MapReduce: Simplified Data Processing on Large Clusters

wbg231

January 2023

1 Introduction

- a lot of big data applications need to be parallelized, but that used to involve writing a lot of user defined code for tasks that were pretty similar conceptually
- map reduce abstracts the parallelization by framing things in terms of a map and reduce structure
- apply a map to each record to produce intermediate key value pairs, apply a reduce to all values with a shared key.
- users can define the map and reduce functions
- this model allows for easy parallelization

programming model

- map written by user takes an input pair and produces a set of intermediate key/value pairs
- the reduce takes values with the same key and merges these values to form a potentially smaller set of values
- consider the task of counting the number of times each word appears in a large document

```
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));
```

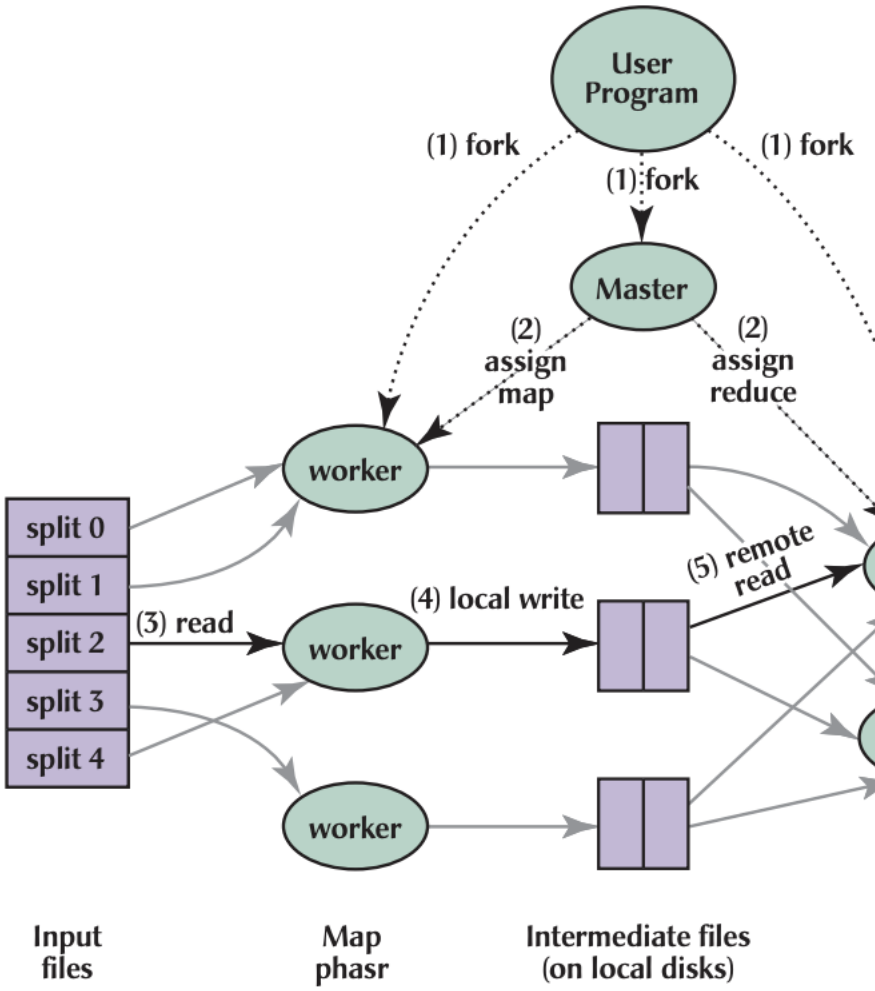
–

– in the above code the map function takes the bag of all words in the document and makes (word,"1") pairs for them

– the reduce function then groups by the key(word) and sums the count

implementation

- there are many implementations of map reduce, this is the one originally used at google
- the map calls are split across machines by breaking the input data into a set of M splits
- the input splits can be worked on in parallel by different machines
- reduce invocations are distributed across many machines by portioning the intermediate key space into R pieces using some partitioning function usually a hash
- the number of partitions (R) is set by the user, so is the partitioning function



- the steps are shown here

  1. map reduce splits the input data into M pieces
  2. one copy of the program is called the master it schedules work the rest are workers that run the map reduce function (the master has to deal with M map tasks and R reduce tasks) the master assigns all idle workers a map or reduce task
  3. a map worker reads data from the input split, parses key value pairs, passes those key value pairs to the map function to produce intermediate key value pairs that are the buffered into memory
  4. periodically the buffered pairs are written to local disk and partitioned into R regions by the partitioning function
  5. the reduce worker reads one of the R partitions of the intermediate

key value pairs from all the map worker nodes once all data for that partition is read, the reduce worker sorts by intermediate keys

6. the reduce worker than runs the reduce program on each unique intermediate key value, and then appends the output of the reduce function to a final output file.
7. when all map tasks and reduce task have been completed the master wakes up teh user program and ends

- if the program successfully completes the outputs are available in R output files (one for each partition)

master data structure

- master stores state (idle, in progress , done) and hte identity of each worker machine
- the master also stores where the values of each partition are for each map worker and then passes it to the reduce worker.

fault tolerance

- fault tolerance is a fact of life with this many machines and data
- worker tolerance
  - the master node pings workers periodically and if they don't respond for some time there state is set to failed
  - after a worker finishes there task there state is set to idle so they can be assigned one of the failed tasks.
  - all map tasks ran by a worker that fails must be re-run since workers store intermediate key value pairs locally
  - when a reducer fails you do not need to re do the reduce calls they have already done since those are written to a file
  - when a map tasks fails all reduce tasks that use data from that worker are told and update there data
  - this paradigm can handel a lot of worker failure
- there are cases where data will be sent to multiple workers, if this is the case the master only pays attention to the first piece of data it gets for this to work the map and reduce functions must be deterministic (this locality is really helpful for speed ups)
- map reduce master tries to have map tasks run on distributed machines that have the data they want to work with

backup tasks

- the program must wait for all map or reduce functions to finish before moving on so workers that take unusually long hold up the whole Processing
-