IBM® Netezza® Analytics
Release 11.x

Map/Reduce Developer's Guide

Revised: Oct. 05, 2017

Part Number 00J2351-05 Rev. 8

Contents

List of Tables

Preface

Audience for This Guide

The IBM Analytics map/reduce feature is intended for developers interested in creating and developing map/reduce applications to run on the IBM Netezza appliances. It is recommended that before using map/reduce, you have an understanding of the basic operation and concepts of the IBM Netezza appliance and the NPS.

Purpose of This Guide

This guide describes the map/reduce feature and describes how to run map/reduce applications using Java.

Symbols and Conventions

Note on Terminology: The terms *User-Defined Analytic Process (UDAP)* and *Analytic Executable (AE)* are synonymous.

The following conventions apply:

- Italics for emphasis on terms and user-defined values, such as user input.
- Upper case for SQL commands, for example, INSERT or DELETE.
- Bold for command line input, for example, **`nzsystem stop`**.
- Bold to denote parameter names, argument names, or other named references.
- Angle brackets (< >) to indicate a placeholder (variable) that should be replaced with actual text, for example, `inza-<release_number>.zip`.
- A single backslash ("\") at the end of a line of code to denote a line continuation. Omit the backslash when using the code at the command line, in a SQL command, or in a file.
- When referencing a sequence of menu and submenu selections, the ">" character denotes the different menu options, for example, **Menu Name > Submenu Name > Selection**.

If You Need Help

If you are having trouble using the IBM Netezza appliance, IBM Netezza Analytics or any of its components:

1. Retry the action, carefully following the instructions in the documentation.
2. Go to the IBM Support Portal at: http://www.ibm.com/support. Log in using your IBM ID and password. You can search the Support Portal for solutions. To submit a support request, click the '**Service Requests & PMRs**' tab.
3. If you have an active service contract maintenance agreement with IBM, you may contact customer support teams via telephone. For individual countries, please visit the Technical Support section of the IBM Directory of worldwide contacts (http://www14.software.ibm.com/webapp/set2/sas/f/handbook/contacts.html#phone)

Comments on the Documentation

We welcome any questions, comments, or suggestions that you have for the IBM Netezza documentation. Please send us an e-mail message at netezza-doc@wwpdl.vnet.ibm.com and include the following information:

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your name, address, and phone number

We appreciate your comments.

CHAPTER 1

Introduction

The MapReduce Paradigm

The MapReduce paradigm was created in 2003 to enable processing of large data sets in a massively parallel manner. The goal of the MapReduce model is to simplify the approach to transformation and analysis of large datasets, as well as to allow developers to focus on algorithms instead of data management. The model allows for simple implementation of data-parallel algorithms. There are a number of implementations of this model, including Google's approach, programmed in C++, and Apache's Hadoop implementation, programmed in Java. Both run on large clusters of commodity hardware in a shared-nothing, peer-to-peer environment.

The MapReduce model consists of two phases: the map phase and the reduce phase, expressed by the map function and the reduce function, respectively. The functions are specified by the programmer and are designed to operate on key/value pairs as input and output. The keys and values can be simple data types, such as an integer, or more complex, such as a commercial transaction.

- **Map**. The map function, also referred to as the map task, processes a single key/value input pair and produces a set of intermediate key/value pairs.
- **Reduce**. The reduce function, also referred to as the reduce task, consists of taking all key/value pairs produced in the map phase that share the same intermediate key and producing zero, one, or more data items.

Note that the map and reduce functions, do not address the parallelization and execution of the MapReduce jobs. This is the responsibility of the MapReduce model, which automatically takes care of distribution of input data, as well as scheduling and managing map and reduce tasks.

Netezza Map/Reduce

The IBM Netezza map/reduce feature is a software framework that allows you to implement MapReduce applications and run them on the Netezza appliance. With the Netezza approach, input data is stored in a distributed table. The output of a job is stored in the database as well. Mapper tasks work on independent parts of an input table called *dataslices*. The map outputsare sorted and redistributed to reduce tasks. The framework creates and runs the appropriate SQL query to perform the map/reduce data flow. Database columns (record fields) are mapped to the keys and values concepts of “the MapReduce model.

This guide provides a comprehensive description of how to start working with Netezza map/reduce. It is divided into the following parts:

- Simple Example for Getting Started – Describes how to quickly write and run your first map/reduce program.
- User Interfaces – Contains basic information about the API. For more detailed information, see the *IBM Netezza Analytics map/reduce API Reference* manual.
- Advanced Functionality – Describes more advanced concepts, such as generic command line options, counters, and logging.
- Netezza Analytics Map/Reduce Examples – Provides four map/reduce examples based on the JARs distributed with the map/reduce software.
- Netezza Analytics Map/Reduce Streaming – Explains map/reduce streaming, which allows you to run map/reduce programs written in languages other than Java.

CHAPTER 2

Simple Example for Getting Started

This section provides a basic example, WordCount, to illustrate how to write and run a map/reduce program on the NPS.

Before running the example, verify that the map/reduce cartridge is installed and registered in the Netezza database. If not, type the following commands to install and register the map/reduce feature:

```
$ nzcm -i mapreduce
$ nzcm -r mapreduce
```

For additional information, see the *IBM Netezza Analytics Administrator's Guide*.

WordCount Example

WordCount is a simple map/reduce application that processes input text and counts the number of occurrences of each word.

