Week # 13

All Whiteboard snaps are part of the Syllabus

Today's Topic

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

- Cluster software architecture for HDFS and MapReduce
- Linkage of HDFS and MapReduce
- MapReduce Programming Model and Execution Workflow
- Programming Examples

2 Programming Model

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: *Map* and *Reduce*.

Map, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the *Reduce* function.

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 intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

```
map      (k1,v1)          → list(k2,v2)
reduce   (k2,list(v2))    → list(v2)
```

I.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values.

Consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following 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));
```

2.1 Example

The `map` function emits each word plus an associated count of occurrences (just '1' in this simple example). The `reduce` function sums together all counts emitted for a particular word.

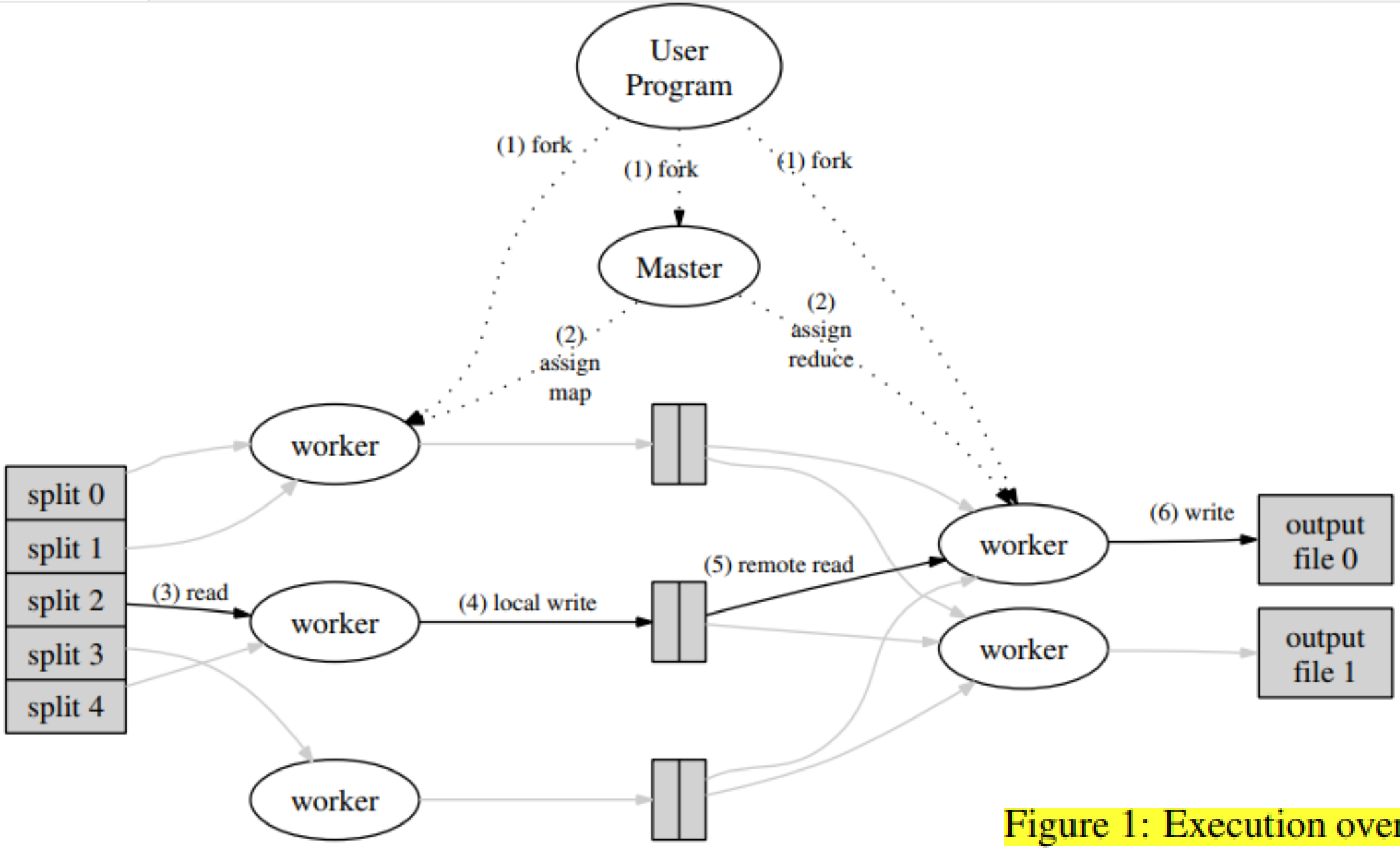
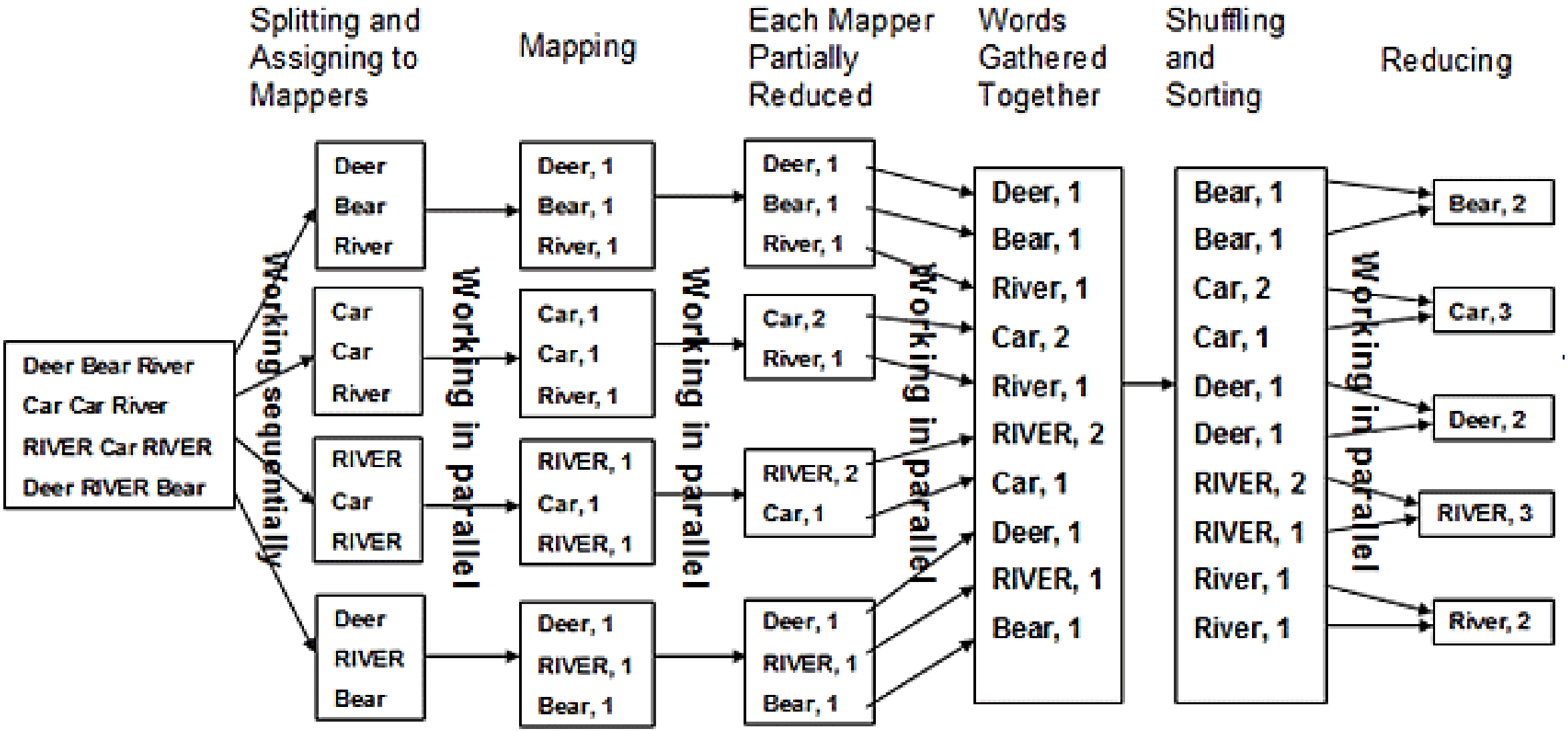


Figure 1: Execution overview

Word Count with four Mappers/Splits



```
#include "mapreduce/mapreduce.h"

// User's map function
class WordCounter : public Mapper {
 public:
  virtual void Map(const MapInput& input) {
    const string& text = input.value();
    const int n = text.size();
    for (int i = 0; i < n; ) {
      // Skip past leading whitespace
      while ((i < n) && isspace(text[i]))
        i++;

