MR Language Reference Manual

Siyang Dai (sd2694)
Jinxiong Tan (jt2649)
Zhi Zhang (zz2219)
Zeyang Yu (zy2156)
Shuai Yuan (sy2420)

1. Introduction

MapReduce is a programming paradigm to support distributed computing on large data sets on clusters of computer. The paradigm is inspired by the map and reduce functions universally used in functional programming. The MR programming language is designed specifically for MapReduce.

1.1 Concept of M1.1.1 List Processing

Essentially, a MapReduce program convert lists of input data elements into lists of output data elements. The transformation is done by two phases: map and reduce.

1.1.2 Map

The first phase of a MapReduce program is called mapping. A list of data pairs are provided, one at a time, to a function called the Mapper, which transforms each input element individually to an output data element. Logically, a map function is defined as the following form:

$$Map(k_1,v_1) \rightarrow list(k_2,v_2)$$

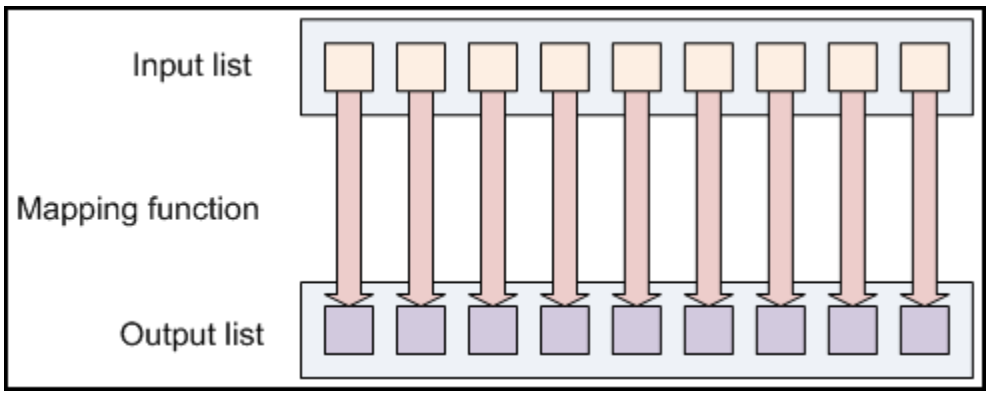

Figure 1 Map[1]

After that, all pairs with the same key from all lists generated by map function will be grouped together, thus creating one group for each one of the different generated keys. The groups will be the input of the next phase.

1.1.3 Reduce

Reducing allows you aggregate values together. A reduce function receives a list of values with the same key. It then combines these values together. Logically, a reduce function is defined as the following form:

$$Reduce(k_2, list\ (v_2)) \rightarrow (k_3,v_3)$$

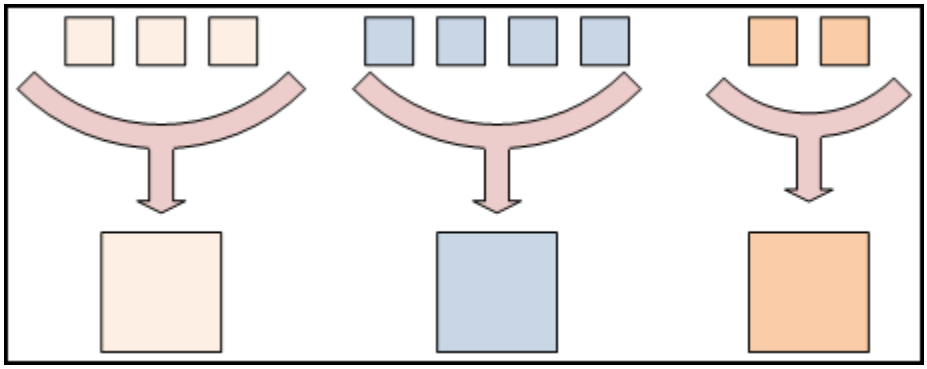

Figure 2 Reduce

[1]Figure 1,2,3 are from Hadoop Tutorial on Yahoo Developer Network

As a result, we get a pair of (k,v) for each distinct key generated by map function.

