1.1 Lecture Summary

1.1 Introduction to MapReduce

Lecture Summary: In this lecture, we learned the MapReduce paradigm, which is a pattern of parallel functional programming that has been very successful in enabling "big data" computations.

The input to a MapReduce style computation is a set of *key-value pairs*. The keys are similar to keys used in hash tables, and the functional programming approach requires that both the keys and values be immutable. When a user-specified *map function*, f, is applied on a key-value pair, (kA,vA), it results in a (possibly empty) set of output key-value pairs, {(kA1,vA1), (kA2,vA2),....} This map function can be applied in parallel on all key-value pairs in the input set, to obtain a set of *intermediate* key-value pairs that is the union of all the outputs.

The next operation performed in the MapReduce workflow is referred to as *grouping*, which groups together all intermediate key-value pairs with the same key. Grouping is performed automatically by the MapReduce framework, and need not be specified by the programmer. For example, if there are two intermediate key- value pairs, (kA1, vA1) and (kB1, vB1) with the same key, kA1 = kB1 =k, then the output of grouping will associate the set of values {vA1,vB1} with key k.

Finally, when a user-specified *reduce function*, g, is applied on two or more grouped values (e.g., vA1, vB1,...) associated with the same key k, it folds or reduces all those values to obtain a single *output key-value pair*, (k, g(vA1, vB1, . . .)), for each key, k. in the intermediate key-value set. If needed, the set of output key-value pairs can then be used as the input for a successive MapReduce computation.

In the example discussed in the lecture, we assumed that the map function, f, mapped a key-value pair like (“WR”,10) to a set of intermediate key-value pairs obtained from factors of 10 to obtain the set, { (“WR”,2), (“WR”,5), (“WR”,10) }, and the reduce function, g, calculated the sum of all the values with the same key to obtain (“WR”,17) as the output key-value pair for key “WR”. The same process can be performed in parallel for all keys to obtain the complete output key-value set.

Optional Reading:

1. Wikipedia article on the MapReduce framework

1.2 Apache Hadoop Project

Lecture Summary: The *Apache Hadoop* project is a popular open-source implementation of the Map-Reduce paradigm for distributed computing. A distributed computer can be viewed as a large set of multicore computers connected by a network, such that each computer has multiple processor cores, e.g., P0, P1, P2, P3, ... Each individual computer also has some *persistent storage* (e.g., hard disk, flash memory), thereby making it possible to store and operate on large volumes of data when

aggregating the storage available across all the computers in a data center. The main motivation for the Hadoop project is to make it easy to write large-scale parallel programs that operate on this “big data”.

The Hadoop framework allows the programmer to specify map and reduce functions in Java, and takes care of all the details of generating a large number of map tasks and reduce tasks to perform the computation as well as scheduling them across a distributed computer. A key property of the Hadoop framework is that it supports automatic *fault-tolerance*. Since MapReduce is essentially a functional programming model, if a node in the distributed system fails, the Hadoop scheduler can reschedule the tasks that were executing on that node with the same input elsewhere, and continue computation. This is not possible with non-functional parallelism in general, because when a non-functional task modifies some state, re-executing it may result in a different answer. The ability of the Hadoop framework to process massive volumes of data has also made it a popular target for higher-level query languages that implement SQL-like semantics on top of Hadoop.

The lecture discussed the *word-count* example, which, despite its simplicity, is used very often in practice for document mining and text mining. In this example, we illustrated how a Hadoop map-reduce program can obtain word-counts for the distributed text “To be or not to be”. There are several other applications that have been built on top of Hadoop and other MapReduce frameworks. The main benefit of Hadoop is that it greatly simplifies the job of writing programs to process large volumes of data available in a data center.

Optional Reading:

1. Wikipedia article on the Apache Hadoop project

1.3 Lecture Summary

1.3 Apache Spark Framework

Lecture Summary: *Apache Spark* is a similar, but more general, programming model than Hadoop MapReduce. Like Hadoop, Spark also works on distributed systems, but a key difference in Spark is that it makes better use of in- memory computing within distributed nodes compared to Hadoop MapReduce. This difference can have a significant impact on the performance of *iterative MapReduce* algorithms since the use of memory obviates the need to write intermediate results to external storage after each map/reduce step. However, this also implies that the size of data that can be processed in this manner is limited by the total size of memory across all nodes, which is usually much smaller than the size of external storage. (Spark can spill excess data to external storage if needed, but doing so reduces the performance advantage over Hadoop.)

