2.4 Extensions to MapReduce

MapReduce proved so influential that it spawned a number of extensions and modifications. These systems typically share a number of characteristics with MapReduce 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.

We begin this section with a discussion of "workflow" systems, which extend MapReduce by supporting acyclic networks of functions, each function implemented by a collection of tasks. While many such systems have been implemented (see the bibliographic notes for this chapter), an increasingly popular choice is UC Berkeley's Spark. Also gaining in importance is Google's TensorFlow. The latter, while not generally recognized as a workflow system because of its very specific targeting of machine-learning applications, in fact has a workflow architecture at heart.

Another family of systems uses a graph model of data. Computation occurs at the nodes of the graph, and messages are sent from any node to any adjacent node. The original system of this type was Google's Pregel, which has its own unique way of dealing with failures. But it has now become common to implement a graph-model facility on top of a workflow system and use the latter's file system and failure-management facility.

2.4.1 Workflow Systems

Workflow systems extend MapReduce 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 an input to function b.

The data passed from one function to the next is a file of elements of one type. If a function has a single input, then that function is applied to each input independently, just as Map and Reduce functions are applied to their input elements individually. The output of the function is a file collected from the result of applying the function to each input. If a function has inputs from more than one file, elements from each of the files can be combined in various ways. But the function itself is applied to combinations of input elements, at most one from each input file. We shall see examples of such combinations when we discuss the implementation of union and the relational join in Section 2.4.2.

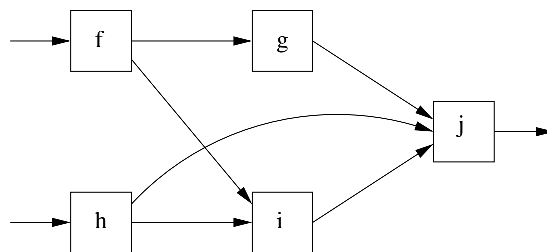

Figure 2.6: An example of a workflow that is more complex than Map feeding Reduce

Example 2.6 : 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 entire input is available. For instance, function h takes its input from a preexisting file of the distributed file system. Each of h's output elements is passed to the functions i and j, while i takes the outputs of both f and h as inputs. The output of j is either stored in the distributed file system or is passed to an application that invoked this dataflow. □

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, possibly 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 controller at the appropriate time – after the task has completed its work.

The functions of a workflow, and therefore the tasks, share with MapReduce tasks an important property: the *blocking property*, in 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.[7] 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.

Some applications of workflow systems are effectively cascades of MapReduce jobs. An example would be the join of three relations, where one MapReduce job joins the first two relations, and a second MapReduce 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 MapReduce 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 MapReduce job and then initiated a second MapReduce job (although Hadoop and other MapReduce systems also try to locate Map tasks where a copy of their input is already present).

2.4.2 Spark

Spark is, at its heart, a workflow system. However, it is an advance over the early workflow systems in several ways, including:

1. A more efficient way of coping with failures.

2. A more efficient way of grouping tasks among compute nodes and scheduling execution of functions.

3. Integration of programming language features such as looping (which technically takes it out of the acyclic workflow class of systems) and function libraries.

The central data abstraction of Spark is called the *Resilient Distributed Dataset*, or *RDD*. An RDD is a file of objects of one type. The primary example of an RDD that we have seen so far is the files of key-value pairs that are used in MapReduce systems. They are also the files that get passed among functions that we talked about in connection with Fig. 2.6. RDD's are "distributed" in the sense that an RDD is normally broken into chunks that may be held at

[7] As we shall discuss in Section 2.4.5, the blocking property only holds for acyclic workflows, and systems that support recursion cannot use it to manage failures.

different compute nodes. They are "resilient" in the sense that we expect to be able to recover from the loss of any or all chunks of an RDD. However, unlike the key-value-pair abstraction of MapReduce, there is no restriction on the type of the elements that comprise an RDD.

