The Korean Journal of Applied Statistics (2016) DOI: http://dx.doi.org/10.5351/KJAS.2016.27.1.001
27(1), 1–14

Processing Large-Scale Data with Apache Spark

Seyoon Ko[a] · Joong-Ho Won[a,1]

[a]Department of Statistics, Seoul National University

(Received August 14, 2016; Revised 00, 0000; Accepted October 03, 2016)

Abstract

Apache Spark is a fast and general-purpose cluster computing package. It provides a new abstraction named resilient distributed dataset, which is capable of support for fault tolerance while keeping the data in memory. This type of abstraction results in a significant speedup compared to legacy large-scale data framework, MapReduce. In particular, Spark framework is suitable for iterative machine learning applications such as logistic regression and K-means clustering, and interactive data querying. Furthermore, Spark supports high level libraries for various applications such as machine learning, streaming data processing, database querying and graph data mining thanks to its versatility. In this work, we introduce the concept and programming model of Spark, and show some implementations of simple statistical computing applications. Furthermore, we review the machine learning package MLlib, and the R languge interface SparkR.

Keywords: Spark, machine learning, cluster computing, parallel computing

1. 서론

이른바 빅 데이터의 시대가 도래하면서 클러스터를 이용한 분산 컴퓨팅의 수요는 폭발적으로 증가했다. 특히 대용량 데이터의 분석에 있어서는 아파치 하둡(Apache Hadoop)(Shvachko et al., 2010)으로 대표되는 맵리듀스(MapReduce) 프레임워크(Dean and Ghemawat, 2008)가 널리 사용되었는데, 이는 해당 프레임워크가 사용자들이 고수준의 추상화(abstraction)를 통해 작업이 어떻게 분산되는지, 작업 수행 중 결함이 발생할 때 어떻게 할 지에 대해 고민하지 않고 병렬 계산을 손쉽게 수행할 수 있도록 지원했기 때문이다. 하지만 이 프레임워크에서의 결함 감내성(fault-tolerence)은 작업의 매 단계마다 디스크에 데이터를 직접 저장함으로써 얻은 것이었기 때문에 몇 가지 한계가 있었다. K-평균 알고리즘이나 로지스틱 회귀 분석과 같이 데이터를 반복적으로 재사용하여 추정량을 갱신하는 알고리즘의 경우, 각각의 반복은 하나의 맵리듀스 작업으로 작성할 수 있었지만 매 반복 절차마다 데이터를 디스크에서 읽어와야 했기 때문에 성능에 상당한 제약이 있었다. 한편, 사용자가 즉석에서 같은 데이터에 여러 번 즉석에서(ad-hoc) 질의를 내리는 방법을 사용하는 대화형(interactive) 데이터 마이닝의 경우 데이터를 메모리에 올린 후 반복적으로 질의를 하는 것이 이상적이겠으나 맵리듀스 구조에서는 그것이 불가능했고, 매 질의마다 디스크에서 데이터를 새로 읽어 와야 했다.

이 논문에서는 이러한 문제를 해결한 클러스터 플랫폼인 아파치 스파크(Apache Spark)(Spark, nd)에 대해 소개한다. 스파크는 기존의 맵리듀스가 가지고 있던 확장성과 결함 감내성이라는 중요한 특징들을 유지하면서 보다 넓은 범위의 작업을 하나의 플랫폼에서 실행할 수 있게 한다. 특히, 스파크는 복구 가능한 분산 데이터셋(resilient distributed dataset, RDD)이라는 추상화를 통해 반복 작업 또는 즉석 작업을 효과적으로 수행할 수 있도록 데이터를 메모리에 캐싱하는 기능을 제공한다.

이 논문의 구성은 다음과 같다. 제 2절에서 스파크의 개념과 역사, 그리고 서드 파티 라이브러리를 살펴 보고, 제 3절에서는 스파크의 프로그래밍 모형에 다룬다. 이어 제 4절에서는 스파크로 통계 계산 알고리즘을 작성하는 몇 가지 예시를 다룬다. 제 5절에서는 스파크의 기계 학습 라이브러리 MLlib에 대해 다루며, 아울러 라이브러리에 구현된 특이값 분해 알고리즘을 살펴봄으로써 복잡한 통계 계산 알고리즘을 어떻게 효율적으로 구현할 수 있는 지 고찰한다. 제 6절에서는 스파크의 R 인터페이스에 대해 다루고, 제 7절에서 실험을 통해 스파크의 확장성에 대해 살펴본다. 마지막으로 제 8절에서 결론을 맺는다.

This work was supported by the National Research Foundation of Korea (NRF) grants funded by the Korea government (MSIP) (Nos. 2013R1A1A1057949 and 2014R1A4A1007895).
[1]Corresponding author: Department of Statistics, Seoul National University, 1 Gwanak-ro, Seoul 08826, Korea. E-mail: wonj@stats.snu.ac.kr

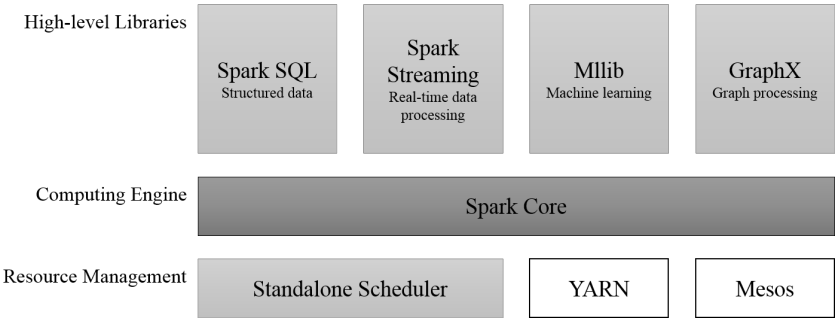

Figure 2.1. Components of Apache Spark. YARN and Mesos are not part of the Spark package, but can be used as a resource manager.

2. 아파치 스파크

아파치 스파크는 빠른 범용 클러스터 컴퓨팅을 위한 플랫폼이다. 속도 측면에 있어서 스파크는 맵리듀스(MapReduce)의 일반화로 기존 맵리듀스 프레임워크(Dean and Ghemawat, 2008)가 효율적으로 동작하지 못했던 반복적인 기계 학습 알고리즘과 대화형 쿼리 형태의 작업에서의 놀라운 성능 향상을 보였다. 특히, 첫 연구 논문이 나온 2010년에도 이미 하둡에 비해 5-10배 빠른 결과를 보여 주었다(Zaharia et al., 2010). 범용성에 있어서 스파크는 서로 다른 파이프라인을 설치해서 사용해야 했던 배치 어플리케이션, 반복 알고리즘, 스트리밍, 사용자 상호작용(interaction) 등 다양한 분산 작업들을 하나의 엔진에서 제공한다는 특징을 갖는다. 이를 통해 다양한 분산 작업들을 손쉽게 조합할 수 있다는 강점을 갖고 있으며 이와 더불어 서로 다른 도구들을 유지 보수해야 하는 부담을 줄여준다.

스파크는 자바 가상 머신(Java Virtual Machine, JVM) 상에서 사용할 수 있는 함수형 프로그래밍 언어인 스칼라(Scala)(Scala, nd)를 사용하여 구현되었다. 하지만 스칼라 이외에도 파이썬(Python), 자바(JAVA), SQL, R 등 다양한 언어로 구성된 API와 풍부한 내부 라이브러리를 통해 쉽게 접근할 수 있다. 또한, 다른 빅 데이터 도구들과 유기적으로 연결된다. 예를 들어 스파크는 하둡 클러스터에서 실행될 수 있고, 이에 따라 HBase(HBase, nd)를 비롯한 다양한 하둡 데이터 소스를 사용할 수 있다.

