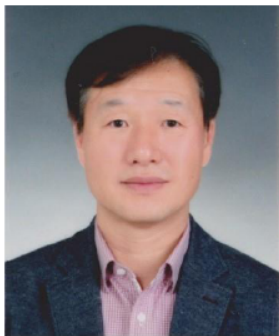


# Apache Spark

김 승 환

[swkim4610@inha.ac.kr](mailto:swkim4610@inha.ac.kr)

# 강사 소개



## 김 승 환 교수

- 인하대학교 통계학 박사(1997)
- 현, 인하대학교 공과대학  
소프트웨어융합공학 연계전공  
연구교수
- 현, 정보화진흥원 개인정보 비식별화  
전문위원

## [ 주요 경력 소개 ]

### [경력]

- 1998.6 ~ 2000.4 나이스컨설팅 수석연구원
- 2000.5 ~ 2005.6 SK 에너지 CRM팀 부장
  - 엔크린 보너스 카드 고객분석 및 CRM 수행
- 2005.7 ~ 2007.12 SK 네트워크 고객사업부문 팀장
- 2008.1 ~ 2014.3 SK 마케팅애펀퍼니 K-Hub 그룹장
  - 오케이캐쉬백 고객정보 관리 및 마케팅 총괄
  - 오케이캐쉬백 고객정보보호 책임자

### [전문 분야]

- Statistical Learning, Deep Learning
- 개인정보 비식별화
- 빅데이터 시스템 개발

# 목차 및 학습 필요사항

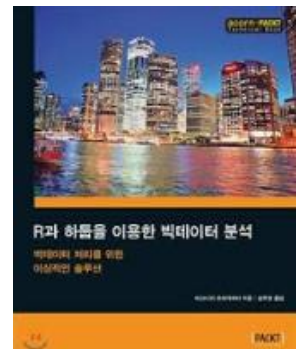
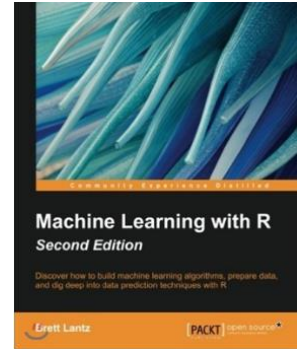
1. 빅데이터와 스몰데이터
2. Hadoop
3. R
4. RHadoop
5. 빅데이터 통계모형
6. 실전 분석

## 참고서적

본 강의에는 우측과 같은 책이  
참조 되었습니다.

소스 및 데이터

<http://naver.me/xJn0Djcc>



# 1. 빅데이터와 스몰데이터

1.1 빅데이터 시대

1.2 분산처리

1.3 빅데이터 속성

1.4 정형과 비정형

1.5 데이터웨어하우스

1.6 Static과 Event Log Data

1.7 데이터 처리 변화

1.8 통계학

1.9 통계학과 빅데이터 접근 차이

# 1. 1 빅데이터 시대

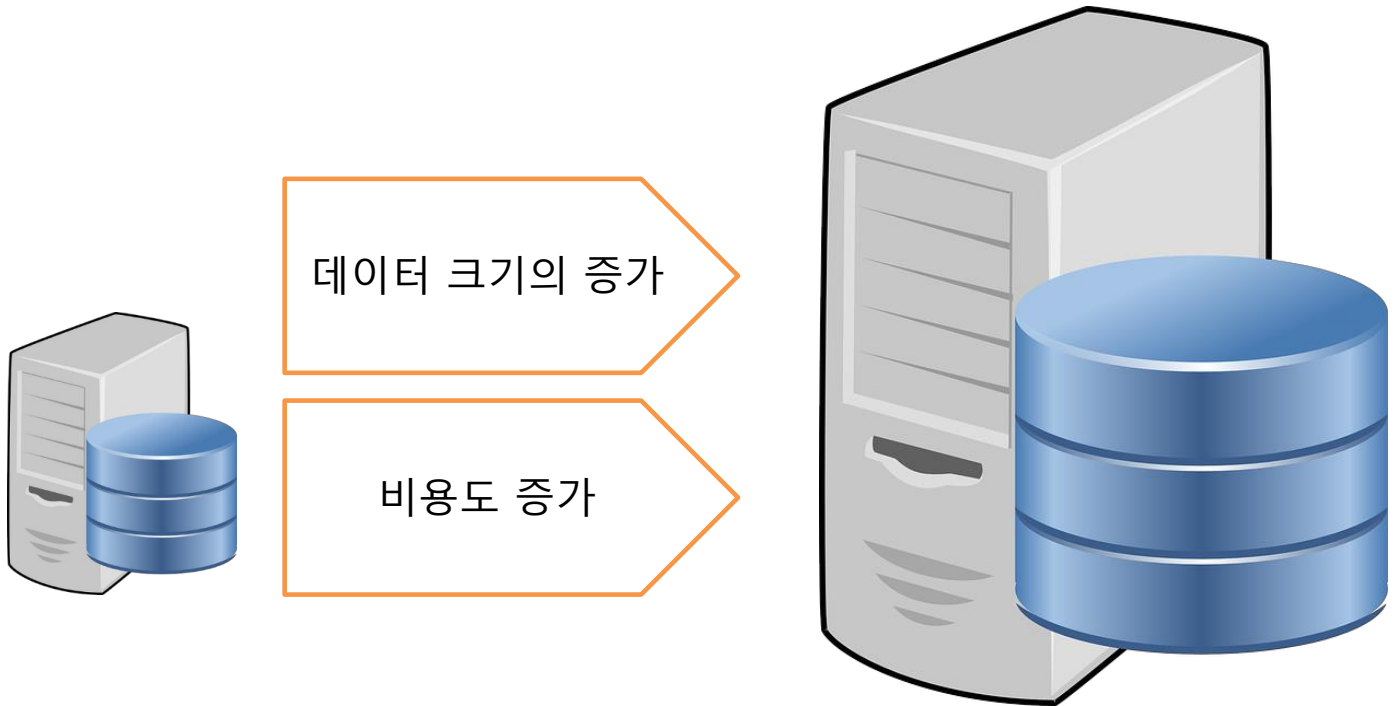
우리는 이미 상당량의 디지털 데이터를 발생시키는 주체이고 이용하는 주체이다.  
이른바, 빅데이터 세상에 살고 있다.



- 뉴욕증권거래소 : 1일에 1테라 바이트의 거래 데이터생성
- Facebook : 100억장의 사진, 수 페타바이트의 스토리지
- 통신사 : 시간당 10G 이상의 통화 데이터, 1일240G 생성, 월 생성 데이터의 크기 200T 이상

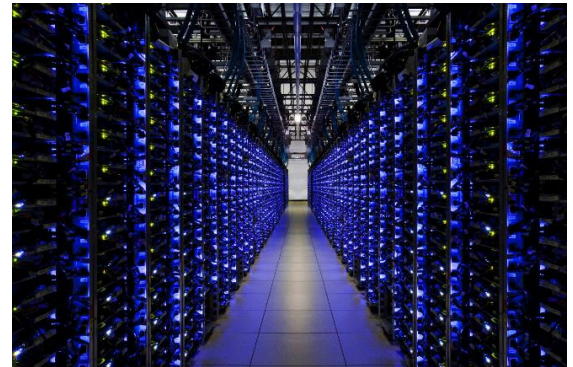
# 1. 1 빅데이터 시대

- ✓ 과거에는 발생하는 데이터 중 상당 부분을 버리거나 요약된 형태로 저장 관리했음
- ✓ 검색서비스, 유튜브 등이 나오면서 데이터가 급격히 증가



## 1.2 분산처리

- ✓ 이에 대한 해결책으로 ... → 분산 처리 Computing 방법론 탄생
- ✓ 거대한 디스크를 가진 컴퓨터가 아닌 PC 계열 리눅스 OS 탑재한 컴퓨터를 병렬로 연결
- ✓ 컴퓨터 군단을 병렬로 제어하는 구글 파일 시스템 탄생
- ✓ 빅데이터를 저장하는 비용이 획기적으로 감소하여 빅데이터를 저장하게 되었음
- ✓ 하지만, 이렇게 저장된 빅데이터를 활용하는 분야는 검색, 집계 등 단순 분야였음



## 1.3 빅데이터 속성

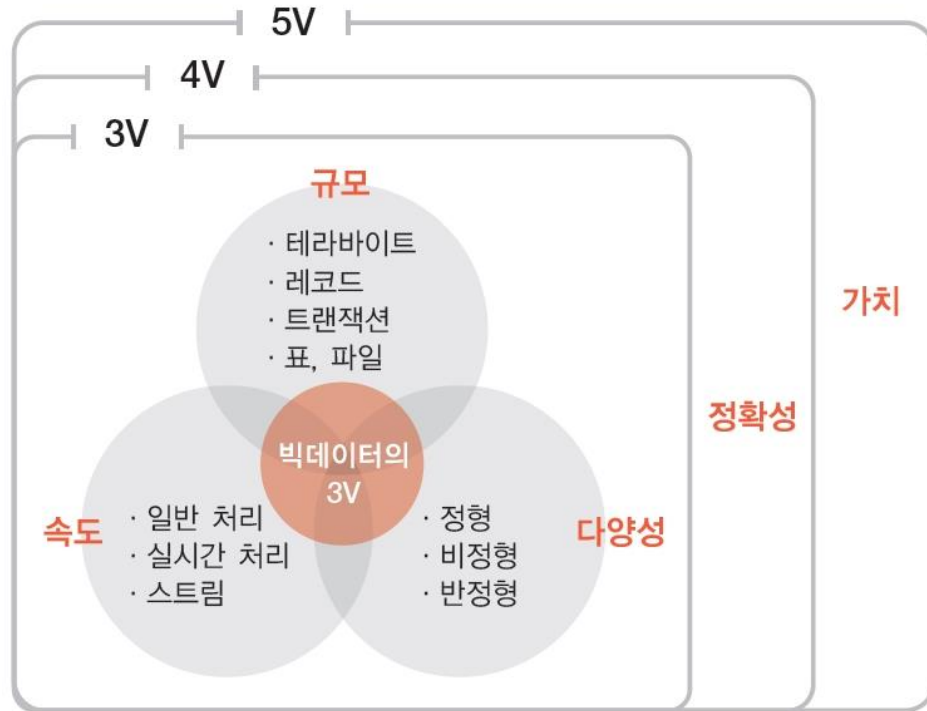


그림 1-2 빅데이터의 속성 [02]

