143 DBMS_PARALLEL_EXECUTE

The DBMS_PARALLEL_EXECUTE package enables incremental update of table data in parallel.

This chapter contains the following topics:

- Overview
- Security Model
- Constants
- Views
- Exceptions
- Examples
- Summary of DBMS_PARALLEL_EXECUTE Subprograms

See Also:

- *Oracle Database Development Guide*
- *Oracle Database Reference*

DBMS_PARALLEL_EXECUTE Overview

This package lets you incrementally update table data in parallel, in two high-level steps.

1. Group sets of rows in the table into smaller-sized chunks.
2. Run a user-specified statement on these chunks in parallel, and commit when finished processing each chunk.

This package introduces the notion of *parallel execution task*. This task groups the various steps associated with the parallel execution of a PL/SQL block, which is typically updating table data.

All of the package subroutines (except the GENERATE_TASK_NAME Function and the TASK_STATUS Procedure) perform a commit.

DBMS_PARALLEL_EXECUTE Security Model

DBMS_PARALLEL_EXECUTE is a SYS-owned package which is granted to PUBLIC.

Users who have the ADM_PARALLEL_EXECUTE_TASK role can perform administrative routines (qualified by the prefix ADM_) and access the DBA view.

Apart from the administrative routines, all the subprograms refer to tasks owned by the current user.

To execute chunks in parallel, you must have CREATE JOB system privilege.

The `CHUNK_BY_SQL`, `RUN_TASK`, and `RESUME_TASK` subprograms require a query, and are executed using `DBMS_SQL`. Invokers of the `DBMS_SQL` interface must ensure that no query contains SQL injection.

DBMS_PARALLEL_EXECUTE Constants

The `DBMS_PARALLEL_EXECUTE` package uses the constants described in these two tables.

Table 143-1 DBMS_PARALLEL_EXECUTE Constants - Chunk Status Value

Constant	Type	Value	Description
`ASSIGNED`	`NUMBER`	1	Chunk has been assigned for processing
`PROCESSED`	`NUMBER`	2	Chunk has been processed successfully
`PROCESSED_WITH_ERROR`	`NUMBER`	3	Chunk has been processed, but an error occurred during processing
`UNASSIGNED`	`NUMBER`	0	Chunk is unassigned

Table 143-2 DBMS_PARALLEL_EXECUTE Constants - Task Status Value

Constant	Type	Value	Description
`CHUNKED`	`NUMBER`	5	Table associated with the task has been chunked, but none of the chunk has been assigned for processing
`CHUNKING`	`NUMBER`	2	Table associated with the task is being chunked
`CHUNKING_FAILED`	`NUMBER`	3	Chunking failed
`CRASHED`	`NUMBER`	9	Only applicable if parallel execution is used, this occurs if a job secondary process crashes or if the database crashes during `EXECUTE`, leaving a chunk in `ASSIGNED` or `UNASSIGNED` state.
`CREATED`	`NUMBER`	1	The task has been created by the CREATE_TASK Procedure
`FINISHED`	`NUMBER`	7	All chunks processed without error
`FINISHED_WITH_ERROR`	`NUMBER`	8	All chunks processed, but with errors in some cases
`NO_CHUNKS`	`NUMBER`	4	Table associated with the task has no chunks created
`PROCESSING`	`NUMBER`	6	Part of the chunk assigned for processing, or which has been processed

Note:

Use constants instead of absolute values, because absolute values might change in future.

DBMS_PARALLEL_EXECUTE Views

The `DBMS_PARALLEL_EXECUTE` package uses the following views.

- DBA_PARALLEL_EXECUTE_CHUNKS
- DBA_PARALLEL_EXECUTE_TASKS
- USER_PARALLEL_EXECUTE_CHUNKS
- USER_PARALLEL_EXECUTE_TASKS

DBMS_PARALLEL_EXECUTE Exceptions

The following table lists the exceptions raised by `DBMS_PARALLEL_EXECUTE`.

