June 17, 2018

**Three interesting topics relevant to Freddie Mac requirements.**

1. Maximizing performance of a query performing replication of two large databases.
2. Java Error Handling with an Exception for a Math Error of division by zero.
3. Freddie Mac Business Model and Conforming Loans
4. **Maximizing performance of a query performing replication of two large databases.**

I will first approach the problem assuming that we are using Oracle 11g.

I found this information courtesy of recognized Oracle expert Steven Feuerstein:

DBMS\_PARALLEL\_EXECUTE now provides the ability to break up a large table according to a variety of criteria, from ROWID ranges to key values and user-defined methods. You can then run a SQL statement or a PL/SQL block against these different “chunks” of the table in parallel, using the database scheduler to manage the processes running in the background. Error logging, automatic retries, and commits are integrated into the processing of these chunks.

To use DBMS\_PARALLEL\_EXECUTE to run tasks in parallel, your schema will need the CREATE JOB system privilege. You can then use the following subprograms of the built-in package (these are the most commonly used of the package’s routines) to achieve your goal:

• CREATE\_TASK creates a named task to be managed by DBMS\_PARALLEL\_EXECUTE.   
• CREATE\_CHUNKS\_BY\_ROWID defines by ROWID the various chunks of the total set of rows to be modified by the SQL statement.   
• CREATE\_CHUNKS\_BY\_SQL defines, by a user-specified SQL statement, the chunking of data.   
• CREATE\_CHUNKS\_BY\_NUMBER\_COL defines, by a numeric column, the chunking of data.   
• RUN\_TASK runs the named task after chunking has been defined.   
• TASK\_STATUS obtains the status of the task.   
• STOP\_TASK stops the task.   
• RESUME\_TASK resumes the task.   
• DROP\_TASK removes the task when it has been completed.

All these subprograms and any others in DBMS\_PARALLEL\_EXECUTE—except for TASK\_STATUS—perform a commit.

Let’s look at a few examples of how to use this package. We will start with the simplest approach: chunking by ROWID. Suppose I need to apply a raise in salary (specified by a percentage) to all the employees in our company. In case my parallelized task fails for some reason, I want to be able to retry a specified number of times to complete it.

Listing 1 displays the code for implementing a parallelized apply\_raise procedure.

**Code Listing 1:** Chunking by ROWID in the apply\_raise procedure

SQL> PROCEDURE apply\_raise (

2 pct\_in IN NUMBER

3 , retries\_in IN PLS\_INTEGER DEFAULT 2

4 )

5 IS

6 c\_update\_statement CONSTANT VARCHAR2 (1000)

7 := 'UPDATE /\*+ ROWID (dda) \*/ EMPLOYEES emp

8 SET emp.salary = emp.salary \* (1.0 + pct\_in/100)

9 WHERE ROWID BETWEEN :starting\_rowid AND :ending\_rowid';

10 c\_task\_name CONSTANT VARCHAR2 (20) := 'Give Raise';

11 l\_attempts PLS\_INTEGER := 1;

12 BEGIN

13 DBMS\_PARALLEL\_EXECUTE.CREATE\_TASK (c\_task\_name);

14

15 DBMS\_PARALLEL\_EXECUTE.

16 CREATE\_CHUNKS\_BY\_ROWID (task\_name => c\_task\_name

17 , table\_owner => USER

18 , table\_name => 'EMPLOYEES'

19 , by\_row => TRUE

20 , chunk\_size => 1000

21 );

22

23 DBMS\_PARALLEL\_EXECUTE.

24 RUN\_TASK (task\_name => c\_task\_name

25 , sql\_stmt => c\_update\_statement

26 , language\_flag => DBMS\_SQL.native

27 , parallel\_level => 10

28 );

29

30 LOOP

31 EXIT WHEN DBMS\_PARALLEL\_EXECUTE.TASK\_STATUS (c\_task\_name) =

32 DBMS\_PARALLEL\_EXECUTE.FINISHED

33 OR l\_attempts > retries\_in;

34 l\_attempts := l\_attempts + 1;

35 DBMS\_PARALLEL\_EXECUTE.RESUME\_TASK (c\_task\_name);

36 END LOOP;

37

38 DBMS\_PARALLEL\_EXECUTE.DROP\_TASK (c\_task\_name);

39\* END apply\_raise;

