|  |
| --- |
| EPAM Systems, RD Dep. |
| MTN.BI.08 PL/SQL Techniques |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial status | [Elias Nema](mailto:Elias_Nema@epam.com) | 20-JAN-2014 |  |  |
|  |  |  |  |  |  |

Contents

[1. Exception Handling 3](#_Toc383578372)

[1.1. Declaring Exceptions 3](#_Toc383578373)

[1.2. Raising Exceptions 3](#_Toc383578374)

[1.3. Scope 4](#_Toc383578375)

[1.4. Propagation 4](#_Toc383578376)

[1.5. WHEN OTHERS clause 4](#_Toc383578377)

[2. PL/SQL Warnings 4](#_Toc383578378)

[2.1. Basics 5](#_Toc383578379)

[2.2. Writing WHEN OTHERS Handling Code 5](#_Toc383578380)

[3. PL/SQL and Parallel Processing 6](#_Toc383578381)

[3.1. Why Parallel Processing? 6](#_Toc383578382)

[3.2. Laws Affecting Parallel Processing 6](#_Toc383578383)

[3.3. The Rise of Big Data 7](#_Toc383578384)

[3.4. Parallel vs. Distributed Processing 7](#_Toc383578385)

[3.5. Parallel Hardware Architectures 7](#_Toc383578386)

[3.6. The MapReduce Programming Model 8](#_Toc383578387)

[3.7. Before Looking to PL/SQL 8](#_Toc383578388)

[3.8. Processes Available for Parallel Activities 8](#_Toc383578389)

[4. MapReduce + Oracle = Table functions 9](#_Toc383578390)

[4.1. Header 9](#_Toc383578391)

[4.2. Body 10](#_Toc383578392)

[4.3. Using this in a SQL Query 12](#_Toc383578393)

[5. Source Books and Articles 12](#_Toc383578394)

# Exception Handling

PL/SQL allows developers to raise and handle errors (exceptions) in a very flexible and powerful way. Each PL/SQL block can have its own exception section in which exceptions can be trapped and handled (resolved or passed on to the enclosing block). When an exception occurs (is raised) in a PL/SQL block, its execution section immediately terminates. Control is passed to the exception section. Every exception in PL/SQL has an error number and error message; some exceptions also have names.

## Declaring Exceptions

Some exceptions have been predefined by Oracle in the STANDARD package or other built-in packages, such as UTL\_FILE. See the following table for some of the most common predefined exceptions. You also can declare your own exceptions as follows:

DECLARE

exception\_name EXCEPTION;

An exception can be declared only once in a block, but nested blocks can declare an exception with the same name as an outer block. If this multiple declaration occurs, scope takes precedence over name when handling the exception. The inner block’s declaration takes precedence over a global declaration.

When you declare your own exception, you must RAISE it explicitly. All declared exceptions have an error code of 1 and the error message “User-defined exception,” unless you use the EXCEPTION\_INIT pragma.

You can associate an error number with a declared exception with the PRAGMA EXCEPTION\_INIT statement using the following syntax:

DECLARE

exception\_name EXCEPTION;

PRAGMA EXCEPTION\_INIT (exception\_name, error\_number);

where error\_number is a literal value (variable references are not allowed). This number can be an Oracle error, such as –955 (object exists), or an error in the user-definable range 20000 to –20999. For example, to execute the dynamic SQL in the variable sql\_stmt, ignoring any ORA-00955 errors, run the following:

DECLARE

ObjectExists EXCEPTION;

PRAGMA EXCEPTION\_INIT(ObjectExists,-955);

sql\_stmt VARCHAR2(100) := 'CREATE TABLE mydual AS SELECT

\* FROM dual';

BEGIN

EXECUTE IMMEDIATE sql\_stmt;

-- Ignore ORA-955 errors (object already exists)

EXCEPTION WHEN ObjectExists THEN NULL;

END;

## Raising Exceptions

An exception can be raised in three ways:

* By the PL/SQL runtime engine
* By an explicit RAISE statement in your code
* By a call to the built-in function RAISE\_APPLICATION\_ERROR

The syntax for the RAISE statement is:

RAISE exception\_name;

where exception\_name is the name of an exception that you have declared, or an exception that is declared in the STANDARD package. If you use the RAISE statement inside an exception handler, you can omit the exception name to reraise the current exception:

RAISE;

This syntax is not valid outside the exception section.

