An Oracle White Paper
January 2010

Integrating Hadoop Data with Oracle Parallel Processing

ORACLE

Introduction

Various industry verticals are seeing vast amounts of data that is stored on file systems. These vast amounts of data are typically data that contains a lot of irrelevant detail and some gems useful for further analysis or enriching other data sources. Despite storing this data outside of the database some customers do want to integrate this data with data stored in the database. The goal of such integration is to extract information that is of value to the business users.

This paper describes in detail how to access data stored in a Hadoop cluster from within an Oracle database. Note that we picked Hadoop and HDFS as an example. These strategies apply to other distributed storage mechanisms. The paper describes various access methods and shows a concrete example of an implementation of such an access method.

Access Methods for External Hadoop Data

The simplest way to access external files or external data on a file system from within an Oracle database is through an external table. See here for an introduction to External tables.

External tables present data stored in a file system in a table format and can be used in SQL queries transparently. External tables could thus potentially be used to access data stored in HDFS (the Hadoop File System) from inside the Oracle database. Unfortunately HDFS files are not directly accessible through the normal operating system calls that the external table driver relies on. The FUSE (File system in Userspace) project provides a workaround in this case. There are a number of FUSE drivers that allow users to mount a HDFS store and treat it like a normal file system. By using one of these drivers and mounting HDFS on the database instance (on every instance if this was a RAC database), HDFS files can be easily accessed using the External Table infrastructure.

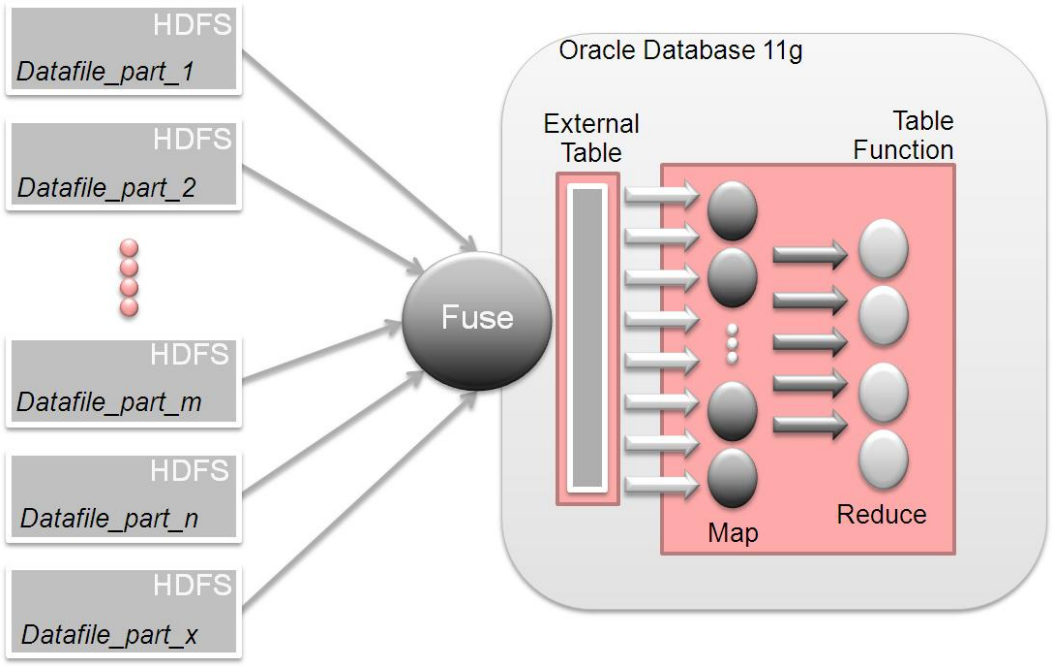

Figure 1. Accessing via External Tables with in-database MapReduce

In Figure 1we are utilizing Oracle Database 11g to implement in-database mapreduce as described in this article. In general, the parallel execution framework in Oracle Database 11g is sufficient to run most of the desired operations in parallel directly from the external table.

