2.4 Extensions to Map-Reduce

Map-reduce has proved so influential that it has spawned a number of extensions and modifications. These systems typically share a number of characteristics with map-reduce systems:

1. They are built on a distributed file system.
2. They manage very large numbers of tasks that are instantiations of a small number of user-written functions.
3. They incorporate a method for dealing with most of the failures that occur during the execution of a large job, without having to restart that job from the beginning.

In this section, we shall mention some of the interesting directions being explored. References to the details of the systems mentioned can be found in the bibliographic notes for this chapter.

2.4.1 Workflow Systems

Two experimental systems called Clustera from the University of Wisconsin and Hyracks from the University of California at Irvine extend map-reduce from the simple two-step workflow (the Map function feeds the Reduce function) to any collection of functions, with an acyclic graph representing workflow among the functions. That is, there is an acyclic *flow graph* whose arcs $a \rightarrow b$ represent the fact that function a's output is input to function b. A suggestion of what a workflow might look like is in Fig. 2.6. There, five functions, f through j, pass data from left to right in specific ways, so the flow of data is acyclic and no task needs to provide data out before its input is available. For instance, function h takes ts input from a preexisting file of the distributed file system. Each of h's output elements is passed to at least one of the functions i and j.

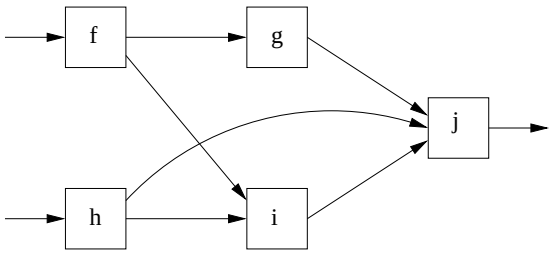

Figure 2.6: An example of a workflow that is more complex than Map feeding Reduce

In analogy to Map and Reduce functions, each function of a workflow can be executed by many tasks, each of which is assigned a portion of the input to the function. A master controller is responsible for dividing the work among the tasks that implement a function, usually by hashing the input elements to decide on the proper task to receive an element. Thus, like Map tasks, each task implementing a function f has an output file of data destined for each of the tasks that implement the successor function(s) of f. These files are delivered by the Master at the appropriate time – after the task has completed its work.

The functions of a workflow, and therefore the tasks, share with map-reduce tasks the important property that they only deliver output after they complete. As a result, if a task fails, it has not delivered output to any of its successors in the flow graph. A master controller can therefore restart the failed task at another compute node, without worrying that the output of the restarted task will duplicate output that previously was passed to some other task.

Many applications of workflow systems such as Clustera or Hyracks are cascades of map-reduce jobs. An example would be the join of three relations, where one map-reduce job joins the first two relations, and a second map-reduce job joins the third relation with the result of joining the first two relations. Both jobs would use an algorithm like that of Section 2.3.7.

There is an advantage to implementing such cascades as a single workflow. For example, the flow of data among tasks, and its replication, can be managed by the master controller, without need to store the temporary file that is output of one map-reduce job in the distributed file system. By locating tasks at compute nodes that have a copy of their input, we can avoid much of the communication that would be necessary if we stored the result of one map-reduce job and then initiated a second map-reduce job (although Hadoop and other map-reduce systems also try to locate Map tasks where a copy of their input is already present).

2.4.2 Recursive Extensions to Map-Reduce

Many large-scale computations are really recursions. An important example is PageRank, which is the subject of Chapter 5. That computation is, in simple terms, the computation of the fixedpoint of a matrix-vector multiplication. It is computed under map-reduce systems by the iterated application of the matrix-vector multiplication algorithm described in Section 2.3.1, or by a more complex strategy that we shall introduce in Section 5.2. The iteration typically continues for an unknown number of steps, each step being a map-reduce job, until the results of two consecutive iterations are sufficiently close that we believe convergence has occurred.

The reason recursions are normally implemented by iterated map-reduce jobs is that a true recursive task does not have the property necessary for independent restart of failed tasks. It is impossible for a collection of mutually recursive tasks, each of which has an output that is input to at least some of the other tasks, to produce output only at the end of the task. If they all followed that policy, no task would ever receive any input, and nothing could be accomplished. As a result, some mechanism other than simple restart of failed tasks must be implemented in a system that handles recursive workflows (flow graphs that are not acyclic). We shall start by studying an example of a recursion implemented as a workflow, and then discuss approaches to dealing with task failures.

