4

Designing Applications for Oracle Real-World Performance

When you design applications for real world performance, you should consider how code for bind variables, instrumentation, and set-based processing.

Topics:

4.1 Using Bind Variables

A bind variable placeholder in a SQL statement or PL/SQL block indicates where data must be supplied at runtime.

Suppose that you want your application to insert data into the table created with this statement:

```
CREATE TABLE test (x VARCHAR2(30), y VARCHAR2(30));
```

Because the data is not known until runtime, you must use dynamic SQL.

The following statement inserts a row into table `test`, concatenating string literals for columns x and y:

```
INSERT INTO test (x,y) VALUES ( ''' || REPLACE (x, '''', '''''') || ''',
                                ''' || REPLACE (y, '''', '''''') || ''');
```

The following statement inserts a row into table `test` using bind variables `:x` and `:y` for columns x and y:

```
INSERT INTO test (x,y) VALUES (:x, :y);
```

The statement that uses bind variable placeholders is easier to code.

Now consider a dynamic bulk load operation that inserts 1,000 rows into table `test` using each of the preceding methods.

The method that concatenates string literals uses 1,000 `INSERT` statements, each of which must be hard-parsed, qualified, checked for security, optimized, and compiled. Because each statement is hard-parsed, the number of latches greatly increases. Latches are mutual-exclusion locking mechanisms—serialization devices, which inhibit concurrency.

A method that uses bind variable placeholders uses only one `INSERT` statement. The statement is soft-parsed, qualified, checked for security, optimized, compiled, and cached in a shared pool. The compiled statement from the shared pool is used for each of the 1000 inserts. This statement caching is a very important benefit of using bind variables.

An application that uses bind variable placeholders is more scalable, supports more users, requires fewer resources, and runs faster than an application that uses string concatenation—

and it is less vulnerable to SQL injection attacks. If a SQL statement uses string concatenation, an end user can modify the statement and use the application to do something harmful.

You can use bind variable placeholders for input variables in `DELETE`, `INSERT`, `SELECT`, and `UPDATE` statements, and anywhere in a PL/SQL block that you can use an expression or literal. In PL/SQL, you can also use bind variable placeholders for output variables. Binding is used for both input and output variables in nonquery operations.

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for more information about using bind variables to protect your application from SQL injection
> - *Oracle Call Interface Programmer's Guide* for more information about using bind variable placeholders in OCI

4.2 Using Instrumentation

To use instrumentation means adding debug code throughout your application. When enabled, this code generates trace files, which contain information that helps you identify and locate problems. Trace files are especially helpful when debugging multitier applications; they help you identify the problematic tier.

> **See Also:**
>
> SQL Trace Facility (SQL_TRACE) for more information

4.3 Using Set-Based Processing

A common task in database applications in a data warehouse environment is querying or modifying a huge data set.

For example, an application might join data sets numbering in the tens of millions of rows, filter on a set of criteria, perform aggregations, and then display the result to the user. Alternatively, an application might filter out rows from one billion-row table based on specified criteria, and then insert matching rows into another table.

The problem for application developers is how to achieve high performance when processing these large data sets. Processing techniques fall into two categories: iterative, and set-based. Over years of testing, the Oracle Real-World Performance group has discovered that set-based processing techniques perform orders of magnitude better for database applications that process large data sets.

Topics:

4.3.1 Iterative Data Processing

Iterative data processing processes data row by row, using arrays, or using manual parallelism.

Topics:

4.3.1.1 About Iterative Data Processing

In this type of processing, applications use conditional logic to iterate through a set of rows.

You can write iterative applications in PL/SQL, Java, or any other procedural or object-oriented language. The technique is "iterative" because it breaks the row source into subgroups containing one or more rows, and then processes each subgroup. A single process can iterate through all subgroups, or multiple processes can iterate through the subgroups in parallel.

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. Each technique obtains the same result, but from a performance perspective, each has its benefits and drawbacks.

4.3.1.2 Iterative Data Processing: Row-By-Row

Of the iterative techniques, row-by-row processing is the most common.

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 executes 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 code uses a cursor loop to perform the following actions:

1. Fetch a single row from `ext_scan_events` to the application running in the client host, or exit the program if no more rows exist.
2. Insert the row into `stage1_scan_events`.
3. Commit the preceding insert.
4. Return to Step 1.