Source Code for the WordCount Example

In this example, the map function splits the input text into words and emits <word, "1"> pairs for all occurrences of each word it encounters. The reduce function then sums all counts emitted for the word. The code to execute the example is as follows:

```
import java.io.*;
import java.util.*;

import org.netezza.inza.mr.conf.*;
import org.netezza.inza.mr.io.*;
import org.netezza.inza.mr.mapreduce.*;
import org.netezza.inza.mr.util.*;

public class WordCount extends Configured implements Tool

       { public static class Map extends
```

```
                        Mapper<LongWritable, Text, Text, IntWritable> {

                private final static IntWritable one = new IntWritable(1);

                private Text word = new Text();

                @Override
                public void map(LongWritable key, Text value, Context context)
                        throws IOException, InterruptedException {

                        String line = value.toString();
                        StringTokenizer tokenizer = new StringTokenizer(line);
                        while (tokenizer.hasMoreTokens()) {
                                word.set(tokenizer.nextToken());
                                context.write(word, one);
                        }
                }
        }

        public static class Reduce extends
                        Reducer<Text, IntWritable, Text, IntWritable> {

                @Override
                public void reduce(Text key, Iterable<IntWritable> values,
                        Context context) throws IOException,
InterruptedException {

                        int sum = 0;
                        for (IntWritable val : values) {
                                sum += val.get();
                        }

                        context.write(key, new IntWritable(sum));
                }
        }

        public int run(String[] args) throws Exception
                { if (args.length != 7) {
                        System.out.printf("Usage: %s <db> " +
                                        "<input_table> <input_key_column>
<input_value_column>" +
                                        "<output_table> <output_key_column>
<output_value_column>\n",
                                        getClass().getSimpleName());

                        return -1;
                }

                Configuration conf = getConf();
                final int MAX_WORD_LENGTH = conf.getInt("max.word.length", 100);

                Job job = new Job(conf, "wordcount");
                job.setJarByClass(getClass());

                job.setDatabaseName(args[0]);

                job.setInputTableName(args[1]);
                job.setInputKeyColumnNames(args[2]);
                job.setInputValueColumnNames(args[3]);

                job.setOutputTableName(args[4]);
                job.setOutputKeyColumnNames(args[5]);
                job.setOutputValueColumnNames(args[6]);

                job.setMapperClass(Map.class);
                job.setReducerClass(Reduce.class);

                job.setMapInputKeyClass(LongWritable.class);
                job.setMapInputValueClass(Text.class);
                job.setMapOutputKeyClass(Text.class);
                job.setMapOutputKeyColumnSize(0, MAX_WORD_LENGTH);
                job.setMapOutputValueClass(IntWritable.class);
                job.setReduceOutputKeyClass(Text.class);
                job.setReduceOutputKeyColumnSize(0, MAX_WORD_LENGTH);
                job.setReduceOutputValueClass(IntWritable.class);
                boolean success = JobRunner.runJob(job);
                return success ? 0 : 1;
        }

        public static void main(String[] args) throws Exception
                { int ret = ToolRunner.run(new WordCount(), args);
                System.exit(ret);
        }
}
```

Running the Example

Complete the following steps to run the WordCount example:

1. Compile **WordCount.java** and create a .jar file:

```
$ export JAVA_HOME=/nz/export/ae/languages/java/java_sdk/host
    $ export MR_HOME=/nz/export/ae/products/netezza/mapreduce/current
    $ mkdir wordcount_classes
    ${JAVA_HOME}/bin/javac \
          -classpath ${MR_HOME}/mapreduce.jar \
          -d wordcount_classes \ WordCount.java

    ${JAVA_HOME}/bin/jar -cvf wordcount.jar -C wordcount_classes/ .
```

2. Create a database, mapreduce_db:

```
CREATE DATABASE mapreduce_db;
```

3. Connect to the mapreduce_db and seed table **wordcount_input** with sample data:

```
CREATE TABLE wordcount_input(
    id int,
    text varchar(100)
);

INSERT INTO wordcount_input VALUES(1, 'Hello World Bye World');
INSERT INTO wordcount_input VALUES(2, 'Hello mapreduce Goodbye
mapreduce');
INSERT INTO wordcount_input VALUES(3, 'Hello INZA');
```

   1. Run the application with the following command:

```
${MR_HOME}/bin/mapreduce jar wordcount.jar WordCount    \
      mapreduce_db                                  \
                                    wordcount_input id text   \
                  wordcount_output word count
```

4. Show output table:

```
$ nzsql ${DB} -c "select * from wordcount_output"
   WORD    | COUNT
           +
----------Hello | -------3
 INZA      |      1
 World     |      2
 mapreduce |      2
 Bye       |      1
 Goodbye   |      1

(6 rows)
```

CHAPTER 3
User Interfaces

This section briefly describe user interfaces and class components of the Netezza Analytics map/reduce API. For a more complete description of each interface and class, see the *IBM Netezza Analytics Map/Reduce API Reference* manual.

The version of the map/reduce API provided with Netezza Analytics was designed to be similar to standard Hadoop map/reduce. However, to take advantage of the AMPP environment of the IBM Netezza appliances, there were changes made in the API that prevent full compatibility and require some map/reduce jobs to be rewritten.

There are two sets of classes that can be used to implement a map/reduce job. These classes are located in two separate packages--**org.netezza.inza.mr.mapred** and **org.netezza.inza.mr.mapreduce**. This document describes classes from the **org.netezza.inza.mr.mapreduce** package. For more information, see the *IBM Netezza Analytics Map/Reduce API Reference* manual.

The Mapper Class

Mapper maps input <key, value> pairs to a set of intermediate <key, value> pairs. The intermediate pairs do not need to be of the same type as the input pairs. A given input pair may map to zero or to many output pairs.

<K1, V1>* → Mapper → <K2, V2>*

Netezza's map/reduce, for a given input table, runs one map task per dataslice. As a result, the number of map tasks is determined by the number of dataslices over which the input table is distributed. To optimize mapper performance, ensure that data is evenly distributed on dataslices.

The Mapper implementation consists of four methods: setup(), map(), cleanup() and run(). By default, the run() method calls the setup() method once, then the map() method for each input <key, value> pair, and finally, the cleanup() method. For example:

```
      public void run(Context context) throws IOException,
    InterruptedException {
        setup(context);
```

```
          while (context.nextKeyValue()) {
            map(context.getCurrentKey(), context.getCurrentValue(), context);
          }
          cleanup(context);
        }
```

The map() method, by default, emits its input as an output, acting as an identity function, and should be overridden. (You can override any of these methods, including run() for advanced use cases.)

The Context Object for Mapper

Mapper has access to the context object, which allows the mapper to interact with the rest of the environment. The context object, which is passed as an argument to the mapper and other methods, provides access to configuration data for the job and allows the mapper to emit output pairs using the Context.write(key, value) method.

The Context.getConfiguration() method returns a configuration object containing configuration data for arunning map/reduce program. You can set arbitrary (key, value) pairs of configuration data in the job, (for example with the Job.getConfiguration().set("myKey", "myVal") method), and then retrieve this data in the mapper with Context.getConfiguration().get("myKey") method. (This is typically done with the mapper's setup() method.)

The cleanup() method is mainly overriden (by default no cleanup occurs). If you need to perform cleanup after all input is processed, you can override the default and set it to match your given use case.

The Reducer Class

After sorting and redistributing intermediate data, all values associated with the same key are processed by a single reducer (the same instance of the Reducer class)., Typically, the reducer aggregates the given values to a final result. The number of reduces is determined by the distribution of intermediate data. That is, the framework launches one reduce task on each dataslice.