2.1. 스파크의 구성

스파크는 모든 계산의 중심이 되는 코어(core)와 스파크 SQL(Spark SQL), 스파크 스트리밍(Spark Streaming), MLlib, GraphX 같은 고수준 라이브러리, 클러스터 매니저로 구성되어 있다. 스파크의 코어(Zaharia et al., 2010, 2012)는 내부 계산 엔진으로 스케줄링과 분산, 메모리 관리, 결함 복구, 저장 시스템과의 상호작용 등을 담당한다. 스파크의 핵심 프로그래밍 추상화(abstraction)인 RDD도 코어의 구성 성분이다. 이 코어가 갖는 범용성 덕에 코어를 사용하여 다양한 고수준 라이브러리들이 구현될 수 있었다고 할 수 있다. 고수준 라이브러리 중 스파크 SQL(Armbrust et al., 2015)은 구조적인 자료를 처리하는 기능을 가지고 있으며, SQL 언어를 사용하여 직접 데이터를 처리할 수 있다. 스파크 스트리밍(Zaharia et al., 2013)은 웹 서버에서 실시간으로 생성되는 로그 파일이나 웹 서비스 사용자의 상태 업데이트와 같은 실시간 데이터 스트림을 처리하는 기능을 가지고 있다. MLlib(Meng et al., 2016)은 자주 사용되는 기계 학습 기능들을 제공한다. 분류, 회귀, 군집화, 협동 필터링 등 다양한 모형과 이들을 사용하기 위한 자료 구조를 제공하며, 경사 하강법을 비롯한 최적화 기법들을 제공한다. GraphX(Xin et al., 2014b)는 병렬적인 그래프 처리에 특화된 패키지로 다양한 형태의 그래프와 페이지랭크(PageRank), 삼각형 세기 등 다양한 그래프 알고리즘을 제공한다. 클러스터 매니저는 클러스터의 자원을 효과적으로 분배하는 역할을 한다. 클러스터 매니저로는 아파치 메소스(Apache Mesos)(Hindman et al., 2011)나 얀(YARN)(Vavilapalli et al., 2013)을 사용할 수 있으며, 간단한 기능이 구현된 독립형(standalone) 클러스터 매니저를 사용할 수도 있다. 이러한 구성 성분들은 그림 2.1에 도식화되어 있다.

2.2. 스파크의 역사

스파크는 2009년 UC 버클리의 AMPLab(Algorithms, Machines and People Lab)에서 마테이 자하리아(Matei Zaharia)가 시작한 프로젝트이다(Zaharia et al., 2010). 이 연구소에서는 맵리듀스를 이용해 반복적인 작업과 사용자 상호작용이 필요한 분석 작업이 효율적이지 않음을 관찰했고, 처음부터 이 두 가지 문제를 해결하기 위해 스파크를 디자인했다. 스파크의 연구 논문은 2010년(Zaharia et al., 2010)과 2012년(Zaharia et al., 2012)에 각각 출판되었다.

스파크의 초기 사용자는 UC 버클리 내부의 다른 그룹들이었다고 한다. 예를 들어, 자하리아가 직접 참여한 모바일 밀레니엄 프로젝트(Hunter et al., 2011)는 스파크를 활용하여 샌프란시스코 만 영역의 교통 혼잡을 모니터하고 예측했

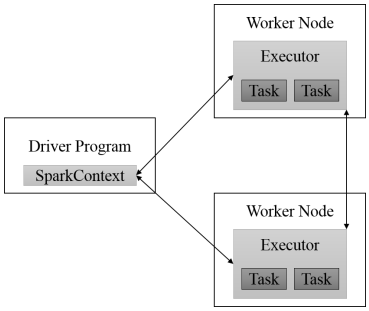

Figure 3.1. A diagram of distributed execution model of Spark

다. 이후 외부 단체들이 스파크를 이용하기 시작했고, 현재는 스파크 위키(Spark Wiki, ndb)의 "Powered By Spark" 페이지에 2016년 8월 30일 현재 95개의 단체가 그 이름을 올려 놓았다. 현재 스파크 프로젝트의 주요 기여자로는 UC 버클리의 AMPLab과 자하리아가 설립한 회사인 데이터브릭스(Databricks)의 개발자들이 있고, 인텔, IBM, 야후, 페이스북 등 다양한 기업의 개발자가 스파크 프로젝트에 직접 기여하고 있다(Spark Wiki, nda).

스파크는 2010년 3월에 BSD 라이센스로 소스를 공개했고, 2011년에는 AMPLab에서 스파크 상에 Shark(Hive on Spark, 현재는 스파크 SQL로 제공됨)이나 스트리밍 데이터를 위한 스파크 스트리밍 등의 고수준 라이브러리를 개발하기 시작했다. 2013년 6월에는 아파치 재단에 기부되었고, 2014년 2월에 최고 수준(top-level) 프로젝트가 되었다. 2014년 11월에는 데이터브릭스에서 스파크를 활용하여 2014 그레이 정렬 벤치마크(2014 Gray Sort Benchmark) 데이토나(Daytona) 100TB 부문에서 기존의 하둡을 사용한 세계 기록을 뛰어넘었다(Xin et al., 2014a).

2.3. 서드 파티 라이브러리

스파크 내부에서 제공하는 다양한 라이브러리뿐만 아니라 다양한 서드 파티 라이브러리들이 오픈 소스로 공개되어 있다. 여기에는 카산드라(Cassandra)(Lakshman and Malik, 2010)를 비롯한 다양한 데이터 소스와 스파크를 연결하는 패키지(Spark-cassandra-connector, nd)를 비롯해 MLlib에서 제공하는 것보다 넓은 범위의 기계 학습 알고리즘을 구현한 패키지들, 제플린(Zeppelin)(Zeppelin, nd)으로 대표되는 스파크의 사용 자체를 편리하게 해 주는 도구들 등등이 있다.

기계 학습(machine learning)을 위한 서드 파티 라이브러리들에 대해 조금 더 살펴 보자. 고성능 컴퓨팅을 활용한 빅데이터 분석 플랫폼인 H2O(H2O.ai, nda)에서는 그 기능들을 스파크로 확장한 스파클링 워터(Sparkling Water)(H2O.ai, ndb)라는 이름의 패키지를 공개하였으며, 대표적인 파이썬 용 오픈 소스 기계학습 라이브러리인 사이킷런(scikit-learn)(Pedregosa et al., 2011)을 스파크로 확장하기 위한 라이브러리도 여러 개가 있다(Sparkit-learn, nd; Spark-sklearn, nd). GPU 한 대의 메모리에는 한계가 있기 때문에 분산 컴퓨팅이 필수적이라고 받아들여지는 심층 학습(deep learning)에서도 스파크를 활용한 여러 시도들이 있었다(Moritz et al., 2015; Kim et al., 2016). 그 외에도 데이터브릭스에서 공개한 볼록 원추 계획법(cone programming) 문제를 풀기 위한 최적화 패키지(Spark-tfocs, nd), 인텔에서 개발한 주제 모델링(topic modeling)(TopicModeling, nd) 패키지 등 다양한 기계학습 라이브러리들이 공개되어 있다.

3. 프로그래밍 모형

3.1. 스파크 인터페이스

각각의 스파크 애플리케이션은 드라이버 프로그램을 통해 "`main`" 함수를 실행하고, 클러스터 상에서 다양한 병렬 연산을 수행한다. 모든 스파크 프로그램에서는 계산 클러스터와의 연결 설정을 나타내는 `SparkContext` 객체가 존재하고 드라이버 프로그램은 이 객체를 통해 스파크에 접근한다. 또한, RDD와 공유 변수들은 이 객체의 메소드를 통해 생성된다. 스파크에서 제공하는 셸(shell)에서는 이 객체가 `sc`라는 이름으로 자동으로 초기화되고, 다른 프로그램에서는 코드 작성 시에 이 객체를 적절히 설정하여 초기화해 주어야 한다. 이 논문에서는 모든 예시에서 `SparkContext`를 나타내는 변수명으로 `sc`를 사용한다. 클러스터 상에서 계산에 사용되는 노드 각각을 스파크에서는 워커(worker) 노드라고 하며, 스파크에서 정의된 연산을 병렬적으로 수행하기 위해서 드라이버 프로그램은 클러스터 매니저를 통해 각각의 워커 노드에서 실행되는 프로세스인 실행자(executor) 여러 개를 관리한다. 이 내용은 그림 3.1에 도식화되어 있다.

스파크에서는 독립형(standalone), 아파치 메소스, 얀으로 이루어진 세 종류의 클러스터 매니저를 지원한다. 우선 아파치 메소스(Apache Mesos)(Hindman et al., 2011)는 범용 클러스터 매니저로, 스파크 이외에도 하둡과 MPI 등 다

양한 프레임워크를 지원한다. 얀(Yet Another Resource Negotiator, YARN)(Vavilapalli et al., 2013)은 하둡 버전 2에서 추가된 하둡의 리소스 매니저로, 하둡 클러스터 상에서 다른 하둡 작업들과 효과적으로 자원을 공유할 수 있도록 해 준다. 마지막으로 독립형 매니저는 스파크에 내장된 리소스 매니저로, 다른 클러스터 매니저를 설치하지 않고도 손쉽게 분산 컴퓨팅이 가능하도록 해 준다. 이러한 클러스터의 종류와 크기는 `SparkContext`의 `master` 파라미터를 통해 설정할 수 있다.

