8

Building Effective Applications

Effective applications are scalable and use recommended programming and security practices.

> **See Also:**
>
> *Oracle Database Development Guide* for more information about creating and deploying applications that are optimized for Oracle Database

Building Scalable Applications

Design your applications to use the same resources, regardless of user populations and data volumes, and not to overload system resources.

About Scalable Applications

A **scalable** application can process a larger workload with a proportional increase in system resource usage.

A **scalable** application can process a larger workload with a proportional increase in system resource usage. For example, if you double its workload, a scalable application uses twice as many system resources.

An **unscalable** application exhausts a system resource; therefore, if you increase the application workload, no more throughput is possible. Unscalable applications result in fixed throughputs and poor response times.

Examples of resource exhaustion are:

- Hardware exhaustion
- Table scans in high-volume transactions causing inevitable disk input/output (I/O) shortages
- Excessive network requests causing network and scheduling bottlenecks
- Memory allocation causing paging and swapping
- Excessive process and thread allocation causing operating system thrashing

Design your applications to use the same resources, regardless of user populations and data volumes, and not to overload system resources.

Using Bind Variables to Improve Scalability

Bind variables, used correctly, let you develop efficient, scalable applications.

A **bind variable** is a placeholder in a SQL statement that must be replaced with a valid value or value address for the statement to run successfully. By using bind variables, you can write a SQL statement that accepts inputs or parameters at run time.

Just as a subprogram can have parameters, whose values are supplied by the invoker, a SQL statement can have bind variable placeholders, whose values (called bind variables) are supplied at runtime. Just as a subprogram is compiled once and then run many times with different parameters, a SQL statement with bind variable placeholders is hard parsed once and then soft parsed with different bind variables.

A **hard parse**, which includes optimization and row source generation, is a very CPU-intensive operation. A **soft parse**, which skips optimization and row source generation and proceeds straight to execution, is usually much faster than a hard parse of the same statement. (For an overview of SQL processing, which includes the difference between a hard and soft parse, see *Oracle Database Concepts*.)

Not only is a hard parse a CPU-intensive operation, it is an unscalable operation, because it cannot be done concurrently with many other operations. For more information about concurrency and scalability, see "About Concurrency and Scalability".

Example 8-1 shows the performance difference between a query without a bind variable and a semantically equivalent query with a bind variable. The former is slower and uses many more latches (for information about how latches affect scalability, see "About Latches and Concurrency"). To collect and display performance statistics, the example uses the Runstats tool, described in "Comparing Programming Techniques with Runstats".

> **Note:**
>
> - Example 8-1 shows the performance cost *for a single user*. As more users are added, the cost escalates rapidly.
> - The result of Example 8-1 was produced with this setting:
>
>   ```
>   SET SERVEROUTPUT ON FORMAT TRUNCATED
>   ```

> **Note:**
>
> - Using bind variables instead of string literals is the most effective way to make your code invulnerable to SQL injection attacks. For details, see *Oracle Database PL/SQL Language Reference*.
> - Bind variables sometimes reduce the efficiency of data warehousing systems. Because most queries take so long, the optimizer tries to produce the best plan for each query rather than the best generic query. Using bind variables sometimes forces the optimizer to produce the best generic query. For information about improving performance in data warehousing systems, see *Oracle Database Data Warehousing Guide*.

Although soft parsing is more efficient than hard parsing, the cost of soft parsing a statement many times is still very high. To maximize the efficiency and scalability of

your application, minimize parsing. The easiest way to minimize parsing is to use PL/SQL.

Example 8-1 Bind Variable Improves Performance

```
CREATE TABLE t ( x VARCHAR2(5) );

DECLARE
  TYPE rc IS REF CURSOR;
  l_cursor rc;
BEGIN
  runstats_pkg.rs_start;  -- Collect statistics for query without bind variable

  FOR i IN 1 .. 5000 LOOP
    OPEN l_cursor FOR 'SELECT x FROM t WHERE x = ' || TO_CHAR(i);
    CLOSE l_cursor;
  END LOOP;

  runstats_pkg.rs_middle;  -- Collect statistics for query with bind variable

  FOR i IN 1 .. 5000 LOOP
    OPEN l_cursor FOR 'SELECT x FROM t WHERE x = :x' USING i;
    CLOSE l_cursor;
  END LOOP;

  runstats_pkg.rs_stop(500);  -- Stop collecting statistics
end;
/
```

