

# Hadoop & Ecosystem

Kim Hye Kyung

topickim@naver.com

# 목차

---

- I 빅데이터
- II Hadoop
- III Yarn
- IV HDFS
- V 맵리듀스
- VI Cloudera
- VII 하이브
- VIII 스쿱

# 빅데이터

# 빅데이터 개념

- 빅데이터 시대는 데이터를 단순 정보로만 보지 않음

과거로부터 현재까지 쌓인 데이터를 분석

현재를 이해하고 이 정보에서 만들어지는 다양한 패턴들을 해석

미래를 예측하기 시작한 것



# 빅데이터 정의

빅데이터는 통상적으로 사용되는 데이터 수집 및 관리, 처리와 관련된 소프트웨어의 수용 한계를 넘어서는 크기의 데이터를 의미하며, 빅데이터의 규모는 단일 데이터 집합의 크기가 수십 테라바이트에서 수 페타바이트에 이르며, 그 크기가 끊임없이 변화하는 것이 특징이다

- 위키피디아 -

일반적인 데이터베이스 소프트웨어로 저장/관리/분석 할 수 있는 범위를 초과하는 규모의 데이터

- 맥킨지 -

# 빅 데이터 정의

대용량 데이터를 활용/분석해서 가치 있는 정보를 추출하고, 생성된 지식을 바탕으로 능동적으로 대응하거나 변화를 예측하기 위한 정보화 기술

- 국가정보화전략위원회 -

단순한 데이터의 크기가 아니라 데이터의 형식과 처리 속도 등을 함께 아우르는 개념으로, 기존 방법으로는 데이터의 수집, 저장, 검색, 분석 등이 어려운 데이터를 총칭해서 일컫는 용어

- ITWorld, 2012 -

# 빅데이터가 중요한 이유

- 빅데이터를 알아야 살아 남는다!!!

데이터를 이해하는 능력, 데이터를 처리하는 능력, 가치를 뽑아내는 능력, 시각화하는 능력, 전달하는 능력이야말로 누구에게나 앞으로 오랫동안 매우 중요한 능력이 될 것이다.

-Hal Varian-

# 빅데이터 학습이 중요한 이유

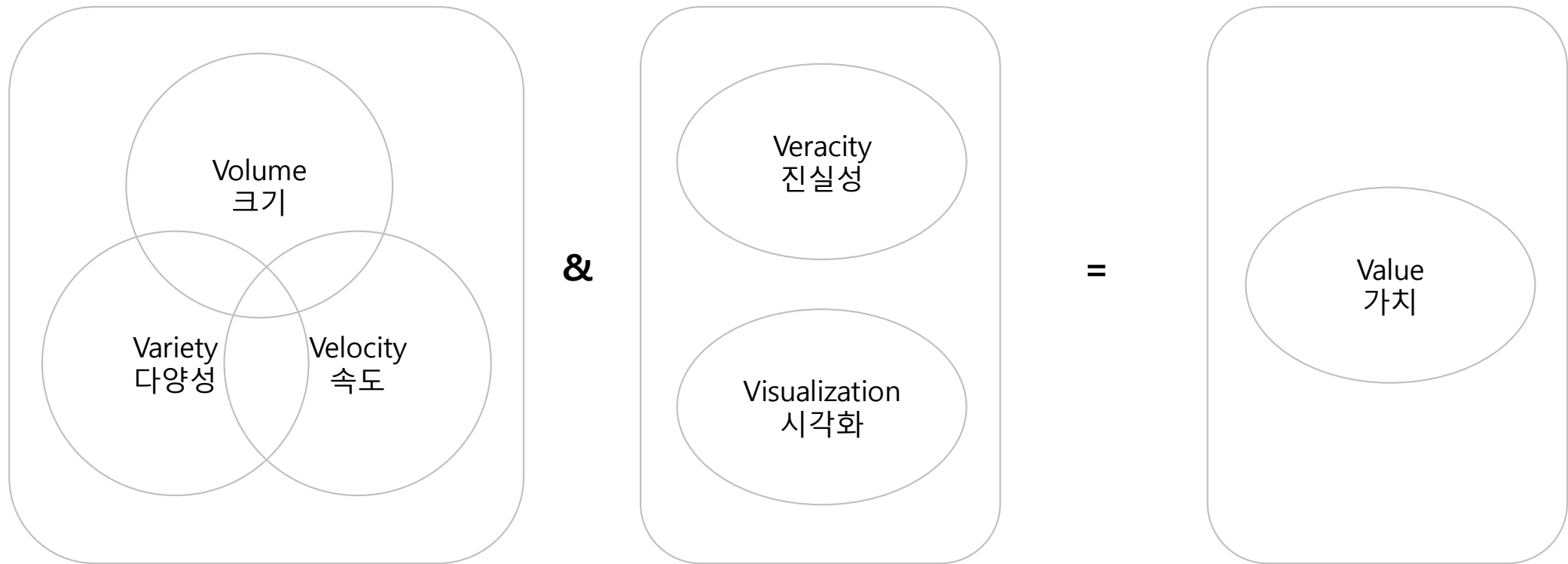
빅데이터가 모든 문제를 해결해 준다는 주장도 있지만 데이터의 처리, 분석, 해석엔  
**인간의 판단이 중요**

IT 업계에서도 단순히 빅데이터뿐만 아니라  
**'빅데이터를 읽어내는 분석력'**이 중요하다는 인식이 확산



# 빅데이터 정의 - 6V 모델

- 데이터의 기하 급수적인 성장, 데이터의 가용성, 데이터의 정보화 관점에서의 정의



# 빅데이터 정의 – 6V 모델

크기(Volume) : 방대한 양의 데이터(테라, 페타바이트 이상의 크기)

다양성(Varity) : 정형(DBMS, 전문 등) + 비정형(SNS, 동영상, 사진, 음성, 텍스트)등

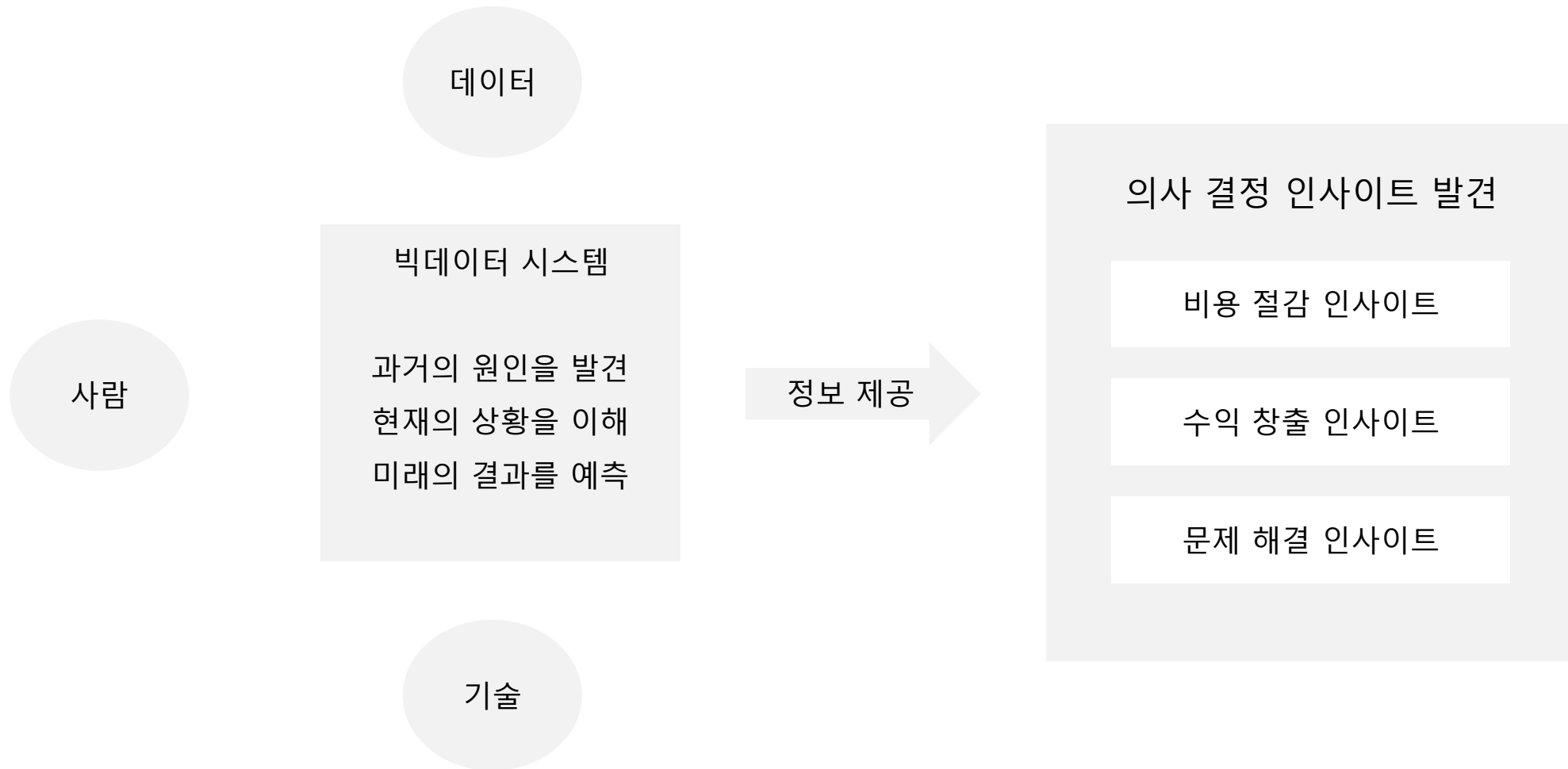
속도(Velocity) : 실시간으로 생산되며, 빠른 속도로 데이터를 처리/분석

진실성(Veracity) : 주요 의사 결정을 위해 데이터의 품질과 신뢰성 확보

시각화(Visualization) : 복잡한 대규모 데이터를 시각적으로 표현

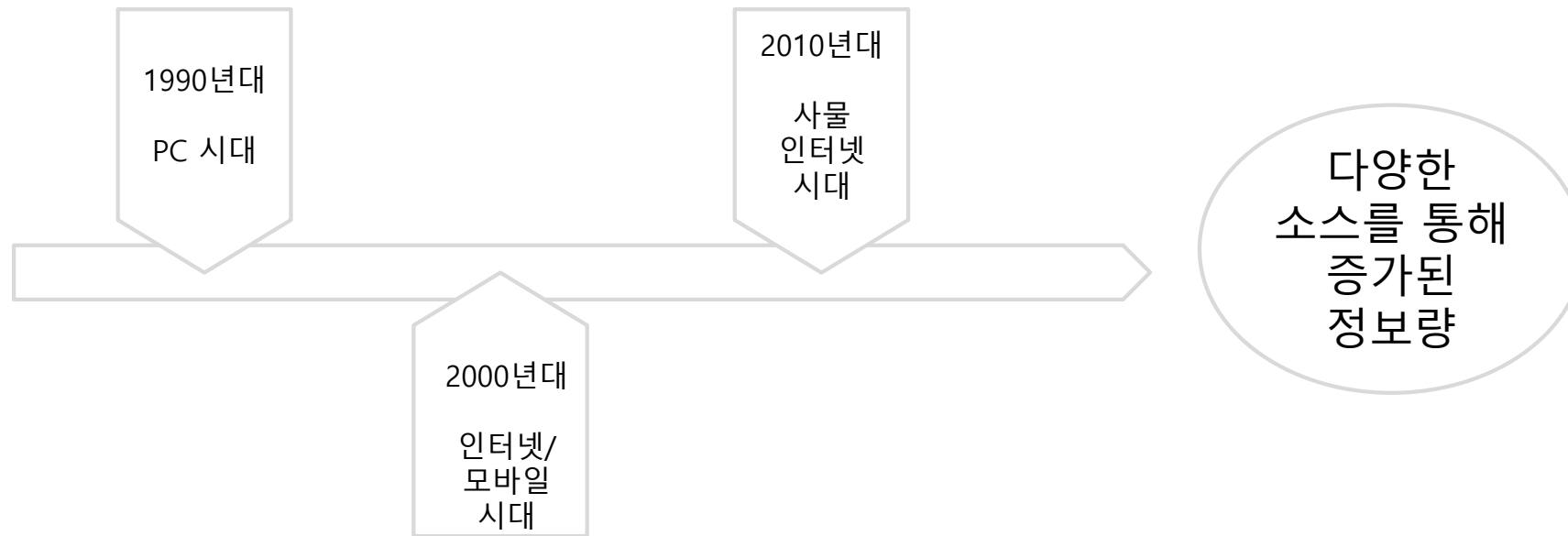
가치(Value) : 비즈니스 효익을 실현하기 위해 궁극적인 가치를 창출

