Journal of the Korean Data & Information Science Society
2013, **24**(5), 1013–1027

http://dx.doi.org/10.7465/jkdi.2013.24.5.1013
한국데이터정보과학회지

하둡과 맵리듀스[†]

박정혁[1] · 이상열[2] · 강다현[3] · 원중호[4]

[1234]고려대학교 산업경영공학부

접수 2013년 7월 7일, 수정 2013년 8월 7일, 게재확정 2013년 8월 12일

요 약

대용량 데이터 분석의 필요성이 급격히 증대되면서 이를 가능케 해 주는 플랫폼인 하둡과 그 내부적인 계산 모형인 맵리듀스에 대한 관심 또한 늘고 있다. 본고에서는 R 등의 통계 프로그래밍에 익숙한 데이터 분석가가 하둡을 사용하고자 할 때 알아야 할 기본 개념들을 R과 하둡을 결합하는 몇가지 예제와 함께 소개한다.

주요용어: 맵리듀스, 분산 컴퓨팅, 빅데이터 분석, 하둡.

1. 서론

정보 기술의 발전과 사회 관계망 서비스의 성장으로 도처에서 생성되는 디지털 데이터의 양이 급격히 증가하고 있는 가운데 이를 효율적으로 처리하는 기술이 요구되고 있다. 이러한 데이터의 특징으로는 이들이 다른 활동의 부산물로 축적되며 그 형태가 비정형성을 띠고 있다는 점을 들 수 있다. 예를 들어 스마트폰용 메시징 앱을 동료와의 친목을 도모할 목적으로 사용할 경우, 그 사용시간, 사용빈도, 전력소비, (만일 앱이 사용 도중 에러로 종료되었다면) 예외적 종료 시각 등에 대한 정보가 로그 파일 형태로 스마트폰 제조사나 그 운영체제의 제작사로 전송된다. 제조사 또는 제작사의 입장에서 보면, 수억 명의 스마트폰 사용자로부터 전송되어 오는 각종 앱에 대한 로그 파일의 축적량은 하루 단위로만 보아도 엄엄날 것이다. 또한 로그 파일은 대개 구조가 약한 텍스트 파일이며 이를 처리하여 스마트폰 또는 그 운영체제의 성능 향상에 필요한 정보를 추출하기 위해서는 하나의 고성능 서버나 기존의 데이터베이스 소프트웨어로는 충분치 못하다는 것을 쉽게 짐작할 수 있다. 이렇게 그 크기가 방대하여 기존의 시스템으로는 획득, 저장, 관리, 혹은 분석이 불가능한 데이터를 흔히 빅데이터라고 부른다 (McKinsey Global Institute, 2011).

빅데이터 처리를 위한 분산 처리 플랫폼으로 아파치 재단의 하둡 (Apache Hadoop; 이하 하둡)이 주목받고 있다. 데이터가 하나의 서버에서 처리할 수 없을 정도로 큰 경우에는 이를 여러 서버에 분산시켜 놓고 동시에 여러 대의 서버가 나누어 처리하도록 하는 방식이 보편적이며, 분산 시스템 관련 기술 자체는 1980년대부터 꾸준히 연구되어 온 것으로 크게 새로운 것은 아니다. 하둡의 특징은 분산 시스템 환경이 데이터 처리 위주로 구성되어 있다는 점이다. 즉, 하둡에서는 데이터의 위치를 추적하는 방식으

[†] 본 연구는 고려대학교 특별연구비에 의하여 수행되었음.
[1] (136-713) 서울시 성북구 안암로 145, 고려대학교 산업경영공학부, 대학원생.
[2] (136-713) 서울시 성북구 안암로 145, 고려대학교 산업경영공학부, 대학원생.
[3] (136-713) 서울시 성북구 안암로 145, 고려대학교 산업경영공학부, 대학원생.
[4] 교신저자: (136-713) 서울시 성북구 안암로 145, 고려대학교 산업경영공학부, 조교수.
E-mail: wonj@korea.ac.kr

로 분산 파일 시스템을 구현하며, 이에 따라 사용자가 원하는 만큼 스토리지 및 컴퓨팅 자원을 확장할 (scalable) 수 있다. 또한 일부 서버가 고장나도 중단 없이 작동이 가능하여 (fault tolerant) 저가의 서버로 분산 시스템을 구성할 수 있다 (Seo 등, 2013). 오픈소스로 시스템 구축에 큰 비용이 들지 않는다는 점 또한 하둡의 장점이다.

하둡은 자바 언어로 작성되어 있으며 크게 아래의 3가지 모듈로 구성되어 있다 (하둡 1.0 기준).

- 하둡 공통 (Hadoop Common): 다른 모듈들을 지원하는 공통 유틸리티 모음. 하둡을 기동하는 스크립트나 분산 파일 시스템에의 접근을 돕는 유틸리티 등이 포함된다.
- 하둡 분산 파일 시스템 (Hadoop distributed file system; 이하 HDFS): 하둡 공통을 이용하여 대용량 데이터를 다수의 컴퓨터에 분산시켜 고속으로 처리하기 위한 분산 파일 시스템.
- 하둡 맵리듀스 (Hadoop MapReduce): 분산 파일 시스템에 저장된 대용량 데이터의 병렬 처리를 위한 소프트웨어 프레임워크.

위와 같은 구성으로 인해 프로그래머는 하둡 플랫폼 상에서의 분산 처리를 위해 일반적인 분산 시스템 프로그래밍에서처럼 데이터를 분할하거나 어느 서버에서 어떤 작업을 실행해야 할 지와 같은 고민을 할 필요 없이 데이터 자체와 그 데이터를 가지고 무엇을 할 것인가에 대해서만 집중할 수 있다. 그 결과 대용량 데이터 처리가 필요한 많은 분야에 하둡을 활용하는 연구가 활발히 진행되고 있다. 예를 들어 대용량 스팸 메일 처리 (Cho 등, 2009), 비디오 트랜스코딩 (McKinsey Global Institute, 2011), 다중 염기서열 정렬 (Park 등, 2011), 내용 기반 음악 검색 (Jung 등, 2011) 등 다양한 분야에의 응용을 찾을 수 있다.

본고에서는 대용량 데이터의 처리 및 분석을 위해 하둡을 사용하고자 할 때 알아야 할 기본 개념들을 예제와 함께 소개하고자 한다. 이해를 돕기 위해 우선 제 2절에서 간략한 하둡의 역사를 다루고나서 제 3절에서 데이터 분석 프로그래밍을 위해 필요한 HDFS와 맵리듀스의 핵심 개념들을 설명한다. 제 4절에서는 기본적인 데이터 분석 알고리즘을 통계 소프트웨어인 R과 연동하여 하둡으로 구현해 본다. 마지막으로, 제 5절에서는 앞 절에서 논의한 바를 간략히 정리하고 추후의 과제를 제기함으로써 글을 맺는다.

2. 하둡의 역사

하둡의 역사를 소개하기에 앞서 본 절의 다음 두 문단은 영문 위키피디어 페이지 `http://en.wikipedia.org/wiki/Apache_Hadoop` (2013년 6월 접근)과 Seo 등 (2013)을 참조하였음을 밝혀 둔다.

