

Multi-Threading Processes

Parin Jhaveri and Lev Moltyaner
1-Sep, 2003

In this article, we will illustrate a tuning technique that splits processes into multiple concurrent sub-processes.

This technique of multi-threading is one of the simplest, non-intrusive and yet most effective performance tuning technique. You can achieve significant performance gains without having to rewrite the entire process. However, it may not be used on all processes. It is applicable for processes that use an outer loop to process all records from a cursor in a similar fashion. Furthermore, it will only yield significant gains on servers with multiple CPUs.

The Concept

The idea of this technique is to split a single process with many iterations of a loop into multiple concurrent processes, each with fewer iterations. We will use our application to demonstrate this concept. We use Oracle Payroll to provide payroll services to the shipping industry. As a payroll service provider, we often need to upload new employees from new customers into the Oracle Payroll application using a supplied API. Our process to load this data uses an outer loop on the staging table and calls the API for every record of staging table (see in Figure A). The process is not scalable because the time increases linearly as we increase the number of rows in our staging table. In fact, it takes more than few hours to load a few thousand employees.

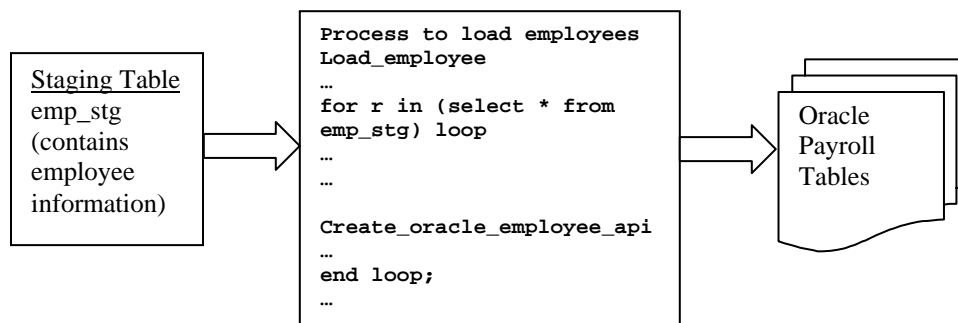


Figure A. Employee Load Process

One way to improve the performance is to apply our multi-threading technique. The idea is to change the procedure to process only a range of records instead of processing all records. This way we can run multiple instances of this process at the same time for a different range of records (see figure B).

Notice that we changed the main procedure and the cursor to accept two range parameters – start_emp_id and end_emp_id. Now every time we need to load a group of employees, we will need to derive ranges based on the number of records in the staging table and the number of processes we can run concurrently. Once we have the ranges, we can submit a separate instance of our process for each range simultaneously. For example, if we have to load 10000 employees, on a 12-cpu server, we can choose to run ten instances of load_employee process with parameters 1:1000, 1001:2000, ..., 9001:10000 simultaneously. This will improve performance almost ten folds. This approach has one problem: all concurrently running processes behave as separate processes with no single point of transaction control. Thus, it is possible for one process to fail while the others complete and commit their changes. We will now show you how to solve this problem by defining a generic algorithm that can be used to multi-thread any process.

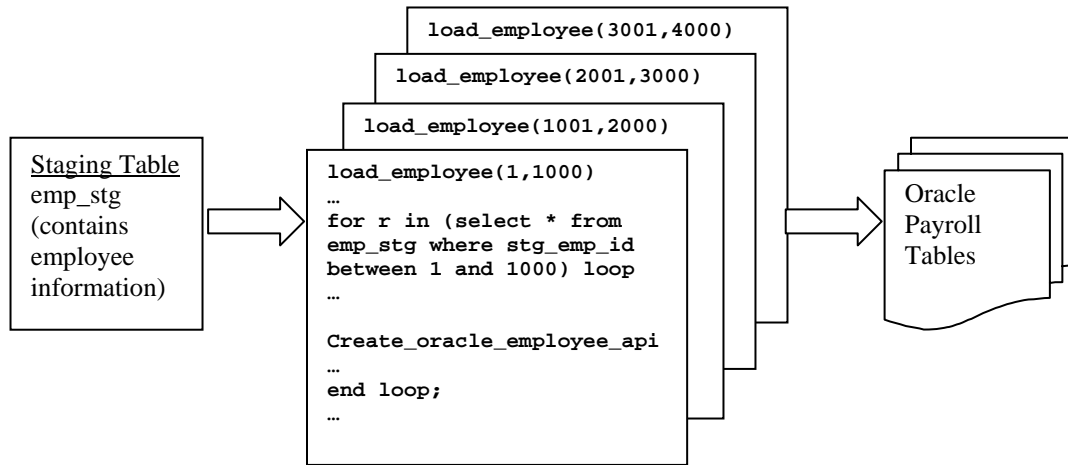


Figure B. Multiple Instances of Load Process Running Concurrently

Our Solution

Our solution has 2 packages. The first package contains 2 procedures – `main` and `load_employee`. `main` splits the employees into ranges. `load_employee` is the original procedure, which now accepts a range of employees. The second package contains 2 procedures for the generic multi-threading algorithm. We will call them `parent` and `child`. The architecture of our solution is diagrammed in Figure C.