# 빅데이터의 목적



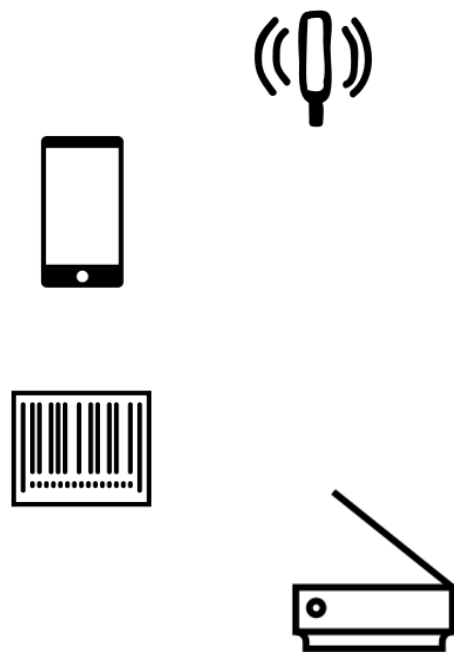
# 빅데이터 처리 기술의 필요성

- 빅데이터 특징 : High-Volume
  - 데이터 볼륨 증가



# 빅데이터 처리 기술의 필요성

- 빅데이터 특성 : High-Velocity
  - 데이터 발생 속도 증가(Velocity)
  - 빠른 입출력 속도



- 데이터 증가 원인 -

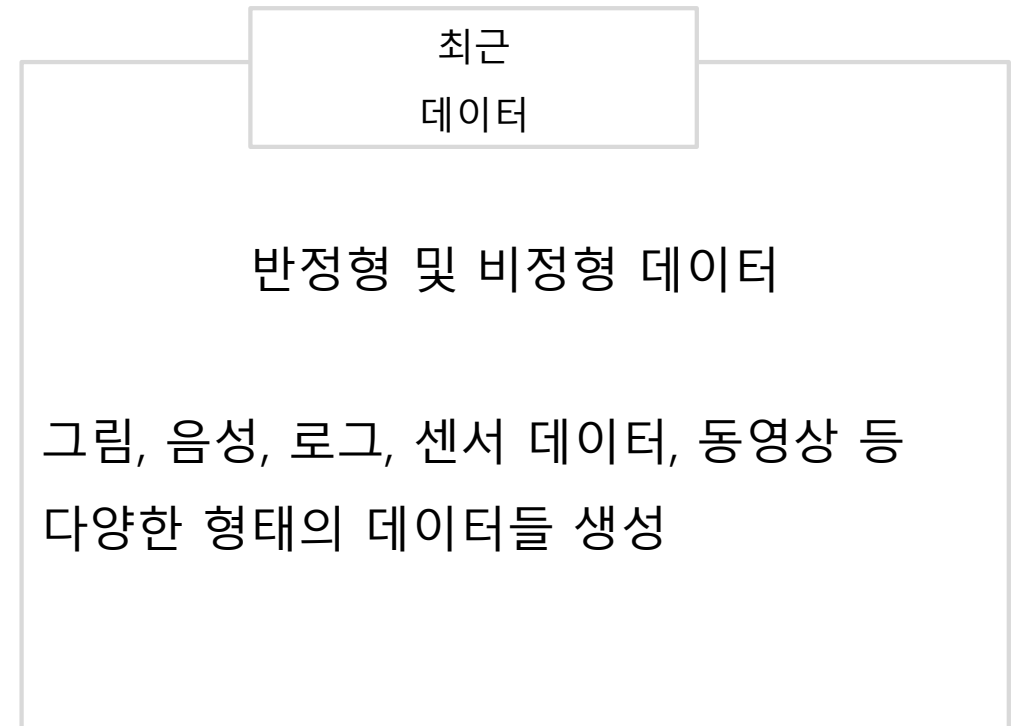
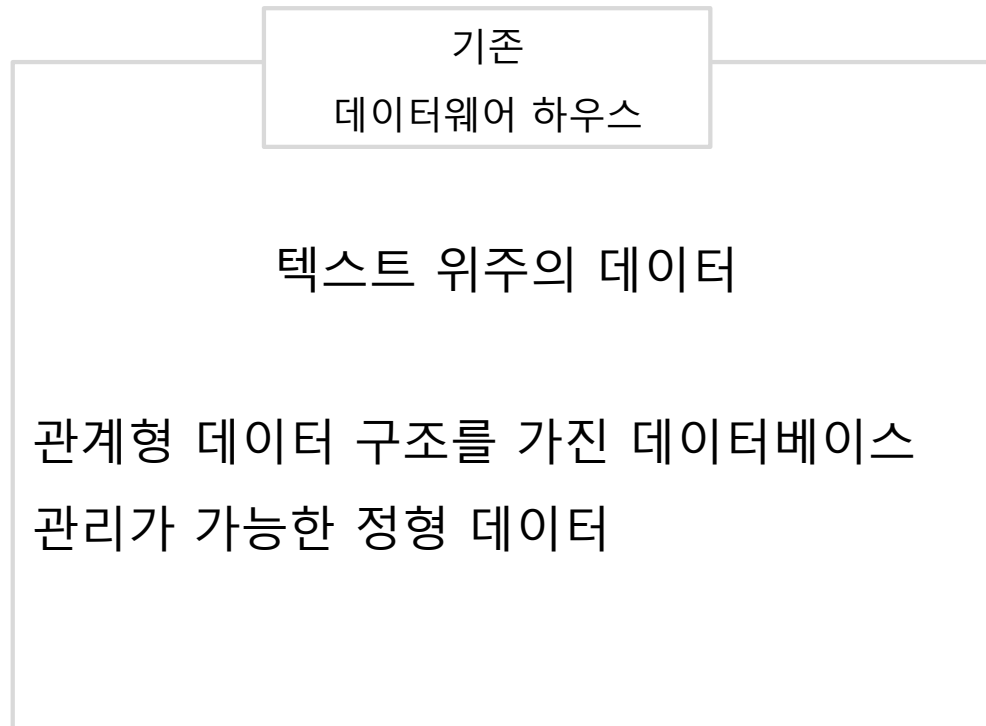
- 1) 모바일 장치 확산
- 2) 소셜 미디어의 성장
- 3) 센서 장비 이용의 확대
- 4) GPS 장치 보급 확산
- 5) 인터넷 이용 증가

등의 장치를 통한 데이터 생성 속도가 매우 빨라짐

...

# 빅데이터 처리 기술의 필요성

- 빅데이터 특성 : High-Variety
  - 데이터 포맷의 다양성 증가(Variety)



# 빅데이터 처리 기술의 필요성

- 인간과 데이터



# 빅데이터 과학 및 과학자

---

- 데이터 과학
  - 빅데이터를 분석하고 연구하는 학문
- 데이터 과학자
  - 데이터에서 의미 있는 정보를 도출하는 사람



# 빅데이터 프로젝트

---

- 세 가지 유형

플랫폼 구축형 프로젝트

빅데이터 분석 프로젝트

빅데이터운영 및 관리

# 빅데이터 프로젝트

- 세 가지 유형

## 플랫폼 구축형 프로젝트

전형적인 빅데이터 SI(System Integration) 구축형 사업  
빅데이터의 하드웨어와 소프트웨어 설치 및 구성하고 빅데이터의 기본 프로세스인  
수집 -> 적재 -> 탐색 -> 분석의 기능 구성

## 빅데이터 분석 프로젝트

분석 주제에 따른 세가지 영역으로 분류  
마케팅 분석 영역 – 주로 고객 분석을 통해 차별화된 맞춤 전략 수립  
상품/서비스 분석 영역 – 가격, 디자인, 타겟팅, 출시일 등 결정하는 데 활용되는 영역  
리스크 분석 영역 – 기업 운영시 내,외부 리스크를 사전에 예측하는데 활용하는 영역

## 빅데이터 운영 및 관리

플랫폼 구축형 & 빅데이터 분석의 두 가지 프로젝트를 모두 유지 관리하는 가장 중요한 프로젝트로 "빅데이터 운영"이라 부름

# 빅데이터 처리 기술의 필요성

- 빅데이터 유형

형태		특징
정형 데이터	DBMS에 적합한 구조의 데이터들 지정된 행과 열에 의해 데이터의 속성이 구별되는 스프레드시트 형태의 데이터  재무정보, 재고 관리, 인사 정보, 거래 정보등	내부 시스템인 경우가 대부분이라 수집이 쉬움 파일 형태의 스프레드시트라도 내부에 형식을 가지고 있어 처리가 쉬움
반정형 데이터	정형과 비정형의 중간 형태  URL 형태로 존재 - HTML 오픈 API 형태로 제공 - XML, JSON 로그형태 - 웹로그, IOT에서 제공하는 센서 데이터	보통 API 형태로 제공되기 때문에 데이터 처리 기술이 요구
비정형 데이터	이진 파일 형태: 동영상, 이미지 스크립트 파일 형태: 소셜 데이터의 텍스트	텍스트 마이닝 혹은 파일일 경우 파일을 데이터 형태로 파싱해야 하기 때문에 수집 데이터 처리가 어려움

# 빅데이터 처리 기술의 필요성

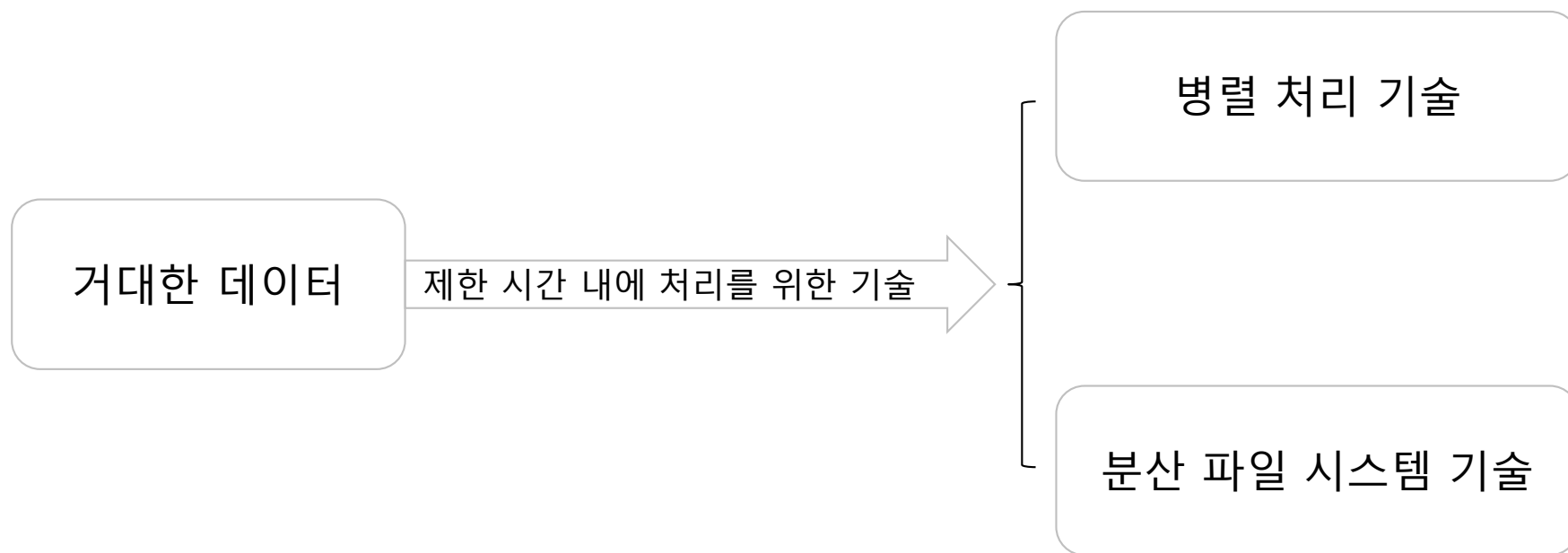
- 데이터의 존재적 특징으로 데이터 구분
  - **정성적 데이터** : 데이터 자체가 하나의 텍스트를 이루고 있기 때문에 데이터 하나 하나가 함축된 정보를 갖고 있음
    - (예시 : “환율이 내리고 있어 올해 목표한 수출 목표의 조기 달성이 가능해 보임”)
  - **정량적 데이터**는: 여러 속성(이름, 나이, 성별, 주소 등)이 모여 하나의 객체를 형성하고, 각 속성은 속성 하나 혹은 여러 개의 속성이 결합해 측정이나 설명이 가능하도록 구성

구분	정성적 데이터	정량적 데이터
형태	비정형 데이터	정형 · 반정형 데이터
특징	객체 하나에 함축된 정보를 갖고 있음	속성이 모여 객체를 이룸
구성	언어, 문자 등으로 이루어짐	수치, 도형, 기호 등으로 이루어짐
저장 형태	파일, 웹	데이터베이스, 스프레드시트
소스위치	외부 시스템(주로 소셜 데이터)	내부 시스템(주로 DBMS)

# 빅데이터 처리 기술의 필요성

- 데이터 획득
  - 다양한 데이터 원천으로 부터 다양하고 가공하지 않은 데이터가 있기 때문에 데이터를 거르고, 압축하고, 데이터를 정제한 후 정보를 추출하는 것이 과제
- 정보 저장과 체계화
  - 가공하지 않은 데이터(raw data)로 부터 정보를 뽑아 내면 데이터 모델을 만들어 저장 장비에 저장
  - 거대 데이터 집합을 효율적으로 저장하는데 있어서 전통적 관계형 시스템은 큰 데이터 단위 때문에 효율적으로 동작하지 않을 것
  - 주로 빅데이터를 다루는 NoSQL 데이터베이스 등장 및 활용
- 정보 검색과 분석
  - 데이터 저장은 데이터 웨어하우스(warehouse)를 구성하는 작업중 일부 작업
  - 데이터는 처리되어야만 유용
  - 처리된 데이터로 만든 정보는 검색, 데이터 마이닝, 행동 모델을 만드는 분석에 사용

# 빅데이터 처리 기술의 필요성



# 빅데이터 처리 기술의 필요성

- **RDBMS vs 빅데이터**

	RDBMS	빅데이터
데이터	정형 데이터	비정형 데이터 문자, 동영상, 이미지 데이터등
하드웨어	고가의 저장 장치 관계형 데이터베이스 서버 Data Warehouse	저가의 Linux 또는 클라우드 환경
소프트웨어	관계형 데이터베이스 시스템 통계 패키지등	오픈소스 형태의 무료 패키지 무료 통계 솔루션(R or Python..)

# 빅데이터 처리 기술의 필요성

- 빅데이터 시스템이란?
  - 대용량의 데이터를 분산 병렬 처리하고 관리하는 시스템
  - 대규모 양의 데이터 수집, 관리, 유통, 분석을 처리하는 일련의 분산 병렬 처리 프레임워크

프레임워크란?

소프트웨어의 구체적인 부분에 해당하는 설계과 구현을 재사용 가능한 협업 형태로 표준화 하고 제공하는 소프트웨어 환경



# 빅데이터 처리 기술의 필요성

- 빅데이터 처리 기술의 요구 사항

실시간 데이터를 처리

많은 데이터를 저비용으로 처리

결함 허용이 되는 시스템이 필요

분산 처리 가능

# 빅데이터 처리 방식

---

- 빅데이터 처리 방식의 종류

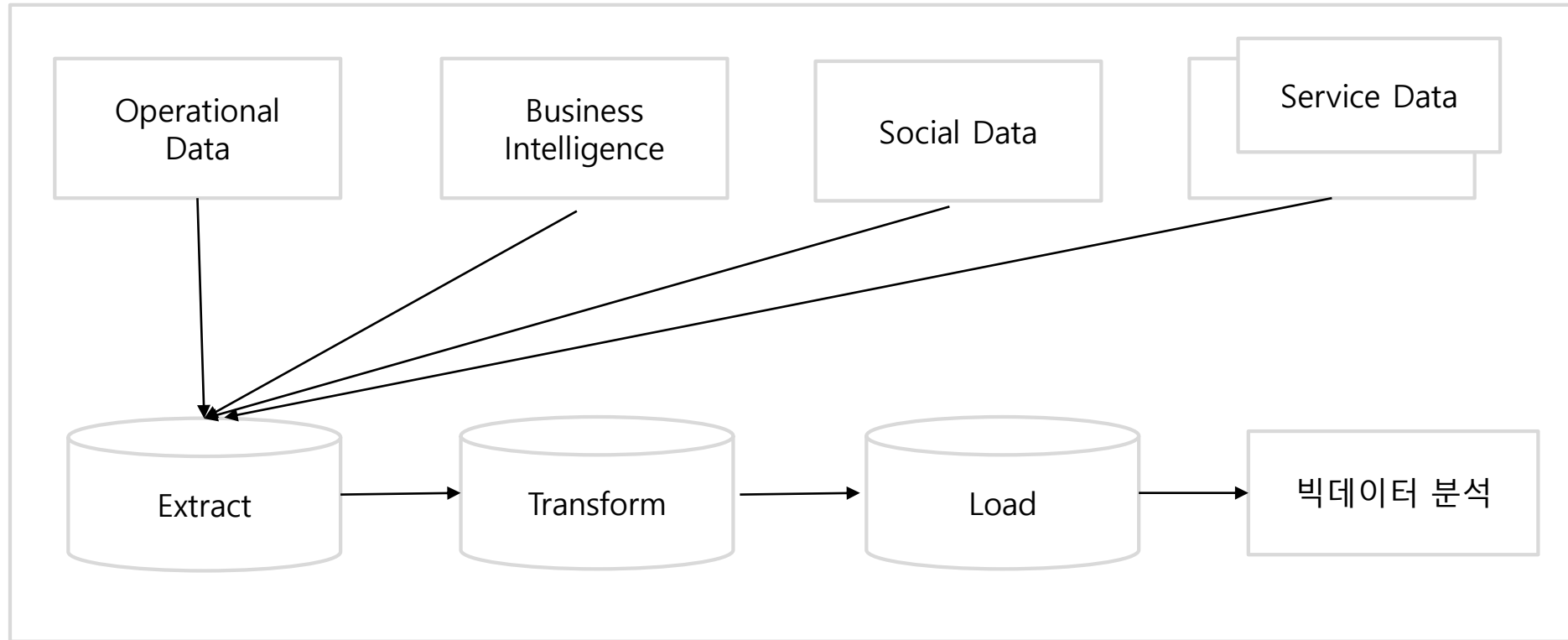
배치 처리  
(Batch Processing)