A Spark program consists of a sequence of steps, each of which typically applies some function to an RDD to produce another RDD. Such operations are called *transformations*. It is also possible to take data from the surrounding file system, such as HDFS, and turn it into an RDD, and to take an RDD and return it to the surrounding file system or to produce a result that is passed back to an application that called a Spark program. The latter kinds of operations are called *actions*.

We shall not try to list all the available transformations and actions that are available. Neither shall we fix on the dictions of a particular programming language, since the Spark operations are designed to be expressable in a number of different programming languages. However, here are some of the commonly used operations.

Map, Flatmap, and Filter

The Map transformation takes a parameter that is a function, and it applies that function to every element of an RDD, producing another RDD. This operation should remind us of the Map of MapReduce, but it is not exactly the same. First of all, in MapReduce, a Map function can only apply to a key-value pair. Second, in MapReduce, a Map function produces a set of key-value pairs, and each key-value pair is considered an independent element of the output of the Map function. In Spark, a Map function can apply to any object type, but it produces exactly one object as a result. The type of the resulting object can be a set, but that is not the same as producing many objects from one input object. If you want to produce a set of objects from a single object, Spark provides for you another transformation called *Flatmap*, which is analogous to Map of MapReduce, but without the requirement that all types be key-value pairs.

Example 2.7 : Suppose our input RDD is a file of documents, as in the "word-count" of Example 2.1. We could write a Spark Map function that takes one document and produces one set of pairs, with each pair of the form $(w, 1)$, where w is one of the words in the document. However, if we do so, then the output RDD is a list of sets, each set consisting of all the words of one document, each word paired with the integer 1. If we want to duplicate the Map function described in Example 2.1, then we need to use Spark's Flatmap transformation. That operation applied to the RDD of documents will produce another RDD, each of whose elements is a single pair $(w, 1)$. □

Spark also provides an operation similar to a limited form of Map, called *Filter*. Instead of a function as a parameter, the Filter transformation takes a predicate that applies to the type of objects in the input RDD. The predicate

returns true or false for each object, and the output RDD of a Filter transformation consists of only those objects in the input RDD for which the filter function returns true.

Example 2.8 : Continuing Example 2.7, suppose we want to avoid counting stop words: the most common words like "the" or "and." We could write a filter function that has built into it the list of words we want to eliminate. When applied to a pair $(w, 1)$, this function returns true if and only if w is not on the list. We can then write a Spark program that first applies Flatmap to the RDD of documents, producing an RDD R_1 consisting of a pair $(w, 1)$ for each occurrence of the word w in any of the documents. The program then applies the stop-word-eliminating Filter to R_1, producing another RDD, R_2. The latter RDD consists of a pair $(w, 1)$ for each occurrence of word w in any of the documents, but only if w is not a stop word. □

Reduce

In Spark, the Reduce operation is an action, not a transformation. That is, the operation Reduce applies to an RDD but returns a value and not another RDD. Reduce takes a parameter that is a function which takes two elements of some particular type T and returns another element of the same type T. When applied to an RDD whose elements are of type T, Reduce is applied repeatedly to each pair of consecutive elements, reducing them to a single element. When only one element remains, that becomes the result of the Reduce operation.

For example, if the parameter is the addition function, and this instance of Reduce is applied to an RDD whose elements are integers, then the result will be a single integer that is the sum of all the integers in the RDD. As long as the function parameter is an associative and commutative function, such as addition, it does not matter in which order elements of the input RDD are combined. However, it is also possible to use an arbitrary function, as long as we are satisfied with combination of elements in any order.

Relational Database Operations

There are a number of built-in Spark operations that behave like relational-algebra operators on relations that are represented by RDD's. That is, think of the elements of the RDD's as tuples of a relation. The transformation Join takes two RDD's, each representing one of the relations. The type of each RDD must be a key-value pair, and the key types of both relations must be the same. The Join transformation then looks for two objects, one from each of its input RDD's, such that the key values are the same, say (k, x) and (k, y). For each such pair found, Join produces the key-value pair $\big(k, (x, y)\big)$, and the output RDD consists of all such objects.