The RAISE\_APPLICATION\_ERROR built-in function has the following header:

RAISE\_APPLICATION\_ERROR (

num BINARY\_INTEGER,

msg VARCHAR2,

keeperrorstack BOOLEAN DEFAULT FALSE);

where num is the error number (an integer between –20999 and -20000), msg is the associated error message, and keep\_errorstack controls whether any previous contents of the error stack are preserved.

Starting with Oracle Database 10g Release 2, you can use the built-in function DBMS\_UTILITY.FORMAT\_ERROR\_BACKTRACE to assist in identifying where in the call stack an error occurred. Prior to Oracle Database 10g Release 2, the only way to capture the full error stack and determine the line number on which an error was raised was to let the exception go unhandled.

## Scope

The scope of an exception section is that portion of the code that is “covered” by the exception section. An exception handler will handle, or attempt to handle, only those exceptions that are raised in the executable section of the PL/SQL block. Exceptions raised in the declaration or exception sections are passed to the outer block automatically. Any line or set of PL/SQL code can be placed inside its own block and given its own exception section. This allows you to limit the propagation of an exception.

## Propagation

Exceptions raised in a PL/SQL block propagate to an outer block if they are unhandled or re-raised in the exception section. When an exception occurs, PL/SQL looks for an exception handler that checks for the exception (or is the WHEN OTHERS clause) in the current block. If a match is not found, PL/SQL propagates the exception to the enclosing block or calling program. This propagation continues until the exception is handled or propagated out of the outermost block, back to the calling program. In this case, the exception is “unhandled” and (1) stops the calling program, and (2) causes an automatic rollback of any outstanding transactions.

Once an exception is handled, it will not propagate upward. If you want to trap an exception, display a meaningful error message, and have the exception propagate upward as an error, you must re-raise the exception. The RAISE statement can reraise the current exception or raise a new exception, as shown here:

PROCEDURE delete\_dept(deptno\_in IN NUMBER)

IS

still\_have\_employees EXCEPTION;

PRAGMA EXCEPTION\_INIT(still\_have\_employees, -2292);

BEGIN

DELETE FROM dept

WHERE deptno = deptno\_in;

EXCEPTION

WHEN still\_have\_employees

THEN

DBMS\_OUTPUT.PUT\_LINE

('Please delete employees in dept first');

ROLLBACK;

RAISE; --Re-raise the current exception.

END;

## WHEN OTHERS clause

Use the WHEN OTHERS clause in the exception handler as a catch-all to trap any exceptions that are not handled by specific WHEN clauses in the exception section. If present, this clause must be the last exception handler in the exception section. Specify this clause as follows:

EXCEPTION

WHEN OTHERS

THEN

...

# PL/SQL Warnings

In Oracle 10.2, Oracle implemented PL/SQL warnings. The feature gives the ability to compile PL/SQL and get warnings if some of the code implements poor practices or conflicts with reserved words or Oracle functions. Prior to Oracle 10.2, your PL/SQL would sometimes compile—only to throw errors at runtime. The purpose of PL/SQL warnings is to warn of potential mistakes at compile time when they are easier to fix, thereby ensuring that code being applied to the database is as robust and optimal as possible.

## Basics

Before you start looking at what you gain from the PL/SQL warnings, first look at some basic aspects of the feature. By default, PL/SQL warnings are disabled. PL/SQL warnings can be enabled at the system, session, or procedure level. PL/SQL warnings are divided into three categories:

* Severe: These are warnings about code that contains a conflict with functions in SYS.STANDARD or SQL functions.
* Performance: These are warnings about code that may have a performance impact, such as the use of wrong data types in queries and the lack of use of NOCOPY compile directive when appropriate.
* Informational: These are warnings relating to code that does no harm and/or may even be removed.