<https://ikkison.tistory.com/66> 참조





# 1.4 정형과 비정형

ID	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Company	Last Name	First Name	E-mail Address	Job Title	Business Home Phone	Mobile Phone	Fax	Number	Address	City	State/Province	ZIP/Postal Code	Country
1	Company A	Batista	Anna	Owner		(123555) 010				(123555) 0121 1st Street	Seattle	WA	98099	USA
2	Company B	Gratcos	Antonio	Owner		(123555) 010				(123555) 0123 2nd Street	Boston	MA	98099	USA
3	Company C	Aven	Thomas	Purchaser		(123555) 010				(123555) 0123 3rd Street	Los Angeles	CA	98099	USA
4	Company D	Lee	Christina	Purchaser		(123555) 010				(123555) 0123 4th Street	New York	NY	98099	USA
5	Company E	O'Donnell	Martin	Owner		(123555) 010				(123555) 0123 5th Street	Minneapolis	MN	98099	USA
6	Company F	Pinto	Clara	Purchaser		(123555) 010				(123555) 0123 6th Street	Minneapolis	MN	98099	USA
7	Company G	Xie	Ming-Yang	Owner		(123555) 010				(123555) 0123 7th Street	Boise	ID	98099	USA
8	Company H	Andersen	Elizabeth	Purchaser		(123555) 010				(123555) 0123 8th Street	Portland	OR	98099	USA
9	Company I	Moffesser	Sue	Purchaser		(123555) 010				(123555) 0123 9th Street	Salt Lake City	UT	98099	USA
10	Company J	Wacker	Paul	Purchaser		(123555) 010				(123555) 0123 10th Street	Chicago	IL	98099	USA
11	Company K	Fischer	Paul	Purchaser		(123555) 010				(123555) 0123 11th Street	Miami	FL	98099	USA
12	Company L	Edwards	John	Purchaser		(123555) 010				(123555) 0123 12th Street	Las Vegas	NV	98099	USA
13	Company M	Ludick	Andre	Purchaser		(123555) 010				(123555) 045 13th Street	Memphis	TN	98099	USA
14	Company N	Gato	Carlos	Purchaser		(123555) 010				(123555) 045 14th Street	Denver	CO	98099	USA
15	Company O	Kupkova	Helena	Purchaser		(123555) 010				(123555) 045 15th Street	Honolulu	HI	98099	USA
16	Company P	Guldichen	Daniel	Purchaser		(123555) 010				(123555) 045 16th Street	San Francisco	CA	98099	USA
17	Company Q	Bagel	Jean-Philippe	Owner		(123555) 010				(123555) 045 17th Street	Seattle	WA	98099	USA
18	Company R	Aulter	Michael	Purchaser		(123555) 010				(123555) 045 18th Street	Boston	MA	98099	USA
19	Company S	Eggner	Alexander	Accountant		(123555) 010				(123555) 078 19th Street	Los Angeles	CA	98099	USA
20	Company T	Li	George	Purchaser		(123555) 010				(123555) 078 20th Street	New York	NY	98099	USA
21	Company U	Tham	Bernard	Accountant		(123555) 010				(123555) 078 21st Street	Minneapolis	MN	98099	USA
22	Company V	Ramos	Luciana	Purchaser		(123555) 010				(123555) 078 22nd Street	Minneapolis	MN	98099	USA
23	Company W	Entin	Michael	Purchaser		(123555) 010				(123555) 078 23rd Street	Portland	OR	98099	USA
24	Company X	Hessalbin	Jones	Purchaser		(123555) 010				(123555) 078 24th Street	Salt Lake City	UT	98099	USA
25	Company Y	Rodman	John	Purchaser		(123555) 010				(123555) 078 25th Street	Chicago	IL	98099	USA
26	Company Z	Liu	Pui	Accountant		(123555) 010				(123555) 078 26th Street	Miami	FL	98099	USA
27	Company AA	Toh	Karen	Purchaser		(123555) 010				(123555) 078 27th Street	Las Vegas	NV	98099	USA
28	Company BB	Raghu	Amitesh	Purchaser		(123555) 010				(123555) 078 28th Street	Memphis	TN	98099	USA
29	Company CC	Lee	Soo Jung	Purchaser		(123555) 010				(123555) 078 29th Street	Denver	CO	98099	USA

정형 Data

```

1 #Software: Microsoft Internet Information Services X.X-
2 #Version: X-
3 #Date: 2010-03-24 07:00:01-
4 #Fields: date time s-sitename s-computername s-ip cs-method cs-uri-stem cs-uri-query s-port cs-
5 2010-03-24 07:00:01 ZZZZC941948879 RUFFLES 222.222.222.222 GET / - 80 - 220.181.7.113 HTTP/1.1
6 2010-03-24 07:00:23 ZZZZC941948879 RUFFLES 222.222.222.222 GET /2009/12/im_not_mean_im_just_ar
7 2010-03-24 07:00:32 ZZZZC941948879 RUFFLES 222.222.222.222 GET /terminal-blank.gif - 80 - 217.
8 2010-03-24 07:00:32 ZZZZC941948879 RUFFLES 222.222.222.222 GET /grep-options.gif - 80 - 217.2
9 2010-03-24 07:00:32 ZZZZC941948879 RUFFLES 222.222.222.222 GET /terminal-cat.gif - 80 - 217.2
10 2010-03-24 07:00:32 ZZZZC941948879 RUFFLES 222.222.222.222 GET /terminal-pwd-cd.gif - 80 - 217
11 2010-03-24 07:00:39 ZZZZC941948879 RUFFLES 222.222.222.222 GET /robots.txt - 80 - 95.55.207.95
12 2010-03-24 07:00:39 ZZZZC941948879 RUFFLES 222.222.222.222 GET /rss-short.xml - 80 - 173.45.2
13 2010-03-24 07:00:43 ZZZZC941948879 RUFFLES 222.222.222.222 GET /2009/08/22-things-you-dont-kno
14 2010-03-24 07:00:44 ZZZZC941948879 RUFFLES 222.222.222.222 GET /screen.css - 80 - 98.88.35.13
15 2010-03-24 07:00:44 ZZZZC941948879 RUFFLES 222.222.222.222 GET /img/rss-header-red.gif - 80 -
16 2010-03-24 07:00:44 ZZZZC941948879 RUFFLES 222.222.222.222 GET /img/logo.jpg - 80 - 98.88.35.1
17 2010-03-24 07:00:44 ZZZZC941948879 RUFFLES 222.222.222.222 GET /img/input-emailsend.jpg - 80 -
18 2010-03-24 07:00:45 ZZZZC941948879 RUFFLES 222.222.222.222 GET /images/cm-ebook-banner.gif - 80 -
19 2010-03-24 07:00:45 ZZZZC941948879 RUFFLES 222.222.222.222 GET /img/bg.jpg - 80 - 98.88.35.13
20 2010-03-24 07:00:45 ZZZZC941948879 RUFFLES 222.222.222.222 GET /img/bg-top.jpg - 80 - 98.88.35
21 2010-03-24 07:00:45 ZZZZC941948879 RUFFLES 222.222.222.222 GET /21things/checkout-login.gif -
22 2010-03-24 07:00:45 ZZZZC941948879 RUFFLES 222.222.222.222 GET /img/topnav-contact.jpg - 80 -
23 2010-03-24 07:00:45 ZZZZC941948879 RUFFLES 222.222.222.222 GET /21things/portent-email-sub.gif
24 2010-03-24 07:00:45 ZZZZC941948879 RUFFLES 222.222.222.222 GET /rss-header.jpg - 80 - 98.88.35

```

Log Data

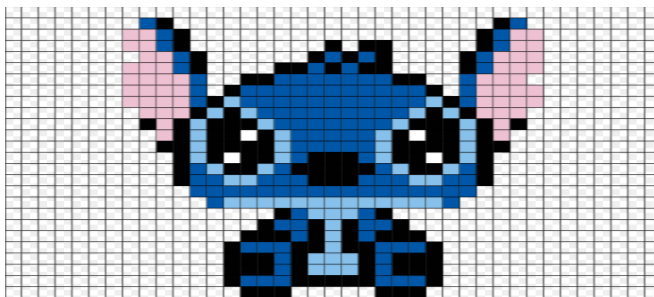
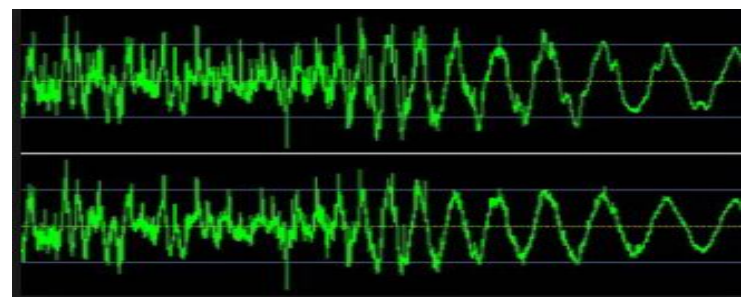