Table 143-3 Exceptions Raised by DBMS_PARALLEL_EXECUTE

Exception	Error Code	Description
`CHUNK_NOT_FOUND`	29499	Specified chunk does not exist
`DUPLICATE_TASK_NAME`	29497	Same task name has been used by an existing task
`INVALID_STATE_FOR_CHUNK`	29492	Attempts to chunk a table that is not in `CREATED` or `CHUNKING_FAILED` state
`INVALID_STATE_FOR_REDSUME`	29495	Attempts to resume execution, but the task is not in `FINISHED_WITH_ERROR` or `CRASHED` state
`INVALID_STATE_FOR_RUN`	29494	Attempts to execute the task that is not in `CHUNKED` state
`INVALID_STATUS`	29493	Attempts to set an invalid value to the chunk status
`INVALID_TABLE`	29491	Attempts to chunk a table by rowid in cases in which the table is not a physical table, or the table is an IOT
`MISSING_ROLE`	29490	User does not have the necessary `ADM_PARALLEL_EXECUTE` role
`TASK_NOT_FOUND`	29498	Specified `task_name` does not exist

DBMS_PARALLEL_EXECUTE Examples

The following examples run on the Human Resources (HR) schema of the Oracle Database Sample Schemas. They requires that the HR schema be created with the `JOB SYSTEM` privilege.

Chunk by ROWID

This example shows the most common usage of this package. After calling the RUN_TASK Procedure, it checks for errors and reruns in the case of error.

```
DECLARE
  l_sql_stmt VARCHAR2(1000);
  l_try NUMBER;
  l_status NUMBER;
BEGIN

  -- Create the TASK
  DBMS_PARALLEL_EXECUTE.CREATE_TASK ('mytask');
  -- Chunk the table by ROWID
  DBMS_PARALLEL_EXECUTE.CREATE_CHUNKS_BY_ROWID('mytask', 'HR', 'EMPLOYEES', true, 100);

  -- Execute the DML in parallel
  l_sql_stmt := 'update EMPLOYEES e
      SET e.salary = e.salary + 10
      WHERE rowid BETWEEN :start_id AND :end_id';
  DBMS_PARALLEL_EXECUTE.RUN_TASK('mytask', l_sql_stmt, DBMS_SQL.NATIVE,
                               parallel_level => 10);

  -- If there is an error, RESUME it for at most 2 times.
  L_try := 0;
  L_status := DBMS_PARALLEL_EXECUTE.TASK_STATUS('mytask');
  WHILE(l_try < 2 and L_status != DBMS_PARALLEL_EXECUTE.FINISHED)
  LOOP
    L_try := l_try + 1;
    DBMS_PARALLEL_EXECUTE.RESUME_TASK('mytask');
    L_status := DBMS_PARALLEL_EXECUTE.TASK_STATUS('mytask');
  END LOOP;

  -- Done with processing; drop the task
  DBMS_PARALLEL_EXECUTE.DROP_TASK('mytask');

END;
/
```

Chunk by User-Provided SQL

A user can specify a chunk algorithm by using the CREATE_CHUNKS_BY_SQL Procedure. This example shows that rows with the same `manager_id` are grouped together and processed in one chunk.

```
DECLARE
  l_chunk_sql VARCHAR2(1000);
  l_sql_stmt VARCHAR2(1000);
  l_try NUMBER;
  l_status NUMBER;
BEGIN

  -- Create the TASK
  DBMS_PARALLEL_EXECUTE.CREATE_TASK ('mytask');

  -- Chunk the table by MANAGER_ID
  l_chunk_sql := 'SELECT distinct manager_id, manager_id FROM employees';
  DBMS_PARALLEL_EXECUTE.CREATE_CHUNKS_BY_SQL('mytask', l_chunk_sql, false);

  -- Execute the DML in parallel
  --   the WHERE clause contain a condition on manager_id, which is the chunk
  --   column. In this case, grouping rows is by manager_id.
  l_sql_stmt := 'update EMPLOYEES e
      SET e.salary = e.salary + 10
      WHERE manager_id between :start_id and :end_id';
  DBMS_PARALLEL_EXECUTE.RUN_TASK('mytask', l_sql_stmt, DBMS_SQL.NATIVE,
                                parallel_level => 10);

  -- If there is error, RESUME it for at most 2 times.
  L_try := 0;
  L_status := DBMS_PARALLEL_EXECUTE.TASK_STATUS('mytask');
  WHILE(l_try < 2 and L_status != DBMS_PARALLEL_EXECUTE.FINISHED)
  Loop
    L_try := l_try + 1;
    DBMS_PARALLEL_EXECUTE.RESUME_TASK('mytask');
    L_status := DBMS_PARALLEL_EXECUTE.TASK_STATUS('mytask');
  END LOOP;

  -- Done with processing; drop the task
  DBMS_PARALLEL_EXECUTE.DROP_TASK('mytask');

end;
/
```