하둡은 오픈소스 개발자인 더그 커팅 (Doug Cutting)과 마이크 카파렐라 (Mike Cafarella)에 의해 2006년에 발표되었다. 하둡의 개발 이전 커팅과 카파렐라는 너치 (Apache Nutch)라는 오픈소스 검색 엔진 프로젝트를 주도하고 있었는데, 너치 검색 엔진은 그 구조상 10억 페이지 이상의 웹 페이지 색인을 관리하는 데 한계를 갖고 있었다. 당시 전체 웹 사이트 규모는 수십 억 페이지 이상으로, 너치만으로는 실용성 있는 검색 엔진의 구현이 어려울 것임이 자명했다. 때마침 구글 (Google Inc.)이 자사의 분산 파일 시스템의 구조에 대한 논문을 발표하였다 (Ghemawat 등, 2003). 이 논문은 검색 엔진의 색인 구축 시 생성되는 거대한 파일 관리에 대한 노하우를 담고 있었고, 커팅은 이 논문의 아이디어를 대부분 차용하여 너치를 위한 분산 파일 시스템을 구현하였다. 이 너치 분산 파일 시스템 (Nutch distributed file system; NDFS)이 후일 HDFS의 전신이 된다. 다음 해 구글은 자사의 분산 파일 시스템 위에서 동작하여 대용량 데이터를 손쉽게 처리할 수 있는 맵리듀스 프레임워크를 발표한다 (Dean과 Ghemawat, 2004). 맵리듀스는 곧 너치 프로젝트에도 도입되었다. 이제 너치는 NDFS와 맵리듀스를 이용하여 수십 억 페이지 상당의 데이터를 저장할 수 있고, 이 많은 데이터를 분산 처리할 수 있게 되었다. 커팅은

NDFS와 맵리듀스가 검색 엔진을 뛰어넘어 많은 분야에 다양하게 사용될 수 있음을 깨닫고 2006년 2월 이들을 너치 프로젝트에서 독립시켜 "하둡"이라 명명하였다. ("하둡"은 당시 커팅의 아들이 가지고 놀던 코끼리 장난감의 이름이라고 한다. 이후 코끼리는 하둡의 상징이 되었다.)

하둡 프로젝트는 야후 (Yahoo! Inc.)의 전폭적인 지원을 받아 급속히 성장하였다. 2006년 당시 야후는 검색 시장에서 구글에 1위 자리를 내준 이후 지속적으로 그 격차가 벌어져 가는 상황이었다. 검색 품질을 개선하고 구글을 따라잡기 위해 야후는 자사 검색 엔진의 웹맵 (WebMap, 크롤러를 이용하여 수집된 웹 페이지들 간의 관계에 대한 데이터베이스를 구축하는 과정) 부분을 하둡으로 대체하는 커다란 결정을 내린다. 야후는 더그 커팅을 영입하고 하둡 프로젝트를 적극적으로 지원하였다. 야후 내에서 웹맵뿐만 아니라 다른 서비스들에도 하둡이 빠르게 적용되었고, 약 2년 뒤인 2008년 2월, 하둡이 아파치 재단 내 최고 수준 오픈소스 프로젝트로 격상됨과 동시에 야후는 자사 서비스의 인덱스가 1만 개의 하둡 코어를 이용하여 생성되고 있다고 발표하였다. 1년여 후인 2009년 5월에는 62초만에 1TB (테라바이트, 1TB는 약 1000GB)의 데이터를 디스크에서 읽어와 정렬한 후 다시 저장하는 데 성공했다. 당시 1TB 용량의 하드디스크의 전송 속도가 초당 0.1GB 수준인 것을 감안할 때 놀라운 결과였다. 현재 야후는 세계에서 가장 큰 하둡 클러스터를 보유한 회사가 되었다. 하둡 공식 홈페이지 (`http://wiki.apache.org/hadoop/PoweredBy`)에 따르면 2013년 6월 현재 4만여 대의 컴퓨터에 탑재된 10만여 개의 CPU에서 하둡이 운영 중이며, 이중 가장 큰 클러스터는 4500대의 서버로 구성되었다. 야후 외에 하둡을 대규모로 활용하고 있는 업체로는 페이스북이 있다. 자사 홈페이지에 따르면 이 회사는 2012년 말 현재 100PB (페타바이트, 1PB는 약 1000TB)의 스토리지로 구성된 하둡 클러스터를 갖추고 있으며, 이를 통해 수억 명의 사용자로부터 생성되는 방대한 내부 로그 등을 기계학습 알고리즘을 이용해 분석하고 있다 (Facebook Engineering Team, 2012).

하둡 프로젝트가 성공적으로 안착하자 대용량 데이터 처리와 분석에 어려움을 겪고 있던 많은 기업에서 하둡을 그 해법으로 이용하고자 하는 수요가 폭발적으로 늘어났다. 그러나 오픈소스로 제공되는 하둡을 기업 환경에서 요구되는 품질 수준에 맞게 설치하고 튜닝하는 일은 쉽지 않은 작업이었고 많은 곳에서 어려움을 호소하였다 (Harris, 2011). 이에 설치가 쉬우면서도 상업적 소프트웨어 수준의 안정성을 갖춘 하둡 배포판을 제작, 판매하는 회사들이 하나 둘씩 생겨나기 시작했고, 이들이 향상된 하둡 관련 코드로 오픈소스 하둡 프로젝트에 기여하면서 본격적인 하둡 생태계가 생성되기 시작한다. (오픈소스 프로젝트에 기반한 상업적 배포판을 제작하고 상업적 노력의 결과로 향상된 품질의 프로그램 소스 코드를 원래의 오픈소스 프로젝트에 돌려주는 것은 오픈소스 생태계의 일반적인 양상이다. 예를 들어 애플 (Apple Inc.)은 자사의 매킨토시 운영체제에 차용한 FreeBSD 운영체제의 향상에 많은 기여를 했다. 야후도 하둡의 초창기인 2009년 6월 자사의 서비스에 사용하는 하둡 소스 코드를 공개했고, 꾸준히 오픈소스 하둡의 버그 수정과 안정성 향상에 기여하고 있다.) 대표적인 회사로 클라우데라 (Cloudera Inc.)를 들 수 있는데, 이 회사는 CDH (Cloudera distribution including Apache Hadoop)로 불리는 배포판을 리눅스 운영체제 이미지 형태로 무료 공개하고 있으며, 기업에게는 관리도구인 클라우데라 매니저와 기술적 지원을 유상으로 제공하고 있다. (더그 커팅은 2013년 현재 이 회사의 수석 개발자로 일하고 있다.) 기업용 하둡 배포판을 제작하는 다른 회사로는 2011년 야후에서 분사한 호튼워크스 (HortonWorks Inc.)와 아마존 (Amazon Inc.)의 빅데이터 분석용 클라우드 컴퓨팅 서비스인 EMR (Elastic MapReduce)에 하둡 맵리듀스를 제공하는 맵아르 (MapR Technologies Inc.) 등이 있으며, 대체로 이들 세 회사 간의 경쟁 관계가 성립되어 있다.