The external table approach may not be suitable in some cases (say if FUSE is unavailable). Oracle Table Functions provide an alternate way to fetch data from Hadoop. Our attached example outlines one way of doing this. At a high level we implement a table function that uses

the DBMS_SCHEDULER framework to asynchronously launch an external shell script that submits a Hadoop Map-Reduce job. The table function and the mapper communicate using Oracle's Advanced Queuing feature. The Hadoop mapper en-queue's data into a common queue while the table function de-queues data from it. Since this table function can be run in parallel additional logic is used to ensure that only one of the slaves submits the External Job.

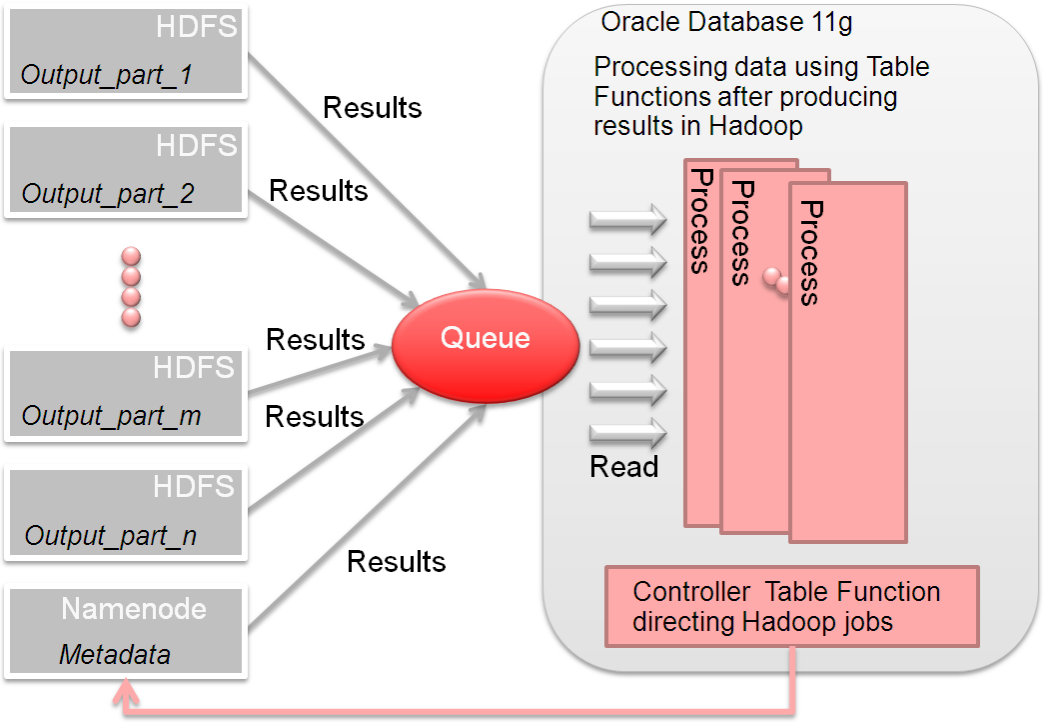

Figure 2. Leveraging Table Functions for parallel processing

The queue gives us load balancing since the table function could run in parallel while the Hadoop streaming job will also run in parallel with a different degree of parallelism and outside the control of Oracle's Query Coordinator.

An Example leveraging Table Functions

As an example we translated the architecture shown in Figure 2 in a real example. Note that our example only shows a template implementation of using a Table Function to access data stored in Hadoop. Other, possibly better, implementations are clearly possible.