<K2, list(V2)>* → Reducer → <K3, V3>*

The Reducer implementation consists of four methods: setup(), reduce(), cleanup() and run(). By default, the run() method calls the setup() method once, then the reduce() method for each <key, (list of values)> pair, and finally, the cleanup() method. For example:

```
        public void run(Context context) throws IOException,
      InterruptedException {
          setup(context);
          while (context.nextKey()) {
            reduce(context.getCurrentKey(), context.getValues(), context);
          }
          cleanup(context);
        }
```

The reduce() method, by default, emits its input as an output, acting as an identity function, and should be overridden. (You can override any of these methods, including run() for advanced use cases.)

The Context Object for Reducer

Reducer has access to the context object, which allows the reducer to interact with the rest of the environment. The context object, which is passed as an argument to the reducer and other methods, provides access to configuration data for the job and allows the reducer to emit output pairs using the Context.write(key, value) method.

The Context.getConfiguration() method returns a configuration object containing configuration data for a running map/reduce program. You can set arbitrary (key, value) pairs of configuration data in the job, (for example with the Job.getConfiguration().set("myKey", "myVal") method), and then retrieve this data in the reducer with the Context.getConfiguration().get("myKey") method. (This is typically done in the reducer's setup() method.)

The cleanup() method is mainly overriden (by default no cleanup occurs). If you need to perform cleanup after all input is processed, you can override the default and set it to match your given use case.

You do not need to define a reducer class for the job. If you do not define a class, the mapper's output records are stored directly in the output table without sorting them and partitioning redistribution and reduce phases are skipped).

Partitioner Class

Partitioner controls the partitioning of intermediate data. It assigns a partition number for each intermediate <key, value> pair. The framework uses this number to redistribute intermediate records before the reduce step starts. Records with the same partition number will be stored on the same dataslice. You can define your own Partitioner class, which overrides the getPartition(KEY key, VALUE value, int numPartitions) partitioning method. The value of *numPartitions* is calculated by the framework and corresponds to the number of available NPS dataslices that can run map/reduce tasks.

When no partitioner is defined for a job, data is redistributed according to values produced by NPS' build-in hash function applied to intermediate keys.

Writable

All key and value classes used in map and reduce signatures have to implement a Writable interface. This allows the framework to serialize/deserialize <key,value> pairs to and from database records.

Netezza Analytics map/reduce API provides implementation of the following basic Writables: BooleanWritable, IntWritable, LongWritable, FloatWritable, DoubleWritable, Text, Ntext, and NullWritable. For more information, see the *IBM Netezza Analytics Map/Reduce API Reference* manual.

A writable interface allows you to specify multicolumn key/value types. The getStorageTypesList() method returns a list of classes that can be mapped by the framework to types of database columns. The following table shows the mapping:

Table 1: Java class mappings

Java Class	**Data Type**
java.lang.Boolean	BOOL
java.lang.Byte	BYTEINT
java.lang.Short	SMALLINT
java.lang.Integer	INTEGER
java.lang.Long	BIGINT
java.lang.Float	FLOAT
java.lang.Double	DOUBLE
java.lang.String	VARCHAR
org.netezza.inza.mr.io.type.NString	NVARCHAR

The following example of a custom Writable consists of three fields: int, float, and boolean. The getStorageTypesList() method used in the example returns a list of corresponding java classes: Integer.class, Float.class, Boolean.class. The method allows the framework to create appropriate readers and writers.

```
public class CustomWritable implements Writable {

        private int a;
        private float b;
        private boolean c;

        @Override
        public void write(RecordOutput out) throws IOException
                { out.writeInt(a);
                out.writeFloat(b);
                out.writeBoolean(c);
        }

        @Override
        public void readFields(RecordInput in) throws IOException
                { a = in.readInt();
                b = in.readFloat();
                c = in.readBoolean();
        }

        @Override
        public List<Class<?>> getStorageTypesList() {
                List<Class<?>> ret = new ArrayList<Class<?>>();
                ret.add(Integer.class);
                ret.add(Float.class);
                ret.add(Boolean.class);
                return ret;
        }

}
```

Output database types are strictly determined by Writable's storage types defined in getStorageTypesList() method. However, the framework supports the following automatic conversions while reading data from database records to Writable types.

Table 2: Automatic conversions from database record to Writable type

	Boolean	Integer	Long	Float	Double	Text	NText
BOOL	♦					♦	♦
FLOAT				♦	♦	♦	♦
DOUBLE					♦	♦	♦
BYTEINT		♦	♦	♦	♦	♦	♦
SMALLINT		♦	♦	♦	♦	♦	♦
INTEGER		♦	♦	♦	♦	♦	♦
BIGINT			♦	♦	♦	♦	♦

	Boolean	Integer	Long	Float	Double	Text	NText
NUMERIC						♦	♦
CHAR	♦	♦	♦	♦	♦	♦	♦
VARCHAR	♦	♦	♦	♦	♦	♦	♦
NCHAR	♦	♦	♦	♦	♦	♦	♦
NVARCHAR	♦	♦	♦	♦	♦	♦	♦
DATE						♦	♦
TIME						♦	♦
TIMESTAMP						♦	♦
TIMETZ						♦	♦
INTERVAL						♦	♦

The Job Object

The Job object specifies a map/reduce job's settings. Typically it is set in the run() method (Tool interface, described later). It should contain all necessary data to create a map/reduce SQL query (which is then submitted to the NPS). This data includes, but is not limited to:

- database name
- input table name
- input column names
- output table name
- output column names
- mapper/reducer/combiner classes
- input/output key/value classes
- partitioner class

See the *IBM Netezza Analytics Map/Reduce API Reference* for more information.

Job is submitted using the JobRunner.runJob(Job) method. Before creating a SQL query and running it on the NPS, the framework performs job validation. If something is wrong with the job's configuration, an error message is logged.

CHAPTER 4
Advanced Functionality

Generic Command-Line Options

The command-line options allow you to define files, archives, and additional libraries, which can then be used by tasks during job execution. There are also options for modifying configuration properties from the command-line.