대화형 처리  
(Interactive Processing)

실시간 처리  
(Real-time Processing)

# 빅데이터 처리 방식

- 빅데이터 처리 배치 or 일괄 처리 방식



데이터를 적재 -> 일정 시간 간격으로 데이터 수집(cron / job scheduler) -> 형태 변경 -> 로드 -> 로드된 데이터 분석

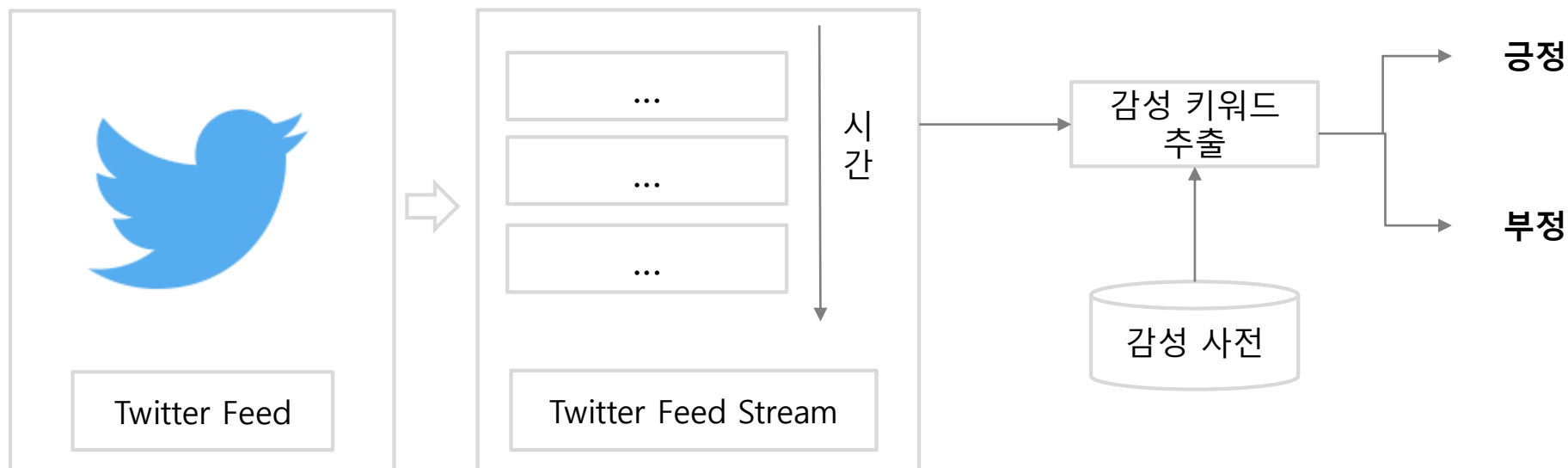
# 빅데이터 처리 방식

---

- 대화형 처리 방식
  - 사용자와 시스템간의 대화
  - 수초내에 사용자가 요청한 데이터가 처리되어야 함
    - 예시 : sql문장으로 sqlplus 사용과 흡사
  - 대표적인 기술
    - 하이브
    - 피크
    - 스파크
    - ...

# 빅데이터 처리 방식

- 실시간 처리
  - 수 초 미만 또는 1초 미만의 실시간 처리 및 이벤트성 응답
  - 데이터가 수집되는 즉시 실시간 전처리, 실시간 계산, 실시간 패턴 분석을 처리
    - 결제나 비정상 카드 사용등에 대한 데이터 분석에 사용
  - 예시
    - Sns의 실시간 데이터 스트림 처리



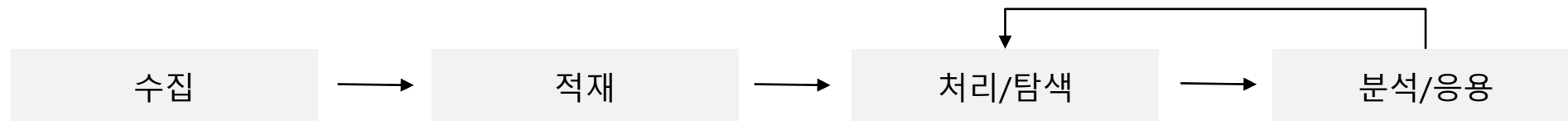
# 빅데이터 구현 기술[아키텍처]

- 역할별로 6단계로 구분

단계	역할	활용기술	
↓ 수집	내, 외부 데이터 연동 내, 외부 데이터 통합	crawling, Open API, RSS, Log Aggregation, DB Aggregation, Streaming	전처리
↓ 적재	대용량/실시간 데이터 처리 분산 파일 시스템 저장	Distributed File, No-SQL Memory Cached, Message Queue	
↓ 처리	데이터 선택, 변환, 통합, 축소 데이터 워크플로 및 자동화	Structured Processing, UnStructured Processing Workflow, Scheduler	
↓ 탐색	대화형 데이터 질의 탐색적 분석	SQL Like Distributed Programming Exploration Visualization	후처리
↓ 분석	빅데이터 마트 구성 통계 분석, 고급 분석	Data Mining Machine Learning Anaysis Visualization	
↓ 응용	보고서 및 시각화 분석 정보 제공	Data Export/Import, Reporting Business Visualization	활용

# 빅데이터 구현 기술[아키텍처]

- 빅데이터 구축 단계
  - 처리 및 탐색 단계와 분석 및 응용 단계
    - 반복 진행하면서 데이터 품질과 분석 수준 향상



# 빅데이터 구현 기술[아키텍처] : 수집

- 빅데이터 구축 단계 : 수집



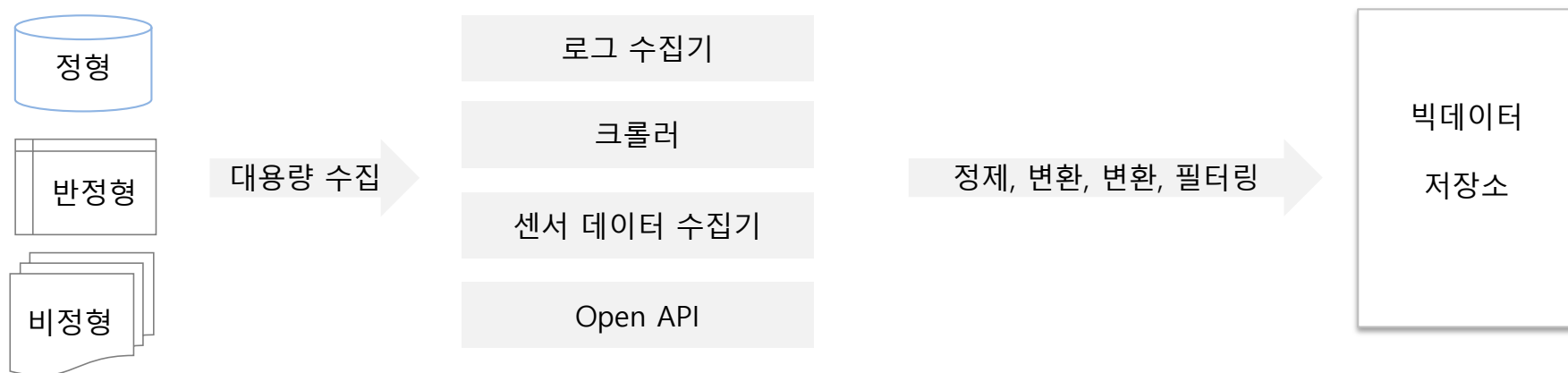
- 원천 데이터를 효과적으로 수집하는 기술
- 기존의 수집 시스템(EAI, ETL, ESB 등) 보다 더 크고 다양한 형식의 데이터를 빠르게 처리해야 하는 기능 필요





# 빅데이터 구현 기술[아키텍처] : 수집

- 빅데이터 구축 단계 : 수집
  - Hadoop & Ecosystem 관점에서의 수집 기술

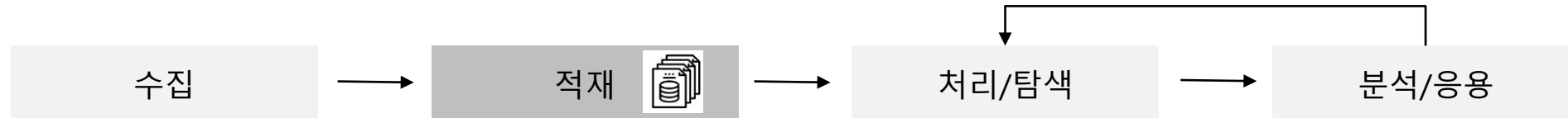


<빅데이터 수집의 주요 기술>

기술	개발	최초 공개	주요 기능 및 특징
Sqoop	아파치	2009년	RDBMS와 HDFS(NoSQL) 간의 데이터 연동
Flume	Cloudera	2010년	방대한 양의 이벤트 로그 수집
Kafka	Linkedin	2010년	분산 시스템에서 메시지 전송 및 수집
Logstash	Elastic	2010년	데이터 수집 및 로그 파싱 엔진

# 빅데이터 구현 기술[아키텍처] : 적재

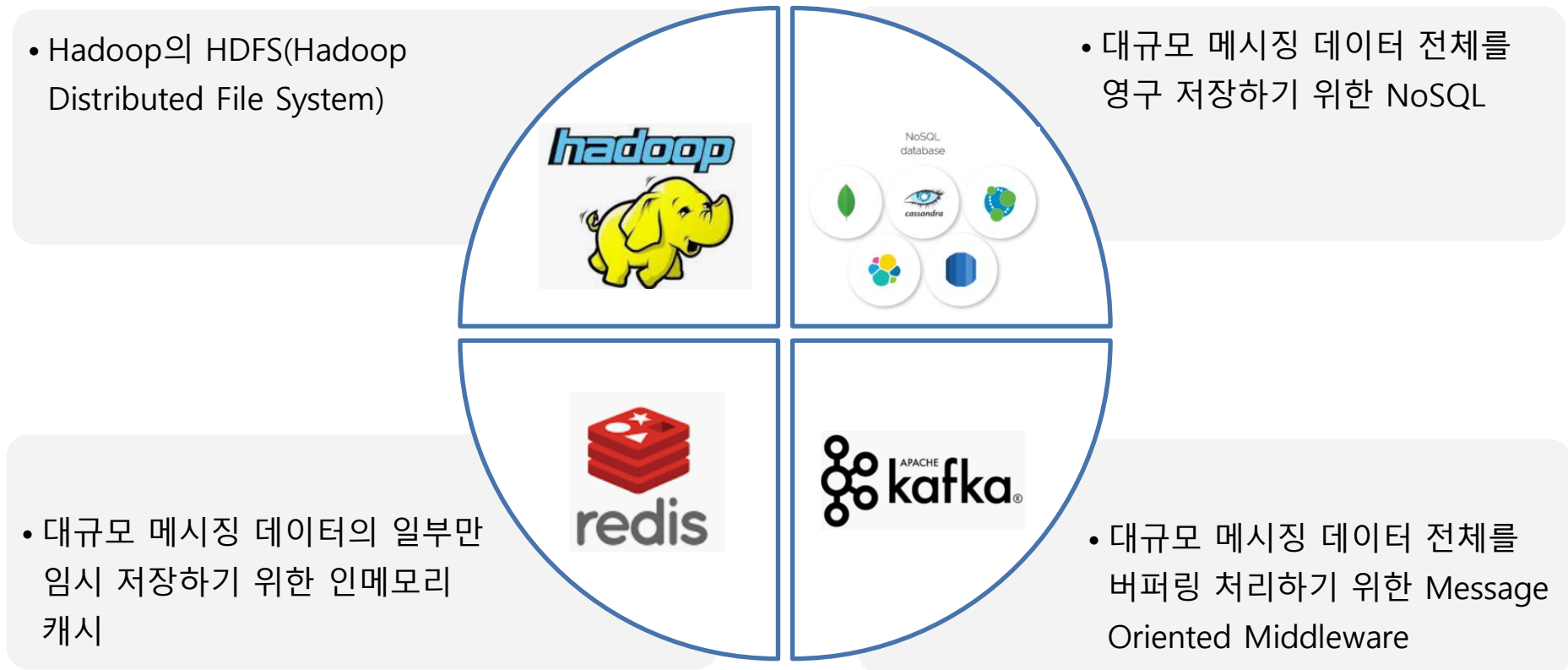
- 빅데이터 구축 단계 : 적재



- 수집한 데이터를 분산 스토리지에 영구 저장 또는 임시로 적재
- 빅데이터 적재시 추가적인 전처리 작업
  - 탐색, 분석 단계를 위해 비정형(음성, 이미지, 텍스트, 동영상 등) 데이터를 정형 데이터(스키마가 있는 구조)로 가공
  - 개인 정보로 의심되는 데이터를 비식별화 처리 하는 작업 선행

# 빅데이터 구현 기술[아키텍처] : 적재

- 빅데이터 구축 단계 : 적재
  - 빅데이터 분산 저장소 종류



# 빅데이터 구현 기술[아키텍처] : 처리/탐색

- 빅데이터 구축 단계 : 처리/탐색



- 대용량 저장소에 적재된 데이터를 분석에 활용하기 위해 데이터를 정형화 및 정규화 하는 기술
- 대규모로 적재된 데이터를 대상으로 처리하기 때문에 크기에 대한 처리 기술이 중요
- 데이터 후처리 작업과 정규화 과정을 통해 데이터의 진실성을 확보하고 후처리된 데이터셋을 시각화 툴로 더욱 용이하게 탐색 가능하게 함
- 처리/탐색 기술
  - Hive(하이브)