Executing Chunks in an User-defined Framework

You can execute chunks in a self-defined framework without using the RUN_TASK Procedure. This example shows how to use GET_ROWID_CHUNK Procedure, `EXECUTE IMMEDIATE`, SET_CHUNK_STATUS Procedure to execute the chunks.

```
DECLARE
  l_sql_stmt varchar2(1000);
  l_try number;
  l_status number;
  l_chunk_id number;
  l_start_rowid rowid;
  l_end_rowid rowid;
  l_any_rows boolean;
  CURSOR c1 IS SELECT chunk_id
               FROM user_parallel_execute_chunks
               WHERE task_name = 'mytask'
                 AND STATUS IN (DBMS_PARALLEL_EXECUTE.PROCESSED_WITH_ERROR,
                                DBMS_PARALLEL_EXECUTE.ASSIGNED);
BEGIN

  -- Create the Objects, task, and chunk by ROWID
  DBMS_PARALLEL_EXECUTE.CREATE_TASK ('mytask');
  DBMS_PARALLEL_EXECUTE.CREATE_CHUNKS_BY_ROWID('mytask', 'HR', 'EMPLOYEES', true, 100);

  l_sql_stmt := 'update EMPLOYEES e
      SET e.salary = e.salary + 10
      WHERE rowid BETWEEN :start_id AND :end_id';

  -- Execute the DML in his own framework
  --
  -- Process each chunk and commit.
  -- After processing one chunk, repeat this process until
  -- all the chunks are processed.
  --
  <<main_processing>>
  LOOP
      --
      -- Get a chunk to process; if there is nothing to process, then exit the
      -- loop;
      --
      DBMS_PARALLEL_EXECUTE.GET_ROWID_CHUNK('mytask',
                                                    l_chunk_id,
                                                    l_start_rowid,
                                                    l_end_rowid,
                                                    l_any_rows);
      IF (l_any_rows = false) THEN EXIT; END IF;

      --
      -- The chunk is specified by start_id and end_id.
      -- Bind the start_id and end_id and then execute it
      --
      -- If no error occured, set the chunk status to PROCESSED.
```

```
        --
        -- Catch any exception. If an exception occured, store the error num/msg
        -- into the chunk table and then continue to process the next chunk.
        --
        BEGIN
          EXECUTE IMMEDIATE l_sql_stmt using l_start_rowid, l_end_rowid;
          DBMS_PARALLEL_EXECUTE.SET_CHUNK_STATUS('mytask',l_chunk_id,
            DBMS_PARALLEL_EXECUTE.PROCESSED);
        EXCEPTION WHEN OTHERS THEN
          DBMS_PARALLEL_EXECUTE.SET_CHUNK_STATUS('mytask', l_chunk_id,
            DBMS_PARALLEL_EXECUTE.PROCESSED_WITH_ERROR, SQLCODE, SQLERRM);
        END;

        --
        -- Finished processing one chunk; Commit here
        --
        COMMIT;
    END LOOP;
```

Summary of DBMS_PARALLEL_EXECUTE Subprograms

This table lists the `DBMS_PARALLEL_EXECUTE` subprograms and briefly describes them.