Result is similar to:

```
Run 1 ran in 740 hsec
Run 2 ran in 30 hsec
Run 1 ran in 2466.67% of the time of run 2

Name                                  Run 1         Run 2    Difference
STAT...recursive cpu usage              729            19          -710
STAT...CPU used by this sessio          742            30          -712
STAT...parse time elapsed             1,051             4        -1,047
STAT...parse time cpu                 1,066             2        -1,064
STAT...session cursor cache hi            1         4,998         4,997
STAT...table scans (short tabl        5,000             1        -4,999
STAT...parse count (total)           10,003         5,004        -4,999
LATCH.session idle bit                5,003             3        -5,000
LATCH.session allocation              5,003             3        -5,000
STAT...execute count                 10,003         5,003        -5,000
STAT...opened cursors cumulati       10,003         5,003        -5,000
STAT...parse count (hard)            10,001             5        -9,996
STAT...CCursor + sql area evic       10,000             1        -9,999
STAT...enqueue releases              10,008             7       -10,001
STAT...enqueue requests              10,009             7       -10,002
STAT...calls to get snapshot s       20,005         5,006       -14,999
STAT...calls to kcmgcs               20,028            35       -19,993
STAT...consistent gets pin (fa       20,013            17       -19,996
LATCH.call allocation                20,002             6       -19,996
STAT...consistent gets from ca       20,014            18       -19,996
STAT...consistent gets               20,014            18       -19,996
STAT...consistent gets pin           20,013            17       -19,996
LATCH.simulator hash latch           20,014            11       -20,003
STAT...session logical reads         20,080            75       -20,005
LATCH.shared pool simulator          20,046             5       -20,041
LATCH.enqueue hash chains            20,343            15       -20,328
STAT...recursive calls               40,015        15,018       -24,997
```

```
LATCH.cache buffers chains            40,480           294       -40,186
STAT...session pga memory max        131,072        65,536       -65,536
STAT...session pga memory            131,072        65,536       -65,536
LATCH.row cache objects              165,209           139      -165,070
STAT...session uga memory max        219,000             0      -219,000
LATCH.shared pool                    265,108           152      -264,956
STAT...logical read bytes from   164,495,360       614,400  -163,880,960

Run 1 latches total compared to run 2 -- difference and percentage
        Run 1         Run 2          Diff       Pct
      562,092           864      -561,228  2,466.67%

PL/SQL procedure successfully completed.
```

Using PL/SQL to Improve Scalability

Certain PL/SQL features can help you to improve application scalability.

How PL/SQL Minimizes Parsing

PL/SQL, which is optimized for database access, silently caches statements. In PL/SQL, when you close a cursor, the cursor closes from your perspective—that is, you cannot use it where an open cursor is required—but PL/SQL actually keeps the cursor open and caches its statement.

If you use the cached statement again, PL/SQL uses the same cursor, thereby avoiding a parse. (PL/SQL closes cached statements if necessary—for example, if your program must open another cursor but doing so would exceed the init.ora setting of OPEN_CURSORS.)

PL/SQL can silently cache only SQL statements that cannot change at runtime.

About the EXECUTE IMMEDIATE Statement

The EXECUTE IMMEDIATE statement builds and runs a dynamic SQL statement in a single operation.

The basic syntax of the EXECUTE IMMEDIATE statement is:

```
EXECUTE IMMEDIATE sql_statement
```

sql_statement is a string that represents a SQL statement. If sql_statement has the same value every time the EXECUTE IMMEDIATE statement runs, then PL/SQL can cache the EXECUTE IMMEDIATE statement. If sql_statement can be different every time the EXECUTE IMMEDIATE statement runs, then PL/SQL cannot cache the EXECUTE IMMEDIATE statement.

See Also:

- *Oracle Database PL/SQL Language Reference* for information about EXECUTE IMMEDIATE
- "About the DBMS_SQL Package"

About OPEN FOR Statements

The OPEN FOR statement has the following basic syntax.

The basic syntax of the OPEN FOR statement is:

```
OPEN cursor_variable FOR query
```

Your application can open cursor_variable for several different queries before closing it. Because PL/SQL cannot determine the number of different queries until runtime, PL/SQL cannot cache the OPEN FOR statement.

If you do not need to use a cursor variable, then use a declared cursor, for both better performance and ease of programming. For details, see *Oracle Database Development Guide*.

See Also:

- *Oracle Database PL/SQL Language Reference* for information about OPEN FOR
- "About Cursor Variables"
- "About Cursors"

About the DBMS_SQL Package

The DBMS_SQL package is an API for building, running, and describing dynamic SQL statements. You must use the DBMS_SQL package instead of the EXECUTE IMMEDIATE statement if the PL/SQL compiler cannot determine at compile time the number or types of output host variables (select list items) or input bind variables.

The DBMS_SQL package is an API for building, running, and describing dynamic SQL statements. Using the DBMS_SQL package takes more effort than using the EXECUTE IMMEDIATE statement, but you must use the DBMS_SQL package if the PL/SQL compiler cannot determine at compile time the number or types of output host variables (select list items) or input bind variables.

See Also:

- *Oracle Database PL/SQL Language Reference* for more information about when to use the DBMS_SQL package
- *Oracle Database PL/SQL Packages and Types Reference* for complete information about the DBMS_SQL package
- "About the EXECUTE IMMEDIATE Statement"

About Bulk SQL

Bulk SQL reduces the number of "round trips" between PL/SQL and SQL, thereby using fewer resources.

