Design Patterns

What's a design pattern?

Design patterns have been making developers’ lives easier for years. They are tools for solving problems in a reusable and general way so that the developer can spend less time figuring out how he’s going to overcome a hurdle and move onto the next one.

Design patterns are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code.

A pattern cannot be just copied into a program, the way you can with off-the-shelf functions or libraries. The pattern is not a specific piece of code, but a general concept for solving a particular problem. Follow the pattern details and implement a solution that suits the realities of your own program.

Patterns are often confused with algorithms, because both concepts describe typical solutions to some known problems. While an algorithm always defines a clear set of actions that can achieve some goal, a pattern is a more high-level description of a solution. The code of the same pattern applied to two different programs may be different.

An analogy to an algorithm is a cooking recipe: both have clear steps to achieve a goal. On the other hand, a pattern is more like a blueprint: you can see what the result and its features are, but the exact order of implementation is up to you.

What does the pattern consist of?

Most patterns are described very formally so people can reproduce them in many contexts. Here are the sections that are usually present in a pattern description:

Intent of the pattern briefly describes both the problem and the solution.

Motivation further explains the problem and the solution the pattern makes possible.

Structure of classes shows each part of the pattern and how they are related.

Code example in one of the popular programming languages makes it easier to grasp the idea behind the pattern.

Summarization Patterns - Numerical Summarization Patterns

Summarization Patterns:

Summarization analytics are all about grouping similar data together and then performing an operation such as calculating a statistic, building an index, or just simply counting.

Calculating some sort of aggregate over groups in your data set is a great way to easily extract value right away. For example, to calculate the total amount of money your stores have made by state or the average amount of time someone spends logged into your website by demographic.

Numerical Summarization Patterns:

Pattern Description

The numerical summarizations pattern is a general pattern for calculating aggregate statistical values over the data. Be careful of how deceptively simple this pattern is! It is extremely important to use the combiner properly and to understand the calculation.

Intent

Group records together by a key field and calculate a numerical aggregate per group to get a top-level view of the larger data set.

Consider θ to be a generic numerical summarization function we wish to execute over some list of values (v1, v2, v3, …, vn) to find a value λ, i.e. λ = θ(v1, v2, v3, …, vn). Examples of θ include a minimum, maximum, average, median, and standard deviation.

Motivation

Many datasets these days are too large for a human to get any real meaning out of it by reading through it manually. For example, website logs were generated based on the details about the user logs onto the website each time, enters a query, or performs any other notable action, it would be extremely difficult to notice any real usage patterns just by reading through terabytes of log files with a text reader. If we group logins by the hour of the day and perform a count of the number of records in each group, you can plot these counts on a histogram and recognize times when website is more active. Similarly, if we group advertisements by types, you can determine how affective your ads are for better targeting. Maybe one of the examples is to cycle ads based on how effective they are at the time of day. All of these types of questions can be answered through numerical summarizations to get a top-level view of data.

Applicability

Numerical summarizations should be used when both of the following are true:

• You are dealing with numerical data or counting.

• The data can be grouped by specific fields.

Structure

Figure 2-1 shows the general structure of how a numerical summarization is executed in MapReduce.

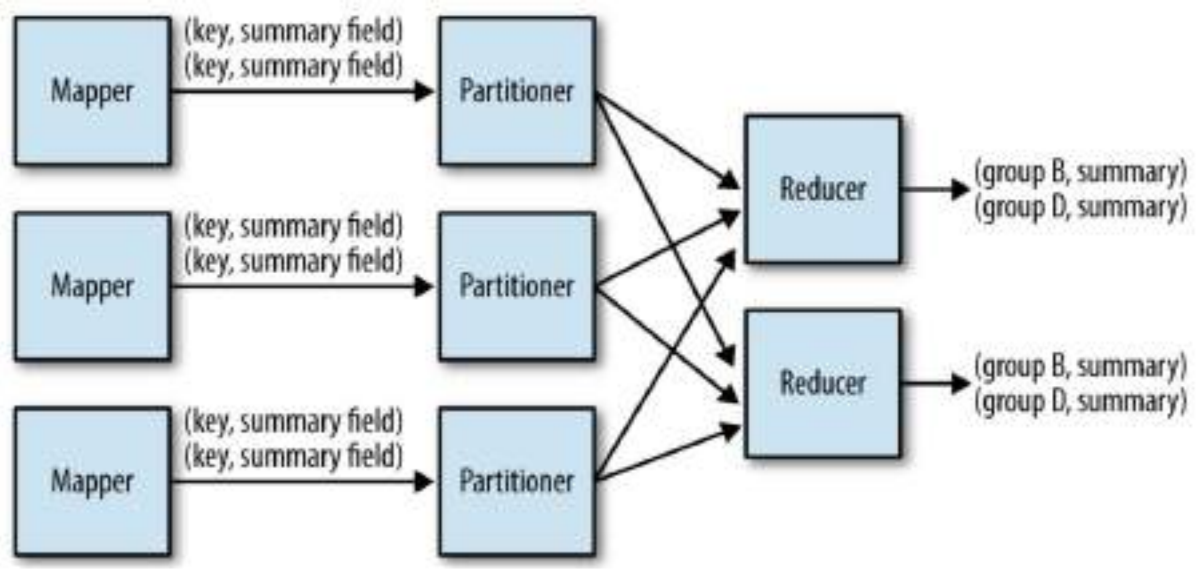


Figure 2-1. The structure of the numerical summarizations pattern

The breakdown of each MapReduce component is described in detail:

- The mapper outputs keys that consist of each field to group by, and values consisting of any pertinent numerical items. Imagine the mapper setting up a relational table, where the columns relate to the fields which the function θ will be executed over and each row contains an individual record output from the mapper. The output value of the mapper contains the values of each column and the output key determines the table as a whole, as each table is created by MapReduce's grouping functionality.
- The combiner can greatly reduce the number of intermediate key/value pairs to be sent across the network to the reducers for some numerical summarization functions. If the

function θ is an associative and commutative operation, it can be used for this purpose. That is, if you can arbitrarily change the order of the values and you can group the computation arbitrarily, you can use a combiner here.

- Numerical summaries can benefit from a custom partitioner to better distribute key/value pairs across n number of reduce tasks. The need for this is rare, but can be done if job execution time is critical, the amount of data is huge, and there is severe data skew.
- The reducer receives a set of numerical values (v1, v2, v3, …, vn) associated with a group-by key records to perform the function λ = θ(v1, v2, v3, …, vn). The value of λ is output with the given input key.

Known uses

Word count

The “Hello World” of MapReduce. The application outputs each word of a document as the key and “1” as the value, thus grouping by words. The reduce phase then adds up the integers and outputs each unique word with the sum.

Record count

A very common analytic to get a heartbeat of your data flow rate on a particular interval (weekly, daily, hourly, etc.).

Min/Max/Count

An analytic to determine the minimum, maximum, and count of a particular event, such as the first time a user posted, the last time a user posted, and the number of times they posted in between that time period. You don’t have to collect all three of these aggregates at the same time, or any of the other use cases listed here if you are only interested in one of them.

Average/Median/Standard deviation

Similar to Min/Max/Count, but not as straightforward of an implementation because these operations are not associative. A combiner can be used for all three, but requires a more complex approach than just reusing the reducer implementation.

Common Part in the Examples

Note: Common Classes Through Out the Unit are:

1) **Driver Class in MapReduce**
2) **MRDPUtils Class (MRDP stands for MapReduce Design Patterns)**

So only Mapper and Reducer Logics gets changed.

1) **Driver Class in MapReduce**

```java
public class ClassName {

   public static class ClassNameMapper extends
            Mapper<Object, Text, Text, CountAverageTuple> {
        @Override
        public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
            // Parse the input string into a nice map
    Map<String, String> parsed =MRDPUtils.transformXmlToMap(value.toString());
        }
    }

    public static class ClassNameReducer  extends
            Reducer<Text, CountAverageTuple, Text, CountAverageTuple> {
        @Override
        public void reduce(Text key, Iterable<CountAverageTuple> values,
                Context context) throws IOException, InterruptedException {
          }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();

Job job = Job.getInstance(conf,"StackOverflow Average Posts Length per user");
        job.setJarByClass(ClassName.class);
        job.setMapperClass(ClassNameMapper.class);
        job.setCombinerClass(ClassNameReducer.class);
        job.setReducerClass(ClassNameReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(CountAverageTuple.class);

        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
```

- **ClassNameMapper:**
    - This is the mapper class, extending the Mapper class provided by Hadoop. The map method is overridden to define the logic for processing each input record.
- **ClassNameReducer:**
    - This is the reducer class, extending the Reducer class provided by Hadoop. The reduce method is overridden to define the logic for combining the intermediate values generated by the mappers.
- **Mapper (ClassNameMapper):**
    - The map method is responsible for processing each input record. In this case, it reads an XML-formatted input string, transforms it into a map using the MRDPUtils.transformXmlToMap method, and stores the result in the parsed variable.
- **Reducer (ClassNameReducer):**
    - The reduce method is responsible for processing the intermediate key-value pairs generated by the mappers. In this case, it takes a key (representing a user) and an iterable of values (representing the post lengths associated with that user).
- **Main Method (main):**

Job Configuration:

1. **Job job = new Job(conf, "StackOverflow");:**

- Creates a new Hadoop job named "StackOverflow Comment Word Count."
- conf is a Configuration object that manages configuration settings for the job.

2. **job.setJarByClass(ClassName.class);:**

- Sets the JAR file containing the classes for the job. In this case, it is assumed that ClassNameMapper.class is the main class containing the main method.

Mapper Configuration:

1. **job.setMapperClass(ClassNameMapper.class);:**

- Sets the class for the mapper. The ClassNameMapper class will be responsible for processing input records and emitting key-value pairs.

Combiner Configuration:

1. **job.setCombinerClass(ClassNameReducer.class);:**

- Sets the class for the combiner. The combiner is an optional local aggregation step that runs on the output of the mappers before sending data to the reducers. In this case, it uses the ClassNameReducer class as a combiner, which is also the reducer class.

Reducer Configuration:

1. **job.setReducerClass(ClassNameReducer.class);:**

- Sets the class for the reducer. The ClassNameReducer class will process the intermediate key-value pairs generated by the mappers and provide the final output.

Output Key and Value Configuration:

1. **job.setOutputKeyClass(Text.class);:**

- Sets the class for the output keys emitted by the reducer. In this case, it uses the Text class, which is a Hadoop class for handling text data.

2. **job.setOutputValueClass(IntWritable.class);:**

- Sets the class for the output values emitted by the reducer. It uses the IntWritable class, a Hadoop class for handling integer data.

Input and Output Paths:

1. **FileInputFormat.addInputPath(job, new Path(otherArgs[0]));:**

- Sets the input path for the job. It assumes that the input data is in the directory specified by otherArgs[0].

2. **FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));:**

- Sets the output path for the job. The final result will be stored in the directory specified by otherArgs[1].

Job Execution and Exit:

1. **System.exit(job.waitForCompletion(true) ? 0 : 1);:**

- Initiates the execution of the job and waits for it to complete. It exits the program with status 0 if the job completes successfully, or 1 if there is an error.

2) MRDPUtils Class code:

```
import java.util.HashMap;
import java.util.Map;

public class MRDPUtils {
    // This helper function parses the stackoverflow into a Map for us.
    public static Map<String, String> transformXmlToMap(String xml) {
        Map<String, String> map = new HashMap<String, String>();
        try {
            // exploit the fact that splitting on double quote
            // tokenizes the data nicely for us
            String[] tokens = xml.trim().substring(5, xml.trim().length() - 3).split("\"");
            for (int i = 0; i < tokens.length - 1; i += 2) {
                String key = tokens[i].trim();
                String val = tokens[i + 1];
                map.put(key.substring(0, key.length() - 1), val);
            }
        } catch (StringIndexOutOfBoundsException e) {
            System.err.println(xml);
        }
        return map;
    }
}
```

Class Declaration: The class is named MRDPUtils, which stands for "MapReduce Design Patterns Utilities".

Method: transformXmlToMap: This method takes a String xml as input and returns a Map<String, String>. The purpose of this method is to parse XML data and convert it into a key-value map. The method starts by creating a new HashMap to store the key-value pairs.

XML Parsing:

- The XML parsing is done inside a try-catch block.
- It attempts to tokenize the XML data by splitting it based on double quotes (") which is a common technique for extracting attribute values from XML tags.
- The substring(5, xml.trim().length() - 3) is used to remove the XML tag brackets (e.g., <tag>...</tag>), assuming the XML has a structure where the first 5 characters and the last 3 characters can be ignored.
- The resulting array of tokens contains alternating keys and values.

Populating the Map:

- The method then iterates over the array of tokens using a for loop with a step of 2 (i += 2), as keys and values are expected to be in alternating positions.
- It trims each key and assigns it to the variable key.
- It assigns the next token (representing the value) to the variable val.
- It puts the key-value pair into the map, where the key is obtained by removing the last character of the key (to handle the possibility of a trailing space in the key).

Exception Handling:

There's a catch block to handle StringIndexOutOfBoundsException, which might occur if the XML is not well-formed or if the indexing assumptions are incorrect. In such cases, the method prints the problematic XML to the standard error stream.

Return:

- The method returns the populated map containing the extracted key-value pairs from the XML.
- This utility class can be used in a MapReduce program to parse XML data and convert it into a format suitable for further processing within the MapReduce framework.

Numerical Summarization Pattern Examples

First Example:

Hadoop Example: Word Count

The “Word Count” program is the canonical example in MapReduce, and for good reason. It is a straightforward application of MapReduce and MapReduce can handle it extremely efficiently.

In this particular example, we’re going to be doing a word count over user-submitted comments on StackOverflow. The content of the Text field will be pulled out and pre-processed a bit, and then we’ll count up how many times we see each word. An example record from this data set is:

```
<row Id="8189677" PostId="6881722" Text="Have you looked at Hadoop?"
CreationDate="2011-07-30T07:29:33.343" UserId="831878" />
```

This record is the 8,189,677th comment on Stack Overflow, and is associated with post number 6,881,722, and is by user number 831,878. The number of the PostId and the UserId are foreign keys to other portions of the data set.

This code is derived from the “Word Count” example that ships with Hadoop Core:

```
public static class WordCountMapper
extends Mapper<Object, Text, Text, IntWritable> {
private final static IntWritable one = new IntWritable(1);
private Text word = new Text();
public void map(Object key, Text value, Context context) throws IOException,
InterruptedException { // Parse the input string into a nice map
Map<String, String> parsed = MRDPUtils.transformXmlToMap(value.toString());
// Grab the "Text" field, since that is what we are counting over
String txt = parsed.get("Text");
// .get will return null if the key is not there
if (txt == null) {// skip this record
return;
}
// Unescape the HTML because the data is escaped.
txt = StringEscapeUtils.unescapeHtml(txt.toLowerCase());
// Remove some annoying punctuation
txt = txt.replaceAll("'", ""); // remove single quotes (e.g., can't)
txt = txt.replaceAll("[^a-zA-Z]", " "); // replace the rest with a space
// Tokenize the string by splitting it up on whitespace into something we can iterate over, then
//send the tokens away
StringTokenizer itr = new StringTokenizer(txt);
while (itr.hasMoreTokens()) {
word.set(itr.nextToken());
context.write(word, one);
}
}
}
```

The first function, MRDPUtils.transformXmlToMap, is a helper function to parse a line of Stack Overflow data in a generic manner. It basically takes a line of the StackOverflow XML and matches up the XML attributes with the values into a Map.

Next, turn your attention to the WordCountMapper class. This code is a bit more complicated than the driver. The first major thing to notice is the type of the parent class:

```
Mapper<Object, Text, Text, IntWritable>
```

They map to the types of the input key, input value, output key, and output value, respectively. We don't care about the key of the input in this case, so that's why we use Object. The data coming in is Text (Hadoop's special String type) because we are reading the data as a line-by-line text document. Our output key and value are Text and IntWritable because we will be using the word as the key and the count as the value.

Up until we start using the StringTokenizer towards the bottom of the code, we're just cleaning up the string. We unescape the data because the string was stored in an escaped manner so that it wouldn't mess up XML parsing. Next, we remove any stray punctuation so that the literal string Hadoop! is considered the same word as Hadoop? and Hadoop.

Finally, for each token (i.e., word) we emit the word with the number 1, which means we saw the word once. The framework then takes over to shuffle and sorts the key/value pairs to reduce tasks.

