Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 15, no. 2, February 2019, pp. 579-590
DOI: 10.23940/ijpe.19.02.p22.579590

Creative Combination of Legacy System and MapReduce in Cloud Migration

Junfeng Zhao* and Wenmeng Wang

College of Computer Science, Inner Mongolia University, Hohhot, 010021, China

Abstract

With the advent of the big data era, the response speed of traditional legacy systems is gradually unable to meet the requirements of users. Because legacy systems carry domain knowledge and critical resources, many organizations are migrating legacy systems to cloud platform so as to maximize the reuse of legacy systems as well as improve the performance of big data processing. MapReduce is recognized as an effective programming model for processing big data in parallel mode in cloud computing. Therefore, how to creatively combine parallelizable legacy code and the MapReduce model to enable legacy code to be accurately mapped into the MapReduce model is a challenging issue. We use the first type of creative computing to propose an approach for legacy code refactoring. The legacy code of big data processing is divided into several types according to the business logic, and then the corresponding refactoring rules are proposed. We use the second type of creative computing to develop a tool to support the refactoring process. The experimental results indicate that the refactoring results are correct and efficient in practical scenarios.

Keywords: legacy system; MapReduce; cloud migration; creative combination

(Submitted on November 20, 2018; Revised on December 23, 2018; Accepted on January 15, 2019)

1. Introduction

Creative computing is a newly emerging research field, which requests to combine information technology with traditional fields. Whether creative computing and new kinds of computing like cloud computing can promote each other should be taken into consideration [1]. Therefore, we can start with this problem and consider making creative computing and cloud computing complementary.

In the past few decades, many legacy systems have been running on a local server. With the advent of the big data era and the quickening pace of people's lives, the response speed of legacy systems has been unable to meet user requirements. Since legacy systems carry a large amount of domain knowledge and critical resources, they cannot be simply discarded. How to use existing technologies to maximize the reuse of legacy systems is a problem that must be solved.

Taking advantage of the scaling and flexibility features of cloud computing can provide opportunities to address the problem, which is the rapid growth of data volume in scientific computing. Therefore, many institutions are migrating their legacy software systems from a local server to the cloud. Correspondingly, the legacy system needs to make adjustments and changes according to the cloud computing programming model to support the migration to the cloud platform.

MapReduce, a programming model developed by Google, makes parallel and distributed programming more convenient [2]. A MapReduce job usually splits the input data set into independent chunks that are processed by the map tasks in a completely parallel manner. Then, the outputs of the maps input to the reduce tasks. As one of the most important cloud computing techniques, MapReduce has been a popular computing model and is effective to process very large amounts of data in parallel over a cluster of machines. When the relevant business in Partially Migrate involves the processing of big data, the related legacy code should be refactored in accordance with the MapReduce programming

* Coresponding author.
E-mail address: cszjf@imu.edu.cn, 694239632@qq.com

model. If the legacy system combines with the MapReduce model, the legacy code involving big data processing in the legacy system can be refactored according to the MapReduce model, which not only ensures the maximum reuse of the legacy system but also greatly reduces the development cycle and risk. Nevertheless, the existing relevant research mainly focuses on the migration of the overall logic, identifying candidate services in legacy systems and completing the refactoring of the architecture [3]. The code refactoring based on MapReduce has not received enough attention until now. Therefore, how to refactor the legacy code to MapReduce code, that is, how to creatively combine the legacy system with the MapReduce model, is a challenging problem facing us.

Some exploratory research is carried out on Java code refactoring in the process of cloud migration in this paper. The iterative business logic is classified into several types, and the corresponding refactoring rules are proposed. Then, a prototype tool called JMR (Java-to-MapReduce) is developed to support code refactoring.

The rest of the paper is organized as follows. Section 2 shows the basic information about creative computing and related work about code refactoring. The iterative business logic is classified into four types and the corresponding refactoring rules are proposed in Section 3. Section 4 describes the design of the prototype tool and experiment evaluation. The last section concludes the paper and presents the issues that should be solved in the future.

2. Related Work

2.1. Creative Computing

Creative computing, a new concept in the big data era, refers to one kind of computing, which is new, surprising, and useful, and creative computing aims to change and enhance people's life dramatically to make it more convenient and comfortable [1]. The following discussion will illustrate the meaning of creative computing from both creativity and computing aspects.

Creativity is a capacity to propose creative ideas and artifacts [4-5] and is essential to create fresh, surprising, and useful services. The creativity is a kernel feature of creative computing, and only if computing has creativity can it be considered creative. The creativity has been divided into psychological creativity, historical creativity, exploratory creativity, transformational creativity, and combinational creativity [6]. Psychological creativity means that creativity is novel in personal opinion. Historical creativity means that the product had never existed before. Exploratory creativity refers to the exploration in an existing conceptual space. The goal of transformational creativity is to create a new concept by changing an existing concept [7]. The creativity of creative computing is considered to belong to combinatorial creativity, because creative computing produces a new thing by combining similar ideas of predecessors [8].

At the present stage, the term computing mainly refers to software [9]. It is worthwhile to study how to improve the creativity of application software or how to develop creative software. Therefore, this paper intends to propose a practical approach for reusing legacy systems in a creative way and to develop a support tool for legacy systems in the process of cloud migration.