상업용 배포판 등의 등장으로 소프트웨어 엔지니어가 아닌 일반인들도 사용할 수 있을 정도로 하둡을 이용한 대용량 데이터 저장 및 처리의 기반이 갖춰지자 실제 대용량 데이터를 이용한 고급 분석에 대한 수요가 대두하였다. 비근한 예로, 아파치 재단의 하둡 프로젝트의 일부이자 하둡 기반의 기계학습 라이브러리인 마하웃 (Apache Mahout)을 들 수 있다. 마하웃은 군집분석, 분류, 협업적 필터링

(collaborative filtering) 등 대표적인 기계학습 알고리즘을 스크립트로 구현하여 맵리듀스를 모르는 사용자라도 대용량 데이터 분석을 수행할 수 있도록 하였으나, 지원하는 알고리즘의 수가 제한되어 있어 유연성이 떨어지는 단점이 있다. 직접 데이터 분석과 통계 알고리즘을 하둡 플랫폼 상에서 구현하고자 하는 사람들은 또다른 오픈소스 프로젝트이자 널리 사용되는 통계 소프트웨어인 R과 하둡의 결합에 눈을 돌리게 되었다. R은 유연성 있는 통계 프로그래밍 환경을 제공하지만 컴퓨터의 주메모리에 올릴 수 있는 크기의 데이터만 처리할 수 있어 대용량 데이터 분석에는 한계를 갖고 있다. 하둡의 분산처리 환경과의 결합으로 이 한계를 극복할 수 있다면 많은 데이터 분석가와 통계학자들이 손쉽게 대용량 데이터 분석 알고리즘을 구현할 수 있을 터였다. 이러한 노력의 일환으로 퍼듀 대학의 통계학과 대학원생이었던 샙타르시 구하 (Saptarshi Guha)는 2009년경 R로 맵리듀스 프로그래밍을 가능하게 한 Rhipe ('리피'라고 읽는다)를 개발한다 (Guha 등, 2009; Guha, 2010). 구하는 이후 병렬처리가 가능한 상업용 R 배포판을 개발하는 레볼루션 어낼리틱스 (Revolution Analytics Inc.)에 합류하여 이 회사의 하둡 기반 R 패키지 모음인 RHadoop의 개발에 일조한다. RHadoop은 R에서 HDFS 접근을 가능케하는 `rhdfs`, R 상의 맵리듀스 프로그래밍을 지원하는 `rmr`, 그리고 R을 통해 분산 데이터베이스인 HBase의 관리를 지원하는 `rhbase`의 세 패키지로 구성되어 있다 (Revolution Analytics, 2011).

3. 기본 개념

3.1. 하둡 분산 파일 시스템 (HDFS)

HDFS는 일반적인 분산 파일 시스템의 설계를 따라 마스터 (master) 노드와 슬레이브 (slave) 노드로 구성되어 있다. HDFS의 마스터 노드는 네임노드 (namenode)라 불리며, 데이터노드 (datanode)로 불리는 슬레이브 노드의 동작 상태를 실시간으로 관리하고 최대 수천 대의 데이터노드에 분산 저장되어 있는 데이터에 대한 메타데이터를 관리하는 일을 한다 (Figure 3.1 하단). 데이터는 여러 개의 블록으로 쪼개어져 몇 대의 데이터노드가 한 블록을 복제해서 보관하는 방식으로 분산 저장된다. 네임노드가 관리하는 메타데이터는 해당 블록이 어느 데이터노드에 저장되어 있는지의 여부 등이다. 이와 같은 방식을 사용하면 데이터노드 한두 대가 고장나더라도 데이터를 잃지 않기 때문에 고장감내성 (fault tolerance)이 확보된다. HDFS는 수십 PB에 이르는 대용량 데이터를 수천 대의 서버를 이용하여 빠르게 처리할 수 있도록 설계되었으며, 비용 문제로 저가의 서버를 이용하는 것을 전제로 하고 있다. 이럴 경우 디스크나 서버의 고장이 자주 발생할 수 있기 때문에 고장감내성이 중요한 고려 요소가 된다. 또한 스토리지 용량이 부족해지면 저가의 데이터노드를 추가하여 네임노드에 등록하는 것으로 시스템의 중단 없이 용량을 확장시킬 수 있다. 이러한 방식으로 하둡의 확장성 (scalability)이 구현된다.

여타의 분산 파일 시스템에 비해 HDFS는 메타데이터에 접근하거나 데이터를 변경하는 작업의 대기시간 (latency)을 희생하는 대신 데이터를 읽어오는 작업의 처리량 (throughput)을 높여 큰 데이터를 한번에 빠르게 가져올 수 있도록 설계되었다. 한 예로, HDFS에서 한 블록의 크기는 기본적으로 64MB로 설정되어 있는데, 이는 일반적인 파일 시스템의 블록 사이즈인 수십 KB와는 큰 차이를 보인다. 이러한 설계는 하둡이 웹 페이지 색인 관리라는 기원 상 시시각각 변하는 데이터의 실시간 처리가 아닌 기존에 수집된 데이터의 배치 처리에 최적화된 플랫폼이라는 데에 기인한다. 데이터의 배치 처리를 위해서는 수집된 데이터를 수정할 필요가 없으므로 HDFS는 한 번 쓰기 완료된 데이터는 수정이 불가능하고 오직 덮어쓰기만이 가능하도록 설계되었다. 이처럼 데이터 저장 방식이 간단해지면 전체 시스템의 관리가 간편해져 수천 대의 서버로 구성된 클러스터도 무리 없는 운영이 가능해진다. 또한 후술할 맵리듀스의 구현도 간단해진다.

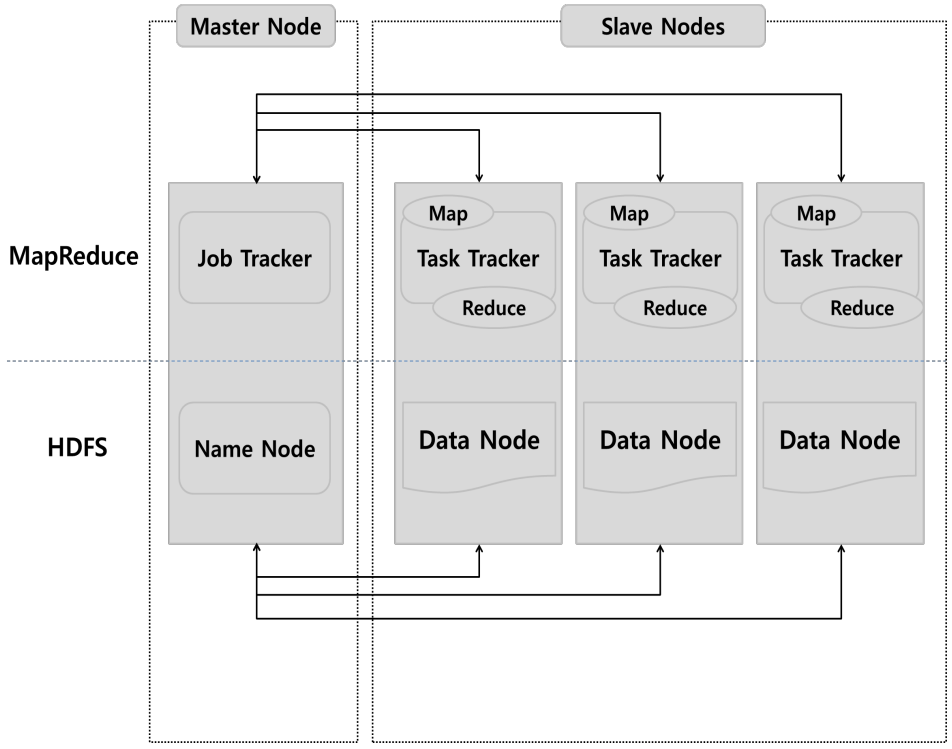


Figure 3.1 Basic architecture of the Hadoop system

3.2. 맵리듀스 (MapReduce)