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

- It performs well on small data sets. Assume that `ext_scan_events` contains 10,000 records. If the application processes each row in 1 millisecond, then the total processing time is 10 seconds.
- 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. If `ext_scan_events` contains 1 billion rows, and if the application processes each row in an average of 1 miliseconds, then the total processing time is 12 days. Processing a trillion-row table requires *32 years*.
- The application executes serially, and thus cannot exploit the native parallel processing features of Oracle Database running on modern hardware. For example, the row-by-row technique cannot benefit from a multi-core computer, Oracle RAC, or Oracle Exadata Machine. For example, if the database host contains 16 CPUs and 32 cores, then 31 cores will be idle when the sole database server process reads or write each row. If multiple instances exist in an Oracle RAC deployment, then only one instance can process the data.

4.3.1.3 Iterative Data 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.

Like the row-by-row technique, array processing is serial, which means that only one database server process operates on a group of rows at one time. In a typical array implementation, the application retrieves each group of rows from the database, processes it in the middle tier, and then sends the group back to the database, which performs DML for the group of rows, and then commits.

Assume that your functional requirement is the same as in the example in Iterative Data Processing: Row-By-Row: 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, which you execute in SQL*Plus on a separate host from the database server, 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 cursor loop performs the following sequence of actions:

1. Fetch an array of rows from ext_scan_events to the application running in the client host, or exit the program when the loop counter equals 0.
2. Loop through the array of rows, and insert each row into the `stage1_scan_events` table.
3. Commit the preceding inserts.
4. Return to Step 1.

In PL/SQL, the array code differs from the row-by-row code in using a counter rather than the cursor attribute `c%notfound` to test the exit condition. The reason is that if the fetch collects the last group of rows in the table, then `c%notfound` forces the loop to exit, which is undesired behavior. When using a counter, each fetch collects the specified number of rows, and when the collection is empty, the program exits.

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. When combined, these factors can potentially reduce the total processing time by an order of magnitude
- The database is more efficient because the server process batches the inserts, and commits after every group of inserts rather than after every insert. Reducing the number of commits reduces the I/O load and lessens the probability of log sync wait events.

The disadvantages of this technique are the same as for row-by-row processing. Processing time can be unacceptable for large data sets. For a trillion-row table, reducing processing time from 32 years to 3.2 years is still unacceptable. Also, the application must run serially on a single CPU core, and thus cannot exploit the native parallelism of Oracle Database.

> **See Also:**
>
> Iterative Data Processing: Row-By-Row

4.3.1.4 Iterative Data Processing: Manual Parallelism

Manual parallelism uses the same iterative algorithm as row-by-row and array processing, but enables multiple server processes to work on the job concurrently.

In a typical implementation, the application scans the source data multiple times, and then uses the `ORA_HASH` function to divide the data among the parallel insert processes.

The `ORA_HASH` function computes a hash value for a given expression. The function accepts three arguments:

- *expr*, which is typically a column name

- *max_bucket*, which specifies the number of hash buckets
- *seed_value*, which enables multiple results from the same data (the default is 0)

For example, the following statement divides the sales table into 10 buckets of rows, numbered 0 to 9, and returns the rows from bucket 1:

```
SELECT * FROM sales WHERE ORA_HASH(cust_id, 9) = 1;
```

If an application uses `ORA_HASH` in this way, and if n hash buckets exists, then each server process operates on 1/*n* of the data.

Assume the functional requirement is the same as in the row-by-row and array examples: to read scan events from source tables, and then insert them into the `stage1_scan_events` table. The primary differences are as follows:

- The scan events are stored in a mass of flat files. The `ext_scan_events_dets` table describes these flat files. The `ext_scan_events_dets.file_seq_nbr` column stores the numerical primary key, and the `ext_file_name` column stores the file name.
- 32 server processes must run in parallel, with each server process querying a different external table. The 32 external tables are named `ext_scan_events_0` through ext_scan_events_31. However, each server process inserts into the same `stage1_scan_events` 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, which you execute in SQL*Plus on a separate host from the database server, 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;
```

This program has three iterative constructs, from outer to inner:

1. An outer `FOR LOOP` that retrieves names of flat files, and uses DDL to specify the flat file name as the location of an external table
2. A middle `LOOP` statement that fetches groups of rows from a query of the external table.
3. An innermost `FORALL` statement that iterates through each group and inserts the rows

In this sample program, you set `$thrs` to 31 in every job, and set `$thr` to a different value between 0 and 31 in every job. For example, job 1 might have `$thr` set to 0, job 2 might have `$thr` set to 1, and so on.

In the program executed by the first job, with $thr set to 0, the outer `FOR LOOP` iterates through the results of the following query:

```
select ext_file_name
from   ext_scan_events_dets
where  ora_hash(file_seq_nbr,31) = 0
```

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 example, the query result set might contain the following file names:

```
/disk1/scan_ev_101
/disk2/scan_ev_003
/disk1/scan_ev_077
...
/disk4/scan_ev_314
```

The middle `LOOP` iterates through the list of file names. For example, the first file name in the result set might be `/disk1/scan_ev_101`. For job 1 the external table is named `ext_scan_events_0`, so the first iteration of the `LOOP` changes the location of this table as follows:

```
alter table ext_scan_events_0 location(/disk1/scan_ev_101);
```

In the innermost `FORALL` statement, the `BULK COLLECT` operator retrieves rows from the `ext_scan_events_0` table into an array, inserts the rows into the `stage1_scan_events` table, and then commits the bulk insert. When the program exits the `FORALL` statement, the program proceeds to the next item in the loop, changes the file location of the external table to `/disk2/scan_ev_003`, and then queries, inserts, and commits rows as in the previous iteration. Job 1 continues processing in this way until all records contained in the flat files corresponding to hash bucket 0 have been inserted in the `stage1_scan_events` table.

While job 1 is executing, the other 31 Oracle Scheduler jobs execute in parallel. For example, job 2 sets `$thr` to 1, which defines the cursor as a query of table `ext_scan_events_1`, and so on through job 32, which sets `$thr` to 31 and defines the cursor as a query of table `ext_scan_events_31`. 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 `stage1_scan_events` 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. For example, if 32 processes are dividing the work, and if the database has sufficient CPU and memory resources and experiences no contention, then the database might perform 32 insert jobs in the time that the array technique took to perform a single job. The performance gain for a large data set is often an order of magnitude greater than serial techniques.
- When the application uses `ORA_HASH` to distribute the workload, each thread of execution can access the same amount of data. If each thread reads and writes the same amount of data, then the parallel processes can finish at the same time, which means that the database utilizes the hardware for as long as the application takes to run.

The manual parallelism technique has the following disadvantages:

- The code is relatively lengthy, complicated, and difficult to understand. The algorithm is complicated because the work of distributing the workload over many threads falls to the developer rather than the database. Effectively, the application runs serial algorithms in parallel rather than running a parallel algorithm.
- Typically, the startup costs of dividing the data have a fixed overhead. 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. This startup limitation does not apply to the competing techniques, which do not divide the data.
- If multiple threads perform the same operations on a common set of database objects, then lock and latch contention is possible. For example, if 32 different server processes are attempting to update the same set of buffers, then buffer busy waits are probable. Also, if multiple server processes are issuing `COMMIT` statements at roughly the same time, then log file sync waits are probable.
- Parallel processing consumes significant CPU resources compared to the competing iterative techniques. If the database host does not have sufficient cores available to process the threads simultaneously, then performance suffers. For example, if only 4 cores are available to 32 threads, then the probability of a thread having CPU available at a given time is 1/8.

4.3.2 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 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. It reduces the number of COMMITs.

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 executes a *single* `COMMIT` after the database has inserted all rows. In contrast, iterative applications execute 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 Oracle 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. The improvement in performance for large data sets is so profound that iterative techniques become extremely difficult to justify.
- A side-effect of the dramatic increase in processing speed is that DBAs can eliminate long-running and error-prone batch jobs, and innovate business processes in real time. For example, instead of running a 6-hour batch job every night, a business can run a 12-seconds job as needed during the day.
- 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. This means that the database, rather than the application, decides the best way to divide, retrieve, and manipulate the rows.
- In contrast to manual parallelism, parallel DML is optimized for performance because the database, rather than the application, manages the processes. Thus, it is not necessary to divide the workload manually in the client application, and hope that each process finishes at the same time.
- 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. In typical ETL workloads, the buffer cache poses a problem. Modifying data inside the buffer cache, and then writing back the data and its associated undo and redo, consumes significant resources. Because the buffer cache cannot manage blocks fast enough, and because the CPU costs of manipulating blocks into the buffer cache and back out again (usually one 8 K block at a time) are high, both the database writer and server processes must work extremely hard to keep up with the volume of buffers.

The disadvantages of set-based processing:

- The techniques are unfamiliar to many database developers, so they are more difficult. The `INSERT` example is relatively simple. However, more complicated algorithms required more complicated statements that may require multiple outer joins. Developers who are not familiar with pipelining outer joins and using `WITH` clauses and `CASE` statements may be daunted by the prospect of both writing and understanding set-based code.
- Because a set-based model is completely different from an iterative model, changing it requires completely rewriting the source code. In contrast, changing row-by-row code to array-based code is relatively trivial.

Despite the disadvantages of set-based processing, the Oracle Real-World Performance group believes that the enormous performance gains for large data sets justify the effort.

Videos:

- 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
- RWP #11: Set-Based Processing--Data Aggregation