전술했다시피 스파크에서는 복구 가능한 분산 데이터셋(RDD)(Zaharia et al., 2012)을 핵심적인 자료 구조로 제공하고, 이와 더불어 공유 변수인 방송 변수(broadcast variable)과 누산기(accumulator)를 제공한다. 이 절의 나머지 부분에서는 이들에 대해 살펴본다.

3.2. 복구 가능한 분산 데이터셋(Resilient Distributed Dataset)

RDD는 분산 공유 메모리의 제한적인 형태로, 값이 변하지 않고(immutable), 읽기만 가능한(read-only) 특성을 가지고 있으며, 데이터는 여러 조각으로 분할(partition)되어 클러스터에 분산된다. 아파치 스파크에서의 모든 작업은 이 RDD를 새로 만들거나, RDD를 변환(transform)하거나, RDD에서 어떤 연산 결과를 받아오는 것으로 이루어진다. 내부적으로 스파크는 자동으로 RDD에 들어 있는 데이터를 클러스터에 분산시키고, 연산을 병렬적으로 수행한다.

RDD의 값 자체가 연산이 주어지는 즉시 계산되지는 않는다. 대신, RDD를 다른 RDD나 저장 공간으로부터 계산하기 위해 필요한 과정, 즉 연산 계통(lineage)만을 기록해 둔 다음 RDD의 값이 필요해질 때 지연 계산(lazy-evaluation)을 통해 계산하게 된다. 이 계통을 통해 결함 감내성을 얻는다. 만약 분산된 데이터의 일부에 손실이 생긴다면, RDD는 저장되어 있는 연산 계통을 통해 그 부분만 다시 계산한다.

아울러, 사용자는 어떤 RDD를 다시 사용할 지, 어떤 저장 공간에 유지(persist)할 지(예: 인메모리, 즉 RAM 안에)를 직접 설정할 수 있으며, RDD의 원소들이 각 레코드의 키(key)를 통해 어떤 식으로 분할될 지 설정할 수 있다.

이러한 방식을 통해 매 단계마다 매우 느린 연산인 디스크 입출력을 통해 결함 감내을 구현하는 하둡에 비해, 인메모리에서 결함 감내성을 얻을 수 있어 반복적인 알고리즘 및 상호작용성(interactive) 데이터 마이닝 도구들에서 상당한 성능 개선을 얻을 수 있게 해 준다.

3.3. RDD의 생성

RDD를 처음 만드는 방법에는 크게 두 가지가 있다. 첫 번째는 안정적인 저장 공간에 저장된 외부 데이터를 불러오는 것이다. 스파크는 하둡 API가 지원하는 저장소(repository)들을 지원하며, 여기에는 지역 파일 시스템, 하둡 분산 파일 시스템(Hadoop Distributed File System, HDFS), 또는 아마존(Amazon) S3와 같은 다양한 저장 공간이 포함된다. 이는 스파크를 사용하기 위해 하둡이 꼭 필요함을 의미하지는 않으며, 스파크 내부적으로 구현한 하둡 API를 사용하여 저장 공간을 지원한다. 두 번째는 드라이버 프로그램에 있는 집합을 병렬화하는 것이다. 이 방법에서는 스칼라나 파이썬의 `List` 등등을 `SparkContext.parallelize` 함수의 파라미터로 입력받아 이를 클러스터에 분산시킨다. 집합을 병렬화할 경우에는 이 집합을 몇 개의 분할로 나눌 지 결정할 수 있는데, 스파크에서 병렬 연산을 실행할 때에는 하나의 분할 당 하나의 태스크를 실행하게 되므로 적절한 수의 분할을 사용하는 것이 중요하며, 스파크 프로그래밍 가이드(Spark, nd)에서는 CPU 당 2-4개의 분할을 사용하는 것을 권장한다.

3.4. RDD의 연산

RDD의 연산에는 변환(transformation)과 액션(action)의 두 종류가 있다. 변환은 RDD를 입력받아 새로운 RDD를 반환하는 연산으로, `map`이나 `filter`와 같은 연산을 포함한다. 액션은 RDD를 입력받아 어떤 연산을 수행하여 그 결과를 드라이버 프로그램에 반환하거나 저장 공간에 쓰는 연산이다. 여기에는 `count`와 `collect`와 같은 연산이 포함된다. 이 두 종류의 연산은 스파크에서 완전히 다르게 처리되므로 이 둘을 잘 구분해야 한다.

3.4.1. 변환 변환이란 기존의 RDD를 입력으로 받아 이를 변환하여 새로운 RDD를 만들어 내는 연산이다. 여기에는 필터링, 맵과 같은 동작이 포함된다. 변환된 RDD는 지연 계산되므로 액션 연산을 하기 전에는 그 안에 들어 있는 값이 계산되지 않는다. 대표적인 변환에는 RDD의 각 원소를 일대일로 변환하는 `map`, RDD의 각 원소를 일대다로 변환하는 `flatMap`, RDD의 원소 중 조건에 맞는 것만 남기는 `filter` 등이 있다.

3.4.2. 액션 액션은 RDD에서 어떤 연산을 수행하여 이를 파일로 저장하거나 드라이버 프로그램에 반환하는 연산이다. 여기에는 RDD의 모든 원소를 반환하는 `collect`, RDD의 원소 일부를 가져오는 `take`, RDD의 원소들에 대해 교환 법칙과 결합 법칙이 성립하는 연산을 수행하는 `reduce` 등이 있다.

3.4.3. 키-값 쌍에 대한 연산 RDD에서 같은 키를 갖는 원소들의 값을 결합법칙과 교환법칙이 성립하는 연산을 통해 하나의 값으로 줄여 각 키마다 하나의 키-값 쌍을 갖는 RDD를 반환하는 변환 `reduceByKey`, 두 개의 RDD에서 같은 키를 갖는 값끼리 짝짓는 변환 `join`, 각 키 별로 그 키를 갖는 원소의 수를 세는 액션인 `countByKey` 등등은 키-값 쌍으로 이루어진 RDD에 대해서만 사용할 수 있다. 또한 이러한 연산들의 경우 키의 값에 따라 각 자료점이 들어갈 분할이 결정된다.

3.5. 셔플과 스테이지

어떤 연산을 통해 만들어지는 RDD의 다른 RDD에 대한 의존성에 대해 살펴보자. `map`과 같은 많은 연산은 그 결과가 입력 원소 하나에만 의존하기 때문에 각 분할에서 다른 분할과의 통신 없이 빠르게 계산이 가능하다. 이러한 의존성을 좁은(narrow) 의존성이라고 한다. 각각의 입력 RDD의 분할은 출력 RDD의 분할 중 최대 하나에서만 사용된다.

반면, `join`이나 `reduceByKey`와 같은 연산의 경우, 출력 RDD의 분할 하나가 입력 RDD의 분할 중 여러 개에 의존하게 된다. 예를 들어, `reduceByKey`의 결과로 나오는 RDD는 입력 RDD가 이미 키를 기준으로 분할되어 있는 경우가 아니면 입력 RDD의 모든 분할에서 같은 키를 갖는 값들을 모두 모아 계산하게 된다. 이러한 의존성을 넓은(wide) 의존성이라고 한다. 일반적으로 스파크 데이터가 특정 연산을 위해 필요한 위치에 있는 것을 보장하지 않기 때문에 이 연산은 모든 노드에 자료를 복사하고 전송하는 하는 과정을 필요로 하게 되고, 이 과정은 디스크 입출력과 네트워크 입출력을 사용하는 매우 복잡하고 느린 연산이다. 입력 RDD 쪽에서는 맵 연산을 통해 자료를 유기적으로 정리하게 되고, 자식 RDD 쪽에서는 리듀스 연산을 통해 결과를 누적한다. 여기서 맵과 리듀스는 고전적인 맵리듀스(Dean and Ghemawat, 2008) 프레임워크에서의 맵과 리듀스를 의미하며, 스파크의 연산 `map`이나 `reduce`와는 직접적인 연관이 없다.