하둡 맵리듀스는 HDFS 상에서 동작하는 데이터 분석 프레임워크이다. 맵리듀스는 일반 프로그래밍 방법과는 다른 데이터 중심 프로그래밍 모형을 제공한다. 일반적인 분산 환경에서의 프로그래밍은 대개의 프로그래머가 익숙한, 단일 서버에서의 프로그래밍과 달리 분산된 작업의 스케줄링이나 일부 서버의 고장, 서버 간 네트워크 구성 등 많은 문제를 고려해야한다. 맵리듀스에서는 이런 복잡한 문제들이 플랫폼 차원에서 단순화되어 프로그래머는 데이터의 배치 처리를 위한 맵 (mapper)과 리듀스 (reducer) 함수만을 작성하면 되도록 구현되어 있다.

맵리듀스 프로그래밍 모형에서는 함수의 입력과 출력이 모두 키 (key)와 값 (value)의 쌍으로 구성되어 있다. 각 함수는 <키, 값> 쌍의 집합을 또다른 <키, 값> 쌍의 집합으로 변환한다. 맵 함수는 HDFS에서 불러온 데이터를 가공하여 새로운 <키, 값> 집합을 출력한다. 맵리듀스 시스템에서는 같은 키를 갖는 값들을 묶어 <키, (값1, 값2,$\cdots$)> 식의 새로운 <키, 값> 쌍의 집합을 만든다. 리듀스 함수는 여기에 집계 연산을 수행하여 또다른 <키, 값> 쌍의 집합을 생성하고 이를 HDFS에 저장한다.

예를 들어 문서 집합에 등장하는 단어의 갯수를 세는 작업을 생각해 보자 (The Apache Software Foundation, 2008). Figure 3.2에서처럼 3개의 문서가 HDFS에서 입력으로 주어졌다고 하면, 3개의 맵 태스크 (task)가 발생한다. 각 맵 태스크는 동일한 맵 함수를 실행하는데, 입력으로 문서의 ID가 키, 문서 텍스트가 값인 <키, 값> 쌍이 주어지면 이 맵 함수는 문서 텍스트를 단어 단위로 쪼개어 각 단어가 키가 되고 값은 1로 고정된 새로운 <키, 값> 쌍의 집합을 출력한다. 맵리듀스 시스템은 이 집합의 원

소들에 대해 섞기 및 정렬 (shuffle & sort) 과정을 통해 동일한 키를 가지는 값들을 하나로 묶는다. 이렇게 묶여진 <키, 값> 쌍들은 적절한 기준에 의해 2개의 리듀스 태스크로 분배 (partition)된다. 각 리듀스 태스크는 역시 동일한 리듀스 함수를 실행하는데, 이 함수는 묶여진 값들을 모두 더하여 입력과 동일한 키의 새로운 값으로 만든다. 그 결과가 HDFS에 저장되며 이는 3개의 문서 집합에서 나타나는 단어의 빈도수가 됨을 알 수 있다. (이 예제에서 맵 태스크 및 리듀스 태스크의 갯수는 임의로 잡은 것이며, 실제로는 맵리듀스 시스템에서 자동적으로 생성한다.)

위의 예제에서 프로그래머는 중간의 섞기 및 정렬, 분배 과정에 신경 쓸 필요 없이 맵 함수와 리듀스 함수만 작성하면 된다. 해당 맵 함수의 의사 코드는 아래와 같다.

```
map(key:document, value:text) {
    for word in tokenize(value)
        emit(word, 1)
}
```

또한 해당 리듀스 함수의 의사 코드는 아래와 같다.

```
reduce(key:word, value: list of 1s) {
    emit(key, sum(value))
}
```

같은 예제에서 만일 전체 문서의 수가 3개가 아닌 10억 개이고 문서 하나의 평균 크기가 100KB라면, 그 전체 용량은 약 100TB가 되며, 이 정도 크기의 데이터를 (저장 가능한 스토리지가 존재한다는 가정하에) 일반적인 프로그래밍 방식으로 한 대의 컴퓨터에서 순차 처리한다면 그 수행 시간이 매우 많이 걸릴 것임은 자명하다. 만일 컴퓨터 한 대의 처리량이 초당 100MB라면 대략 1백만 초, 즉 12일이 걸릴 것이다. 같은 처리량을 가진 컴퓨터 1,000대로 분산 처리한다면 그 시간은 (분산 처리의 오버헤드를 무시할 때) 1,000초, 즉 17분으로 줄어들 것이다. 분산 처리의 오버헤드를 줄이기 위해 맵리듀스는 노드 간에 데이터를 보내는 대신 계산 코드를 데이터가 있는 곳으로 보내는 방법을 채택하였다. 따라서 데이터의 연산은 실제 데이터가 위치한 컴퓨터에서 바로 이루어지며, 새로운 연산의 경우 해당 코드를 다시 노드에 보내 처리하는 방식을 사용함으로써 노드 간에 데이터 전송을 최소화한다 (Lam, 2012).

노드 간 데이터 전송을 최소화하기 위한 또다른 방법은 컴바인 함수 (combiner)를 사용하는 것이다. 기본적으로 맵 태스크에서 생성된 결과물이 리듀스 태스크의 입력으로 들어가기 때문에 많은 ¡키, 값¿ 쌍이 한 노드로 전달된다면 데이터 전송 오버헤드가 커진다. 이를 감소시키기 위해 맵 함수는 선택적으로 컴바인 함수를 가질 수 있다. 컴바인 태스크는 맵 태스크와 동일 노드에서 실행되며 맵 태스크의 출력에 같은 키를 가진 <키, 값>쌍이 있다면, 이들을 리듀서에 보내기 전에 묶어버린다. 따라서 교환 법칙과 결합 법칙이 성립하는 연산의 경우 리듀스 함수와 동일한 컴바인 함수를 사용하여 오버헤드를 줄일 수 있다. Figure 3.2에서는 '`hello`'가 세 번째 문서에서 두 번 나타나므로 <`hello`, 1>, <`hello`, 1>이 <`hello`, 2>로 컴바인되어 리듀스 태스크로 보내진다.

소프트웨어 구조상 맵리듀스 역시 마스터-슬레이브 구조로 구성되어 있다. 맵리듀스의 마스터 노드는 잡 트래커 (job tracker)라고 불리며, 위에 언급한 맵 태스크와 리듀스 태스크, 컴바인 태스크 등 태스크들의 스케줄링을 담당한다. 슬레이브 노드는 태스크 트래커 (task tracker)라고 불리며 데이터 노드에 위치하여 이곳에서 실행되는 태스크의 실행과 관리를 담당한다 (Figure 3.1 상단).