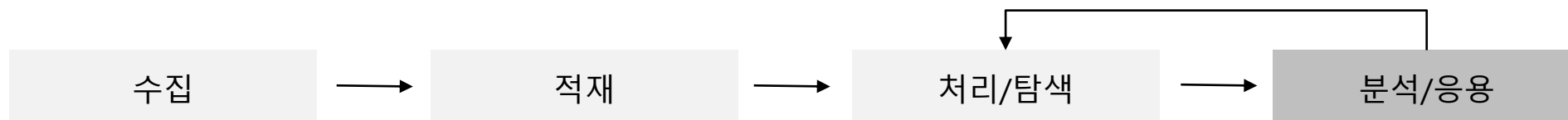


Spark SQL



# 빅데이터 구현 기술[아키텍처] : 분석/응용

- 빅데이터 구축 단계 : 분석/응용



- 대규모 데이터로부터 새로운 패턴을 찾고, 그 패턴을 해석해서 통찰력을 확보하기 위한 기술
- 주 목적
  - 과거의 데이터로부터 문제의 원인을 찾아 현재를 개선할 뿐 아니라 인간의 힘으로 찾기 어려웠던 패턴들을 빅데이터 분석 기술로 찾아 알고리즘화 해서 미래를 예측하는 분석 모델을 만드는데 기여
- 활용 영역에 따른 구분
  - 통계, 데이터 마이닝, 텍스트 마이닝, 소셜 미디어 등 다양하게 분류

# 빅데이터 처리 기술의 프로세스



# 빅데이터 처리 솔루션

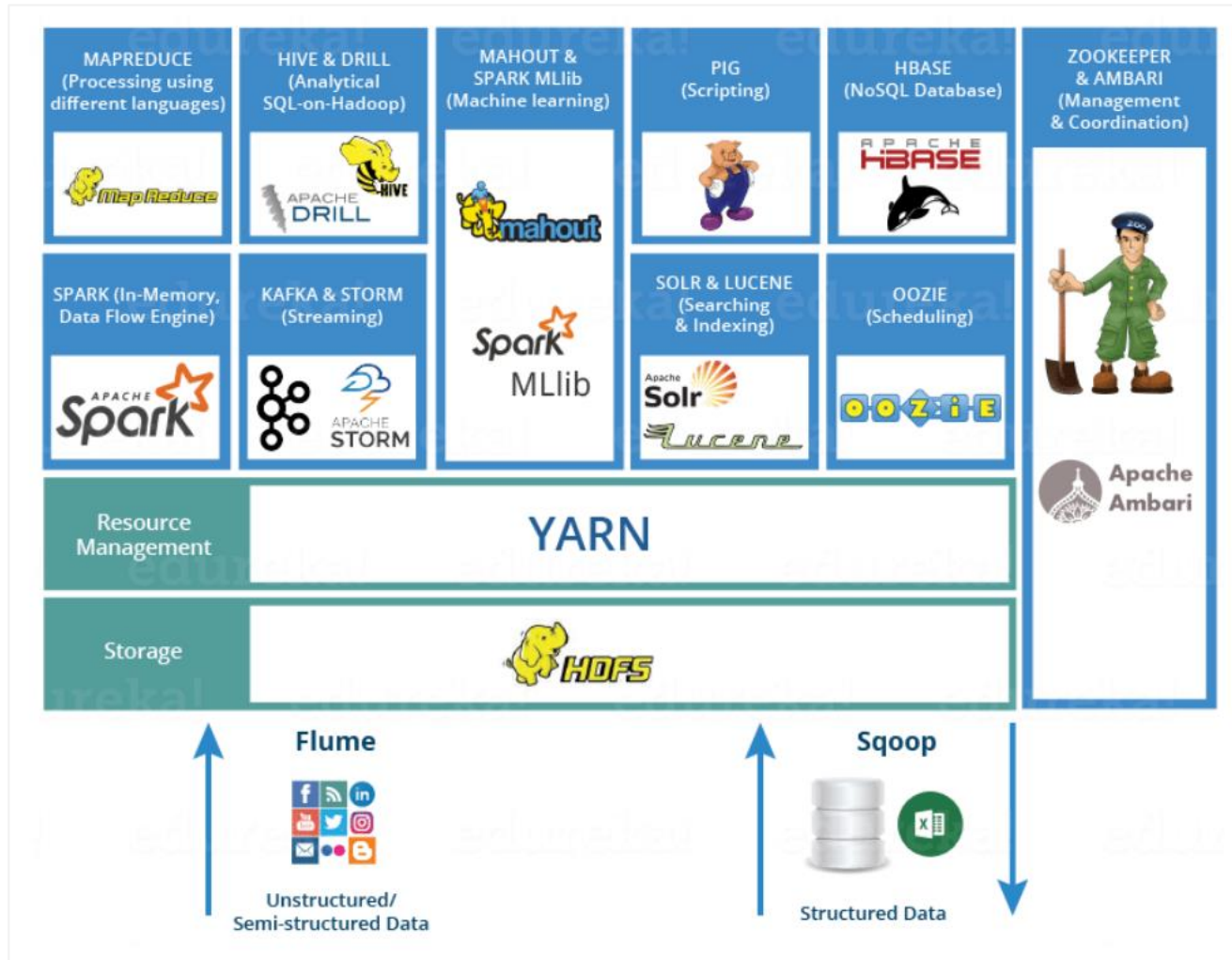
- Apache 재단



1. 1999년에 구성된 비영리 재단
2. 빅데이터 관련 오픈소스 프로젝트를 수행하여 무료로 공유
3. Hadoop 및 Ecosystem 개발 및 공유

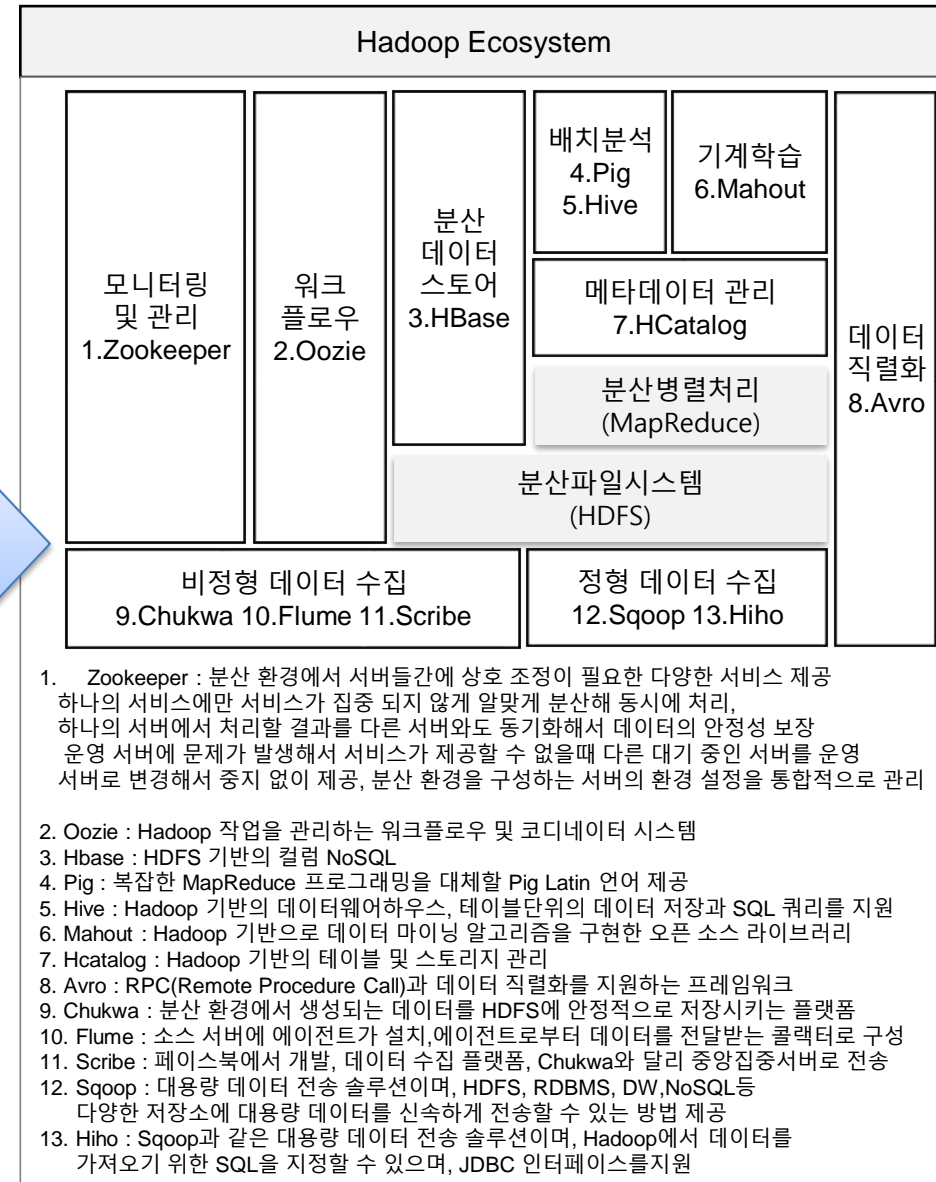
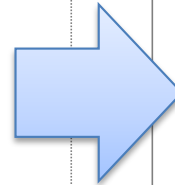
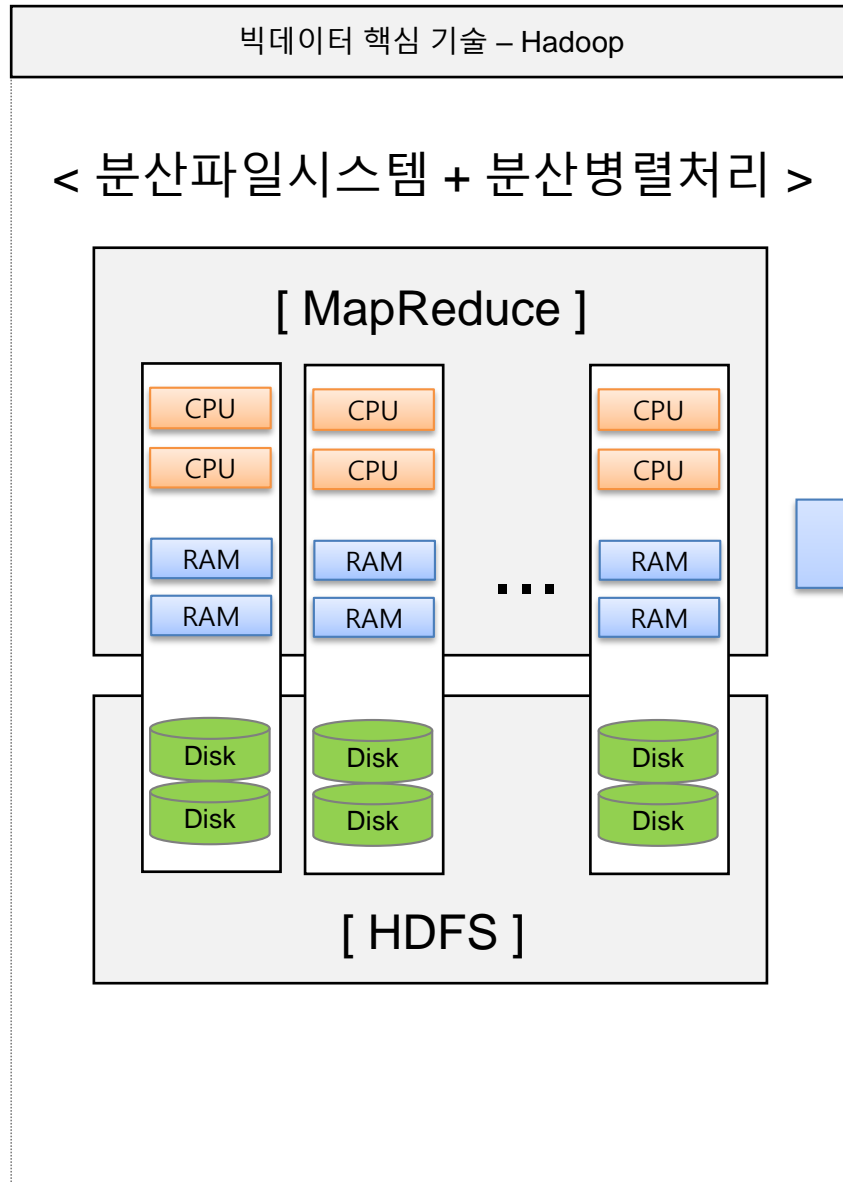
# 빅데이터 처리 솔루션

- Apache 재단
  - Apache Hadoop Ecosystem



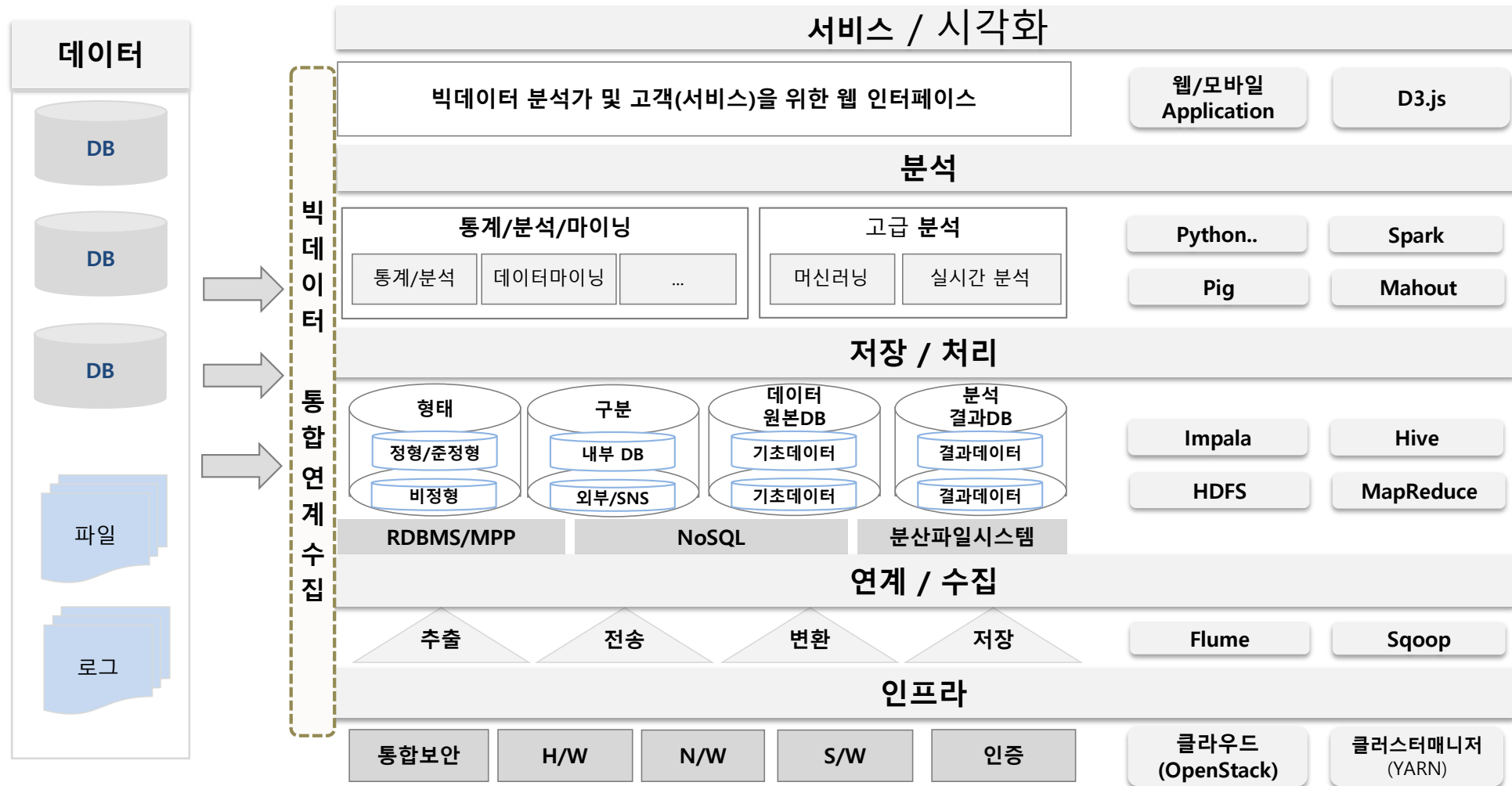


# 빅데이터 처리 솔루션



# 빅데이터 처리 솔루션

- 빅데이터 Architecture



# | Hadoop

# Googole의 시사점

- 일일 데이터 처리량이 기하급수적으로 증가
- 빠른 데이터 처리에 비해 느린 I/O(Read/Write) 작업
- 대용량 데이터 처리를 위한 컴퓨팅이 이전과는 근본적으로 다른 접근 방법이 필요
- 해결책
  - 데이터 중심의 병렬 reads 시스템 필요
- 빅데이터 저장 및 처리

대용량 데이터는 어디에 저장?