The supported command-line options are described in the table below:

Table 3: Supported command line options

Option	Description
-D property=value	Sets a configuration property to a given value.
-conf file.xml	Loads the set of configuration properties defined in the specified XML file.
-files file1,file2	Copies the specified files to a job-unique shared directory, making them visible for map/reduce tasks.
-archives archive1,archive2	Copies the specified archives to a job-unique shared directory and unarchives them, making them visible for map/reduce tasks. (Supported archive types are: zip, tar, tgz, tar.gz)
-libjars jar1,jar2	Copies the specified JAR files to a job-unique shared directory and adds them to the CLASSPATH for job and map/reduce tasks.

Netezza's map/reduce application implements the Tool interface to support generic command-line options. The application then delegates the handling of standard command-line options to GenericOptionsParser via ToolRunner.run(Tool, String[]) and only handle its custom arguments. For more information, see the *IBM Netezza Analytics Map/Reduce API Reference* manual.

Counters

Counters, both built-in and user-defined, are used for counting a variety of values during job execution. For example, built-in Counters monitor the number of processed records and the number of output records for each phase of the map/reduce process. You can also define your own counters using the Counter object's getCounter() method in the task Context. You can increment the user-defined counters locally with the Counter.increment(long) method. At the end of the task execution, the Counter's values are saved in a <task><id>.cnt file. After the job finishes, counters are summed up to the total values, saved in the total.cnt file, and available in the log (main.log).

A summary of counters for the WordCount example, from above, may look like this:

```
JOB_COUNTERS
        Launched map tasks=3
        Launched reduce tasks=3
MAPPER_COUNTERS
        Map input bad records=0
        Map input records=3
        Map output records=10
REDUCER_COUNTERS
        Reduce input groups=6
        Reduce input records=10
        Reduce output records=6
```

The following is the full list of built-in Counters:

Table 4: Built-in counters

Group	Counter
JOB_COUNTERS	Launched map tasks
	Launched reduce tasks
	Launched combine tasks
MAPPER_COUNTERS	Map input records
	Map input bad records
	Map output records
COMBINER_COUNTERS	Combine input records
	Combine output records

Group	Counter
REDUCER_COUNTERS	Reduce input records
	Reduce input groups
	Reduce output records

Logging

The Netezza Analytics map/reduce framework incorporates log4j loggers. All logs are saved in the $MPAREDUCE_HOME/jobs/<job_dir>/logs directory. Logs are divided into the following groups:

Table 5: Available logs in the map/reduce framework

Log	Description
Main log	
main.log	Contains basic information about the job, such as validating args, validating job, job completion status, number of counters, deploy directory, number of mapper/reducer/combiner tasks.
Shapers' logs	
mapper_shaper.log	Contains success notification or error messages regarding running the shaper during the map phase.
combiner_shaper.log	Contains success notification or error messages regarding running the shaper during the combiner phase.
reducer_shaper.log	Contains success notification or error messages regarding running the shaper during the reducer phase.
Tasks' logs	
mapper<id>.log	Each mapper creates one log file containing mapper-specific information (for example, number of records read by the mapper, number of bad records, number of output records).
combiner<id>.log	An optional log file that is created only if combiners were used. These files contain basic combiner-specific information such as number of combine input records and number of combine output records.

Log	Description
reducer<id>.log	An optional log file that is created only if reducers were used. These files contain basic reducer-specific information such as number of records read by the reducer and number of output records.

You can also use your own loggers to provide more detailed information in the log files. To do this, you must acquire a Logger instance in your source code (see the following example, with the Logger instance related lines marked in red).

The following is an example using a user-defined logger:

```
import  org.apache.log4j.Logger;
import  …

public class WordCount extends Configured implements Tool
      { static Logger log =
      Logger.getLogger(WordCount.class);
      ...
      public static class Map extends Mapper<LongWritable, Text, Text,
IntWritable> {
      ...
            @Override
            public void map(LongWritable key, Text value,
Context context) throws IOException, InterruptedException {
                  String line = value.toString();
                  StringTokenizer tokenizer = new StringTokenizer(line);
                  while (tokenizer.hasMoreTokens()) {
                        word.set(tokenizer.nextToken());
                        log.info("word = " + word);
                        context.write(word, one);
                  }
            }
      }

      public static class Reduce extends Reducer<Text, IntWritable, Text,

IntWritable> {

      ...
            @Override
            public void reduce(Text key, Iterable<IntWritable> values,

Context context) throws IOException, InterruptedException {

                  int sum = 0;
                  for (IntWritable val : values) {
                        sum += val.get();
                  }
                  totalSum.set(sum);
                  log.info("sum = " + sum);
                  context.write(key, totalSum);
            }
      }

      public int run(String[] args) throws Exception {
```

```
                Job job = new Job(getConf(),
                getClass().getSimpleName()); ...
        }

        public static void main(String[] args) throws Exception
                { int ret = ToolRunner.run(new WordCount(), args);
                System.exit(ret);
        }
}
```

Using Map/Reduce AEs in SQL Queries

The Netezza Analytics map/reduce framework provides the ability to invoke mappers and reducers directly from nzsql. There are three general Analytic Executables (AEs) registered with the mapreduce cartridge in the **inza** database. These can be used in a SQL query to execute a particular mapper, reducer, or combiner from a Java class. The AEs are registered as User-Defined Table Functions (UDTF) with the following names:

- MapperAE
- ReducerAE
- CombinerAE

These AEs have a common signature:

<Task>AE(K1, … , Kn, V1, … , Vm, 'NZAE_PARAMETER2=<conf.xml>')

where:

K1, … , Kn	are key columns
V1, … ,Vm	are value columns
<conf.xml>	is a path to a configuration file that contains all properties needed to run the task.

Properties

The following table lists the properties available to each AE.