Finally comes the reducer code, which is relatively simple. The reduce function gets called once per key grouping, in this case each word. We'll iterate through the values, which will be numbers, and take a running sum. The final value of this running sum will be the sum of the ones.

```
public static class IntSumReducer
extends Reducer<Text, IntWritable, Text, IntWritable> {
private IntWritable result = new IntWritable();
public void reduce(Text key, Iterable<IntWritable> values,
Context context) throws IOException, InterruptedException {
int sum = 0;
for (IntWritable val : values) {
sum += val.get();
}
result.set(sum);
context.write(key, result);
}}
```

As in the mapper, we specify the input and output types via the template parent class. Also like the mapper, the types correspond to the same things: input key, input value, output key, and output value. The input key and input value data types must match the output key/value types from the mapper. The output key and output value data types must match the types that the job’s configured FileOutputFormat is expecting. In this case, we are using the default TextOutputFormat, which can take any two Writable objects as output.

The reduce function has a different signature from map, though: it gives you an Iterator over values instead of just a single value. This is because you are now iterating over all values that have that key, instead of just one at a time. The key is very important in the reducer of pretty much every MapReduce job, unlike the input key in the map. Anything we pass to context.write will get written out to a file.

Second Example:

Minimum, maximum, and count example

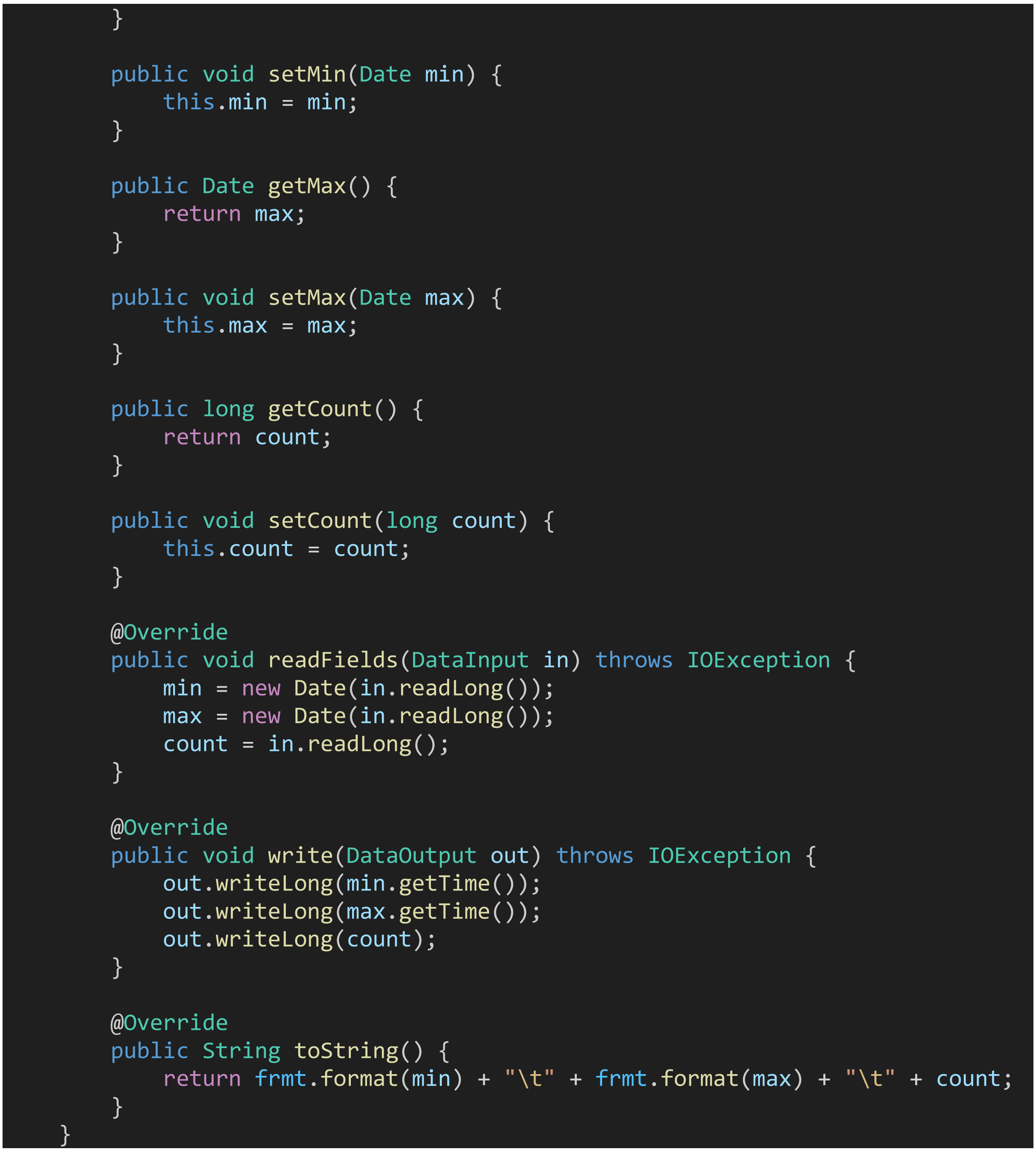
Calculating the minimum, maximum, and count of a given field are all excellent applications of the numerical summarization pattern. After a grouping operation, the reducer simply iterates through all the values associated with the group and finds the min and max, as well as counts the number of members in the key grouping. Due to the associative and commutative properties, a combiner can be used to vastly cut down on the number of intermediate key/value pairs that need to be shuffled to the reducers. If implemented correctly, the code used for your reducer can be identical to that of a combiner.

The following descriptions of each code section explain the solution to the problem.

Problem: Given a list of user’s comments, determine the first and last time a user commented and the total number of comments from that user.

MinMaxCountTuple Code:

```
public static class MinMaxCountTuple implements Writable {
        private Date min = new Date();
        private Date max = new Date();
        private long count = 0;

        private final static SimpleDateFormat frmt = new SimpleDateFormat(
                "yyyy-MM-dd'T'HH:mm:ss.SSS");

        public Date getMin() {
            return min;
```



```
    }

    public void setMin(Date min) {
      this.min = min;
    }

    public Date getMax() {
      return max;
    }

    public void setMax(Date max) {
      this.max = max;
    }

    public long getCount() {
      return count;
    }

    public void setCount(long count) {
      this.count = count;
    }

    @Override
    public void readFields(DataInput in) throws IOException {
      min = new Date(in.readLong());
      max = new Date(in.readLong());
      count = in.readLong();
    }

    @Override
    public void write(DataOutput out) throws IOException {
      out.writeLong(min.getTime());
      out.writeLong(max.getTime());
      out.writeLong(count);
    }

    @Override
    public String toString() {
      return frmt.format(min) + "\t" + frmt.format(max) + "\t" + count;
    }
  }
```

This code defines a public static class named MinMaxCountTuple, which implements the Writable interface. The purpose of this class is to represent a tuple containing a minimum Date, a maximum Date, and a count of occurrences.

Let's break down the code step by step:

1. **Instance Variables:**

- **private Date min:** Represents the minimum date in the tuple. It is initialized with the current date and time.

- **private Date max:** Represents the maximum date in the tuple. It is also initialized with the current date and time.
- **private long count:** Represents the count of occurrences. It is initialized to 0.

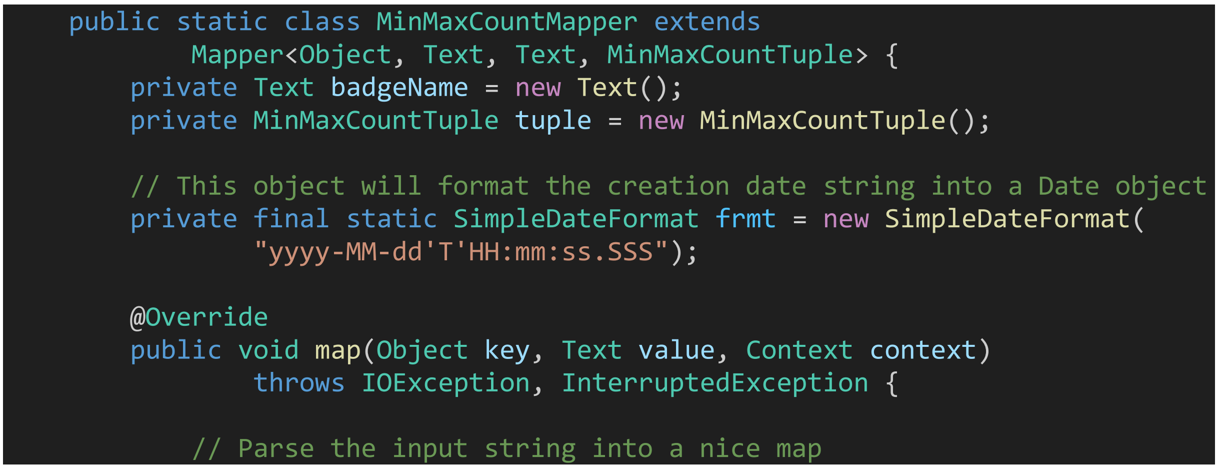
2. **SimpleDateFormat:**
    - **private final static SimpleDateFormat frmt:** This is a static constant SimpleDateFormat object used for formatting dates as strings. It uses the pattern "yyyy-MM-dd'T'HH:mm:ss.SSS".
3. **Getter and Setter Methods:**
    - **getMin(), getMax(), getCount():** These methods provide access to the min, max, and count variables, respectively.
    - **setMin(Date min), setMax(Date max), setCount(long count):** These methods allow setting values for min, max, and count variables.
4. **readFields and write Methods:**

- **readFields(DataInput in):** This method is part of the Writable interface and is used for deserialization. It reads the values of min, max, and count from the provided DataInput stream.
- **write(DataOutput out):** This method is also part of the Writable interface and is used for serialization. It writes the values of min, max, and count to the provided DataOutput stream.

5. **toString Method:**

- **toString():** Overrides the toString method to provide a custom string representation of the object. It uses the SimpleDateFormat to format the min and max dates as strings and concatenates them with the count, separated by tabs.

Mapper Code:



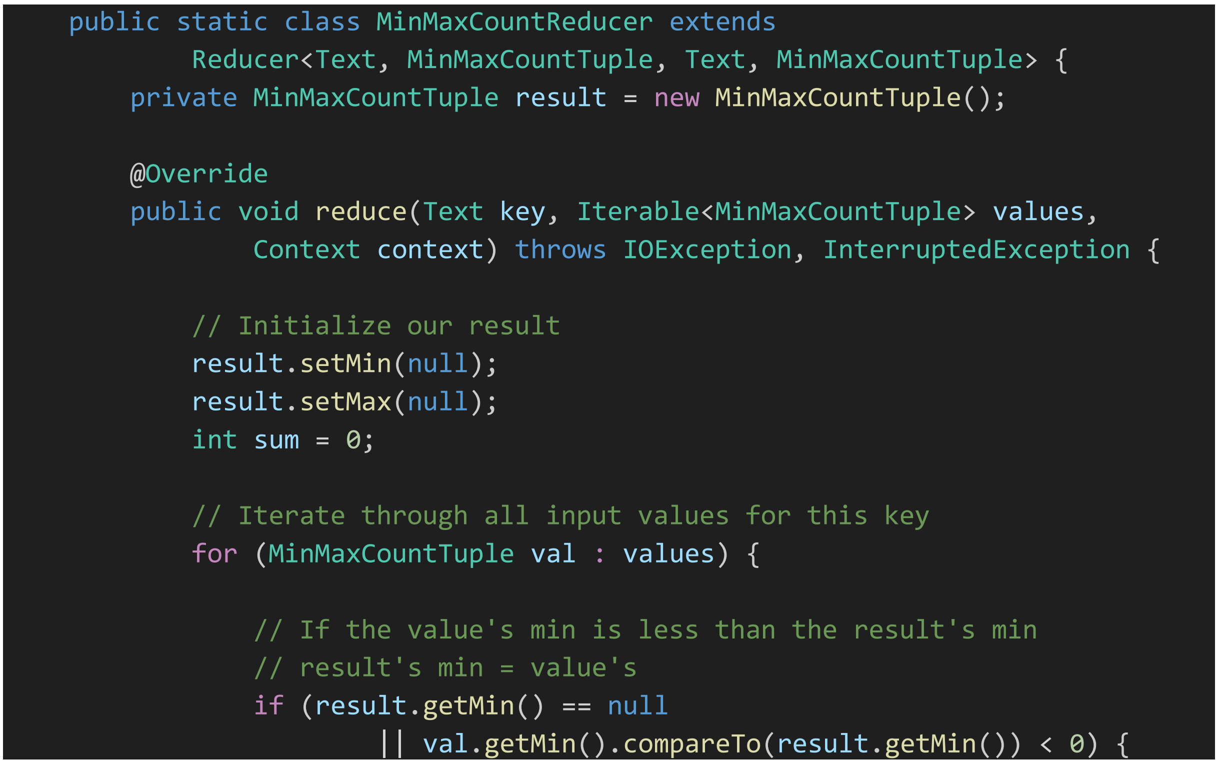
```
    public static class MinMaxCountMapper extends
            Mapper<Object, Text, Text, MinMaxCountTuple> {
        private Text badgeName = new Text();
        private MinMaxCountTuple tuple = new MinMaxCountTuple();

        // This object will format the creation date string into a Date object
        private final static SimpleDateFormat frmt = new SimpleDateFormat(
                "yyyy-MM-dd'T'HH:mm:ss.SSS");

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

            // Parse the input string into a nice map
            Map<String, String> parsed =
MRDPUtils.transformXmlToMap(value.toString());

            // Get the "Date" field
            // We are trying to find the Min/Max for Badge Names
            String date = parsed.get("Date");

            //Get the Badge Name for this record
            String name = parsed.get("Name");

            if (date == null || name == null) {
                // skip this Badge record
                return;
            }

            try {
                // Parse the string into a Date object
                Date valueDate = frmt.parse(date);

                // Set the minimum and maximum date to the valueDate
                tuple.setMin(valueDate);
                tuple.setMax(valueDate);

                // Set the Badge count to 1
                tuple.setCount(1);

                // Set the Badge Name as the output key for this record
                badgeName.set(name);

                // Write key / value pair to the Context
                context.write(badgeName, tuple);
            } catch (ParseException e) {
                // An error occurred parsing the Date string
                // skip this record
            }
        }
}
```

- This class is defined as a static nested class named MinMaxCountMapper within some enclosing class. It extends the Mapper class provided by the Hadoop MapReduce framework.
- The generic parameters <Object, Text, Text, MinMaxCountTuple> represent the input key type, input value type, output key type, and output value type for the Mapper.
- Two instance variables are declared: badgeName of type Text and tuple of type MinMaxCountTuple. These will be used to store the Badge Name and associated MinMaxCountTuple for each record processed by the Mapper.
- An instance of SimpleDateFormat named frmt is created to parse date strings with the specified format ("yyyy-MM-dd'T'HH:mm:ss.SSS").

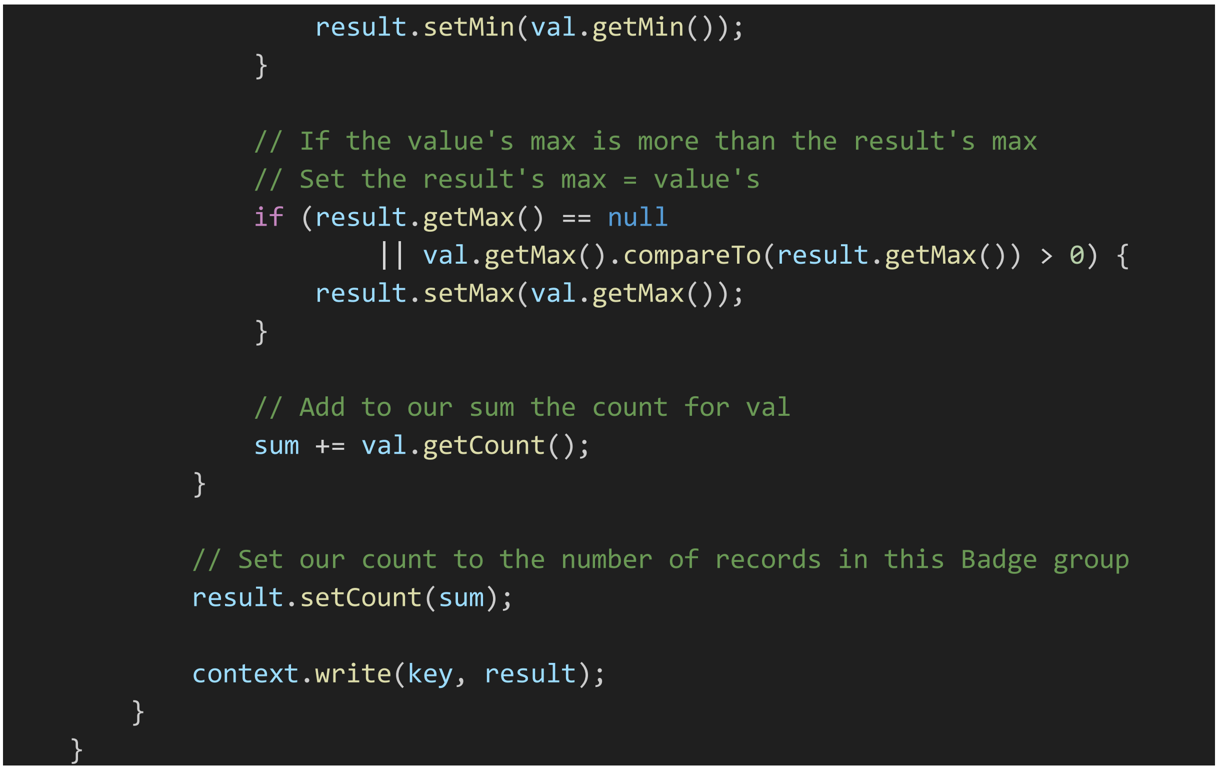
- The map method is an overridden method from the Mapper class. It is called for each input key/value pair.
- The transformXmlToMap method (presumably part of the MRDPUtils class) is used to convert the XML-formatted input into a Map of key-value pairs.
- Extracts the "Date" and "Name" values from the parsed Map.
- Checks if either the "Date" or "Name" is null. If so, it skips processing the current record.
- Tries to parse the date string using the specified date format. If successful, it sets the Min and Max dates in the tuple, sets the count to 1, associates the Badge Name with the tuple, and writes the output key-value pair to the context.
- If a ParseException occurs during date parsing, the record is skipped.