1.2 Data-flow of MapReduce

Combining map and reduce, we can have the following overview for the data-flow of a MapReduce program on a cluster consisting of three nodes:

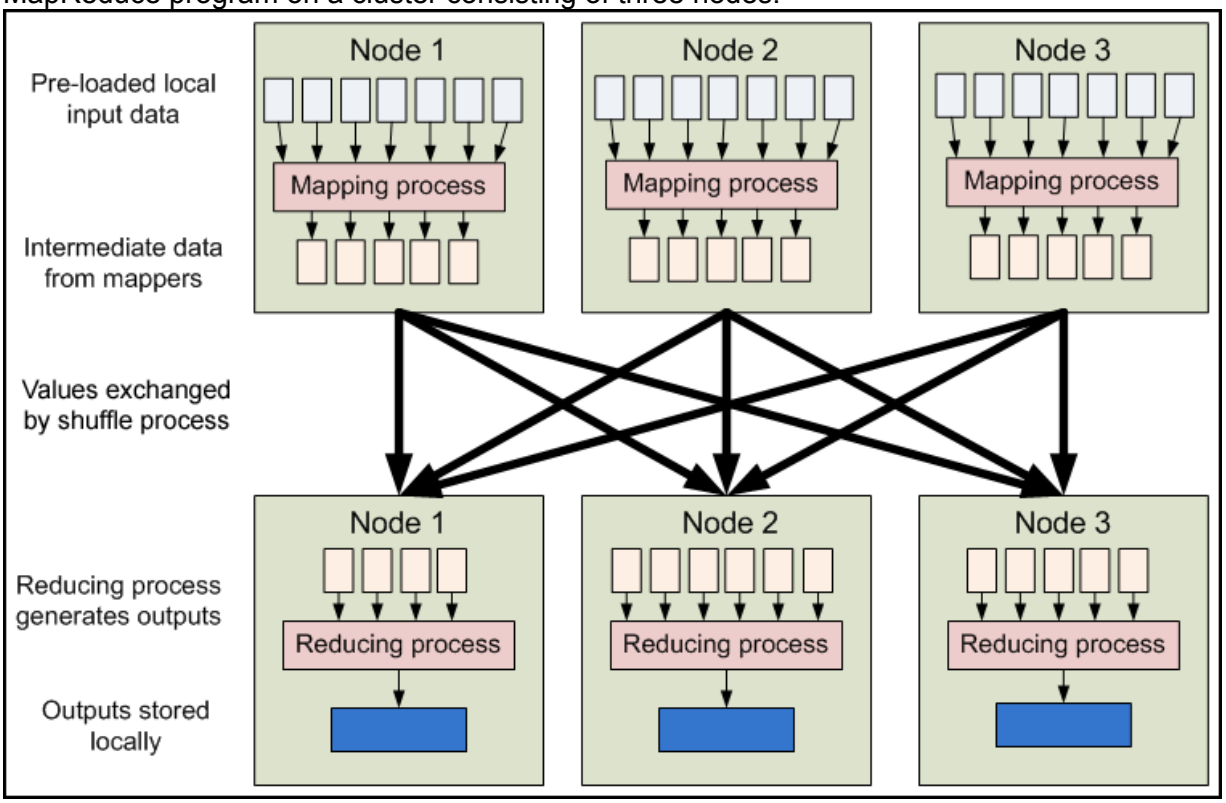

Figure 3 MapReduce

1.3 The MR Programming Language

MR is designed to support MapReduce paradigm. It hides the details of MapReduce framework from the programmers. All the programmers need to do is to define a map function and a reduce function. The program will be run according to the data-flow of MapReduce.

1.4 Input and Output of MR Program

An MR program takes two arguments from command line. The first one is the input directory. And the second one is the output directory.

1.4.1 Input

All files under the input directory are used as input files. MR treats each line of each input file as a separate record, and performs no parsing. It feeds the map function with the byte offset of the line as key and the line content as value. Therefore, for map function, k_1 is always an integer and v_1 is always one line of text.