Table 6: Properties available to the AEs registered by the mapreduce cartridge

Property	Mapper	Combiner	Reducer
mapreduce.job.run.dir	♦	♦	♦
mapreduce.job.jar	♦^{1}	♦^{1}	♦^{1}
mapreduce.job.map.class	♦		
mapreduce.map.input.key.class	♦		
mapreduce.map.input.value.class	♦		
mapreduce.map.output.key.class	♦	♦	♦
mapreduce.map.output.key.columns.sizes	♦		
mapreduce.map.output.value.class	♦	♦	♦
mapreduce.map.output.value.columns.sizes	♦		
mapreduce.job.combine.class	♦	♦	
mapreduce.combine.output.key.class	♦	♦	♦2
mapreduce.combine.output.key.columns.sizes	♦	♦	
mapreduce.combine.output.value.class	♦	♦	♦2
mapreduce.combine.output.value.columns.sizes		♦	
mapreduce.job.reduce.class			♦
mapreduce.reduce.output.key.class			♦
mapreduce.reduce.output.key.columns.sizes			♦
mapreduce.reduce.output.value.class			♦
mapreduce.reduce.output.value.columns.sizes			♦

♦=required
x^1=required when job's jar available
♦^{2}=required when mapreduce.job.combine.class is defined

Example

Input data

```
$ nzsql -d mapreduce_db -c "select * from wordcount_input"
 ID |              TEXT
----+-----------------------------------
  1 | Hello World Bye World
  2 | Hello mapreduce Goodbye mapreduce
  3 | Hello INZA
(3 rows)
```

Configuration (wordcount.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<property><name>mapreduce.job.run.dir</name><value>/nz/export/ae/workspac
e/mapreduce/myjob</value></property>
<property><name>mapreduce.job.jar</name><value>/nz/export/ae/products/net
ezza/mapreduce/current/mapreduce-examples.jar</value></property>
<property><name>mapreduce.job.map.class</name><value>org.netezza.inza.mr.
examples.WordCount$Map</value></property>
<property><name>mapreduce.map.input.key.class</name><value>org.netezza.in
za.mr.io.LongWritable</value></property>
<property><name>mapreduce.map.input.value.class</name><value>org.netezza.
inza.mr.io.Text</value></property>
<property><name>mapreduce.map.output.key.class</name><value>org.netezza.i
nza.mr.io.Text</value></property>
<property><name>mapreduce.map.output.key.columns.sizes</name><value>100</
value></property>
<property><name>mapreduce.map.output.value.class</name><value>org.netezza
.inza.mr.io.IntWritable</value></property>
<property><name>mapreduce.map.output.value.columns.sizes</name><value>-
1</value></property>
<property><name>mapreduce.job.reduce.class</name><value>org.netezza.inza.
mr.examples.WordCount$Reduce</value></property>
<property><name>mapreduce.reduce.output.key.class</name><value>org.netezz
a.inza.mr.io.Text</value></property>
<property><name>mapreduce.reduce.output.key.columns.sizes</name><value>10
0</value></property>
<property><name>mapreduce.reduce.output.value.class</name><value>org.nete
zza.inza.mr.io.IntWritable</value></property>
<property><name>mapreduce.reduce.output.value.columns.sizes</name><value>
-1</value></property>
</configuration>
```

SQL query (wordcount.sql)

```
CREATE TABLE wordcount_output AS
WITH
MAPOUT_RAW AS (
        SELECT DATASLICEID, M.K0 AS K0, M.V0 AS V0
        FROM wordcount_input AS I, TABLE WITH FINAL (INZA..MAPAE(I.id,
I.text,
'NZAE_PARAMETER2="/nz/export/ae/workspace/mapreduce/myjob/wordcount.xml"'
))ASM
 )

,
INTERMEDIATEOUT_REDISTRIBUTED_GLOBALLY AS (
        SELECT RIN.K0 AS K0, RIN.V0 AS V0,
                RANK() OVER (PARTITION BY RIN.K0 ORDER BY -1) AS
        RK FROM MAPOUT_RAW AS RIN
)
,
REDUCEOUT_RAW AS (
        SELECT
                CASE WHEN IRG.RK IS NULL OR IRG.RK IS NOT NULL THEN R.K0
END AS K0
                ,
                R.V0 AS V0
        FROM INTERMEDIATEOUT_REDISTRIBUTED_GLOBALLY AS IRG,
        TABLE WITH FINAL(INZA..REDUCEAE(IRG.K0, IRG.V0,
'NZAE_PARAMETER2="/nz/export/ae/workspace/mapreduce/myjob/wordcount.xml"' ))ASR

)

SELECT JO.K0 AS word, JO.V0 AS count
FROM REDUCEOUT_RAW AS JO
```

Output

```
$ nzsql -d mapreduce_db -c "select * from wordcount_output"
   WORD    | COUNT
-----------+-------
 Hello     |     3
 INZA      |     1
 World     |     2
 mapreduce |     2
 Bye       |     1
 Goodbye   |     1
(6 rows)
```

CHAPTER 5
Netezza Analytics Map/Reduce Examples

The mapreduce-examples.jar

The mapreduce-examples.jar, shipped with the map/reduce cartridge, includes a set of four examples. Each example, DBCountPageView, Grep, QuasiMonteCarlo, and WordCount, is described in this chapter. The jar is located in /nz/export/ae/products/netezza/mapreduce/current/. You can run it directly on a remote machine using the **mapreduce** command.

Before You Begin

Before running the examples, you must install and register the Netezza Analytics map/reduce cartridge. Use the following commands to install and register map/reduce in the **inza** database:

```
$ nzcm -i mapreduce
$ nzcm -r mapreduce
$ export PATH=$PATH:/nz/export/ae/products/netezza/mapreduce/current/bin/
```

The **mapreduce-examples.jar** file contains a set of examples with the following class names:

- **dbcount**, for click counting
- **grep**, for counting words matching a pattern
- **pi**, for estimating Pi number using the quasi-Monte Carlo method
- **wordcount**, for counting words in a document collection

All examples in **mapreduce-examples.jar** can be run using the following command structure:

```
$ mapreduce jar mapreduce-examples.jar <class_name> <args>
```

To view the usage, run the command without arguments. For example, using the wordcount class_name without arguments:

```
$ mapreduce jar mapreduce-examples.jar wordcount
```

Produces the following output:

```
Missing required options: outTable, outKeyCols, outValueCols, inTable,
inKeyCols, db, inValueCols, maxWordLength
usage: WordCount -db
                        database name
 <arg> -enableCombiner  enable default combiner
                        enable default partitioner
 -enablePartitioner -   list of input key columns names (comma separated)
 inKeyCols <arg> -      input table name
 inTable <arg> -        list of input value columns names (comma
                        separated)
 inValueCols <arg>      maximum length of a single word in the input table
 -maxWordLength <arg> - list of output key columns names (comma separated)
                        output table name
 outKeyCols <arg> -     list of output value columns names (comma
 outTable <arg> -       separated)
 outValueCols <arg>
```

WordCount Example

The WordCount example below counts the number of occurrences of each word in a given input set. (This example, available in **mapreduce-example.jar**, extends the WordCount example at the beginning of this manual by implementing the Tool interface.

The following sections describe the map and reduce functions, including input and output, and provides the commands needed to run this example.

Running the Example

The command to run the WordCount example is:

```
$ mapreduce jar
/nz/export/ae/products/netezza/mapreduce/current/mapreduce-examples.jar
wordcount -db mapreduce_db -inTable sample_wordcount -inKeyCols key -
inValueCols line -outTable sample_wordcount_out -outKeyCols word -
outValueCols count -maxWordLength 25
```

Before running the command above, be sure that:

- the **mapreduce_db** database exists
- the **sample_wordcount** input table exists (in the mapreduce_db database)
- the **sample_wordcount** input table has key (for example, INT) and line (for example, VARCHAR) columns

Map() Method

The following are characteristics of the map function, including examples of the input and output and the action the map method performs.

- Input: A line key and a line
- Output: A (word, 1) pair for each occurrence of a word in the line
- Action: Splits the line into words

Input Data

```
 KEY |                 LINE
-----+---------------------------------
  2  | Hello map Goodbye reduce
  3  | Hello INZA

  1  | Hello World Bye World
```

Intermediate Output Data

```
WORD    | COUNT
        +--------------------------------
Hello-------- | 1
map     | 1
Goodbye | 1
reduce  | 1
Hello   | 1
INZA    | 1
Hello   | 1
World   | 1
Bye     | 1

World   | 1
```

Reduce() Method

The following are characteristics of the Reduce function, including examples of the input and output and the action the reduce method performs.

- Input: Word and a list of counts of occurrences of this word
- Output: A (word, total count of occurrences) pair
- Action: Sums all counts for a particular word

Input Data

```
WORD    | TOTAL_COUNT
--------+------------
Hello   | (1,1,1)
World   | (1,1)
Goodbye | (1)
INZA    | (1)
map     | (1)
reduce  | (1)
Bye     | (1)
```

Output Data

```
  WORD    | COUNT
         +
---------INZA | -------1
 World    |       2
 Bye      |       1
 Goodbye  |       1
 map      |       1
 Hello    |       3
 reduce   |       1
```

DBCountPageView Example

The DBCountPageView example counts the number of visits to each listed URL page.

The following sections describe the map and reduce functions, including input and output, and provides the commands needed to run this example.

Running the Example

The command to run the CountPageView example is:

```
$ mapreduce jar
/nz/export/ae/products/netezza/mapreduce/current/mapreduce-examples.jar
dbcount -db mapreduce_db -inTable sample_countpageview -inValueCols
url,ref,count -outTable sample_countpageview_out -outKeyCols
url,total_count -maxUrlLength 30
```

Before running the command above, be sure that:

- the **mapreduce_db** database exists
- the **sample_countpageview** input table exists in the mapreduce_db database
- the **sample_countpageview** input table has key (INT) and three value columns: url (VARCHAR), ref (VARCHAR), and count (INT)

Map() Method

The following are characteristics of the map function, including examples of the input and output.

- Input: An access record ID and an access record with a <URL,referrer,count> schema
- Output: A (URL,1) pair for each schema

Input Data

```
 KEY |     URL     | REF  | COUNT
-----+------------+------+
  24 | noname.org | aka  |-------6
  32 | ibm.com    | aka  |       2
  36 | ibm.com    | tul  |      20
  21 | noname.org | aka  |       3
  33 | ibm.com    | aka  |       4
  23 | noname.org | prb  |       5
  31 | ibm.com    | aka  |       1
  35 | ibm.com    | tul  |      16
  22 | noname.org | tul  |       4
  34 | ibm.com    | tul  |       8
```

Intermediate Output Data

```
URL          | COUNT
             +--------------
noname.org------------ | 1
noname.org   | 1
noname.org   | 1
ibm.com      | 1
ibm.com      | 1
ibm.com      | 1
ibm.com      | 1
noname.org   | 1
ibm.com      | 1
ibm.com      | 1
```

Reduce() Method

The following are characteristics of the reduce function, including examples of the input and output and the action the reduce method performs.

- Input: URL and a list of counts of visits of this page
- Output: A (URL, total count of visits) pair
- Action: Sums all counts that were output for each URL

Input Data

```
          URL | LIST OF COUNTS
              +--------------------
-------------ibm.com| {1,1,1,1,1,1}
 noname.org   | {1,1,1,1}
```

Output Data

```
      URL       | TOTAL_COUNT
              +
-----------ibm.com |-------------6
 noname.org |               4
```

QuasiMonteCarlo (PiEstimator) Example

QuasiMonteCarlo (PiEstimator) is a program that estimates the value of Pi using the quasi-Monte Carlo method. The method generates random points in a unit square and counts the number of points that lie inside an inscribed circle of the square. The probability of a point to land in the circle is proportional to the relative areas of the circle and the square. The more points that are generated, the more accurate the approximation is.

For this example, assume that numTotal = numInside + numOutside. The fraction numInside/numTotal is a rational approximation of the value:

(Area of the circle)/(Area of the square)

where the area of the inscribed circle is Pi/4 and the area of the unit square is 1. Then, Pi is estimated to be 4 (numInside/numTotal).

The following sections describe the map and reduce functions, including input and output, and provides the commands needed to run this example.

Running the Example

The command to run the QuasiMonteCarlo example is:

```
$ mapreduce jar
/nz/export/ae/products/netezza/mapreduce/current/mapreduce-examples.jar
pi -db mapreduce_db -nMaps 5 -nSamples 10
```

where nMaps is number of maps, and nSamples is number of samples per map.

Before running the command above, be sure that:

- the **mapreduce_db** database exists

Map() Method

The following are characteristics of the map function, including examples of the input and output:

- Input: Offset and number of samples to be generated. Offset is calculated on the basis of the number of mappers specified.
- Output: Two key value pairs:
- (true, number of points inside the inscribed circle of the square)
- (false, number of points outside of the inscribed circle of the square)
- Action: Calculates whether each generated point lies inside or outside of the inscribed circle of the square and counts points within both categories.

Reduce() Method

The following are characteristics of the Reduce function, including examples of the input and output and the action the reduce method performs.

- Input: Boolean indicating whether points lie inside or outside of the inscribed circle of the square and the number of such points.
- Output: Either (true, number of points inside of the inscribed circle of the square) or (false, number of points outside of the inscribed circle of the square).
- Action: Sums up all counts within each category.

The following is sample output:

```
12/04/16 08:12:17 INFO mapreduce.JobRunner: Validating args
12/04/16 08:12:17 INFO mapreduce.JobRunner: Validating job
12/04/16 08:12:17 INFO mapreduce.JobRunner: Deploying job 12/04/16 08:12:17
INFO mapreduce.JobRunner:
/nz/export/ae/products/netezza/mapreduce/current/jobs/job_QuasiMonteCarlo
_20120416_081217
12/04/16 08:12:18 INFO mapreduce.JobRunner: Running job
12/04/16 08:12:18 INFO mapreduce.JobRunner:     Generating MapReduce
query
12/04/16 08:12:18 INFO mapreduce.JobRunner:     Running MapReduce query
12/04/16 08:12:20 INFO mapreduce.JobRunner: Job complete: SUCCESS
12/04/16 08:12:20 INFO mapreduce.JobRunner: Aggregating counters
12/04/16 08:12:20 INFO mapreduce.JobRunner: Counters: 8
      JOB_COUNTERS
            Launched map tasks=2
            Launched reduce tasks=2
      MAPPER_COUNTERS
            Map input bad records=0
            Map input records=5
            Map output records=10
      REDUCER_COUNTERS
            Reduce input groups=2
            Reduce input records=10
            Reduce output records=2
12/04/16 08:12:20 INFO mapreduce.JobRunner: Cleaning up
run.dir (archives, files, libjars)
Estimated value of Pi is 3.28000000000000000000
```

GrepCount

The GrepCount example takes a regular expression and processes through the input data set to find matching words. It then produces output that lists matching words with the number of occurrences. The following sections describe the map and reduce functions, including input and output, and provides the commands needed to run this example.

Running the Example

The command to run the GrepCount example is:

```
$ mapreduce jar
/nz/export/ae/products/netezza/mapreduce/current/mapreduce-examples.jar
grep -db mapreduce_db -inTable sample_grepcount -inValueCols text -
maxMatchingLength 100 -outTable sample_grepcount_out -outKeyCols word -
outValueCols total_count -pattern '[A-Z][a-z]+[A-Z]'
```

Before running the command above, be sure that:

- the **mapreduce_db** database exists
- the **sample_grepcount** input table exists in the **mapreduce_db** database
- the **sample_grepcount** input table has a text (VARCHAR) column

Map() Method

The following are characteristics of the map function, including examples of the input and output.

- Input: A file ID and a file content
- Output: A ('word', 1) pair for each word matching the regex
- Action: Splits every file content to single words and performs matching against given regEx

Input Data

```
Regex: [A-Z][a-z]+[A-Z]

 KEY | TEXT
-----+-----------------------------------
   1 | GREP grep GreP gREp
   2 | Grep example
   3 | AbbA DooR MOON
   4 | Ab bA Do oR Mo oN
   5 | MooN Earth SuN
   6 | BACKDooR SuNrise SuNset MooNlight
```

Intermediate Output Data

```
      WORD | NUM
-----------+-------------
      GreP | 1
      AbbA | 1
      DooR | 2
      MooN | 2
       SuN | 3
      DooR | 1
       SuN | 1
       SuN | 1
      MooN | 1
```

Reduce() Method

The following are characteristics of the reduce function, including examples of the input and output and the action the reduce method performs.

- Input: Word and a list of its occurrences
- Output: A (word, total number of occurrences) pair
- Action: Sums together all counts that were output for a particular word

Input Data

```
      WORD | LIST OF COUNTS
-----------+------------------
      GreP | {1}
      AbbA | {1}
      DooR | {1,1}
      MooN | {1,1}
       SuN | {1,1,1}
```

Output Data

```
      WORD | TOTAL_COUNT
-----------+-------------
      GreP | 1
      AbbA | 1
      DooR | 2
      MooN | 2
       SuN | 3
```

CHAPTER 6
Netezza Analytics Map/Reduce Streaming

In general, streaming allows you to run programs written in languages other than Java as map/reduce jobs. Such a program reads data from standard input and then outputs results to standard output. Each line of input consists of a key and a value, separated by a tabulator. The program processes data line-by-line and outputs tab-separated key-value pairs. Streaming is suitable only for processing text data.

Using Netezza Analytics Map/Reduce Streaming

Programs using Netezza Analytics map/reduce Streaming are launched on individual SPUs. User-created scripts or programs must be able to be executed on the SPU.

Each program used as a mapper, reducer, or combiner job receives subsequent lines containing a key and value separated by a tabulator. Keys and values are taken from the specified database and table. Only text data is supported. The program performs computations and outputs new tab-separated key-value pairs that are then inserted into the output table. Both key and value columns have VARCHAR type with a user-specified size.

Run Parameters

Netezza Analytics map/reduce streaming is run using the **mapreduce jar** command. When executing, you must also specify the path to the Netezza Analytics map/reduce streaming jar. Each streaming invocation must have at least one mapper job specified. Following is the syntax for the **mapreduce-streaming.jar** command:

```
mapreduce jar
/nz/export/ae/products/netezza/mapreduce/current/mapreduce-streaming.jar
-db <db>
-input <table name> <key_column> <value column>
-output <output_table> <key_column>
<value_column> -mapper <mapper_cmd>
-mapper_out_key_size <size>
-mapper_out_value_size <size>
```