## Writing WHEN OTHERS Handling Code

Include the WHEN OTHERS clause in the exception section to trap any otherwise unhandled exceptions. Because you have not explicitly handled any specific exceptions, you will very likely want to take advantage of the built-in error functions, such as SQLCODE and DBMS\_UTILITY.FORMAT\_ERROR\_STACK, to give you information about the error that has occurred.

Combined with WHEN OTHERS, SQLCODE provides a way for you to handle different, specific exceptions without having to use the EXCEPTION\_INIT pragma.

You should use WHEN OTHERS with care, because it can easily “swallow up” errors and hide them from the outer blocks and the user. Specifically, watch out for WHEN OTHER handlers that do not re-raise the current exception or raise some other exception in its place. If WHEN OTHERS does not propagate out an exception, then the outer blocks of your application will never know that an error occurred.

Oracle Database 11g offers a new warning to help you identify programs that may be ignoring or swallowing up errors:

PLW-06009: procedure "string" OTHERS handler does not end in RAISE or RAISE\_APPLICATION\_ERROR

Here is an example of using this warning:

SQL> ALTER SESSION SET plsql\_warnings = 'enable:all'

2 /

SQL> CREATE OR REPLACE PROCEDURE plw6009\_demo

2 AS

3 BEGIN

4 DBMS\_OUTPUT.put\_line ('I am here!');

5 RAISE NO\_DATA\_FOUND;

6 EXCEPTION

7 WHEN OTHERS

8 THEN

9 NULL;

10 END plw6009\_demo;

11 /

SP2-0804: Procedure created with compilation warnings

SQL> SHOW ERRORS

Errors for PROCEDURE PLW6009\_DEMO:

LINE/COL ERROR

-------- -----------------------------------------------------------------

7/9 PLW-06009: procedure "PLW6009\_DEMO" OTHERS handler does not end

in RAISE or RAISE\_APPLICATION\_ERROR

If you want to configure Oracle Database 11g warning message number 06009 (“OTHERS handler does not end in RAISE or RAISE\_APPLICATION\_ERROR”) as an error and enable all warnings in the performance category except warning number 07202 (“Parameter may benefit from use of the NOCOPY compiler hint”), execute:

ALTER SESSION SET plsql\_warnings = 'error:06009'

,'enable:performance'

,'disable:07203';

# PL/SQL and Parallel Processing

Parallel processing activities are common in nature and society; we’ve all see armies of insects and schools of fish divide and conquer amazingly large tasks. Large data processing tasks are becoming more common as the cost of collecting and storing information keeps getting less and less expensive. As companies start to accumulate vast amounts of data, they are turning to parallel processing options as ways to analyze and deal with data across many processors and machines. A parallel programming technique called MapReduce has gained popularity as a method for generically building programs that can run in parallel.

## Why Parallel Processing?

In a word: speed. All of us want things to go faster, jobs to complete more quickly, to get answers on demand. And while computers have been getting faster and faster, doubling their number of circuits, increasing their clock speeds, and adding more and more on-chip capabilities, we’re still not satisfied. As we encounter physical limitations on how fast individual processors can be, we start to look for other ways to speed up our programs.

It would be nice if computers could automatically figure out ways to divide problems into smaller parts and run procedures on those smaller parts in parallel. There has been significant research in automatic identification of parallelism, but it turns out that it’s really hard for computers to understand where opportunities for parallelism can be applied. Attempts have been made to write compliers that can take high-level programming languages and generate explicitly parallel instruction streams for CPUs, but there hasn’t been much success in this area, leaving us with the need to manually design parallel processing strategies on our own.

The good news is that when properly implemented, parallel processing techniques provide us with several benefits beyond simple speedup, including the following:

* Better system utilization in that all processors within a multi-processor system can be applied to a task.
* Removal of an all-or-nothing procedure, allowing for partial completion in case some sub-tasks fail.
* Ability to handle larger and larger data sets in a fixed amount of time, as opposed to simply speeding up a fixed data set into a smaller amount of time.

It turns out that these benefits are often worth the overhead and hard work required to design and implement parallel procedures.

