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. $\square$

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. $\square$

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.