```
    -reducer <reducer_cmd>
    -reducer_out_key_size <size>
    -reducer_out_value_size <size>
    -file <file>
```

The following table describes the required parameters.

Table 7: Required parameters for the mapreduce jar command

Parameter	Description
db	Specifies the name of the database containing input data.
input <table_name> <key_column> <value_column>	Specifies the name of the table containing the input data and the names of the columns where key and value data is stored.
output <table_name> <key_column> <value_column>	Specifies the name of the table where output data will be stored, followed by the names of key and value columns.
mapper	Executes the map step on the SPU.
mapper_out_key_size	Specifies the size (number of characters) of the output key column created after the map step.
mapper_output_value_size	Specifies the size (number of characters) of the output value column created after the map step.

The combine and reduce steps are optional. They can be defined by specifying the following parameters.

Table 8: Parameters for optional combine and reduce

Parameter	Description
combiner	Executes the combine step on the SPU.
combiner_out_key_size	Specifies the size (number of characters) of the output key column created after the combine step.
combiner_output_value_size	Specifies the size (number of characters) of the output value column created after the combine step.
reducer	Executes the reduce step on the SPU.
reducer_out_key_size	Specifies the size (number of characters) of the output key column created after the reduce step.
reducer_output_value_size	Specifies the size (number of characters) of the output value column created after the reduce step.

You must use the **file** parameter (specified by the **streaming** command syntax)to specify each file that you want to run mapper/combiner/reducer commands on. Multiple file parameters are allowed. All files are copied to a temporary directory that is accessible by the SPUs.

Within the streaming program, one line of input and output consists of key and value entries separated by a tab character. However, you can define other separators to be used to distinguish key from value by specifying the following parameters.

Table 9: Alternate separators

Input Separators	
mapper_output_separator combiner_output_separator reducer_output_separator	Tab character by default, or set to any chosen symbol.
Output Separators	
mapper_output_separator combiner_output_separator reducer_output_separator	Tab character by default, or set to any chosen symbol.

Parameters to the mapper, combiner, or reducer are passed using environment variables on the SPUs. To pass parameters, use the **cmdenv** option. For example, enter the following to set the environment variable “NAME” on the SPU to contain the value ADAM:

-cmdenv “NAME=ADAM”

The variable can be subsequently read in a program or script with commands specific to the programming language being used.

Note: It is possible to mix streaming functionality with standard Java mapper/reducer/combiner classes. Instead of defining mapper/combiner/reducer parameters, you specify **mapperClass/combinerClass/reducerClass** parameters with appropriate class names. All java classes should be included in jar archives and passed to the **mapreduce_streaming** executable through the **libJar** parameter.

Netezza Analytics Map/Reduce Streaming Examples

WordCount in Perl

This section includes source code and instructions to run the WordCount example written in Perl.

Following is the source code for the mapper. Save this code to a file called **WordCount_mapper.pl**:

```
#!/usr/bin/env perl
foreach (<STDIN>) {
        my @pair = split("\t", $_);
        my @words = split(" ", $pair[1]);
        foreach (@words) {
                print "$_\t1\n";
        }
}
```

Following is the source code for the reducer. Save this code to a file called **WordCount_reducer.pl**:

```
#!/usr/bin/env perl

my $word = "";
my $sum = 0;
my $key = "";
my $value = "";

foreach (<STDIN>) {
        my @pair = split("\t", $_);
        $key = $pair[0];
        $value = $pair[1];
        if ($key eq $word) { $sum += $value; }
        else {
                if ($word ne "") { print "$word\t$sum\n"; }
                $word = $key;
                $sum = 1;
        }
}

if ($key ne "") { print "$key\t$sum\n"; }
```

To run this example for data stored in the words1 table (which includes the **id** and **sentence** columns) within the **mapreduce_db** database, run the following command:

```
mapreduce jar
/nz/export/ae/products/netezza/mapreduce/current/mapreduce-streaming.jar
                  -db mapreduce_db
                      -input 'words1' 'id' 'sentence'
                     -output 'results' 'word' 'count'
                     -mapper 'WordCount_mapper.pl'
                     -mapper_out_key_size 20
                     -mapper_out_value_size 20
                     -file <path to WordCount_mapper.pl file>
                     -reducer 'WordCount_reducer.pl'
                     -reducer_out_key_size 20
                     -reducer_out_value size 20
                     -file <path to WordCount_reducer.pl file>
```

After execution, the command creates a **results** table containing two columns: **word** and **count**, both of VARCHAR(20) type.

WordCount in Python

This section includes source code and instruction to run the WordCount example written in Python.

Following is the source code for the mapper. Save this code to a file called **WordCount_mapper.py**:

```
#!/usr/bin/env python

import sys
for line in sys.stdin:
        (key,value) = line.split('\t')
        wordList = value.split(' ')
        for word in wordList:
                print(word.strip() + "\t1")
```

Following is the source code for the reducer. Save this code to a file called **WordCount_reducer.py**:

```
#!/usr/bin/env python

import sys

word = ""
sum = 0
for line in sys.stdin:
        (key,value) = line.split('\t')
        if word == key:
                sum = sum + int(value)
        else:
                if word != "":
                        print(word + "\t" + str(sum))
                word = key
                sum = 1

if key != "": print(key + "\t" + str(sum))
```

To run this example for data stored in the words1 table (which includes the **id** and **sentence** columns) within the **mapreduce_db** database, run the following command:

```
mapreduce jar
/nz/export/ae/products/netezza/mapreduce/current/mapreduce-streaming.jar
                  -db mapreduce_db
                     -input 'words1' 'id' 'sentence'
                     -output 'results' 'word' 'count'
                     -mapper 'WordCount_mapper.py'
                     -mapper_out_key_size 20
                     -mapper_out_value_size 20
                     -file <path to WordCount_mapper.py file>
                     -reducer 'WordCount_reducer.py'
                     -reducer_out_key_size 20
                     -reducer_out_value size 20
                     -file <path to WordCount_reducer.py file>
```

After execution, the command creates a **results** table containing two columns: **word** and **count**, both of VARCHAR(20) type.

APPENDIX A
Notices and Trademarks

Notices

Trademarks

Corporation.

Other company, product or service names may be trademarks or service marks of others.

Regulatory and Compliance

Regulatory Notices

Install the NPS system in a restricted-access location. Ensure that only those trained to operate or service the equipment have physical access to it. Install each AC power outlet near the NPS rack that plugs into it, and keep it freely accessible. Provide approved circuit breakers on all power sources.

Product may be powered by redundant power sources. Disconnect ALL power sources before servicing. High leakage current. Earth connection essential before connecting supply. Courant de fuite élevé. Raccordement à la terre indispensable avant le raccordement au réseau.

Homologation Statement

This product may not be certified in your country for connection by any means whatsoever to interfaces of public telecommunications networks. Further certification may be required by law prior to making any such connection. Contact an IBM representative or reseller for any questions.

FCC - Industry Canada Statement

This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to part 15 of the FCC rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio-frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference, in which case users will be required to correct the interference at their own expense.

This Class A digital apparatus meets all requirements of the Canadian Interference-Causing Equipment Regulations.

Cet appareil numérique de la classe A respecte toutes les exigences du Règlement sur le matériel brouilleur du Canada.

CE Statement (Europe)

This product complies with the European Low Voltage Directive 73/23/EEC and EMC Directive 89/336/EEC as amended by European Directive 93/68/EEC.

Warning: This is a class A product. In a domestic environment this product may cause radio interference in which case the user may be required to take adequate measures.

VCCI Statement

　この装置は、情報処理装置等電波障害自主規制協議会　（VCCI）　の基準に基づくクラス A　情報技術装置です。この装置を家庭環境で使用すると電波妨害を引き起越すことがあります。この場合には使用者が適切な対策を講ずるう要求されることがあります。