The following diagrams are a technically more accurate and more detailed representation of the original schematic in Figure 2 explaining where and how we use the pieces of actual code that follow:

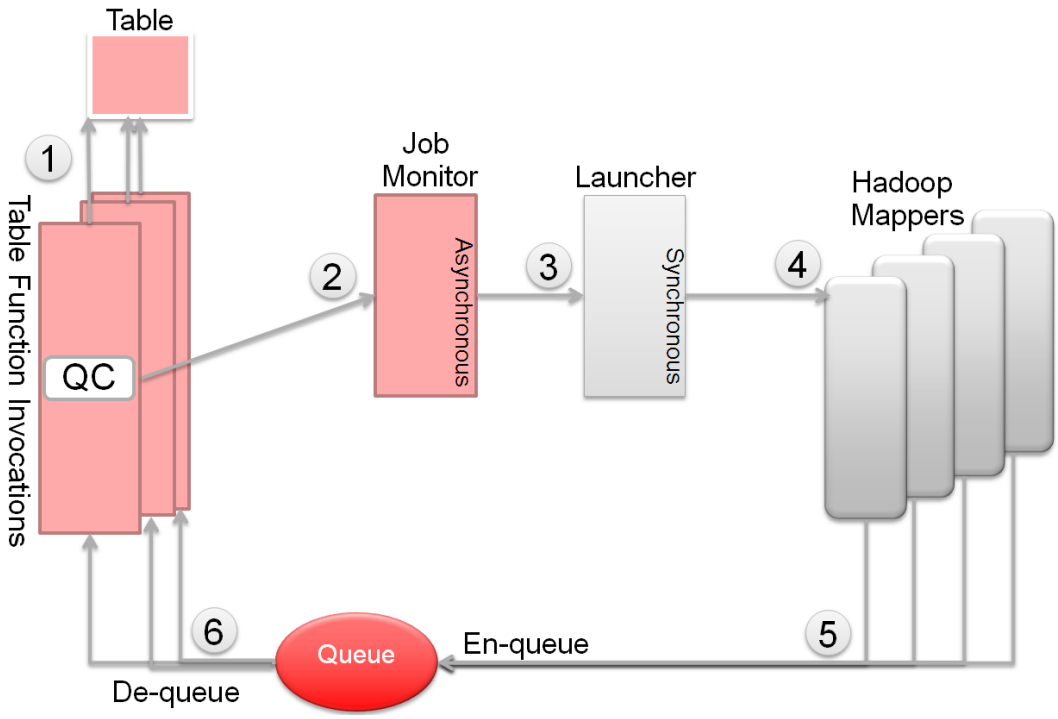

Figure 3. Starting the Mapper jobs and retrieving data

In step 1 we figure out who gets to be the query coordinator. For this we use a simple mechanism that writes records with the same key value into a table. The first insert wins and will function as the query coordinator (QC) for the process. Note that the QC table function invocation does play a processing role as well.

In step 2 this table function invocation (QC) starts an asynchronous job using dbms_scheduler – the Job Controller in Figure 3 – that than runs the synchronous bash script on the Hadoop cluster. This bash script, called the launcher in Figure 3 starts the mapper processes (step 3) on the Hadoop cluster.

The mapper processes process data and write to a queue in step 5. In the current example we chose a queue as it is available cluster wide. For now we simply chose to write any output directly into the queue. You can achieve better performance by either batching up the outputs and then moving them into the queue. Obviously you can choose various other delivery mechanisms, including pipes and relational tables.

Step 6 is then the de-queuing process which is done by parallel invocations of the table function running in the database. As these parallel invocations process data it gets served up to the query that is requesting the data. The table function leverages both the Oracle data and the data from the queue and thereby integrates data from both sources in a single result set for the end user(s).