The group-by operation of SQL is also implemented in Spark by the transformation GroupByKey. This transformation takes as input an RDD whose type is key-value pairs. The output RDD is also a set of key-value pairs with

the same key type. The value type for the output is a list of values of the input type. GroupByKey sorts its input RDD by key and for each key k produces the pair $(k, [v_1, v_2, \ldots, v_n])$ such that the v_i's are all the values associated with key k in the input RDD. Notice that GroupByKey is exactly the operation that is performed behind the scenes by MapReduce in order to group the output of the Map function by key.

2.4.3 Spark Implementation

There are a number of ways that Spark implementation differs from Hadoop or other MapReduce implementations. We shall discuss two important improvements: lazy evaluation of RDD's and lineage for RDD's. Before we do, we should mention one way in which Spark is similar to MapReduce: the way large RDD's are managed.

Recall that when applying Map to a large file, MapReduce divides that file into chunks and creates a Map task for each chunk or group of chunks. The chunks and their tasks are typically distributed among many different compute nodes. Likewise, many Reduce tasks can run in parallel on different compute nodes, and each of these tasks takes a portion of the entire set of key-value pairs that are passed from Map to Reduce. Spark also allows any RDD to be divided into chunks, which it calls *splits*. Each split can be given to a different compute node, and the transformation on that RDD can be performed in parallel on each of the splits.

Lazy Evaluation

As mentioned in Section 2.4.1, it is common for workflow systems to exploit the blocking property for error handling. To do so, a function is applied to a single intermediate file (analogous to an RDD) and the output of that function is made available to consumers of that output only after the function completes. However, Spark does not actually apply transformations to RDD's until it is required to do so, typically because it must apply some action, e.g., storing a computed RDD in the surrounding file system or returning a result to an application.

The benefit of this strategy of *lazy evaluation* is that many RDD's are not constructed all at once. When one split of an RDD is created at a node, it may be used immediately at the same compute node to apply another transformation. The benefit of this startegy is that this RDD is never stored on disk and never transmitted to another compute nodes, thus saving orders of magnitude in running time in some cases.

Example 2.9 : Consider the situation suggested in Example 2.8, where Flatmap is applied to one RDD, which we shall refer to as R_0. Note that RDD R_0 is created by converting the external file of documents into an RDD. As R_0 is a large file, we shall want to divide it into splits and operate on the splits in parallel.

The first transformation on R_0 applies Flatmap to create a set of pairs $(w, 1)$ for each word. For each split of R_0, a split of the resulting RDD, which we called R_1 in Example 2.8, is created at the same compute node. This split of R_1 is then passed to the transformation Filter, which eliminates pairs whose first component is a stop word. When this Filter is applied to the split, the result is a split of the RDD R_2, located at the same compute node.

However, neither the Flatmap nor Filter transformations occur unless an action is applied to R_2. For example, the Spark program may store R_2 in the surrounding file system or perform a Reduce operation that counts occurrences of the words. Only when the program reaches this action does Spark apply the Flatmap and Filter transformations to R_0, running these transformations at each of the compute nodes that holds a split of R_0, in parallel. Thus, the splits of R_1 and R_2 exist only locally at the compute node that created them, and unless the programmer explicitly calls for them to be maintained, these splits are dropped as soon as they are used locally. □

Resilience of RDD's

One may naturally ask what happens in Example 2.9 if a compute node fails after creating a split of R_1 and before transforming that split into a split of R_2. Since R_1 is not backed up to the file system, is it not lost forever? Spark's substitute for redundant storage of intermediate values is to record the *lineage* of every RDD it creates. The lineage tells the Spark system how to recreate the RDD, or a split of the RDD, if that is needed.