1.4.2 Output

The output directory must not exist before the MR program runs. The MR program will create one automatically. The output of reduce function will be written to files under the output directory

in form “key \t value” per line.

2. Lexical Elements

2.1 Tokens

There are five kinds of tokens in MR, i.e., literals, keywords, identifiers, operators and other separators. Blanks, newlines and comments are ignored during lexical analysis except that they separate tokens.

2.2 Constants

2.2.1 Text Constant

Text constant is a string containing a sequence of characters surrounded by a pair of double quotes, i.e. “...”. For example, “hello world!” is a Text constant. Identical Text constants are the same. All Text literal are immutable.

One thing to note is that, in MR, there is no character type. Even a single character is Text constant type which can be regarded as an extended character set.

2.2.2 Int Constant

A Int constant refers to a integer consisting of a sequence of digits. It supports signed and unsigned integers. Int constant cannot start with a 0 (digit zero). All integers are default to be decimal (base 10). For example, -15 and 2012 are valid Int constant.

2.2.3 Double Constant

In MR, a double constant refers to a floating constant which consists a integer part, a decimal point and a fraction part. In addition, it supports an ‘e’ followed by an optionally signed integer exponent. The integer part and fraction part can be one digit or a sequence of digits. Either of them can be missing, but not both. Also either the decimal point or the e and the exponent (not both) may be missing. The following are valid Double constants: 1. or 0.5e15 or .3e+3 or .2 or 1e5

2.3 Keywords

The following words are reserved as the keywords which cannot be used otherwise.

Text	**Int**	**Double**	**Boolean**	**List**
def	**if**	**else**	**foreach**	**emit**
and	**or**	**Mapper**	**Reducer**	**split**
by	**true**	**false**		

2.4 Identifiers

Identifiers are used for naming variables, parameters and functions. Identifier consists of a sequence of letters, digits and the underscore _ , but it must start with a letter. Identifier should not be the keywords listed above. It is case-sensitive.

2.5 Operators

An operator is a special token that performs an operation, such as addition or subtraction, on

either one or two operands. More details will be covered in later section.

2.6 Separators

A separator separates tokens. Other separators (Blanks, newlines and comments) are ignored during lexical analysis except the following:

( ) < > { } ;

2.7 Comments

// is used to indicate the rest of the line is comment (C++/Java style comment)

3. Data Types

3.1 Int

The 64-bit Int data type can hold integer values in the range of −9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

3.2 Double

The Double type covers a range from 4.94065645841246544e-324d to 1.79769313486231570e+308d (positive or negative).

3.3 Boolean

A variable of Boolean may take on the values true and false only.

3.4 List

It is used as List<T>, i.e. List<Int> represents a list of Int values. It has unlimited size.

3.5 Conversions

When a value of Double type is converted to Int type, the fractional part is discarded. When a value of integral type is converted to Double type, and the value is not exactly representable, then the result may be either the next higher or next lower representable value.

No other conversion is allowed.

4. Program Structure

A MR program must exist entirely within a single source file (with a ".mr" extension). By convention, a typical MR source file must include three parts: configuration, mapper and reducer. That is,

program -> configuration-declaration mapper-definition reducer-definition

Here is an example program:

```
//wordcount.mr
#JobName = "WordCount"

//map function definition
def wordcount_map <(Int, Text) -> (Text , Int)> (offset, line): Mapper
{
        List<Text> words;
        words = split line by " ";
```

```
        foreach Text word in words {
                emit(word,  1);
        }
}
//reduce function definition
def wordcount_reduce <(Text , Int) -> (Text, Int)> (word, counts): Reducer
{
        Int total;
        total = 0;
        foreach Int count in counts {
                total = total + count;
        }
        emit(word, total);
}
```

4.1 Configuration

configuration-declaration -> **#** *configuration-attribute* **= Text**$_{\textbf{const}}$**;**
configuration-attribute -> **JobName**

In this field, users can specify attribute **JobName** using a Text constant. (The support of specifying the number of Mapper/Reducer process will be extended in the future.)

