MapReduce

from the paper

MapReduce: Simplified Data Processing on Large Clusters (2004)

What it is

- MapReduce is a **programming model and an associated implementation for processing and generating large data sets**.

Users specify

- **a *map* function** that processes a key/value pair to generate a set of intermediate key/value pairs,
- **and a *reduce* function** that merges all intermediate values associated with the same intermediate key.

map and reduce functions

- map(String key, String value):
- reduce(String key, Iterator values)

- map (k1,v1) → list(**k2**,v2)
- reduce (**k2**,list(v2)) → list(v2)

sample problem: word counting

- counting the number of occurences of each word in a large collection of documents
- map(String key, String value):

```
// key: document name (k1=filename)
// value: document contents (v1)

for each word w (= k2) in value:
    EmitIntermediate(w, "1" (= v2) );
```

sample problem: word counting

- reduce(String key, String value):
  // key: word (= k2)
  // value: list of v2 = list of "1"
  for each word w (= k2) in value:
    EmitIntermediate(w, "1" (= v2));

sample problem: distributed grep

- grep (pattern matching according to a regular expression) in a large collection of documents
- map: k1 = filename
  k2 = line that matches pattern
  v2 = *filename + linenumber within file*
- reduce:
  output of
  k2
  list(v2)

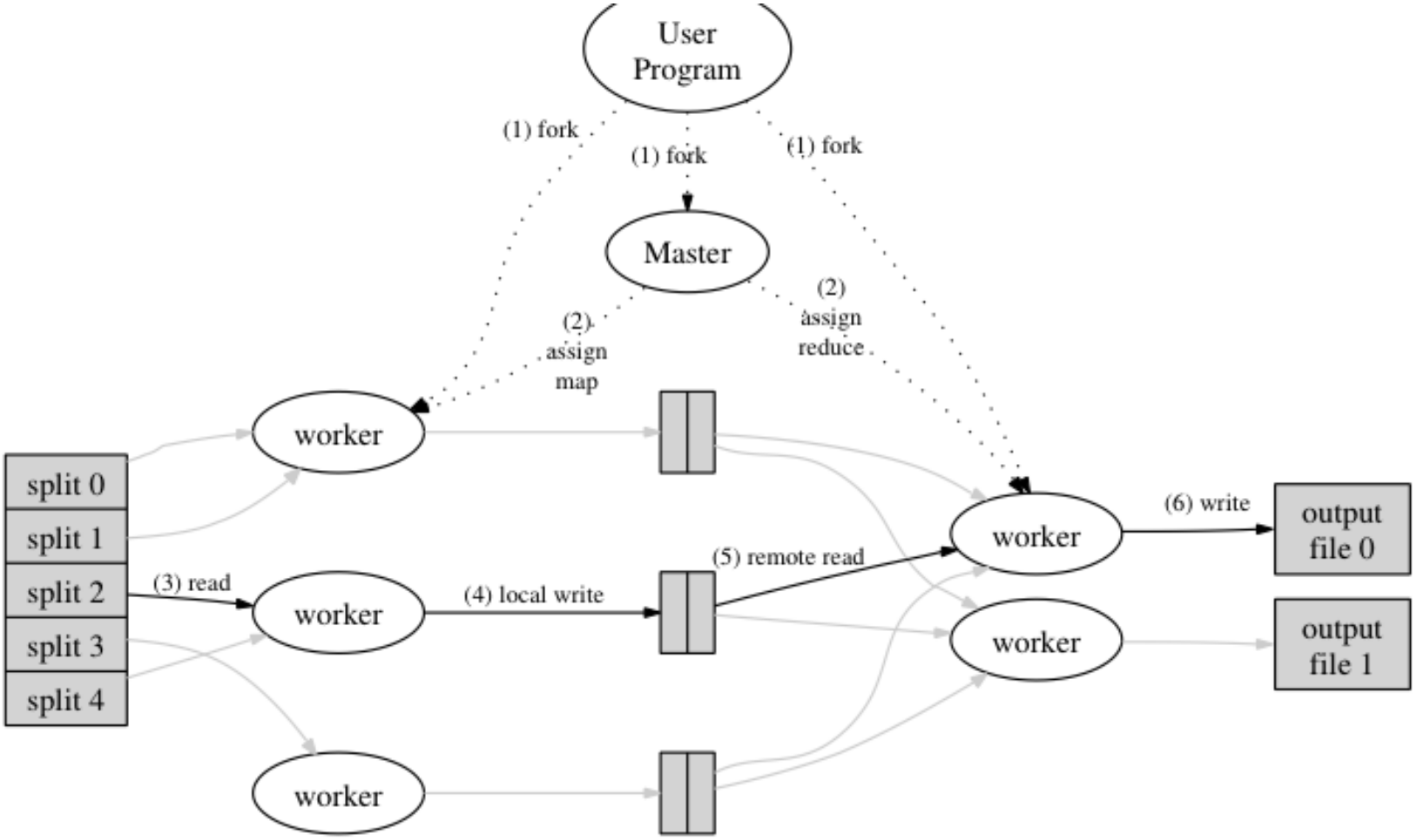
sample: URL access frequency

- counting the number of occurences of each URL in a large collection of documents which contain URL lists
- map: k1 = filename
  k2 = URL
  v2 = *"1"*
- reduce:
  for each k2: size (list(v2))