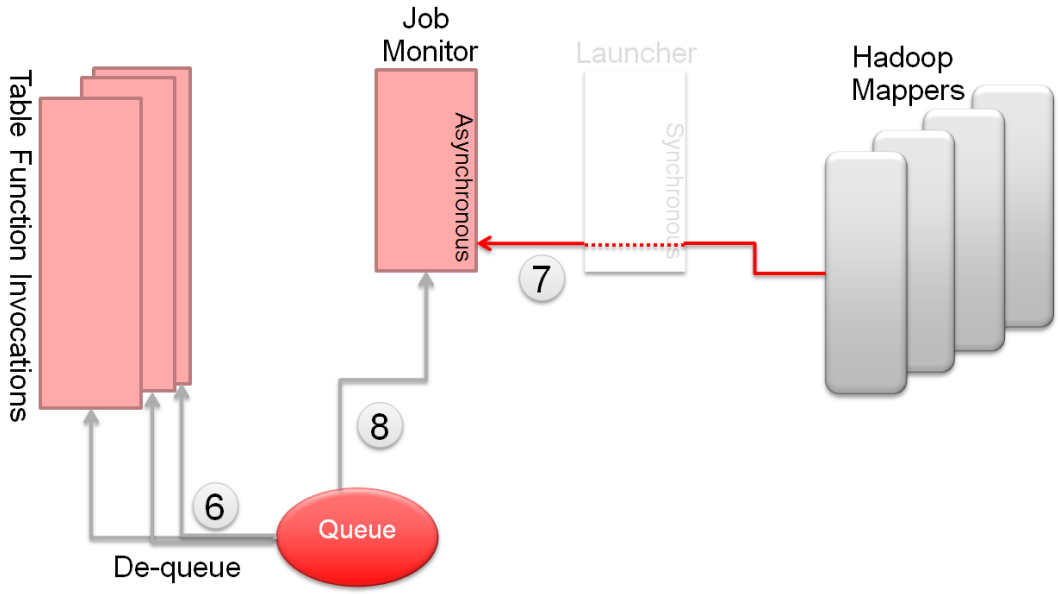

Figure 4. Monitoring the process

After the Hadoop side processes (the mappers) are kicked off, the job monitor process keeps an eye on the launcher script. Once the mappers have finished processing data on the Hadoop cluster, the bash script finishes as is shown in Figure 4.

The job monitor monitors a database scheduler queue and will notice that the shell script has finished (step 7). The job monitor checks the data queue for remaining data elements, step 8. As long as data is present in the queue the table function invocations keep on processing that data (step 6).

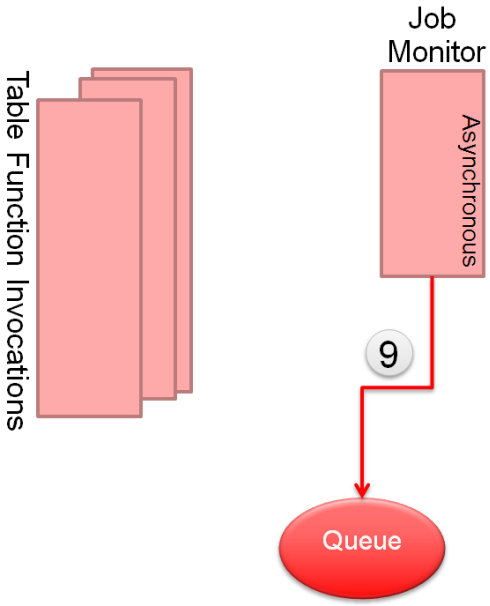

Figure 5. Closing down the processing

When the queue is fully de-queued by the parallel invocations of the table function, the job monitor terminates the queue (step 9 in Figure 5) ensuring the table function invocations in Oracle stop. At that point all the data has been delivered to the requesting query.

Example Code

The solution implemented in Figure 3 through Figure 5 uses the following code. All code is tested on Oracle Database 11g and a 5 node Hadoop cluster. As with most whitepaper text, copy these scripts into a text editor and ensure the formatting is correct.