Call 'parent' (input parameter: PL/SQL table)

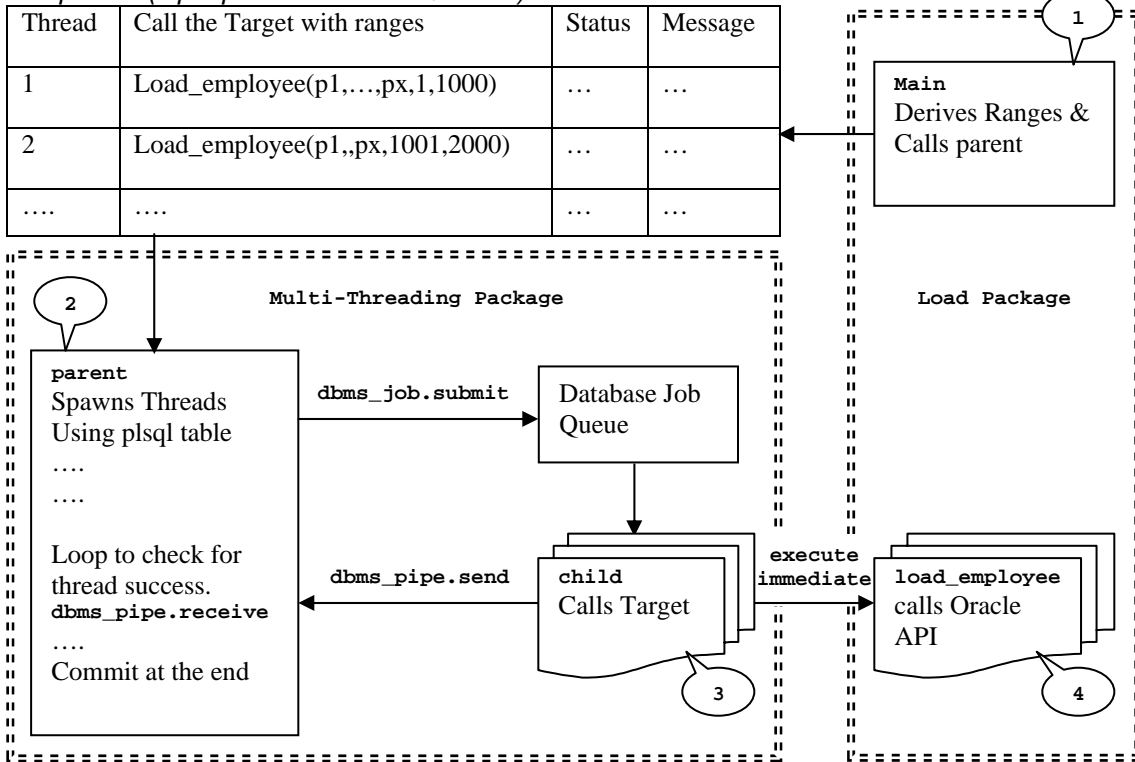


Figure C. Architecture of a Generic Multi-threading Package

Changes To Original Procedure

First step is to modify our original procedure to accept a range of employees as shown below:

```
procedure load_employee( start_emp_id number
, end_emp_id number) is
...
begin
...
for r in (select * from emp_stg where stg_emp_id
        between start_emp_id and end_emp_id) loop
    ...
    -- Create_oracle_employee_api
    ...
end loop;
..
end load_employee;
```

The next step is to add a new procedure to populate a PLSQL table with calls to `load_employee` for each range of employees. This PLSQL table contains one record per thread. Each row has a thread number, a call to `load_employee` procedure with all of its parameters, status of `child` procedure, and message which maybe returned by `load_employee` procedure. The PLSQL table definition is as follows:

```
type ps_record is record
    (thread_number number
    ,what varchar2(255)
    ,child_status varchar2(1)
    ,message_from_child varchar2(2000));

type ps_table is table of ps_record
    index by binary_integer;
```