Without bulk SQL, you retrieve one row at a time from the database (SQL), process it (PL/SQL), and return it to the database (SQL). With bulk SQL, you retrieve a set of rows from the database, process the set of rows, and then return the whole set to the database.

Oracle recommends using Bulk SQL when you retrieve multiple rows from the database *and* return them to the database, as in Example 8-2. You do not need bulk SQL if you retrieve multiple rows but do not return them; for example:

```
FOR x IN (SELECT * FROM t WHERE ... )  -- Retrieve row set (implicit array fetch)
  LOOP
    DBMS_OUTPUT.PUT_LINE(t.x);          -- Process rows but do not return them
  END LOOP;
```

Example 8-2 loops through a table t with a column object_name, retrieving sets of 100 rows, processing them, and returning them to the database. (Limiting the bulk FETCH statement to 100 rows requires an explicit cursor.)

Example 8-3 does the same job as Example 8-2, without bulk SQL.

As these TKPROF reports for Example 8-2 and Example 8-3 show, using bulk SQL for this job uses almost 50% less CPU time:

```
SELECT ROWID RID, OBJECT_NAME FROM T T_BULK

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      721      0.17       0.17          0      22582          0       71825
********************************************************************************
UPDATE T SET OBJECT_NAME = :B1 WHERE ROWID = :B2

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute    719     12.83      13.77          0      71853      74185       71825
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      720     12.83      13.77          0      71853      74185       71825


SELECT ROWID RID, OBJECT_NAME FROM T T_SLOW_BY_SLOW

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      721      0.17       0.17          0      22582          0       71825
********************************************************************************
UPDATE T SET OBJECT_NAME = :B2 WHERE ROWID = :B1

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute  71824     21.25      22.25          0      71836      73950       71824
Fetch        0      0.00       0.00          0          0          0           0
```

```
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total    71825     21.25      22.25          0      71836      73950       71824
```

However, using bulk SQL for this job uses more CPU time—and more code—than using a single SQL statement, as this `TKPROF` report shows:

```
UPDATE T SET OBJECT_NAME = SUBSTR(OBJECT_NAME,2) || SUBSTR(OBJECT_NAME,1,1)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      1.30       1.44          0       2166      75736       71825
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2      1.30       1.44          0       2166      75736       71825
```

Example 8-2 Bulk SQL

```
CREATE OR REPLACE PROCEDURE bulk AS
  TYPE ridArray IS TABLE OF ROWID;
  TYPE onameArray IS TABLE OF t.object_name%TYPE;

  CURSOR c is SELECT ROWID rid, object_name  -- explicit cursor
              FROM t t_bulk;

  l_rids    ridArray;
  l_onames  onameArray;
  N         NUMBER := 100;
BEGIN
  OPEN c;
  LOOP
    FETCH c BULK COLLECT
    INTO l_rids, l_onames LIMIT N;    -- retrieve N rows from t

    FOR i in 1 .. l_rids.COUNT
      LOOP                            -- process N rows
        l_onames(i) := substr(l_onames(i),2) || substr(l_onames(i),1,1);
      END LOOP;

      FORALL i in 1 .. l_rids.count  -- return processed rows to t
        UPDATE t
        SET object_name = l_onames(i)
        WHERE ROWID = l_rids(i);
        EXIT WHEN c%NOTFOUND;
  END LOOP;
  CLOSE c;
END;
/
```

Example 8-3 Without Bulk SQL

```
CREATE OR REPLACE PROCEDURE slow_by_slow AS
BEGIN
  FOR x IN (SELECT rowid rid, object_name FROM t t_slow_by_slow)
    LOOP
      x.object_name := substr(x.object_name,2) || substr(x.object_name,1,1);

      UPDATE t
      SET object_name = x.object_name
      WHERE rowid = x.rid;
```

```
    END LOOP;
END;
```

> **See Also:**
>
> - *Oracle Database Development Guide* for an overview of bulk SQL
> - *Oracle Database Development Guide* for more specific information about when to use bulk SQL
> - *Oracle Database PL/SQL Language Reference* for more information about bulk SQL

About Concurrency and Scalability

Concurrency is the simultaneous execution of multiple transactions. A **scalable** application can process a larger workload with a proportional increase in system resource usage.

Statements within concurrent transactions can update the same data. The better your application handles concurrency, the more scalable it is. For example, if you double its workload, a scalable application uses twice as many system resources.

Concurrent transactions must produce meaningful and consistent results. Therefore, a multiuser database must provide the following:

- **Data concurrency**, which ensures that users can access data at the same time.
- **Data consistency**, which ensures that each user sees a consistent view of the data, including visible changes from their own transactions and committed transactions of other users

Oracle Database maintains data consistency by using a multiversion consistency model and various types of locks and transaction isolation levels. For an overview of the Oracle Database locking mechanism, see *Oracle Database Concepts*. For an overview of Oracle Database transaction isolation levels, see *Oracle Database Concepts*.

To describe consistent transaction behavior when transactions run concurrently, database researchers have defined the **serializable** transaction isolation category. A **serializable transaction** operates in an environment that appears to be a single-user database. Serializable transactions are desirable in specific cases, but for 99% of the work load, read committed isolation is most useful.