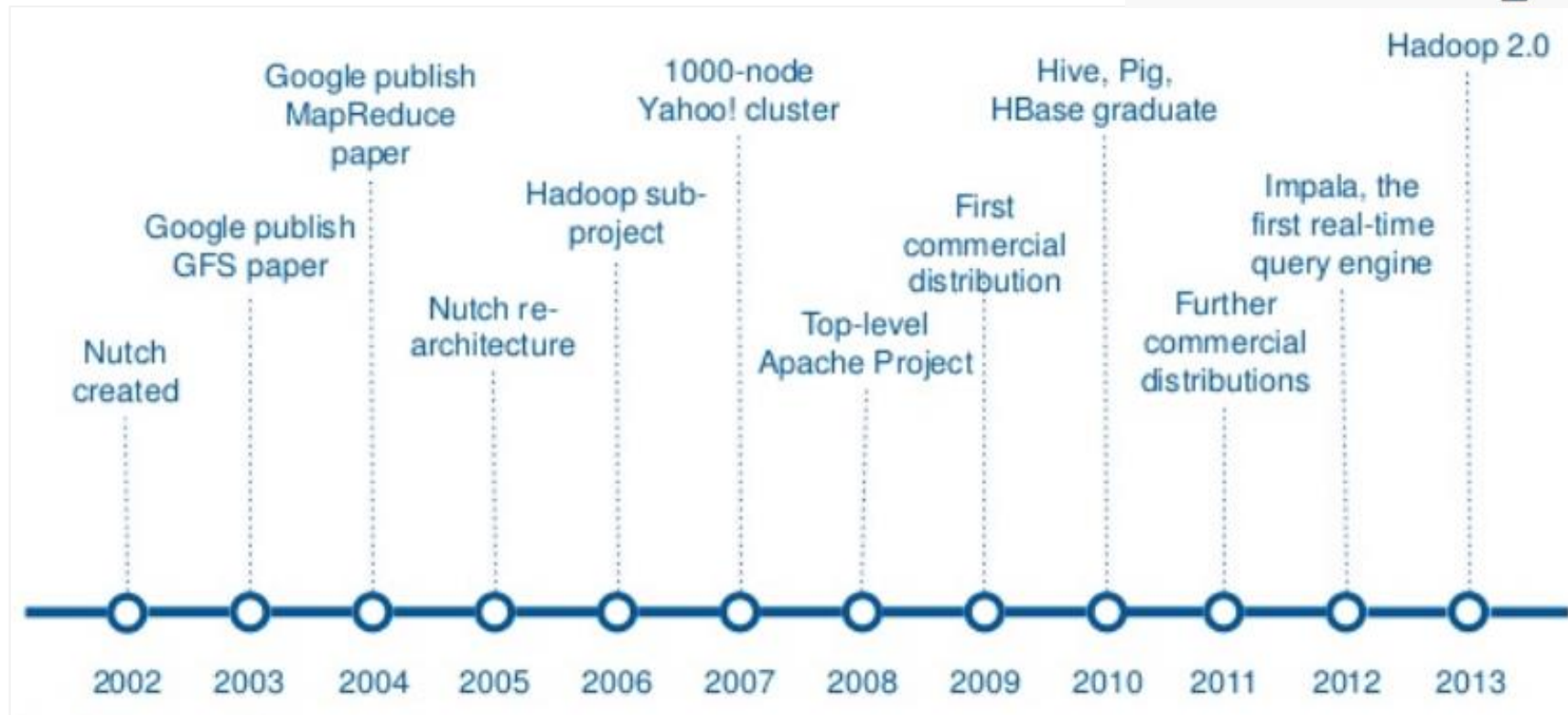
HDFS

대용량 데이터는 어떻게 처리?

MapReduce

# Hadoop의 역사

- A Brief History of Hadoop



# Hadoop 개요

---

- 빅데이터 처리용 오픈 SW 프레임워크
  - 성능 좋은 하나의 컴퓨터 대신 적당한 성능의 범용 컴퓨터 여러 대를 클러스터화 하고, 큰 그기의 데이터를 클러스터에 병렬로 동시에 처리하여 처리 속도를 향상시키는 것이 주 목적
  - 분산 처리를 위한 오픈 소스 프레임워크



# Hadoop 개요

- HDFS & MapReduce로 구성
  - 파일 시스템
  - 파일을 블록 단위로 쪼개 여러 서버에 분산 · 저장
  - 분산 병렬 처리
  - 다양한 Hadoop 에코 시스템
- 특징
  - 결함 허용
    - 결함이 발생해도 작업 중지 없이 실행됨을 의미
    - 결함이 생겨도 문제 없이 실행을 유지하기 위해 데이터 백업이 중요 [즉 복제본 중요]
  - 데이터 블록의 복사본을 중복 저장하고 관리 및 유지
  - 하나의 클러스터 구성시 수백~수천 대 이상의 서버를 이용
    - 페타바이트급 이상의 데이터도 손쉽게 저장할 수 있음



# Hadoop의 장단점

---

- 장점
  - 오픈소스로 라이선스에 대한 비용 부담이 적음
  - 시스템을 중단하지 않고, 장비의 추가가 용이(Scale Out)
  - 일부 장비에 장애가 발생하더라도 전체 시스템 사용성에 영향이 적음(Fault tolerance)
  - 저렴한 구축 비용과 비용대비 빠른 데이터 처리
  - 오프라인 배치 프로세싱에 최적화
- 단점
  - HDFS에 저장된 데이터를 변경 불가
  - 실시간 데이터 분석 같이 신속하게 처리해야 하는 작업에는 부적합
  - 너무 많은 버전과 부실한 서포트
  - 설정의 어려움



# Hadoop의 데이터 처리 방식과 구성 요소

- Hadoop의 데이터 처리 방식

## 데이터 블록 전송 단계

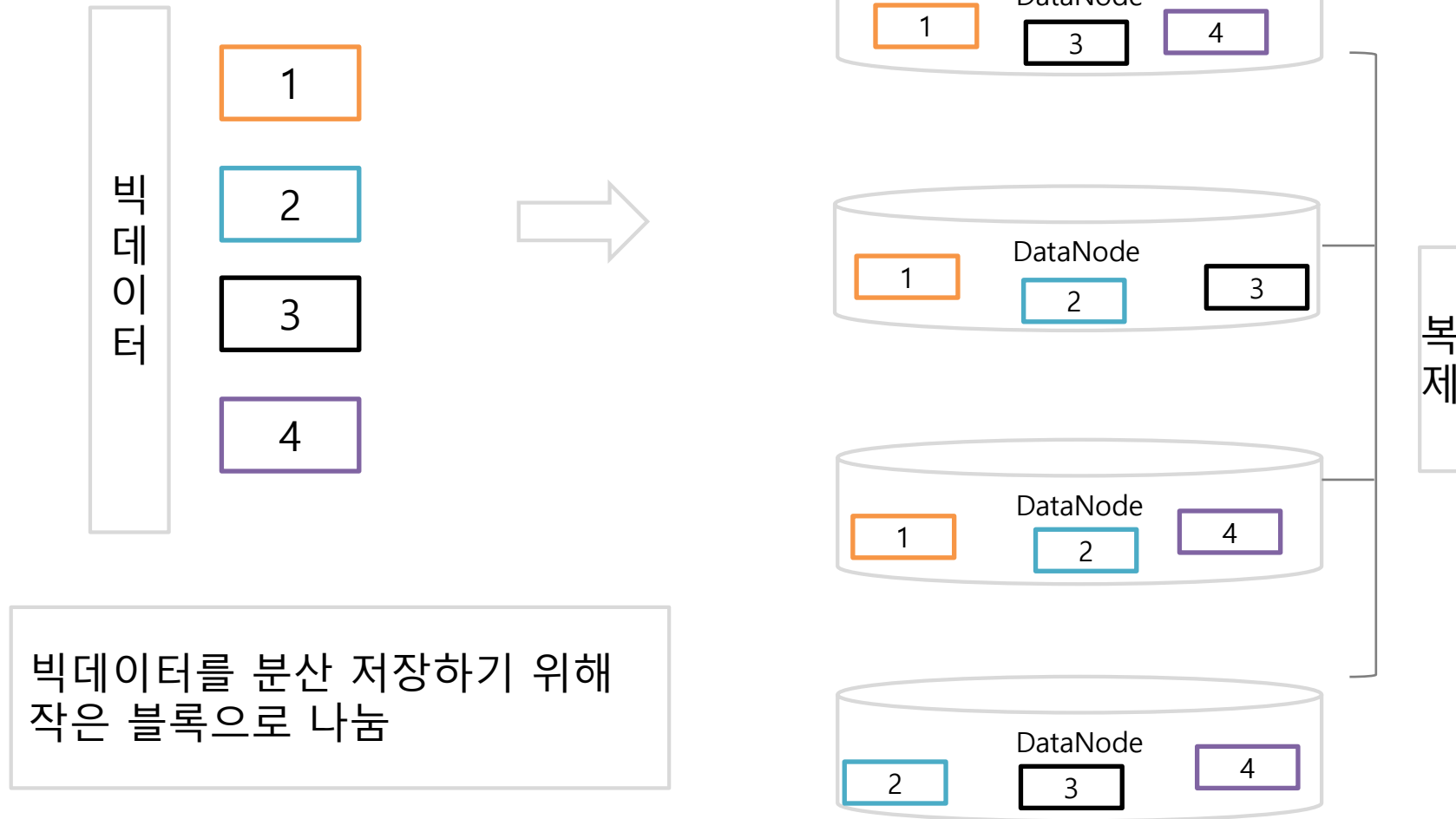
- 하나의 파일을 여러 블록으로 나누어 클러스터에 있는 데이터 노드들에게 분산 저장

## 데이터 블록 복제 단계

- 하나의 블록은 여러개(3)의 복제본을 생성하여 분산 저장

# Hadoop의 데이터 처리 방식과 구성 요소

- Hadoop의 데이터 처리 방식



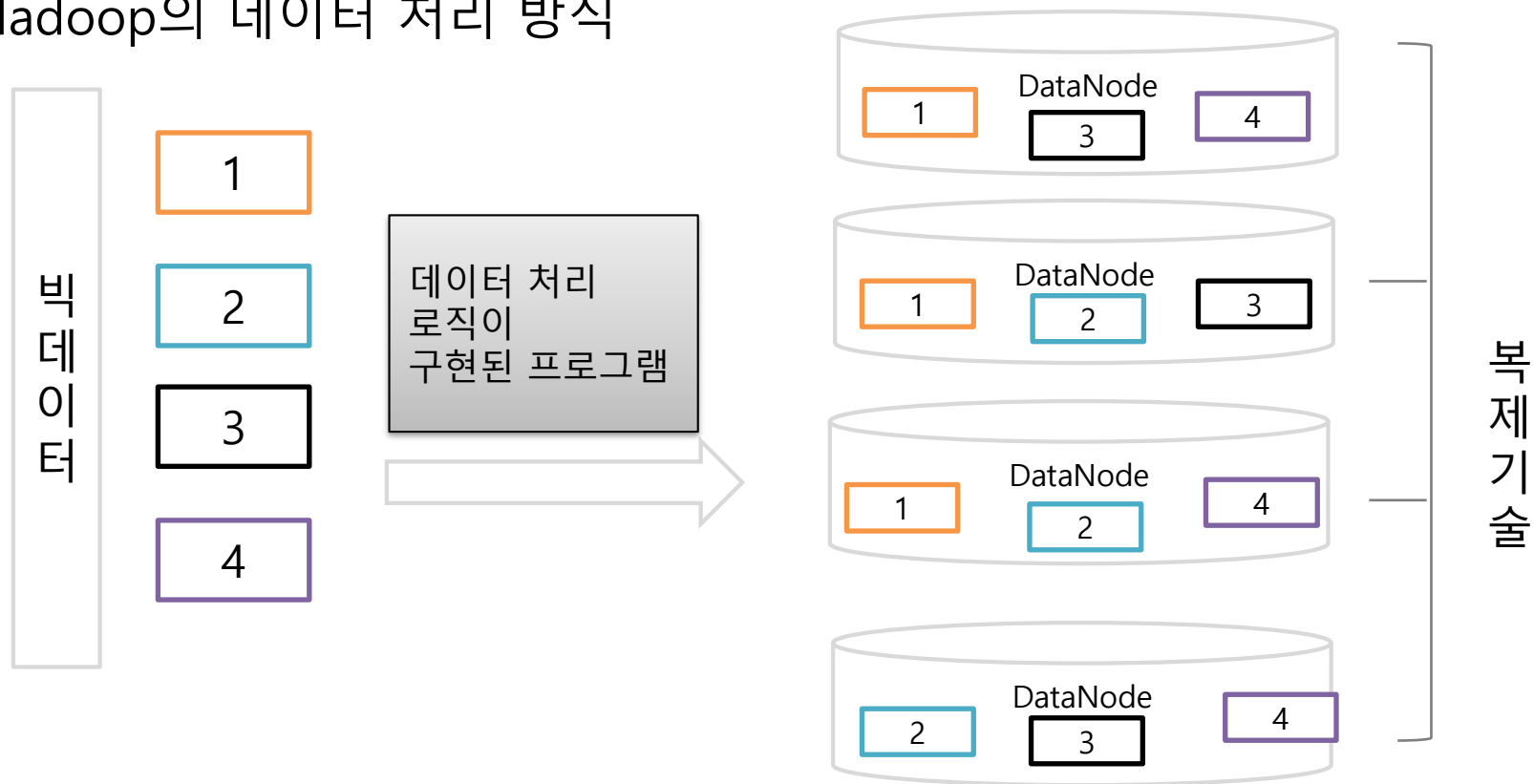
# Hadoop의 데이터 처리 방식과 구성 요소

---

- 데이터 전송 후 데이터를 어떻게 처리 할 것인가?
  - 프로그램 코드 전송 단계
    - 패키지 된 프로그램 코드를 해당 노드들에게 전달
  - 데이터 병렬 처리 단계
    - 데이터를 병렬 처리