In summary, this Mapper is responsible for extracting information from XML-formatted input records, specifically Badge Names and Dates. It then processes this information to find the minimum and maximum dates associated with each Badge Name, along with the count of occurrences. The output is a key-value pair where the key is the Badge Name, and the value is a MinMaxCountTuple containing the minimum date, maximum date, and count.

Reducer Code



```
    public static class MinMaxCountReducer extends
            Reducer<Text, MinMaxCountTuple, Text, MinMaxCountTuple> {
        private MinMaxCountTuple result = new MinMaxCountTuple();

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

            // Initialize our result
            result.setMin(null);
            result.setMax(null);
            int sum = 0;

            // Iterate through all input values for this key
            for (MinMaxCountTuple val : values) {

                // If the value's min is less than the result's min
                // result's min = value's
                if (result.getMin() == null
                        || val.getMin().compareTo(result.getMin()) < 0) {
```



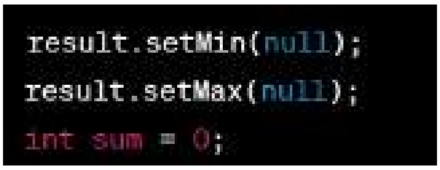
```
                result.setMin(val.getMin());
            }

            // If the value's max is more than the result's max
            // Set the result's max = value's
            if (result.getMax() == null
                    || val.getMax().compareTo(result.getMax()) > 0) {
                result.setMax(val.getMax());
            }

            // Add to our sum the count for val
            sum += val.getCount();
        }

        // Set our count to the number of records in this Badge group
        result.setCount(sum);

        context.write(key, result);
    }
}
```

- This class is defined as a static nested class named MinMaxCountReducer within some enclosing class. It extends the Reducer class provided by the Hadoop MapReduce framework.
- The generic parameters <Text, MinMaxCountTuple, Text, MinMaxCountTuple> represent the input key type, input value type, output key type, and output value type for the Reducer.
- An instance variable named result of type MinMaxCountTuple is declared. This variable will be used to store the final result for each Badge Name.
- The reduce method is an overridden method from the Reducer class. It is called for each unique key (Badge Name) and the associated list of values.

@Override

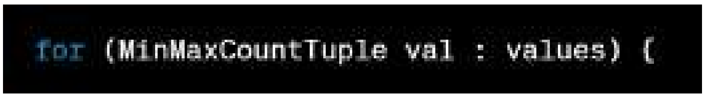
public void reduce(Text key, Iterable<MinMaxCountTuple> values, Context context) throws IOException, InterruptedException {

- The result is initialized by setting both the minimum and maximum dates to null, and sum is initialized to 0.



```
result.setMin(null);
result.setMax(null);
int sum = 0;
```

- A loop iterates through all the MinMaxCountTuple values associated with the current Badge Name.



```
for (MinMaxCountTuple val : values) {
```

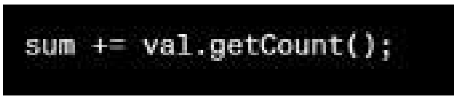
- Checks if the minimum date in the current MinMaxCountTuple is smaller than the minimum date in the result. If true, updates the minimum date in the result with the minimum date from the current value.

```
if (result.getMin() == null || val.getMin().compareTo(result.getMin()) < 0) {
        result.setMin(val.getMin());
}
```

- Checks if the maximum date in the current MinMaxCountTuple is greater than the maximum date in the result. If true, updates the maximum date in the result with the maximum date from the current value.

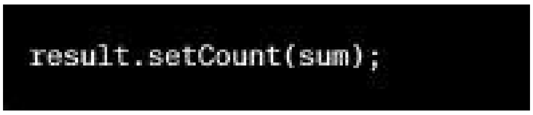
```
if (result.getMax() == null || val.getMax().compareTo(result.getMax()) > 0) {
        result.setMax(val.getMax());
}
```

- Adds the count from the current MinMaxCountTuple to the running total (sum).



```
sum += val.getCount();
```

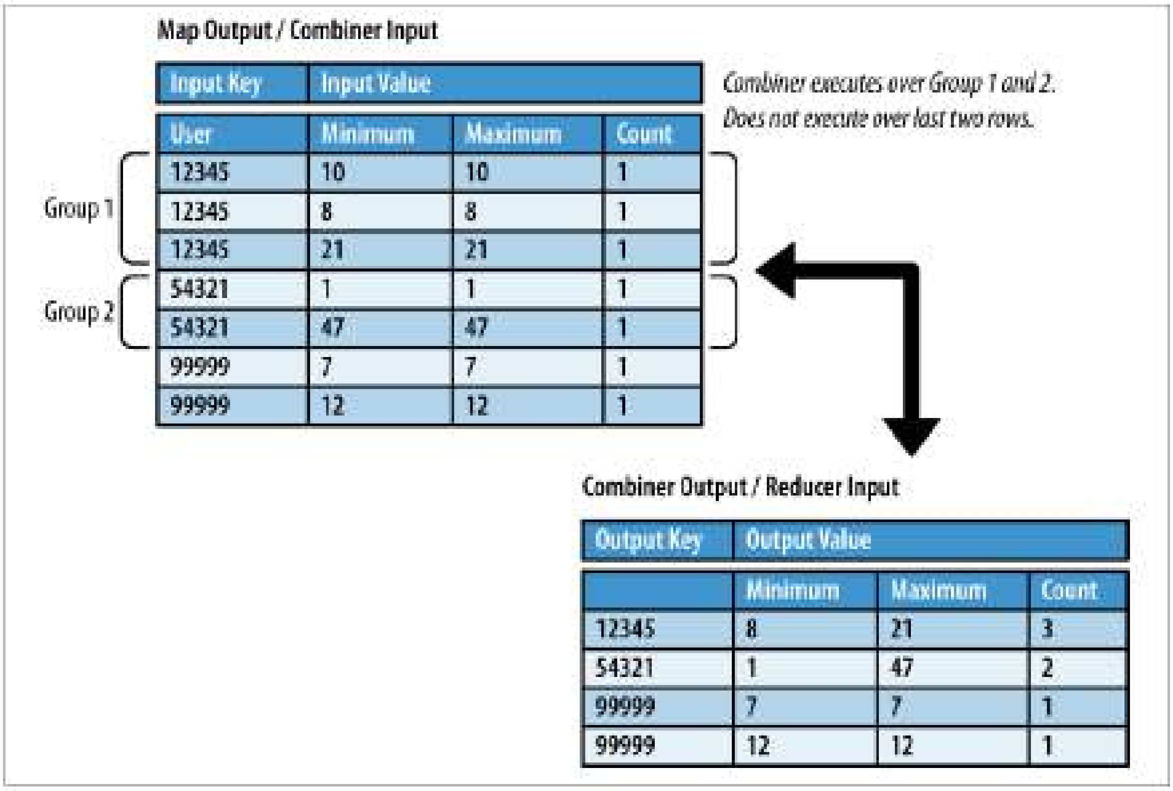
- Sets the count in the result to the total count (sum) for the current Badge Name.



```
result.setCount(sum);
```

- Writes the final result for the current Badge Name to the output context.

```
context.write(key, result);
```



Map Output / Combiner Input

Input Key	Input Value		
User	Minimum	Maximum	Count
12345	10	10	1
12345	8	8	1
12345	21	21	1
54321	1	1	1
54321	47	47	1
99999	7	7	1
99999	12	12	1

Combiner executes over Group 1 and 2. Does not execute over last two rows.

Combiner Output / Reducer Input

Output Key	Output Value		
	Minimum	Maximum	Count
12345	8	21	3
54321	1	47	2
99999	7	7	1
99999	12	12	1

Figure 2-2. The Min/Max/Count MapReduce data flow through the combiner

Third Example:

Average example

To calculate an average, we need two values for each group: the sum of the values that we want to average and the number of values that want into the sum. These two values can be calculated on the reduce side very trivially, by iterating through each value in the set and adding to a running sum while keeping a count. After the iteration, simply divide the sum by the count and output the average.

The following descriptions of each code section explain the solution to the problem.

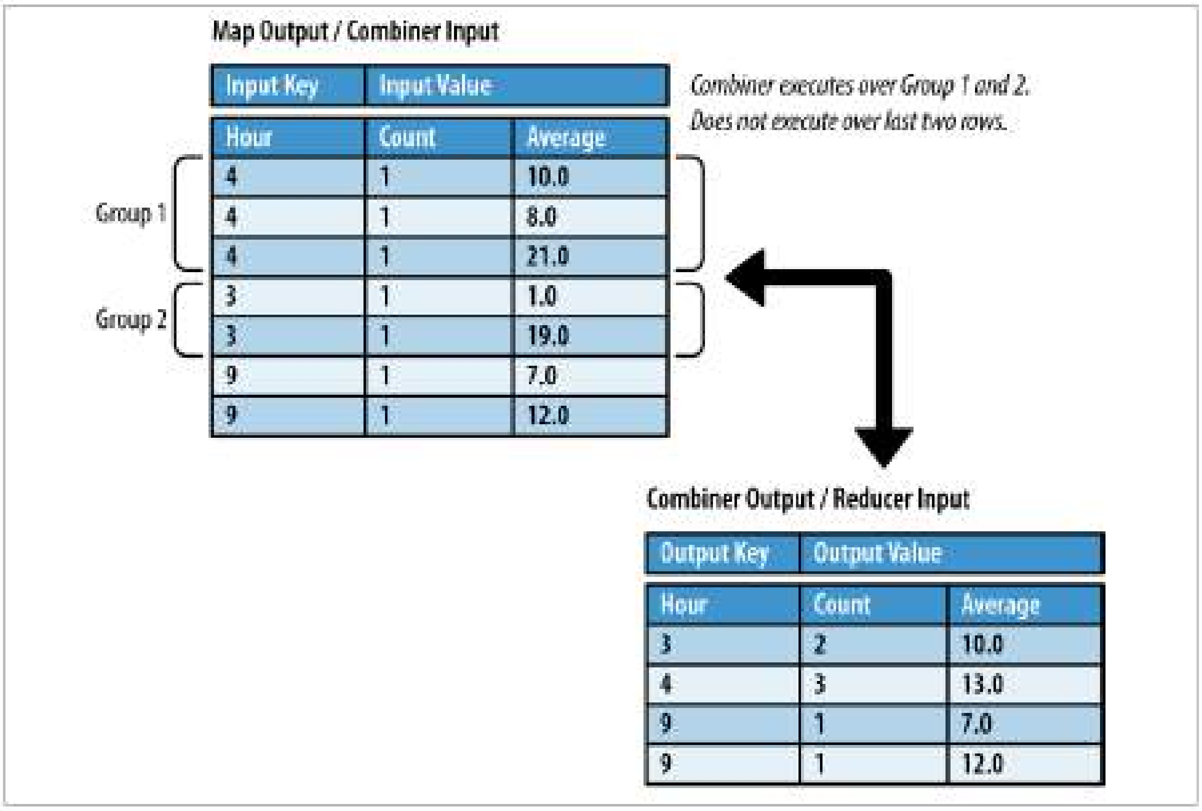
Problem: Given a list of user's comments, determine the average comment length per hour of day.

Mapper code. The mapper will process each input record to calculate the average comment length based on the time of day. The output key is the hour of day, which is parsed from the creation date XML attribute. The output value is two columns, the comment count and the average length of the comments for that hour. Because the mapper operates on one record at a time, the count is simply 1 and the average length is equivalent to the comment length. These two values are output in a custom Writable, a CountAverageTuple. This type contains two float values, a count, and an average.

```
public static class AverageMapper extends
Mapper<Object, Text, IntWritable, CountAverageTuple> {
private IntWritable outHour = new IntWritable();
private CountAverageTuple outCountAverage = new CountAverageTuple();
private final static SimpleDateFormat frmt = new SimpleDateFormat(
"yyyy-MM-dd'T'HH:mm:ss.SSS");
public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
Map<String, String> parsed = transformXmlToMap(value.toString());
// Grab the "CreationDate" field,
// since it is what we are grouping by
String strDate = parsed.get("CreationDate");
// Grab the comment to find the length
String text = parsed.get("Text");
// get the hour this comment was posted in
Date creationDate = frmt.parse(strDate);
outHour.set(creationDate.getHours());
// get the comment length
outCountAverage.setCount(1);
outCountAverage.setAverage(text.length());
// write out the hour with the comment length
context.write(outHour, outCountAverage);
}
}
```

Reducer code. The reducer code iterates through all given values for the hour and keeps two local variables: a running count and running sum. For each value, the count is multiplied by the average and added to the running sum. The count is simply added to the running count. After iteration, the input key is written to the file system with the count and average, calculated by dividing the running sum by the running count.

```
public static class AverageReducer extends
Reducer<IntWritable, CountAverageTuple,
IntWritable, CountAverageTuple> {
private CountAverageTuple result = new CountAverageTuple();
public void reduce(IntWritable key, Iterable<CountAverageTuple> values,
Context context) throws IOException, InterruptedException {
float sum = 0;
float count = 0;
// Iterate through all input values for this key
for (CountAverageTuple val : values) {
sum += val.getCount() * val.getAverage();
count += val.getCount();
}
result.setCount(count);
result.setAverage(sum / count);
context.write(key, result);
}
}
```



Map Output / Combiner Input

Input Key	Input Value	
Hour	Count	Average
4	1	10.0
4	1	8.0
4	1	21.0
3	1	1.0
3	1	19.0
9	1	7.0
9	1	12.0

Combiner executes over Group 1 and 2.
Does not execute over last two rows.

Combiner Output / Reducer Input

Output Key	Output Value	
Hour	Count	Average
3	2	10.0
4	3	13.0
9	1	7.0
9	1	12.0

Figure 2-3. Data flow for the average example

Filtering Patterns

Filtering Patterns don't change the actual records. These patterns all find a subset of data, whether it be small, like a top-ten listing, or large, like the results of a deduplication. Filtering is more about understanding a smaller piece of your data, such as all records generated from a particular user, or the top ten most used verbs in a corpus of text. In short, filtering allows you to apply a microscope to your data.

Sampling, one common application of filtering, is about pulling out a sample of the data, such as the highest values for a particular field or a few random records. Sampling can be used to get a smaller, yet representative, data set in which more analysis can be done without having to deal with the much larger data set. Many machine learning algorithms simply do not work efficiently over a large data set, so tools that build models need to be applied to a smaller subset.

Four patterns are presented:

1. **Filtering**
2. **Bloom Filtering**
3. **Top Ten**
4. **Distinct**

Filtering

Pattern Description

As the most basic pattern, filtering serves as an abstract pattern for some of the other patterns. Filtering simply evaluates each record separately and decides, based on some condition, whether it should stay or go.

Intent

Filter out records that are not of interest and keep ones that are. Consider an evaluation function f that takes a record and returns a Boolean value of true or false. If this function returns true, keep the record; otherwise, toss it out.

Motivation

Data set is large and you want to take a subset of data to focus in on it and perhaps do follow-on analysis. The subset might be a significant portion of the data set. Either way, you need to use the parallelism of MapReduce to wade through all of the data and find the keepers.

For example, you might be interested only in records that have something to do with Hadoop: Hadoop is either mentioned in the raw text or the event is tagged by a “Hadoop” tag. Filtering can be used to keep records that meet the “something to do with Hadoop” criteria and keep them, while tossing out the rest of the records.