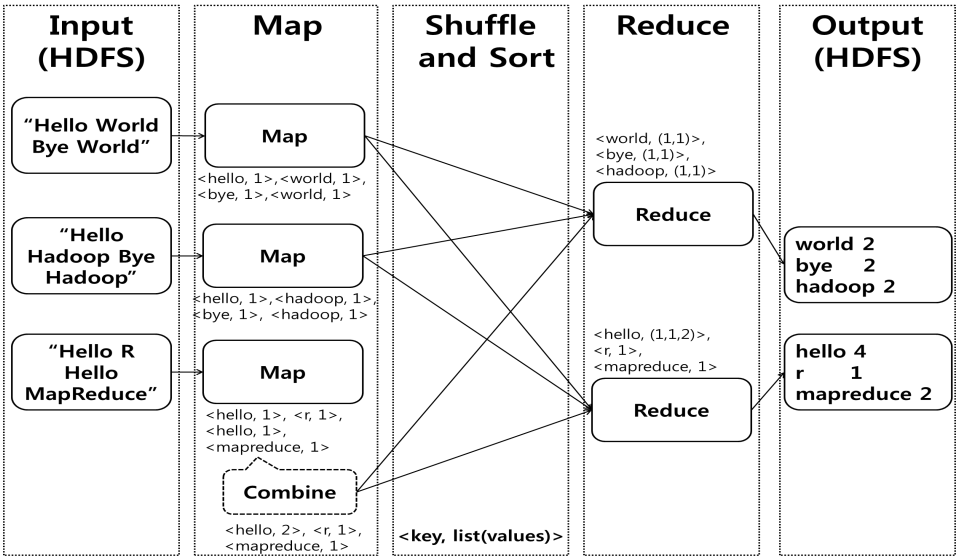


Figure 3.2 Word counting using MapReduce

3.3. 완전 분산 모드와 의사 분산 모드

하둡은 여러 대의 서버로 구성되어 있는 완전 분산 모드 (fully distributed mode)에서 작동하는 것을 기본으로 설계되었으나 맵리듀스 프로그램 개발 및 테스트, 버그 수정 등의 용이성을 위해 하둡에 관련된 모든 프로세스가 한 대의 서버에서 동작하는 의사 분산 모드 (pseudo-distributed mode)로 동작하도록 설치할 수 있다. 이렇게 구축된 하둡 시스템은 네임노드와 데이터노드, 잡 트래커와 태스크 트래커가 모두 하나의 컴퓨터에서 동작한다. 리눅스 운영체제에서의 완전 분산 모드와 의사 분산 모드 설치에 대해서는 Seo 등 (2013)을 참조하라.

4. RHadoop을 이용한 통계 계산 알고리즘의 맵리듀스 구현

본 절에서는 상술한 RHadoop을 이용하여 R에서 간단한 통계 계산 알고리즘을 맵리듀스로 구현하는 예를 보인다. 본 절에서 사용하는 예제는 Piccolboni (2013)를 참조하였으며, 모든 실험은 3.40GHz Intel i5 CPU, 4GB 메모리, 760GB HDD를 탑재한 우분투 12.04 리눅스 워크스테이션에서 Hadoop 1.0.4 의사 분산 모드로 수행하였다.

참고로, RHadoop에서는 상당히 투명한 방법으로 R과 하둡간의 연동을 제공한다. 의사 분산 환경 기준으로, 하둡이 설치되어 동작 중일 때, 아래와 같은 명령만으로 R에서 HDFS와 맵리듀스를 사용할 준비가 끝난다.

```
Sys.setenv(HADOOP_HOME=path_to_hadoop_home)
Sys.setenv(HADOOP_CMD=path_to_hadoop_binaries)
library(rmr2)
library(rhdfs)
```

HDFS에서 R의 메모리 공간으로 <키, 값> 쌍을 가져오기 위해서는 함수 `from.dfs()`를 사용하며, 그 반대의 경우는 `to.dfs()`를 사용한다.

4.1. 선형 회귀 분석

선형 회귀 분석의 경우 R에서 `lm()` 명령으로 간단히 수행할 수 있지만 여기서는 표본 크기가 매우 크다고 가정하자. 계획 행렬을 $X \in \mathbb{R}^{n \times p}$, 반응 변수 벡터를 $y \in \mathbb{R}^n$라 할 때 최소제곱법에 의한 선형 회귀 계수 $\hat{\beta}$는 정규 방정식

$$(X^T X)\hat{\beta} = X^T y \tag{4.1}$$

의 해로 주어진다. 이때, 표본 크기 n이 매우 크다면 변수 갯수 p가 많이 크지 않더라도 단일 서버에서는 행렬 X를 메모리에 한꺼번에 올리기 어려워지고 따라서 양변의 $X^T X$, $X^T y$의 계산이 힘들어진다. 만일 하둡에서 X가 행단위로 분할 및 분산 저장되어 있다고 하면, $X^T X$의 계산은 RHadoop의 `rmr`에서 다음과 같이 수행할 수 있다.

```
XtX = values( from.dfs( mapreduce( input = X,
                        map = function(., Xi) keyval(1, list(t(Xi) %*% Xi)),
                        reduce = function(., YY) keyval(1, list(Reduce(' +' , YY))),
                        combine = TRUE )
                      )
            )[[1]]
```

만일 X의 n개 행이 L개로 분할되었다고 하면 $X = [X_1^T, X_2^T, \ldots, X_L^T]^T$라고 쓸 수 있고 위의 맵 함수 (`mapreduce` 함수의 `map` 인자)는

$$< 1, X_1^T X_1 >, < 1, X_2^T X_2 >, \ldots, < 1, X_L^T X_L >$$

의 <키, 값> 쌍 집합을 출력한다. 이 집합이 섞기 및 정렬 과정을 거쳐 리듀스 함수 (`mapreduce` 함수의 `reduce` 인자)의 입력

$$< 1, (X_1^T X_1, X_2^T X_2, \ldots, X_L^T X_L) >$$

이 되고, 리듀스 함수는 이 <키, 값> 쌍에 대해 값의 모든 원소를 더하여

$$< 1, X_1^T X_1 + X_2^T X_2 + \cdots + X_L^T X_L >$$

을 출력하여 HDFS에 저장한다. 모든 <키, 값> 쌍이 동일한 키 (1)를 가지므로 컴바인 함수를 사용하여 데이터 전송 오버헤드를 줄일 필요가 있으며, 행렬의 합은 교환 및 결합 법칙이 성립하므로 리듀스 함수와 동일한 컴바인 함수를 사용한다 (`mapreduce` 함수의 `combine` 인자). 함수 `from.dfs()`는 앞서 언급한 대로 HDFS에서 R의 메모리 공간으로 <키, 값> 쌍을 가져오는 명령이며, `values()`는 그중 값만을 취한다. 이 값이 $p \times p$ 행렬 $X^T X$임은 쉽게 알 수 있다. 변수 갯수 p가 표본 크기 n에 비해 많이 작다면 계산된 $X^T X$ 자체는 메모리에 어려움 없이 로드될 수 있다.