이러한 연산을 셔플(shuffle)이라고 하며, 하나의 액션은 변환 중 셔플 연산으로 구분된 스테이지(stage)라는 개념으로 구분된다. 즉, 하나의 스테이지는 몇 개의 좁은 의존성을 갖는 연산들로 이루어지며 이들은 하나의 클러스터 노드에서 빠르게 계산될 수 있다. 이 안에서는 노드 하나가 고장나도 잃어버린 입력 RDD의 분할만 다시 계산하면 되기 때문에 효율적으로 복구가 가능하다. 반면, 스테이지의 경계에 해당하는 RDD 연산 사이에는 넓은 의존성이 있고, 각각의 자식 RDD 분할을 계산하기 위해 모든 입력 RDD의 분할이 필요하다. 이 과정에서 상당히 느린 셔플 연산이 일어나게 되고, 하나의 노드가 고장나면 모든 이전 RDD의 자료를 잃어버릴 수 있으며 그 과정에서 전체를 다시 계산해야 할 수 있다. 이를 방지하기 위해 셔플 연산에서는 맵리듀스와 유사하게 많은 수의 파일을 디스크에 쓰게 된다. 셔플 연산을 통해 병렬화하기 쉬운 모양을 만들 수 있는 예외적인 경우를 제외하면 일반적으로 셔플 연산의 수를 될 수 있는 대로 줄이는 것이 효과적이다. 특히 반복적인 기계 학습 작업에서는 매 반복마다 셔플 연산을 줄이면 큰 성능 향상을 불러올 수 있다.

3.6. 예시: 단어 세기 프로그램

이상의 구성이 실제 프로그램에서 어떻게 사용되는 지 살펴보기 위해 파이썬으로 구현된 단어 세기 예시를 살펴 보자. 단어 세기 프로그램은 말 그대로 주어진 텍스트에서 각 단어가 등장하는 횟수를 세는 프로그램이다. 매우 큰 텍스트 파일에 대해 이 계산을 분산 처리하는 것은 빅 데이터 프로그래밍에서 순차적 프로그래밍의 "Hello, World!"에 비견될 정도로 대표적인 예제로 쓰인다. 스파크에서는 이 프로그램을 파이썬으로 다음과 같이 구현할 수 있다.

```
1 lines = sc.textFile(...)
2 counts = lines.flatMap(lambda x: x.split(" ")) \
3             .map(lambda x: (x,1)) \
4             .reduceByKey(lambda x,y: x+y)
5 output = counts.collect()
6 for (word, count) in output:
7     print("%s: %i" % (word, count))
```

1행은 텍스트 파일의 각 줄을 원소로 하는 RDD를 `lines`라는 변수에 저장한다. 텍스트 파일의 주소는 생략되어 있지만 앞서 언급한 바와 같이 지역 저장 공간이나 HDFS, S3 등 다양한 저장 공간에 있는 텍스트 파일을 사용할 수 있다.

2-4행에서는 이 `lines`를 변환하여 (단어, 횟수)의 쌍을 원소로 갖는 RDD를 `counts`라는 변수에 저장한다. 여기서 사용되는 변환 세 가지에 대해 차례차례 살펴 보자. 먼저 `flatMap`은 입력 RDD의 각 원소에 대해 일 대 다 변환을 하는 역할을 한다. 입력된 함수 `lambda x: x.split(" ")`는 RDD의 원소(텍스트파일의 각 줄)를 공백을 기준으로 분리하여 리스트에 저장하는 함수이다. 따라서 `flatMap`의 결과로는 처음 텍스트 파일을 이루는 각 단어(공백을 기준으로 구분된 조각)를 원소로 갖는 RDD가 반환된다. 이어지는 `map` 함수는 RDD의 각 원소를 일 대 일로 변환한다. 여기서는 각 단어가 (단어, 1)의 형태를 갖는 길이 2의 튜플(tuple)로 변환된다. 파이썬 인터페이스에서 이렇게 길이 2의 튜

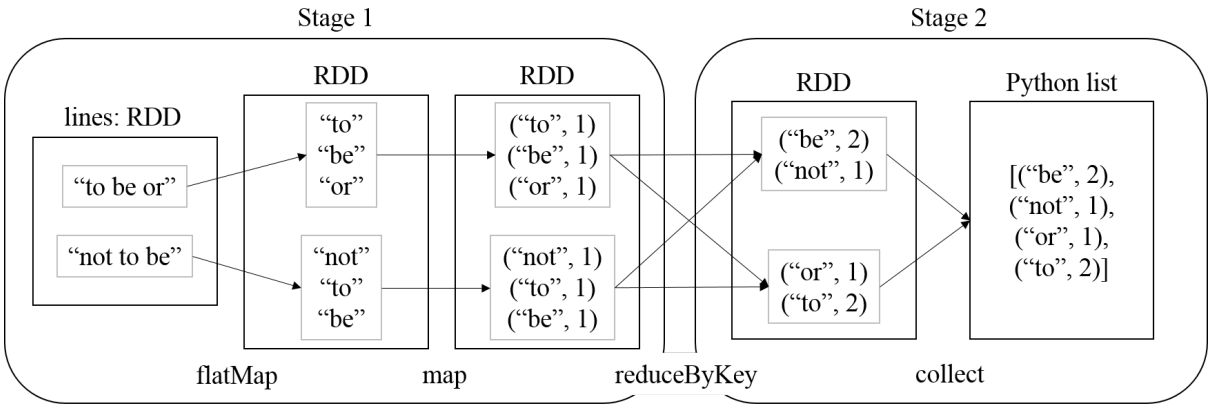

Figure 3.2. Diagram for the word count program with a two-line sample text, "to be or"//"not to be". We assumed that the original RDD, `lines` is consisted of two partitions with one element in each partition. Gray squares denote partitions of an RDD.

플을 원소로 갖는 RDD는 스파크에서 키-값 쌍으로 인식한다. 마지막 변환 `reduceByKey`는 이 키-값 쌍 RDD에서 같은 키를 갖는 원소들의 값에 대해 교환 및 결합법칙이 성립하는 연산을 수행한다. 여기서는 덧셈(`lambda x,y:x+y`)을 수행하며, 이에 따라 각각의 단어마다 그 단어가 나타나는 횟수가 계산된다. 계산 결과는 (단어, 횟수)의 쌍을 원소로 갖는 RDD이다. 모든 변환은 지연 계산되므로 액션이 수행되기 전까지는 직접 계산되지 않으며, 연산 계통만 기록된다.

5행에서는 계산된 (단어, 횟수)의 쌍을 드라이버 프로그램으로 가져오기 위해 액션 `collect`를 수행한다. 이 액션을 수행할 때에서야 비로소 이전의 변환들이 직접 계산되며, 그 결과는 파이썬 `list`의 형태로 주어진다. 이 리스트의 내용은 6-7행을 통해 출력할 수 있다.

마지막으로 이 작업에서 셔플 동작을 살펴 보자. `flatMap`, `map`은 결과의 각 원소가 입력 RDD의 원소 하나에만 의존하므로 "좁은" 의존성을 가졌다. 하지만, `reduceByKey`는 결과의 각 원소를 계산하기 위해 모든 분할의 원소를 살펴봐야 하므로 "넓은" 의존성을 가지므로, 이 변환을 위해 셔플이 일어난다. 이 셔플 이전까지의 연산이 하나의 스테이지를 구성하고, 셔플 이후 액션 `collect`까지의 연산이 또 하나의 스테이지를 구성하므로 이상의 액션은 두 개의 스테이지로 구분되는 것을 볼 수 있다. 이 단어 세기 작업을 나타낸 다이어그램은 그림 3.2에서 확인할 수 있다.

3.7. RDD의 유지(persisting)

스파크의 가장 중요한 기능 중 하나는 자료를 인메모리에 유지(persisting)하거나 캐싱(caching)하는 것이다. RDD를 메모리에 유지하게 되면 각각의 노드는 그 노드에서 계산되는 분할들을 메모리에 저장하고 그 RDD를 활용한 다른 액션에서 이를 재사용한다. 이는 이후의 액션의 속도를 매우 빠르게(10배 이상 빨라지는 경우가 많다) 실행할 수 있게 한다. 캐싱은 반복 알고리즘과 사용자 상호작용을 빠르게 수행할 수 있기 위한 핵심적인 기능이다.