Big data and processing systems like Hadoop, in general, are about bringing all of your organization’s data to one location. Filtering is the way to pull subsets back out and deliver them to analysis shops that are interested in just that subset. Filtering is also used to zoom in on a particular set of records that match your criteria that you are more curious about. The exploration of a subset of data may lead to more valuable and complex analytics that are based on the behavior that was observed in the small subset.

Applicability

Filtering is very widely applicable. The only requirement is that the data can be parsed into “records” that can be categorized through some well-specified criterion determining whether they are to be kept.

Structure

The structure of the filter pattern is perhaps the simplest of all the patterns Figure 3-1 shows this pattern.

```
map(key, record):
        if we want to keep record then
                emit key,value
```

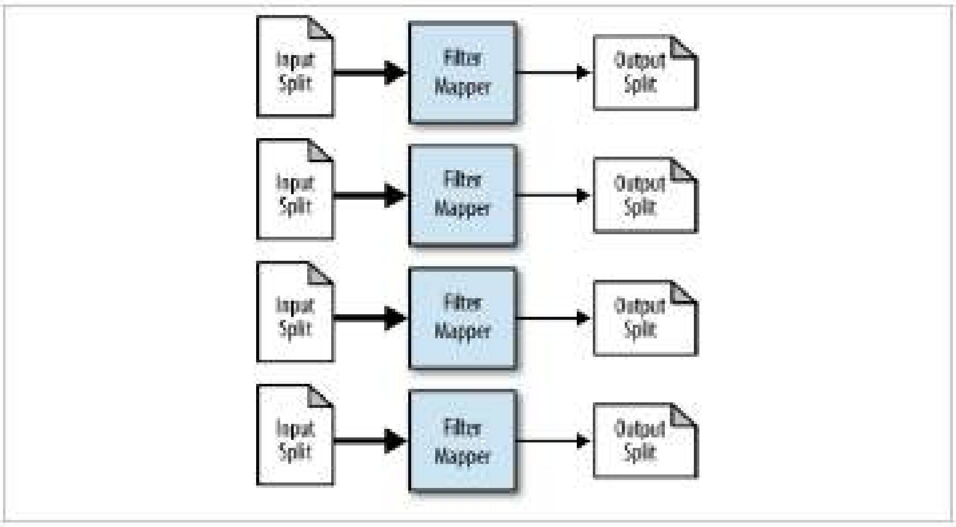


Figure 3-1. The structure of the filter pattern

Filtering is unique in not requiring the “reduce” part of MapReduce. This is because it doesn’t produce an aggregation. Each record is looked at individually and the evaluation of whether or not to keep that record does not depend on anything else in the data set.

The mapper applies the evaluation function to each record it receives. Typically, the mapper outputs the same key/value type as the types of the input, since the record is left unchanged. If the evaluation function returns true, the mapper simply output the key and value verbatim.

Consequences

The output of the job will be a subset of the records that pass the selection criteria. If the format was kept the same, any job that ran over the larger data set should be able to run over this filtered data set, as well.

Known uses

Closer view of data

Prepare a particular subset of data, where the records have something in common or something of interest, for more examination. For example, a local office in Maryland may only care about records originating in Maryland from your international dataset.

Tracking a thread of events

Extract a thread of consecutive events as a case study from a larger data set. For example, you may be interested in how a particular user interacts with your website by analyzing Apache web server logs. The events for a particular user are interspersed with all the other events, so it’s hard to figure out what happened. By filtering for that user’s IP address, you are able to get a good view of that particular user’s activities.

Distributed grep

Grep, a very powerful tool that uses regular expressions for finding lines of text of interest, is easily parallelized by applying a regular expression match against each line of input and only outputting lines that match.

Data cleansing

Data sometimes is dirty, whether it be malformed, incomplete, or in the wrong format. The data could have missing fields, a date could be not formatted as a date, or random bytes of binary data could be present. Filtering can be used to validate that each record is well-formed and remove any junk that does occur.

Simple random sampling

If you want a simple random sampling of your data set, you can use filtering where the evaluation function randomly returns true or false. A simple random sample is a sample of the larger data set in which each item has the same probability of being selected. You can tweak the number of records that make it through by having the evaluation function return true a smaller percentage of the time. For example, if your data set contains one trillion records and you want a sample size of about one million, have the evaluation function return true once in a million (because there are a million millions in a trillion).

Removing low scoring data

If you can score your data with some sort of scalar value, you can filter out records that don't meet a certain threshold. If you know ahead of time that certain types of records are not useful for analysis, you can assign those records a small score and they will get filtered out. This effectively has the same purpose as the top ten pattern discussed later, except that you do not know how many records you will get.

Filtering Examples

Distributed grep

Grep is a popular text filtering utility that dates back to Unix and is available on most Unix-like systems. It scans through a file line-by-line and only outputs lines that match a specific pattern. We'd like to parallelize the regular expression search across a larger body of text. In this example, we'll show how to apply a regular expression to every line in MapReduce.

Mapper code. The mapper is pretty straightforward since we use the Java built-in libraries or regular expressions. If the text line matches the pattern, we' ll output the line. Otherwise, we do nothing and the line is effectively ignored. We use the setup function to retrieve the map regex from the job configuration.

```
public static class GrepMapper
extends Mapper<Object, Text, NullWritable, Text> {
private String mapRegex = null;
public void setup(Context context) throws IOException,
InterruptedException {
mapRegex = context.getConfiguration().get("mapregex");
}
public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
if (value.toString().matches(mapRegex)) {
context.write(NullWritable.get(), value);
}
}
}
```

As this is a map-only job, there is no combiner or reducer. All output records will be written directly to the file system.

Simple Random Sampling

In simple random sampling (SRS), we want to grab a subset of our larger data set in which each record has an equal probability of being selected. Typically this is useful for sizing down a data set to be able to do representative analysis on a more manageable set of data.

Implementing SRS as a filter operation is not a direct application of the filtering pattern, but the structure is the same. Instead of some filter criteria function that bears some relationship to the content of the record, a random number generator will produce a value, and if the value is below a threshold, keep the record. Otherwise, toss it out.

Mapper Code. In the mapper code, the setup function is used to pull the filter_percentage configuration value so we can use it in the map function. In the map function, a simple check against the next random number is done. The random number will be anywhere between 0 and 1, so by comparing against the specified threshold, we can keep or throw out the record.

```
public static class SRSMapper
extends Mapper<Object, Text, NullWritable, Text> {
private Random rands = new Random();
private Double percentage;
protected void setup(Context context) throws IOException,
InterruptedException {
// Retrieve the percentage that is passed in via the configuration
// like this: conf.set("filter_percentage", .5);
// for .5%
String strPercentage = context.getConfiguration()
.get("filter_percentage");
percentage = Double.parseDouble(strPercentage) / 100.0;
}
public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
if (rands.nextDouble() < percentage) {
context.write(NullWritable.get(), value);
}
}
}
```

As this is a map-only job, there is no combiner or reducer. All output records will be written directly to the file system. When using a small percentage, you will find that the files will be tiny and plentiful. If this is the case, set the number of reducers to 1 without specifying a reducer class, which will tell the MapReduce framework to use a single identity reducer that simply collects the output into a single file. The other option would be to collect the files as a post-processing step using hadoop fs -cat.

Bloom Filtering

Pattern Description

Bloom filtering does the same thing as the previous pattern, but it has a unique evaluation function applied to each record.

Intent

Filter such that we keep records that are member of some predefined set of values. It is not a problem if the output is a bit inaccurate, because we plan to do further checking. The predetermined list of values will be called the set of hot values. For each record, extract a feature of that record. If that feature is a member of a set of values represented by a Bloom filter, keep it; otherwise toss it out (or the reverse).

Motivation

Bloom filtering is similar to generic filtering in that it is looking at each record and deciding whether to keep or remove it. However, there are two major differences that set it apart from generic filtering. First, we want to filter the record based on some sort of set membership operation against the hot values. For example: keep or throw away this record if the value in the user field is a member of a predetermined list of users.

Second, the set membership is going to be evaluated with a Bloom filter, described in the Appendix A. In one sense, Bloom filtering is a join operation in which we don’t care about the data values of the right side of the join.

It is comparing one list to another and doing some sort of join logic, using only map tasks. Instead of replicating the hot list everywhere with the distributed cache, as in the replicated join, we will send a Bloom filter data object to the distributed cache. This allows a filter like operation with a Bloom filter instead of the list itself, which allows you to perform this operation across a much larger data set because the Bloom filter is much more compact. Instead of being constrained by the size of the list in memory, you are mostly confined by the feature limitations of Bloom filters.

Using a Bloom filter to calculate set membership in this situation has the consequence that sometimes you will get a false positive. That is, sometimes a value will return as a member of the set when it should not have. If the Bloom filter says a value is not in the Bloom filter, we can guarantee that it is indeed not in the set of values. For more information on why this happens, refer to Appendix A. However, in some situations, this is not that big of a concern. In an example we'll show code for at the end of this chapter, we'll gather a rather large set of "interesting" words, in which when we see a record that contains one of those words, we'll keep the record, otherwise we'll toss it out. We want to do this because we want to filter down our data set significantly by removing uninteresting content. If we are using a Bloom filter to represent the list of watch words, sometimes a word will come back as a member of that list, even if it should not have. In this case, if we accidentally keep some records, we still achieved our goal of filtering out the majority of the garbage and keeping interesting stuff.

Applicability

The following criteria are necessary for Bloom filtering to be relevant:

- Data can be separated into records, as in filtering.
- A feature can be extracted from each record that could be in a set of hot values.
- There is a predetermined set of items for the hot values.
- Some false positives are acceptable.

Structure

Figure 3-2 shows the structure of Bloom filtering and how it is split into two major components. First, the Bloom filter needs to be trained over the list of values. The resulting data object is stored in HDFS. Next is the filtering MapReduce job, which has the same structure as the previous filtering pattern in this chapter, except it will make use of the distributed cache as well. There are no reducers since the records are analyzed one-by-one and there is no aggregation done.

The first step of this job is to train the Bloom filter from the list of values. This is done by loading the data from where it is stored and adding each item to the Bloom filter. The trained Bloom filter is stored in HDFS at a known location.

The second step of this pattern is to do the actual filtering. When the map task starts, it loads the Bloom filter from the distributed cache. Then, in the map function, it iterates through the records and checks the Bloom filter for set membership in the hot values list. Each record is either forwarded or not based on the Bloom filter membership test.

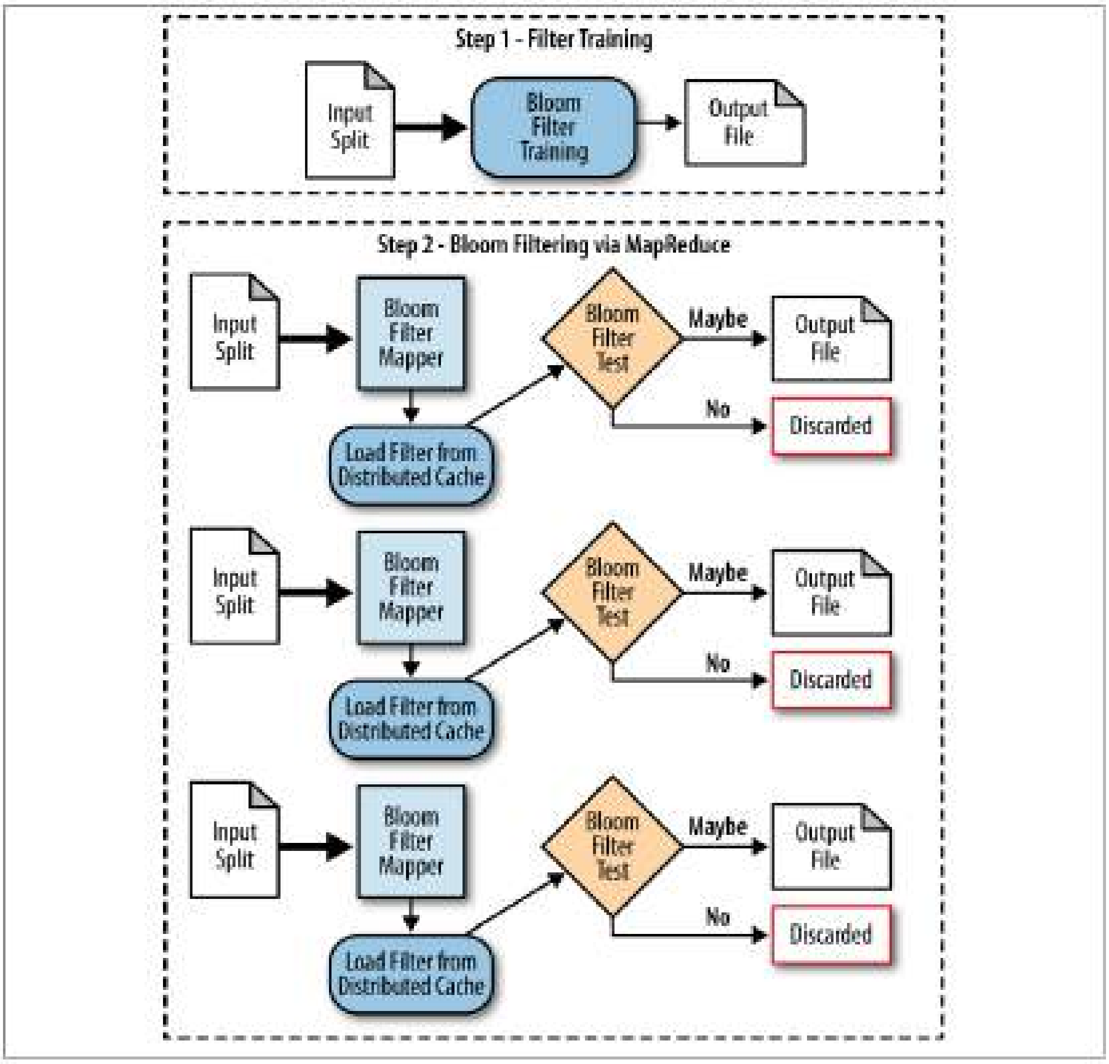


Figure 3-2. The structure of the Bloom filtering pattern

The Bloom filter needs to be re-trained only when the data changes. Therefore, updating the Bloom filter in a lazy fashion (i.e., only updating it when it needs to be updated) is typically appropriate.

Consequences

The output of the job will be a subset of the records in that passed the Bloom filter membership test. You should expect that some records in this set may not actually be in the set of hot values, because Bloom filters have a chance of false positives.

Known uses

Removing most of the nonwatched values

The most straightforward use case is cleaning out values that aren't hot. For example, you may be interested only in data that contains a word in a list of 10,000 words that deal with Hadoop, such as "map," "partitioning," etc. You take this list, train a Bloom filter on it, then check text as it is coming in to see whether you get a Bloom filter hit on any of the words. If you do, forward the record, and if not don't do anything. The fact that you'll get some false positives isn't that big of a deal, since you still got rid of most of the data.

Prefiltering a data set for an expensive set membership check

Sometimes, checking whether some value is a member of a set is going to be expensive. For example, you might have to hit a webservice or an external database to check whether that value is in the set. The situations in which this may be the case are far and few between, but they do crop up in larger organizations. Instead of dumping this list periodically to your cluster, you can instead have the originating system produce a Bloom filter and ship that instead. Once you have the Bloom filter in place and filter out most of the data, you can do a second pass on the records that make it through to double check against the authoritative source. If the Bloom filter is able to remove over 95% of the data, you'll see the external resource hit only 5% as much as before! With this approach, you'll eventually have 100% accuracy but didn't have to hammer the external resource with tons of queries.

Bloom Filtering Examples

Hot list

One of the most basic applications of a Bloom filter is what it was designed for: representing a data set. For this example, a Bloom filter is trained with a hot list of keywords. We use this Bloom filter to test whether each word in a comment is in the hot list. If the test returns true, the entire record is output. Otherwise, it is ignored. Here, we are not concerned with the inevitable false positives that are output due to the Bloom filter. The next example details how one way to verify a positive Bloom filter test using HBase. The following descriptions of each code section explain the solution to the problem. Problem: Given a list of user's comments, filter out a majority of the comments that do not contain a particular keyword.