Table 143-4 DBMS_PARALLEL_EXECUTE Package Subprograms

Subprogram	Description
ADM_DROP_CHUNKS Procedure	Drops all chunks of the specified task owned by the specified owner
ADM_DROP_TASK Procedure	Drops the task of the given user and all related chunks
ADM_TASK_STATUS Function	Returns the task status
ADM_STOP_TASK Procedure	Stops the task of the given owner and related job secondary processes
CREATE_TASK Procedure	Creates a task for the current user
CREATE_CHUNKS_BY_NUMBER_COL Procedure	Chunks the table associated with the given task by the specified column
CREATE_CHUNKS_BY_ROWID Procedure	Chunks the table associated with the given task by `ROWID`
CREATE_CHUNKS_BY_SQL Procedure	Chunks the table associated with the given task by means of a user-provided `SELECT` statement
DROP_TASK Procedure	Drops the task and all related chunks
DROP_CHUNKS Procedure	Drops the task's chunks
GENERATE_TASK_NAME Function	Returns a unique name for a task
GET_NUMBER_COL_CHUNK Procedure	Picks an unassigned `NUMBER` chunk and changes it to `ASSIGNED`
GET_ROWID_CHUNK Procedure	Picks an unassigned `ROWID` chunk and changes it to `ASSIGNED`
PURGE_PROCESSED_CHUNKS Procedure	Deletes all the processed chunks whose status is `PROCESSED` or `PROCESSED_WITH_ERROR`
RESUME_TASK Procedures	Retries the given the task if the RUN_TASK Procedure finished with an error, or resumes the task if a crash occurred.

Table 143-4 (Cont.) DBMS_PARALLEL_EXECUTE Package Subprograms

Subprogram	Description
RUN_TASK Procedure	Executes the specified SQL statement on the chunks in parallel
SET_CHUNK_STATUS Procedure	Sets the status of the chunk
STOP_TASK Procedure	Stops the task and related job secondary processes
TASK_STATUS Procedure	Returns the task status

ADM_DROP_CHUNKS Procedure

This procedure drops all chunks of the specified task owned by the specified owner.

Syntax

```
DBMS_PARALLEL_EXECUTE.ADM_DROP_CHUNKS (
   task_owner      IN  VARCHAR2,
   task_name       IN  VARCHAR2);
```

Parameters

Table 143-5 ADM_DROP_CHUNKS Procedure Parameters

Parameter	Description
`task_owner`	Owner of the task
`task_name`	Name of the task

ADM_DROP_TASK Procedure

This procedure drops the task of the specified user and all related chunks.

Syntax

```
DBMS_PARALLEL_EXECUTE.ADM_DROP_TASK (
   task_owner      IN  VARCHAR2,
   task_name       IN  VARCHAR2);
```

Parameters

Table 143-6 ADM_DROP_TASK Procedure Parameters

Parameter	Description
`task_owner`	Owner of the task
`task_name`	Name of the task

ADM_TASK_STATUS Function

This function returns the task status.

Syntax

```
DBMS_PARALLEL_EXECUTE.ADM_TASK_STATUS  (
   task_owner      IN  VARCHAR2,
   task_name       IN  VARCHAR2)
 RETURN NUMBER;
```

Parameters

Table 143-7 ADM_TASK_STATUS Function Parameters

Parameter	Description
`task_owner`	Owner of the task
`task_name`	Name of the task

ADM_STOP_TASK Procedure

This procedure stops the task of the specified owner and related job secondary processes.

Syntax

```
DBMS_PARALLEL_EXECUTE.ADM_STOP_TASK (
   task_owner      IN  VARCHAR2,
   task_name       IN  VARCHAR2);
```

Parameters

Table 143-8 ADM_STOP_TASK Procedure Parameters

Parameter	Description
`task_owner`	Owner of the task
`task_name`	Name of the task

CREATE_TASK Procedure

This procedure creates a task for the current user. The pairing of `task_name` and `current_user` must be unique.

Syntax

```
DBMS_PARALLEL_EXECUTE.CREATE_TASK (
   task_name        IN   VARCHAR2,
   comment          IN   VARCHAR2 DEFAULT NULL);
```

Parameters

Table 143-9 CREATE_TASK Procedure Parameters

Parameter	Description
`task_name`	Name of the task. The task_name can be any string in which related length must be less than or equal to 128 bytes.
`comment`	Comment field. The comment must be less than 4000 bytes.

CREATE_CHUNKS_BY_NUMBER_COL Procedure

This procedure chunks the table (associated with the specified task) by the specified column. The specified column must be a `NUMBER` column. This procedure takes the `MIN` and `MAX` value of the column, and then divides the range evenly according to `chunk_size`.

