads
          0  redo size
      11515  bytes sent via SQL*Net to client
       1031  bytes received via SQL*Net from client
         50  SQL*Net roundtrips to/from client
         31  sorts (memory)
          0  sorts (disk)
        730  rows processed
```

LIST6 ： 改善後のSQL

表7 ： 上位SQLの改善によるDB全体処理の改善効果予測

区分		実行時間（秒）	物理読み取り（ブロック）	SQL単位検証による改善率		備考
				実行時間	物理読み取り	
改善前（9/12 実測統計）	DB全体[A]	111,072.20	6,288,401			
	SQL1	22,224.70	1,094,865			
	SQL2	14,509.85	635,326			
	SQL3	22,193.05	1,730,082			
	SQL4	8,254.80	567,845			
	SQL5	10,591.80	809,052			
	SQL1 ～ 5 合計[B]	77,774.20	4,837,170			
改善後（改善率による予測値）	SQL1	1,324.59	280,833	94.04%	74.35%	
	SQL2	6,696.30	197,459	53.85%	68.92%	
	SQL3	683.55	144,808	96.92%	91.63%	
	SQL4	8,254.80	567,845	0.00%	0.00%	改善対象外
	SQL5	10,591.80	809,052	0.00%	0.00%	改善対象外
	SQL1 ～ 5 合計[C]	27,551.03	1,999,997			
	DB全体[D=A-B+C]	60,849.03	3,451,228			
	DB全体の改善率[(A-D)/A]	45.2%	45.1%			

化有無のケースなども考慮してほしい。また、改善前後の実行パスのイメージを**図8**にまとめているが、大容量の表ブロックへのアクセスがなくなったことが確認できる。

なお、検証結果の改善効果に基づいて運用環境への適用後のDB全体処理時間の改善予測を**表7**にまとめている。

運用環境への適用結果

運用環境にSQL1 ～ 3の改善案の索引を適用した結果、24時間トレンドの接続エラー時間帯で、DB全体の滞留時間、CPU使用率、DB接続数、物理読み取り数が大幅に減少したことが確認された（**図9**）。また、同時間帯のSQL処理統計から、改善対象の上位SQLの処理時間が大幅に改善されたことと、DB全体のSQLの処理時間も大きく改善されたことが確認できた（**表8**）。

単位SQLの性能検証予測より改善効果が大きかったのは、改善対象外のSQLの実行時間が短くなったことから、物理読み取り削減の恩恵が全SQLに広まったためと考えられる。

まとめ

DB性能問題はSQLが引き金となって処理時間の遅延として現われるため、性能診断はDB処理時間の分析から始めて、改善対象のSQLのピックアップまで行なう。同様の考え方で、DB性能改善はSQL周辺のチューニングから始めて、DB処理時間の結果分析で完了する（**図10**）。

システムの運用開始後に予想されなかった性能問題が発生するのは、運用前にはユーザーの利用パターンを正確に予測できないことが多いからだ。そのため、利用統計が把握されている運用開始後には、そのデータがなかった運用前とは違ってより正確な性能診断と改善が可能になる。しかも、運用前はすべての箇所に一定のコストで一般的なガイドラインを適用することで性

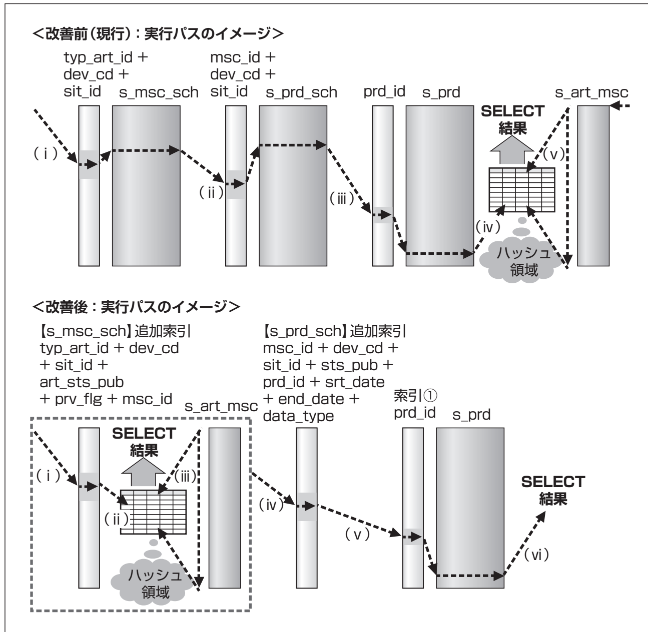

図8：実行パスのイメージ

Column バッファキャッシュヒット率が高ければI/Oの効率が良い?