Table Functions for Processing Data

This script contains certain set up components. For example the initial part of the script shows the creation of the arbitrator table in step 1 of Figure 3. In the example we use the always popular OE schema.

```
connect oe/oe

-- Table to use as locking mechanisim for the hdfs reader as

-- leveraged in Figure 3 step 1

DROP TABLE run_hdfs_read;


CREATE TABLE run_hdfs_read(pk_id number, status varchar2(100),
primary key (pk_id));


-- Object type used for AQ that receives the data

CREATE OR REPLACE TYPE hadoop_row_obj AS OBJECT (

a number,

b number);

/


connect / as sysdba

-- system job to launch external script
-- this job is used to eventually run the bash script
-- described in Figure 3 step 3

CREATE OR REPLACE PROCEDURE launch_hadoop_job_async(in_directory
IN VARCHAR2, id number) IS

cnt number;

BEGIN

  begin DBMS_SCHEDULER.DROP_JOB ('ExtScript'||id, TRUE);
```

```
  exception when others then null; end;

  -- Run a script
  DBMS_SCHEDULER.CREATE_JOB (
   job_name   => 'ExtScript' || id,
   job_type   => 'EXECUTABLE',
   job_action => '/bin/bash',
   number_of_arguments => 1
  );
  DBMS_SCHEDULER.SET_JOB_ARGUMENT_VALUE ('ExtScript' || id, 1,
in_directory);
  DBMS_SCHEDULER.ENABLE('ExtScript' || id);

  -- Wait till the job is done. This ensures the hadoop job is
completed
  loop
    select count(*) into cnt
    from DBA_SCHEDULER_JOBS where job_name = 'EXTSCRIPT'||id;
    dbms_output.put_line('Scheduler Count is '||cnt);
    if (cnt = 0) then
      exit;
    else
      dbms_lock.sleep(5);
    end if;
  end loop;

  -- Wait till the queue is empty and then drop it
  -- as shown in Figure 5
  -- The TF will get an exception and it will finish quietly
  loop
```

```
    select sum(c) into cnt
    from
    (
      select enqueued_msgs - dequeued_msgs c
      from gv$persistent_queues where queue_name =
'HADOOP_MR_QUEUE'
      union all
      select num_msgs+spill_msgs c
      from gv$buffered_queues where queue_name =
'HADOOP_MR_QUEUE'
      union all
      select 0 c from dual
    );
    if (cnt = 0) then
      -- Queue is done. stop it.
      DBMS_AQADM.STOP_QUEUE ('HADOOP_MR_QUEUE');
      DBMS_AQADM.DROP_QUEUE ('HADOOP_MR_QUEUE');
      return;
    else
      -- Wait for a while
      dbms_lock.sleep(5);
    end if;
  end loop;

END;
/

-- Grants needed to make hadoop reader package work
grant execute on launch_hadoop_job_async to oe;
```

```
grant select on v_$session to oe;
grant select on v_$instance to oe;
grant select on v_$px_process to oe;
grant execute on dbms_aqadm to oe;
grant execute on dbms_aq to oe;

connect oe/oe
-- Simple reader package to read a file containing two numbers
CREATE OR REPLACE PACKAGE hdfs_reader IS

-- Return type of pl/sql table function
TYPE return_rows_t IS TABLE OF hadoop_row_obj;

-- Checks if current invocation is serial
FUNCTION is_serial RETURN BOOLEAN;

-- Function to actually launch a Hadoop job
FUNCTION launch_hadoop_job(in_directory IN VARCHAR2, id in out