The chunks are:

```
START_ID                              END_ID
---------------------------           ---------------------------
min_id_val                            min_id_val+1*chunk_size-1
min_id_val+1*chunk_size               min_id_val+2*chunk_size-1
…                                     …
min_id_val+i*chunk_size               max_id_val
```

Syntax

```
DBMS_PARALLEL_EXECUTE.CREATE_CHUNKS_BY_NUMBER_COL (
   task_name       IN  VARCHAR2,
   table_owner     IN  VARCHAR2,
   table_name      IN  VARCHAR2,
   table_column    IN  VARCHAR2,
   chunk_size      IN  NUMBER);
```

Parameters

Table 143-10 CREATE_CHUNKS_BY_NUMBER_COL Procedure Parameters

Parameter	Description
`task_name`	Name of the task
`table_owner`	Owner of the table
`table_name`	Name of the table
`table_column`	Name of the `NUMBER` column
`chunk_size`	Range of each chunk

CREATE_CHUNKS_BY_ROWID Procedure

This procedure chunks the table (associated with the specified task) by `ROWID`.

`num_row` and `num_block` are approximate guidance for the size of each chunk. The table to be chunked must be a physical table with physical `ROWID` having views and table functions. Index-organized tables are not allowed.

Syntax

```
DBMS_PARALLEL_EXECUTE.CREATE_CHUNKS_BY_ROWID (
   task_name        IN  VARCHAR2,
   table_owner      IN  VARCHAR2,
   table_name       IN  VARCHAR2,
   by_row           IN  BOOLEAN,
   chunk_size       IN  NUMBER);
```

Parameters

Table 143-11 CREATE_CHUNKS_BY_ROWID Procedure Parameters

Parameter	Description
`task_name`	Name of the task
`table_owner`	Owner of the table
`table_name`	Name of the table
`by_row`	`TRUE` if `chunk_size` refers to the number of rows, otherwise, `chunk_size` refers to the number of blocks
`chunk_size`	Approximate number of rows/blocks to process for each commit cycle

CREATE_CHUNKS_BY_SQL Procedure

This procedure chunks the table (associated with the specified task) by means of a user-provided `SELECT` statement.

The `SELECT` statement that returns the range of each chunk must have two columns: `start_id` and `end_id`. If the task is to chunk by `ROWID`, then the two columns must be of `ROWID` type. If the task is to chunk the table by `NUMBER` column, then the two columns must be of `NUMBER` type. The procedure provides the flexibility to users who want to deploy user-defined chunk algorithms.

Syntax

```
DBMS_PARALLEL_EXECUTE.CREATE_CHUNKS_BY_SQL (
   task_name        IN  VARCHAR2,
   sql_stmt         IN  CLOB,
   by_rowid         IN  BOOLEAN);
```

Parameters

Table 143-12 CREATE_CHUNKS_BY_SQL Procedure Parameters

Parameter	Description
`task_name`	Name of the task
`sql_stmt`	SQL that returns the chunk ranges
`by_rowid`	`TRUE` if the table is chunked by rowids

DROP_TASK Procedure

This procedure drops the task and all related chunks.

Syntax

```
DBMS_PARALLEL_EXECUTE.DROP_TASK (
   task_name       IN VARCHAR2);
```

Parameters

Table 143-13 DROP_TASK Procedure Parameters

Parameter	Description
`task_name`	Name of the task

DROP_CHUNKS Procedure

This procedure drops the task's chunks.

Syntax

```
DBMS_PARALLEL_EXECUTE.DROP_CHUNKS (
   task_name       IN VARCHAR2);
```

Parameters

Table 143-14 DROP_CHUNKS Procedure Parameters

Parameter	Description
`task_name`	Name of the task

GENERATE_TASK_NAME Function

This function returns a unique name for a task.

The name is of the form *prefix*N where N is a number from a sequence. If no prefix is specified, the generated name is, by default, `TASK$_1`, `TASK$_2`, `TASK$_3`, and so on. If '`SCOTT`' is specified as the prefix, the name is `SCOTT1`, `SCOTT2`, and so on.

Syntax

```
DBMS_PARALLEL_EXECUTE.GENERATE_TASK_NAME (
   prefix      IN      VARCHAR2 DEFAULT 'TASK$_')
 RETURN VARCHAR2;
```

Parameters

Table 143-15 GENERATE_TASK_NAME Function Parameters

Parameter	Description
`prefix`	The prefix to use when generating the task name

GET_NUMBER_COL_CHUNK Procedure

This procedure picks an unassigned NUMBER chunk and changes it to ASSIGNED. If there are no more chunks to assign, any_rows is set to FALSE. Otherwise, the chunk_id, start, and end_id of the chunk are returned as OUT parameters.