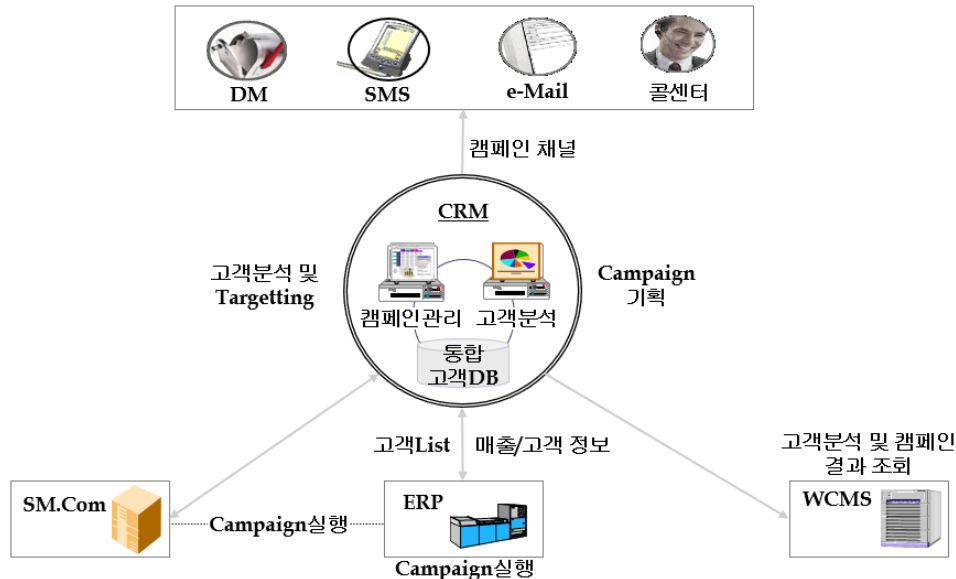
Image Data



Sound Data

## 1.5 데이터웨어하우스

- 아래는 2000년 초반의 데이터웨어하우스 구성도로 정형 데이터베이스로 구성되어 있음
- 현재는 위의 구성에 추가로 이메일, 콜센터 녹취 자동 Text 변환, .com 로그 정보 등이 추가되어 하둡 시스템으로 관리되고 있음
- 과거에는 콜센터 녹취나 .com 로그 정보는 관리하지 않았음
- 오프라인에서 온라인으로 고객과 접촉 채널이 바뀌면서 온라인 로그 정보가 중요해짐



## 1.6 Static Data와 Event Log Data

- Static Data / Event Log Data

Static Data는 개체의 속성에 해당하는 데이터로 시간에 따라 바뀌지 않는 데이터를 말함

예: 성별, 연령, 지역, 제조사, 생산일 등

Event Log Data는 개체의 상태에 해당하는 데이터로 시간에 따라 바뀌는 데이터를 말함

RDB 형태로 저장하는 것이 유리함

예: 현 위치, 조회 키워드, 클릭 페이지 등

일반적으로 행은 관측단위, 열은 변수로 지정되는 정형 데이터의 구조에서는

Event Log 데이터를 처리하기 어려움

하지만, 현재 Event Log를 이용한 데이터 처리 수요가 증가하고 있음

Event Log는 하둡 시스템을 저장하는 것이 유리함

예: 구글, 페이스북 보유 텍스트, 센서 데이터 등

## 1.6 Static Data와 Event Log Data

- 쿠팡이 보유한 정보

거래정보: 누가, 언제, 무엇을 구매했는지에 관한 정보

상품정보: 바코드, 상품분류, 상품명, 제조사, Seller 정보, 가격 등

고객정보: 배송, 과금을 위한 정보

Web, App 사용정보: 접속일자, 링크 클릭 정보, 검색 정보 등

각 정보를 RDB, Hadoop 중 어느 곳에 저장하는 것이 효율적일까요?



<https://youtu.be/IU9OLSVyluw> 참조

## 1.7 데이터 처리 변화

데이터수집

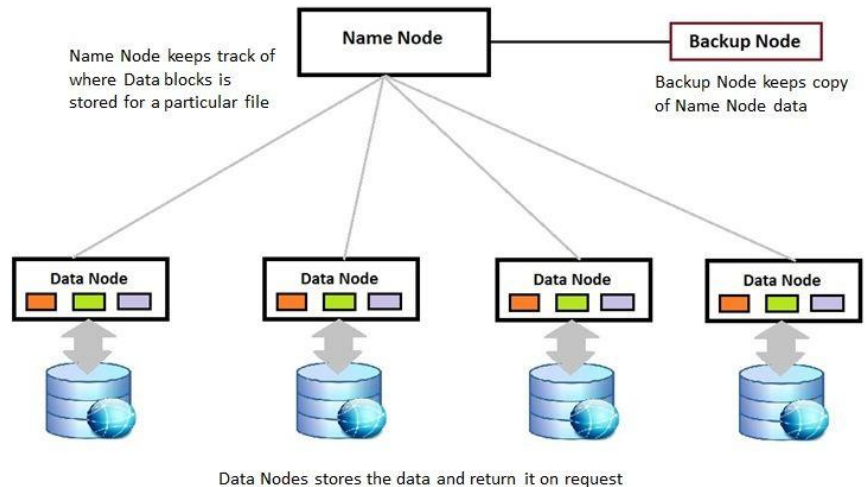
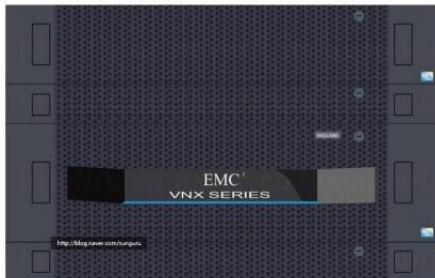
저장

집계

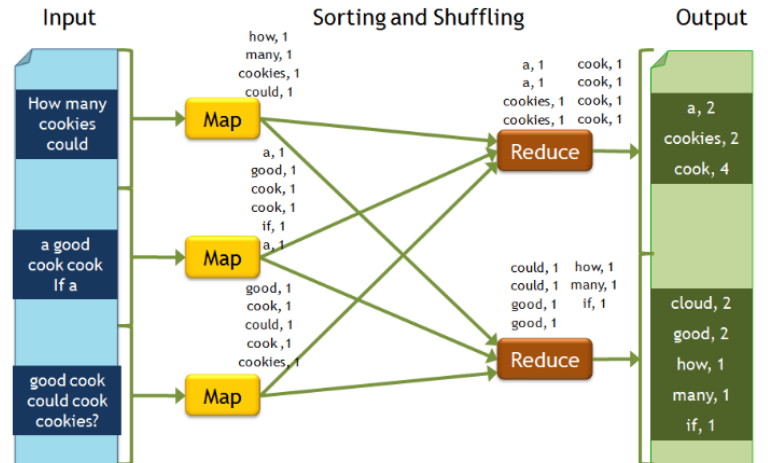
지능화



# 1.7 데이터 처리 변화



# 1.7 데이터 처리 변화



By Manaranjan Pradhan



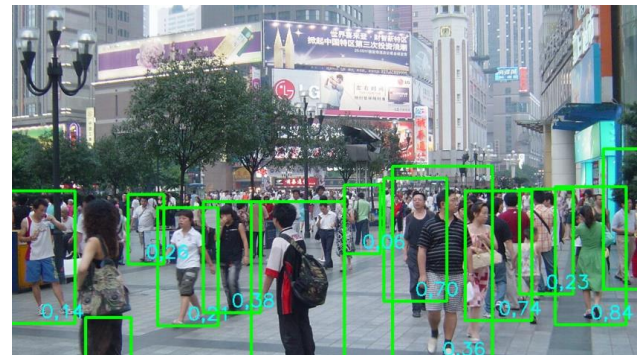
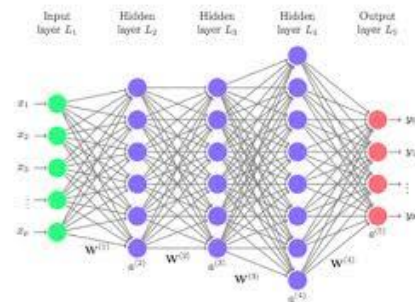
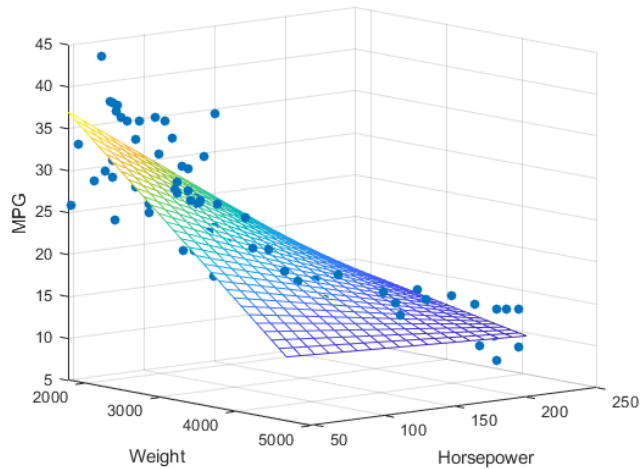
# 1.7 데이터 처리 변화

데이터수집

저장

집계

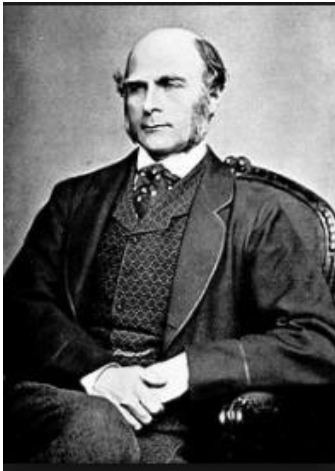
지능화





## 1.8 통계학

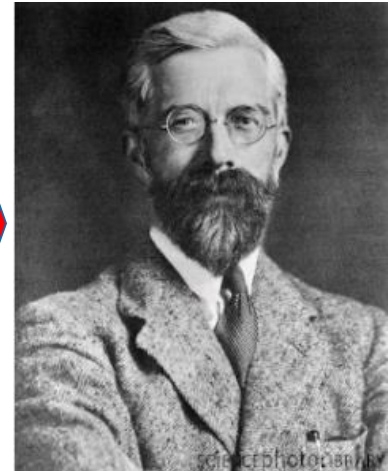
통계학: 표본으로 부터 모집단의 정보를 추정하거나 모집단의 상태를 추측하는 과학



상관/회귀 기본개념  
제시



상관/회귀분석 완성  
중심극한정리



표본이론 정립  
현대 추측 통계학 정립

**빅데이터가 왜 필요한가?**

## 1.8 통계학

간단한 문제: 인구 중에 키 185cm 이상인 사람의 비율은?

