**Reading Assignment – Big Data**

**Aparna PAvithran (axp161730)**

**1. Introduction:**

**a. What prompted the need for a new abstraction of parallel computation? What types of "messy details" does it hide from a programmer?**

The issues of, how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues. As a reaction to this complexity, they designed a new abstraction that allows expressing the simple computations trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library.

**2. Programming Model:**

**a. What are the inputs and outputs for the map and reduce functions? What is the function of the MapReduce library?**

The user writes **map** and it takes an input pair and produces a set of intermediate key/value pairs. The **Reduce** function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce invocation. The MapReduce library groups together all intermediate values associated with the same intermediate key I and pass them to the Reduce function.

**b. The paper mentions several examples where MapReduce model can be applied. The first one is the famous WordCount problem and the paper lists the map and reduce pseudo-code. Try to write pseudo-code for the remaining examples in a similar format.**

**Distributed Grep:** The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

**pseudo-code:**

map(String key, String value):

// key: document name

// value: document contents

for each line li in value:

if(li not matching the pattern)

EmitIntermediate(li);

reduce(String key):

// key: a line

Emit(key);

**Count of URL Access Frequency:** The map function processes logs of web page requests and outputs (URL, 1). The reduce function adds together all values for the same URL and emits a (URL, total count) pair.

**pseudo-code:**

map(String key, String value):

// key: logs file name

// value: file contents

for each URL u in value:

EmitIntermediate(u, "1");

reduce(String key, Iterator values):

// key: a URL

// values: a list of counts

int result = 0;

for each v in values:

result += ParseInt(v);

Emit(AsString(result));

**Reverse Web-Link Graph:** The map function outputs (target, source) pairs for each link to a target URL found in a page named source. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: (target, list(source))

**pseudo-code:**

map(String key, String value):

// key: document name

// value: document contents

for each URL u in value:

EmitIntermediate(URL, u);

reduce(String key, Iterator values):

// key: a URL

// values: a list of sources

String result;

for each v in values:

result += v;

Emit (result);

**Term-Vector per Host:** A term vector summarizes the most important words that occur in a document or a set of documents as a list of (word, frequency) pairs. The map function emits a (hostname, term vector) pair for each input document (where the hostname is extracted from the URL of the document). The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together; throwing away infrequent terms, and then emits a final (hostname, term vector) pair.

**pseudo-code:**

map(String key, String value):

// key: document name

// value: document contents

for each word w in value:

EmitIntermediate(w, "1");

reduce(String key, Iterator values):

// key: a word

// values: a list of counts

int result = 0;

for each v in values:

result += ParseInt(v);

Emit(AsString(result));

**Inverted Index:** The map function parses each document, and emits a sequence of (word, document ID) pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a (word, list(document ID)) pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

**pseudo-code:**

map(String key, String value):

// key: document id

// value: documents location

for each document doc in value:

for each word w in doc:

EmitIntermediate(w, key);

reduce(String key, Iterator value):

// key: a word

// values: a list of document ids

// sort(key,value) : sorts document ids

Emit(key, sort(key,value));

**Distributed Sort:** The map function extracts the key from each record, and emits a (key, record) pair. The reduce function emits all pairs unchanged.

**pseudo-code:**

map(String key, String value):

// key: document name

// value: records

for each record r in value:

k=find(r) //finds key of record r

EmitIntermediate(k, r);

reduce(String key, String value):

// key: a record key

// value: record

Emit(key,value)

**3.1. Implementation:**

**3.1. Execution Overview:**

**a. What is the role of the partitioning function in MapReduce?**

Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function.

**b. In MapReduce, where is the output of the Map task buffered? What happens if there is an overflow? What would happen if the node running the Map task fails? (Hint: Read points 3 and 4)**

The intermediate key/value pairs produced by the Map function are buffered in memory. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function.

Any map tasks completed by the worker are reset back to their initial idle state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to idle and becomes eligible for rescheduling. Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible.

**c. Where is the sorting of the intermediate keys done? Why is sorting necessary?**

When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together.

The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

**d. After the reduce phase, how many output files are produced? Generally, what is done with these files?**

After successful completion, the output of the mapreduce execution is available in the R output files (one per reduce task, with file names as specified by the user). Typically, users do not need to combine these R output files into one file – they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

**3.2. Execution Overview:**

**a. What type of data structures does the master store for each task? What information does the master convey to the reduce tasks?**

The master keeps several data structures. For each map task and reduce task, it stores the state (idle, in-progress, or completed), and the identity of the worker machine (for non-idle tasks).