The procedure definition is as follows:

```
procedure main(in_number_of_threads in number ) is
    cursor lc_empid is
        select emp_id from stg_emp;
...
begin
...
    select ceil(count(distinct(emp_id))
        /in_number_of_threads)
    into lv_chunksize
    from emp_stg;
...
    -- create plsql table with different ranges
    for lr_empid in lc_empid
    loop
        ...
        lv_rownum := lv_rownum + 1;
        if lr_empid.emp_id <> lv_prev_epid
        and lv_rownum >= lv_chunksize then
            lt_ps(lv_ps_index).what :=
                'begin loademp.load_employee('
                ||lv_low_wpid||','||lv_prev_wpid||');end;';
            ...
        end loop;
...
    -- call parent procedure with plsql table
    mt.parent(lt_ps);
...
end main;
```

Generic Algorithm

Finally, we will define a generic package that does the following:

- Accepts an input list of ranges,
- Submits a sub-process for each range,
- Waits for each sub-process to complete and monitor for exceptions,
- Issues one Commit or Rollback at the end.

Generic package has two procedures – the `parent` to submit threads and the `child` to execute threads. The `parent` accepts PLSQL table as an input parameter from `main`.

`parent` procedure opens a unique pipe to monitor all threads using `dbms_pipe`, submits `child` procedures for all threads using `dbms_job` and waits for all threads to finish using `dbms_pipe.receive_message` as shown here:

```
procedure parent (p_pst in out ps_table) is
...
-- open a unique pipe session to monitor all threads.
  lv_pipe_name := dbms_pipe.unique_session_name;
...
-- submit all threads using 'child' process..
  for lv_child_id in p_pst.first..p_pst.last
  loop
    ...
    dbms_job.submit
      (p_pst(lv_child_id).thread_number
      ,'mt.child('' || lv_pipe_name
      || ',' || lv_child_id
      || ',' || p_pst(lv_child_id).what || ');'
      ,sysdate
      ,null);
  end loop;
...
-- Check for completion of threads..
  while not lv_all_children_completed
  loop
    if dbms_pipe.receive_message(lv_pipe_name)
      = 0 then
      dbms_pipe.unpack_message(lv_wrapped_message);
      -- process the message
      ...
    end if;
  end loop;
...

```

`child` procedure acts as a thin wrapper to `load_employee`. When executed by the job queue, it calls `load_employee` using `execute immediate` and returns success or failure with a message using `dbms_pipe.send` to the `parent` as shown here:

```
procedure child
(p_pipe_name in varchar2
,p_child_id in number
,p_what in varchar2) is
...
begin
  execute immediate replace(p_what||',';',';',';');
  -- create message and send to parent
  dbms_pipe.pack_message(wrap_message(lr_pipe_msg));
  if dbms_pipe.send_message(p_pipe_name) <> 0 then
    -- error checking
  end if;
  exception when others then
    ...
    -- create message and send to parent
  ...
end child;

```

Notice that we are using `dbms_pipe` to handle communication between `parent` and `child` procedures, and `dbms_job` to submit child procedures for all threads. This makes our solution generic (i.e. all of the control logic resides in our separate package).

Conclusion

The generic nature of the package allows it to be used for any process. We have utilized this technique for many processes in our application. In all cases, we achieved nearly linear performance gains as we increased the number of threads. However, there are several issues you should be aware of:

- The number of threads that can be executed at a time is limited by number of `job_queue_processes` defined in `init.ora` file. For example, if `job_queue_processes` is set to 10, you can only have 10 snp (queue) jobs running at a time. This means, if you choose to run the procedure with 12 threads, only 10 threads will run concurrently and two of them will wait till two others finish.
- There are several table level parameters that may affect the performance adversely when multiple processes perform DML operations on the table simultaneously. Two most critical parameters are `FREELISTS` and `INITRANS`. If your process does a lot of `INSERTs`, we recommend that you set the number of `FREELISTS` to the maximum number of concurrent processes for the table. Also `INITRANS` needs to be increased to accommodate multiple threads updating the same data block. It is hard to predict the maximum number of threads that will update each data block. Setting this parameter unnecessarily high is not recommended because it reserves space within each data block, which may have an impact on performance. As a guideline, you may consider setting `INITRANS` to half the number of threads.
- If your original process had an exclusive table level lock, you must replace it by row level locks using a `SELECT FOR UPDATE`. Furthermore, you must ensure that each thread acquires row level locks on mutually exclusive sets of data. If this is not the case, some threads will have to wait for others to release the locks and this will degrade performance.
- The performance gains are almost linear so long as the number of threads does not exceed the number of CPUs. Thus we recommend setting the number of threads to be the same or slightly less as the number of CPUs. For example, if you have 3 CPUs, create 3 threads, however, if you have 12 CPUs, create 10-11 threads.