It is worth mentioning that compared to a term called computational creativity in artificial intelligence, creative computing requires that the computation itself is creative [10], while computational creativity uses formal computational methods to simulate human creativity [11]. Computational creativity is considered to be one of the ways to achieve creative computing. According to the properties of creativity [12], three types of creative computing were defined [6]. The first is to develop software products in a creative way. The second is to develop creative products. The third is to construct an environment that makes computing more creative. In this paper, we use the first type of creative computing to propose an approach for legacy code refactoring and use the second type of creative computing to develop a tool of code refactoring, so as to enhance the user experience of legacy systems.

2.2. Code Refactoring

Several code translators like X-to-MapReduce (X is a programming language) have been implemented, and they can translate sequential code to MapReduce code. Zhang et al. proposed a method and a translator called M2M for translating MATLAB code to MapReduce code. In the translation process, a newly built function library is used instead of standard library functions [13]. Similarly, some SQL-to-MapReduce translators have also been implemented, and they focus on SQL-like queries. Lee et al. proposed a translator called YSmart, which applies a set of rules to translate complex SQL-like queries to the minimal number of MapReduce jobs with good performance [14]. Hive, a data warehouse tool, is used to process structured data on top of Hadoop and supports the transformation from queries represented in HiveSQL into MapReduce tasks [15]. Pig [16], a high-level dataflow system, can compile the Pig Latin [17] program into Map-Reduce

jobs, and then those jobs execute on a Hadoop cluster. Li et al. implemented J2M, a Java to MapReduce translator, which can translate the Java single-layer loop without data dependency to the MapReduce code [18].

Meanwhile, some refactoring tools based upon the specific compiler have been proposed. OpenMR is an execution model based on OpenMP semantics and MapReduce parallel processor, and it can execute loops in a parallel manner by mapping loop iterations to MapReduce nodes according to the special parallel pragma in OpenMP source code [19]. In addition, HadoopDB, a hybrid system of MapReduce and DBMS technologies, was proposed [20]. The goal of HadoopDB is to transform a single-node DBMS into a parallel data analysis platform in the cloud [21]. SCOPE, a distributed computation system, was used for massive data analysis on large clusters and combines benefits of parallel databases with the MapReduce model to achieve scalability and high performance [22-23].

Above all, the research about Java code refactoring in cloud migration is insufficient. Currently, only J2M implemented the refactoring from a simple single-layer for loop to MapReduce code by extending the two phases of the compiler to four phases. Our research aims to improve the efficiency and effectiveness for legacy code refactoring in the way of creative computing. In order to combine legacy systems with MapReduce and realize Java code refactoring in cloud migration efficiently, a new general approach that can consider more scenarios and obtain a more practical effect in code refactoring is proposed.

3. Data Processing Types and Refactoring Rules

This paper aims to combine legacy systems with the MapReduce model in a creative way to provide a new refactoring approach from Java sequential loops to the cloud code. Firstly, the target code template based on the MapReduce model is built up. Then, what the refactoring needs to resolve is to write the specific business operations into the code template according to the refactoring rules.

3.1. MapReduce Code Template

In the MapReduce model, a computation task is represented by a MapReduce job. By analyzing the process of the MapReduce job, it can be extracted into the following main steps: (1) execute the same operation for all data; (2) perform the mapping operation; (3) write the intermediate result <key, value> to the local hard disk; (4) perform the reducing operation; (5) write the final result to the distributed file system. Therefore, the MapReduce task consists of two main phases, which are map and reduce. In addition, the optional combine function can be used to combine the outputs of the map function so as to reduce the amount of data transferred to the reduce function, whose function is same as the reduce.

In the premise of maximum reuse to legacy code, the refactoring needs to decompose the business logic of legacy code into map and reduce functions. The target code template of the MapReduce model is proposed as shown in Figure 1. In Figure 1, KEYIN1 and VALUEIN1 are the type of Map's input key and input value, KEYOUT1 and VALUEOUT1 are the types of Map's output key and output value, and KEYOUT2 and VALUEOUT2 are the types of Reduce's output key and output value, respectively. The italicized parts of the template will be supplemented during the refactoring process. During the refactoring process, the mapping between Java data types and Hadoop data types should also be considered, which is shown in Table 1.

3.2. Data Processing Types and Rules

It is unrealistic to provide a set of common refactoring rules for all parallelizable legacy code. In order to implement code refactoring more accurately and automatically, the legacy code that needs to be refactored should be divided into different data processing types according to their business logic, and then the corresponding refactoring rules for each type is defined. The approach people use for legacy code refactoring is new, which is the first type of creative computing.

Miner and Shook proposed six MapReduce design patterns, which introduce the issues that are suitable to be solved by the MapReduce framework [24]. The work provides some reference for the division of data processing types in our paper. It is a combinatorial creativity that generates a new approach by referring to related research of predecessors. Combining the characteristics of business logic in the legacy code and the applicable scenarios of MapReduce presented by Miner and Shook, four data processing types are extracted and the corresponding refactoring rules are proposed in this paper.

```
public static class Map extends Mapper<KEYIN1, VALUEIN1,
KEYOUT1, VALUEOUT1>
{
    Global variable declaration;
    public void map(KEYIN1 k1, VALUEIN1 v1, Context context)
    throws IOException, InterruptedException
    {
        Local variable declaration;
        Record type conversion;
        Mapping operations;
        context.write(KEYOUT1 k2,VALUEOUT1 v2);
    }
}
public static class Reduce extends Reducer<KEYOUT1, VALUEOUT1,
KEYOUT2, VALUEOUT2>
{
    Global variable declaration;
    public void reduce(KEYOUT1 k2, Iterable<VALUEOUT1> values,
    Context context) throws IOException, InterruptedException
    {
        Local variable declaration;
        for(VALUEOUT1 val : values)
        {
            Reduction operations;
        }
        context.write(KEYOUT2 k3, VALUEOUT2 v3);
    }
}
```