## Laws Affecting Parallel Processing

Techniques (or algorithms) for solving problems with computers are studied and codified by computer scientists, often in resulting in so-called laws of computing. Several of these laws have interacted with trends in computer architecture and data growth over the past few years to influence our thinking about parallel processing.

* Moore’s Law (1965) states that the number of transistors that can fit on an integrated circuit doubles every two years and has implied that the speed and performance of computer microprocessors doubles as well. From the mid 1980’s to approximately 2004, this speedup remained fairly true as processor frequency scaling kept pace. However, power consumption started to become a problem, leading to multi-core processors which require parallelism to take full advantage of the law.
* Amdahl’s Law (1967) states that the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. Amdahl’s Law had a chilling effect on parallel processing since it states that programs could not be any faster than the speed of their sequential component. This implies a limit on the amount of speedup you can gain by parallelizing your programs, often getting programs that are only 10 to 20 times faster than their serial counterparts.
* Gustafson’s Law (1988) states that problems with very large, repetitive datasets can be efficiently parallelized, almost embarrassingly so (problems that have extremely small serial components and are easy to divide into small, independent units are formally called embarrassingly parallel).

Tension between these laws met up with one other industry trend in the mid 2000’s—the rise of so called “Big Data.”

## The Rise of Big Data

Up until roughly 2007, data growth remained fairly steady and linear. However, as technology has made it easier to capture, store, and send information, there has been an acceleration in the data growth curve. In 2011, the world will generate ten times the amount of data generated in 2006. Much of this data is actually machine generated, as the cost of sensors and data capture from web-based activities has come down. Where we once might have referred to gigabytes of data as being immense, we now look at terabytes, Petabytes, and Exabytes of data.

Awash in this sea of data, we start to see that attempting to put this data to work in a serial fashion just isn’t feasible. With only 86,400 seconds in a day, we have to achieve extreme speeds to process this kind of data if we process one bit at time. Amdahl’s Law, designed to predict the speedup of fixed problems sizes and serial portions ranging from 5 to 10%, simply doesn’t provide a framework for dealing with problem sizes this large that are growing at this new rate. Ten to twenty times speedups just aren’t enough; we need to look beyond serial speedup at ways to handle larger and larger data sets. Gustafson’s Law, which includes the idea that people would like solve ever larger problems within a relatively fixed time, coupled with the decline in processor frequency speedups in 2004, and has led to the renewed interest in parallel programming.

## Parallel vs. Distributed Processing

At this point, you may be wondering about how to achieve parallel processing in a database environment. You may have had exposure to some of the newer data processing frameworks that use many inexpensive individual computers networked together to perform data analysis tasks in a way that sounds like parallel processing. After all, our examples from nature seem like perfect matches for this kind of architecture.

These kinds of environments are usually classified as distributed computing environments, which are distinguished by a lack of access to a shared memory area and consequently the need to communicate via message passing over some kind of network. In a distributed environment, there is significant overhead due to the need for systems to coordinate their activities over this slow (compared to local processing speed) network. Also, in distributed systems, it’s assumed that each process can only see a portion of the data—none of the processes has access to the entire data set. This limits the applicability of distributed algorithms. They work well when the data is already distributed among the nodes, or if the data set is so large that the overhead tasks of splitting and distributing the data (including the inter-node communication overhead) are minuscule when compared to the overall data set size. Even so, there is significant overlap between parallel and distributed processing. A parallel system may be described as a tightly coupled distributed system, while a distributed system may be described as a loosely coupled parallel system.

The good news is that most distributed algorithms are easily translated to parallel procedures, so if you come across a distributed procedure that sounds promising, you should be able to modify it to run on a parallel architecture.

## Parallel Hardware Architectures

There are several kinds of hardware architectures that may be suitable for parallel processing. It’s also useful to understand which architectural features make a system less suitable for the requirements of a parallel procedure.