Oracle Database has features that improve concurrency and scalability—for example, sequences, latches, nonblocking reads and writes, and shared SQL.

> **See Also:**
>
> *Oracle Database Concepts* for more information about data concurrency and consistency

About Sequences and Concurrency

Sequences eliminate serialization, thereby improving the concurrency and scalability of your application.

A **sequence** is a schema object from which multiple users can generate unique integers, which is very useful when you need unique primary keys.

Without sequences, unique primary key values must be produced programmatically. A user gets a new primary key value by selecting the most recently produced value and incrementing it. This technique requires a lock during the transaction and causes multiple users to wait for the next primary key value—that is, the transactions serialize. Sequences eliminate serialization, thereby improving the concurrency and scalability of your application.

See Also:

- *Oracle Database Concepts* for information about concurrent access to sequences
- "Creating and Managing Sequences"

About Latches and Concurrency

An increase in latches means more concurrency-based waits, and therefore a decrease in scalability.

A **latch** is a simple, low-level serialization mechanism that coordinates multiuser access to shared data structures. Latches protect shared memory resources from corruption when accessed by multiple processes.

An increase in latches means more concurrency-based waits, and therefore a decrease in scalability. If you can use either an approach that runs slightly faster during development or one that uses fewer latches, use the latter.

See Also:

- *Oracle Database Concepts* for information about latches
- *Oracle Database Concepts* for information about mutexes, which are like latches for single objects

About Nonblocking Reads and Writes and Concurrency

In Oracle Database, **nonblocking reads and writes** let queries run concurrently with changes to the data they are reading, without blocking or stopping. Nonblocking reads and writes let one session read data while another session is changing that data.

About Shared SQL and Concurrency

Oracle Database compiles a SQL statement into an executable object once, and then other sessions can reuse the object for as long as it exists. This Oracle Database feature, called **shared SQL**, lets the database do very resource-intensive operations—compiling and optimizing SQL statements—only once, instead of every time a session uses the same SQL statement.

> **See Also:**
>
> *Oracle Database Concepts* for more information about shared SQL

Limiting the Number of Concurrent Sessions

The more concurrent sessions you have, the more concurrency-based waits you have, and the slower your response time is.

If your computer has *n* CPU cores, then at most *n* sessions can really be concurrently active. Each additional "concurrent" session must wait for a CPU core to be available before it can become active. If some waiting sessions are waiting only for I/O, then increasing the number of concurrent sessions to slightly more than *n* might slightly improve runtime performance. However, increasing the number of concurrent sessions too much will significantly reduce runtime performance.

The SESSIONS initialization parameter determines the maximum number of concurrent users in the system. For details, see *Oracle Database Reference*.

> **See Also:**
>
> `http://www.youtube.com/watch?v=xNDnVOCdvQ0` for a video that shows the effect of reducing the number of concurrent sessions on a computer with 12 CPU cores from thousands to 96

Comparing Programming Techniques with Runstats

The Runstats tool lets you compare the performance of two programming techniques to see which is better.

About Runstats

The Runstats tool lets you compare the performance of two programming techniques to see which is better.

Runstats measures:

- Elapsed time for each technique in hundredths of seconds (hsec)
- Elapsed time for the first technique as a percentage of that of the second technique

- System statistics for the two techniques (for example, parse calls)
- Latching for the two techniques

Of the preceding measurements, the most important is latching (see "About Latches and Concurrency").

> **See Also:**
>
> Example 8-1, which uses Runstats

Setting Up Runstats

The Runstats tool is implemented as a package that uses a view and a temporary table.

> **Note:**
>
> For step 1 of the following procedure, you need the SELECT privilege on the dynamic performance views V$STATNAME, V$MYSTAT, and V$LATCH. If you cannot get this privilege, then have someone who has the privilege create the view in step 1 and grant you the SELECT privilege on it.

To set up the Runstats tool:

1. Create the view that Runstats uses:

```
CREATE OR REPLACE VIEW stats
AS SELECT 'STAT...' || a.name name, b.value
FROM V$STATNAME a, V$MYSTAT b
WHERE a.statistic# = b.statistic#
UNION ALL
SELECT 'LATCH.' || name, gets
FROM V$LATCH;
```

2. Create the temporary table that Runstats uses:

```
DROP TABLE run_stats;

CREATE GLOBAL TEMPORARY TABLE run_stats
( runid VARCHAR2(15),
  name VARCHAR2(80),
  value INT )
ON COMMIT PRESERVE ROWS;
```

3. Create this package specification:

```
CREATE OR REPLACE PACKAGE runstats_pkg
AS
  PROCEDURE rs_start;
  PROCEDURE rs_middle;
  PROCEDURE rs_stop( p_difference_threshold IN NUMBER DEFAULT 0 );
end;
/
```

The parameter `p_difference_threshold` controls the amount of statistics and latching data that Runstats displays. Runstats displays data only when the difference for the two techniques is greater than `p_difference_threshold`. By default, Runstats displays all data.