Figure 1. MapReduce code template

Table 1. The mapping between Java and Hadoop data types

Java basic types	Hadoop basic types
Int	IntWritable
Long	LongWritable
Boolean	BooleanWritable
Float	FloatWritable
Double	DoubleWritable
String	Text

3.2.1. Data Analysis

This type of data analysis focuses on the analysis and extraction of the original data set information. Resolving text and transforming data format are examples of this type of application. The business of this type does not need a reduce process but rather involves processing each record separately. There are two following features of data analysis:

- Only map operation is needed. The mapper deals with input records one by one and then outputs the required information of records or records with the required format.
- The result of the map operation should be written in HDFS file.

The refactoring rules of data analysis are shown in Figure 2. As with the previous refactoring of three data processing types, the refactoring of data analysis begins with a type judgment of the first statement in the tagged code segment. (1) If the statement is a variable declaration statement, it is refactored to the local variable declaration part of the Map class, and then the next statement is analyzed until the last statement of the segment. (2) If it is a read statement, it is refactored into the record type conversion part of the template. (3) If it is the output statement, it is refactored into the context.write() of the map function. (4) If it is an expression statement, it is refactored into the mapping operation. At this point, the read statement, output statement, and expression statement are all needed for a judgment, that is, whether there exists a variable used in this statement that is not declared in the loop segment, and if so, the variable information will be looked from the legacy code outside the tagged code, and then the variable declaration will be written to the global variable declaration part of the Map class. After this step, the first statement has completed the mapping to the target code, and the next step is to analyze the next statement until the entire tagged code segment is mapped to the target code.

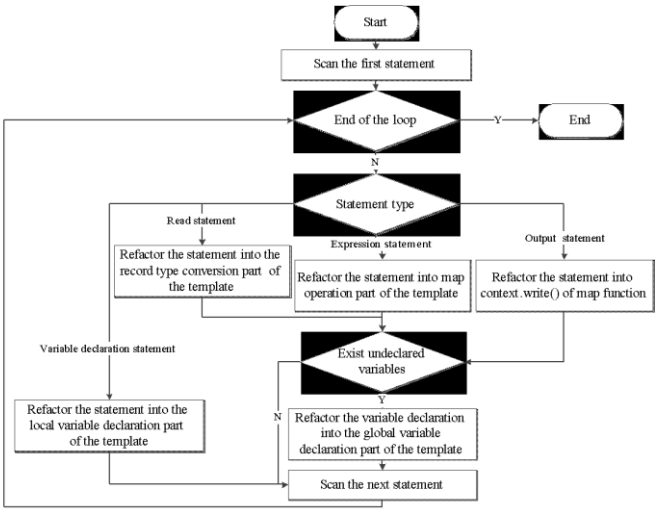

Figure 2. Refactoring rules of data analysis

3.2.2. Data Filtering

This type of data filtering aims at finding a specific subset from a large record set according to some conditions. The records with specific information will be retained, and the other records that do not meet the conditions will be filtered out. Finding out all the prime numbers or the top *n* values from a large data set is a typical application scenario of this data processing type. There are three following features of data filtering:

- When every record in the input data set is a data unit, mapper deals with input records one by one and then filters out the records that meet the conditions.
- When the records in the input data set are more complex, mapper maps the required information of records to the key-value pairs and passes them to reducer to filter out the final records subset that meets the conditions.
- If only the map operation is needed, the result of the map operation should be written in Hadoop Distributed File System (HDFS) file. If the reduce operation is also needed, the final result of reducer should be written in HDFS file.

The refactoring rules of data filtering are shown in Figure 3. When refactoring this type of legacy code into the MapReduce model, we determine the type of the statement one by one to implement the mapping of the source legacy code to the target code. (1) If the statement is a variable declaration statement, it is refactored to the local variable declaration part of the template, and then the next statement is scanned and analyzed until the last statement of the segment. (2) If the statement is a read statement, it is refactored into the record type conversion part of the template according to the Table 1. (3) If the statement is the output statement, it is refactored into the context.write(). (4) If the statement is an expression statement (denoted as α), it needs to be judged whether it is a comparison statement. If not, α is refactored into the mapping operation. If α is a comparison statement, it needs to be determined whether its next statement (denoted as β) is an output statement. If β is an output statement, α is refactored into the mapping operation. On the contrary, α is refactored into the reduce operation part. Next, the output statement, read statement, and expression statement are all needed for judgment, that is, whether there is a variable in the statement currently parsed that is not declared in the tagged code segment. If there exists an undeclared variable, the variable information is found from the legacy code outside the loop and the variable declaration is written into the global variable declaration part of the template. Finally, we continue to scan and analyze the next statement until the end of the tagged code segment.

3.2.3. Data Reorganization

This type of data reorganization pays attention to reorganizing the disordered data in the original data set into relatively ordered and well-organized data according to a certain standard. For example, dividing the telephone number by the area code is an application scenario of this type. For this data processing type, there are three following features:

- Mapper extracts the output key through the analysis of the record, taking the whole or part of a record as the output value.
- Reducer simply outputs the key-value pairs sent from the mapper side. It does not do anything specific because the business logic has been automatically completed in the shuffle process of MapReduce.