솔루션 1: 1,000명의 키를 검사하여 185cm 이상인 사람의 비율로 추정

솔루션 2: 185cm 이상인 사람이 나올 때까지 검사하여 비율은  $1/n$  으로 추정

솔루션 3: 30명의 키를 측정하여 평균과 표준편차를 구해 정규분포 가정 하에 계산함

가장 정확한  
방법은?

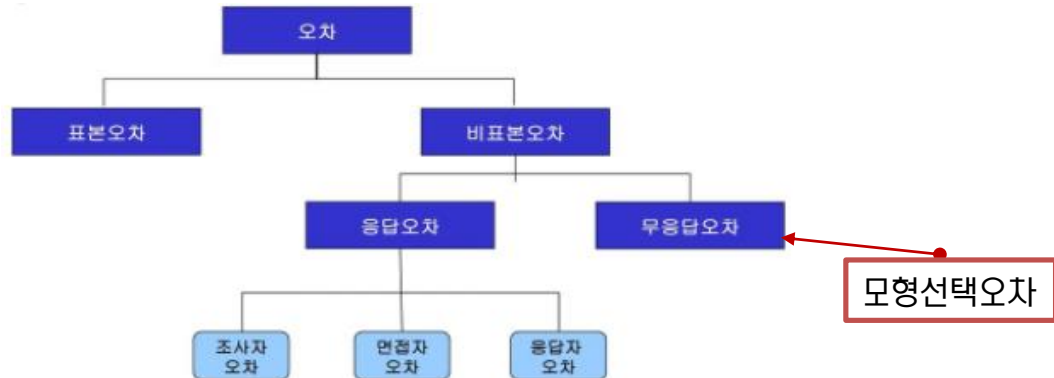


가장 효율적인  
방법은?

## 1.8 통계학

기본적으로 표본 수를 최소화하고자 함(표본 수 증가는 비용의 증가)

표본오차는 표본 수가 증가하면 줄어들지만, 비 표본 오차는 과학적으로 접근이 어려워 계산이 불가능함  
비표본 오차에 의해 참값과 통계적 추정값이 다른 결과를 줄 가능성이 존재함



Small Data의 경우, 분석 정확도를 위해 모든 정보를 사용해 추정 혹은 의사결정 수행함  
분포 가정을 통해 재현성을 검정하고자 함

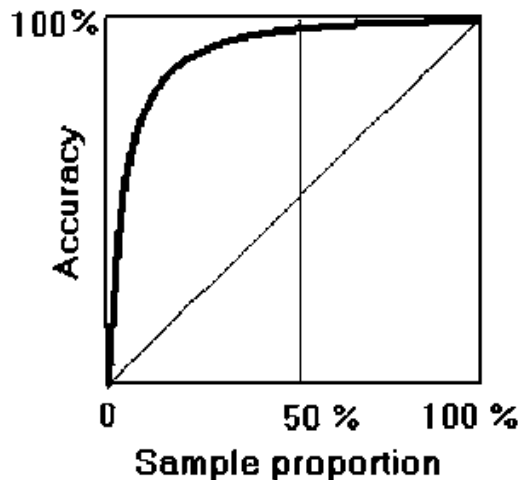
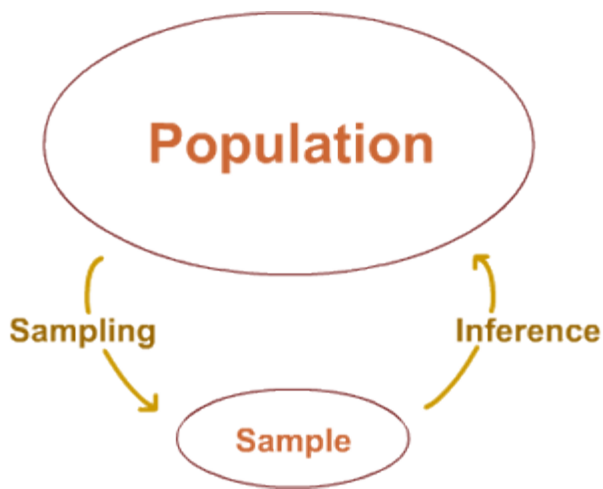
재현성: 다른 표본에서도 우리의 추정이나 의사결정이 재현됨을 증명하는 것

Bid Data의 경우는 풍부한 데이터 셋을 가지고 있기 때문에 재현성을 분포를 통해 증명하는 것이 아니고 실증적으로 재현이 가능함

## 1.8 통계학

기존 통계적 방법론은 빅데이터가 존재해도 추정 혹은 의사결정의 정확도가 크게 증가하지 않는다.

왜냐하면, 표본오차는 거의 "0"에 가깝게 줄었지만 비표본 오차는 그대로 존재하기 때문임



## 1.9 통계학과 빅데이터 접근 차이

2000년대 이후, 분석에 대한 요구사항은 다양해지고 있음

**과거: 자신의 연구가설을 증명(주로 학문적 요구)**

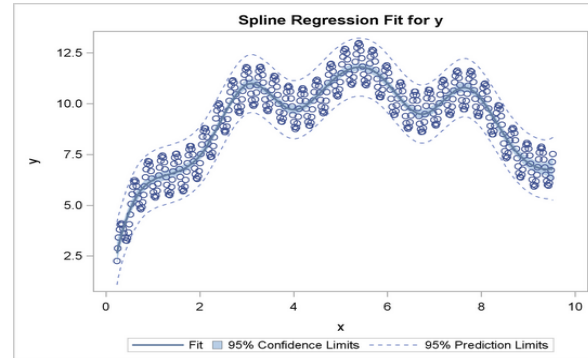
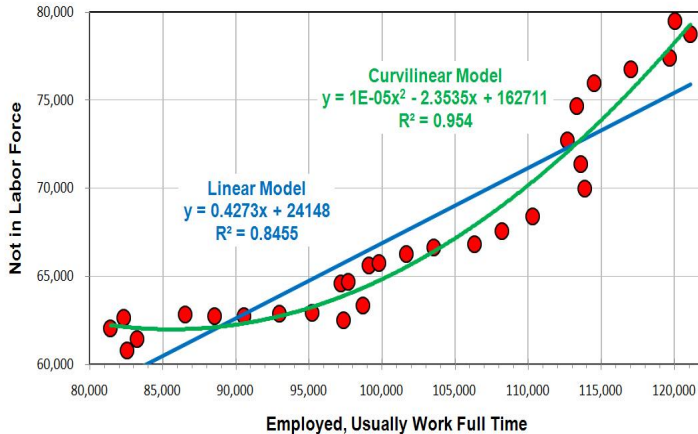
**현재: Prediction/Forecasting, Decision Making 등 다양한 Biz. 요구 등장**

	데이터	관점	모형	모수 절약
전통적 통계학	분석을 위해 필요 최소 데이터를 수집함	모집단의 구조파악을 통한 추론 (보수적 관점)	Linear 중심	설명변수의 수를 최소화하여 분석의 자유도를 확보 하고자 함
빅데이터 접근	기존에 축적된 자료를 통해 분석함	모집단의 구조보다 예측에 초점 (적극적 관점)	Non- Linear 로 확장	데이터가 많으므로 변수의 수에 구애 받지 않음

Q: Linear 모형을 사용하는 이유?

## 1.9 통계학과 빅데이터 접근 차이

### Linear Regression and Non-Linear Regression Model



스몰 데이터에서 고차항의 모형을 사용하면 할 수는 있으나 Over-fitting 위험이 커진다.

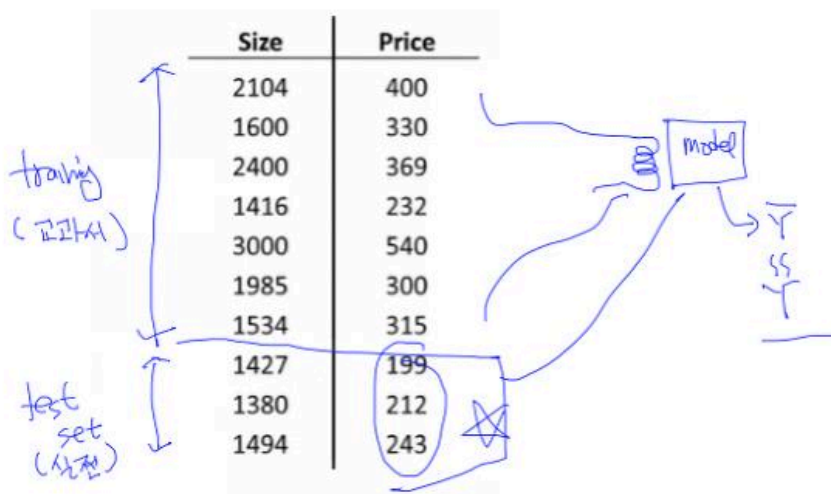
빅데이터에서는 복잡한 모형을 사용해도 Over-fitting 위험이 덜하다.

## 1.9 통계학과 빅데이터 접근 차이

빅 데이터의 경우, 모델을 만드는데 모든 데이터를 사용하지 않고 모델 검증을 위한 Test-Set을 남겨 둘 수 있는 여유가 존재한다.

이 방법이 실증적 재현성이다.