4. Create this package body:

```
CREATE OR REPLACE PACKAGE BODY runstats_pkg
AS
  g_start NUMBER;
  g_run1 NUMBER;
  g_run2 NUMBER;

  PROCEDURE rs_start
  IS
  BEGIN
    DELETE FROM run_stats;

    INSERT INTO run_stats
    SELECT 'before', stats.* FROM stats;

    g_start := DBMS_UTILITY.GET_TIME;
  END rs_start;

  PROCEDURE rs_middle
  IS
  BEGIN
    g_run1 := (DBMS_UTILITY.GET_TIME - g_start);

    INSERT INTO run_stats
    SELECT 'after 1', stats.* FROM stats;

    g_start := DBMS_UTILITY.GET_TIME;
  END rs_middle;

  PROCEDURE rs_stop( p_difference_threshold IN NUMBER DEFAULT 0 )
  IS
  BEGIN
    g_run2 := (DBMS_UTILITY.GET_TIME - g_start);

    DBMS_OUTPUT.PUT_LINE
      ('Run 1 ran in ' || g_run1 || ' hsec');

    DBMS_OUTPUT.PUT_LINE
      ('Run 2 ran in ' || g_run2 || ' hsec');

    DBMS_OUTPUT.PUT_LINE
      ('Run 1 ran in ' || round(g_run1/g_run2*100, 2) || '% of the time of
run 2');

    DBMS_OUTPUT.PUT_LINE( CHR(9) );

    INSERT INTO run_stats
    SELECT 'after 2', stats.* FROM stats;

    DBMS_OUTPUT.PUT_LINE
      ( RPAD( 'Name', 30 ) ||
        LPAD( 'Run 1', 14) ||
        LPAD( 'Run 2', 14) ||
        LPAD( 'Difference', 14)
      );
```

```
    FOR x IN
    ( SELECT RPAD( a.name, 30 ) ||
             TO_CHAR( b.value - a.value, '9,999,999,999' ) ||
             TO_CHAR( c.value - b.value, '9,999,999,999' ) ||
             TO_CHAR( ( (c.value - b.value) - (b.value - a.value)),
               '9,999,999,999' ) data
      FROM run_stats a, run_stats b, run_stats c
      WHERE a.name = b.name
        AND b.name = c.name
        AND a.runid = 'before'
        AND b.runid = 'after 1'
        AND c.runid = 'after 2'
        AND (c.value - a.value) > 0
        AND abs((c.value - b.value) - (b.value - a.value)) >
          p_difference_threshold
     ORDER BY ABS((c.value - b.value) - (b.value - a.value))
    ) LOOP
        DBMS_OUTPUT.PUT_LINE( x.data );
    END LOOP;

    DBMS_OUTPUT.PUT_LINE( CHR(9) );

    DBMS_OUTPUT.PUT_LINE(
      'Run 1 latches total compared to run 2 -- difference and percentage' );

    DBMS_OUTPUT.PUT_LINE
      ( LPAD( 'Run 1', 14) ||
        LPAD( 'Run 2', 14) ||
        LPAD( 'Diff', 14) ||
        LPAD( 'Pct', 10)
      );

    FOR x IN
    ( SELECT TO_CHAR( run1, '9,999,999,999' ) ||
             TO_CHAR( run2, '9,999,999,999' ) ||
             TO_CHAR( diff, '9,999,999,999' ) ||
             TO_CHAR( ROUND( g_run1/g_run2*100, 2), '99,999.99' ) || '%' data
      FROM ( SELECT SUM (b.value - a.value) run1,
                    SUM (c.value - b.value) run2,
                    SUM ( (c.value - b.value) - (b.value - a.value)) diff
             FROM run_stats a, run_stats b, run_stats c
             WHERE a.name = b.name
               AND b.name = c.name
               AND a.runid = 'before'
               AND b.runid = 'after 1'
               AND c.runid = 'after 2'
               AND a.name like 'LATCH%'
           )
    ) LOOP
        DBMS_OUTPUT.PUT_LINE( x.data );
    END LOOP;

  END rs_stop;

END;
/
```

> **See Also:**
>
> - "Creating Views"
> - "Creating Tables"
> - "Tutorial: Creating a Package Specification"
> - "Tutorial: Creating a Package Body"
> - *Oracle Database Reference* for information about dynamic performance views

Using Runstats

This topic gives the syntax for using the Runstats tool.

To use Runstats to compare two programming techniques, invoke the runstats_pkg procedures from an anonymous block, using this syntax:

```
[ DECLARE local_declarations ]
BEGIN
  runstats_pkg.rs_start;
  code_for_first_technique
  runstats_pkg.rs_middle;
  code_for_second_technique
  runstats_pkg.rs_stop(n);
END;
/
```

> **See Also:**
>
> Example 8-1, which uses Runstats

Real-World Performance and Data Processing Techniques

A common task in database applications in a data warehouse environment is querying or modifying a huge data set. The problem for application developers is how to achieve high performance when processing large data sets.