- The final result of reduce operation should be written in HDFS file.

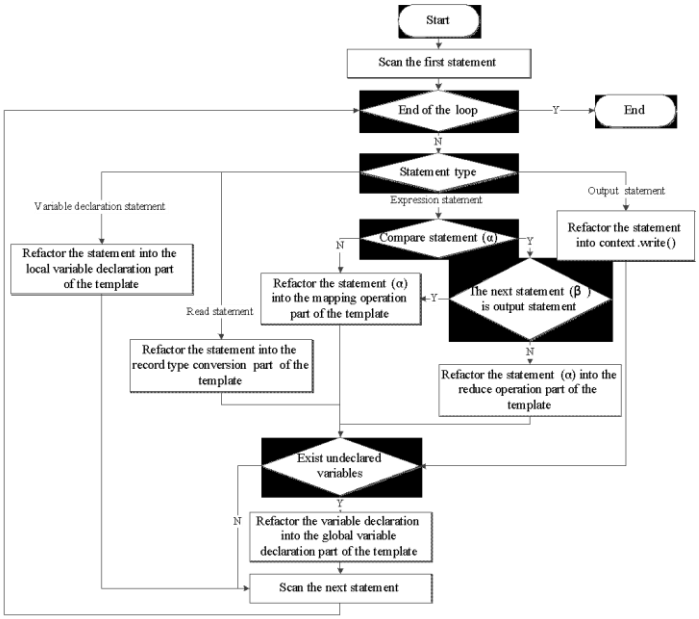

Figure 3. Refactoring rules of data filtering

The refactoring rules of data reorganization are shown in Figure 4. The type of the statement is determined one by one to implement the mapping of the legacy code to the target MapReduce code during the refactoring process. The analysis starts with the first statement. (1) If the statement belongs to the variable declaration statement, it is refactored to the local variable declaration part of the template, and then the next statement is analyzed until the end of the legacy code segment. (2) If the statement belongs to the read statement, it is refactored into the record type conversion part of the template. (3) If the statement belongs to the output statement, it is refactored into the context.write(). (4) If the statement belongs to the expression statement, we continue to determine whether the statement does the operation of analyzing and extracting the record information. If the answer is yes, it is refactored into the mapping operation part of the template. Otherwise, the statement is refactored into the reduce operation part. Next is a judgment that the read statement, the output statement, and the expression statement have to do, that is, whether there exist variables that are not declared in the tagged code segment in this statement. If they exist, the variable information is found in the legacy code, except the tagged segment, and the variable declaration is written into the global variable declaration part. Then, the next statement is scanned and analyzed until the end of the loop segment.

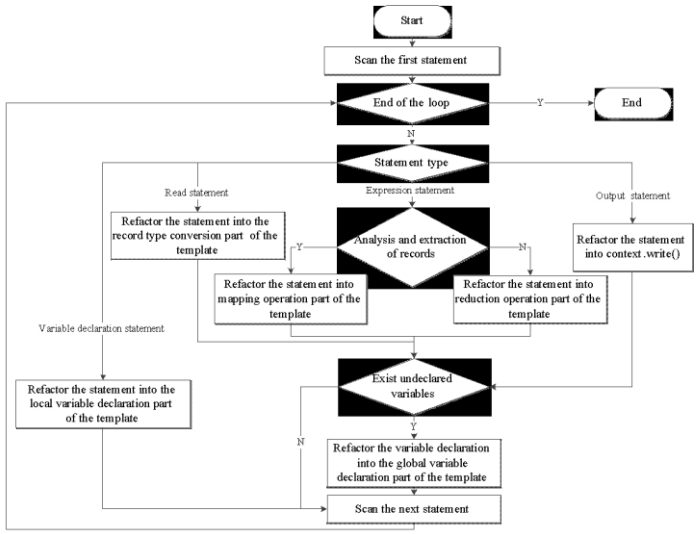

Figure 4. Refactoring rules of data reorganization

3.2.4. Data Statistics

This type of data statistics focuses on grouping similar data together and performing subsequent analysis operations like statistical calculations and simple counting. This is a general application scenario, grouping records based on keys and computing the aggregated value for each group. WordCount, a canonical example of this data processing type, realizes word frequency statistics for large documents. The business logic of data statistics in MapReduce can be expressed as follows:

$$x_n = f_m(a_n) \tag{1}$$

$$y = f_r(x_1, x_2, x_3, \cdots, x_n) \tag{2}$$

Here, f_m is a generic data statistics map function that executes mapping operations on every input value a_n (n stands for 1,2,3,⋯,n) to get the output value x_n of the map function, and f_r is a generic data statistics reduce function that will be used on a list of values $(x_1, x_2, x_3, \cdots, x_n)$ to get an output value y. For this data processing type, there are four following features:

- Mapper groups similar data together and outputs all the fields that need to be grouped as keys and all related fields x_n executed by f_m as values.
- If the order of the values can be changed arbitrarily and the calculations can be grouped randomly, that is, if the function f_r satisfies the associative law and commutative law, it will be a good choice to consider using the combine function.
- Reducer accepts a list of values $(x_1, x_2, x_3, \cdots, x_n)$ and executes f_r to get the result value associated with the given input key.
- The final result of reduce operation should be written in HDFS file.