RDD의 `persist`나 `cache` 메소드를 호출함으로써 RDD를 메모리에 유지할 수 있고, `persist` 함수의 파라미터를 통해 RDD를 유지하는 방식을 선택할 수 있다. 여기에는 RDD를 메모리 위에만 유지할 지(`MEMORY_ONLY`), RDD를 메모리에 유지하되 메모리가 부족하면 디스크에 저장할 지(`MEMORY_AND_DISK`), 아니면 디스크에만 저장할 지(`DISK_ONLY`) 등등이 있다.

이 기능은 다음 절의 로지스틱 회귀 분석과 K-평균 알고리즘의 구현에 사용될 것이다.

3.8. 공유 변수

스파크에서 `map`이나 `filter`와 같은 연산의 파라미터로서 전달되는 함수들을 생각해 보자. 이 함수는 드라이버 프로그램의 전역 변수를 사용할 수 있으며, 이러한 변수들은 각 태스크가 워커 노드에 전달될 때 함수의 내용과 함께 복사되어 전달된다. 워커 노드에서는 전달받은 변수의 사본을 계산에 활용하며 워커 노드가 함수의 실행 중에 이 사본의 내용을 변경하더라도 그 내용을 드라이버 프로그램에 전달할 수 없다. 스파크의 공유 변수들은 자주 사용되는 통신 패턴인 방송과 결과의 누산에 대해 이러한 제약을 완화시키는 역할을 한다.

3.8.1. 방송 변수 (broadcast variables) 만약 해시 테이블(hash table)과 같이 큰 읽기 전용 자료를 여러 병렬 태스크가 사용한다면 이 자료를 태스크마다 복사하여 통신을 통해 전달하는 것은 같은 워커 노드에서 실행되는 태스크에도 자료를 반복하여 전달하기 때문에 비효율적이다. 스파크에서는 이런 경우 필요한 변수들을 각각의 워커 노드에 한

번 전달하고 이를 각 워커에 할당된 여러 태스크가 공유할 수 있는 방송 변수를 제공한다. 예를 들어 워커 노드에 분산되어 저장된 행렬과 어떤 벡터의 곱을 계산하고자 한다면 방송 변수를 통해 벡터를 워커들에 전달하여 워커에서 부분 행렬과 벡터의 필요한 부분을 곱하는 것이 효율적일 것이다.

3.8.2. 누산기 (accumulators) 누산기는 병렬 작업 각각에서 결합 및 교환법칙이 성립하는 연산 결과값을 누적하는 기능을 하며, 누적된 값은 드라이버 프로그램에서만 확인할 수 있다. 이러한 누산기는 카운터를 구현하거나 병렬 합과 같은 간단한 연산에 유용하게 쓰일 수 있다. 누산기는 교환 및 결합법칙이 성립하는 연산과 이 연산에 대한 항등원이 존재하는 임의의 자료형에 대해 정의할 수 있다.

4. 알고리즘 예제

본 절에서는 스파크를 이용하여 간단한 통계 계산 알고리즘을 구현한 예시를 다룬다. 예시들은 정적 타입을 사용하는 스칼라나 자바에 비해 동적 타입을 지원하여 보다 쉽게 읽고 이해할 수 있는 파이썬으로 구성하였다. R 인터페이스는 다른 언어 인터페이스에 비해 다른 부분이 많아 제 6절에서 따로 다룬다. 모든 예시는 스파크 패키지에 포함된 예시 코드의 변형임을 밝혀 둔다.

4.1. 몬테 카를로 추출을 통한 원주율 계산

이 문제는 정사각형 $[-1,1]\times[-1,1]$ 안에서 n개의 점을 무작위로 추출하여 그 중 단위원 안에 포함된 점의 수 c 를 활용하여 원주율 π를 계산하는 문제로, π의 추정량은 $\hat{\pi}=4c/n$으로 주어진다. 이러한 간단한 몬테 카를로 추출은 "당황스럽게 병렬적인(embarassingly parallel)" 문제로 잘 알려져 있다.

먼저, `map` 함수에 파라미터로 넘겨주기 위한 함수 `f`를 만들자. 이 함수는 점을 추출하고 그 점이 단위원에 속하는지 확인하는 역할을 한다.

```
def f(x):
    x = random() * 2 - 1
    y = random() * 2 - 1
    return 1 if x**2 + y**2 < 1 else 0
```

위 함수를 사용하면 원주율은 다음과 같이 근사할 수 있다.

```
c = sc.parallelize(range(n)).map(f).reduce(lambda x,y: x+y)
print(4.0*c/n)
```

위 코드는 크기 n의 RDD를 만든 뒤 함수 `f`를 통해 `map`을 함으로써 각 원소가 0(추출된 점이 단위원에 포함되는 경우) 또는 1인 RDD가 만들어지며, 이를 덧셈 함수를 통해 `reduce`함으로써 n개의 점 중 단위원에 들어간간 점의 수를 계산하게 된다.

4.2. 로지스틱 회귀 분석

이번에는 로지스틱 회귀 분석의 계수 추정을 경사 하강법(gradient descent method)을 통해 시행하는 법을 스파크로 구현해 보자. 로지스틱 회귀 모형의 로그가능도의 경사는

$$\nabla ell(\beta)=\sum_{i=1}^{n}\left(\frac{1}{1+\exp(-y_i\beta^T x_i)}-1\right)y_i x_i \tag{4.1}$$

로 주어진다. 여기서 n은 데이터의 크기이고, 회귀계수는 $\beta\in\mathbb{R}^D$, 각 데이터는 $x_i\in\mathbb{R}^D$, 라벨은 $y_i\in\{-1,+1\}$을 따른다.

텍스트 파일의 각 행이 분류 레이블과 D차원의 변수(feature)로 주어져 있고 `parsePoint` 함수가 이를 MLlib의 `LabeledPoint`로 변환하는 함수로 주어져 있다고 가정하자. 이 때 하나의 `LabeledPoint` 객체는 레이블을 저장하는 멤버 변수 `label`과 길이 D의 변수 벡터를을 저장하는 `features`로 되어 있다. 이를 활용하여 다음과 같이 로지스틱 회귀 분석을 구현할 수 있다.

```
1 points = sc.textFile(...).map(parsePoint).cache()
2 w = 2 * np.ranf(size=D)-1
```

```
3
4 for i in range(iterations):
5     w -= points.map(
6         lambda p: p.features * (1/(1+np.exp(-p.label * w.dot(p.features)))-1.0)*p.label
7         ).reduce(lambda x,y:x+y)
```

1행에서 자료 points가 반복해서 사용될 것이므로 cache 함수를 통해 인메모리에 그 값을 유지하게 하였다. 만약 cache를 사용하지 않았다면 메모리에 points 변수의 값이 저장되지 않으므로 for 문에서 매 반복마다 텍스트 파일을 새로 읽어오게 된다.

4-7행의 for 문에서는 points부터 시작하여 map을 통해 현재 w에서의 로그가능도의 경사를 점마다 계산하고, 이를 reduce를 통해 합치게 된다.

4.3. K-평균 알고리즘

K-평균 군집화 알고리즘은 다음 과정을 따른다.

1. 원하는 군집의 수 K를 입력받는다.
2. K개의 군집 중심을 임의로 정한다 (여기서는 임의로 선택된 자료점 하나로 설정).
3. 각각의 자료점과 K개의 군집 중심과의 거리를 계산한 후 가장 가까운 군집으로 자료점을 분류한다.
4. 새로이 분류된 각 군집의 새로운 군집 중심을 해당 군집에 속한 점들의 평균으로 계산한다.
5. 군집 중심이 수렴했다면 종료하고, 그렇지 않다면 3단계로 이동한다.