sample: reverse web link graph

- all references to a Web page, given a large collection of documents which contain <source-URL, target-URL> pairs
- map: k1 = filename
  k2 = target-URL
  v2 = source-URL
- reduce:
  for each k2: output list(v2)

execution of a map reduce process



execution of a map reduce process

- The MapReduce library in the user program first shards the **input files into M pieces of typically 16 megabytes to 64 megabytes (MB)** per piece. It then starts up many copies of the program on a cluster of machines.
- One of the copies of the program is special: the **master**. The rest are **workers that are assigned work by the master**. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.
- A **worker** who is assigned a map task reads the contents of the corresponding input split. It **parses key/value pairs out of the input data and passes each pair to the user-defined Map function**. 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. **The locations of these buffered pairs on the local disk are passed back to the master**, who is responsible for forwarding these locations to the reduce workers.

execution of a map reduce process

- When a **reduce worker** is notified by the master about these locations, it **uses remote procedure calls to read the buffered data from the local disks of the map workers**. 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**. If the amount of intermediate data is too large to fit in memory, an external sort is used.
- The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it **passes the key and the corresponding set of intermediate values to the user's Reduce function**. The output of the Reduce function is appended to a final output file for this reduce partition.
- **When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.**
- After successful completion, the output of the MapReduce execution is available in the R output files.

master data structures

- For each map task and reduce task:
  - the state (idle, *in-progress,* or *completed),* and
  - the identity of the worker machine (for non-idle tasks)
- for each completed map task:
  - the locations and sizes of the R intermediate file regions produced by the map task.

fault tolerance (1)

- worker failure:
  - **timeout check**: master pings every worker periodically; if no response is received from a worker in a certain amount of time, the master marks the worker as failed.
  - When a map task is executed first by worker A and then later executed by worker B (because A failed), **all workers executing reduce tasks are notified of the reexecution**. Any reduce task that has not already read the data from worker A will read the data from worker B.
- master failure:
  - **MapReduce computation aborted if the master fails.** Clients can check for this condition and retry the MapReduce operation if they desire.

fault tolerance (2)

- **“straggling” workers**: a machine takes an unusually long time to complete one of the map or reduce tasks in the computation.
- general mechanism to alleviate the problem of stragglers: 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.

fault tolerance (3)

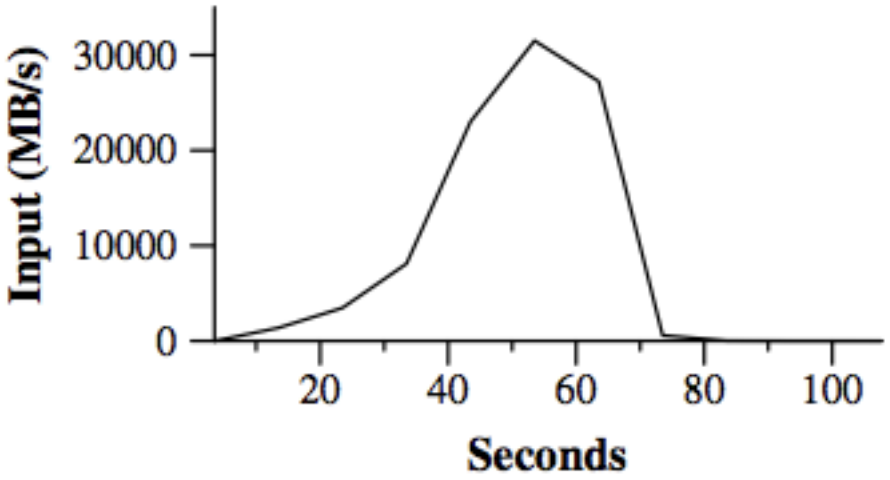
- **bad records**: sometimes there are bugs in user code that cause the *Map* or *Reduce* functions to crash deterministically on certain records
- sometimes it is acceptable to ignore a few records, for example when doing statistical analysis on a large data set.
- Each worker process installs a signal handler that catches segmentation violations and bus errors.
- when the master has seen more than one failure on a particular record, it indicates that the record should be skipped when it issues the next re-execution of the corresponding Map or Reduce task

optimizations

- **combiner function: i**n 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
  - thus, the intermediate keys should be processed locally
  - combiner function = reduce function except for output destination (intermediate file for combiner)
  - the *Combiner* function is executed on each machine that performs a map task

sample: distributed grep

- The *grep* program scans through 1010 100-byte records, searching for a relatively rare three-character pattern (the pattern occurs in 92,337 records). The input is split into approximately 64MB pieces (M = 15000), and the en- tire output is placed in one file (R = 1).



usage statistics 2004-2007

MapReduce statistics for different months

	Aug. 2004	Mar. 2006	Sep. 2007
Number of jobs (1000s)	29	171	2,217
Avg. completion time (secs)	634	874	395
Machine years used	217	2,002	11,081
`map` input data (TB)	3,288	52,254	403,152
`map` output data (TB)	758	6,743	34,774
`reduce` output data (TB)	193	2,970	14,018
Avg. machines per job	157	268	394
Unique implementations			
`map`	395	1,958	4,083
`reduce`	269	1,208	2,418

http://www.niallkennedy.com/blog/2008/01/google-mapreduce-stats.html