# Hadoop의 데이터 처리 방식과 구성 요소

- Hadoop의 데이터 처리 방식



## DataNode 동작 원리

- 데이터 저장 후 데이터 처리 로직이 구현된 프로그램을 받음
- 이 프로그램 로직을 기반으로 데이터 처리 후 데이터를 저장

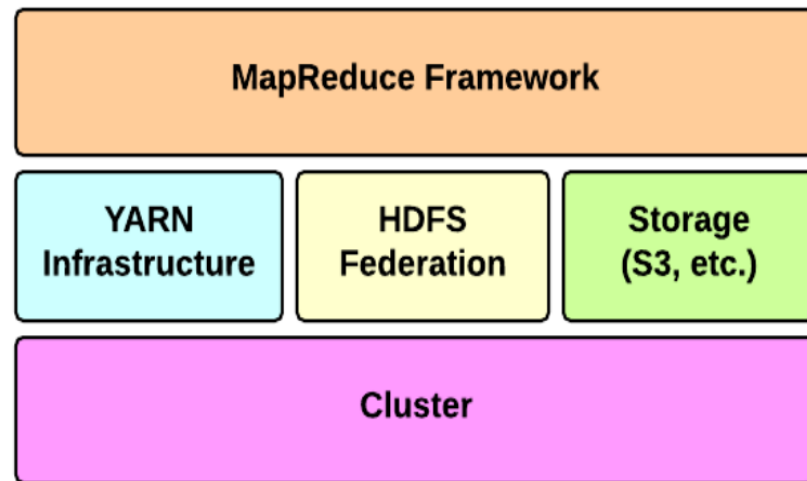
# Hadoop의 Architecture

- Hadoop의 Runtime 내부 구조

- cluster
  - host system
  - node는 랙으로 분할 될 수 있음
  - infra의 하드웨어 부분
- Yarn Infrastructure
  - 응용 프로그램의 실행에 필요한 전산 자원(cpu, 메모리)를 제공할 책임이 있는 framework
- HDFS Federation
  - Name Node는 파일 정보 메타데이터를 메모리에서 관리
  - 파일이 많아지면 메모리 사용량 증가 및 메모리 관리 문제 발생
  - 해결책 : Hadoop2부터 HDFS Federation 지원
  - 기능 : 네임스페이스(디렉토리) 단위로 Name Node를 등록하여 사용

dir1, dir2, dir3 개의 디렉토리가 존재할 경우 각 디렉토리 단위 별로 총 3개의 name node 가 실행 되고 파일을 관리 하게 되는 구조  
파일, 디렉토리의 정보를 가지는 네임스페이스와 블록의 정보를 가지는 블록 풀을 각 네임노드가 독립적으로 관리  
네임스페이스와 블록풀을 네임스페이스 볼륨이라하고 네임스페이스 볼륨은 독립적으로 관리되기 때문에  
하나의 네임노드에 문제가 생겨도 다른 네임노드에 영향을 주지 않음

- Storage



# Hadoop의 코어 구성 요소

---

HDFS

MapReduce

# Hadoop의 코어 구성 요소

## HDFS

여러 Hadoop 클러스터 노드에 분산 방식으로 데이터를 저장하는 역할

대규모 분산 파일 시스템 구축의 성능과 안전성 제공

## MapReduce

분산 데이터 처리를 분할 정복 전략으로 나눔

풍부한 계산 API를 개발자에게 제공  
API로 구현한 코드는 Hadoop 클러스터에 맵리듀스 테스크를 실행

HDFS에 저장된 대규모 분산 파일에 대한 로그분석, 색인 구축, 검색에 탁월한 능력 발휘

---

 Yarn



# Hadoop의 코어 구성 요소

---

- Hadoop Yarn 주요 기능
  - Hadoop 2버전 부터 지원하는 자원 관리 기능
  - 클러스터에 있는 컴퓨팅 자원들(CPU, 메모리)등을 동적으로 관리하는 플랫폼 서비스
  - 작업을 함께 수행하면서 클러스터 자원의 사용을 증대
  - 여러 개의 프레임워크(맵리듀스, 스파크)를 수행하는 동안 동적으로 CPU와 메모리 자원을 공유할 수 있도록 함
  - 맵리듀스의 경우 대규모 테이블 데이터를 검색하여 많은 디스크 관리 I/O를 사용하여 적은 메모리를 사용

# Hadoop의 코어 구성 요소

- Hadoop Yarn 두 가지 데몬 실행

## Resource Manager

다양한 응용 프로그램에  
자원을 할당

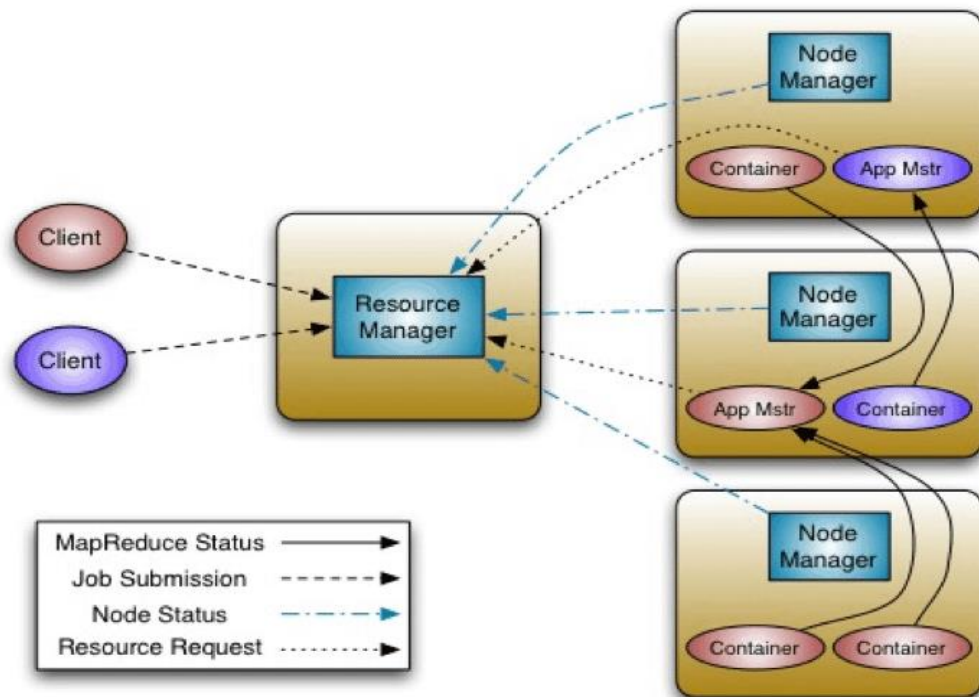
자원 스케줄러

## Application Master

프로세스의 실행을 모니터링  
자원 관리자에게 자원을 요청

# Hadoop의 코어 구성 요소 : Yarn

- Hadoop Yarn 계층



YARN(Yet Another Resource Negotiator)

- Hadoop 2에 도입한 클러스터 리소스 관리 및 애플리케이션 라이프 사이클 관리를 위한 architecture

- Hadoop 1에서의 병목 현상 발생의 단점 보완

- 구성

- Resource Manager : 자원관리
- Node Manager : 자원 관리
- Application Master : 애플리케이션 라이프 사이클 관리
- Container : 애플리케이션 라이프 사이클 관리

# Hadoop의 코어 구성 요소 : Yarn

- 자원 관리
  - ResourceManager와 NodeManager를 이용하여 처리

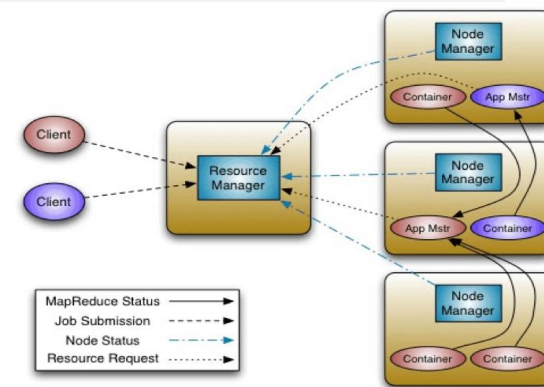
## ResourceManager

Node Manager로 부터 전달받은 정보를 이용하여  
클러스터 전체의 자원 관리(자원 스케줄러)  
자원 사용 상태를 모니터링  
Application Master에서 자원 요청시 해당 자원을  
사용할 수 있도록 처리

자원 분배 규칙 설정은 Scheduler의 설정된  
규칙에 따라 효율적인 분배

## NodeManager

클러스터 각 노드마다 실행  
주기적으로 자원 관리자에게 heartbeat 전송  
현 노드의 자원 상태를 관리하고,  
Resource Manager에게 현재 자원 상태를 보고



# Hadoop의 코어 구성 요소 : Yarn

- 자원 관리
  - ResourceManager과 NodeManager를 이용하여 처리

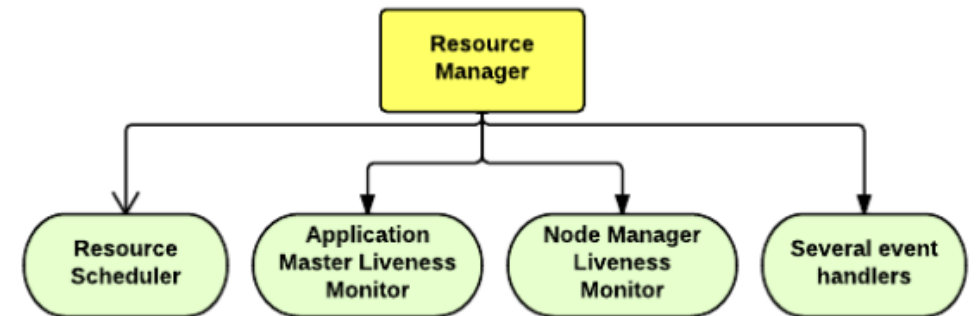
ResourceManager

Node Manager로 부터 전달받은 정보를 이용하여 클러스터 전체의 자원 관리(자원 스케줄러)

자원 사용 상태를 모니터링

Application Master에서 자원 요청시 해당 자원을 사용할 수 있도록 처리

자원 분배 규칙 설정은 Scheduler의 설정된 규칙에 따라 효율적인 분배

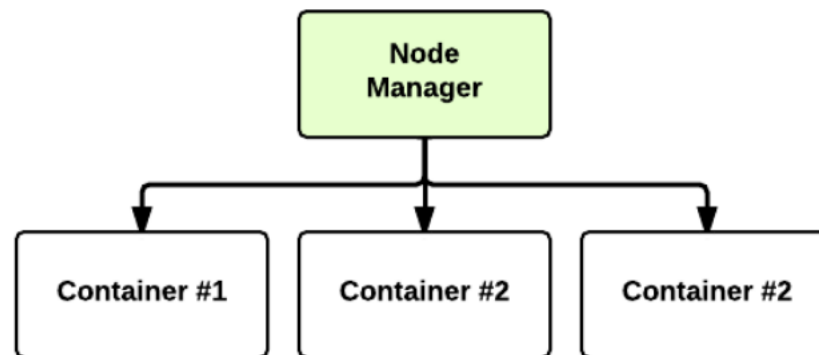


# Hadoop의 코어 구성 요소 : Yarn

- 자원 관리
  - ResourceManager와 NodeManager를 이용하여 처리

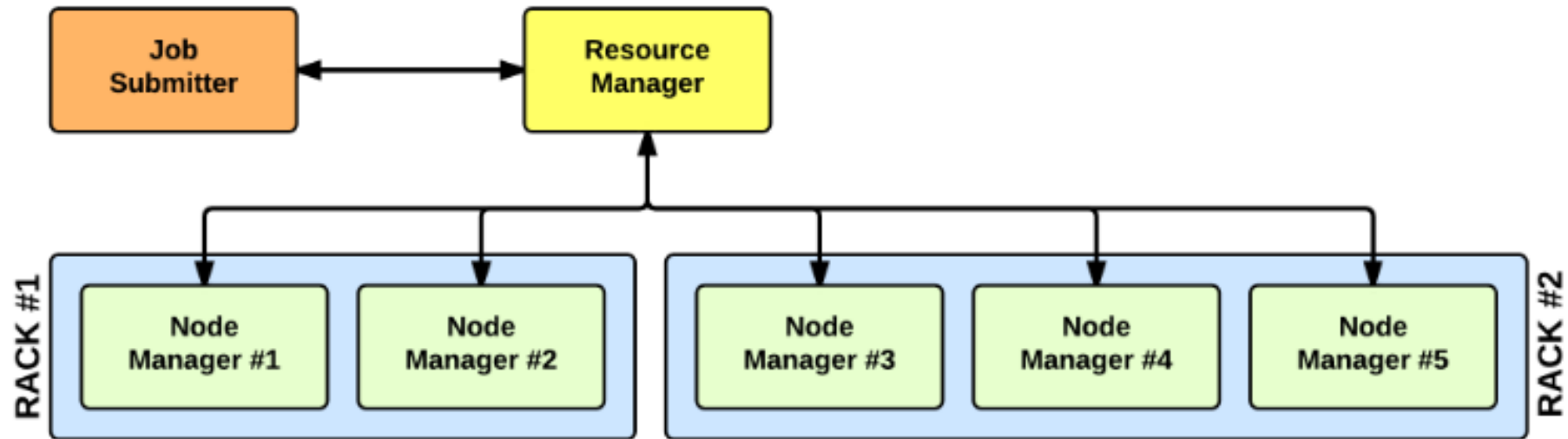
NodeManager

클러스터 각 노드마다 실행  
주기적으로 자원 관리자에게 heartbeat 전송  
현 노드의 자원 상태를 관리하고,  
Resource Manager에게 현재 자원 상태를 보고