Processing techniques fall into two categories: iterative, and set-based. Over years of testing, the Real-World Performance group has discovered that **set-based processing techniques perform orders of magnitude better** for database applications that process large data sets.

This topic includes the following major subtopics:.

About Iterative Data Processing

In iterative processing, applications use conditional logic to loop through a set of rows.

Typically, although not necessarily, iterative processing uses a client/server model as follows:

1. Transfer a group of rows from the database server to the client application.
2. Process the group within the client application.
3. Transfer the processed group back to the database server.

You can implement iterative algorithms using three main techniques: row-by-row processing, array processing, and manual parallelism.

Iterative Processing: Row-By-Row

In row-by-row processing, a single process loops through a data set and operates on a single row a time. In a typical implementation, the application retrieves each row from the database, processes it in the middle tier, and then sends the row back to the database, which runs DML and commits.

Assume that your functional requirement is to query an external table named ext_scan_events, and then insert its rows into a heap-organized staging table named stage1_scan_events. The following PL/SQL block uses a row-by-row technique to meet this requirement:

```
declare
  cursor c is select s.* from ext_scan_events s;
  r c%rowtype;
begin
  open c;
  loop
    fetch c into r;
    exit when c%notfound;
    insert into stage1_scan_events d values r;
    commit;
  end loop;
  close c;
end;
```

The row-by-row technique has the following advantages:

- It performs well on small data sets.
- The looping algorithm is familiar to all professional developers, easy to write quickly, and easy to understand.

The row-by-row technique has the following disadvantages:

- Processing time can be unacceptably long for large data sets.
- The application runs serially, and thus cannot exploit the native parallel processing features of Oracle Database running on modern hardware.

See Also: RWP #7 Set-Based Processing

Iterative Processing: Arrays

Array processing is identical to row-by-row processing, except that it processes a group of rows in each iteration rather than a single row.

Assume that your functional requirement is the same as in Example X-X: query an external table named ext_scan_events, and then insert its rows into a heap-organized staging table

named stage1_scan_events. The following PL/SQL block uses an array technique to meet this requirement:

```
declare
  cursor c is select s.* from ext_scan_events s;
  type t is table of c%rowtype index by binary_integer;
  a t;
  rows binary_integer := 0;
begin
  open c;
  loop
    fetch c bulk collect into a limit array_size;
    exit when a.count = 0;
    forall i in 1..a.count
      insert into stage1_scan_events d values a(i);
    commit;
  end loop;
  close c;
end;
```

The preceding code differs from the equivalent row-by-row code in using a BULK COLLECT operator in the FETCH STATEMENT, which is limited by the `array_size` value of type PLS_INTEGER. For example, if `array_size` is set to 100, then the application fetches rows in groups of 100.

The array technique has the following advantages over the row-by-row technique:

- The array enables the application to process a group of rows at the same time, which means that it reduces network round trips, COMMIT time, and the code path in the client and server.
- The database is more efficient because the server process batches the inserts, and commits after every group of inserts rather than after every insert.

The disadvantages of this technique are the same as for row-by-row processing. Processing time can be unacceptable for large data sets. Also, the application must run serially on a single CPU core, and thus cannot exploit the native parallelism of Oracle Database.

Iterative Processing: Manual Parallelism

Manual parallelism uses the same iterative algorithm as row-by-row and array processing, but enables multiple server processes to divide the work and run in parallel.

Assume the functional requirement is the same as in the row-by-row and array examples. The primary differences are as follows:

- The scan event records are stored in a mass of flat files.
- 32 server processes must run in parallel, with each server process querying a different external table.
- You use PL/SQL to achieve the parallelism by executing 32 threads of the same PL/SQL program, with each thread running simultaneously as a separate job managed by Oracle Scheduler. A job is the combination of a schedule and a program.

The following PL/SQL code uses manual parallellism:

```
declare
  sqlstmt varchar2(1024) := q'[
-- BEGIN embedded anonymous block
  cursor c is select s.* from ext_scan_events_${thr} s;
  type t is table of c%rowtype index by binary_integer;
  a t;
  rows binary_integer := 0;
begin
  for r in (select ext_file_name from ext_scan_events_dets where
ora_hash(file_seq_nbr,${thrs}) = ${thr})
  loop
    execute immediate
      'alter table ext_scan_events_${thr} location' || '(' ||
r.ext_file_name || ')';
    open c;
    loop
      fetch c bulk collect into a limit ${array_size};
      exit when a.count = 0;
      forall i in 1..a.count
        insert into stage1_scan_events d values a(i);
      commit;
--  demo instrumentation
      rows := rows + a.count; if rows > 1e3 then exit when not
sd_control.p_progress('loading','userdefined',rows); rows := 0; end if;
    end loop;
    close c;
  end loop;
end;
-- END   embedded anonymous block
]';

begin
  sqlstmt := replace(sqlstmt, '${array_size}', to_char(array_size));
  sqlstmt := replace(sqlstmt, '${thr}', thr);
  sqlstmt := replace(sqlstmt, '${thrs}', thrs);
  execute immediate sqlstmt;
end;
```

The ORA_HASH function divides the ext_scan_events_dets table into 32 evenly distributed buckets, and then the SELECT statement retrieves the file names for bucket 0. For each file name in the bucket, the program sets the location of the external table to this file name. The program then uses batch processing to query the external table, insert into the staging table, and then commit.