이제 $X^T y$도 완전히 같은 방식으로 계산할 수 있다.

```
Xty = values( from.dfs( mapreduce( input = X,
                        map = function(., Xi) keyval(1, list(t(Xi) %*% y)),
                        reduce = function(., YY) keyval(1, list(Reduce(' +' , YY))),
                        combine = TRUE )
                      )
            )[[1]]
```

마지막으로 (4.1)을 풀어 원하는 선형 회귀 계수를 얻는다.

```
betahat = solve(XtX, Xty)
```

이제 맵리듀스 알고리즘의 성능을 평가하기 위해서 UCI machine learning repository (Bache와 Lichman, 2013)에서 제공하는 가구별 전력 소비 (individual household electric power consumption) 자료를 이용하여 RHadoop으로 구현된 선형 회귀 분석 코드를 실행하였다. 이 자료는 약 207만 개의 개체, 9개의 변수로 구성되어 있다. 알고리즘 성능을 검증하기 위해 10만, 50만, 100만, 200만 개의 표본을 비복원 추출하여 표본 크기에 따라 수행 시간이 `lm()`과 어떤 차이를 보이는지 살펴보았다 (Figure 4.1). 실험이 의사 분산 모드로 한 대의 컴퓨터에서 수행되었으므로 프로세스 간 통신 오버헤드 등을 고려하면 맵리듀스 구현이 `lm()`보다 수행 시간이 다소 늘어나는 것을 볼 수 있으나 표본 크기가 커질 수록 그 격차가 급격히 줄어드는 것을 관찰할 수 있다 (개체 수가 10만일 때 26배, 200만일 때 1.9배). 보다 큰 자료에 대해, 복수 개의 서버를 이용하여 완전 분산 모드로 수행한다면 이 격차는 역전될 것으로 예상된다.

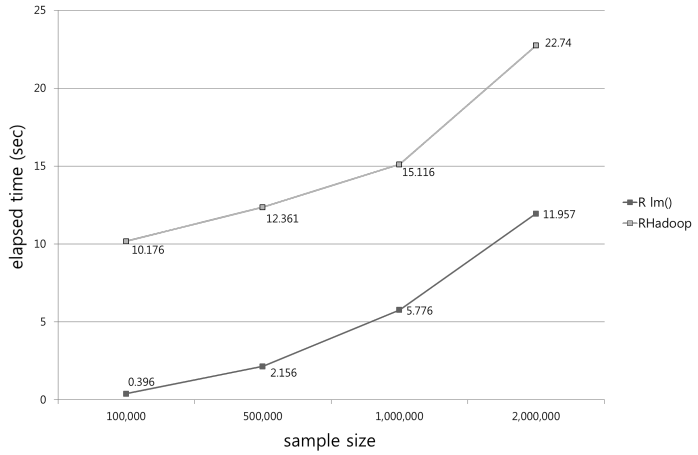


Figure 4.1 Execution time of the linear regression example

4.2. 로지스틱 회귀 분석

여기서는 로지스틱 회귀 분석의 계수 추정을 맵리듀스를 이용하여 경사하강법 (gradient descent method)으로 행하는 방법에 대해 알아본다. 로지스틱 회귀 분석 모형

$$P(y=+1)=\frac{1}{1+\exp(-\beta^T x)},\quad \beta\in\mathbb{R}^p, x\in\mathbb{R}^p, y\in\{+1,-1\}$$

의 로그 우도 함수는 아래와 같이 주어진다.

$$l(\beta)=\sum_{i=1}^{n}\log g(y_i\beta^T x_i).$$

여기서 $g(z)=1/(1+e^{-z})$이다. 따라서 로그 우도 함수 $l(\beta)$의 기울기는

$$\nabla l=\sum_{i=1}^{n} y_i g(-y_i\beta^T x_i)x_i \tag{4.2}$$

가 되므로 경사하강법에 의한 계수 갱신 법칙은 적당한 $\alpha > 0$에 대해

$$\beta := \beta + \alpha \sum_{i=1}^{n} y_i g(-y_i \beta^T x_i) x_i \tag{4.3}$$

가 된다.

식 (4.3)의 맵리듀스 구현은 앞 절의 선형 회귀 분석과 비슷하게 식 (4.2)의 각 항을 맵 함수로 계산하고 그 합을 리듀스 함수로 구하도록 함으로써 이루어진다. 따라서 맵 함수는 다음과 같다.

```
lr.map = function(., M) {
    y = M[,1]
    X = M[,-1]
    keyval( 1, y * X * 1/(1 + exp(-y * as.numeric(X %*% t(beta)))) )
}
```

단, 맵 함수의 입력은 반응 변수 벡터 $\mathbf{y} \in \{-1, +1\}^n$와 계획 행렬 $X = [x_1, x_2, \ldots, x_n]^T \in \mathbb{R}^{n \times p}$가 열 단위로 묶인 행렬이 <키, 값> 쌍의 값이 된다. 리듀스 함수는 다음과 같다.

```
lr.reduce = function(k, Z)
    keyval(k, t(as.matrix(apply(Z,2,sum))))
```

이제 전체 경사하강법의 맵리듀스 구현은 다음과 같다.

```
lr.mr = function(input, iterations, p, alpha) {
    beta = t(rep(0, p))
    for (i in 1:iterations) {
        gradient = values( from.dfs( mapreduce( input,
                                        map = lr.map,
                                        reduce = lr.reduce,
                                        combine = T)
                                    )
                           )
        beta = beta + alpha * gradient
    }
    beta
}
```

선형 회귀 분석의 경우와 같이 맵 함수의 출력이 동일한 키 (1)를 가지므로 컴바인 함수가 사용되었다. 추정 알고리즘의 실행 예는 다음과 같다.

```
lr.mr(testdata, 20, 8, 0.05)
```

맵리듀스 알고리즘의 성능을 평가하기 위해 UCI machine learning repository (Bache와 Lichman, 2013)에서 제공하는 Poker Hand 자료를 이용하였으며, 이 자료는 약 100만 개의 개체, 11개의 독립 변수로 구성되어 있다. 반응 변수를 이진 (binary)으로 처리하기 위해 'Nothing In Hand'인 경우는 반응 변수를 -1로 하였으며 'One Pair' 등과 같이 특정한 포커패가 있는 경우는 모두 1로 하였다. 알고리

즘의 성능을 검증하기 위해 R과 RHadoop을 이용한 구현의 수행시간을 비교하였다. 전체 자료에서 각각 40만, 80만 개의 표본을 비복원 추출하였으며, 독립변수를 각각 4개, 8개로 구분하여 비교하였다. 또한 경사하강법의 반복회수를 각각 1회와 2회로 나누어 비교 실험을 하였다. Table 4.1의 결과에서 데이터 수 40만, 반복 1회, 독립변수 4개인 경우, R과 RHadoop의 수행시간을 비교해보면, 각각 4.016초와 41.567로 약 10.4배의 수행시간 차이를 보인다. 그러나 데이터 수 80만, 반복 2회, 독립 변수 8개인 경우를 비교해면, 각각 17.462초와 94.940초로 약 5.4배의 수행 시간 차이를 보였다. 즉, 선형 회귀 분석의 경우와 마찬가지로 표본 크기가 커질 수록 R과 RHadoop의 수행시간 격차가 감소함을 볼 수 있다. 역시 보다 큰 자료에 대해, 복수 개의 서버를 이용하여 완전 분산 모드로 수행한다면 이 격차는 역전될 것으로 예상된다.