Example 2.10 : Considering again the situation described in Example 2.9, the lineage for R_2 would say that it is created by applying to R_1 the particular Filter operation that eliminates stop words. In turn, R_1 is created from R_0 by the Flatmap operation that turns words of a document into $(w, 1)$ pairs. And R_0 was created from a particular file of the surrounding file system.

For instance, if we lose a split of R_2, we know we can reconstruct it from the corresponding split of R_1. But since that split exists at the same compute node, we've probably lost that split also. If so, we could reconstruct it from the corresponding split of R_0, which is also probably lost if this compute node has failed. But we know that we can reconstruct the split of R_0 from the surrounding file system, which is presumably redundant and will not be lost. Thus, Spark will find another compute node, reconstruct the lost split of R_0 from the file system there, and then apply the known transformations needed to reconstruct the corresponding splits of R_1 and R_2. □

As we can see from Example 2.10, recovery from a node failure can be more complex in Spark than in MapReduce or in workflow systems that store intermediate values redundantly. However, the tradeoff of more complex recovery when things go wrong against greater speed when things go right is generally a good one. The faster a Spark program runs, the less chance there is of a node failure while running.

We should contrast Spark's need to be able to execute a program in the face of failures with the need for redundant storage of files that are expected to exist for a long period. Over a long period, failures are almost certain, so we are very likely to lose pieces of a file if we do not store it redundantly. But over a short period – minutes or even hours – there is a good chance of avoiding failures. Thus, it is reasonable to be willing to pay more when there *is* a failure in this case.

2.4.4 TensorFlow

TensorFlow is an open-source system developed initially at Google to support machine-learning applications. Like Spark, TensorFlow provides a programming interface in which one writes a sequence of steps. Programs are typically acyclic, although like Spark it is possible to iterate blocks of code.

One major difference between Spark and TensorFlow is the type of data that is passed between steps of the program. In place of the RDD, TensorFlow uses *tensors*; a tensor is simply a multidimensional matrix.

Example 2.11: A constant, e.g. 3.14159, is regarded as a 0-dimensional tensor. A vector is a 1-dimensional tensor. For instance, the vector $(1, 2, 3)$ can be written in tensorFlow as $[1., 2., 3.]$. A matrix is a 2-dimensional tensor. For example, the matrix

$$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{matrix}$$

is expressed as $[[1., 2., 3., 4.],\ [5., 6., 7., 8.],\ [9., 10., 11., 12.]]$.

Higher-dimensional arrays are possible as well. For instance, a 2-by-2-by-2 cube of 0's is represented as $[[[0., 0.],\ [0., 0.]],\ \ [[0., 0.],\ [0., 0.]]]$. □

Although tensors are in fact a restricted form of RDD, the power of TensorFlow comes from its selection of built-in operations. Linear algebra operations are available as functions. For example, if you want matrix C to be the product of matrices A and B, you can write

```
C = tensorflow.matmul(A,B)
```

Even more powerful are the common approaches to machine learning that are built in as operations. a single statement in the TensorFlow language can cause a model that is a tensor to be constructed from training data, which is also represented as a tensor, using a method like gradient descent. (We discuss gradient descent in Sections 9.4.5 and 12.3.4).

2.4.5 Recursive Extensions to MapReduce

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 MapReduce 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 MapReduce job, until the results of two consecutive iterations are sufficiently close that we believe convergence has occurred. A second important example of a recursive algorithm on massive data is gradient descent, which we just mentioned in connection with TensorFlow.

Recursions present a problem for failure recovery. Recursive tasks inherently lack the blocking 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 2.12 : 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. That is, P is the *transitive closure* of E. 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

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

   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 known path from Z to Y.

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.