Bloom filter training. To demonstrate how to use Hadoop Bloom filters, the following code segment generates a Bloom filter off a predetermined set of words. This is a generic application that takes in an input gzip file or directory of gzip files, the number of elements in the file, a desired false positive rate, and finally the output file name.

```
public class BloomFilterDriver {
public static void main(String[] args) throws Exception {
// Parse command line arguments
Path inputFile = new Path(args[0]);
int numMembers = Integer.parseInt(args[1]);
float falsePosRate = Float.parseFloat(args[2]);
Path bfFile = new Path(args[3]);
// Calculate our vector size and optimal K value based on approximations
int vectorSize = getOptimalBloomFilterSize(numMembers, falsePosRate);
int nbHash = getOptimalK(numMembers, vectorSize);
// Create new Bloom filter

BloomFilter filter = new BloomFilter(vectorSize, nbHash,
Hash.MURMUR_HASH);
System.out.println("Training Bloom filter of size " + vectorSize
+ " with " + nbHash + " hash functions, " + numMembers
+ " approximate number of records, and " + falsePosRate
+ " false positive rate");
// Open file for read
String line = null;
int numElements = 0;
FileSystem fs = FileSystem.get(new Configuration());
for (FileStatus status : fs.listStatus(inputFile)) {
BufferedReader rdr = new BufferedReader(new InputStreamReader(
new GZIPInputStream(fs.open(status.getPath()))));
System.out.println("Reading " + status.getPath());
while ((line = rdr.readLine()) != null) {
filter.add(new Key(line.getBytes()));
++numElements;
}
rdr.close();
}
System.out.println("Trained Bloom filter with " + numElements
+ " entries.");
System.out.println("Serializing Bloom filter to HDFS at " + bfFile);
FSDataOutputStream strm = fs.create(bfFile);
filter.write(strm);
strm.flush();
strm.close();
System.exit(0);
}
}
```

A new BloomFilter object is constructed using the optimal vector size and optimal number of hash functions (k) based on the input parameters. Each file returned from listStatus is read line-by-line, and each line is used to train the Bloom filter. After all the input files are ready, the

Bloom filter is serialized to the filename provided at the command line. Because a BloomFilter is also a Writable object, serializing it is fairly trivial. Simply use the FileSystem object to create a new FSDataOutputStream, pass the stream to the filter's write method, then just flush and close the stream!

This Bloom filter can later be deserialized from HDFS just as easily as it was written. Just open up the file using the FileSystem object and pass it to BloomFilter.readFields. Deserialization of this Bloom filter is demonstrated in the setup method of the following Mapper code.

Mapper code. The setup method is called once for each mapper by the Hadoop framework prior to the many calls to map. Here, the Bloom filter is deserialized from the DistributedCache before being used in the map method. The DistributedCache is a Hadoop utility that ensures that a file in HDFS is present on the local file system of each task that requires that file. The Bloom filter was previously trained with a hot list of words.
In the map method, the comment is extracted from each input record. The comment is tokenized into words, and each word is cleaned of any extraneous characters. The clean words are testing against the Bloom filter. If the word is a member, the entire record is output to the file system.

```
public static class BloomFilteringMapper extends
Mapper<Object, Text, Text, NullWritable> {
private BloomFilter filter = new BloomFilter();
protected void setup(Context context) throws IOException,
InterruptedException {
// Get file from the DistributedCache
URI[] files = DistributedCache.getCacheFiles(context
.getConfiguration());
System.out.println("Reading Bloom filter from: "
+ files[0].getPath());
// Open local file for read.
DataInputStream strm = new DataInputStream(new FileInputStream(
files[0].getPath()));
// Read into our Bloom filter.
filter.readFields(strm);
strm.close();
}
public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {

Map<String, String> parsed = transformXmlToMap(value.toString());
// Get the value for the comment
String comment = parsed.get("Text");
StringTokenizer tokenizer = new StringTokenizer(comment);
// For each word in the comment
while (tokenizer.hasMoreTokens()) {
// If the word is in the filter, output the record and break
String word = tokenizer.nextToken();
if (filter.membershipTest(new Key(word.getBytes()))) {
context.write(value, NullWritable.get());
break;
}
}
}
}
```

Because this is a map-only job, there is no combiner or reducer. All output records will be written directly to the file system.

Top Ten

Pattern Description

The top ten pattern is a bit different than previous ones in that you know how many records you want to get in the end, no matter what the input size. In generic filtering, however, the amount of output depends on the data.

Intent

Retrieve a relatively small number of top K records, according to a ranking scheme in your data set, no matter how large the data.

Motivation

Finding outliers is an important part of data analysis because these records are typically the most interesting and unique pieces of data in the set. The point of this pattern is to find the best records for a specific criterion so that you can take a look at them and perhaps figure out what caused them to be so special. If you can define a ranking function or comparison function between two records that determines whether one is higher than the other, you can apply this pattern to use MapReduce to find the records with the highest value across your entire data set. The reason why this pattern is particularly interesting springs from a comparison with how you might implement the top ten pattern outside of a MapReduce context. In SQL, you might be inclined to sort your data set by the ranking value, then take the top K records from that. In MapReduce, as we'll find out in the next chapter, total ordering is extremely involved and uses significant resources on your cluster. This pattern will instead go about finding the limited number of high-values records without having to sort the data.

Plus, seeing the top ten of something is always fun! What are the highest scoring posts on Stack Overflow? Who is the oldest member of your service? What is the largest single order made on your website? Which post has the word “meow” the most number of times?

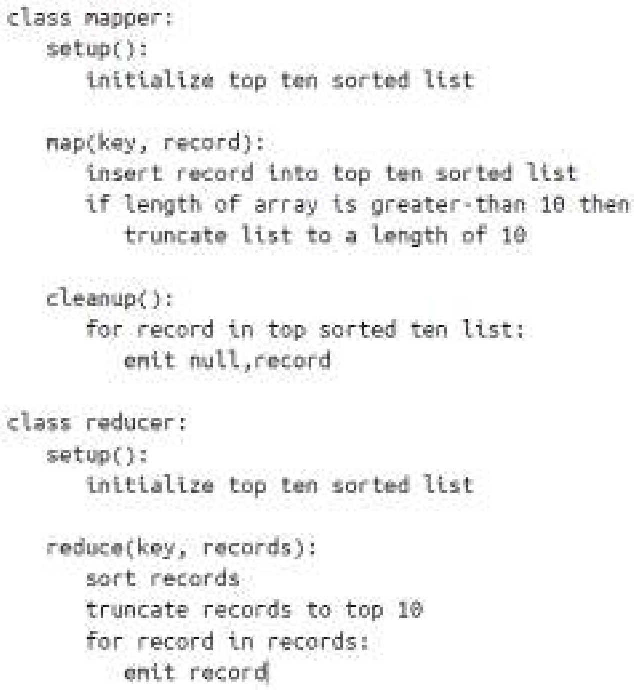
Applicability

• This pattern requires a comparator function ability between two records. That is, we must be able to compare one record to another to determine which is “larger.”

• The number of output records should be significantly fewer than the number of input records because at a certain point it just makes more sense to do a total ordering of the data set.

Structure

This pattern utilizes both the mapper and the reducer. The mappers will find their local top K, then all of the individual top K sets will compete for the final top K in the reducer. Since the number of records coming out of the mappers is at most K and K is relatively small, we’ll only need one reducer. You can see the structure of this pattern in Figure 3-3.



```
class mapper:
   setup():
      initialize top ten sorted list

   map(key, record):
      insert record into top ten sorted list
      if length of array is greater-than 10 then
         truncate list to a length of 10

   cleanup():
      for record in top sorted ten list:
         emit null,record

class reducer:
   setup():
      initialize top ten sorted list

   reduce(key, records):
      sort records
      truncate records to top 10
      for record in records:
         emit record
```

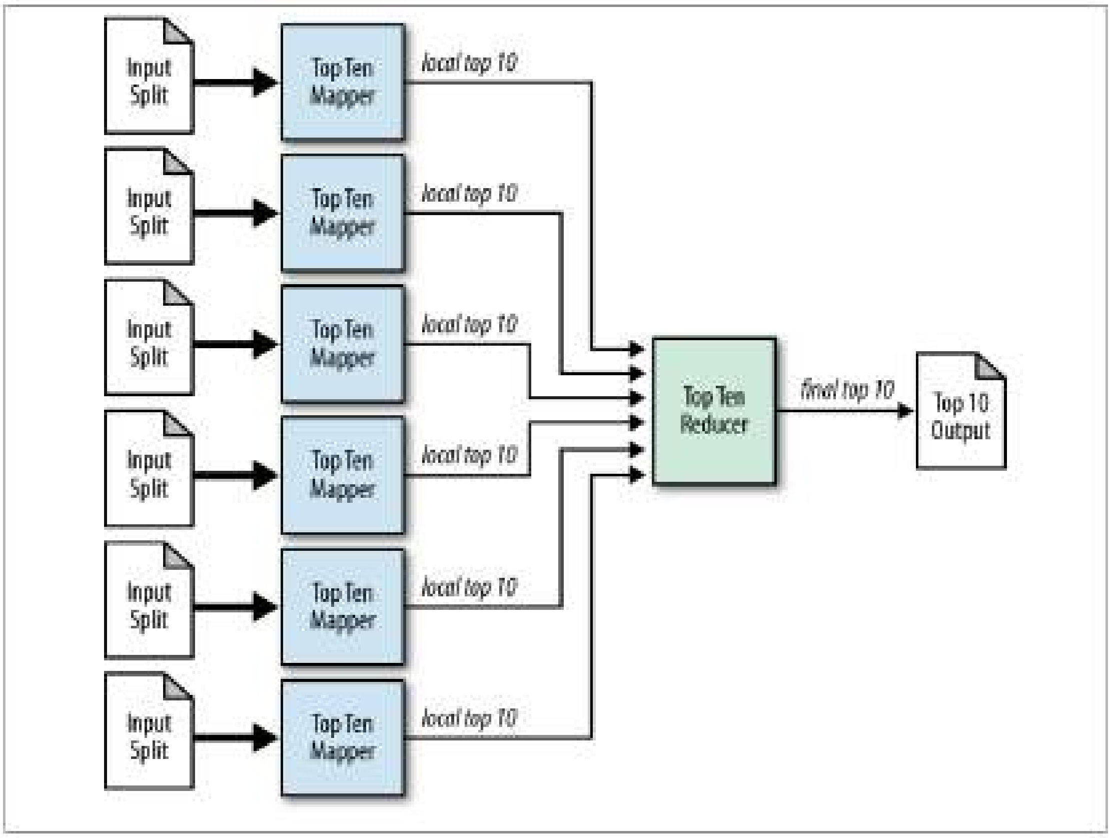


Figure 3-3. The structure of the top ten pattern

The mapper reads each record and keeps an array object of size K that collects the largest K values. In the cleanup phase of the mapper (i.e., right before it exits), we'll finally emit the K records stored in the array as the value, with a null key. These are the lowest K for this particular map task.

We should expect K * M records coming into the reducer under one key, null, where M is the number of map tasks. In the reduce function, we'll do what we did in the mapper: keep an array of K values and find the top K out of the values collected under the null key.

The reason we had to select the top K from every mapper is because it is conceivable that all of the top records came from one file split and that corner case needs to be accounted for.

Consequences

The top K records are returned.

Known uses

Outlier analysis

Outliers are usually interesting. They may be the users that are having difficulty using your system, or power users of your website. Outliers, like filtering and grouping, may give you another perspective from your data set.

Select interesting data

If you are able to score your records by some sort of value score, you can pull the "most valuable" data. This is particularly useful if you plan to submit data to follow on processing, such as in a business intelligence tool or a SQL database, that cannot handle the scale of your original data set. Value scoring can be as complex as you make it by applying advanced algorithms, such as scoring text based on how grammatical it is and how accurate the spelling is so that you remove most of the junk.

Catchy dashboards

This isn't a psychology book, so who knows why top ten lists are interesting to consumers, but they are. This pattern could be used to publish some interesting top ten stats about your website and your data that will encourage users to think more about your data or even to instill some competition.

Top Ten Examples

Top ten users by reputation

Determining the top ten records of a data set is an interesting use of MapReduce. Each mapper determines the top ten records of its input split and outputs them to the reduce phase. The mappers are essentially filtering their input split to the top ten records, and the reducer is responsible for the final ten. Just remember to configure your job to only use one reducer! Multiple reducers would shard the data and would result in multiple "top ten" lists. The following descriptions of each code section explain the solution to the problem.

Problem: Given a list of user information, output the information of the top ten users based on reputation.

Mapper code. The mapper processes all input records and stores them in a TreeMap. A TreeMap is a subclass of Map that sorts on key. The default ordering of Integers is ascending. Then, if there are more than ten records in our TreeMap, the first element (lowest value) can be removed. After all the records have been processed, the top ten records in the TreeMap are output to the reducers in the cleanup method. This method gets called once after all key/value pairs have been through map, just like how setup is called once before any calls to map.

```
public static class TopTenMapper extends
Mapper<Object, Text, NullWritable, Text> {
// Stores a map of user reputation to the record
private TreeMap<Integer, Text> repToRecordMap = new TreeMap<Integer, Text>();
public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
Map<String, String> parsed = transformXmlToMap(value.toString());
String userId = parsed.get("Id");
String reputation = parsed.get("Reputation");
// Add this record to our map with the reputation as the key
repToRecordMap.put(Integer.parseInt(reputation), new Text(value));
// If we have more than ten records, remove the one with the lowest rep
// As this tree map is sorted in descending order, the user with
// the lowest reputation is the last key.
if (repToRecordMap.size() > 10) {
repToRecordMap.remove(repToRecordMap.firstKey());
}
}
protected void cleanup(Context context) throws IOException,
InterruptedException {
// Output our ten records to the reducers with a null key
for (Text t : repToRecordMap.values()) {
context.write(NullWritable.get(), t);
}
}
}
```

Reducer code. Overall, the reducer determines its top ten records in a way that's very similar to the mapper. Because we configured our job to have one reducer using job.setNumReduceTasks(1) and we used NullWritable as our key, there will be one input group for this reducer that contains all the potential top ten records. The reducer iterates through all these records and stores them in a TreeMap. If the TreeMap's size is above ten, the first element (lowest value) is remove from the map. After all the values have been iterated over, the values contained in the TreeMap are flushed to the file system in descending order. This ordering is achieved by getting the descending map from the TreeMap prior to outputting the values. This can be done directly in the reduce method, because there will be only one input group, but doing it in the cleanup method would also work.

```
public static class TopTenReducer extends
Reducer<NullWritable, Text, NullWritable, Text> {
```

```
// Stores a map of user reputation to the record
// Overloads the comparator to order the reputations in descending order
private TreeMap<Integer, Text> repToRecordMap = new TreeMap<Integer, Text>();
public void reduce(NullWritable key, Iterable<Text> values,
Context context) throws IOException, InterruptedException {
for (Text value : values) {
Map<String, String> parsed = transformXmlToMap(value.toString());
repToRecordMap.put(Integer.parseInt(parsed.get("Reputation")),
new Text(value));
// If we have more than ten records, remove the one with the lowest rep
// As this tree map is sorted in descending order, the user with
// the lowest reputation is the last key.
if (repToRecordMap.size() > 10) {
repToRecordMap.remove(repToRecordMap.firstKey());
}
}
for (Text t : repToRecordMap.descendingMap().values()) {
// Output our ten records to the file system with a null key
context.write(NullWritable.get(), t);
}
}
}
```

Distinct

Pattern Description

This pattern filters the whole set, but it’ s more challenging because you want to filter out records that look like another record in the data set. The final output of this filter application is a set of unique records.

Intent

You have data that contains similar records and you want to find a unique set of values.

Motivation

Reducing a data set to a unique set of values has several uses. One particular use case that can use this pattern is deduplication. In some large data sets, duplicate or extremely similar records can become a nagging problem. The duplicate records can take up a significant amount of space or skew top-level analysis results. For example, every time someone visits your website, you collect what web browser and device they are using for marketing analysis. If that user visits your website more than once, you’ll log that information more than once. If you do some analysis to calculate the percentage of your users that are using a specific web browser, the number of times users have used your website will skew the results. Therefore, you should first deduplicate the data so that you have only one instance of each logged event with that device. Records don’t necessarily need to be exactly the same in the raw form. They just need to be able to be translated into a form in which they will be exactly the same. For example, if our web browser analysis done on HTTP server logs, extract only the user name, the device, and the browser that user is using. We don’t care about the time stamp, the resource they were accessing, or what HTTP server it came from.

Applicability

The only major requirement is that you have duplicates values in your data set. This is not a requirement, but it would be silly to use this pattern otherwise!

Structure

This pattern is pretty slick in how it uses MapReduce. It exploits MapReduce’s ability to group keys together to remove duplicates. This pattern uses a mapper to transform the data and doesn’t do much in the reducer. The combiner can always be utilized in this pattern and can help considerably if there are a large number of duplicates. Duplicate records are often located close to another in a data set, so a combiner will deduplicate them in the map phase.

```
map(key, record):
    emit record,null
        reduce(key, records):
            emit key
```

The mapper takes each record and extracts the data fields for which we want unique values. In our HTTP logs example, this means extracting the user, the web browser, and the device values. The mapper outputs the record as the key, and null as the value. The reducer groups the nulls together by key, so we’ll have one null per key. We then simply output the key, since we don’t care how many nulls we have. Because each key is grouped together, the output data set is guaranteed to be unique.