Table 4.1 Execution time of the logistic regression example (unit: second)

	R				RHadoop			
	1 iteration		2 iterations		1 iteration		2 iterations	
	$p = 4$	$p = 8$	$p = 4$	$p = 8$	$p = 4$	$p = 8$	$p = 4$	$p = 8$
$n = 400,000$	4.016	4.954	7.672	9.234	41.567	42.182	81.980	83.035
$n = 800,000$	7.280	9.642	13.889	17.462	43.894	56.099	91.492	94.940

4.3. K-평균 알고리즘

K-평균 알고리즘은 대표적인 군집화 (clustering) 알고리즘으로 크게 아래의 4단계로 구성된다.

1. 원하는 군집의 수 K를 정한다.
2. 각각의 자료점 (data point)과 K개의 군집 중심과의 거리를 계산한 후 가장 가까운 군집으로 데이터를 분류한다.
3. 군집 내의 자료점과의 평균 거리를 최소화하는 점으로 군집 중심을 이동시킨다.
4. 군집 중심의 변화가 없을 때까지 2단계로 돌아가 알고리즘을 수행한다.

유클리드 거리를 사용할 경우 3단계의 군집 중심은 군집 내의 자료점들의 좌표의 평균으로 얻어진다.

맵리듀스 알고리즘은 2단계와 3단계를 각각 맵 함수와 리듀스 함수로 사용한다. 맵 함수는 다음과 같다.

```
kmeans.map = function(., points) {
    nearest = {
      if(is.null(centers))
          sample( 1:K, nrow(points), replace = T)
      else {
          D = apply( centers, 1, function(x) colSums((t(points) - x)^2))
          nearest = max.col(-D)
      }
    }
    keyval(nearest, points)
}
```

여기서 입력 값 `points`는 데이터 행렬 ($n \times p$, n은 표본 크기, p는 차원)로 각 행이 자료점 하나에 해당한다. (키는 중요하지 않으므로 무시된다.) 이 `points` 행렬은 HDFS 내에 분산 저장되어 행을 기준으

로 분할되어 맵 함수가 실행된다. 함수 내에서 centers는 군집 중심을 나타내는 $K \times p$ 행렬로 각 행이 군집 중심 하나에 해당한다. 군집 중심이 주어지지 않을 경우 (첫 번째 반복시) 자료점들을 K개의 군집 중 하나로 임의로 분배된다. 그 외에는 각 자료점에서 K개의 군집 중심까지의 유클리드 거리 제곱을 계산한 후 가장 가까운 중심을 찾는다. 출력은 각 점이 속하는 1부터 K까지 중의 군집 번호를 키로 하고 그 점의 좌표를 값으로 하는 <키, 값> 쌍이 된다 (Figure 4.2a). 리듀스 함수에는 군집 번호가 키, 해당 군집 번호로 정렬된 자료점들의 모임이 값인 <키, 값> 쌍이 입력으로 들어온다. 리듀스 함수는 단순히 이들 점들의 평균 좌표를 계산하여 동일한 키와 함께 출력한다 (Figure 4.2b).

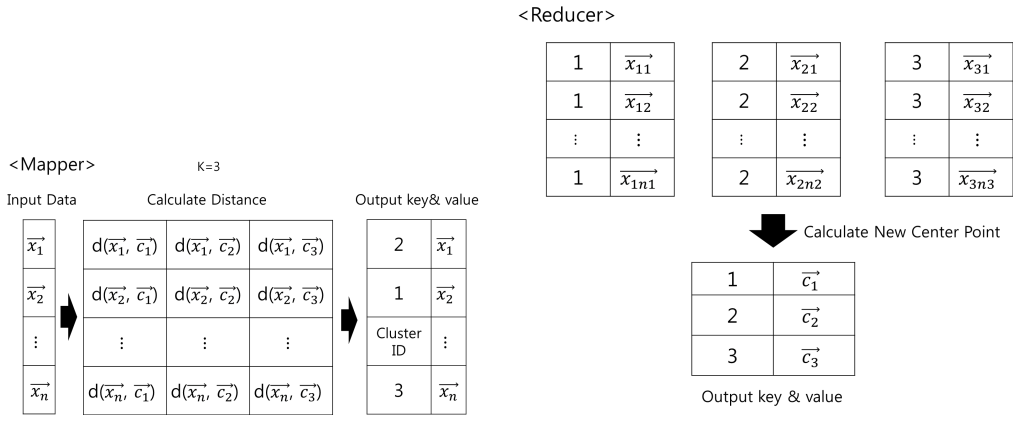
```
kmeans.reduce = function(., points) {
    t(as.matrix(apply(points, 2, mean)))
}
```

이제 전체 K-평균 알고리즘의 맵리듀스 구현은 다음과 같이 기술할 수 있다.

```
kmeans.mr = function( points, K, num.iter ) {
    centers = NULL
    for(i in 1:num.iter ) {
        centers = values( from.dfs( mapreduce( points,
                                    map = kmeans.map,
                                    reduce = kmeans.reduce )
                                )
                        )
    }
    centers
}
```

실행 예는 다음과 같다.

```
kmeans.mr(points, K = 12, num.iter = 5)
```



(a) Mapper function (b) Reducer function

Figure 4.2 MapReduce implementation of the K-means algorithm

맵리듀스 알고리즘의 성능을 평가하기 위해 UCI machine learning repository (Bache와 Lichman, 2013)에서 제공하는 인구 통계 (US Census Data, 1990) 자료를 이용하였다. 이 자료의 개체 수는 약 240만 개, 변수의 수는 68개로 구성되어 있다. 알고리즘 성능을 검증하기 위해 맵리듀스로 구현하지 않은 결과와 수행시간을 비교하였다. 전체 자료에서 각각 10만, 50만, 100만, 200만 개의 표본을 비복원 추출하여 표본 크기에 따른 수행 시간의 변화를 살펴보았다. 또한 알고리즘 반복 회수와 군집 수 K의 변화에 따라 어떤 차이가 나타나는지 확인하였다. 그 결과가 Table 4.2에 요약되어 있다. 반복 2회, 군집 수 500개, 표본 크기 10만 개인 경우, R과 RHadoop 구현은 수행 시간이 각각 4.880초와 92.270초로 약 19배의 차이를 보인다. 그러나 반복 2회, 군집 수 2000개, 표본 크기 200만 개인 경우, R과 Rhadoop의 수행시간은 각각 277.271초와 588.895초로 약 2.1배로 그 차이가 급격히 좁혀짐을 확인할 수 있다. 이는 앞의 두 경우와 동일한 양상이며, 보다 큰 자료에 대해, 복수 개의 서버를 이용하여 완전 분산 모드로 수행한다면 이 격차 역시 역전될 것으로 예상된다.