### Training and test sets



Size	Price
2104	400
1600	330
2400	369
1416	232
3000	540
1985	300
1534	315
1427	199
1380	212
1494	243

[http://www.holehouse.org/mlclass/10\\_Advice\\_for\\_applying\\_machine\\_learning.html](http://www.holehouse.org/mlclass/10_Advice_for_applying_machine_learning.html)

## 1.9 통계학과 빅데이터 접근 차이

- ✓ 통계학은 분석 목적에 따라 데이터를 수집하므로 교락 등의 문제를 해결하는 형태로 데이터를 수집하지만, 빅데이터는 분석 목적과 상관없이 축적된 자료이기 때문에 분석 및 해석시 주의가 필요
- ✓ 빅데이터는 데이터가 자동으로 확보 됨에 따라 여러 분야에서 분석을 다양하게 시도할 수 있는 긍정적인 측면 존재
- ✓ 빅데이터에서는 보유자료가 모집단에 근사한 경우가 많아 통계적 접근이 아닌 집계에 가까움  
이 경우, 집계결과를 확대해서 전체로 적용하는 오류 역시 존재함
- ✓ 빅데이터 분야에서도 오랫동안 연구 개발된 통계 이론을 적용하고 있음

Machine Learning <- Machine Learning + Statistics

- ✓ 통계학 이론도 빅데이터 상황에 맞도록 새로운 이론들이 개발되고 있음  
(Bagging, Boosting, Lasso 등)



## 2. 하둡

2.1 하둡의 등장 배경

2.2 분산처리

2.3 하둡 예코 시스템

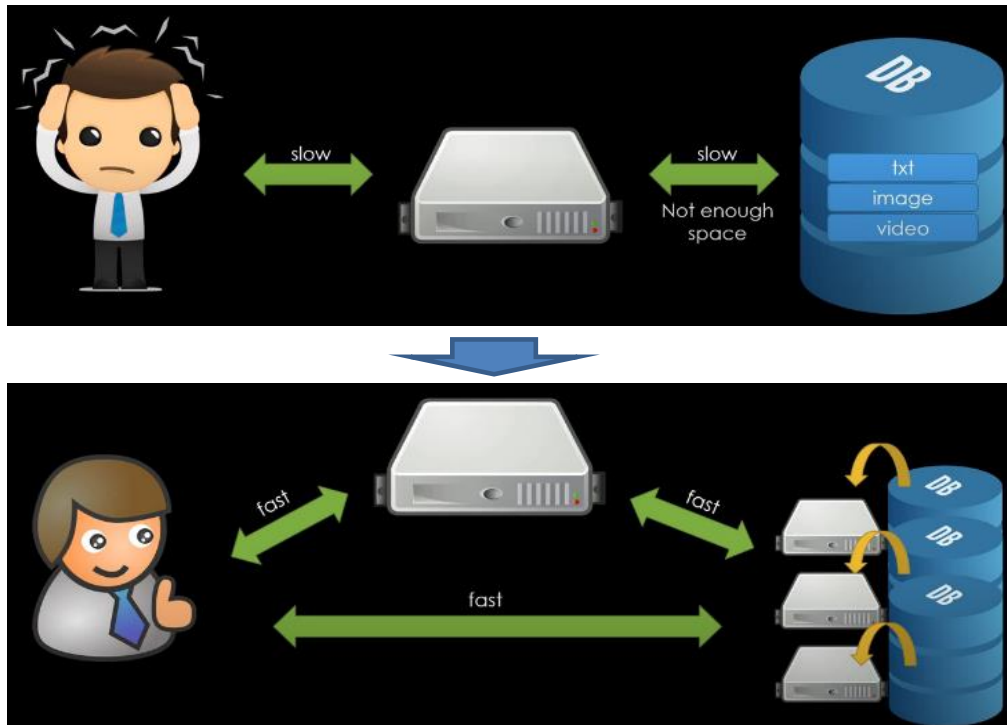
2.4 맵리듀스 프로그래밍

2.5 Word Count 예제

## 2.1 하둡의 등장 배경

- ✓ 구글의 고민은 데이터가 증가하면서 컴퓨터 처리 속도가 현저하게 감소 것임
- ✓ CPU 연산보다 Disk I/O에 시간이 많이 사용됨. 병렬 처리를 수행하는 파일 시스템 개발

<https://youtu.be/IU9OLSVyluw> 참조

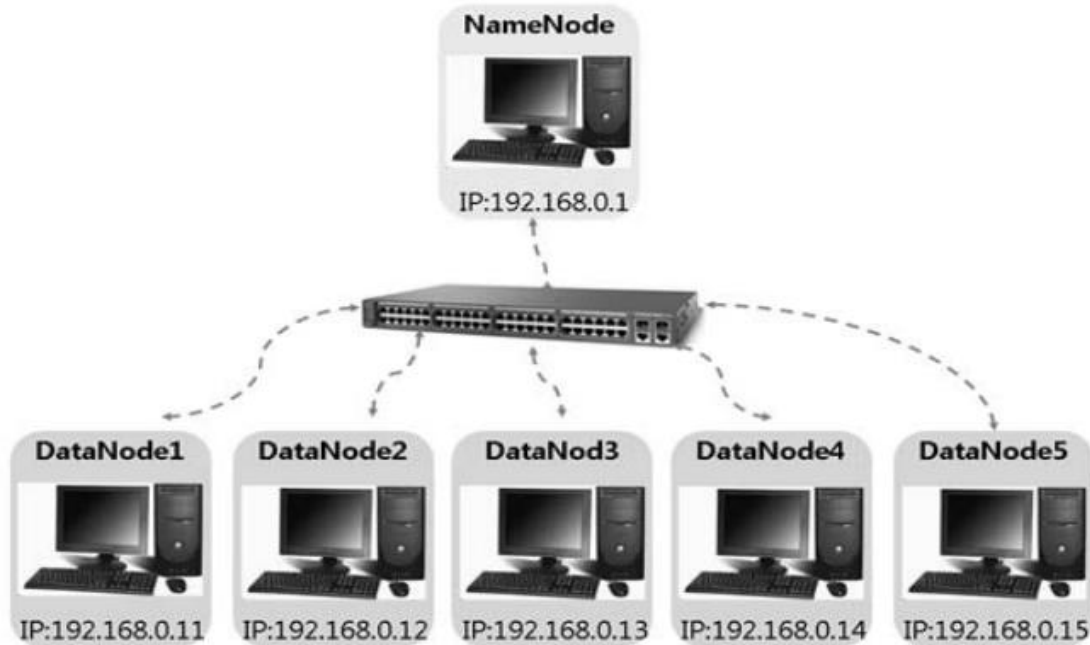


## 2.1 하둡의 등장 배경

- ✓ 하둡은 구글이 개발한 분산처리 오픈소스 프레임워크임
- ✓ 하둡은 HDFS(Hadoop Distributed File System)과 MapReduce로 구성됨
- ✓ HDFS는 파일 시스템으로 네임 노드(name node)와 데이터 노드(data node)로 구성
- ✓ HDFS는 Google File System으로 출발
- ✓ HDFS는 데이터 분산저장과 데이터 노드 모니터링(Heartbeat Monitor)를 통해 데이터 노드의 작동여부를 감시한다. 또한, 블록관리 기능을 통해 하나의 데이터를 여러 블록으로 복제하여 장애 발생에 대응한다.
- ✓ 네임 노드는 데이터 노드의 파일 구조를 정리한 곳이고 데이터 노드가 실제 데이터를 저장
- ✓ 맵리듀스는 Job Tracker와 Task Tracker로 구성되어 있음
- ✓ Job Tracker는 큰 “job”을 작은 Task로 나누어 실행시키고 결과를 병합 받음
- ✓ 작은 Task로 나누어 실행하는 것이 분산처리임

## 2.2 분산처리

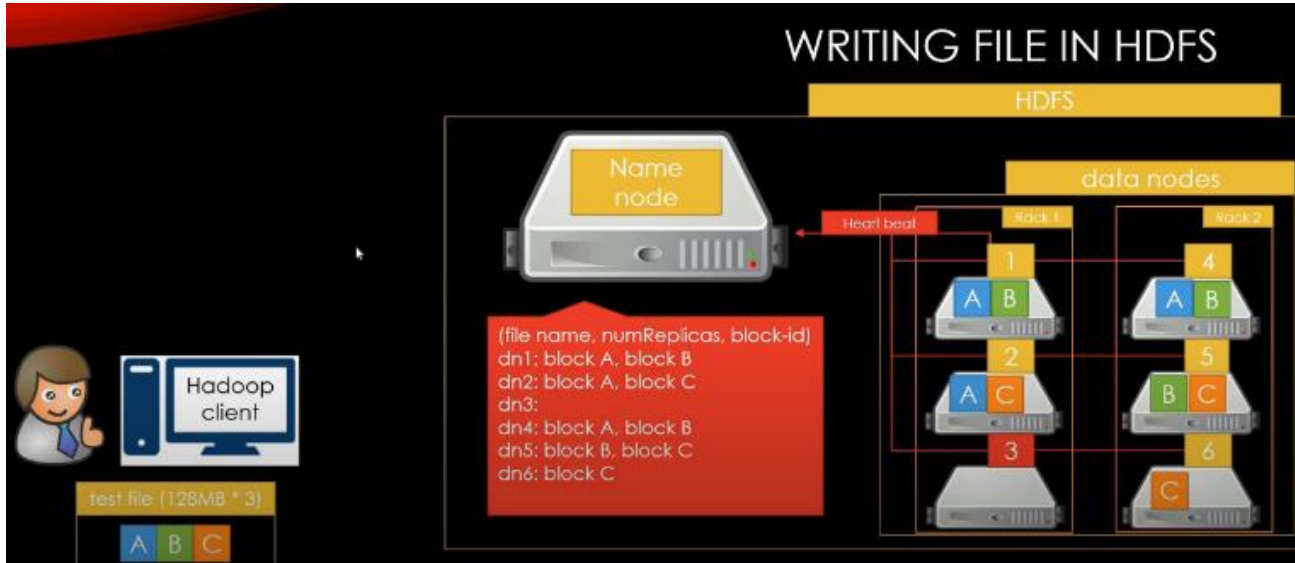
- ✓ 아래는 네임노드와 데이터노드의 연결에 대한 개념도임



참조: 신지은, 오윤식, 임동훈, 빅데이터 K-평균 클러스터링을 위한 RHadoop 플랫폼, 한국데이터정보과학회지, 2016

## 2.2 분산처리

- ✓ 하둡 클라이언트는 데이터를 HDFS에 저장하는 역할을 하고, 네임노드는 데이터 노드의 정상 작동여부와 각 데이터가 어디에 저장되어 있는지의 정보를 가지고 있음
- ✓ 데이터 노드에는 원본 데이터를 여러 조각으로 나누어 분산 보관 및 데이터 손실에 대비한 여분을 보관함
- ✓ 파일을 읽을 때에는 네임노드에서 원하는 정보의 위치를 찾아 데이터노드를 Access함