The chunk information in DBMS_PARALLEL_EXECUTE_CHUNKS$ is updated as follows: STATUS becomes ASSIGNED; START_TIMESTAMP records the current time; END_TIMESTAMP is cleared.

See Also:

Views

Syntax

```
DBMS_PARALLEL_EXECUTE.GET_NUMBER_COL_CHUNK (
   task_name        IN VARCHAR2,
   chunk_id         OUT NUMBER,
   start_id         OUT NUMBER,
   end_id           OUT NUMBER,
   any_rows         OUT BOOLEAN);
```

Parameters

Table 143-16 GET_NUMBER_COL_CHUNK Procedure Parameters

Parameter	Description
task_name	Name of the task
chunk_id	Chunk ID of the chunk
start_id	ID of the start row in the returned range
end_id	ID of the end row in the returned range
any_rows	Indicates if there could be any rows to process in the range

Usage Notes

If the task is chunked by ROWID, then use get_rowid_range. If the task is chunked by NUMBER column, then use get_number_col_range. If you make the wrong function call, the returning chunk_id and any_rows have valid values but start_id and end_id are NULL.

GET_ROWID_CHUNK Procedure

This procedure picks an unassigned ROWID chunk and changes it to ASSIGNED.

If there are no more chunks to assign, any_rows is set to FALSE. Otherwise, the chunk_id, start, and end_rowid of the chunk are returned as OUT parameters. The chunk info in DBMS_PARALLEL_EXECUTE_CHUNKS$ is updated as follows: STATUS becomes ASSIGNED; START_TIMESTAMP records the current time; END_TIMESTAMP is cleared.

See Also:

Views

Syntax

```
DBMS_PARALLEL_EXECUTE.GET_ROWID_CHUNK (
   task_name       IN VARCHAR2,
   chunk_id        OUT NUMBER,
   start_rowid     OUT ROWID,
   end_rowid       OUT ROWID,
   any_rows        OUT BOOLEAN);
```

Parameters

Table 143-17 GET_ROWID_CHUNK Procedure Parameters

Parameter	Description
`task_name`	Name of the task
`chunk_id`	Chunk ID of the chunk
`start_rowid`	Start rowid in the returned range
`end_rowid`	End rowid in the returned range
`any_rows`	Indicates that the range could include rows to process

Usage Notes

If the task is chunked by `ROWID`, then use `get_rowid_range`. If the task is chunked by `NUMBER` column, then use `get_number_col_range`. If you make the wrong function call, the returning `chunk_id` and `any_rows` will still have valid values but `start_id` and `end_id` are `NULL`.

PURGE_PROCESSED_CHUNKS Procedure

This procedure deletes all the processed chunks whose status is `PROCESSED` or `PROCESSED_WITH_ERROR`.

Syntax

```
DBMS_PARALLEL_EXECUTE.PURGE_PROCESSED_CHUNKS (
   task_name       IN VARCHAR2);
```

Parameters

Table 143-18 PURGE_PROCESSED_CHUNKS Procedure Parameters

Parameter	Description
`task_name`	Name of the task

RESUME_TASK Procedures

This procedure retries the specified the task if the RUN_TASK Procedure finished with an error, or resumes the task if a crash occurred.

You can only invoke this procedure if the task is in a `CRASHED` or `FINISHED_WITH_ERROR` state.

For a crashed serial execution, the state remains in `PROCESSING`. The `FORCE` option allows you to resume any task in `PROCESSING` state. However, it is your responsibility to determine that a crash has occurred.

The procedure resumes processing the chunks which have not been processed. Also, chunks which are in `PROCESSED_WITH_ERROR` or `ASSIGNED` (due to crash) state are processed because those chunks did not commit.

This procedure takes the same argument as the RUN_TASK Procedure. The overload which takes `task_name` as the only input argument re-uses the arguments provided in the previous invoking of the RUN_TASK Procedure or RESUME_TASK Procedures.