While job 1 is executing, the other 31 Oracle Scheduler jobs run in parallel. In this way, each job simultaneously reads a different subset of the scan event files, and inserts the records from its subset into the same staging table.

The manual parallelism technique has the following advantages over the alternative iterative techniques:

- It performs far better on large data sets because server processes are working in parallel.

- When the application uses ORA_HASH to distribute the workload, each thread of execution can access the same amount of data, which means that the parallel processes can finish at the same time.

The manual parallelism technique has the following disadvantages:

- The code is relatively lengthy, complicated, and difficult to understand.
- The application must perform a certain amount of preparatory work before the database can begin the main work, which is processing the rows in parallel.
- If multiple threads perform the same operations on a common set of database objects, then lock and latch contention is possible.
- Parallel processing consumes significant CPU resources compared to the competing iterative techniques.

See Also: RWP #8: Set-Based Parallel Processing

About Set-Based Processing

Set-based processing is a SQL technique that processes a data set inside the database.

In a set-based model, the SQL statement defines the result, and allows the database to determine the most efficient way to obtain it. In contrast, iterative algorithms use conditional logic to pull each each row or group of rows from the database to the client application, process the data on the client, and then send the data back to the database. Set-based processing eliminates the network round-trip and database API overhead because the data never leaves the database.

Assume the same functional requirement as in the previous examples. The following SQL statements meet this requirement using a set-based algorithm:

```
alter session enable parallel dml;
insert /*+ APPEND */ into stage1_scan_events d
  select s.* from ext_scan_events s;
commit;
```

Because the INSERT statement contains a subquery of the ext_scan_events table, a single SQL statement reads and writes all rows. Also, the application runs a single COMMIT after the database has inserted all rows. In contrast, iterative applications run a COMMIT after the insert of each row or each group of rows.

The set-based technique has significant advantages over iterative techniques:

- As demonstrated in Real-World Performance demonstrations and classes, the performance on large data sets is orders of magnitude faster. It is not unusual for the run time of a program to drop from several hours to several seconds.
- A side-effect of the orders of magnitude increase in processing speed is that DBAs can eliminate long-running and error-prone batch jobs, and innovate business processes in real time.
- The length of the code is significantly shorter, a short as two or three lines of code, because SQL defines the result and not the access method.
- In contrast to manual parallelism, parallel DML is optimized for performance because the database, rather than the application, manages the processes.

- When joining data sets, the database automatically uses highly efficient hash joins instead of relatively inefficient application-level loops.
- The APPEND hint forces a direct-path load, which means that the database creates no redo and undo, thereby avoiding the waste of I/O and CPU.

Set-based processing does have some potential disadvantages:

- The techniques are unfamiliar to many database developers, so they may be more difficult.
- Because a set-based model is completely different from an iterative model, changing it requires completely rewriting the source code.

See Also: RWP #7 Set-Based Processing, RWP #8: Set-Based Parallel Processing, RWP #9: Set-Based Processing--Data Deduplication, RWP #10: Set-Based Processing--Data Transformations, and RWP #11: Set-Based Processing--Data Aggregation

Recommended Programming Practices

Use the following recommended programming practices.

Use Instrumentation Packages

Oracle Database supplies instrumentation packages whose subprograms let your application generate trace information whenever necessary. Using this trace information, you can debug your application without a debugger and identify code that performs badly.

Instrumentation provides your application with considerable functionality; therefore, it is not overhead. Overhead is something that you can remove without losing much benefit.

Some instrumentation packages that Oracle Database supplies are:

- DBMS_APPLICATION_INFO, which enables a system administrator to track the performance of your application by module.

  For more information about DBMS_APPLICATION_INFO, see *Oracle Database PL/SQL Packages and Types Reference*.
- DBMS_SESSION, which enables your application to access session information and set preferences and security levels

  For more information about DBMS_SESSION, see *Oracle Database PL/SQL Packages and Types Reference*.
- UTL_FILE, which enables your application to read and write operating system text files

  For more information about UTL_FILE, see *Oracle Database PL/SQL Packages and Types Reference*.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for a summary of PL/SQL packages that Oracle Database supplies

Statistics Gathering and Application Tracing

Database statistics provide information about the type of load on the database and the internal and external resources used by the database. To accurately diagnose performance problems with the database using ADDM, statistics must be available.

For information about statistics gathering, see *Oracle Database Get Started with Performance Tuning*.

> **Note:**
>
> If Oracle Enterprise Manager is unavailable, you can gather statistics using DBMS_MONITOR subprograms as described in *Oracle Database PL/SQL Packages and Types Reference*.