Table 1 explains the use of DBMS\_PARALLEL\_EXECUTE subprograms in the apply\_raise procedure in Listing 1. The steps in lines 6 through 28 create the task, specify chunking, and run the task. If you are sure the task will complete or if you do not want to recover from any failures, you can simply proceed to line 38 and drop the task.

|  |  |
| --- | --- |
| Line(s) | Significance |
| 6 | The SQL statement that performs the required update. The hint explicitly chooses a table scan by rowid for the employees table. I also include two placeholders (:starting\_rowid and :ending\_rowid). This statement will be executed with DBMS\_SQL, and the placeholders will be replaced with specific ROWID values as determined by chunk size (see line 20). |
| 10 | Defining the task name in a constant to avoid using the hard-coded literal throughout the program. |
| 13 | Creating the new task. |
| 15-21 | Specifying chunking by ROWID for this task and the specified table. The by\_row parameter is set to TRUE so that the chunk size (next argument) refers to the number of rows, not the number of blocks (by\_row = FALSE). |
| 23-28 | Running this task for the specified UPDATE statement with 10 simultaneous jobs. |
| 30-33 | Starting up a simple loop. Exit the loop if the task status returns “finished” (specified through a package constant) or if the number of attempts exceeds the retry parameter. |
| 35 | Calling the RESUME\_TASK procedure to resume the task, finishing any incomplete tasks. |
| 38 | Dropping the task when it is completed or retries are exhausted. |

**Table 1:** Use of DBMS\_PARALLEL\_EXECUTE subprograms in the apply\_raise procedure

Given that any number of errors can occur during execution of DML statements, however, you may want to build into your parallel task execution the ability to check the status of the task and resume it if a failure has occurred. Lines 30 through 35 in the apply\_raise procedure address that challenge.

As you can see, DBMS\_PARALLEL\_EXECUTE offers an elegant, high-level API for specifying the parallel execution of a DML statement.

**Chunking by User SQL Statement**

As an alternative to chunking by ROWID, you can provide your own query to specify how you want your data chunked and modified in parallel. The query must contain two columns—start\_id and end\_id—both of which must be ROWIDs or numbers. Each row retrieved from this query against the “chunking table” must specify the start and end values of the chunk.

Suppose I want to execute my update against employees by ranges of department IDs. I can create a chunking table (see Listing 2) and then modify my apply\_raise procedure. To do this, first I change the WHERE clause of the update statement to specify ranges of department IDs:

c\_update\_statement

CONSTANT VARCHAR2 (1000)

:= 'UPDATE EMPLOYEES emp

SET emp.salary =

emp.salary \* (1.0 + pct\_in/100)

WHERE department\_id

BETWEEN :starting\_deptid

AND :ending\_deptid' ;

Then I define the query used to specify the chunking rows:

c\_chunk\_statement

CONSTANT VARCHAR2 (1000)

:= 'SELECT start\_id, end\_id

FROM department\_chunks';

Finally, creating chunks by this SQL statement, specifying that ROWIDs are not used:

DBMS\_PARALLEL\_EXECUTE.

CREATE\_CHUNKS\_BY\_SQL (

task\_name => c\_task\_name

, sql\_stmt => c\_chunk\_statement

, by\_rowid => FALSE

);

The remainder of the apply\_raise procedure remains unchanged.

**Code Listing 2:** Creating a chunking table

CREATE TABLE department\_chunks (start\_id INTEGER, end\_id INTEGER)

/

BEGIN

INSERT INTO department\_chunks

VALUES (1, 500);

INSERT INTO department\_chunks

VALUES (501, 1000);

INSERT INTO department\_chunks

VALUES (1001, 1500);

COMMIT;

END;

/

**Chunking by Numeric Column**

If the table I am changing contains a numeric column whose values can be used to define the ranges of rows to be updated in parallel, I can specify chunking with this procedure:

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);

In other words, I simply need to provide the name of the numeric column. Oracle Database 11*g* Release 2 then computes the MIN and MAX values of the specified column and divides the ranges evenly, as specified by the chunk size. To implement this approach, I would change the original apply\_raise procedure.