* Single computer, single processor, single core: As a base case, this kind of computer is rarely encountered, but it’s useful to note that a single core, single processor, isolated computer can only do one thing at a time. There is no way to implement a parallel procedure on such a system.
* Single computer, single processor, multi-core: If the system exposes the multiple cores as separate processors to the operating system, then it’s possible to exploit those cores in parallel. Generally, these cores share access to a high-speed memory cache on the chip and therefore can easily share coordination information.
* Single computer, multi-processor, single or multi-core: Often called symmetric multiprocessing systems (SMP), these kinds of computers have been the mainstay of open systems. These systems have processors that may not share a local high-speed memory, but all of them share access to the main computer memory through a memory bus. This shared memory access enables the processors to coordinate their activities. In some of these systems, the processors are grouped together on processor boards including a large amount of memory. If that memory is shared across processor boards, such access may be non-uniform, resulting in uneven memory access times.
* Clustered systems of SMP computers: Clustered systems usually employ some kind of high-speed dedicated system interconnection network to facilitate communication between the computer nodes. These kinds of computer clusters may or may not share access to disks. When they share access to disks, they are called shared everything architectures. When they don’t, they are called shared nothing architectures and more closely resemble distributed systems. In some variants of clusters, the high-speed interconnect actually enables remote access to the memory of each individual computer node from all other nodes. While such remote memory access is much slower than local memory access, it is much faster than using the disk subsystem to transfer information between nodes. Oracle Real Application Clustering technology using Cache Fusion has taken advantage of this capability since version 9.
* Distributed or networked systems of SMP computers: This architecture has limited shared resources as each computer is designed to be independent. While this architecture has more fault tolerance than a clustered set of systems, it relies on message passing for activity coordination and is much more suitable for distributed algorithms.

## The MapReduce Programming Model

MapReduce is a programming model for processing large data sets in parallel. The programming model is simple in that primitives are defined in such a way as to process chunks of data in an independent fashion, leading to the ability to process those chunks in parallel. The model begins by “sharding” or splitting the data into sets that are passed to map functions. A map function typically takes a set of input data and produces sets of key/value pairs which can then be handed to a set of reduce functions which perform aggregation of the mapped lists. The model has become popular due to its generic primitives, which have been used to perform a wide variety of tasks related to text scanning and filtering.

Many new database systems support MapReduce primitives as a way to perform large-scale aggregation and filtering tasks. If you’re familiar with the paradigm.

## Before Looking to PL/SQL

It’s often good to remember that the database can automatically parallelize many SQL DML and DDL operations without requiring any special PL/SQL programming. You may want to examine these options before attempting to manually parallelize your procedures to see if the automatic parallelism gives you the benefit you’re looking for. You’ll exploit some of the architectural underpinnings of the automatic parallelism in the examples, but first it might be useful for you to become familiar with what’s available. Oracle calls this architecture parallel execution and it can be applied to several different kinds of operations.

* Parallel Query: Large queries involving scans across full tables and partitions can have those scans performed in parallel, including table scans, index fast full scans, and partitioned index range scans. Join methods such as nested loops, sort merge joins, hash joins, and star transformations can also be parallelized. Aggregation and set processing methods including GROUP BY, NOT IN, UNION, UNION ALL, CUBE, and ROLLUP can be parallelized, too.
* Parallel DML: INSERT AS SELECT, UPDATE, DELETE, and MERGE operations all can be performed in parallel, including the ability to insert into multiple tables with a single DML statement.
* Parallel DDL: CREATE TABLE AS SELECT, CREATE INDEX, REBUILD INDEX, REBUILD INDEX PARTITION, and MOVE, SPLIT, or COALESCE PARTITION can all be run in parallel.

As always, you should spend the time to make sure your serial operation is as efficient as possible before looking to parallel alternatives. While parallel execution may result in faster execution time, you should be wary of the efficiencies associated with speedup and scaling up, as covered earlier in this chapter. Since most parallelization techniques are less than 100% efficient, you’ll be introducing extra resource consumption into your system, so you’ll want to make sure that the extra consumption is worth the benefits. This is especially important for systems that tend to run near 100% utilization; you may want to see if you can recoup the overhead of parallelization by running more efficient serial-type operations.