The refactoring rules of data statistics are shown in Figure 5. In the process of implementing the mapping of legacy code segment to the target code, it is necessary to judge the type of the statement one by one. (1) If it is a variable declaration statement, it is refactored to the local variable declaration part of the target code template, and then the next statement type is analyzed until the last statement of the segment. (2) If it is a read statement, it is refactored into the record type conversion part. (3) If it is an output statement, it is refactored into the context.write(). (4) If it is an expression statement, the statement needs to be judged whether it is an assignment statement. If not, it is refactored into the mapping operation part of the template. Conversely, if the statement is an assignment statement, it needs to be further judged whether the assignment operator is a compound assignment operator. If it is negative, the statement needs to be judged whether there are the same variables on both sides of the assignment operator. If there are no same variables on both sides of the assignment operator, this statement will be refactored into the mapping operation part. If there exist the same variables on both sides of the assignment operator, then this statement will be refactored into the reduction operation part. If the assignment operator in the statement is a compound assignment operator, then it must be determined whether the right side of the operator is an expression, and if so, the right side of the operator is refactored into the mapping operation. The left side along with the output of f_m are refactored into the reduction operation part. Otherwise, the statement is refactored into the reduction operation part of the template by f_r. After the refactoring of the map and reduction of parts are completed, a judgment still needs to be performed whether f_r meets the associative and commutative laws. If f_r is satisfied, the combine function can be chosen. Next, there is a judgment involving the read statement, the output statement, and the expression statement, that is, whether there exist variables in the statement that are not declared in the tagged loop. If they exist, the variable information is found in the legacy code, except for the tagged segment, and the variable declaration is written into the global variable declaration part of the template. At this point, the first statement is analyzed and the next statement will be analyzed until the end of the tagged segment.

4. The Refactoring Tool and Evaluation

This paper uses the second type of creative computing, the development of a creative computing product, to develop a semi-automated refactoring tool JMR. The aim of the JMR is to improve creativity and support refactoring. During the process of refactoring, developers need to choose the data processing type, and then the legacy code will be refactored according to the related refactoring rules implemented by the tool. The target code will be encapsulated as a service, so the original

application can invoke it. In order to verify the validation of the refactoring rules and the efficiency of the target code, two experiments are carried out.

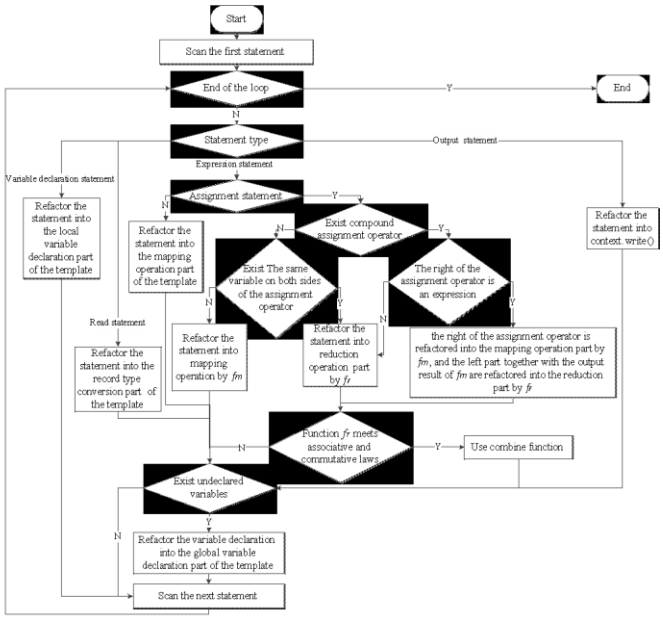

Figure 5. Refactoring rules of data statistics

4.1. Working Process of JMR

Although there may exist many loops in the legacy code, not all loops can be parallelized. In another work of ours, an approach is proposed to identify the parallelizable loops and mark them with the special tag [25]. Based on the parallel tag, JMR can locate the specific loops that need to be refactored in the legacy code. The work process of JMR is shown in Figure 6.

The structure of JMR consists of the following three main components:

- The first component is the recognizer, used for detecting the loop segment with a parallel tag. The recognizer finds the "#MR parallel#" tag by statically scanning the legacy code, and then it sends the loop segment with a parallel tag to the ANother Tool for Language Recognition (ANTLR).
- ANTLR is an existing powerful parser generator that can be used as the second component of JMR. It can convert the parallelizable loop segment into Abstract Syntax Tree (AST) and send it to the refactorer. AST is a tree-like representation of the abstract syntax structure of the source code, and each node in the tree represents a grammatical structure.
- The third component is the refactorer, used for producing the target code with the MapReduce model. Refactorer obtains the statement information by analyzing the AST and then refactors the original code by taking advantage of the proposed refactoring rules. On the basis of the MapReduce code template, the specific statements will be supplemented and the target code will be produced.