Oracle Database provides several tracing tools that can help you monitor and analyze Oracle Database applications. For details, see *Oracle Database SQL Tuning Guide*.

Use Existing Functionality

An application that uses existing functionality is easier to develop and maintain than one that does not, and it also runs faster.

When developing your application, use the existing functionality of your programming language, your operating system, Oracle Database, and the PL/SQL packages and types that Oracle Database supplies as much as possible.

Examples of existing functionality that many developers reinvent are:

- **Constraints**

  For introductory information about constraints, see "Ensuring Data Integrity in Tables."

- **SQL functions** (functions that are "built into" SQL)

  For information about SQL functions, see *Oracle Database SQL Language Reference*.

- **Sequences** (which can generate unique sequential values)

  See "Creating and Managing Sequences".

- **Auditing** (the monitoring and recording of selected user database actions)

  For introductory information about auditing, see *Oracle Database Security Guide*.

- **Replication** (the process of copying and maintaining database objects, such as tables, in multiple databases that comprise a distributed database system)

  For information about replication, see the Oracle GoldenGate documentation..

- **Message queuing** (how web-based business applications communicate with each other)

  For introductory information about Oracle Database Advanced Queuing (AQ), see *Oracle Database Advanced Queuing User's Guide*.

- **Maintaining a history of record changes**

  For introductory information about Workspace Manager, see *Oracle Database Workspace Manager Developer's Guide*.

In Example 8-4, two concurrent transactions dequeue messages stored in a table (that is, each transaction finds and locks the next unprocessed row of the table). Rather than simply invoking the DBMS_AQ.DEQUEUE procedure (described in *Oracle Database PL/SQL Packages and Types Reference*), the example creates a function-based index on the table and then uses that function in each transaction to retrieve the rows and display the messages.

The code in Example 8-4 implements a feature similar to a DBMS_AQ.DEQUEUE invocation but with fewer capabilities. The development time saved by using existing functionality (in this case, function-based indexes) can be large.

Example 8-4 Concurrent Dequeuing Transactions

Create table:

```
DROP TABLE t;
CREATE TABLE t
  ( id             NUMBER PRIMARY KEY,
    processed_flag VARCHAR2(1),
    payload        VARCHAR2(20)
  );
```

Create index on table:

```
CREATE INDEX t_idx ON
  t( DECODE( processed_flag, 'N', 'N' ) );
```

Populate table:

```
INSERT INTO t
  SELECT r,
         CASE WHEN MOD(r,2) = 0 THEN 'N' ELSE 'Y' END,
         'payload ' || r
  FROM (SELECT LEVEL r FROM DUAL CONNECT BY LEVEL <= 5);
```

Show table:

```
SELECT * FROM t;
```

Result:

```
        ID P PAYLOAD
---------- - --------------------
         1 Y payload 1
         2 N payload 2
         3 Y payload 3
         4 N payload 4
         5 Y payload 5

5 rows selected.
```

First transaction:

```
DECLARE
  l_rec t%ROWTYPE;
  CURSOR c IS
    SELECT *
```

```
    FROM t
    WHERE DECODE(processed_flag,'N','N') = 'N'
    FOR UPDATE
    SKIP LOCKED;
BEGIN
  OPEN c;

  FETCH c INTO l_rec;

  IF ( c%FOUND ) THEN
    DBMS_OUTPUT.PUT_LINE( 'Got row ' || l_rec.id || ', ' || l_rec.payload );
  END IF;

  CLOSE c;
END;
/
```

Result:

```
Got row 2, payload 2
```

Concurrent transaction:

```
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
  l_rec t%ROWTYPE;
  CURSOR c IS
    SELECT *
    FROM t
    WHERE DECODE(processed_flag,'N','N') = 'N'
    FOR UPDATE
    SKIP LOCKED;
BEGIN
  OPEN c;

  FETCH c INTO l_rec;

  IF ( c%FOUND ) THEN
    DBMS_OUTPUT.PUT_LINE( 'Got row ' || l_rec.id || ', ' || l_rec.payload );
  END IF;

  CLOSE c;
  COMMIT;
END;
/
```

Result:

```
Got row 4, payload 4
```

See Also:

- *Oracle Database New Features Guide* (with each release)
- *Oracle Database Concepts* (with each release)

Cover Database Tables with Editioning Views

If your application uses database tables, then cover each one with an editioning view so that you can use edition-based redefinition (EBR) to upgrade the database component of your application while it is in use, thereby minimizing or eliminating down time.

For information about edition-based redefinition, see *Oracle Database Development Guide*.

Recommended Security Practices

When granting privileges on the schema objects that comprise your application, use the **principle of least privilege**.

That is, users and middle tiers should be given the fewest privileges necessary to perform their actions, to reduce the danger of inadvertent or malicious unauthorized activities.

> **See Also:**
>
> "Using Bind Variables to Improve Scalability" for information about using bind variables instead of string literals, which is the most effective way to make your code invulnerable to SQL injection attacks