Example 2.6: Suppose we have a directed graph whose arcs are represented by the relation $E(X, Y)$, meaning that there is an arc from node X to node Y.

We wish to compute the paths relation $P(X, Y)$, meaning that there is a path of length 1 or more from node X to node Y. A simple recursive algorithm to do so is:

1. Start with $P(X, Y) = E(X, Y)$.
2. While changes to the relation P occur, add to P all tuples in

$$\pi_{X,Y}\big(R(X, Z) \bowtie R(Z, Y)\big)$$

That is, find pairs of nodes X and Y such that for some node Z there is known to be a path from X to Z and also a path from Z to Y.

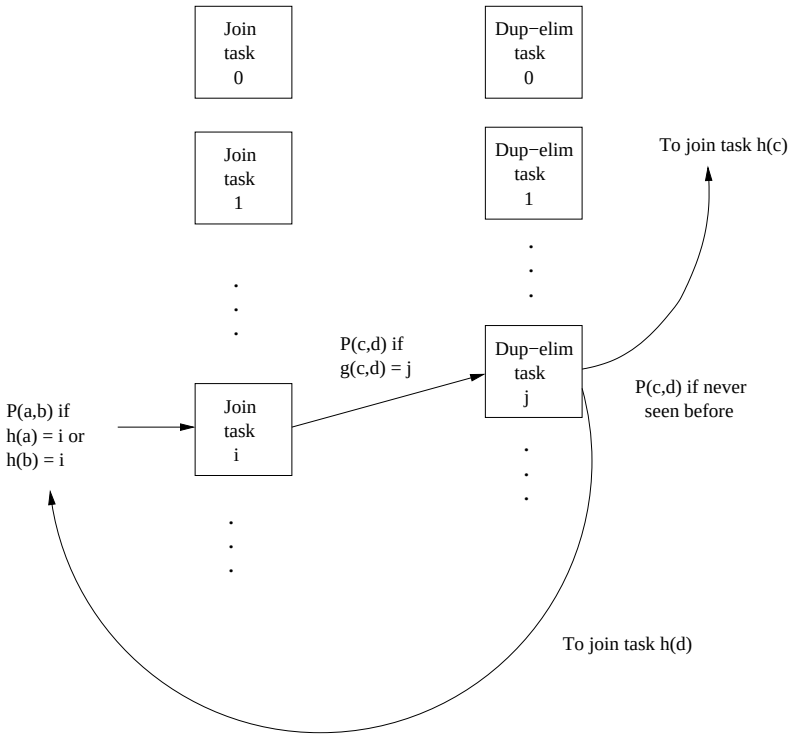

Figure 2.7: Implementation of transitive closure by a collection of recursive tasks

Figure 2.7 suggests how we could organize recursive tasks to perform this computation. There are two kinds of tasks: *Join tasks* and *Dup-elim tasks*. There are n Join tasks, for some n, and each corresponds to a bucket of a hash function h. A path tuple $P(a, b)$, when it is discovered, becomes input to two Join tasks: those numbered $h(a)$ and $h(b)$. The job of the ith Join task, when it receives input tuple $P(a, b)$, is to find certain other tuples seen previously (and stored locally by that task).

1. Store $P(a, b)$ locally.

2. If $h(a) = i$ then look for tuples $P(x, a)$ and produce output tuple $P(x, b)$.

3. If $h(b) = i$ then look for tuples $P(b, y)$ and produce output tuple $P(a, y)$.

Note that in rare cases, we have $h(a) = h(b)$, so both (2) and (3) are executed. But generally, only one of these needs to be executed for a given tuple.

There are also m Dup-elim tasks, and each corresponds to a bucket of a hash function g that takes two arguments. If $P(c, d)$ is an output of some Join task, then it is sent to Dup-elim task $j = g(c, d)$. On receiving this tuple, the jth Dup-elim task checks that it had not received it before, since its job is duplicate elimination. If previously received, the tuple is ignored. But if this tuple is new, it is stored locally and sent to two Join tasks, those numbered $h(c)$ and $h(d)$.