First, I change the WHERE clause of the update statement to specify ranges of department IDs:

c\_update\_statement

CONSTANT VARCHAR2 (1000)

:= 'UPDATE EMPLOYEES emp

SET emp.salary =

emp.salary \* (1.0 + pct\_in/100)

WHERE department\_id

BETWEEN :starting\_deptid

AND :ending\_deptid' ;

Next, create chunks by the department\_id column:

DBMS\_PARALLEL\_EXECUTE.

CREATE\_CHUNKS\_BY\_NUMBER\_COL (

task\_name => c\_task\_name

, table\_owner => USER

, table\_name => 'EMPLOYEES'

, table\_column => 'DEPARTMENT\_ID'

, chunk\_size => 1000

);

The remainder of the apply\_raise procedure remains unchanged.

DBMS\_PARALLEL\_EXECUTE provides many more subprograms and supports many more features than can be covered in this column. For example, rather than simply asking to run a task with RUN\_TASK, you can control chunk execution by getting a specific chunk with GET\_ROWID\_CHUNK and then executing it with EXECUTE IMMEDIATE. You can then immediately resolve any errors and decide if you want to commit the changes.

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

**I would like to next go to an alternative Oracle compatible solution offered by Laravel which utilizes PHP and has a chunk() method which splits all data into separate selects, like pagination:**

User::chunk(100, function ($users) {

foreach ($users as $user) {

$some\_value = ($user->some\_field > 0) ? 1 : 0;

// might be more logic here

$user->update(['some\_other\_field' => $some\_value]);

}

What it actually does is running a loop of selecting 100 entries, then doing updates with them, and then another 100 entries, another update and so on. database – you are working with a chunk of entries, not the whole table Which means that at no point there is a huge amount of data taken from the.});

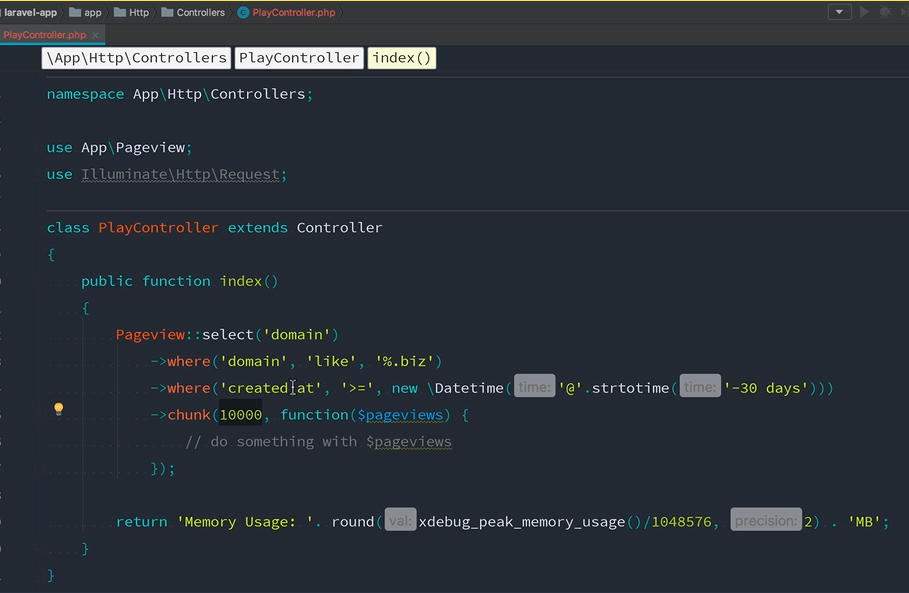
The chunk method has an inbuilt $page counter initialized to 1 and a do while loop. It accepts 2 parameters $count and $callback [see: Illuminate\Database\Query\Builder namespace]. So each time it loops through the do while loop it generates sql select query and pass the results through a callback function. The generated select sql query is like this:

SELECT \* FROM tbl WHERE $constraints LIMIT $offset, $row\_count; To cover all basis SELECT \* FROM tbl WHERE $constraints LIMIT 120,100; // will retrieve 100 rows from row 121- 220

//How $offset and $row\_count is determined in Laravel  
$offset = ($page -1) \* $count; //$count parameter passed in.  
$row\_count = $count;

So sql query generated for the first 3 iterations from the example above of …chunk(100, $callback) assuming the User model uses a ‘users’ will be:  
SELECT \* FROM users WHERE approved = 0 LIMIT 0,100 # Retrieve 100 rows from row 1- 100  
SELECT \* FROM users WHERE approved = 0 LIMIT 100,100 # Retrieve 100 rows from row 101- 200  
SELECT \* FROM users WHERE approved = 0 LIMIT 200,100 # Retrieve 100 rows from row 201- 300

According to **GitHub**, Laravel is Oracle DB compatible: Laravel-OCI8 is an Oracle Database Driver package for Laravel. Laravel-OCI8 is an extension of Illuminate/Database that uses OCI8 extension to communicate with Oracle. Laravel appears to offer the ability to fine tune memory usage as one is rewriting the code.