ディスクI/Oの健康度を計る伝統的な指標としてはバッファキャッシュヒット率(BCHR)が多く使われ、このヒット率が90%以上となるようにバッファサイズを増加させることが一般的なチューニングガイドとして推奨されている。しかし、性能改善前にすでに99%を超えている本稿の事例ではどうだろうか。ヒット率の高さがそのまま性能に影響しているわけではないことが分かる。また、逆に世の中には90%を超えなくても性能要件を満たしているシステムが多く存在するので、性能診断を行なう際には当指標は1つの参考程度にしてほしい。この事例の結果ではヒット率は「0.27%」しか改善されなかったものの、論理読み取りとともに物理読み取りが大幅に改善された。

区分	session logical reads	physical reads	physical reads direct	physical reads direct (lob)	BCHR
9月12日(A：改善前)	1,316,410,984	6,871,002	42,281	0	99.481%
10月24日(B：改善後)	1,037,153,952	2,631,860	54,489	0	99.751%
差分(A-B)	279,257,032	4,239,142	-12,208	0	-0.27%

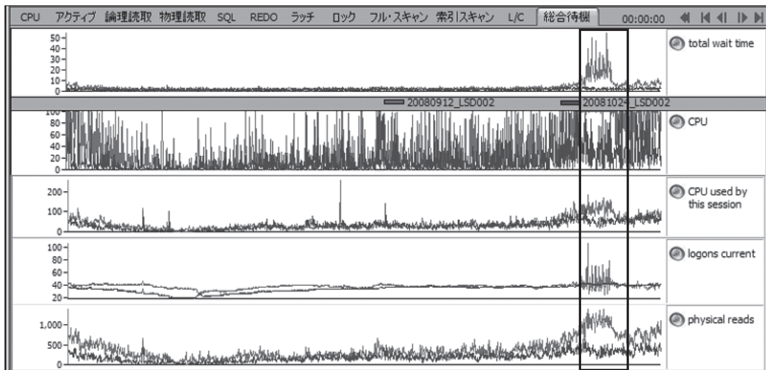

図9：統計指標の24時間トレンド比較(性能改善前後の時期)

表8：性能改善前後(9/12 vs.10/24)「20:00～22:00」時間帯のDB処理統計比較

タイム区分	9月12日	10月24日	改善率
DB処理時間[A＝B＋C]	103,016.30	15,048.70	85.4%
CPU処理時間[B]	7,550.30	4,191.10	44.5%
合計待機時間[C≒D～I]	95,466.00	10,857.60	88.6%
待機領域：SQL解析[D]	4,368.40	41.60	99.0%
待機領域：物理読取[E]	89,481.80	10,681.30	88.1%
待機領域：論理読取[F]	619.60	2.50	99.6%
待機領域：ロック[G]	221.80	1.00	99.5%
待機領域：REDOログ[H]	344.90	33.40	90.3%
待機領域：制御ファイル[I]	427.50	96.80	77.4%

能対策を行なうが、ボトルネック箇所が明確になっている運用環境ではその重要さに見合ったコスト(作業時間／CPU／メモリなど)を集中的にかけることが最も効果的なアプローチになる。

昨今は技術書籍の充実やエンジニアのスキルアップによって改善方法そのものはある程度まで浸透しているが、実態としては何を、どれくらい改善すべきかという点で悩んでいるケースが少なくないと思う。特に運用環境では、「How tune」から「Where and What tune」へと思考回路を変えてみることで、今まで見えなかった部分が突然鮮明になることもある。そして、正確な診断と予測はシステムの延命活動を促して、不況というこの時勢を乗り切る原動力にもなるだろうと確信している。 DBM

図10：性能診断と性能改善のアプローチ

■参考資料
- DBマガジン2007年9月号「特集2 OWIによるOracleの性能改善と障害対策」
- MaxGauge3.1 イベントヘルプ
- http://support.oracle.co.jp

■使用ツール
MaxGauge 3.1 for Oracle(日本エクセム)

※本記事で使用した診断と改善結果以外のデータは一部編集されています。

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

金 圭福(きむぎゅうぼく)
日本エクセム株式会社所属。AP開発、DBAの経験を経て、現在はDB性能とトラブル関連技術サポートおよびコンサルティングを担当している。