## Processes Available for Parallel Activities

Now that you’ve decided to try to implement a parallel process in PL/SQL, it’s useful to understand your execution options. Since the database enables concurrent activity to occur through sessions that usually map to individual connections and operating system processes, you’ll need to become familiar with which operating system processes can support your procedure. There are three options available.

* Parallel Execution Servers: These processes are sometimes called parallel query slaves and are automatically configured by the database to support parallel execution of SQL statements. Generally set by the parameters parallel\_min\_servers and parallel\_max\_servers to set up a pool of operating system processes for each database instance.
* Job Queue Processes: Using the DBMS\_JOB or DBMS\_SCHEDULER package, you can submit PL/SQL routines to be run by server job queue processes in an asynchronous manner. While there is no guarantee how many processes will be available at any time, the maximum number is controlled by the job\_queue\_processes parameter.
* User Session Processes: By manually creating your own operating system processes that establish connections to the database, you can create your own parallel processing support framework. You would need to control the degree of parallelism and task management outside of the database or by using database data structures.

# MapReduce + Oracle = Table functions

We are doing fairly simple here. Create a simple table that has a few records and loop over it. The reducer is than doing an aggregation. That setup is as follows:

CREATE TABLE sls (salesman VARCHAR2(30), quantity number)

/

INSERT INTO sls VALUES('Tom', 100);

INSERT INTO sls VALUES('Chu', 200);

INSERT INTO sls VALUES('Tom', 300);

INSERT INTO sls VALUES('Mike', 100);

INSERT INTO sls VALUES('Scott', 300);

INSERT INTO sls VALUES('Tom', 250);

INSERT INTO sls VALUES('Scott', 100);

commit;

/

## Header

create or replace package oracle\_map\_reduce

is

-- The types we define here is similar to the input files

-- and output files that are used in MR code and are used to

-- store data while we run the actual package.

-- The big advantage is that we do not need to write to disk for

-- intermediate results.

type sales\_t is table of sls%rowtype;

type sale\_cur\_t is ref cursor return sls%rowtype;

type sale\_rec\_t is record (name varchar2(30), total number);

type total\_sales\_t is table of sale\_rec\_t;

-- Next we define the funtions that do the work and make them known

-- to the outside world

-- Note that both mapper and reducer are tablefunctions!

-- Both mapper and reducer are pipelined and executable in parallel

-- the parallel degree is driven from the database side and is not

-- scheduled by the actual program

function mapper(inp\_cur in sys\_refcursor) return sales\_t

pipelined parallel\_enable (partition inp\_cur by any);

-- the pipelined keyword tells the caller that this function acts as

-- a row source

--

-- parallel\_enable indicates that this function can be executed in parallel

-- by the parallel query framework.

function reducer(in\_cur in sale\_cur\_t) return total\_sales\_t

pipelined parallel\_enable (partition in\_cur by hash(salesman))

-- Finally we can cluster the results so that similar rows are chunked

-- together when used (note this does not drive distribution over the

-- parallel slaves, which is done by the partition clause shown in the mapper

-- and reducers)

cluster in\_cur by (salesman);

end;

/

-- The body of the package has the mapper and the reducer code

-- The header as is shown here by itself defines the signature of

-- the package and declares types and variables to be used in the

-- package.

## Body

create or replace package oracle\_map\_reduce

is

type sales\_t is table of sls%rowtype;

type sale\_cur\_t is ref cursor return sls%rowtype;

type sale\_rec\_t is record (name varchar2(30), total number);

type total\_sales\_t is table of sale\_rec\_t;

function mapper(inp\_cur in sys\_refcursor) return sales\_t

pipelined parallel\_enable (partition inp\_cur by any);

function reducer(in\_cur in sale\_cur\_t) return total\_sales\_t

pipelined parallel\_enable (partition in\_cur by hash(salesman))

cluster in\_cur by (salesman);

end;

/

-- The upper part is the header the following part if the body

-- Note the difference in the create statement below as compared

-- to the header

create or replace package body oracle\_map\_reduce

is

function mapper(inp\_cur in sys\_refcursor) return sales\_t

pipelined parallel\_enable (partition inp\_cur by any)

is

sales\_rec sls%ROWTYPE;