Table 4.2 Execution time of the K-means example (unit: second)

		R			RHadoop		
		$K = 100$	$K = 200$	$K = 300$	$K = 100$	$K = 200$	$K = 300$
2 iterations	$n = 500,000$	10.016	11.353	13.364	90.481	103.118	156.867
	$n = 1,000,000$	18.991	23.140	27.006	114.521	125.367	248.715
	$n = 2,000,000$	40.064	48.088	55.433	155.944	198.352	326.345
4 iterations	$n = 500,000$	10.785	14.249	17.353	185.436	206.597	365.284
	$n = 1,000,000$	22.760	29.456	35.479	246.201	264.886	527.910
	$n = 2,000,000$	46.105	59.542	73.095	322.543	408.278	896.305

5. 결론

본고에서는 빅데이터 처리 및 분석을 위한 플랫폼인 하둡의 역사와 개념을 간략히 살펴보고 하둡 맵리듀스와 RHadoop을 이용한 간단한 통계 계산 알고리즘들의 구현을 소개하였다. 빅데이터는 그 정의상 기존의 도구로는 처리와 분석이 쉽지 않으므로, 효율적인 빅데이터 처리를 위해서는 새로운 하드웨어/소프트웨어 구조가 도입될 필요가 있으며 새 구조 하에서의 데이터 분석 또한 새로운 계산 모형을 필요로 한다. 이러한 통찰과 그 현실화를 위한 노력들이 하둡과 그 일부인 맵리듀스를 통해 결실을 맺고 있다. 최근 국내에서의 하둡 또는 맵리듀스에 대한 관심은 주로 시스템의 구축을 통해 날로 증가하는 데이터를 담아둘 수 있는 솔루션을 확보하는 데에 머무르고 있는 듯 하나, 해외의 사례로 보아 조만간 확보한 데이터로 무엇을 할 것인가에 대한 문제가 화두가 될 것으로 예상된다. 이 부분은 데이터 분석가 또는 통계학 연구자들의 영역에 해당하며, 당사자들의 많은 관심이 요구된다. 특히 본고에서 보인 바와 같이 대중적인 데이터 분석 도구인 R이 하둡과 손쉽게 결합될 수 있으므로, 앞으로 R과 하둡을 이용한 고급 데이터 분석에 대한 필요성이 증대될 것이다. 연구자의 입장에서 보면 전대미문의 거대한 데이터를 다룰 수 있는 미개척의 영역이 눈앞에 펼쳐져 있다고 할 수 있다. 하둡 및 맵리듀스가 제공하는 새로운 계산 모형을 연구자들이 보다 적극적으로 받아들여 새로운 빅데이터 분석 방법론들이 많이 개발되기를 기대해 본다.

References

Bache, K. and Lichman, M. (2013). UCI machine learning repository. `http://archive.ics.uci.edu/ml`. [Online; accessed June 2013].

Cho, S., Lee, S., Lee, K. and Kim, Y. (2009). Distributed filtering service model for spam mails based on hadoop framework. In *Proceedings of the 2009 Korean Society for Internet Information*, Korean Society for Internet Information, Seoul, 165–168.

Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *OSDI4: Proceedings of the 6th Symposium on Operating Systems Design and Implementation*. USENIX Association, San Francisco.

Facebook Engineering Team (2012). Under the hood: scheduling MapReduce jobs more efficiently with Corona. `https://www.facebook.com/notes/facebook-engineering/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-with-corona/10151142560538920`. [Online; accessed June 2013].

Ghemawat, S., Gobioff, H. and Leung, S.-T. (2003). The google file system. *ACM SIGOPS Operating Systems Review*, **37**, 29–43.

Guha, S. (2010). *Computing environment for the statistical analysis of large and complex data*, PhD thesis, Department of Statistics, Purdue University, West Lafayette.

Guha, S., Hafen, R. P., Kidwell, P. and Cleveland, W. S. (2009). Visualization databases for the analysis of large complex datasets. *Journal of Machine Learning Research*, **5**, 193–200.

Harris, D. (2011). Why the pace of Hadoop innovation has to pick up. `http://gigaom.com/2011/04/25/why-we-need-more-hadoop-innovation/`. [Online; accessed June 2013].

Jung, H., Kim, J., Park, H. and Lee, J. (2011). The design of content-based music search system using hadoop. In *Proceedings of the 2011 Korean Institute of Information Scientists and Engineers*, The Korean Institute of Information Scientists and Engineers, Seoul, 377–380.

Kim, M., Cui, Y., Han, S. and Lee, H. (2012). A hadoop-based media transcoding system for mobile media service. In *Proceedings of the 2012 Korean Society for Internet Information*, Korean Society for Internet Information, Seoul, 233–234.

Lam, C. (2012). *Hadoop in action* (Korean translation), Ji & Son, Seoul.

McKinsey Global Institute (2011). *Big data: The next frontier for innovation, competition, and productivity*, McKinsey Global Institute, New York.

Park, S., Lee, B., Kim, H., Kim, D. and Yoon, S. (2011). A study on speedup of multiple sequence alignment using mapreduce on cloud infrastructure. In *Proceedings of the 2011 Korean Institute of Information Scientists and Engineers*, The Korean Institute of Information Scientists and Engineers, Seoul, 123-126.

Piccolboni, A. (2013). Mapreduce in R. `https://github.com/RevolutionAnalytics/rmr2/blob/master/docs/tutorial.md`. [Online; accessed June 2013].

Revolution Analytics (2011). Advanced big dataanalytics with R and Hadoop. `http://www.revolutionanalytics.com/why-revolution-r/whitepapers/advanced-big-data-analytics-with-r-and-hadoop.php`. [Online; accessed June 2013].

Seo, S., Kim, J., Park, Y., Lee, J. and Myeong, J. (2013). *Hadoop & NoSQL*, Gilbut, Seoul.

The Apache Software Foundation (2008). MapReduce tutorial. `http://hadoop.apache.org/docs/stable/mapred_tutorial.html`. [Online; accessed June 2013].

Journal of the Korean Data & Information Science Society
2013, **24**(5), 1013–1027

http://dx.doi.org/10.7465/jkdi.2013.24.5.1013
한국데이터정보과학회지

Hadoop and MapReduce[†]

Jeong-Hyeok Park[1] · Sang-Yeol Lee[2] · Da Hyun Kang[3] · Joong-Ho Won[4]

[1234]School of Industrial Management Engineering, Korea University

Received 7 July 2013, revised 7 August 2013, accepted 12 August 2013

Abstract

As the need for large-scale data analysis is rapidly increasing, Hadoop, or the platform that realizes large-scale data processing, and MapReduce, or the internal computational model of Hadoop, are receiving great attention. This paper reviews the basic concepts of Hadoop and MapReduce necessary for data analysts who are familiar with statistical programming, through examples that combine the R programming language and Hadoop.

Keywords: Big data analytics, distributed computing, Hadoop, MapReduce.

[†] This research has been supported by a Korea University Grant.

[1] Graduate student, School of Industrial Management Engineering, Korea University, Seoul 136-713, Korea.
[2] Graduate student, School of Industrial Management Engineering, Korea University, Seoul 136-713, Korea.
[3] Graduate student, School of Industrial Management Engineering, Korea University, Seoul 136-713, Korea.
[4] Corresponding author: Assistant professor, School of Industrial Management Engineering, Korea University, Seoul 136-713, Korea. E-mail: wonj@korea.ac.kr