See Also:

Table 143-2

Syntax

```
DBMS_PARALLEL_EXECUTE.RESUME_TASK (
   task_name                      IN  VARCHAR2,
   sql_stmt                       IN  CLOB,
   language_flag                  IN  NUMBER,
   edition                        IN  VARCHAR2  DEFAULT NULL,
   apply_crossedition_trigger     IN  VARCHAR2  DEFAULT NULL,
   fire_apply_trigger             IN  BOOLEAN   DEFAULT TRUE,
   parallel_level                 IN  NUMBER    DEFAULT 0,
   job_class                      IN  VARCHAR2  DEFAULT 'DEFAULT_JOB_CLASS',
   force                          IN  BOOLEAN   DEFAULT FALSE);

DBMS_PARALLEL_EXECUTE.RESUME_TASK (
   task_name                      IN  VARCHAR2,
   force                          IN  BOOLEAN   DEFAULT FALSE);
```

Parameters

Table 143-19 RESUME_TASK Procedure Parameters

Parameter	Description
`task_name`	Name of the task
`sql_stmt`	SQL statement; must have `:start_id` and `:end_id` placeholders
`language_flag`	Determines how Oracle handles the SQL statement. The following options are recognized: • `V6` (or 0) specifies version 6 behavior • `NATIVE` (or 1) specifies normal behavior for the database to which the program is connected • `V7` (or 2) specifies Oracle database version 7 behavior

Table 143-19 (Cont.) RESUME_TASK Procedure Parameters

Parameter	Description
`edition`	Specifies the edition in which to run the statement. Default is the current edition.
`apply_crossedition_trigger`	Specifies the unqualified name of a forward crossedition trigger that is to be applied to the specified SQL. The name is resolved using the edition and `current_schema` setting in which the statement is to be executed. The trigger must be owned by the user who executes the statement.
`fire_apply_trigger`	Indicates whether the specified `apply_crossedition_trigger` is itself to be executed, or only to used as be a guide in selecting other triggers
`parallel_level`	Number of parallel jobs; zero if run in serial; `NULL` uses the default parallelism
`job_class`	If running in parallel, the jobs all belong to the specified job class
`force`	If `TRUE`, do not raise an error if the status is `PROCESSING`.

Examples

Suppose the chunk table contains the following chunk ranges:

```
START_ID                              END_ID
---------------------------           ---------------------------
1                                     10
11                                    20
21                                    30
```

And the specified SQL statement is:

```
UPDATE employees
      SET salary = salary + 10
      WHERE e.employee_id  BETWEEN :start_id AND :end_id
```

This procedure executes the following statements in parallel:

```
UPDATE employees
      SET salary =.salary + 10  WHERE employee_id BETWEEN 1 and 10;
      COMMIT;

UPDATE employees
      SET salary =.salary + 10  WHERE employee_id between 11 and 20;
      COMMIT;

UPDATE employees
      SET salary =.salary + 10  WHERE employee_id between 21 and 30;
      COMMIT;
```

Related Topics

- RUN_TASK Procedure
  This procedure executes the specified statement (`sql_stmt`) on the chunks in parallel.

RUN_TASK Procedure

This procedure executes the specified statement (`sql_stmt`) on the chunks in parallel.

It commits after processing each chunk.

The specified statement must have two placeholders called `start_id` and `end_id`, respectively, which represent the range of the chunk to be processed. The type of each placeholder must be `ROWID` where `ROWID`-based chunking was used, or `NUMBER` where `NUMBER`-based chunking was used.

Syntax

```
DBMS_PARALLEL_EXECUTE.RUN_TASK (
   task_name                    IN  VARCHAR2,
   sql_stmt                     IN  CLOB,
   language_flag                IN  NUMBER,
   edition                      IN  VARCHAR2  DEFAULT NULL,
   apply_crossedition_trigger   IN  VARCHAR2  DEFAULT NULL,
   fire_apply_trigger           IN  BOOLEAN   DEFAULT TRUE,
   parallel_level               IN  NUMBER    DEFAULT 0,
   job_class                    IN  VARCHAR2  DEFAULT 'DEFAULT_JOB_CLASS');
```