-- construct a record to hold an entire row from the SLS table

begin

-- First loop over all records in the table

loop

fetch inp\_cur into sales\_rec;

exit when inp\_cur%notfound;

-- Place the found records from SLS into the variable

-- end the loop when there are no more rows to loop over

pipe row (sales\_rec);

-- by using pipe row here we are giving back rows in streaming

-- fashion as you would expect from a table

-- this in combination with pipelined in the definition allows

-- the pipelining (e.g. giving data as it comes on board) of

-- a table function

end loop;

return;

-- Return is a mandatory piece that allows the consumer of data (our reducer

-- in this case)

-- to ensure all data has been sent. After return the rowsource is exhausted

-- and no more data comes from this function.

end mapper;

-- The above mapper does in effect nothing other than streaming data

-- partitioned

-- over to the next step. In MR the stream would be written to a file and then -- redistributed to the reducers

-- The reducer below computes and emits the sales figures

function reducer(in\_cur in sale\_cur\_t) return total\_sales\_t

pipelined parallel\_enable (partition in\_cur by hash(salesman))

-- The partition by clause indicates that all instances of a particular

-- salesman must be sent to one instances of the reducer function

cluster in\_cur by (salesman)

-- The cluster by clause tells the parallel query framework to cluster

-- all instances of a particular salesman together.

IS

sale\_rec sls%ROWTYPE;

total\_sale\_rec sale\_rec\_t;

-- two containers are created, one as input the other as output

begin

total\_sale\_rec.total := 0;

total\_sale\_rec.name := NULL;

-- reset the values to initial values

loop

fetch in\_cur into sale\_rec;

exit when in\_cur%notfound;

-- some if then logic to ensure we pipe a row once all is processed

if (total\_sale\_rec.name is null) then

-- The first instance is arriving, set the salesman value to that

-- input value

-- update 0 plus the incoming value for total

total\_sale\_rec.name := sale\_rec.salesman;

total\_sale\_rec.total := total\_sale\_rec.total +

sale\_rec.quantity;

elsif ( total\_sale\_rec.name <> sale\_rec.salesman) then

-- We now switch sales man, and are done with the first

-- salesman (as rows are partitioned and clustered)

-- First pipe out the result of the previous salesman we

-- processed

-- then update the information to work on this new salesman

pipe row (total\_sale\_rec);

total\_sale\_rec.name := sale\_rec.salesman;

total\_sale\_rec.total := sale\_rec.quantity;

else

-- We get here when we work on the same salesman and just add

-- the totals, the move on to the next record

total\_sale\_rec.total := total\_sale\_rec.total +

sale\_rec.quantity;

end if;

end loop;

-- The next piece of code ensures that any remaining rows that

-- have not been piped out

-- are piped out to the consumer. If there is a single salesman,

-- he is only piped out

-- in this piece of logic as we (in the above example code) only

-- pipe out upon a change

-- of salesman

if total\_sale\_rec.total<> 0 then

pipe row (total\_sale\_rec);

end if;

return;

-- Again, we are now done and have piped all rows to our consumer

end reducer;

end;

/

## Using this in a SQL Query

It took a bit, but once you see the select you start to see how you can build a set of pre-defined programs that you can then string together to achieve a set of results.

select \*

from table(oracle\_map\_reduce.reducer(cursor(

select \* from table(oracle\_map\_reduce.mapper(cursor(

select \* from sls))) map\_result)));

All of the logic pipelines data to the next consumer and all of it is done in parallel. This makes it ideal for anything like heavy duty in-database ETL (which is what we invented it for in the first place) and anything that requires a lot of logic to be applied to records (like analytical processing).

# Source Books and Articles

1. Kyte T. Expert Oracle Database Architecture: Oracle Database 9i, 10g, and 11g Programming Techniques and Solutions, Second Edition. Apress, 2010.
2. Morton K., & Osborne K., & Sands R., & Shamsudeen R., & Still J. Pro Oracle SQL. Apress, 2013.
3. Dijcks J-P. Oracle In-Database Map-Reduce, 2010.