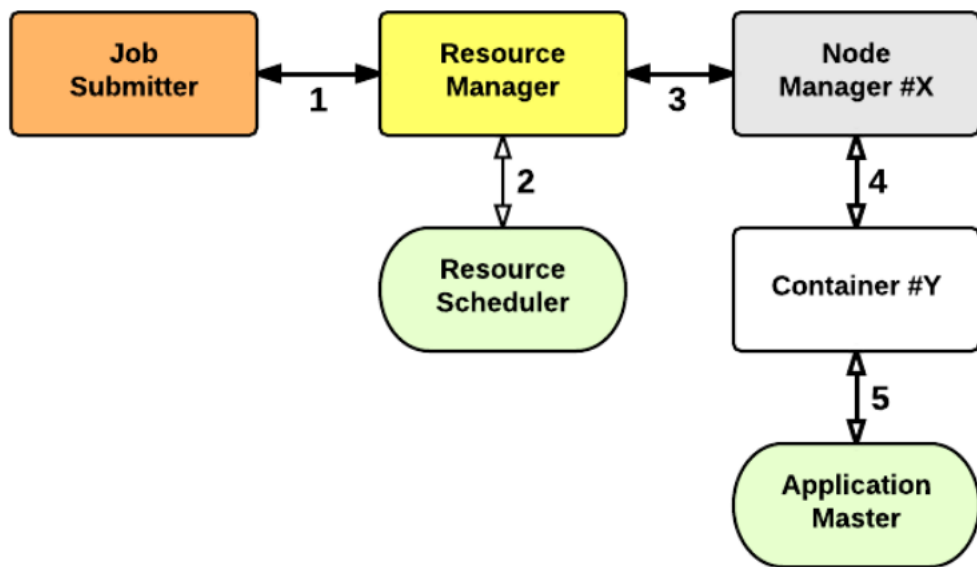
# Hadoop의 코어 구성 요소 : Yarn

- YARN: Application Startup



# Hadoop의 코어 구성 요소 : Yarn

- YARN: Application Startup
  - 응용 프로그램 시작 Process



1단계 : 클라이언트가 응용 프로그램을 Resource Manager에 제출

2단계 : 자원 관리자가 컨테이너를 할당

3단계 : 자원 관리자가 관련 노드 관리자에 접속

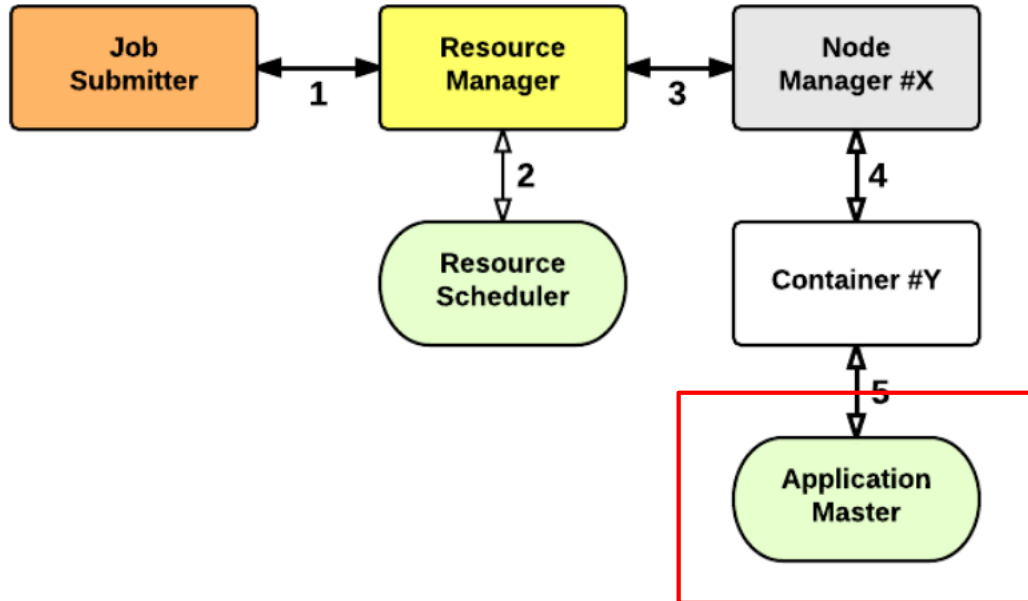
4단계 : 노드 관리자가 컨테이너를 시작

5단계 : 컨테이너는 응용 프로그램 마스터를 실행



# Hadoop의 코어 구성 요소 : Yarn

- YARN: Application Startup
  - 응용 프로그램 시작 Process



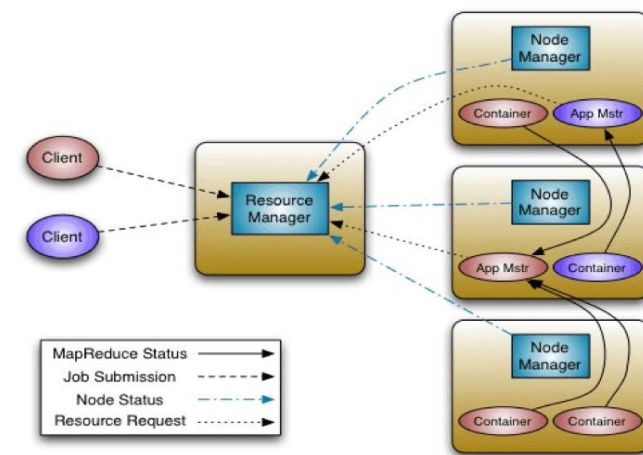
## Application Master

단일 응용 프로그램의 실행을 담당

Resource Scheduler (Resource Manager)에서 컨테이너를 요청하고  
획득한 컨테이너에서 특정 프로그램 (예 : Java 클래스의 메인)을 실행

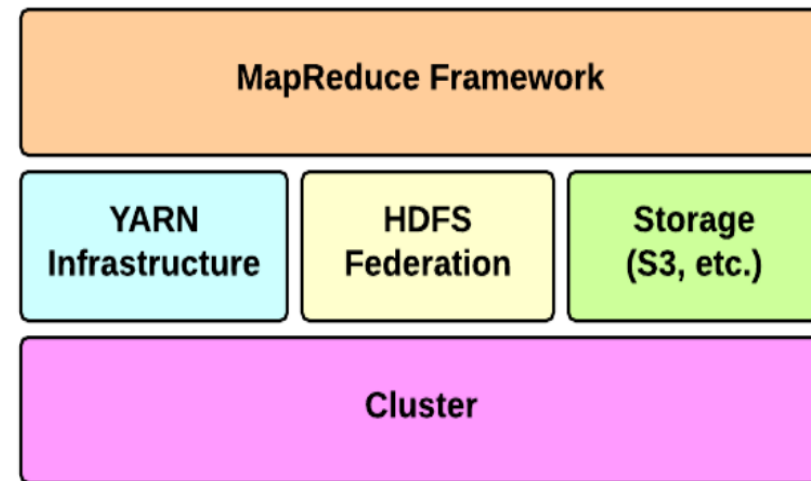
# Hadoop의 코어 구성 요소 : review

- Hadoop Yarn 자원 관리자[Resource Manager]
  - 클러스터에 1개 존재
  - 각 애플리케이션 시작을 초기화
  - 작업 노드에 존재하는 자원들을 어떻게 할당할 것인가 결정
    - 작업 노드에 있는 노드 관리자로 부터 주기적으로 정보를 받음
  - 각 작업에 대한 애플리케이션 마스터에 대한 컨테이너를 생성
    - 애플리케이션 마스터의 상태를 주기적으로 관찰
  - 컨테이너는 작업 노드 메모리와 CPU의 쌍으로 구성
    - 각 작업의 테스크들이 컨테이너를 할당 받아 작업을 수행
  - Yarn 애플리케이션은 한 개 이상의 컨테이너에서 수행
  - 애플리케이션 마스터는 하나의 Yarn 애플리케이션 마다 생성
    - 자원 관리자에게 필요한 컨테이너를 요청해서 각 작업 노드에서 해당 테스크들을 수행



# Hadoop의 Architecture

- Hadoop의 Runtime 내부 구조
  - cluster
    - host system
    - node는 랙으로 분할 될 수 있음
    - infra의 하드웨어 부분
  - Yarn Infrastructure
    - 응용 프로그램의 실행에 필요한 전산 자원(cpu, 메모리)를 제공할 책임이 있는 framework
  - HDFS Federation
    - 네임스페이스(디렉토리) 단위로 Name Node를 등록하여 사용
  - Storage



# | HDFS

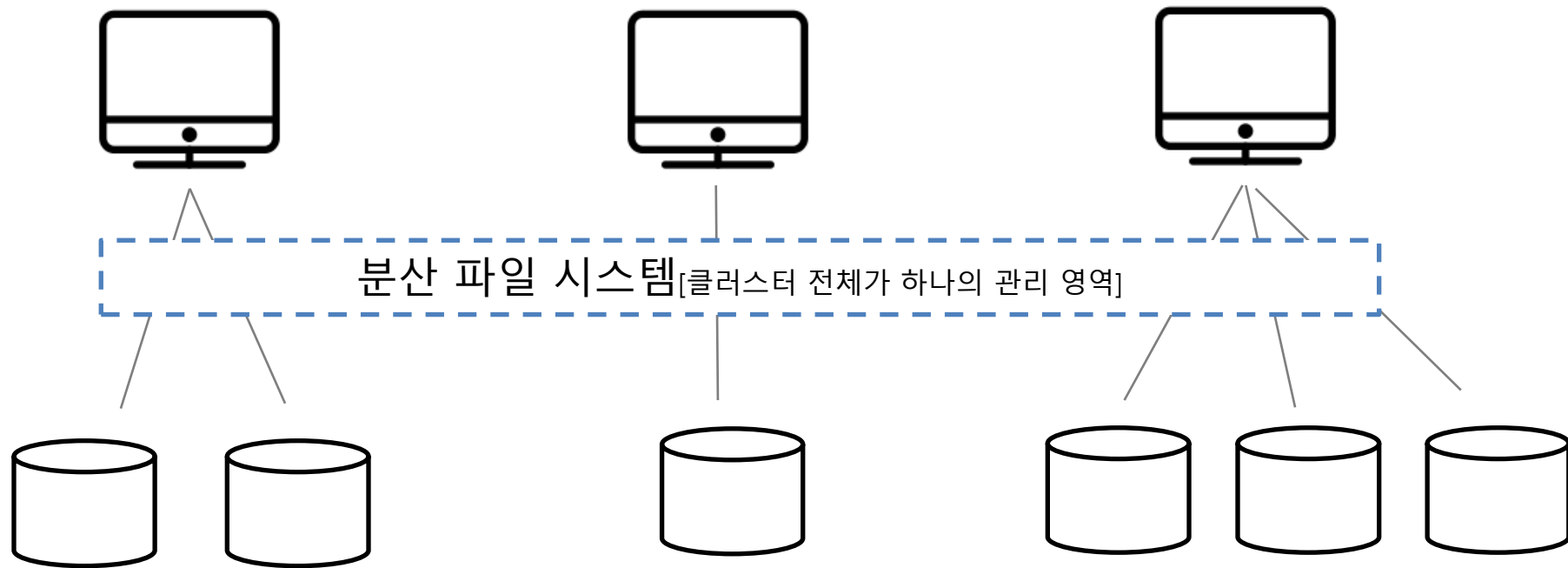
# Hadoop 분산 파일 시스템[HDFS] 아키텍처 목표

---

- HDFS가 효과적인 분산 파일 시스템이 되기 위한 목표
  - 큰 데이터 덩어리를 다룸
  - 고가용성을 가지로 하드웨어 실패 문제를 끊임 없이 해결
  - 데이터를 스트리밍(Streaming) 방식으로 접근
  - 하드웨어 추가로 성능을 향상하는 확장성
  - 장애가 발생해도 데이터를 잃지 않는 견고성
  - 여러 형식의 소프트웨어 또는 하드웨어를 이용할 수 있는 이식성
  - 클러스터 내의 여러 노드에 데이터 분할(Partitioning)

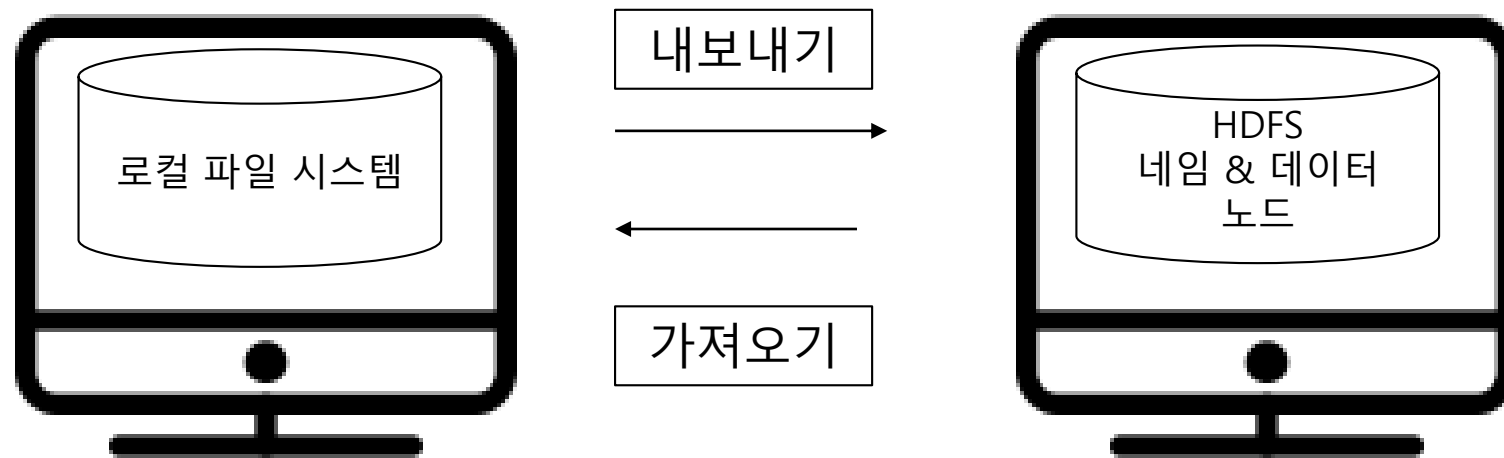
# Hadoop 분산 파일 시스템[HDFS]

- 수십 테라 또는 페타바이트 이상의 대용량 파일을 분산된 서버에 저장하고 수많은 클라이언트가 저장된 데이터를 빠르게 처리할 수 있게 설계된 파일 시스템
- Hadoop 클러스터 노드에 데이터를 저장하는 분산 파일 시스템



# Hadoop 분산 파일 시스템[HDFS]

- 대용량 파일 읽기 & 쓰기 작업에 최적화된 파일 시스템
- 로컬 파일 시스템과 HDFS로 분리되어 있음
- 사용자는 필요한 파일을 로컬 시스템에서 HDFS간의 데이터 전송이 가능



# Hadoop 분산 파일 시스템[HDFS]

---

- 내보내기(쓰기 과정)
  - HDFS에 파일을 저장할 경우
  - 파일의 데이터를 슬라이스로 분할
  - 슬라이스로 분할된 것을 블록이라 고 함
  - 여러 개의 데이터 노드에 복제본을 만들면서 저장
- 가져오기(읽기 과정)
  - HDFS에서 로컬 파일 시스템으로 가져올 경우
  - 분한 슬라이스(블록)을 다시 합쳐서 하나의 블록으로 구성
  - 하나의 파일로 만들 후 로컬 파일 시스템에 저장