1. Java Error Handling with Throwing an Exception for a Math Error of division by zero.

**First a simple example with an Arithmetic Exception:**

package mainclass2;

public class MainClass2 {

public static void main(String[] args) {

int d, a;

try {

d = 0;

a = 42 / d;

System.out.println("This will not be printed.");

} catch (ArithmeticException e) {

System.out.println("Division by zero.");

}

System.out.println("After catch statement.");

}

}

run:

Division by zero.

After catch statement.

BUILD SUCCESSFUL (total time: 0 seconds)

**I looked into this problem and found this interesting example run twice below:**

package exceptiontester6;

public class ExceptionTester6 {

private static int quotient(int numerator, int denominator) throws ArithmeticException {

return(numerator / denominator);

}

public static void main(String args[]) {

int number1=0, number2=0, result=0;

try {

number1 = 8;

number2 = 2;

result = quotient(number1,number2);

System.out.print(number2 + " goes into " + number1);

System.out.println(" this many times: " + result);

}

catch (Exception e) {

System.out.println(e.toString());

System.out.println("An Exception occured");

System.exit(-1);

}

}

}

run:

2 goes into 8 this many times: 4

BUILD SUCCESSFUL (total time: 0 seconds)

package exceptiontester6;

public class ExceptionTester6 {

private static int quotient(int numerator, int denominator) throws ArithmeticException {

return(numerator / denominator);

}

public static void main(String args[]) {

int number1=0, number2=0, result=0;

try {

number1 = 8;

number2 = 0;

result = quotient(number1,number2);

System.out.print(number2 + " goes into " + number1);

System.out.println(" this many times: " + result);

}

catch (Exception e) {

System.out.println(e.toString());

System.out.println("An Exception occured");

System.exit(-1);

}

}

}

run:

java.lang.ArithmeticException: / by zero

An Exception occured

C:\Users\Hal14\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: -1

BUILD FAILED (total time: 0 seconds) (\*Compiled successfully in NetBeans before running.)

**Three main types of errors: Syntax, Runtime, and Logic Errors of which a subset is Syntax/Runtime**

**Java Exceptions and Errors**

In Java, it is possible to define two catergories of Exceptions and Errors.

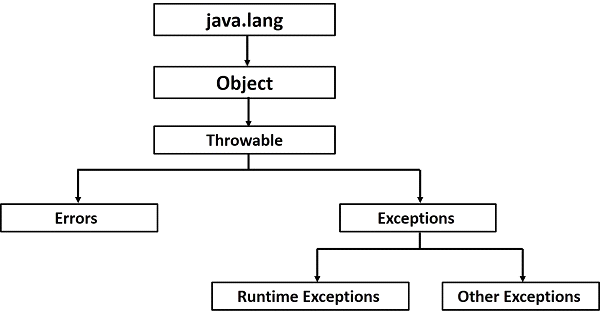
JVM Exceptions − These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples: NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException.

Programmatic Exceptions − These exceptions are thrown explicitly by the application or the API programmers. Examples: IllegalArgumentException, IllegalStateException.

**Checked exceptions − A checked exception is an exception that occurs at the compile time**, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

**Unchecked exceptions − An unchecked exception is an exception that occurs at the time of execution. These are also called as Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

Errors − These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.



Catching Exceptions

A method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception.

If a method does not handle a checked exception, the method must declare it using the throws keyword. The throws keyword appears at the end of a method's signature.

Throws is used to postpone the handling of a checked exception and throw is used to invoke an exception explicitly.

Finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

1. **Freddie Mac Business Model:**

While I did answer this question I could have specifically mentioned Capital Markets which was also an important part of Prosperity Home Mortgage where I did a prior project working with the controller. Rather than buying loans, of course, they needed to sell them after originating them.