먼저, 각 자로점에서 가장 가까운 군집 중심을 찾는 함수 closestPoint를 작성하자.

```
def closestPoint(p, centers):
    bestIndex = 0
    closest = float("+inf")
    for i in range(len(centers)):
        tempDist = np.sum((p - centers[i]) ** 2)
        if tempDist < closest:
            closest = tempDist
            bestIndex = i
    return bestIndex
```

로지스틱 회귀 분석에서와 비슷하게 텍스트 파일로 이루어진 데이터를 다음과 같이 벡터로 변환할 수 있다. 이 벡터들은 반복적으로 사용되므로 cache 함수를 통해 메모리에 유지한다.

```
lines = sc.textFile(...)
data = lines.map(parseVector).cache()
```

또한, RDD에서 정해진 개수만큼의 샘플을 추출하는 액션인 takeSample로 군집 중심을 다음과 같이 초기화한다. 여기서 함수의 첫 번째 파라미터는 복원추출 여부를, 두 번째 파라미터는 추출하고자 하는 점의 수를 나타낸다. 즉, 여기서는 전체 점의 집합에서 비복원추출된 K개의 점을 초기 군집 중심으로 사용한다.

```
kPoints = data.takeSample(False, K, 1)
```

이후 반복 알고리즘을 따라 군집화를 진행한다.

```
1 tempDist = 1.0
2 while tempDist > convergeDist:
3     closest = data.map (
4         lambda p: closestPoint(p, kPoints), (p,1))
5     pointStats = closest.reduceByKey(
6         lambda p1_c1, p2_c2: (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))
7     newPoints = pointStats.map(
8         lambda st: (st[0], st[1][0]/st[1][1])).collect()
```

```
9     tempDist = sum(np.sum((kPoints[iK]-p) **2) for (iK, p) in newPoints)
10
11    for (iK, p) in newPoints:
12        kPoints[iK] = p
```

3행의 `closest`는 가장 가까운 군집 중심의 인덱스를 키로, 자료점의 정보 `p`와 숫자 1의 쌍을 값으로 갖는 RDD가 된다. 숫자 1은 각 군집에 속한 자료점의 수를 세기 위한 값이다. 5-6행에서 계산되는 `pointStats`는 `reduceByKey`를 통해 각 군집 중심 별로 점의 좌표값들과 숫자 1 들을 각각 더한다. 이를 통해 각 군집 별 좌표값의 합과 자료점의 수가 계산된다. 7-8행에서는 새 군집 중심인 `newPoints`를 구하기 위해 계산된 좌표의 합을 군집에 속한 점의 수로 나눈 뒤, 이를 `collect`를 통해 드라이버 프로그램에 가져온다. 9행의 `tempDist`는 새로 계산된 군집 중심들과 이전 반복에서 계산된 군집 중심들의 거리를 계산하고, 마지막으로 11-12행에서 새로운 군집 중심을 `kPoints`에 업데이트하게 된다.

로지스틱 회귀 분석과 K-평균 군집화 알고리즘의 최적화된 사용자 인터페이스는 다음 장 MLlib에서 다시 다루기로 한다.

5. MLlib

MLlib(Meng et al., 2016)은 스파크에서 제공하는 기계 학습 라이브러리로 분류, 회귀, 의사 결정 트리, 랜덤 포레스트, 추천 엔진, 군집화, 주제 모형화, 특징 변환, 모형의 평가와 초모수(hyperparameter) 튜닝, 기계학습 파이프라인 구성, 기계학습 모형의 저장, 생존 분석, 분산 선형 대수, 통계와 같은 다양한 기계학습 도구를 클러스터 상에서 병렬적으로 처리할 수 있도록 구성되어 있다. 이를 활용하여 사용자들은 다양한 기계 학습 모형을 구성해 볼 수 있으며, `RowMatrix`, `IndexedRowMatrix`, `CoordinateMatrix`, `BlockMatrix` 등의 다양한 분산 행렬 자료구조와 (확률적) 경사 강하법, L-BFGS와 같은 다양한 최적화 기법을 활용하여 새로운 종류의 모형을 구현해 볼 수 있다. MLlib은 2012년에 MLbase(Kraska et al., 2013) 프로젝트의 일환으로서 개발되기 시작했고, 2013년 9월에 소스 코드가 공개되었으며, 이 때부터 스파크에 편입되었다(스파크 0.8).

5.1. 기본 예제

MLlib은 RDD로 구성되어 있는 분산 자료에 다양한 알고리즘을 실행하게끔 설계되어 있다. 여기서는 앞서 살펴본 로지스틱 회귀 분석과 K-평균 알고리즘을 MLlib으로 구현하는 방법에 대해 살펴보자.

5.1.1. 로지스틱 회귀 모형 MLlib에서 로지스틱 회귀 모형은 확률적 경사 강하법을 활용하는 `LogisticRegressionWithSGD`와 L-BFGS를 활용하는 `LogisticRegressionWithLBFGS`의 두 가지 클래스로 제공된다. 여기서는 `LogisticRegressionWithLBFGS`를 통해 로지스틱 회귀를 수행하는 코드를 살펴보자. 이전 장에서와 같이 데이터를 `LabeledPoint`로 이루어진 RDD의 형태로 `points`라는 변수에 저장했다고 가정하고, 반복 횟수 `iterations`가 주어져 있다고 가정하면 로지스틱 회귀 분석은 다음과 같이 간단히 구현된다.

```
model = LogisticRegressionWithSGD.train(points, iterations)
```

위 코드를 실행하면 회귀 계수와 절편은 `model`의 멤버 변수로 저장된다.

```
print(model.weight) # model weights
print(model.intercept) # intercept
```

5.1.2. K-평균 알고리즘 MLlib에서는 K-평균 알고리즘을 K-평균||(Bahmani et al., 2012)라는 병렬화된 K-평균++ 알고리즘을 활용하여 구현하였다. 사용자는 클러스터의 수(`k`)와 최대 반복 수(`maxIterations`), 수렴 문턱(`epsilon`) 등을 설정할 수 있다. 역시 앞에서와 같이 `k`와 `data`가 초기화되어 있다고 생각하자. 그러면 K-평균 알고리즘은 다음과 같이 구현된다.

```
model = KMeans.train(data, k)
```

앞에서와 같이 훈련 결과는 `model`의 멤버 변수로 저장된다.

```
print(model.clusterCenters) # cluster centers
print(model.computeCost(data)) # final cost of the clustering
```

5.2. 행렬 계산: 특이값 분해

MLlib에서 선형대수 계산을 위한 모듈은 `linalg`라는 명칭으로 제공되며, 행렬 전치나 곱셈과 같은 간단한 것부터 QR 분해, 특이값 분해와 같은 복잡한 것까지 다양한 범위의 행렬 연산 알고리즘이 빠르고 확장성 있도록 구현되어 있다. 여기서는 특이값 분해(singular value decomposition)의 MLlib 구현을 통해 복잡한 통계계산 알고리즘을 스파크 상에서 어떻게 구현할 수 있는지 살펴 보자. 아래 내용에 대한 보다 자세한 설명은 (Zadeh et al., 2016)에서 확인할 수 있다.

랭크가 k인 $m \times n$ 행렬 A의 특이값 분해는 $A = U\Sigma V^T$로 주어진다. 여기서 U는 k개의 m차원 좌특이(left-singular) 벡터들로 이루어진 직교 행렬이고 Σ는 특이값으로 이루어진 대각행렬, V는 k개의 n차원 우특이(rigt-singular) 벡터들로 이루어진 직교 행렬이다. 스파크는 A^TA의 고유값 분해에 기반하여 특이값 분해를 계산하며, 내부적으로 $m \gg n$인지, 아니면 그렇지 않은지에 따라 자동으로 두 가지의 특이값 분해 알고리즘 중 하나를 선택하여 사용한다.

5.2.1. 일반적인 형태의 분산 특이값 분해 먼저 $m \gg n$이 아닌 경우를 살펴 보자. 스파크에서는 이런 경우 행렬 연산과 벡터 연산을 분리하여 행렬 연산은 클러스터에서 계산하고 벡터 연산은 드라이버 프로그램에서 계산한다. 이 방식은 행렬의 크기는 보통 벡터의 크기의 제곱에 비례하므로 벡터가 드라이버 노드의 메모리 안에 올라갈 수 있지만 행렬은 그러지 못 할 것이라는 가정에 바탕을 둔 방식이다. 특이값 분해 계산을 위해서는 행렬이 행 기준으로 쪼개어져 분산 저장된 클러스터 쪽에서 행렬의 고유값 분해를 위한 수치 계산 패키지 ARPACK(Lehoucq et al., 1998)을 사용한다. ARPACK이 사용하는 아놀디(Arnoldi) 알고리즘에서 필요로 하는 행렬 연산은 행렬-벡터 곱 뿐이고, 이 연산은 벡터를 방송함으로써 쉽게 분산 계산이 가능하므로 스파크에서 특이값 분해를 구할 수 있다. 자바 가상 머신에서 ARPACK을 사용하기 위한 인터페이스로는 `netlib-java`와 `breeze` 패키지를 사용하며, 이들은 스파크에 기본적으로 포함되어 있다.