      // Find word end
      int start = i;
      while ((i < n) && !isspace(text[i]))
        i++;
      if (start < i)
        Emit(text.substr(start,i-start),"1");
    }
  }
};
REGISTER_MAPPER(WordCounter);
```

```
// User's reduce function
class Adder : public Reducer {
  virtual void Reduce(ReduceInput* input) {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }

    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};
REGISTER_REDUCER(Adder);
```

```
int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);

  MapReduceSpecification spec;

  // Store list of input files into "spec"
  for (int i = 1; i < argc; i++) {
    MapReduceInput* input = spec.add_input();
    input->set_format("text");
    input->set_filepattern(argv[i]);
    input->set_mapper_class("WordCounter");
  }

  // Specify the output files:
  //    /gfs/test/freq-00000-of-00100
  //    /gfs/test/freq-00001-of-00100
  //    ...
  MapReduceOutput* out = spec.output();
  out->set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  out->set_reducer_class("Adder");
```

main() …contd.

```
  // Optional: do partial sums within map
  // tasks to save network bandwidth
  out->set_combiner_class("Adder");

  // Tuning parameters: use at most 2000
  // machines and 100 MB of memory per task
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100);

  // Now run it
  MapReduceResult result;
  if (!MapReduce(spec, &result)) abort();

  // Done: 'result' structure contains info
  // about counters, time taken, number of
  // machines used, etc.

  return 0;
}
```

MapReduce Programming Problem

- Suppose a large web log is stored on a HDFS as CSV file. This file has the following fields per line: hostname, host IP address, timestamp, browser name (e.g. Chrome, Firefox, Edge), URL requested, amount of data sent.
- Write Map & Reduce C code, as in MapReduce paper, for the following problems:
  1. How much data is requested per unique URL? The output file contains URL and total data requested.
  2. Which host provided the URL to the user? The output file contains hostname and a string which contains URLs count concatenated with a list of all URLs per hostname.
  3. Count the number of hosts in the cluster. The MapReduce run should produce a single output file that contains the sentence "This cluster consists of 250 hosts", where 250 is the computed value.

```
//
//Q8(a):How much data is requested per unique URL? The output file contains URL and total data requested.
//
Map(const MapInput& input) {
   const string& str = input.value();
   const int n = text.size();

   string& p_key = get_key (str,"REQ_URL"); // string operation per line
   string& p_value = get_value (str,"DATA_SENT"); // strings operations per line
   Emit(p_key,p_value);
}
Reduce(ReduceInput* input) {
   // Iterate over all entries with the same key and concatenate the values
   int64 value = 0;

   while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
   }
   Emit(value); // Emit sum for input->key()
}
```

```
//
// Q8(b) Which host provided the URL to the user? The output file contains hostname and a list of URLs.
//
Map(const MapInput& input) {
   const string& str = input.value();
   const int n = text.size();

   string& p_key = get_key (str,"HOST"); // string operation per line
   string& p_value = get_value (str,"REQ_URL"); // strings operations per line
   Emit(p_key,p_value);
}
Reduce(ReduceInput* input) {
   // Iterate over all entries with the same key and concatenate the values. Also count
   string& p_value_list;
   int64 count = 0;
   while (!input->done()) {
      count++;
      p_value_list = p_value_list + "," + input->value();
      input->NextValue();
      p_value_list = IntToString(count) + " URLs. List: " + p_value_list;
   }
   Emit(p_value_list); // Emit sum for input->key()
}
```

```
//
//c)  Count the number of hosts in the cluster. The output file contains the sentence
//    "This cluster consists of 250 hosts", where 250 is the computed value.
//
Map(const MapInput& input) {
   const string& str = input.value();
   const int n = text.size();

   string& p_key = get_key (str,"HOST"); // string operation per line
   if check_valid_host(p_key) // check validity of host
      Emit("HOST",1); // for counting we ignore host name and use only "HOST" string
}
Reduce(ReduceInput* input) {  // As there is only one key HOST this is the only reducer in the system
   // Iterate over all entries with the same key and concatenate the values
   int64 value = 0;
   string& p_value_list;

   while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
   }
   p_value_list = "This cluster consist of " + IntToString(value) + "hosts";
   Emit(p_value_list); // Emit sum for input->key()
}
```