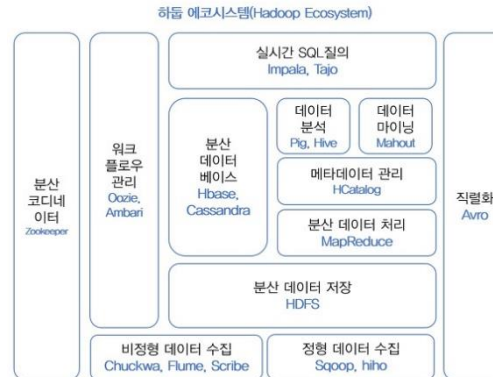


<https://youtu.be/IU9OLSVyluw> 참조

## 2.3 하둡 에코 시스템

- ✓ 대용량 데이터를 여러 대의 컴퓨터가 분산 처리하는 것이 비용이 저렴한 경우 유용함
- ✓ RDB의 경우, 국내 대기업에서 1테라를 유지하는데 1억 정도 소요됨
- ✓ 하둡은 리눅스 OS에 설치하여 사용하고 멀티 분산 모드를 사용하여 작동함
- ✓ 가상분산모드: 한대의 컴퓨터에 여러 대의 컴퓨터를 만드는 가상분산 방식
- ✓ 멀티분산모드: 네임 노드 한대와 여러 데이터 노드를 서로 다른 컴퓨터에 설치하는 방식
- ✓ 하둡 에코시스템은 HDFS와 MapReduce외에 많은 유틸리티와 분석 도구로 이루어져 있음
- ✓ 하둡 에코 시스템 안에는 Pig(돼지), Hive(벌떼), ZooKeeper(사육사) 등

각종 동물 이름의 시스템이 있는데 이 프로젝트의 시초가 된 노란 코끼리 네이밍에 영향을 받음



## 2.4 MapReduce 프로그래밍

- ✓ 하둡은 자바언어로 만들어져 있고, 자바언어로 작동할 수 있음
- ✓ R, Hive 등 많은 분석 라이브러리가 존재하여 자바를 사용하지 않고도 분석 가능

```
1 package hadoop;
2
3 import java.io.IOException;
4 import java.util.StringTokenizer;
5
6 import org.apache.hadoop.io.IntWritable;
7 import org.apache.hadoop.io.LongWritable;
8 import org.apache.hadoop.io.Text;
9 import org.apache.hadoop.mapreduce.Mapper;
10
11 public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
12     private final static IntWritable one = new IntWritable(1);
13     private Text word = new Text();
14
15     @Override
16     protected void map(LongWritable key, Text value,
17         Mapper<LongWritable, Text, Text, IntWritable>.Context context)
18         throws IOException, InterruptedException {
19         StringTokenizer itr = new StringTokenizer(value.toString());
20         while (itr.hasMoreTokens()) {
21             word.set(itr.nextToken());
22             context.write(word, one);
23         }
24     }
25 }
```

```
package hadoop;
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();

    @Override
    protected void reduce(Text key, Iterable<IntWritable> values,
        Reducer<Text, IntWritable, Text, IntWritable>.Context context)
        throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

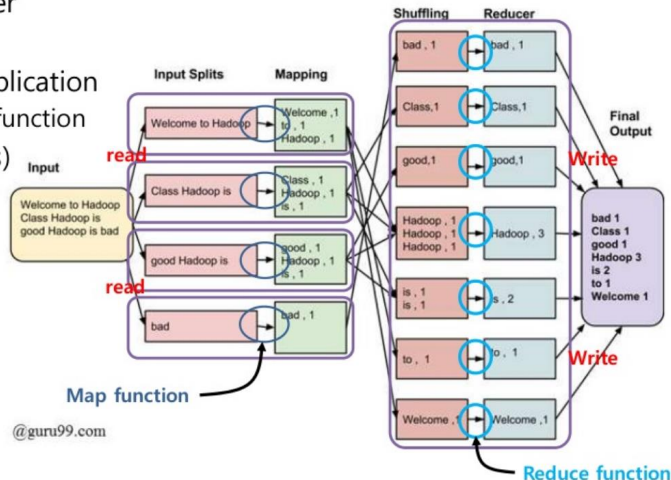
## 2.4 MapReduce 프로그래밍

- ✓ 모든 하둡 프로그램은 맵과 리듀스 단계를 거친다.
- ✓ Mapper는 데이터를 나누어 분산 처리하고 이를 다시 정렬하여 Reducer로 보낸다.
- ✓ Mapper가 데이터를 정렬하기 위해서 키가 필요하기 때문에 <key, value> 형태로 리턴 해야 한다.
- ✓ Reducer는 정렬된 <key, value> 를 받아 원하는 형태로 출력한다.

### Word Counting Example

- Computer cluster
- Input File
- Hadoop MR application
  - Map/Reduce function
- Output File(hdfs)

Computing Node





## 2.4 MapReduce 프로그래밍

- ✓ 데이터를 N개의 조각으로 나누어 처리하고 이를 종합할 수 있는 단순 업무에 적합함
- ✓ 빅데이터라고 해도 하둡이 적합하지 않은 업무도 많이 있음
- ✓ 데이터를 분할하여 연산이 불가능한 경우, 사용할 수 없음

- 웹로그에서 특정 페이지를 조회한 고객을 찾아라
- <xml>에서 그룹별로 특정 값을 더하는 등의 단순연산
- 빅 데이터에서 회귀계수 구하기

$$X'X = \begin{bmatrix} N & \sum x_1 & \sum x_2 & \sum x_3 \\ \sum x_1 & \sum x_1^2 & \sum x_1x_2 & \sum x_1x_3 \\ \sum x_2 & \sum x_2x_1 & \sum x_2^2 & \sum x_2x_3 \\ \sum x_3 & \sum x_3x_1 & \sum x_3x_2 & \sum x_3^2 \end{bmatrix} \quad X'y = \begin{bmatrix} \sum y \\ \sum X_1y \\ \sum X_2y \\ \sum X_3y \end{bmatrix}$$

$$\hat{\beta} = (X'X)^{-1}X'y$$

## 2.5 Word Count 예제

- ✓ 하둡을 실행한다.

```
$ start-all.sh
```

- ✓ 하둡이 제대로 실행되었는지를 JPS(JAVA Virtual Machine process status)확인  
네임노드와 데이터 노드 프로세스가 성공적으로 올라옴

```
$ jps
```

```
1984 NodeManager  
1430 SecondaryNameNode  
3430 Jps  
1255 DataNode  
1118 NameNode  
1646 ResourceManager
```

\$ 는 linux prompt를 의미함

## 2.5 Word Count 예제

- ✓ 하둡 작동 Test를 위해 ~/stack/hadoop/README.txt 파일에 있는 단어 수를 세기
- ✓ ~/stack/hadoop/README.txt 파일을 HDFS에 /example 폴더로 복사
- ✓ 자바로 만들어진 hadoop-mapreduce-examples-2.9.2.jar의 wordcount class를 이용하여 /example/README.txt 파일의 단어를 센 결과를 /output 폴더에 저장

```
$ hadoop fs -mkdir /example  
$ hadoop fs -copyFromLocal ~/stack/hadoop/README.txt /example  
  
$ hadoop jar hadoop-mapreduce-examples-2.9.2.jar wordcount /example/README.txt  
/output
```

리눅스 명령