For each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have in-progress reduce tasks.

**3.3. Fault Tolerance:**

**a. In case of node failure, which type of tasks (map or reduce) will need to be re-executed? Explain the reason.**

Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible.

**b. Why does large scale worker failure not affect MapReduce operation?**

MapReduce is resilient to large-scale worker failures. For example, during one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The MapReduce mastersimply re-executed the work done by the unreachable worker machines, and continued to make forward progress, eventually completing the MapReduce operation.

**3.4. Locality:**

**a. How does MR master try to preserve network bandwidth?**

We conserve network bandwidth by taking advantage of the fact that the input data is stored on the local disks of the machines that make up our cluster. GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines. The MapReduce master takes the location information of the input files into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task’s input data (e.g., on a worker machine that is on the same network switch as the machine containing the data). When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no network bandwidth.

**3.5. Task Granularity:**

**a. In practice, what are some good choices for M and R?**

The master must make O(M + R) scheduling decisions and keeps O(M ∗ R) state in memory. In practice, we tend to choose M so that each individual task is roughly 16 MB to 64 MB of input data and we make R a small multiple of the number of worker machines we expect to use. We often perform MapReduce computations with M = 200, 000 and R = 5, 000, using 2,000 worker machines.

**3.6. Backup Tasks:**

**a. What are stragglers? What is a general mechanism to alleviate this problem?**

Straggler: a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation.

When a MapReduce operation is close to completion, the master schedules backup executions of the remaining in-progress tasks. The task is marked as completed whenever either the primary or the backup execution completes. We have tuned this mechanism so that it typically increases the computational resources used by the operation by no more than a few percent.

**4. Refinements:**

**4.1. Partitioning Function:**

**a. What is the default partitioning function? When would you want to override it?**

A default partitioning function is provided that uses hashing (e.g. hash (key) mod R).

This tends to result in fairly well balanced partitions. In some cases, however, it is useful to partition data by some other function of the key.

For example, sometimes the output keys are URLs, and we want all entries for a single host to end up in the same output file. To support situations like this, the user of the MapReduce library can provide a special partitioning function. For example, using “hash(Hostname(urlkey)) mod R” as the partitioning function causes all URLs from the same host to end up in the same output file.

**4.2. Ordering Guarantees:**

**a. What guarantee is made by the framework about the keys arriving at a partition? How is this useful?**

Guarantee that within a given partition, the intermediate key/value pairs are processed in increasing key order. This ordering guarantee makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, or users of the output find it convenient to have the data sorted.

**4.3. Combiner Function:**

**a. What is the advantage of a combiner function? Where is it executed? When properties does the reduce function need to have before a combiner can be used?**

Combiner function that does partial merging of this data before it is sent over the network. The Combiner function is executed on each machine that performs a map task. In some cases, there is significant repetition in the intermediate keys produced by each map task, and the user specified Reduce function is commutative and associative.

**4.4. Input and Output Types:**

**a. If you are using a "text" mode input reader, what will be the input (K, V) values?**

For example, “text” mode input treats each line as a key/value pair: the key is the offset in the file and the value is the contents of the line.

**4.9. Counters:**

**a. What are some uses of counter objects?**

The MapReduce library provides a counter facility to count occurrences of various events. Users have found the counter facility useful for sanity checking the behavior of MapReduce operations.

**8. Conclusions:**

**a. According to the authors, what are the three main reasons why MapReduce programming model has been successful at Google?**

First, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing.

Second, a large variety of problems are easily expressible as MapReduce computations.

Third, we have developed an implementation of MapReduce that scales to large clusters of machines comprising thousands of machines. The implementation makes efficient use of these machine resources and therefore is suitable for use on many of the large computational problems encountered at Google.

**b. What are the three main things that the authors have learned from this work?**

First, restricting the programming model makes it easy to parallelize and distribute computations and to make such computations fault-tolerant. Second, network bandwidth is a scarce resource. A number of optimizations in our system are therefore targeted at reducing the amount of data sent across the network: the locality optimization allows us to read data from local disks, and writing a single copy of the intermediate data to local disk saves network bandwidth. Third, redundant execution can be used to reduce the impact of slow machines, and to handle machine failures and data loss.