4.2 Mapper/Reducer Definition

mapper-definition -> **def identifier** *mapping-relation parameters* **:** *function-type block*
reducer-definition -> **def identifier** *mapping-relation parameters***:** *function-type block*

The keyword **def** explicitly indicates the following code is a function definition. **identifier** field is used to specify the name of function.

mapping-relation -> **<(***type-specifier*$_1$*, type-specifier*$_2$**) -> (***type-specifier*$_3$*, type-specifier*$_4$**)>**

mapping-relation defines the mapping relation of a pair of input and output for the function. The format is given as < (*type-specifier*$_1$*, type-specifier*$_2$) -> (*type-specifier*$_3$*, type-specifier*$_4$) >. For mapper, it specifies k_1, v_1 and k_2, v_2 as in Map(k_1,v_1) → list(k_2,v_2). For reducer, it specifies k_2, v_2 and k_3, v_3 as in Reduce(k_2, list (v_2)) → (k_3,v_3).

parameters -> **(identifier**$_1$**, identifier**$_2$ **)**

Parameters refers to the identifiers that receive values passed to a function. **identifier**$_1$ is of type as *type-specifier*$_1$ specifies. **identifier**$_2$ is default to be a List of type as *type-specifier*$_2$ *specifies.*

function-type -> **Mapper**
| **Reducer**

At the end of the function declaration, it is compulsory for users to explicitly specify the function

type. Exactly one mapper and one reducer is allowed and needed in one MR program.

4.3 Scope

A declared object can be visible only within a particular function. Also a declaration is not visible to declarations that came before it. A variable name cannot be referred before declared.

5. Expression

An expression consists of at least one operand and zero or more operators. The operands may be any value, including constants and variables.

5.1 Operators

The general view of the precedence and associative can be shown in the following table.

Operator	Description	Associativity
()	Parentheses	left-to-right
split by	Split a Text value by delimiter	
-	Unary Negative Operator (%prec)	
* /	Multiplication/division	left-to-right
+ -	Addition/subtraction	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
and	Logical AND	left-to-right
or	Logical OR	left-to-right
=	Assignment	

The operators on top on the table possess a higher precedence than those on the bottom of the table. The detail of the expression will be included in the following subsections with the order of precedence from high to low.

5.2 Primary Expression

expression:

literal
identifier
(expression**)**

An identifier is a primary expression, e.g., age. Its type should be specifically declared in the program before it is evaluated in an expression. A literal is a primary constant, e.g., 1, 2, 1.1, true. The type could be Boolean, Int, Double and Text. A parenthesized expression is a primary expression, e.g., (x+y). This expression allows you to group expressions together to allocate them a higher precedence.

5.3 Unary Negative Operator

expression:

- expression

The operand of this operation should have a type of Int or Double. This operation converts the value of the expression from a positive number to a negative number or vise versa.

5.4 Binop Operation

5.4.1 Arithmetic operators

The arithmetic operators include *, /, +, -.
binop-expression:

expression ***** expression
expression **/** expression
expression **+** expression
expression **-** expression

The operands must be of type Int or Double.
The binop expression will return the arithmetic result of the operation. Operator * denotes multiplication, / denotes division, + denotes addition, and – denotes subtraction. When applying the division operation, the second operand could not be zero. Example: 11+22, 21.1*21.5

5.4.2 Relational operation

The relational operators group left-to-right.
relational-expression:

expression **<** expression
expression **>** expression
expression **<=** expression
expression **>=** expression

The result of operations < (less), > (greater), <= (less or euqal), >= (greater or equal), == (equal to), and != (not equal to) is Boolean true/false according to the result of the boolean logic. Examples: x<y, 11>=33

5.4.3 Logical Operation

logical-expression:

expression **and** expression
expression **or** expression

The **and** operator groups left-to-right. It returns true if both its operands are evaluated to be true. Otherwise, it returns false. The **or** operator also groups from left-to-right. It returns true if either of its operands is evaluated to be true. Otherwise it returns false. Both **and** and **or** follows short-circuit evaluation, a.k.a. the second argument is only executed or evaluated if the first argument does not suffice to determine the value of the expression. Examples: (1+1) and 0, (x>2) or (x < 1)

5.5 Split Operation

expression:

split identifier by Text$_{\textbf{constant}}$

This operation will separate a Text constant or variable according to the delimiters specified by the Text constant. The Text constant represents a regular expression used as delimiter. For example, **split** “a-b-c” **by** “-” gives a list of Text [“a”, “b”, “c”] using “-” as delimiter.