Every Join task has m output files – one for each Dup-elim task – and every Dup-elim task has n output files – one for each Join task. These files may be distributed according to any of several strategies. Initially, the $E(a, b)$ tuples representing the arcs of the graph are distributed to the Dup-elim tasks, with $E(a, b)$ being sent as $P(a, b)$ to Dup-elim task $g(a, b)$. The Master can wait until each Join task has processed its entire input for a round. Then, all output files are distributed to the Dup-elim tasks, which create their own output. That output is distributed to the Join tasks and becomes their input for the next round. Alternatively, each task can wait until it has produced enough output to justify transmitting its output files to their destination, even if the task has not consumed all its input. □

In Example 2.6 it is not essential to have two kinds of tasks. Rather, Join tasks could eliminate duplicates as they are received, since they must store their previously received inputs anyway. However, this arrangement has an advantage when we must recover from a task failure. If each task stores all the output files it has ever created, and we place Join tasks on different racks from the Dup-elim tasks, then we can deal with any single compute node or single rack failure. That is, a Join task needing to be restarted can get all the previously generated inputs that it needs from the Dup-elim tasks, and vice-versa.

In the particular case of computing transitive closure, it is not necessary to prevent a restarted task from generating outputs that the original task generated previously. In the computation of the transitive closure, the rediscovery of a path does not influence the eventual answer. However, many computations cannot tolerate a situation where both the original and restarted versions of a task pass the same output to another task. For example, if the final step of the computation were an aggregation, say a count of the number of nodes reached by each node in the graph, then we would get the wrong answer if we counted a path twice. In such a case, the master controller can record what files each task generated and passed to other tasks. It can then restart a failed task and ignore those files when the restarted version produces them a second time.

Pregel and Giraph

Like map-reduce, Pregel was developed originally at Google. Also like map-reduce, there is an Apache, open-source equivalent, called Giraph.

2.4.3 Pregel

Another approach to managing failures when implementing recursive algorithms on a computing cluster is represented by the Pregel system. This system views its data as a graph. Each node of the graph corresponds roughly to a task (although in practice many nodes of a large graph would be bundled into a single task, as in the Join tasks of Example 2.6). Each graph node generates output messages that are destined for other nodes of the graph, and each graph node processes the inputs it receives from other nodes.

Example 2.7: Suppose our data is a collection of weighted arcs of a graph, and we want to find, for each node of the graph, the length of the shortest path to each of the other nodes. Initially, each graph node a stores the set of pairs (b, w) such that there is an arc from a to b of weight w. These facts are initially sent to all other nodes, as triples (a, b, w).[6] When the node a receives a triple (c, d, w), it looks up its current distance to c; that is, it finds the pair (c, v) stored locally, if there is one. It also finds the pair (d, u) if there is one. If $w + v < u$, then the pair (d, u) is replaced by $(d, w + v)$, and if there was no pair (d, u), then the pair $(d, w + v)$ is stored at the node a. Also, the other nodes are sent the message $(a, d, w + v)$ in either of these two cases. □

Computations in Pregel are organized into *supersteps*. In one superstep, all the messages that were received by any of the nodes at the previous superstep (or initially, if it is the first superstep) are processed, and then all the messages generated by those nodes are sent to their destination.

In case of a compute-node failure, there is no attempt to restart the failed tasks at that compute node. Rather, Pregel *checkpoints* its entire computation after some of the supersteps. A checkpoint consists of making a copy of the entire state of each task, so it can be restarted from that point if necessary. If any compute node fails, the entire job is restarted from the most recent checkpoint.

Although this recovery strategy causes many tasks that have not failed to redo their work, it is satisfactory in many situations. Recall that the reason map-reduce systems support restart of only the failed tasks is that we want assurance that the expected time to complete the entire job in the face of failures is not too much greater than the time to run the job with no failures.

[6] This algorithm uses much too much communication, but it will serve as a simple example of the Pregel computation model.

Any failure-management system will have that property as long as the time to recover from a failure is much less than the average time between failures. Thus, it is only necessary that Pregel checkpoints its computation after a number of supersteps such that the probability of a failure during that number of supersteps is low.

2.4.4 Exercises for Section 2.4

! **Exercise 2.4.1:** Suppose a job consists of n tasks, each of which takes time t seconds. Thus, if there are no failures, the sum over all compute nodes of the time taken to execute tasks at that node is nt. Suppose also that the probability of a task failing is p per job per second, and when a task fails, the overhead of management of the restart is such that it adds $10t$ seconds to the total execution time of the job. What is the total expected execution time of the job?

! **Exercise 2.4.2:** Suppose a Pregel job has a probability p of a failure during any superstep. Suppose also that the execution time (summed over all compute nodes) of taking a checkpoint is c times the time it takes to execute a superstep. To minimize the expected execution time of the job, how many supersteps should elapse between checkpoints?