Another major difference between Spark and Hadoop MapReduce, is that the primary data type in Spark is the *Resilient Distributed Dataset* (RDD), which can be viewed as a generalization of sets of key-value pairs. RDDs enable Spark to support more general operations than map and reduce. Spark supports intermediate operations called *Transformations* (e.g., map,filter,join,...) and terminal

operations called Actions (e.g., reduce,count,collect,...). As in Java streams, intermediate transformations are performed *lazily*, i.e., their evaluation is postponed to the point when a terminal action needs to be performed.

In the lecture, we saw how the Word Count example can be implemented in Spark using Java APIs. (The Java APIs use the same underlying implementation as Scala APIs, since both APIs can be invoked in the same Java virtual machine instance.) We used the Spark flatMap() method to combine all the words in all the lines in an input file into a single RDD, followed by a mapToPair() Transform method call to emit pairs of the form, (*word, 1*), which can then be processed by a reduceByKey() operation to obtain the final word counts.

Optional Reading:

1. Spark Programming Guide

2. Wikipedia article on the Apache Spark project

1.4 Lecture Summary

1.4 TF-IDF Example

Lecture Summary: In this lecture, we discussed an important statistic used in information retrieval and document mining, called *Term Frequency – Inverse Document Frequency (TF-IDF)*. The motivation for computing TF-IDF statistics is to efficiently identify documents that are most similar to each other within a large corpus.

Assume that we have a set of N documents D1,D2,…,DN, and a set of terms TERM1,TERM2,… that can appear in these documents. We can then compute total frequencies TFi,j for each term TERMi in each document Dj . We can also compute the document frequencies DF1,DF2,... for each term, indicating how many documents contain that particular term, and the inverse document frequencies (IDF):IDFi=N/DFi. The motivation for computing inverse document frequencies is to determine which terms are common and which ones are rare, and give higher weights to the rarer terms when searching for similar documents. The weights are computed as: Weight(TERMi,Dj)=TFi,j x log(N/DFi).

Using MapReduce, we can compute the TFi,j values by using a MAP operation to find all the occurrences of TERMi in document Dj, followed by a REDUCE operation to add up all the occurrences of TERMi as key-value pairs of the form, ((Dj,TERMi),TFi,j) (as in the Word Count example studied earlier). These key-value pairs can also be used to compute DFi values by using a MAP operation to identify all the documents that contain TERMi and a REDUCE operation to count the number of documents that TERMi appears in. The final weights can then be easily computed from the TFi,j and DFi values. Since the TF−IDF computation uses a fixed (not iterative) number of MAP and REDUCE operations, it is a good candidate for both Hadoop and Spark frameworks.

Optional Reading:

1. Wikipedia article on the Term Frequency – Inverse Document Frequency (TF-IDF) statistic

1.5 Lecture Summary

1.5 PageRank Example

Lecture Summary: In this lecture, we discussed the *PageRank* algorithm as an example of an iterative algorithm that is well suited for the Spark framework. The goal of the algorithm is to determine which web pages are more important by examining links from one page to other pages. In this algorithm, the rank of a page, B, is defined as follows,

where SRC(B) is the set of pages that contain a link to B, while DEST_COUNT(A) is the total number of pages that A links to. Intuitively, the PageRank algorithm works by splitting the weight of a page A (i.e., RANK(A)) among all of the pages that A links to (i.e. DEST_COUNT(A)). Each page that A links to has its own rank increased proportional to A's own rank. As a result, pages that are linked to from many highly-ranked pages will also be highly ranked.

The motivation to divide the contribution of A in the sum by DEST_COUNT(A) is that if page A links to multiple pages, each of the successors should get a fraction of the contribution from page A. Conversely, if a page has many outgoing links, then each successor page gets a relatively smaller weightage, compared to pages that have fewer outgoing links. This is a recursive definition in general, since if (say) page X links to page Y, and page Y links to page X, then RANK(X) depends on RANK(Y) and vice versa. Given the recursive nature of the problem, we can use an iterative algorithm to compute all page ranks by repeatedly updating the rank values using the above formula, and stopping when the rank values have converged to some acceptable level of precision. In each iteration, the new value of RANK(B) can be computed by accumulating the contributions from each predecessor page, A. A parallel implementation in Spark can be easily obtained by implementing two steps in an iteration, one for computing the contributions of each page to its successor pages by using the flatMapToPair() method, and the second for computing the current rank of each page by using the reduceByKey() and mapValues() methods.. All the intermediate results between iterations will be kept in main memory, resulting in a much faster execution than a Hadoop version (which would store intermediate results in external storage).