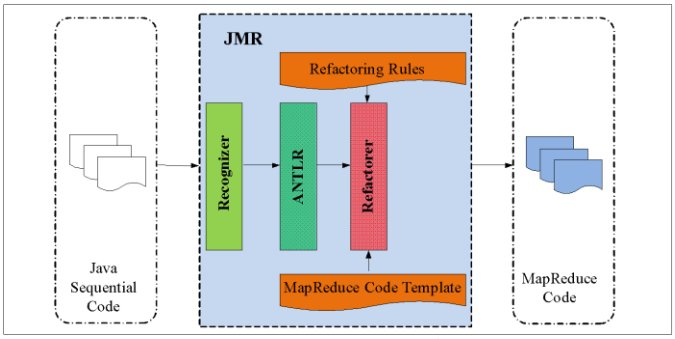

Figure 6. Working process of JMR

4.2. Experiment Evaluation

To verify the availability of the refactoring approach, two experiments were carried out. The correctness of the refactoring approach can be verified, and the execution efficiency of the target code can be validated.

4.2.1. Refactoring Experiments of JMR

The first experiment uses the refactoring rules of data statistics to complete refactoring. The sequential tagged code is shown in Figure 7. The original legacy system is a financial information management system. One of the business functions is to calculate the tax amount. This experiment refactors a simple loop segment in this functional code. The target MapReduce code is shown in Figure 8.

Java sequential code

```
#MR parallel#
1   while((line=br.readLine())!=null)
2   {
3       long i= Long.parseLong(line);
4       sum+=0.1*i-105;
5   }
```

Figure 7. Tagged code segment of data statistics experiment

MapReduce code

```
1   public static class Map extends Mapper< LongWritable, Text, LongWritable, LongWritable >
2   {
3       String line;
4       long sum=0;
5       public void map(LongWritable key, Text value, Context context)  throws IOException,InterruptedException
6       {
7           line = value.toString();
8           long i=Long.parseLong(line);
9           sum=0.1*i-105;
10          context.write(new LongWritable(1),  new LongWritable(sum));
11      }
12  }
13  public static class Reduce extends Reducer< LongWritable, LongWritable, NullWritable, LongWritable >
14  {
15      String line;
16      long sum=0;
17      public void reduce(LongWritable key, Iterable< LongWritable > values, Context context) throws IOException,InterruptedException
18      {
19          for(LongWritable val:values)
20          {
21              sum += val.get();
22          }
23          context.write(NullWritable,get(), new IntWritable(sum));
24      }
25  }
```

Figure 8. MapReduce code of data statistics experiment

In the process of refactoring, the first statement of the tagged code is analyzed to be a read statement firstly, so it is refactored into the record format conversion part of the template, that is, the seventh line of Figure 8. The variable line is not declared in the loop, so it will be declared in the global variable declaration part of the template, corresponding to the third line and fifteenth line of Figure 8. Since the refactoring process only parses the statements, the brackets in the code segment are skipped directly during analysis. Next, the third line of the tagged code, which is a variable declaration statement, is refactored into the mapping operation section of the map function, that is the eighth line of Figure 8. The fourth line of the tagged code is found to be an assignment statement, and there is a compound assignment operator, and the right side of the assignment operator is an expression. Thus, the right side of the compound assignment operator is refactored into the mapping operation part of the template, that is, the ninth line of Figure 8. The output of the map is written in the tenth line of Figure 8. The compound assignment operator is refactored along with the left part and the output value of the map into the reduction operation part, which is the 21st line of Figure 8. Then, it is analyzed that the f_r is an accumulation function, which satisfies the associative law and commutative law, so the target code can choose to use the combine function. It is analyzed that there is still a variable sum in the statement that is not declared in the loop, so its declaration is refactored into the fourth and sixth lines of Figure 8. Sum is finally used as the output value of the reduce function. At this point, the refactoring of the tagged code loop to the target MapReduce code is completed.

The second experiment uses the refactoring rules of data filtering to complete refactoring. The sequential tagged code is shown in Figure 9. The legacy system is a weather information management system. One of the sub-modules provides users with weather information in previous years. This experiment refactors a loop segment in the sub-module that looks for the highest temperature. The highest temperature of every region in a year can be found by analyzing the large meteorological data sets that come from the China National Meteorological Information Center. The meteorological data is stored in line-oriented ASCII format, where each row is a record. The target MapReduce code is shown in Figure 10.

Java sequential code

```
    #MR parallel#
1   while ((line = br.readLine()) != null)
2   {
3       data = line.substring(4, 9);
4       int airTemperature;
5       if (line.charAt(87) == '+')
6           airTemperature = Integer.parseInt(line.substring(88, 92));
7       else
8           airTemperature = Integer.parseInt(line.substring(87, 92));
9       String quality = line.substring(92, 93);
10      if (airTemperature != MISSING && quality.matches("[01459]"))
11          maxValue = Math.max(maxValue, airTemperature);
12  }
```

Figure 9. Tagged code segment of data filtering experiment

MapReduce code

```
1   public static class Map extends Mapper<LongWritable, Text, Text, IntWritable>
2   {
3       String line;
4       String data;
5       int MISSING = 9999;
6       int maxValue = Integer.MIN_VALUE;
7       public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException
8       {
9           line = value.toString();
10          data = line.substring(4, 9);
11          int airTemperature;
12          String quality = line.substring(92, 93);
13          if (line.charAt(87) == '+')
14              airTemperature = Integer.parseInt(line.substring(88, 92));
15          else
16              airTemperature = Integer.parseInt(line.substring(87, 92));
17          if (airTemperature != MISSING && quality.matches("[01459]"))
18           context.write(new Text(data), new IntWritable(airTemperature));
19       }
20  }
21  public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable>
22  {
23      String line;
24      String data;
25      int MISSING = 9999;
26      int maxValue = Integer.MIN_VALUE;
27      public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException
28      {
29          for (IntWritable val : values)
30              maxValue = Math.max(maxValue, val.get());
31          context.write(key, new IntWritable(maxValue));
32       }
33  }
```