5.2.2. "크고 가는" 특이값 분해 $m \gg n$인 경우의 특이값 분해는 "크고 가는(tall and skinny)" 특이값 분해라고도 하는데, 여기서는 특히 A^TA가 드라이버 노드의 메모리에 올라갈 수 있을 정도로 k가 작은 경우를 생각한다. 먼저, A^TA의 고유값 분해를 통해 Σ와 V를 계산하는 과정을 살펴보자. A^TA는 셔플 없이 한 번의 `reduce` 액션으로 계산할 수 있다. 크기가 $n \times n$인 행렬 $A^TA = V\Sigma^2V^T$가 드라이버 노드의 메모리에 올라갈 수 있을 정도로 작다고 가정했으므로 드라이버 프로그램에서 이 행렬의 고유값 분해를 ARPACK을 사용해 직접 계산할 수 있고, 이를 통해 Σ와 V를 직접 구할 수 있다. 가정에 의해 $n \times n$ 크기의 행렬 V와 대각행렬 Σ는 드라이버 프로그램의 메모리에 충분히 올라갈 수 있지만, U는 $m \times k$ 크기로 노드 하나에 올릴 수 없을 것이다. 이를 분산 계산하기 위해서 계산하기 쉬운 행렬인 $V\Sigma^{-1}$을 드라이버 프로그램에서 계산한 뒤 이를 A의 각 행을 담고 있는 클러스터에 방송하면 $U = AV\Sigma^{-1}$을 쉽게 계산할 수 있다.

6. SparkR

스파크에 포함된 R 인터페이스인 SparkR(SparkR, nd)에 대해 알아보자. SparkR은 스파크 버전 1.4(2015년 6월)부터 제공된 R 인터페이스로, R이 가진 Scala, Java, Python과는 다른 언어적 특징으로 인해 다른 언어 인터페이스에 비해 특이한 형태의 인터페이스를 제공한다. SparkR의 핵심 자료 구조는 R의 데이터 프레임과 유사한 `SparkDataFrame`이며, 이는 Spark SQL(Armbrust et al., 2015)에 기반하여 연산을 수행한다. 인터페이스의 구조는 효과적인 데이터 프레임 연산 패키지인 `dplyr`(dplyr, nd)의 모형을 따와서 만들었다고 한다. SparkR에서는 버전 2.0부터는 MLlib의 일부 인터페이스를 제공한다. 여기서는 MLlib 인터페이스를 활용한 K-평균 모형 예시를 살펴 보자.

```
# Fit a k-means model
irisDF <- suppressWarnings(createDataFrame(iris))
kmeansDF <- irisDF
kmeansTestDF <- irisDF
kmeansModel <- spark.kmeans(kmeansDF, ~ Sepal_Length + Sepal_Width + Petal_Length + Petal_Width,
                            k = 3)

# Model summary
summary(kmeansModel)

# Get fitted result from the k-means model
showDF(fitted(kmeansModel))
```

```
13
14 # Prediction
15 kmeansPredictions <- predict(kmeansModel, kmeansTestDF)
16 showDF(kmeansPredictions)
```

2행에서는 스파크의 데이터 프레임 구조를 만들고, 5행에서 K-means 모형을 만든다. 여기서 `spark.kmeans` 함수는 R의 `kmeans` 함수에 대응된다. 9행과 15행의 `summary`와 `predict`는 R에서 일반적으로 사용하는 문법을 그대로 사용한다. 그리고 12행과 16행의 `showDF`는 스파크 DataFrame을 화면에 출력하는 함수이다.

이와 같이 SparkR에서는 R의 문법을 최대한 활용하면서 스파크를 사용할 수 있는 환경을 제공한다. SparkR과는 별개로 RStudio에서는 스파크가 `dplyr`를 직접 백엔드(backend)로서 활용하게끔 만들어진 패키지인 sparklyr(RStudio, nd)을 제공한다.

7. 실험

계산 규모의 확장성(scalability)을 알아보기 위해 MLlib을 활용한 K-평균 실험 결과를 살펴보자. 실험에는 마스터 노드 1대와 슬레이브 노드 32대로 구성된 클러스터가 사용되었다. 각각의 노드는 모두 같은 구성을 가지고 있으며 Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz CPU와 16GB의 RAM을 가지고 있다. 실험은 1,000,000개의 100차원 자료를 100개의 클러스터로 군집화하는 문제에서 300회의 반복에 걸리는 시간을 측정하였다. 4, 8, 12, 16, 20, 24, 28개의 노드를 계산에 사용하였으며 각 경우마다 10회씩 실험을 반복하였다. 클러스터 매니저로는 하둡 얀을 사용하였으며, 이에 따라 매 실험마다 계산에 사용되는 것과 별개의 노드 1개를 'ApplicationMaster'로 지정하였다. 실험 결과는 표 7.1와 그림 7.1에 정리되어 있다. 사용하는 노드의 수가 늘어날수록 평균 작업 수행 시간이 줄어들어 스파크가 좋은 확장성을 가짐을 볼 수 있었다. 실험에 사용되는 노드는 매번 얀이 자동으로 설정하는데, 이 구성에 따라 실험 시간에 편차가 나타난다. 이 부분에 의한 변동성 역시 노드 수가 많아짐에 따라 점점 줄어드는 경향을 보임을 알 수 있었다.

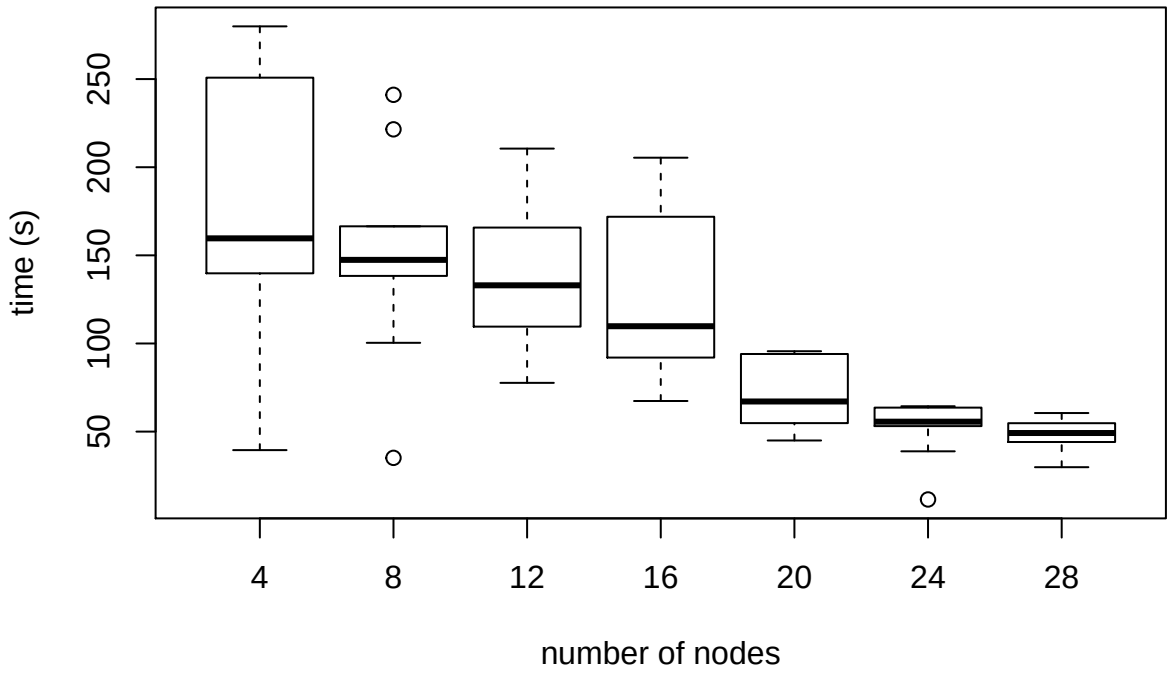

Figure 7.1. Boxplot of time elapsed for the K-means benchmark experiment. One million 100-dimensional data points were clustered into 100 clusters for 300 iterations. The experiment was performed with 4, 8, 12, 16, 20, 24, 28 nodes, with 10 repetitions each.