5.6 Assignment Expression

expression:

identifier = expression

The value of the expression replaces that of the object referred to by **identifier**. The right operand is converted to the type of the left by the assignment if applicable. Examples: x = 1

5.7 Declaration Expression

expression:

declaration

declaration*:*

type-specifier **identifier**

Identifiers must be preceded by the type of the identifier.

6. Statements

statements -> statements statement | *ϵ*

Statements are a list of statement. Statements are executed in sequence, which are executed for their effect, and do not have values. Statements should not occur within the Literals. They fall into one of the following production:

statement -> *expression***;**
| *block*
| **emit (***expression***,** *expression***);**
| **if** *(expression) block* **else** *block*
| **foreach** *declaration* **in** *expression block*

6.1 Expression Statement

Most statements are expression statements, which have the following form:

statement:

*expression***;**

Usually expression statements are expressions evaluated for their side effects, such as assignments.

6.2 Block statement

statement:

{ *statements* **}**

Block statement is the compound statement surrounded by brackets. It groups a set of statements into a syntactic unit, so that the several statements can be used where one is expected.

6.3 Emit Statement

statement:

emit (*expression*, *expression***);**

The emit statement is used for output in map function and reduce function: both a key and a value must be emitted to the next list in the data flow.

6.4 Conditional Statement

statement:

if *(expression) block* **else** *block*
if *(expression) block*

Conditional statement chooses one of the two blocks to execute, based on the evaluation of the expression. If the expression is evaluated as true, the first sub-statement is evaluated. If the expression is false, the second sub-statement is executed.

6.5 Iteration Statement

statement*:*

foreach *declaration* **in** *expression block*

The foreach structure is used to traverse a list given by the expression. It iterates through each object in the list and execute the block statement.

7. Grammar Summery

Notation Convention:

italic = non-terminal
bold = terminal

1. Types

type-specifier -> *atom-type-specifier*
| *list-type-specifier*
atom-type-specifier -> **Text**
| **Int**
| **Double**
| **Boolean**

list-type-specifier -> **List<***atom-type-specifier***>**

2. Program Structure

program -> configuration-declaration mapper-definition reducer-definition

configuration-declaration -> **#** *configuration-attribute* **= Text**$_{\textbf{const}}$**;**
configuration-attribute -> **JobName**

mapper-definition -> **def identifier** *mapping-relation parameters* **:** *function-type block*
reducer-definition -> **def identifier** *mapping-relation parameters***:** *function-type block*

block -> **{** *statements* **}**

mapping-relation -> **<** **(***type-specifier, type-specifier***)** **->** **(***type-specifier, type-specifier***)** **>**
parameters -> **(identifier, identifier)**
function-type -> **Mapper**
| **Reducer**

---

3. Expression

literal -> **Text**$_{\text{constant}}$ | **Int**$_{\text{constant}}$ | **Double**$_{\text{const}}$ | **Boolean**$_{\text{const}}$
expression -> *literal*
| **identifier**
| **(**expression**)**
| **-***expression*
| *expression binop expression*
| **identifier =** *expression*
| **split identifier by Text**$_{\text{constant}}$
| *declaration*

binop -> **+ - * / and or < > <= >= == !=**
declaration -> *type-specifier* **identifier**
| *type-specifier* **identifier =** *expression*

4. Statement

statements -> *statements statement* | *ϵ*
statement -> *expression***;**
| *block*
| **emit (***expression***,** *expression***);**
| **if** *(expression) block*
| **if** *(expression) block* **else** *block*
| **foreach** *declaration* **in** *expression block*

8. Reference

Hadoop Tutorial on Yahoo Developer Network, http://developer.yahoo.com/hadoop/tutorial/
Wikipedia, http://en.wikipedia.org/wiki/MapReduce