Figure 10. MapReduce code of data filtering experiment

During the refactoring process, the statements of the tagged code segment are analyzed one by one. The first statement type is determined to be a read statement, and then it is refactored into the ninth line of Figure 10. Next, the variable line is analyzed and is not declared within the loop segment, so it is declared as the global variable in the 3rd and 23rd lines of Figure 10. Then, the third line of the tagged segment is known to be an expression statement through analyzing, so it is refactored into the tenth line of Figure 10. The variable data not declared in the loop is refactored as global variables into the 4th and 24th lines of Figure 10. The fourth statement is known as a variable declaration statement that maps directly to the eleventh line of Figure 10. Therefore, according to refactoring rules of data filtering and the target code template, the 3rd, 5th, 6th, 7th, 8th, and 10th statements of the tagged loop are refactored into the mapping operation part of Figure 10, the 4th and 9th statements correspond to the local variable declaration part, the 11th statement is refactored into the reduction

operation part, and the variables not declared in the loop are refactored into the global variable declaration part of Figure 10. The final result is written into HDFS through the context.write().

4.2.2. Validity of the Refactoring Approach

The sequential code and the corresponding refactored code are deployed on the cloud platform. The two data sets with 10GB size and 20GB size are processed in the experiments respectively. By observing the experiment results, it can be found that the execution result of the legacy code is the same as the target MapReduce code, which illustrates the validity of the refactoring approach.

4.2.3. Performance Comparison

The experiments were carried out on a cluster with eight nodes. Each node has 16GB main memory and uses AMD Opteron(TM) Processor 2376 CPU (four cores with a clock frequency of 2.3GHz). A node acts as the master node, which is used to run JobTracker. The other seven nodes are slave nodes, which are used to run TaskTracker.

For the above two experiments, the data sets with different sizes were used to compare the execution efficiency between the legacy code and target code. The result of the first experiment is shown in Figure 11. The result of the second experiment is shown in Figure 12. The experiment results are consistent with our expectations, that is, the process time of the target MapReduce code is less than that of the legacy code when the size of the data set reaches a certain volume. The sequential Java code is more efficient than the MapReduce code when dealing with small data sets. When the calculation size is large enough, the MapReduce code is faster than the legacy code. This result is caused by the relationship between intercommunication cost and parallelization advantage of the MapReduce model. When the size of the data set is small, the intercommunication cost is bigger than the parallelization benefit, so the efficiency of the sequential code exceeds that of the refactored code. On the contrary, the refactored code will surpass the sequential code when the size of the data set is large enough.

Based on the experimental results, the conclusion can be obtained that the creative combination of the legacy system and MapReduce model is feasible, and the new refactoring approach proposed in this paper is available. After the programmer selects the data processing type for the tagged code, the corresponding refactoring rules can refactor the source code to the target code more accurately. The execution result of the target code is consistent with the legacy sequential code and more efficient in massive data processing.

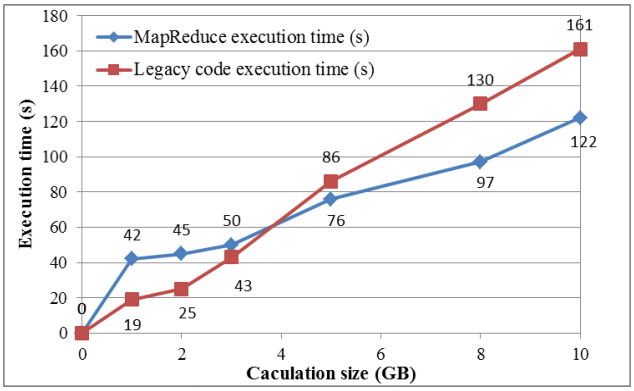

Figure 11. Performance comparison of data statistics example

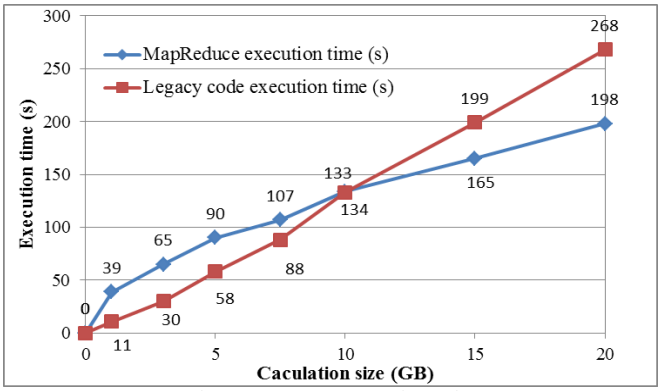

Figure 12. Performance comparison of data filtering example

5. Conclusions

This work takes combinational creativity, legacy systems, and big data processing as the foundation, combines legacy code with the MapReduce model and makes the reuse of legacy systems more creative. The research proposes a new refactoring approach from sequential Java code to MapReduce code, where the business logic is divided into different types and the refactoring rules corresponding to every type are proposed. Then, a supporting tool JMR, a creative computing product, is developed. By using the tool JMR, sequential Java code can be refactored into the MapReduce model. The legacy system benefits from the creative approach and tools, which help save the cost and time of software migration. The experiments demonstrate that the target code has better performance than the sequential code when it comes to larger data sets.