```
$ hadoop fs -ls /output  
  
$ hadoop fs -cat /output/part-r-00000 | more
```

# 3. Apache Spark

2.1 하둡의 등장 배경

2.2 분산처리

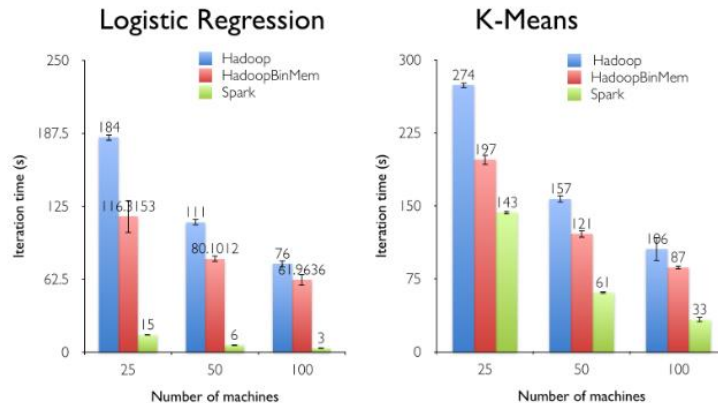
2.3 하둡 예코 시스템

2.4 맵리듀스 프로그래밍

2.5 Word Count 예제

## 3.1 Spark 소개

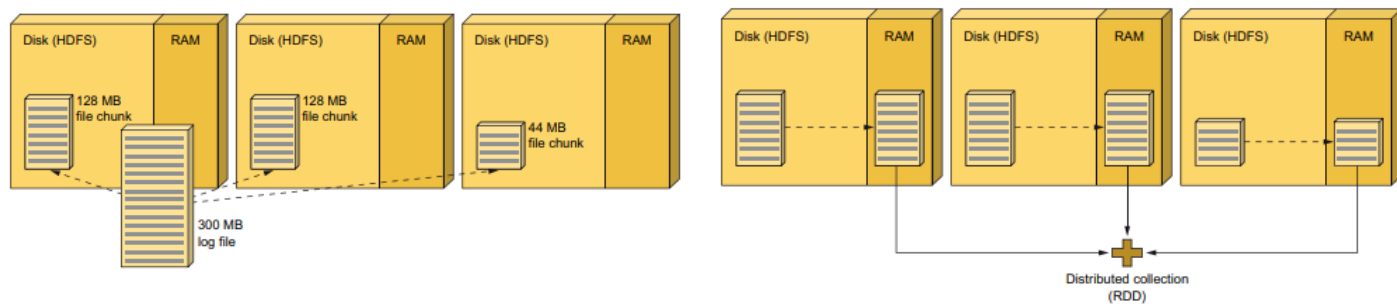
- Hadoop-Map Reduce보다 10~100배 빠른 속도를 자랑한다.
- 사용자가 분산 클러스터를 사용하고 있다는 사실을 인지할 필요 없다.  
즉, 일반적 DB를 사용하는 것처럼 명령하면 알아서 분산처리를 수행한다.
- 스파크는 스칼라, 파이썬, 자바, R 등에서 사용할 수 있다.
- Map Reduce는 처리 결과를 디스크에 저장하기 때문에 많은 디스크 I/O를 거치지만,  
스파크는 디스크 대신 메모리를 사용해 속도를 개선하였다.
- 하둡 에코 시스템의 상당 부분을 스파크로 대체 가능하다.



## 3.1 Spark 소개

- 아래는 Word Count 예제를 파이썬-스파크로 구현한 결과로 자바-하둡에 비해 쉽다.
- 예를 들어, 300MB 파일을 처리한다고 하면 아래와 같이 클러스터에 나누고 이를 메모리로 로드하여 처리하기 때문에 속도가 맵리듀스에 비해 10배 이상 빠르다.

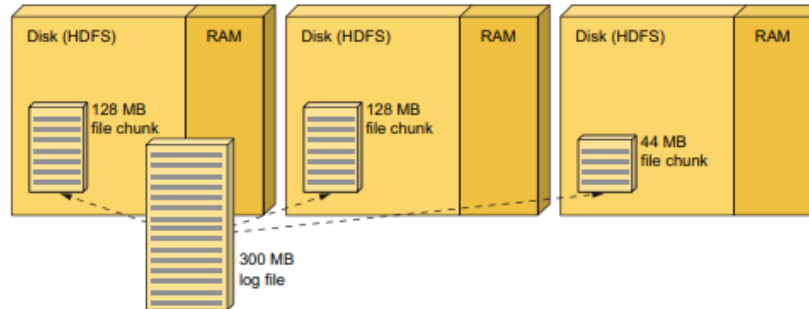
```
file = spark.textFile("hdfs://...")
counts = file.flatMap(lambda line: line.split()
                     .map(lambda word: (word, 1)) \
                     .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```



## 3.1 Spark 소개

- 스파크는 MLib 알고리즘을 지원한다. 때문에 로지스틱 회귀, 나이브 베이즈, SVM, Decision Tree, Random Forest, Regression, K-means Clustering 등 다양한 머신러닝 알고리즘을 분산처리할 수 있다.
- 엄청난 계산을 요구하는 그래프 이론을 분산처리로 구현하는 GraphX 기능을 제공한다.

```
file = spark.textFile("hdfs://...")
counts = file.flatMap(lambda line: line.split()
                    .map(lambda word: (word, 1)) \
                    .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```



## 3.2 Spark 설치

- 스파크 설치를 위해 <http://spark.apache.org/downloads.html>에서 Pre-built for Apache Hadoop 2.7을 받는다.
- 받은 파일을 압축을 해제한 후, spark 폴더로 이동한다.  
여기서는 /home/Hadoop/stack/spark를 스파크 홈으로 한다.
- .bashrc 파일에 아래와 같이 YARN\_CONF\_DIR과 SPARK\_HOME을 지정하고 PATH에 스파크 바이너리 폴더를 추가한다.
- 이후, source ~/.bashrc 명령으로 위의 지정을 활성화 시킨다.

```
export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
export SPARK_HOME=/home/hadoop/stack/spark
export PATH=$PATH:$SPARK_HOME/bin
```



- ```
Spark session available as 'spark'.  
Welcome to  
  
      / _ \   __| |__\___/\_/_/  
     W W    W W    \_/  /_\_  
    / _ \| ._|W_,||| /|\_W_W version 3.0.0  
       ||
```

```
Using Scala version 2.12.10 (OpenJDK 64-Bit Server VM, Java 1.8.0_222-ea)
Type in expressions to have them evaluated.
Type :help for more information.
```

- 스파크는 기본적으로 scala 라는 언어를 사용한다. 우리는 파이썬 언어를 사용할 것이다.
- \$ pyspark 명령으로 스파크 파이썬을 실행할 수 있지만, 불편하다.
- 아나콘다를 설치하고 주피터 노트북이나 파이참에서 스파크를 사용할 수 있도록 환경을 만든다.

## 3.3 pySpark 설치

- <https://www.anaconda.com/products/individual> 에서 리눅스용 바이너리 주소를 가져온다.
- \$ wget [https://repo.anaconda.com/archive/Anaconda3-2020.07-Linux-x86\\_64.sh](https://repo.anaconda.com/archive/Anaconda3-2020.07-Linux-x86_64.sh) 명령으로 아나콘다를 설치한다.
- pyspark 설치를 위해 아나콘다 프롬프트에서 pyspark를 설치한다.  
\$ conda activate  
\$ pip install pyspark
- start-all.sh 명령으로 하둡을 실행한다. hdfs 영역이 활성화한다.
- 스파크에서 사용할 모든 데이터는 hdfs 영역으로 복사해 사용해야 한다.
- \$ jupyter notebook을 실행하여 pyspark를 사용한다.

## 3.4 첫 예제

- 첫 예제로 /home/hadoop/stack/spark/LICENSE 파일의 라인수를 세어보자.
- 이를 위해서 LICENSE 파일을 hdfs 로 옮겨야 한다.

```
$ hadoop fs -copyFromLocal /home/hadoop/stack/spark/LICENSE
```

- LICENSE 파일은 hdfs root에 존재하므로 폴더명이 없다.
- 아래 결과로 LICENSE 파일의 라인수는 582임을 알 수 있다.

```
In [1]: import pyspark
```

```
In [2]: from pyspark import SparkContext  
sc = SparkContext()
```

```
In [4]: licLines = sc.textFile("LICENSE")  
lineCnt = licLines.count()
```

```
In [5]: lineCnt
```

```
Out[5]: 582
```

## 3.4 RDD

- 또한, 이 파일에서 “BSD” 를 포함하는 줄은 3개임을 알 수 있다.

```
bsdLines = licLines.filter(lambda line: "BSD" in line)
bsdLines.count()
```

- 여기서, bsdLines 를 RDD 라고 한다. RDD는 Resilient Distributed Dataset의 약어로 분산 된 데이터셋을 말한다.
- RDD는 빅데이터를 분산 저장하고 여기서 filter 혹은 count와 같은 연산을 수행한다.
- 파이썬 람다 함수를 풀어서 쓰면 아래와 같이 쓸 수 있다.

```
def isBSD(line):
    return "BSD" in line
bsdLines1 = licLines.filter(isBSD)
bsdLines1.count()
```

## 3.4 RDD

- 아래 코드는 “BSD” 를 포함하는 세 줄을 프린트하는 예다.
- bsdLine1은 RDD 이고 RDD의 collect 함수로 한줄씩 선택해 프린트 한다.

```
for line in bsdLines1.collect():  
    print(line)
```

```
BSD 2-Clause  
BSD 3-Clause  
is distributed under the 3-Clause BSD license.
```

- 아래 코드는 10, 20, ..., 50까지의 정수를 데이터로 가지는 RDD를 생성하는 예다.
- numbers는 RDD 이므로 프린트하면 pythonRDD 라고 출력된다.

```
numbers = sc.parallelize(range(10, 51, 10))  
print(numbers)
```

```
PythonRDD[27] at RDD at PythonRDD.scala:53
```

## 3.4 RDD

- 아래 코드는 numbers RDD의 모든 데이터를 맵 함수로 제곱을 수행한다.
- 수행한 결과 역시, RDD 다.
- 여기서, 알 수 있는 것은 사용자가 맵 리듀스 과정을 이해할 필요가 없다는 것이다.

```
numbersSquared = numbers.map(lambda num: num * num)
def println(col):
    for line in col.collect():
        print(line)
println(numbersSquared)
```

- RDD는 병렬로 동작하고 액션단계에서 수행되는 게으른 실행을 한다.
- 아래 코드는 숫자들에 대해 개수, 합, 평균, 표준편차와 도수를 구해주는 예다.

```
numbers = sc.parallelize([15,16,20,20,7,80,94,94,98,16,31,31,15,20])
println(numbers)
print(numbers.count(), numbers.sum(), numbers.mean(), numbers.stdev())
numbers.histogram([10,20,30,40,50,60,70,80,90,100])
```

## 3.5 Dataframe

- 데이터프레임은 RDD의 확장형으로 컬럼이름을 가지는 RDD이다.
- R/Python에서의 data frame/pandas모듈과 유사하고 Spark내부에서 최적화 할 수 있도록 하는 기능들이 추가되었다.
- JSON, CSV와 같이 컬럼정보가 포함된 텍스트 파일을 분산 저장한다.