One nice feature of this pattern is that the number of reducers doesn’t matter in terms of the calculation itself. Set the number of reducers relatively high, since the mappers will forward almost all their data to the reducers.

Consequences

The output data records are guaranteed to be unique, but any order has not been preserved due to the random partitioning of the records.

Known uses

Deduplicate data

If you have a system with a number of collection sources that could see the same event twice, you can remove duplicates with this pattern.

Getting distinct values

This is useful when your raw records may not be duplicates, but the extracted information is duplicated across records.

Protecting from an inner join explosion

If you are about to do an inner join between two data sets and your foreign keys are not unique, you risk retrieving a huge number of records. For example, if you have 3,000 of the same key in one data set, and 2,000 of the same key in the other data set, you' ll end up with 6,000,000 records, all sent to one reducer! By running the distinct pattern, you can pair down your values to make sure they are unique and mitigate against this problem.

Distinct Examples

Distinct user IDs

Finding a distinct set of values is a great example of MapReduce's power. Because each reducer is presented with a unique key and a set of values associated with that key, in order to produce a distinct value, we simply need to set our key to whatever we are trying to gather a distinct set of. The following descriptions of each code section explain the solution to the problem.

Problem: Given a list of user's comments, determine the distinct set of user IDs. Mapper code. The Mapper will get the user ID from each input record. This user ID will be output as the key with a null value.

```
public static class DistinctUserMapper extends
Mapper<Object, Text, Text, NullWritable> {
private Text outUserId = new Text();
public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
```

```
Map<String, String> parsed = transformXmlToMap(value.toString());
// Get the value for the UserId attribute
String userId = parsed.get("UserId");
// Set our output key to the user's id
outUserId.set(userId);
// Write the user's id with a null value
context.write(outUserId, NullWritable.get());
}
}
```

Reducer code. The grunt work of building a distinct set of user IDs is handled by the MapReduce framework. Each reducer is given a unique key and a set of null values. These values are ignored and the input key is written to the file system with a null value.

```
public static class DistinctUserReducer extends
Reducer<Text, NullWritable, Text, NullWritable> {
public void reduce(Text key, Iterable<NullWritable> values,
Context context) throws IOException, InterruptedException {
// Write the user's id with a null value
context.write(key, NullWritable.get());
}
}
```

Data Organization Pattern

In many organizations, Hadoop and other MapReduce solutions are only a piece in the larger data analysis platform. Data will typically have to be transformed in order to interface nicely with the other systems. Likewise, data might have to be transformed from its original state to a new state to make analysis in MapReduce easier.

Partitioning

Pattern Description

The partitioning pattern moves the records into categories (i.e., shards, partitions, or bins) but it doesn't really care about the order of records.

Intent

The intent is to take similar records in a data set and partition them into distinct, smaller data sets.

Motivation

If you want to look at a particular set of data—such as postings made on a particular date—the data items are normally spread out across the entire data set. So, looking at just one of these subsets requires an entire scan of all of the data. Partitioning means breaking a large set of data into smaller subsets, which can be chosen by some criterion relevant to your analysis. To improve performance, you can run a job that takes the data set and breaks the partitions out into separate files. Then, when a particular subset for the data is to be analyzed, the job needs only to look at that data.

Partitioning by date is one of the most common schemes. This helps when we want to analyze a certain span of time, because the data is already grouped by that criterion. For instance, suppose you have event data that spans three years in your Hadoop cluster, but for whatever reason the records are not ordered at all by date. If you only care about data from January 27 to February 3 of the current year, you must scan all of the data since those events could be anywhere in the data set. If instead you had the events partitioned into months (i.e., you have a file with January data, a file with February data, etc.), you would only need to run your MapReduce job over the January and February partitions. It would be even better if they were partitioned by day!

Applicability

The one major requirement to apply this pattern is knowing how many partitions you are going to have ahead of time. For example, if you know you are going to partition by day of the week, you know that you will have seven partitions. You can get around this requirement by running an analytic that determines the number of partitions. For example, if you have a bunch of timestamped data, but you don't know how far back it spans, run a job that figures out the date range for you.

Structure

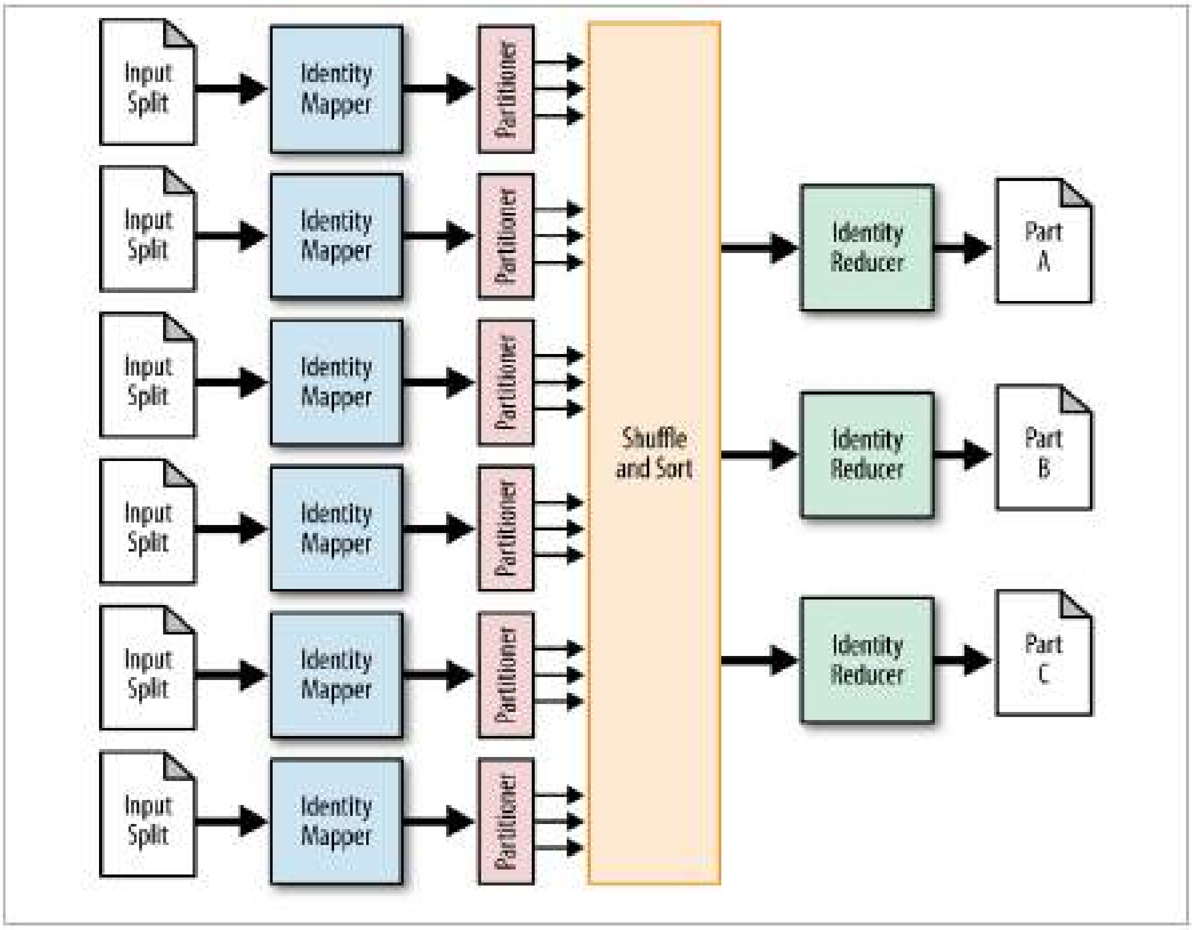


Figure 4-2. The structure of the partitioning pattern

This pattern is interesting in that it exploits the fact that the partitioner partitions data (imagine that!). There is no actual partitioning logic; all you have to do is define the function that determines what partition a record is going to go to in a custom partitioner. Figure 4-2 shows the structure of this pattern.

- In most cases, the identity mapper can be used.
- The custom partitioner is the meat of this pattern. The custom partitioner will determine which reducer to send each record to; each reducer corresponds to particular partitions.
- In most cases, the identity reducer can be used. But this pattern can do additional processing in the reducer if needed. Data is still going to get grouped and sorted, so data can be deduplicated, aggregated, or summarized, per partition.

Consequences

The output folder of the job will have one part file for each partition.

Known uses

Partition pruning by continuous value

You have some sort of continuous variable, such as a date or numerical value, and at any one time you care about only a certain subset of that data. Partitioning the data into bins will allow your jobs to load only pertinent data.

Partition pruning by category

Instead of having some sort of continuous variable, the records fit into one of several clearly defined categories, such as country, phone area code, or language.

Sharding

A system in your architecture has divisions of data—such as different disks—and you need to partition the data into these existing shards.

Partitioning Examples

Partitioning users by last access date

In the StackOverflow data set, users are stored in the order in which they registered. Instead, we want to organize the data into partitions based on the year of the last access date. This is done by creating a custom partitioner to assign record to a particular partition based on that date.

The following descriptions of each code section explain the solution to the problem.

Problem: Given a set of user information, partition the records based on the year of last access date, one partition per year.

Driver code. This driver is a little different than the norm. The job needs to be configured to use the custom built partitioner, and this partitioner needs to be configured. The minimum last access year needs to be configured, which is 2008. The reason for this is explained in the partitioner code section. Also, the number of reducers is important to make sure the full range of partitions is accounted for. Given that the authors are running this example in 2012, the maximum last access year was in 2011, spanning 4 years from 2008 to 2011. Users can fall into these dates as well as those in between, meaning the job is configured to have exactly 4 reducers.

```
...
// Set custom partitioner and min last access date
job.setPartitionerClass(LastAccessDatePartitioner.class);
LastAccessDatePartitioner.setMinLastAccessDate(job, 2008);
// Last access dates span between 2008-2011, or 4 years
job.setNumReduceTasks(4);
...
```

Mapper code. The mapper pulls the last access date out of each input record. This date is output as the key, and the full input record is output as the value. This is so the partitioner can do the work of putting each record into its appropriate partition. This key is later ignored during output from the reduce phase.

```
public static class LastAccessDateMapper extends
Mapper<Object, Text, IntWritable, Text> {
// This object will format the creation date string into a Date object
private final static SimpleDateFormat frmt = new SimpleDateFormat(
"yyyy-MM-dd'T'HH:mm:ss.SSS");
private IntWritable outkey = new IntWritable();
protected void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
Map<String, String> parsed = MRDPUtils.transformXmlToMap(value
.toString());
// Grab the last access date
String strDate = parsed.get("LastAccessDate");
// Parse the string into a Calendar object
Calendar cal = Calendar.getInstance();
cal.setTime(frmt.parse(strDate));
outkey.set(cal.get(Calendar.YEAR));
// Write out the year with the input value
context.write(outkey, value); } }
```

Partitioner code. The partitioner examines each key/value pair output by the mapper to determine which partition the key/value pair will be written. Each numbered partition will be copied by its associated reduce task during the reduce phase. The partitioner implements the Configurable interface. The setConf method is called during task construction to configure the partitioner. Here, the minimum value of the last access date is pulled from the configuration. The driver is responsible for calling LastAccess DatePartitioner.setMinLastAccessDate during job configuration. This date is used to subtract from each key (last access date) to determine what partition it goes to. The minimum last access date is 2008, so all users who last logged into StackOverflow in 2008 will be assigned to partition zero.

```
public static class LastAccessDatePartitioner extends
Partitioner<IntWritable, Text> implements Configurable {
private static final String MIN_LAST_ACCESS_DATE_YEAR =
"min.last.access.date.year";
private Configuration conf = null;
private int minLastAccessDateYear = 0;
public int getPartition(IntWritable key, Text value, int numPartitions) {
return key.get() - minLastAccessDateYear;
}
public Configuration getConf() {
return conf;
}
public void setConf(Configuration conf) {
this.conf = conf;
minLastAccessDateYear = conf.getInt(MIN_LAST_ACCESS_DATE_YEAR, 0);
}
public static void setMinLastAccessDate(Job job,
int minLastAccessDateYear) {
job.getConfiguration().setInt(MIN_LAST_ACCESS_DATE_YEAR,
minLastAccessDateYear);
}
}
```

Reducer code. The reducer code is very simple since we simply want to output the values. The work of partitioning has been done at this point.

```
public static class ValueReducer extends
Reducer<IntWritable, Text, Text, NullWritable> {
protected void reduce(IntWritable key, Iterable<Text> values,
Context context) throws IOException, InterruptedException {
for (Text t : values) {
context.write(t, NullWritable.get());
} }  }
```

Join Patterns

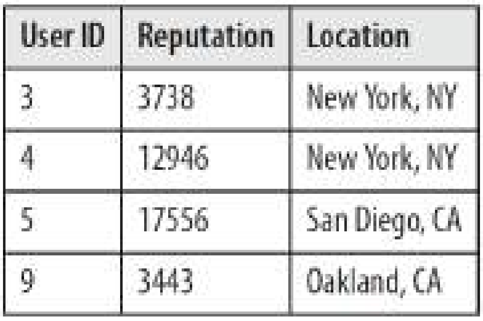
- In SQL, joins are accomplished using simple commands, joins in MapReduce are not nearly this simple.
- MapReduce operates on a single key/value pair at a time, typically from the same input. We are now working with at least two data sets that are probably of different structures, so we need to know what data set a record came from in order to process it correctly.
- Typically, no filtering is done prior to the join operation, so some join operations will require every single byte of input to be sent to the reduce phase, which is an overhead on network.
- For example, joining a terabyte of data onto another terabyte data set could require at least two terabytes of network bandwidth and that's before any actual join logic can be done.

A Refresher Joins:

A join is an operation that combines records from two or more data sets based on a field or set of fields, known as the foreign key. The foreign key is the field in a relational table that matches the column of another table, and is used as a means to cross-reference between tables.

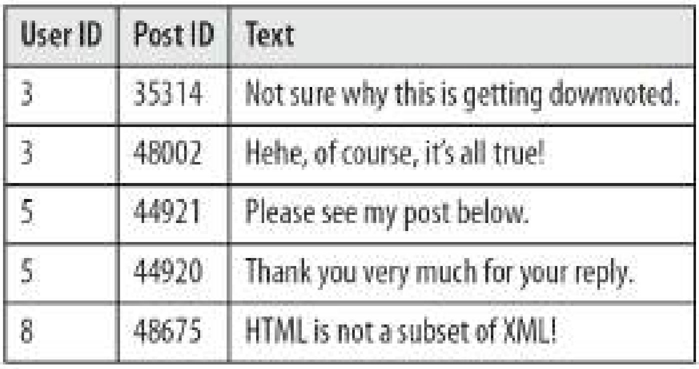
Examples are the simplest way to go about explaining joins, so let's dive right in. To simplify explanations of the join types, two data sets will be used, A and B, with the foreign key defined as f. As the different types of joins are described, keep the two tables A (Table 5-1) and B (Table 5-2) in mind.

Table 5-1. Table A



User ID	Reputation	Location
3	3738	New York, NY
4	12946	New York, NY
5	17556	San Diego, CA
9	3443	Oakland, CA

Table 5-2. Table B



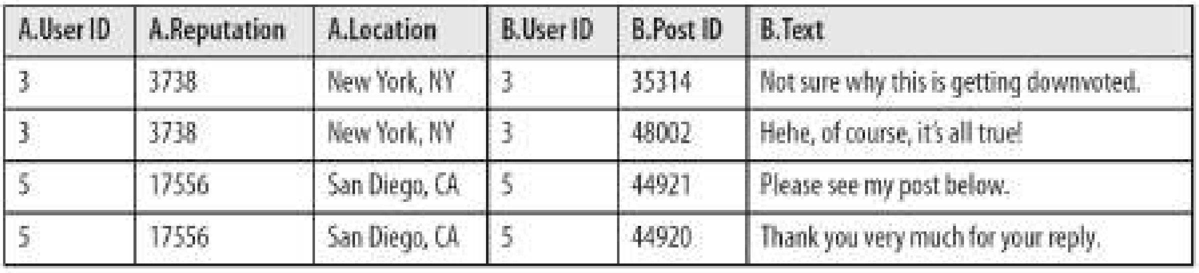
User ID	Post ID	Text
3	35314	Not sure why this is getting downvoted.
3	48002	Hehe, of course, it's all true!
5	44921	Please see my post below.
5	44920	Thank you very much for your reply.
8	48675	HTML is not a subset of XML!

INNER JOIN

With this type of join, records from both A and B that contain identical values for a given foreign key f are brought together, such that all the columns of both A and B now make a new table. Records that contain values of f that are contained in A but not in B, and vice versa, are not represented in the result table of the join operation.

Table 5-3 shows the result of an inner join operation between A and B with User ID as f.