Meanwhile, the current approach is not perfect enough. The classification of data processing types and refactoring rules are not yet complete. Therefore, further work needs to be done in the future to address these deficiencies. In addition, the tool JMR needs to be improved to support more automatic and accurate refactoring.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 61462066), Inner Mongolia Science and Technology Innovation Team of Cloud Computing and Software Engineering, and Inner Mongolia Application Technology Research and Development Funding Project "Mutual Creation Service Platform Research and Development Based on Service Optimizing and Operation Integrating".

References

1. L. Zhang and H. Yang, "Definition, Research Scope and Challenges of Creative Computing," in *Proceedings of 2013 19th International Conference on Automation and Computing*: *Future Energy and Automation*, 2013
2. J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, Vol. 51, No. 1, pp. 107-113, January 2008
3. J. F. Zhao and J. T. Zhou, "Strategies and Methods for Cloud Migration," *International Journal of Automation and Computing*, Vol. 11, No. 2, pp. 143-152, April 2014
4. R. Lustig, "The Creative Mind: Myths and Mechanisms," Artificial Intelligence, 1995
5. G. A. Wiggins, "A Preliminary Framework for Description, Analysis and Comparison of Creative Systems," *Knowledge-based Systems*, Vol. 19, No. 7, pp. 449-458, 2006
6. A. Hugill and H. Yang, "The Creative Turn: New Challenges for Computing," *International Journal of Creative Computing*, Vol. 1, No. 1, pp. 4-19, 2013
7. R. J. Sternberg, "Handbook of Creativity," Cambridge University Press, 1999
8. J. Sawle, F. Raczinski, and H. Yang, "A Framework for Creativity in Search Results," in *Proceedings of the Third International Conference on Creative Content Technologies*, Rome, Italy, 2011
9. L. Zhang and H. Yang, "Knowledge Discovery in Creative Computing for Creative Tasks," in *Proceedings of the 1st Conference on Creativity in Intelligent Technologies and Data Science*, Volgograd, Russia, 2015
10. A. Hugill, H. Yang, F. Raczinski, and J. Sawle, "The Pataphysics of Creativity: Developing a Tool for Creative Search," *Digital Creativity*, Vol. 24, No. 3, 2013
11. T. Colburn and G. Shute, "Abstraction in Computer Science," *Minds and Machines*, Vol. 17, No. 2, pp.169-184, 2007
12. R. E. Mayer, "Fifty Years of Creativity Research: In Handbook of Creativity," Cambridge University Press, pp. 449-460, 1999
13. J. B. Zhang, D. Xiang, T. R. Li, and Y. Pan, "M2M: A Simple Matlab-to-MapReduce Translator for Cloud Computing," *Tsinghua Science and Technology*, Vol. 18, No. 1, pp. 1-9, 2013
14. R. Lee, T. Luo, Y. Huai, F. S. Wang, Y. Q. He, and X. D. Zhang, "YSmart: Yet Another SQL-to-MapReduce Translator," in *Proceedings of 2011 31st International Conference on Distributed Computing Systems* (*ICDCS*), pp. 25-36, Minneapolis, MN, USA, 2011
15. A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, et al., "Hive: A Warehousing Solution over a Map-Reduce Framework," *Proceedings of the VLDB Endowment*, Vol. 2, No. 2, pp.1626-1629, 2009
16. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, et al., "Building a High-Level Dataflow System on Top of Map-Reduce: The Pig Experience," *Proceedings of the VLDB Endowment*, Vol. 2, No. 2, pp. 1414-1425, 2009
17. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Piglatin: A Not-so-foreign Language for Data Processing," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pp. 1099-1110, 2008
18. B. Li, J. B. Zhang, N. Yu, and Y. Pan, "J2M: A Java to MapReduce Translator for Cloud Computing," *Journal of Supercomputing*, Vol. 72, No. 5, pp. 1928-1945, 2016
19. R. Wottrich, R. Azevedo, and G. Araujo, "Cloud-based OpenMP Parallelization using a MapReduce Runtime," in *Proceedings of the 26th International Symposium on Computer Architecture and High Performance Computing*, pp. 334-341, SBAC-PAD, Paris, France, IEEE Computer Society, 2014
20. A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads," *Proceedings of the VLDB Endowment*, Vol. 2, No.1, pp. 922-933, 2009
21. K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and E. Paulson, "Efficient Processing of Data Warehousing Queries in a Split Execution Environment," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1165-1176, 2011
22. R. Chaiken, B. Jenkins, Per-Ake. Larson, J. R. Zhou, et al. "SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets," *Proceedings of the VLDB Endowment*, Vol. 1, No. 2, pp. 1265-1276, August 2008
23. J. R. Zhou, N. Bruno, M. C. Wu, Per-Ake. Larson, R. Chaiken, and D. Shakib, "SCOPE: Parallel Databases Meet MapReduce," *VLDB Journal*, Vol. 21, No. 5, pp. 611-636, 2012
24. Miner and A. Sbook, "MapReduce Design Patterns," O'Reilly Media, pp. 256, 2012
25. J. F. Zhao and Z. M. Zhao, "Distributed Parallelizability Analysis of Legacy Code," in *Proceedings of the 16th IEEE International Symposium on Parallel and Distributed Processing with Applications*, 2018