8. 결론

본 논문에서는 일반적인 클러스터 컴퓨팅을 위한 계산 엔진인 스파크의 개념을 살펴보고 이를 통한 간단한 통계 계산 알고리즘들의 구현을 소개하였다. 또한, 스파크가 지원하는 다양한 고수준 라이브러리 중 기계 학습 전문 라이브러리인 MLlib에 대해 살펴 보았고, 특히 대표적인 행렬 계산 알고리즘인 특이값 분해의 구현을 간단히 살펴봄으로써 복잡

nodes	4	8	12	16	20	24	28
average time (s)	177.23	149.97	139.14	123.40	71.83	51.84	48.86
standard deviation (s)	74.72	57.42	41.35	47.62	20.48	16.01	9.10

Table 7.1. Running time for 300 iterations of K-means clustering

한 통계 계산 알고리즘을 스파크에서 구현하는 방법에 대해 고찰해 보았다. 또한, 이러한 알고리즘이 클러스터에서 확장 가능하게 동작함을 확인해 보았다. 아울러 통계학계에서 일반적으로 사용되는 언어인 R을 위한 스파크의 인터페이스에 대해 간단히 살펴보았다. 스파크는 하둡으로 대표되는 기존의 분산 컴퓨팅 프레임워크에 비해 구현이 쉽고, 범용성이 뛰어나고, 속도가 빠른 것을 비롯한 다양한 장점을 갖고 있다. 특히 R을 위한 개발이 활발하게 진행되고 있으므로 컴퓨터 한 대의 메모리에 올라가지 않을 정도로 큰 데이터를 다루는 일이 빈번한 고차원 자료 분석 분야의 통계학자들에게 스파크는 좋은 도구가 될 것이다.

References

Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. (2015). Spark SQL: Relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM.

Bahmani, B., Moseley, B., Vattani, A., Kumar, R., and Vassilvitskii, S. (2012). Scalable k-means++. *Proceedings of the VLDB Endowment*, 5(7):622–633.

Dean, J. and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

dplyr (n.d.). dplyr: A grammar of data manipulation. `https://github.com/hadley/dplyr`. Accessed on: 2016-08-27.

H2O.ai (n.d.a). H2O.ai - AI for Business. `http://www.h2o.ai/`. Accessed on: 2016-08-30.

H2O.ai (n.d.b). Sparkling Water. `http://www.h2o.ai/product/sparkling-water/`. Accessed on: 2016-08-30.

HBase (n.d.). Apache HBase. `https://hbase.apache.org`. Accessed on: 2016-08-27.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S., and Stoica, I. (2011). Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 13th USENIX conference on Networked Systems Design and Implementation*. USENIX Association.

Hunter, T., Moldovan, T., Zaharia, M., Merzgui, S., Ma, J., Franklin, M. J., Abbeel, P., and Bayen, A. M. (2011). Scaling the mobile millennium system in the cloud. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM.

Kim, H., Park, J., Jang, J., and Yoon, S. (2016). DeepSpark: Spark-based deep learning supporting asynchronous updates and Caffe compatibility. *arXiv preprint arXiv:1602.08191*.

Kraska, T., Talwalkar, A., Duchi, J. C., Griffith, R., Franklin, M. J., and Jordan, M. I. (2013). MLbase: A distributed machine-learning system. In *The 6th biennial Converence on Innovative Data Systems Research*.

Lakshman, A. and Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40.

Lehoucq, R. B., Sorensen, D. C., and Yang, C. (1998). *ARPACK users' guide: solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods*, volume 6. SIAM.

Meng, X., Bradley, J., Yuvaz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al. (2016). MLlib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7.

Moritz, P., Nishihara, R., Stoica, I., and Jordan, M. I. (2015). SparkNet: Training deep networks in Spark. *arXiv preprint arXiv:1511.06051*.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12(Oct):2825–2830.

RStudio (n.d.). sparklyr–R interface for Apache Spark. `http://spark.rstudio.com`. Accessed on: 2016-08-27.

Scala (n.d.). The Scala programming language. `http://www.scala-lang.org`. Accessed on: 2016-08-27.

Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The Hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. IEEE.

Spark (n.d.). Apache spark. `https://spark.apache.org/`. Accessed on: 2016-08-27.

spark-cassandra-connector (n.d.). Spark Cassandra Connector. `https://github.com/datastax/spark-cassandra-connector`. Accessed on: 2016-08-30.

spark-sklearn (n.d.). Scikit-learn integration package for Apache Spark. `https://github.com/databricks/spark-sklearn`. Accessed on: 2016-08-30.

spark-tfocs (n.d.). TFOCS for Spark: : A communy port of TFOCS for Apache Spark. `https://github.com/databricks/spark-tfocs`. Accessed on: 2016-08-27.

Spark Wiki (n.d.a). Committers. `https://cwiki.apache.org/confluence/display/SPARK/Committers`. Accessed on: 2016-08-27.

Spark Wiki (n.d.b). Powered By Spark. `https://cwiki.apache.org/confluence/display/SPARK/Powered+By+Spark`. Accessed on: 2016-08-27.

sparkit-learn (n.d.). Sparkit-learn. `https://github.com/lensacom/sparkit-learn`. Accessed on: 2016-08-30.

SparkR (n.d.). SparkR (R on spark). `https://spark.apache.org/docs/latest/sparkr.html`. Accessed on: 2016-08-27.

TopicModeling (n.d.). Topic modeling on Apache Spark. `https://github.com/intel-analytics/TopicModeling`. Accessed on: 2016-08-30.

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al. (2013). Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM.

Xin, R., Deyhim, P., Ghodsi, A., Meng, X., and Zaharia, M. (2014a). GraySort on Apache Spark by Databricks. *GraySort Competition*.

Xin, R. S., Crankshaw, D., Dave, A., Gonzalez, J. E., Franklin, M. J., and Stoica, I. (2014b). GraphX: Unifying data-parallel and graph-parallel analytics. *arXiv preprint arXiv:1402.2394*.

Zadeh, R. B., Meng, X., Ulanov, A., Yavuz, B., Pu, L., Venkataraman, S., Sparks, E., Staple, A., and Zaharia, M. (2016). Matrix computations and optimization in Apache Spark. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 31–38. ACM.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. USENIX Association.

Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. (2013). Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM.

Zeppelin (n.d.). Apache Zeppelin. `https://zeppelin.apache.org/`. Accessed on: 2016-08-30.

Apache Spark를 활용한 대용량 데이터의 처리

고세윤[a] · 원중호[a,1]

[a]서울대학교 통계학과

(2016년 08월 14일 접수, 0000년 00월 00일 수정, 2016년 10월 03일 채택)

요 약

아파치 스파크는 빠르고 범용성이 뛰어난 클러스터 컴퓨팅 패키지로, 복구 가능한 분산 데이터셋이라는 새로운 추상화를 통해 데이터를 인메모리에 유지하면서도 결함 감내성을 얻을 수 있는 방법을 제공한다. 이러한 추상화는 하드디스크에 직접 데이터를 읽고 쓰는 방식으로 결함 감내성을 제공하는 기존의 대표적인 대용량 데이터 분석 기술인 맵리듀스 프레임워크에 비해 상당한 속도 향상을 거두었다. 특히 로지스틱 회귀 분석이나 K-평균 군집화와 같은 반복적인 기계 학습 알고리즘이나 사용자가 실시간으로 데이터에 관한 질의를 하는 대화형 자료 분석에서 스파크는 매우 효율적인 성능을 보인다. 뿐만 아니라, 높은 범용성을 바탕으로 하여 기계 학습, 스트리밍 자료 처리, SQL, 그래프 자료 처리와 같은 다양한 고수준 라이브러리를 제공한다. 이 논문에서는 스파크의 개념과 프로그래밍 모형에 대해 소개하고, 이를 통해 몇 가지 통계 분석 알고리즘을 구현하는 방법에 대해 소개한다. 아울러, 스파크에서 제공하는 기계 학습 라이브러리인 MLlib과 R 언어 인터페이스인 SparkR에 대해 다룬다.

주요용어: 스파크, 기계 학습, 클러스터 컴퓨팅, 병렬 컴퓨팅

이 논문은 2016년도 정부(미래창조과학부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(Nos. 2013R1A1A1057949, 2014R1A4A1007895).

[1]교신저자: (08826) 서울특별시 관악구 관악로 1, 서울대학교 통계학과. E-mail: wonj@stats.snu.ac.kr