Table 5-3. Inner Join of A + B on User ID



A.User ID	A.Reputation	A.Location	B.User ID	B.Post ID	B.Text
3	3738	New York, NY	3	35314	Not sure why this is getting downvoted.
3	3738	New York, NY	3	48002	Hehe, of course, it's all true!
5	17556	San Diego, CA	5	44921	Please see my post below.
5	17556	San Diego, CA	5	44920	Thank you very much for your reply.

Records with a User ID of 3 or 5 are present in both tables, so they will be in the final table. Users 4 and 9 in table A and User 8 in table B are not represented in the other table, so the records will be omitted.

OUTER JOIN

An outer join is similar to an inner join, but records with a foreign key not present in both tables will be in the final table. There are three types of outer joins and each type will directly affect which unmatched records will be in the final table.

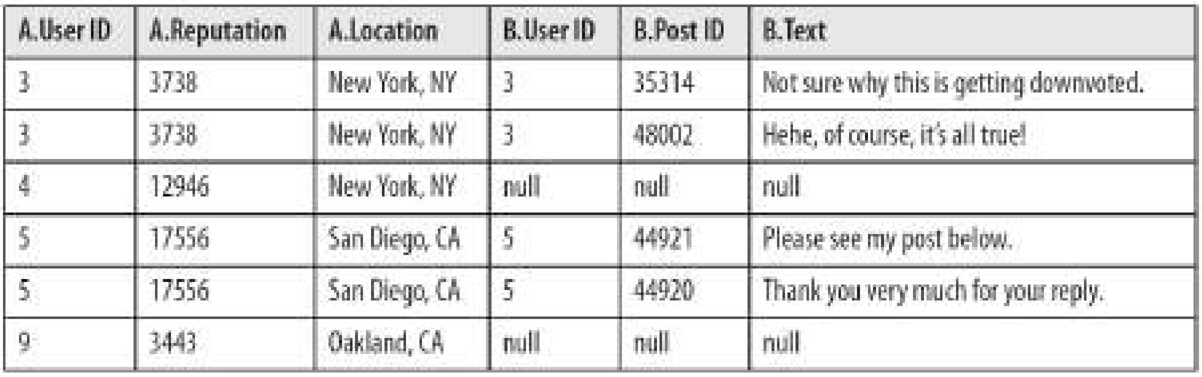
In a left outer join, the unmatched records in the "left" table will be in the final table, with null values in the columns of the right table that did not match on the foreign key. Unmatched records present in the right table will be discarded.

A right outer join is the same as a left outer, but the difference is the right table records are kept and the left table values are null where appropriate.

A full outer join will contain all unmatched records from both tables, sort of like a combination of both a left and right outer join.

Table 5-4 shows the result of a left outer join operation between A and B on UserID.

Table 5-4. Left Outer Join of A + B on User ID

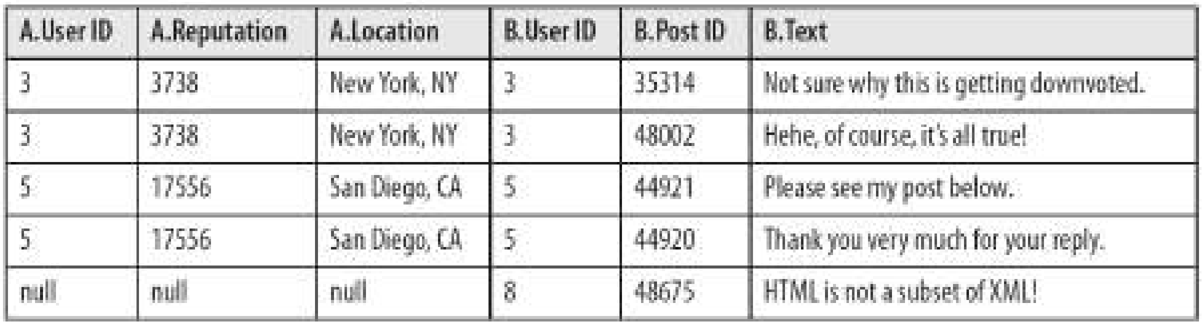


A.User ID	A.Reputation	A.Location	B.User ID	B.Post ID	B.Text
3	3738	New York, NY	3	35314	Not sure why this is getting downvoted.
3	3738	New York, NY	3	48002	Hehe, of course, it's all true!
4	12946	New York, NY	null	null	null
5	17556	San Diego, CA	5	44921	Please see my post below.
5	17556	San Diego, CA	5	44920	Thank you very much for your reply.
9	3443	Oakland, CA	null	null	null

Records with a user ID of 3 or 5 are present in both tables, so they will be in the final table. Users 4 and 9 in table A does not have a corresponding value in table B, but since this is a left outer join and A is on the left, these users will be kept but contain null values in the columns present only in table B. User 8 in B does not have a match in A, so it is omitted.

Table 5-5 shows the result of a right outer join operation between A and B on UserID.

Table 5-5. Right Outer Join of A + B on User ID

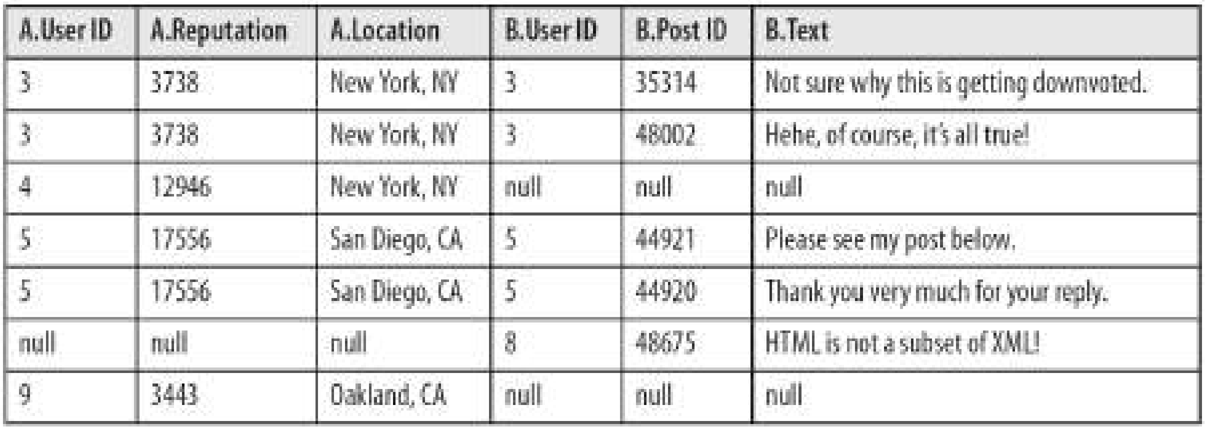


A.User ID	A.Reputation	A.Location	B.User ID	B.Post ID	B.Text
3	3738	New York, NY	3	35314	Not sure why this is getting downvoted.
3	3738	New York, NY	3	48002	Hehe, of course, it's all true!
5	17556	San Diego, CA	5	44921	Please see my post below.
5	17556	San Diego, CA	5	44920	Thank you very much for your reply.
null	null	null	8	48675	HTML is not a subset of XML!

Again, records with a user ID of 3 or 5 are present in both tables, so they will be in the final table. User 8 in B does not have a match in A, but is kept because B is the right table. Users 4 and 9 are omitted as they doesn't have a match in table B.

Table 5-6 shows the result of a full outer join operation between A and B on UserID.

Table 5-6. Full Outer Join of A + B on User ID



A.User ID	A.Reputation	A.Location	B.User ID	B.Post ID	B.Text
3	3738	New York, NY	3	35314	Not sure why this is getting downvoted.
3	3738	New York, NY	3	48002	Hehe, of course, it's all true!
4	12946	New York, NY	null	null	null
5	17556	San Diego, CA	5	44921	Please see my post below.
5	17556	San Diego, CA	5	44920	Thank you very much for your reply.
null	null	null	8	48675	HTML is not a subset of XML!
9	3443	Oakland, CA	null	null	null

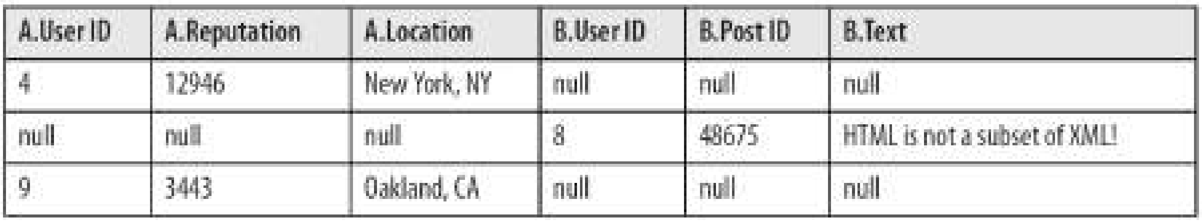
Once again, records with a user ID of 3 or 5 are present in both tables, so they will be in the final table. Users 4, 8, and 9 are present in the resulting table even though they do not contain matches in their respective opposite table.

ANTIJOIN

An antijoin is a full outer join minus the inner join. That is, the resulting table contains only records that did not contain a match on f.

Table 5-7 shows the result of an antijoin operation between A and B on User ID.

Table 5-7. Antijoin of A + B on User ID



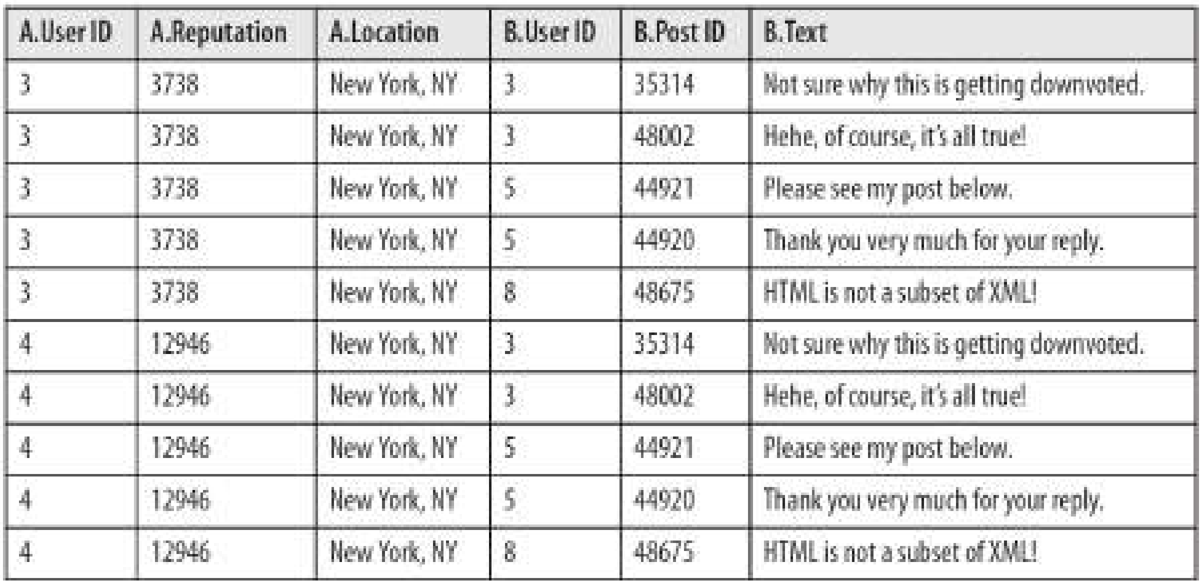
A.User ID	A.Reputation	A.Location	B.User ID	B.Post ID	B.Text
4	12946	New York, NY	null	null	null
null	null	null	8	48675	HTML is not a subset of XML!
9	3443	Oakland, CA	null	null	null

Users 4, 8, and 9 do not contain a value of f in both tables, so they are in the resulting table. Records from user 3 and 5 are not present, as they are in both tables.

CARTESIAN PRODUCT

Table 5-8 shows the sample result of a Cartesian product between A and B.

Table 5-8. Cartesian Product, A × B



A.User ID	A.Reputation	A.Location	B.User ID	B.Post ID	B.Text
3	3738	New York, NY	3	35314	Not sure why this is getting downvoted.
3	3738	New York, NY	3	48002	Hehe, of course, it's all true!
3	3738	New York, NY	5	44921	Please see my post below.
3	3738	New York, NY	5	44920	Thank you very much for your reply.
3	3738	New York, NY	8	48675	HTML is not a subset of XML!
4	12946	New York, NY	3	35314	Not sure why this is getting downvoted.
4	12946	New York, NY	3	48002	Hehe, of course, it's all true!
4	12946	New York, NY	5	44921	Please see my post below.
4	12946	New York, NY	5	44920	Thank you very much for your reply.
4	12946	New York, NY	8	48675	HTML is not a subset of XML!

A Cartesian product or cross product takes each record from a table and matches it up with every record from another table. If table X contains n records and table Y contains m records, the cross product of X and Y, denoted X × Y, contains n × m records. Unlike the other join operations, a Cartesian product does not contain a foreign key. As we will see in the upcoming pattern, this operation is extremely expensive to perform no matter where you implement it, and MapReduce is no exception.

Reduce Side Join

Pattern Description

The reduce side join pattern can take the longest time to execute compared to the other join patterns, but it is simple to implement and supports all the different join operations (discussed in the previous section).

Intent

Join large multiple data sets together by some foreign key.

Motivation

A reduce side join is arguably one of the easiest implementations of a join in MapReduce, and therefore is a very attractive choice. It can be used to execute any of the types of joins described above with relative ease and there is no limitation on the size of your data sets. Also, it can join as many data sets together at once as you need. All that said, a reduce side join will likely require a large amount of network bandwidth because the bulk of the data is sent to the reduce phase. This can take some time, but if you have resources available and aren't concerned about execution time, by all means use it! Unfortunately, if all of the data sets are large, this type of join may be your only choice.

Applicability

A reduce side join should be used when:

- Multiple large data sets are being joined by a foreign key. If all but one of the data sets can be fit into memory, try using the replicated join.
- You want the flexibility of being able to execute any join operation.

Structure

- The mapper prepares the join operation by taking each input record from each of the data sets and extracting the foreign key from the record. The foreign key is written as the output key, and the entire input record as the output value. This output value is flagged by some unique identifier for the data set, such as A or B if two data sets are used. See Figure 5-1.
- A hash partitioner can be used, or a customized partitioner can be created to distribute the intermediate key/value pairs more evenly across the reducers.

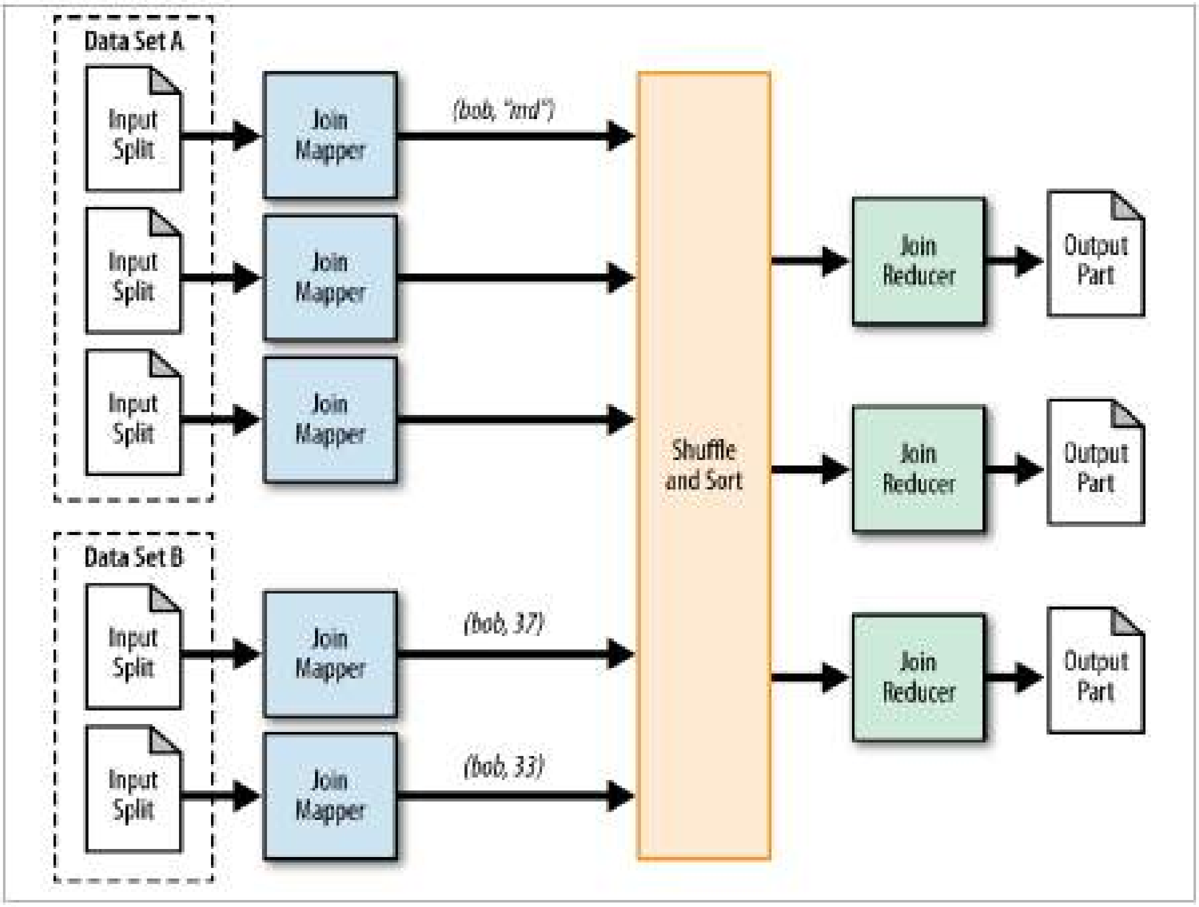


Figure 5-1. The structure of the reduce side join pattern

- The reducer performs the desired join operation by collecting the values of each input group into temporary lists. For example, all records flagged with A are stored in the 'A' list and all records flagged with B are stored in the 'B' list. These lists are then iterated over and the records from both sets are joined together. For an inner join, a joined record is output if all the lists are not empty. For an outer join (left, right, or full), empty lists are still joined with non empty lists. The antijoin is done by examining that exactly one list is empty. The records of the non-empty list are written with an empty writable.