Optional Reading:

1. Wikipedia article on the PageRank algorithm

Mini Project 1: Page Rank with Spark

---

miniproject_1.zip

Project Goals and Outcomes

This week we learned about two different implementations of the MapReduce programming abstraction: Hadoop and Spark.

Hadoop was one of the first popular distributed MapReduce programming systems, and really paved the way for Big Data analysis at massive scales on commodity hardware. As described in lectures this week, its primary strengths are three-fold: 1) scalability, 2) programmability, and 3) fault tolerance. Hadoop allows programmers to write simple Java applications that run across thousands of networked machines and which, thanks to its MapReduce abstractions, can recover from complete hardware failure of many of those machines.

Spark, on the other hand, is a successor to Hadoop that emphasizes in-memory caching of data and a more flexible API. While Hadoop regularly persisted replicated data to disk to ensure fault tolerance, Spark instead caches information in memory and ensures that it remembers the transformations that were applied to generate a particular piece of data. Spark also offers more than simple map and reduce transformations.

You also learned about the PageRank algorithm, and how it can be used to rank inter-connected websites based on existing ranks and link counts.

In this mini-project, you will gain experience with Spark by implementing the PageRank algorithm using Spark transformations. In particular, this assignment will ask you to implement the transformations needed to complete a single iteration of the iterative PageRank algorithm given an input Spark Resilient Distributed Dataset (RDD) of websites.

Recall from lectures this week that an important concept in MapReduce frameworks is that of "key-value pairs". The **key** of a key-value pair is some unique identifier for some logical object, e.g. a website. The **value** is some metadata associated with the logical object the key identifies, e.g. information on the outbound links for a website. In Spark, a key-value pair is represented using the Tuple2 class. For example, to create a new key-value pair from some key k and some value v you can use the following code:

Spark uses special-purpose RDD objects to store datasets containing Tuple2 objects (i.e. key-value pairs). Rather than using the standard JavaRDD class, RDDs of Tuple2 objects use the JavaPairRDD class.

Project Setup

Please refer to Mini-Project 0 for a description of the build and testing process used in this course.

Once you have downloaded and unzipped the project files using the gray button labeled miniproject_1.zip at the top of this description, you should see the project source code file at

miniproject_1/src/main/java/edu/coursera/distributed/PageRank.java

and the project tests in

miniproject_1/src/test/java/edu/coursera/distributed/SparkTest.java

It is recommended that you review the demo video for this week before starting this assignment, in which Professor Sarkar works through a similar example.

Project Instructions

Your modifications should be made entirely inside of PageRank.java. You should not change the signatures of any public or protected methods inside of PageRank.java, but you can edit the method bodies and add any new methods that you choose. We will use our copy of SparkTest.java in the final grading process, so do not change that file or any other file except PageRank .java.

Your main goals for this assignment are as follows:

1. Inside PageRank.java, implement the sparkPageRank method to perform a single iteration of the PageRank algorithm using the Spark Java APIs.

There are helpful TODOs in PageRank.java to help guide your implementation.

Project Evaluation

Your assignment submission should consist of only the PageRank.java file that you modified to implement this mini-project. As before, you can upload this file through the assignment page for this mini-project. After that, the Coursera autograder will take over and assess your submission, which includes building your code and running it on one or more tests. Your submission will be evaluated on Coursera's auto-grading system using 2 and 4 CPU cores. Note that the performance observed for tests on your local machine may differ from that on Coursera's auto-grading system, but that you will only be evaluated on the measured performance on Coursera. Also note that for all assignments in this course you are free to resubmit as many times as you like. See the Common Pitfalls page under Resources for more details. Please give it a few minutes to complete the grading. Once it has completed, you should see a score appear in the "Score" column of the "My submission" tab based on the following rubric:

- 10% - performance of testUniformThirtyThousand on 2 cores
- 10% - performance of testUniformOneHundredThousand on 2 cores
- 5% - performance of testIncreasingThirtyThousand on 2 cores
- 5% - performance of testIncreasingOneHundredThousand on 2 cores
- 10% - performance of testRandomThirtyThousand on 2 cores
- 10% - performance of testRandomOneHundredThousand on 2 cores
- 10% - performance of testUniformThirtyThousand on 4 cores
- 10% - performance of testUniformOneHundredThousand on 4 cores
- 5% - performance of testIncreasingThirtyThousand on 4 cores
- 5% - performance of testIncreasingOneHundredThousand on 4 cores
- 10% - performance of testRandomThirtyThousand on 4 cores
- 10% - performance of testRandomOneHundredThousand on 4 cores