```
import pyspark from pyspark
import SparkContext
sc = SparkContext()
from pyspark.sql import SparkSession
spark = SparkSession.builder\
    .master('yarn')\
    .appName('')\
    .getOrCreate()
```

## 3.5 Dataframe

- 아래는 videodata.json 파일을 데이터 프레임으로 생성하는 코드다.

```
videoJSON = spark.read.json("videodata.json")  
videoJSON.createOrReplaceTempView("videoJSON")
```

- 이 데이터는 고객이 어떤 동영상을 언제 오픈하고, 클로즈 했는지에 대한 로그 정보다.

```
{"customer_id":24,"show_id":308,"state":"open","timestamp":1510505885}  
{"customer_id":78,"show_id":580,"state":"open","timestamp":1510505888}  
{"customer_id":84,"show_id":378,"state":"open","timestamp":1510505895}  
{"customer_id":71,"show_id":830,"state":"open","timestamp":1510505898}  
{"customer_id":64,"show_id":473,"state":"open","timestamp":1510505903}  
{"customer_id":58,"show_id":648,"state":"open","timestamp":1510505908}  
{"customer_id":25,"show_id":842,"state":"open","timestamp":1510505915}  
{"customer_id":67,"show_id":628,"state":"open","timestamp":1510505919}  
{"customer_id":57,"show_id":721,"state":"open","timestamp":1510505923}
```



## 3.5 Dataframe

```
videoJSON.printSchema()
```

```
root
|-- customer_id: long (nullable = true)
|-- show_id: long (nullable = true)
|-- state: string (nullable = true)
|-- timestamp: long (nullable = true)
```

```
videoJSON.count()
```

```
1166450
```

```
spark.sql("select distinct state from videoJSON").collect()
```

```
[Row(state='finish_incomplete'),
 Row(state='heartbeat'),
 Row(state='open'),
 Row(state='finish_completed')]
```

”open”—the customer starts watching a new show

”heartbeat”—the customer is still watching this show

”finish\_completed”—the customer finishes the entire show

”finish\_incomplete”—the customer closed the show without finishing it

## 3.5 Dataframe

```
videoJSON.show()
```

| customer_id | show_id | state            | timestamp  |
|-------------|---------|------------------|------------|
| 24          | 308     | open             | 1510505885 |
| 78          | 580     | open             | 1510505888 |
| 84          | 378     | open             | 1510505895 |
| 71          | 830     | open             | 1510505898 |
| 64          | 473     | open             | 1510505903 |
| 58          | 648     | open             | 1510505908 |
| 25          | 842     | open             | 1510505915 |
| 67          | 628     | open             | 1510505919 |
| 57          | 721     | open             | 1510505923 |
| 54          | 729     | open             | 1510505930 |
| 38          | 107     | open             | 1510505933 |
| 79          | 788     | open             | 1510505940 |
| 33          | 841     | open             | 1510505941 |
| 46          | 566     | open             | 1510505946 |
| 25          | 842     | finish_completed | 1510505952 |
| 76          | 91      | open             | 1510505959 |
| 90          | 707     | open             | 1510505965 |
| 85          | 413     | open             | 1510505966 |
| 4           | 200     | open             | 1510505975 |
| 81          | 45      | open             | 1510505979 |

only showing top 20 rows

## 3.5 Dataframe

```
videoJSON.printSchema()
```

```
root
|-- customer_id: long (nullable = true)
|-- show_id: long (nullable = true)
|-- state: string (nullable = true)
|-- timestamp: long (nullable = true)
```

```
videoJSON.count()
```

```
1166450
```

```
spark.sql("select distinct state from videoJSON").show()
```

```
+-----+
|          state|
+-----+
|finish_incomplete|
|          heartbeat|
|              open|
|  finish_completed|
+-----+
```

”open”—the customer starts watching a new show

”heartbeat”—the customer is still watching this show

”finish\_completed”—the customer finishes the entire show

”finish\_incomplete”—the customer closed the show without finishing it

## 3.5 Dataframe

- 이 데이터에서 동영상 별로 open은 했으나 다보지 않고 중단한 비율을 구해보자.
- 아래 세 개의 명령으로 비디오 ID별로 open 회수와 finish\_incomplete 회수를 구하고 이를 left join 하여 다시 데이터프레임을 생성했다.

```
showid_open = spark.sql("select show_id, count(*) as cnt_open  
                        from videoJSON  
                        where state='open'  
                        group by show_id")
```

```
showid_incomplete = spark.sql("select show_id, count(*) as cnt_Fin  
                              from videoJSON  
                              where state='finish_incomplete'  
                              group by show_id")
```

```
left_join = showid_open.join(showid_incomplete,  
                             showid_open.show_id == showid_incomplete.show_id,  
                             how = 'left')
```

## 3.5 Dataframe

- 데이터 프레임을 판다스로 가져와 아래와 같이 연산하여 구할 수 있다.

```
import pandas as pd
left_join_PD = left_join.toPandas()
```

```
1 left_join_PD['ratio'] = left_join_PD['cnt_Fin'] * 100 / left_join_PD['cnt_open']
2 left_join_PD
```

|     | show_id | cnt_open | show_id | cnt_Fin | ratio     |
|-----|---------|----------|---------|---------|-----------|
| 0   | 26      | 635      | 26      | 177     | 27.874016 |
| 1   | 29      | 676      | 29      | 158     | 23.372781 |
| 2   | 474     | 647      | 474     | 186     | 28.748068 |
| 3   | 964     | 666      | 964     | 166     | 24.924925 |
| 4   | 65      | 694      | 65      | 190     | 27.377522 |
| ... | ...     | ...      | ...     | ...     | ...       |
| 995 | 458     | 651      | 458     | 178     | 27.342550 |
| 996 | 739     | 705      | 739     | 150     | 21.276596 |
| 997 | 211     | 671      | 211     | 160     | 23.845007 |
| 998 | 469     | 642      | 469     | 162     | 25.233645 |
| 999 | 526     | 694      | 526     | 164     | 23.631124 |

## 3.5 Dataframe

- 이제 다른 예제를 해보자.
- 이 예제는 항공 데이터로 airport-codes-na.txt와 departuredelays.csv로 구성되어 있다.
- 공항정보는 4개 컬럼이 탭으로 구분되어 있다.
- IATA는 공항코드를 나타내고 나머지는 이 공항의 위치를 나타낸다.

```
airports = spark.read.csv("airport-codes-na.txt", header = True,  
                          inferSchema = True, sep='\t')  
airports.createOrReplaceTempView("airports")
```

```
airports.show()
```

| City       | State | Country | IATA |
|------------|-------|---------|------|
| Abbotsford | BC    | Canada  | YXX  |
| Aberdeen   | SD    | USA     | ABR  |
| Abilene    | TX    | USA     | ABI  |
| Akron      | OH    | USA     | CAK  |
| Alamosa    | CO    | USA     | ALS  |
| Albany     | GA    | USA     | ABY  |

## 3.5 Dataframe

- 연착정보는 아래와 같이 날짜, 연착시간(분), 거리(마일) 그리고 출발지, 도착지 정보다.
- delay가 음수인 것은 일찍 도착한 것 같다.

```
flightPerf = spark.read.csv("departuredelays.csv", header = True)
flightPerf.createOrReplaceTempView("FlightPerformance")
```

```
flightPerf.show()
```

| date     | delay | distance | origin | destination |
|----------|-------|----------|--------|-------------|
| 01011245 | 6     | 602      | ABE    | ATL         |
| 01020600 | -8    | 369      | ABE    | DTW         |
| 01021245 | -2    | 602      | ABE    | ATL         |
| 01020605 | -4    | 602      | ABE    | ATL         |
| 01031245 | -4    | 602      | ABE    | ATL         |
| 01030605 | 0     | 602      | ABE    | ATL         |
| 01041243 | 10    | 602      | ABE    | ATL         |
| 01040605 | 28    | 602      | ABE    | ATL         |

## 3.5 Dataframe

- 워싱턴 주에 있는 공항은?

```
spark.sql("select distinct city from airports where state='WA']").show()
```

- 워싱턴 주에 있는 공항에서 출발한 건을 나열하라?

```
spark.sql("select distinct a.state, a.City, f.origin, f.destination from
FlightPerformance f join airports a on a.IATA = f.origin where a.State =
'WA').collect()
```

- 위 건의 연착시간의 합을 출발지 별로 구하시오

```
# Query Sum of Flight Delays by City and Origin Code (for Washington
State)spark.sql("select a.City, f.origin, sum(f.delay) as Delays from
FlightPerformance f join airports a on a.IATA = f.origin where a.State =
'WA' group by a.City, f.origin order by sum(f.delay) desc").show()
```

| City    | origin | Delays   |
|---------|--------|----------|
| Seattle | SEA    | 159086.0 |
| Spokane | GEG    | 12404.0  |
| Pasco   | PSC    | 949.0    |



## 3.5 Dataframe

- 일반적으로 텍스트 데이터를 정형 DB로 변환하면 5배~10배 정도 크기가 커진다.
- 빅데이터의 경우, 이러한 사이즈 증가는 엄청난 문제다.
- 지금까지 본 바와 같이 JSON, CSV와 같은 텍스트 데이터를 그대로 정형 DB처럼 SQL을 사용할 수 있다는 점이 스파크의 놀라운 장점이다.

## 3.6 Data 다루기

- 스파크를 데이터베이스 엔진이라고 생각하면 오산이다.
- 스파크는 데이터베이스 기능부터 머신러닝 기법까지 모든 기능을 제공한다.
- 예를 들어, 데이터에 Missing Value 처리, 극단값 탐지 등 통계적 기법을 상단히 수용했다.

끝?

끝? 이 아닌 이제 시작이다.