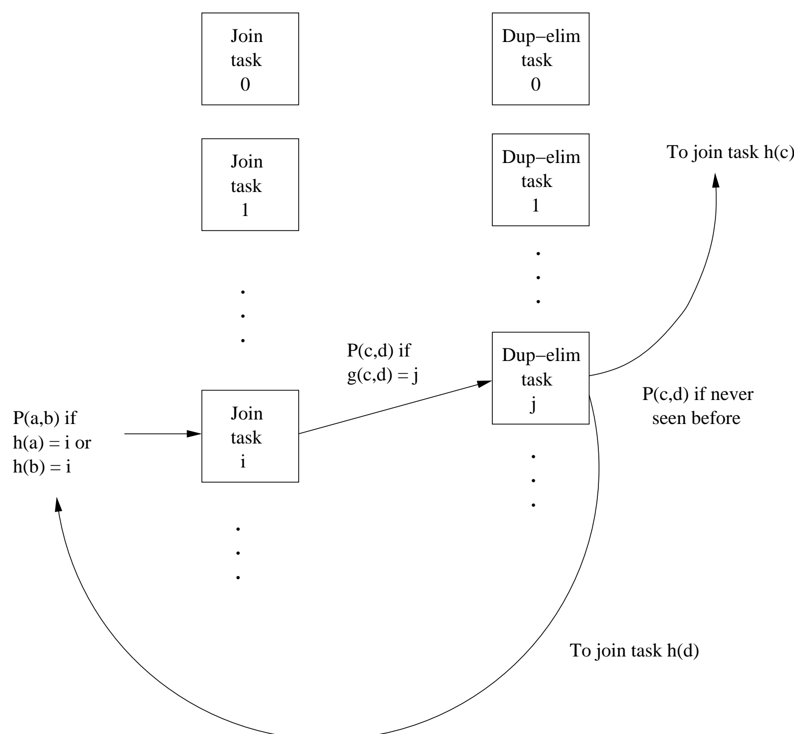

Figure 2.7: Implementation of transitive closure by a collection of recursive tasks

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 has not received this tuple 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 controller

waits 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. □

In Example 2.12 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.

There are at least three different approaches that have been used to deal with failures while executing a recursive program.

1. *Iterated MapReduce*: Write the recursion as repeated execution of a MapReduce job or of a sequence of MapReduce jobs. We can then rely on the failure mechanism of the MapReduce implementation to handle failures at any step. The first example of such a system was HaLoop (see the bibliographic notes for this chapter).

2. *The Spark Approach*: The Spark language actually includes iterative statements, such as for-loops that allow the implementation of recursions. Here, failure management is implemented using the lazy-evaluation and lineage mechanisms of Spark. In addition, the Spark programmer has options to store intermediate states of the recursion.

3. *Bulk-Synchronous Systems*: These systems use a graph-based model of computation that we shall describe next. They typically use another resilience approach: periodic checkpointing.

2.4.6 Bulk-Synchronous Systems

Another approach to implementing recursive algorithms on a computing cluster is represented by the Google's Pregel system, which was the first example of a

graph-based, bulk-synchronous system for processing massive amounts of data. Such a 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 2.12). 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 2.13 : 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. As the algorithm executes, each node a will store a set of pairs (b, w), where w is the length of the shortest path from node a to node b that is currently known.

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 sent to all other nodes, as triples (a, b, w), with the intended meaning that node a knows about a path of length w to node b.[8] When the node a receives a triple (c, d, w), it must decide whether this fact implies a shorter path than a already knows about from itself to node d. Node a 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. It is this packaging of many messages into one that gives this approach the name "bulk-synchronous."

There is a very important advantage to grouping messages in this way. Communication over a network generally requires a large amount of overhead to send any message, however short. Suppose that in Example 2.13 we sent a single new shortest-distance fact to the relevant node every time one was discovered. The number of messages sent would be enormous if the graph was large, and it would not be realistic to implement such an algorithm. However, in a bulk-synchonous system, a task that has the responsibility for managing many nodes of the graph can bundle together all the messages being sent by its nodes to any of the nodes being managed by another task. That choice typically saves orders of magnitude in the time required to send all the needed messages.

Failure Management in Pregel

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

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

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 MapReduce 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. 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.

2.4.7 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?