Consequences

The output is a number of part files equivalent to the number of reduce tasks. Each of these part files together contains the portion of the joined records. The columns of each record depend on how they were joined in the reducer. Some column values will be null if an outer join or antijoin was performed.

Reduce Side Join Example

User and comment join

In this example, we'll be using the users and comments tables from the StackOverflow data set. Storing data in this matter makes sense, as storing repetitive user data with each comment is unnecessary. This would also make updating user information difficult. However, having disjoint data sets poses problems when it comes to associating a comment with the user who wrote it. Through the use of a reduce side join, these two data sets can be merged together using the user ID as the foreign key.

In this example, we'll perform an inner, outer, and antijoin. The choice of which join to execute is set during job configuration. Hadoop supports the ability to use multiple input data types at once, allowing you to create a mapper class and input format for each input split from different data sources. This is extremely helpful, because you don't have to code logic for two different data inputs in the same map implementation.

- In the following example, two mapper classes are created: one for the user data and one for the comments.
- Each mapper class outputs the user ID as the foreign key, and the entire record as the value along with a single character to flag which record came from what set.
- The reducer then copies all values for each group in memory, keeping track of which record came from what data set.
- The records are then joined together and output.

The following descriptions of each code section explain the solution to the problem.

Problem: Given a set of user information and a list of user's comments, enrich each comment with the information about the user who created the comment.

Driver code: The job configuration is slightly different from the standard configuration due to the user of the multiple input utility. We also set the join type in the job configuration to args[2] so it can be used in the reducer. The relevant piece of the driver code to use the MultipleInput follows:

```
...
// Use MultipleInputs to set which input uses what mapper
// This will keep parsing of each data set separate from a logical standpoint
// The first two elements of the args array are the two inputs
MultipleInputs.addInputPath(job, new Path(args[0]), TextInputFormat.class,
UserJoinMapper.class);
MultipleInputs.addInputPath(job, new Path(args[1]), TextInputFormat.class,
CommentJoinMapper.class);
job.getConfiguration()..set("join.type", args[2]);
...
```

User mapper code. This mapper parses each input line of user data XML. It grabs the userID associated with each record and outputs it along with the entire input value. It prepends the letter A in front of the entire value. This allows the reducer to know which values came from what data set.

```
public static class UserJoinMapper extends Mapper<Object, Text, Text, Text> {
private Text outkey = new Text();
private Text outvalue = new Text();
public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
// Parse the input string into a nice map
Map<String, String> parsed =
MRDPUtils.transformXmlToMap(value.toString());
String userId = parsed.get("Id");
// The foreign join key is the user ID
outkey.set(userId);
// Flag this record for the reducer and then output
outvalue.set("A" + value.toString());
context.write(outkey, outvalue);
}
}
```

Comment mapper code. This mapper parses each input line of comment XML. Very similar to the UserJoinMapper, it too grabs the user ID associated with each record and outputs it along with the entire input value. The only different here is that the XML attribute UserId represents the user that posted to comment, where as Id in the user data set is the user ID. Here, this mapper prepends the letter B in front of the entire value.

```
public static class CommentJoinMapper extends
Mapperlt;Object, Text, Text, Text> {
private Text outkey = new Text();
private Text outvalue = new Text();
public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
Map<String, String> parsed = transformXmlToMap(value.toString());
// The foreign join key is the user ID
outkey.set( parsed.get("UserId"));
// Flag this record for the reducer and then output
outvalue.set("B" + value.toString());
context.write(outkey, outvalue);
}
}
```

Reducer code. The reducer code iterates through all the values of each group and looks at what each record is tagged with and then puts the record in one of two lists. After all values are binned in either list, the actual join logic is executed using the two lists. The join logic differs slightly based on the type of join, but always involves iterating through both lists and writing to the Context object.

```
public static class UserJoinReducer extends Reducer<Text, Text, Text, Text> {
private static final Text EMPTY_TEXT = Text("");
private Text tmp = new Text();
private ArrayList<Text> listA = new ArrayList<Text>();
private ArrayList<Text> listB = new ArrayList<Text>();
private String joinType = null;
public void setup(Context context) {// Get the type of join from our configuration
joinType = context.getConfiguration().get("join.type");
}
public void reduce(Text key, Iterable<Text> values, Context context)
throws IOException, InterruptedException {
// Clear our lists
listA.clear();
listB.clear();
// iterate through all our values, binning each record based on what
// it was tagged with. Make sure to remove the tag!
while (values.hasNext()) {
tmp = values.next();
if (tmp.charAt(0) == 'A') {
listA.add(new Text(tmp.toString().substring(1)));
} else if (tmp.charAt('0') == 'B') {
listB.add(new Text(tmp.toString().substring(1)));
}
}
// Execute our join logic now that the lists are filled
executeJoinLogic(context);
}
private void executeJoinLogic(Context context)
throws IOException, InterruptedException {
...
}
```

The input data types to the reducer are two Text objects. The input key is the foreign join key, which in this example is the user's ID. The input values associated with the foreign key contain one record from the "users" data set tagged with 'B', as well as all the comments the user posted tagged with 'B'. Any type of data formatting you would want to perform should be done here prior to outputting. For simplicity, the raw XML value from the left data set (users) is output as the key and the raw XML value from the right data set (comments) is output as the value.

Next, let's look at each of the join types. First up is an inner join. If both the lists are not empty, simply perform two nested for loops and join each of the values together.

```
if (joinType.equalsIgnoreCase("inner")) {  // If both lists are not empty, join A with B
if (!listA.isEmpty() && !listB.isEmpty()) {
for (Text A : listA) {
for (Text B : listB) {
context.write(A, B);
} }  } } ...
```

For a left outer join, if the right list is not empty, join *A* with *B*. If the right list is empty, output each record of *A* with an empty string.

```
... else if (joinType.equalsIgnoreCase("leftouter")) {
// For each entry in A,
for (Text A : listA) { // If list B is not empty, join A and B
if (!listB.isEmpty()) {
for (Text B : listB) {
context.write(A, B);
} } else { // Else, output A by itself
context.write(A, EMPTY_TEXT);
} }  } ...
```

A right outer join is very similar, except switching from the check for empty elements from *B* to *A*. If the left list is empty, write records from *B* with an empty output key.

```
... else if (joinType.equalsIgnoreCase("rightouter")) { // For each entry in B,
for (Text B : listB) { // If list A is not empty, join A and B
if (!listA.isEmpty()) {
for (Text A : listA) {
context.write(A, B);
} } else { // Else, output B by itself
context.write(EMPTY_TEXT, B);
} } } ...
```

A full outer join is more complex, in that we want to keep all records, ensuring that we join records where appropriate. If list A is not empty, then for every element in *A*, join with *B* when the *B* list is not empty, or output *A* by itself. If *A* is empty, then just output *B*.

```
... else if (joinType.equalsIgnoreCase("fullouter")) {
// If list A is not empty
if (!listA.isEmpty()) {
// For each entry in A
for (Text A : listA) {
// If list B is not empty, join A with B
if (!listB.isEmpty()) {
for (Text B : listB) {
context.write(A, B);
}
} else {
// Else, output A by itself
context.write(A, EMPTY_TEXT);
}
}
} else {
// If list A is empty, just output B
for (Text B : listB) {
context.write(EMPTY_TEXT, B);
}
}
} ...
```

For an antijoin, if at least one of the lists is empty, output the records from the nonempty list with an empty Text object.

```
... else if (joinType.equalsIgnoreCase("anti")) {
// If list A is empty and B is empty or vice versa
if (listA.isEmpty() ^ listB.isEmpty()) {
// Iterate both A and B with null values
// The previous XOR check will make sure exactly one of
// these lists is empty and therefore the list will be skipped
for (Text A : listA) {
context.write(A, EMPTY_TEXT);
}
for (Text B : listB) {
context.write(EMPTY_TEXT, B);
} } } ...
```

Input and Output Patterns

Sometimes the basic Hadoop paradigm of file blocks and input splits doesn't do what you need, so this is where a custom InputFormat or OutputFormat comes into play.

Customizing Input and Output in Hadoop

Hadoop allows, to modify the way data is loaded on disk in two major ways:

1. configuring how contiguous chunks of input are generated from blocks in HDFS
2. configuring how records appear in the map phase.

The two classes, to do this are RecordReader and InputFormat. These work with the Hadoop MapReduce framework in a very similar way to how mappers and reducers are plugged in. Hadoop also allows you to modify the way data is stored in an analogous way: with an OutputFormat and a RecordWriter.

InputFormat

Hadoop relies on the input format of the job to do three things:

1. Validate the input configuration for the job (i.e., checking that the data is there).

2. Split the input blocks and files into logical chunks of type InputSplit, each of which is assigned to a map task for processing.

3. Create the RecordReader implementation to be used to create key/value pairs from the raw InputSplit. These pairs are sent one by one to their mapper.

- The most common input formats are subclasses of FileInputFormat, with the Hadoop default being TextInputFormat.
- The input format first validates the input into the job by ensuring that all of the input paths exist.
- Then it logically splits each input file based on the total size of the file in bytes, using the block size as an upper bound.
- For example, a 160-megabyte file in HDFS will generate three input splits along the byte ranges 0MB-64MB, 64MB-128MB and 128MB-160MB.
- Each map task will be assigned exactly one of these input splits, and then the RecordReader implementation is responsible for generate key/value pairs out of all the bytes it has been assigned.
- Typically, the RecordReader has the additional responsibility of fixing boundaries, because the input split boundary is arbitrary and probably will not fall on a record boundary.

- For example, the TextInputFormat reads text files using a LineRecordReader to create key/value pairs for each map task for each line of text (i.e., separated by a newline character).
- The key is the number of bytes read in the file so far and the value is a string of characters up to a newline character.
- Because it is very unlikely that the chunk of bytes for each input split will be lined up with a newline character, the LineRecordReader will read past its given "end" in order to make sure a complete line is read.
- This bit of data comes from a different data block and is therefore not stored on the same node, so it is streamed from a DataNode hosting the block.
- This streaming is all handled by an instance of the FSDataInputStream class, and we don't have to deal with any knowledge of where these blocks are.

The InputFormat abstract class contains two abstract methods:

getSplits

The implementation of getSplits typically uses the given JobContext object to retrieve the configured input and return a List of InputSplit objects. The input splits have a method to return an array of machines associated with the locations of the data in the cluster, which gives clues to the framework as to which TaskTracker should process the map task. This method is also a good place to verify the configuration and throw any necessary exceptions, because the method is used on the front-end.

createRecordReader

This method is used on the back-end to generate an implementation of Record Reader. Typically, a new instance is created and immediately returned, because the record reader has an initialize method that is called by the framework.

RecordReader

- The RecordReader abstract class creates key/value pairs from a given InputSplit.
- While the InputSplit represents the byte-oriented view of the split, the RecordReader makes sense out of it for processing by a mapper.
- This is why Hadoop and MapReduce is considered schema on read.
- It is in the RecordReader that the schema is defined, based solely on the record reader implementation, which changes based on what the expected input is for the job.

- Bytes are read from the input source and turned into a Writable Comparable key and a Writable value.
- Custom data types are very common when creating custom input formats, as they are a nice object-oriented way to present information to a mapper.
- A RecordReader uses the data within the boundaries created by the input split to generate key/value pairs.
- In the context of file-based input, the “start” is the byte position in the file where the RecordReader should start generating key/value pairs.
- The “end” is where it should stop reading records.
- These are not hard boundaries as far as the API is concerned—there is nothing stopping a developer from reading the entire file for each map task.
- While reading the entire file is not advised, reading outside of the boundaries it often necessary to ensure that a complete record is generated.
- Consider the case of XML. While using a TextInputFormat to grab each line works, XML elements are typically not on the same line and will be split by a typical MapReduce input.
- By reading past the “end” input split boundary, you can complete an entire record.
- After finding the bottom of the record, you just need to ensure that each record reader starts at the beginning of an XML element.
- After seeking to the start of the input split, continue reading until the beginning of the configured XML tag is read.
- This will allow the MapReduce framework to cover the entire contents of an XML file, while not duplicating any XML records.
- Any XML that is skipped by seeking forward to the start of an XML element will be read by the preceding map task.
- The RecordReader abstract class has a number of methods that must be overridden.

initialize

This method takes as arguments the map task’s assigned InputSplit and TaskAttemptContext, and prepares the record reader. For file-based input formats, this is a good place to seek to the byte position in the file to begin reading.

getCurrentKey and getCurrentValue

These methods are used by the framework to give generated key/value pairs to an implementation of Mapper.

nextKeyValue

Like the corresponding method of the InputFormat class, this reads a single key/value pair and returns true until the data is consumed.

getProgress

Like the corresponding method of the InputFormat class, this is an optional method used by the framework for metrics gathering.

close

This method is used by the framework for cleanup after there are no more key/value pairs to process.

OutputFormat

Similarly, to an input format, Hadoop relies on the output format of the job for two main tasks:

1. Validate the output configuration for the job.
2. Create the RecordWriter implementation that will write the output of the job.

On the flip side of the FileInputFormat, there is a FileOutputFormat to work with file-based output. Because most output from a MapReduce job is written to HDFS, the many file-based output formats that come with the API will solve most of yours needs. The default used by Hadoop is the TextOutputFormat, which stores key/value pairs to HDFS at a configured output directory with a tab delimiter. Each reduce task writes an individual part file to the configured output directory. The TextOutputFormat also validates that the output directory does not exist prior to starting the MapReduce job.

The TextOutputFormat uses a LineRecordWriter to write key/value pairs for each map task or reduce task, depending on whether there is a reduce phase or not. This class uses the toString method to serialize each each key/value pair to a part file in HDFS, delimited by a tab. This tab delimiter is the default and can be changed via job configuration.

Again, much like an InputFormat, you are not restricted to storing data to HDFS. As long as you can write key/value pairs to some other source with Java (e.g., a JDBC database connection), you can use MapReduce to do a parallel bulk write. Just make sure whatever you are writing to can handle the large number of connections from the many tasks.

The OutputFormat abstract class contains three abstract methods for implementation:

checkOutputSpecs

This method is used to validate the output specification for the job, such as making sure the directory does not already exist prior to it being submitted. Otherwise, the output would be overwritten.

getRecordWriter

This method returns a RecordWriter implementation that serializes key/value pairs to an output, typically a FileSystem object.

getOutputCommiter

The output committer of a job sets up each task during initialization, commits the task upon successful completion, and cleans up each task when it finishes — successful or otherwise. For file-based output, a FileOutputCommitter can be used to handle all the heavy lifting. It will create temporary output directories for each map task and move the successful output to the configured output directory when necessary.

RecordWriter

The RecordWriter abstract class writes key/value pairs to a file system, or another output. Unlike its RecordReader counterpart, it does not contain an initialize phase. However, the constructor can always be used to set up the record writer for whatever is needed. Any parameters can be passed in during construction, because the record writer instance is created via OutputFormat.getRecordWriter.

The RecordWriter abstract class is a much simpler interface, containing only two methods:

write

This method is called by the framework for each key/value pair that needs to be written. The implementation of this method depends very much on your use case. The examples will write each key/value pair to an external in-memory key/value store rather than a file system.

close

This method is used by the framework after there are no more key/value pairs to write out. This can be used to release any file handles, shut down any connections to other services, or any other cleanup tasks needed.