number) RETURN BOOLEAN;

-- Tf to read from Hadoop
-- This is the main processing code reading from the queue in
-- Figure 3 step 6. It also contains the code to insert into
-- the table in Figure 3 step 1
FUNCTION read_from_hdfs_file(pcur IN SYS_REFCURSOR, in_directory
IN VARCHAR2) RETURN return_rows_t
         PIPELINED PARALLEL_ENABLE(PARTITION pcur BY ANY);
END;
/
```

```
CREATE OR REPLACE PACKAGE BODY hdfs_reader IS
-- Checks if current process is a px_process
FUNCTION is_serial RETURN BOOLEAN IS
c NUMBER;
BEGIN
  SELECT COUNT (*) into c FROM v$px_process WHERE sid =
SYS_CONTEXT('USERENV','SESSIONID');
  IF c <> 0 THEN
    RETURN false;
  ELSE
    RETURN true;
  END IF;
exception when others then
  RAISE;
END;

FUNCTION launch_hadoop_job(in_directory IN VARCHAR2, id IN OUT
NUMBER) RETURN BOOLEAN IS
PRAGMA AUTONOMOUS_TRANSACTION;
instance_id NUMBER;
jname varchar2(4000);
BEGIN
  if is_serial then
    -- Get id by mixing instance # and session id
    id := SYS_CONTEXT('USERENV', 'SESSIONID');
    SELECT instance_number INTO instance_id FROM v$instance;
    id := instance_id * 100000 + id;
  else
    -- Get id of the QC
```

```
    SELECT ownerid into id from v$session where sid =
SYS_CONTEXT('USERENV', 'SESSIONID');
  end if;

  -- Create a row to 'lock' it so only one person does the job
  -- schedule. Everyone else will get an exception
  -- This is in Figure 3 step 1
  INSERT INTO run_hdfs_read VALUES(id, 'RUNNING');
  jname := 'Launch_hadoop_job_async';
  -- Launch a job to start the hadoop job
  DBMS_SCHEDULER.CREATE_JOB (
   job_name   => jname,
   job_type   => 'STORED_PROCEDURE',
   job_action => 'sys.launch_hadoop_job_async',
   number_of_arguments => 2
  );
  DBMS_SCHEDULER.SET_JOB_ARGUMENT_VALUE (jname, 1, in_directory);
  DBMS_SCHEDULER.SET_JOB_ARGUMENT_VALUE (jname, 2, CAST (id AS
VARCHAR2));
  DBMS_SCHEDULER.ENABLE('Launch_hadoop_job_async');
 COMMIT;
 RETURN true;
EXCEPTION
  -- one of my siblings launched the job. Get out quitely
  WHEN dup_val_on_index THEN
        dbms_output.put_line('dup value exception');
        RETURN false;
  WHEN OTHERs THEN
  RAISE;
END;
```

```
FUNCTION read_from_hdfs_file(pcur IN SYS_REFCURSOR, in_directory
IN VARCHAR2) RETURN return_rows_t

PIPELINED PARALLEL_ENABLE(PARTITION pcur BY ANY)

IS

PRAGMA AUTONOMOUS_TRANSACTION;

cleanup BOOLEAN;

payload hadoop_row_obj;

id      NUMBER;

dopt    dbms_aq.dequeue_options_t;

mprop   dbms_aq.message_properties_t;

msgid   raw(100);

BEGIN

-- Launch a job to kick off the hadoop job

  cleanup := launch_hadoop_job(in_directory, id);

  dopt.visibility    := DBMS_AQ.IMMEDIATE;

  dopt.delivery_mode := DBMS_AQ.BUFFERED;

  loop

    payload := NULL;

-- Get next row

    DBMS_AQ.DEQUEUE('HADOOP_MR_QUEUE', dopt, mprop, payload,
msgid);

    commit;

    pipe row(payload);

  end loop;

exception when others then

  if cleanup then

    delete run_hdfs_read where pk_id = id;

  commit;

  end if;
```

```
END;

END;

/
```

