Research Laboratory

Java Web Crawler & Hadoop MapReduce
Anri Morchiladze && Bachana Dolidze
Supervisor Nodar Momtselidze
1. Java Web Crawler
   ● Description
   ● Java Code
2. MapReduce
   ● Overview
   ● Example of mapreduce program
   ● Code & Run
   ● Walk-Through
Tasks, have to be developed – this semester

- Create Hadoop multycomputer (S) System.
- Install OS (Linux).
- Install and configure map/reduce, Hive, Hbase, Crawler.
- Develop programm to collect information from internet.
- Develop Hbase dictionary, and Hive database.
- Calculates number of needed word combination, using map/reduce technology, filters with the help of Hbase dictionary. Then summarize number of searched word combination and create database with Hive technology.
- Create Data Warehouse
1: class Mapper
2:   method Map(docid a, doc d)
3:     for all term t \in doc d do
4:       if term t \notin HBase
5:         Emit(term t, count 1)

1: class Reducer
2:   method Reduce(term t, counts [c1, c2, ...])
3:     sum \leftarrow 0
4:     for all count c \in counts [c1, c2, ...] do
5:       sum \leftarrow sum + c
6:     Emit(term t, count sum)
7: To Hive
A web crawler (also known as a web spider or web robot) is a program or automated script which browses the World Wide Web in a methodical, automated manner.

This process is called Web crawling or spidering.
Web crawlers are mainly used to create a copy of all the visited pages for later processing by a search engine, that will index the downloaded pages to provide fast searches.

Crawlers can also be used for automating maintenance tasks on a Web site, such as checking links or validating HTML code.

Also, crawlers can be used to gather specific types of information from Web pages, such as harvesting e-mail addresses (usually for spam).
Our Aim

1. Our aim is to search specified word in web pages.
2. We used library Jsoup for it and its commands.
   http://jsoup.org/
Java Code

1. Using Eclipse
2. Add a Jsoup library to project
3. You can change web site url or text word in this program and see results.
4. results are kept in a file
Hadoop MapReduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.

A MapReduce job usually splits the input data-set into independent chunks which are processed by the map tasks in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the reduce tasks. Typically both the input and the output of the job are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.
Overview

The MapReduce framework consists of a single master JobTracker and one slave TaskTracker per cluster-node. The master is responsible for scheduling the jobs' component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves execute the tasks as directed by the master.

The MapReduce framework operates exclusively on <key, value> pairs, that is, the framework views the input to the job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job, conceivably of different types.

Input and Output types of a MapReduce job:

(input) <k1, v1> -> map -> <k2, v2> -> combine -> <k2, v2> -> reduce -> <k3, v3> (output)
Example

Before we jump into the details, let's walk through an example MapReduce application to get a flavour for how they work.

WordCount is a simple application that counts the number of occurrences of each word in a given input set.

This works with a local-standalone, pseudo-distributed or fully-distributed Hadoop installation (Single Node Setup)
Usage

Assuming HADOOP_HOME is the root of the installation and HADOOP_VERSION is the Hadoop version installed, compile WordCount.java and create a jar:

$ mkdir wordcount_classes

$ javac -classpath ${HADOOP_HOME}/hadoop-${HADOOP_VERSION}-core.jar -d wordcount_classes WordCount.java

$ jar -cvf /usr/joe/wordcount.jar -C wordcount_classes/.

Assuming that:

- /usr/joe/wordcount/input - input directory in HDFS
- /usr/joe/wordcount/output - output directory in HDFS
Sample text-files as input:

$ bin/hadoop dfs -ls /usr/joe/wordcount/input/
/usr/joe/wordcount/input/file01
/usr/joe/wordcount/input/file02

$ bin/hadoop dfs -cat /usr/joe/wordcount/input/file01
Hello World Bye World

$ bin/hadoop dfs -cat /usr/joe/wordcount/input/file02
Hello Hadoop Goodbye Hadoop

Run the application:

$ bin/hadoop jar /usr/joe/wordcount.jar org.myorg.WordCount /usr/joe/wordcount/input /usr/joe/wordcount/output
$ bin/hadoop dfs -cat /usr/joe/wordcount/output/part-00000
Bye 1
Goodbye 1
Hadoop 2
Hello 2
World 2
Walk-Through

The WordCount application is quite straight-forward. The Mapper implementation (lines 14-26), via the map method (lines 18-25), processes one line at a time, as provided by the specified TextInputFormat (line 49). It then splits the line into tokens separated by whitespaces, via the StringTokenizer, and emits a key-value pair of < <word>, 1>.

For the given sample input the first map emits:
< Hello, 1>
< World, 1>
< Bye, 1>
< World, 1>

The second map emits:
< Hello, 1>
< Hadoop, 1>
< Goodbye, 1>
< Hadoop, 1>
WordCount also specifies a combiner (line 46). Hence, the output of each map is passed through the local combiner (which is same as the Reducer as per the job configuration) for local aggregation, after being sorted on the keys.

The output of the first map:

< Bye, 1>
< Hello, 1>
< World, 2>

The output of the second map:

< Goodbye, 1>
< Hadoop, 2>
< Hello, 1>
The Reducer implementation (lines 28-36), via the reduce method (lines 29-35) just sums up the values, which are the occurrence counts for each key (i.e. words in this example).

Thus the output of the job is:

< Bye, 1>
< Goodbye, 1>
< Hadoop, 2>
< Hello, 2>
< World, 2>

The run method specifies various facets of the job, such as the input/output paths (passed via the command line), key/value types, input/output formats etc., in the JobConf. It then calls the JobClient.runJob (line 55) to submit the and monitor its progress.
## Electrical Consumption

<table>
<thead>
<tr>
<th>Year</th>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
<th>Apr</th>
<th>May</th>
<th>Jun</th>
<th>Jul</th>
<th>Aug</th>
<th>Sep</th>
<th>Oct</th>
<th>Nov</th>
<th>Dec</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1979</td>
<td>23</td>
<td>23</td>
<td>2</td>
<td>43</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td>25</td>
<td>26</td>
<td>25</td>
</tr>
<tr>
<td>1980</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>28</td>
<td>30</td>
<td>31</td>
<td>31</td>
<td>31</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>29</td>
</tr>
<tr>
<td>1981</td>
<td>31</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>36</td>
<td>34</td>
<td>34</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>1984</td>
<td>39</td>
<td>38</td>
<td>39</td>
<td>39</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>40</td>
<td>39</td>
<td>38</td>
<td>38</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>1985</td>
<td>38</td>
<td>39</td>
<td>39</td>
<td>39</td>
<td>41</td>
<td>41</td>
<td>41</td>
<td>00</td>
<td>40</td>
<td>39</td>
<td>39</td>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>
If the above data is given as input, we have to write applications to process it and produce results such as finding the year of maximum usage, year of minimum usage, and so on. This is a walkover for the programmers with finite number of records. They will simply write the logic to produce the required output, and pass the data to the application written.

But, think of the data representing the electrical consumption of all the largescale industries of a particular state, since its formation.

When we write applications to process such bulk data,

- They will take a lot of time to execute.
- There will be a heavy network traffic when we move data from source to network server and so on.

To solve these problems, we have the MapReduce framework.
Thank you for pay attention!!!