<https://www.fhfa.gov/SupervisionRegulation/FannieMaeandFreddieMac/Pages/About-Fannie-Mae---Freddie-Mac.aspx>

Fannie Mae and Freddie Mac were created by Congress. They perform an important role in the nation’s housing finance system – to provide liquidity, stability and affordability to the mortgage market. They provide liquidity (ready access to funds on reasonable terms) to the thousands of banks, savings and loans, and mortgage companies that make loans to finance housing.

Fannie Mae and Freddie Mac buy mortgages from lenders and either hold these mortgages in their portfolios or package the loans into mortgage-backed securities (MBS) that may be sold. Lenders use the cash raised by selling mortgages to the Enterprises to engage in further lending. The Enterprises’ purchases help ensure that individuals and families that buy homes and investors that purchase apartment buildings and other multifamily dwellings have a continuous, stable supply of mortgage money.

By packaging mortgages into MBS and guaranteeing the timely payment of principal and interest on the underlying mortgages, Fannie Mae and Freddie Mac attract to the secondary mortgage market investors who might not otherwise invest in mortgages, thereby expanding the pool of funds available for housing. That makes the secondary mortgage market more liquid and helps lower the interest rates paid by homeowners and other mortgage borrowers.

Fannie Mae and Freddie Mac also can help stabilize mortgage markets and protect housing during extraordinary periods when stress or turmoil in the broader financial system threaten the economy. The Enterprises’ support for mortgage lending that finances affordable housing reduces the cost of such borrowing.

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

Regarding a **Conforming Loan**: From Investopedia

DEFINITION of 'Conforming Loan'

A conforming loan is a mortgage that is equal to or less than the dollar amount established by the conforming loan limit set by The Federal Housing Finance Agency (FHFA) and meets the funding criteria of Freddie Mac and Fannie Mae.

For borrowers with excellent credit, conforming loans are advantageous due to the low interest rates affixed to the loan BREAKING DOWN 'Conforming Loan'

A conforming loan is a mortgage that is eligible for purchase by the Federal National Mortgage Association (FNMA or Fannie Mae) and Federal Home Loan Mortgage Corporation (FHLMC or Freddie Mac), government-sponsored entities that drive the market for home loans. The quasi-governmental agencies created standardized rules and guidelines that mortgages must conform to in order to be a conforming loan. The term "conforming" is most often used when speaking specifically about the mortgage amount which must fall under a certain limit, known as the conforming loan limit, set by the Federal Housing Finance Agency (FHFA). For 2018, this limit is $453,100, an increase from $424,100 in 2017. In high-cost markets the limit is higher. The new ceiling loan limit for one-unit properties in most high-cost areas, such as San Francisco and New York City, is $679,650 — or 150 percent of $453,100. The Housing and Economic Recovery Act (HERA) requires that the baseline conforming loan limit be adjusted each year for Fannie Mae and Freddie Mac to reflect the change in the average U.S. home price.

Other than the size of the loan, other guidelines conforming loans adhere to include borrower's loan-to-value ratio (i.e. the size of down payment), debt-to-income ratio, credit score and history, documentation requirements, etc. For example, a conforming loan through Fannie or Freddie can have a down payment as low as 3 percent and the borrower must be a first-time homebuyer. In addition, private mortgage insurance (PMI) of about 1.05 percent per year for 30-year loans up to $453,100 is required on the loan. Part or all of the cost of the insurance is tax-deductible if the borrower’s household adjusted gross income (AGI) is no more than $109,000.

The FHFA, which sets the conforming loan limit on an annual basis, has regulatory oversight to ensure that Fannie Mae and Freddie Mac fulfill their charters and missions of promoting homeownership for lower income and middle class Americans. FHFA uses the October to October percentage increase/decrease in average housing prices in the Monthly Interest Rate Survey (MIRS) to adjust the conforming loan limits for the subsequent year. To conduct this survey, FHFA asks a sample of mortgage lenders to report the terms and conditions on all single-family, fully amortized, purchase-money, nonfarm loans that they close during the last five business days of the month. The survey provides monthly information on interest rates, loan terms, and house prices by property type, by loan type (fixed- or adjustable-rate), and by lender type, as well as information on 15-year and 30-year fixed-rate loans.

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