The Bash Script

This short script is the outside-of-the-database controller as shown in Figure 3 steps 3 and 4. This is a synchronous step remaining on the system for as long as the Hadoop mappers are running.

```
#!/bin/bash

cd –HADOOP_HOME-

A="/net/scratch/java/jdk1.6.0_16/bin/java -classpath
/home/hadoop:/home/hadoop/ojdbc6.jar StreamingEq"

bin/hadoop fs -rmr output

bin/hadoop jar ./contrib/streaming/hadoop-0.20.0-streaming.jar -
input input/nolist.txt -output output -mapper "$A" -jobconf
mapred.reduce.tasks=0
```

The Java Mapper Script

For this example we wrote a small and simple mapper process to be executed on our Hadoop cluster. Undoubtedly more comprehensive mappers exist. This one converts a string into two numbers and provides those to the queue in a row-by-row fashion.

```
// Simplified mapper example for Hadoop cluster

import java.sql.*;

//import oracle.jdbc.*;

//import oracle.sql.*;

import oracle.jdbc.pool.*;

//import java.util.Arrays;

//import oracle.sql.ARRAY;

//import oracle.sql.ArrayDescriptor;
```

```
    public class StreamingEq {
        public static void main (String args[]) throws Exception
{
             // connect through driver
             OracleDataSource ods = new OracleDataSource();
             Connection conn = null;
             Statement stmt = null;
             ResultSet rset = null;
             java.io.BufferedReader stdin;
             String line;
             int countArr = 0, i;
             oracle.sql.ARRAY sqlArray;
             oracle.sql.ArrayDescriptor arrDesc;

             try {
             ods.setURL("jdbc:oracle:thin:oe/oe@$ORACLE_INSTANCE
");
             conn = ods.getConnection();
             }
             catch (Exception e){
                 System.out.println("Got exception " + e);
                 if (rset != null) rset.close();
                 if (conn != null) conn.close();
                 System.exit(0);
                 return ;
             }
            System.out.println("connection works conn " + conn);
            stdin = new java.io.BufferedReader(new
java.io.InputStreamReader(System.in));
```

```
            String query = "declare dopt
dbms_aq.enqueue_options_t; mprop dbms_aq.message_properties_t;
msgid raw(100); begin dopt.visibility := DBMS_AQ.IMMEDIATE;
dopt.delivery_mode := DBMS_AQ.BUFFERED;
dbms_Aq.enqueue('HADOOP_MR_QUEUE', dopt, mprop, hadoop_row_obj(?,
?), msgid); commit; end;";
            CallableStatement pstmt = null;
            try {
              pstmt = conn.prepareCall(query);
            }
            catch (Exception e){
                System.out.println("Got exception " + e);
                if (rset != null) rset.close();
                if (conn != null) conn.close();
            }
             while ((line = stdin.readLine()) != null) {
                 if (line.replaceAll(" ", "").length() < 2)
                     continue;
                 String [] data = line.split(",");
                 if (data.length != 2)
                     continue;
                 countArr++;
                 pstmt.setInt(1, Integer.parseInt(data[0]));
                 pstmt.setInt(2, Integer.parseInt(data[1]));
                 pstmt.executeUpdate();
              }
           if (conn != null) conn.close();
           System.exit(0);
           return;
        }
```

```
    }
```

Query Requesting Data

An example of running a select on the system as described above – leveraging the table function processing – would be:

```
-- Set up phase for the data queue
execute DBMS_AQADM.CREATE_QUEUE_TABLE('HADOOP_MR_QUEUE',
'HADOOP_ROW_OBJ');
execute DBMS_AQADM.CREATE_QUEUE('HADOOP_MR_QUEUE',
'HADOOP_MR_QUEUE');
execute DBMS_AQADM.START_QUEUE ('HADOOP_MR_QUEUE');

-- Query being executed by an end user or BI tool
-- Note the hint is not a real hint, but a comment
-- to clarify why we use the cursor
select max(a), avg(b)
from table(hdfs_reader.read_from_hdfs_file
           (cursor
           (select /*+ FAKE cursor to kick start parallelism */ *
            from orders), '/home/hadoop/eq_test4.sh'));
```

Conclusion

As the examples in this paper show integrating a Hadoop system with Oracle Database 11g is easy to do.

The approaches discussed in this paper allow customers to stream data directly from Hadoop into an Oracle query. This avoids fetching the data into a local file system as well as materializing it into an Oracle table before accessing it in a SQL query.

Integrating Hadoop Data
with Oracle Parallel Processing
January 2010
Authors: Shrikanth Shankar, Alan Choi and
Jean-Pierre Dijcks

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com

Oracle is committed to developing practices and products that help protect the environment

Copyright © 2010, Oracle and/or its affiliates. All rights reserved. This document is provided for information purposes only and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

0109