Parameters

Table 143-20 RUN_TASK Procedure Parameters

Parameter	Description
`task_name`	Name of the task
`sql_stmt`	SQL statement; must have `:start_id` and `:end_id` placeholders
`language_flag`	Determines how Oracle handles the SQL statement. The following options are recognized: • `V6` (or 0) specifies version 6 behavior • `NATIVE` (or 1) specifies normal behavior for the database to which the program is connected • `V7` (or 2) specifies Oracle database version 7 behavior
`edition`	Specifies the edition in which to run the statement. Default is the current edition.
`apply_crossedition_trigger`	Specifies the unqualified name of a forward crossedition trigger that is to be applied to the specified SQL. The name is resolved using the edition and `current_schema` setting in which the statement is to be executed. The trigger must be owned by the user executes the statement.
`fire_apply_trigger`	Indicates whether the specified `apply_crossedition_trigger` is itself to be executed, or only a guide to be used in selecting other triggers.
`parallel_level`	Number of parallel jobs; zero if run in serial; `NULL` uses the default parallelism.
`job_class`	If running in parallel, the jobs belong to the specified job class

Usage Notes

- The SQL statement is executed as the current user.

- Since this subprogram is subject to reexecution on error, you need to take great care in submitting a statement to `RUN_TASK` that is not idempotent.
- Chunks can be executed in parallel by `DBMS_SCHEDULER` job secondary processes. Therefore, parallel execution requires the `CREATE JOB` system privilege. The job secondary processes are created under the current user. The default number of job secondary processes is computed as the product of the Oracle parameters `cpu_count` and `parallel_threads_per_cpu`. On a Real Application Clusters installation, the number of job secondary processes is the sum of individual settings on each node in the cluster. This procedure returns only when all the chunks are processed. In parallel cases, this procedure returns only when all the secondary processes are finished.

Examples

Suppose the chunk table contains the following chunk ranges:

```
START_ID                              END_ID
---------------------------           ---------------------------
1                                     10
11                                    20
21                                    30
```

And the specified SQL statement is:

```
UPDATE employees
      SET salary = salary + 10
      WHERE e.employee_id  BETWEEN :start_id AND :end_id
```

This procedure executes the following statements in parallel:

```
UPDATE employees
      SET salary =.salary + 10  WHERE employee_id BETWEEN 1 and 10;
      COMMIT;

UPDATE employees
      SET salary =.salary + 10  WHERE employee_id between 11 and 20;
      COMMIT;

UPDATE employees
      SET salary =.salary + 10  WHERE employee_id between 21 and 30;
      COMMIT;
```

SET_CHUNK_STATUS Procedure

This procedure sets the status of the chunk.

The `START_TIMESTAMP` and `END_TIMESTAMP` of the chunk is updated according to the new status:

```
Value of the new Status               Side Effect
---------------------------           ---------------------------
UNASSIGNED                            START_TIMESTAMP and END_TIMESTAMP
                                      will be cleared
ASSIGNED                              START_TIMESTAMP will be the current time
                                      and END_TIMESTAMP will be cleared.
PROCESSED or PROCESSED_WITH_ERROR     The current time will be recorded
                                      in END_TIMESTAMP
```

See Also:

Views

Syntax

```
DBMS_PARALLEL_EXECUTE.SET_CHUNK_STATUS (
   task_name        IN VARCHAR2,
   chunk_id         OUT NUMBER,
   status           IN  NUMBER,
   err_num          IN  NUMBER   DEFAULT NULL,
   err_msg          IN  VARCHAR2 DEFAULT NULL);
```

Parameters

Table 143-21 SET_CHUNK_STATUS Procedure Parameters

Parameter	Description
`task_name`	Name of the task
`chunk_id`	Chunk_id of the chunk
`status`	Status of the chunk: `UNASSIGNED`, `ASSIGNED`, `PROCESSED` `PROCESSED_WITH_ERROR`
`err_num`	Error code returned during the processing of the chunk
`err_msg`	Error message returned during the processing of the chunk

STOP_TASK Procedure

This procedure stops the task and related secondary processes.

Syntax

```
DBMS_PARALLEL_EXECUTE.STOP_TASK (
   task_name        IN VARCHAR2);
```

Parameters

Table 143-22 STOP_TASK Procedure Parameters

Parameter	Description
`task_name`	Name of the task

TASK_STATUS Procedure

This procedure returns the task status.

Syntax

```
DBMS_PARALLEL_EXECUTE.TASK_STATUS (
   task_name        IN VARCHAR2);
```

Parameters

Table 143-23 TASK_STATUS Procedure Parameters

Parameter	Description
`task_name`	Name of the task