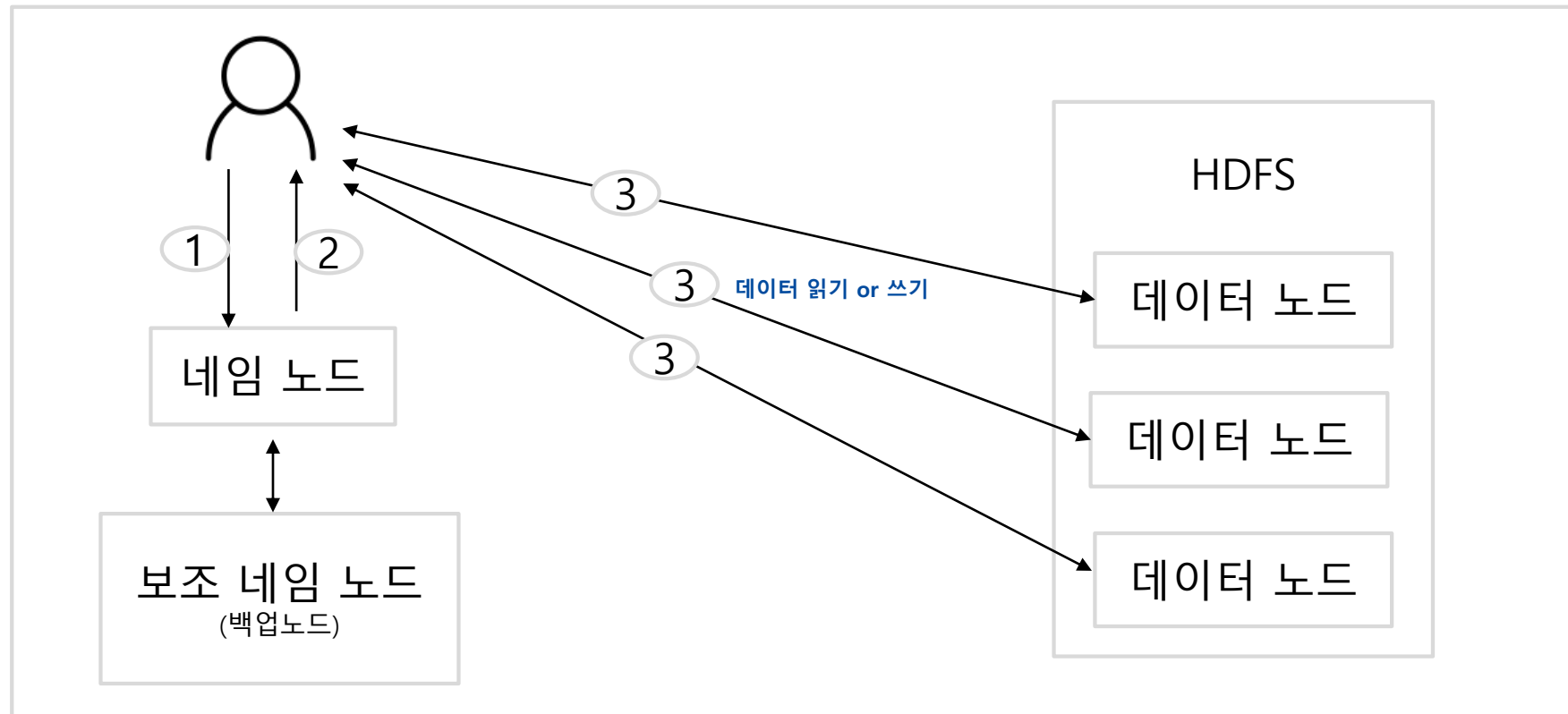
# Hadoop 분산 파일 시스템[HDFS]

---

- 파일 저장 형태
  - 블록 구조의 파일 시스템
  - 64MB or 128MB 블록 단위로 나누어 여러 개의 데이터 노드에 분산 저장
    - 블록이 큰 사유 : 탐색 비용을 최소화 할 수 있음
  - 복제본(replication)의 수는 시스템에서 설정할 수 있으니 기본은 3개
  - 파일의 메타정보는 마스터 노드의 네임노드가 관리
  - 실제 블록 데이터는 작업 노드인 데이터 노드에 분산 저장됨

# Hadoop 분산 파일 시스템[HDFS]

- 분산 시스템을 관리하기 위한 필요 정보
  - 디스크에 분산되어 저장된 각각의 데이터들의 위치 정보가 필요
- 데이터 읽기 / 쓰기 과정



# Hadoop 분산 파일 시스템[HDFS] 구성

종류	특징
네임노드	<p>Hadoop을 이루는 가장 기본적이고 필수적인 일종의 데몬 소프트웨어</p> <p>터 노드와 함께 파일을 분산 저장하고, HDFS 마스터/슬레이브 구조에서 마스터노드의 역할을 함</p> <p>파일시스템의 디렉터리, 파일명, 파일블록 등 네임 스페이스를 관리하는 일종의 마스터 역할을 하며, 슬레이브에 해당하는 데이터 노드에게 입출력에 관련된 작업을 지시</p> <p>즉 파일이 어떤 형태의 블록 단위로 나누어져 있는지, 어느 노드에 필요한 블록이 있는지, 시스템 전반에 대한 상태 등을 모니터링</p> <p>일종의 메타 정보 또는 인덱스 정보를 갖고 있는 것으로 보아도 무방</p> <p>주로 시스템 메모리와 기타 입출력에 관한 지시</p>
데이터 노드	<p>파일에 대한 실질적인 데이터 입출력에 대한 처리를 수행</p> <p>다양한 소스에서 발생하는 각종 데이터를 실시간 또는 주기적으로 파일 시스템에 저장할 경우 먼저 이를 지정된 블록 단위로 쪼개고 데이터 노드에 분산 · 저장 · 처리</p> <p>네임 노드는 HDFS상의 각 블록이 어느 데이터 노드에 존재를 하는지를 사용자에게 알려 줌</p> <p>동일한 파일을 여러 개의 블록으로 나누어 저장하게 되는데 통상적으로 3벌로 복제</p> <p>(데이터의 유실을 방지하기 위한 대책)</p>
보조 네임 노드	<p>HDFS의 상태를 모니터링하는 보조 기능을 하는 일종의 데몬 소프트웨어</p> <p>주기적으로 네임 노드의 파일 시스템 이미지를 스냅샷해 생성하는 기능을 수행</p> <p>네임 노드 복구 시 데이터의 손실에 따른 장애를 최소화하는 기능을 하기 때문에 보조 네임 노드를 체크 포인팅 서버라고도 함</p>

# Hadoop 분산 파일 시스템[HDFS]

- HDFS Architecture

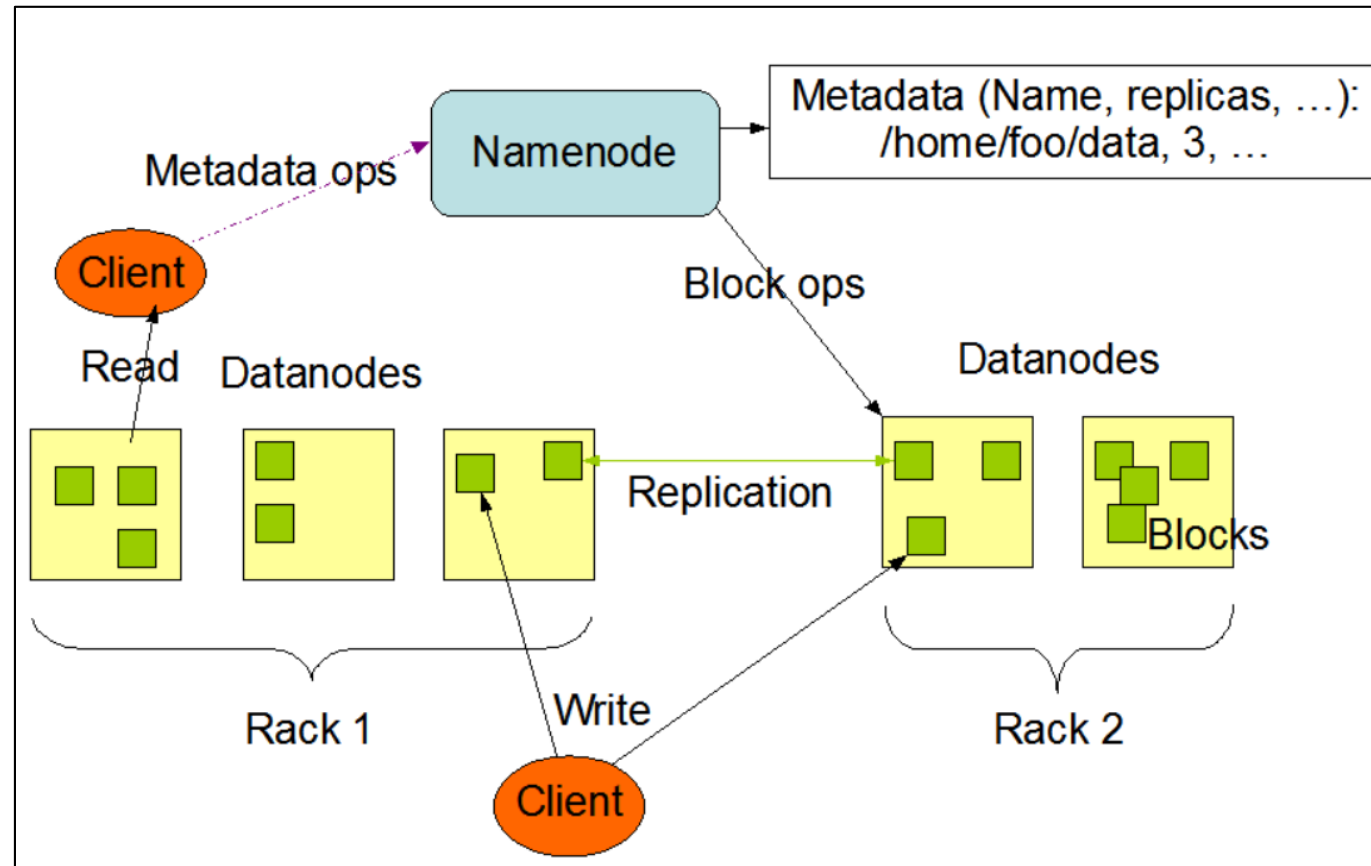


그림 출처 : [https://hadoop.apache.org/docs/r1.2.1/hdfs\\_design.html#NameNode+and+DataNodes](https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#NameNode+and+DataNodes)

# Hadoop 분산 파일 시스템[HDFS]

- 블록 복제

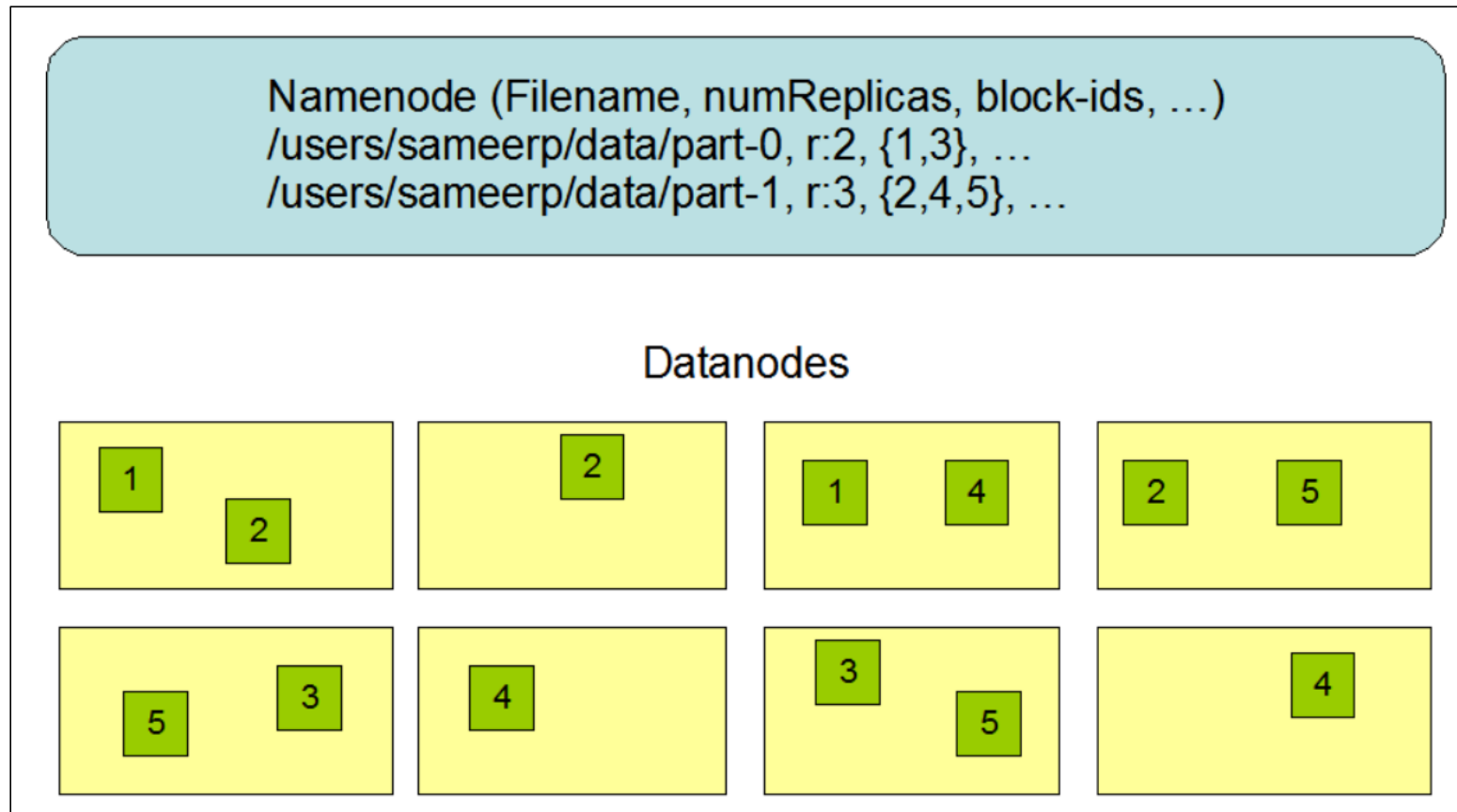


그림 출처 : [https://hadoop.apache.org/docs/r1.2.1/hdfs\\_design.html#NameNode+and+DataNodes](https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#NameNode+and+DataNodes)

# Hadoop 분산 파일 시스템[HDFS]

- HDFS Architecture

- Namenode

- 메타데이터 관리

- 메타 데이터 : 파일 이름, 파일 크기, 파일 생성시간, 파일 접근 권한, 파일 소유자 및 그룹 소유자, 파일이 위치한 블록의 정보 등으로 구성

- 각 데이터 노드에서 전달하는 모든 메타 데이터를 받아서 전체 노드의 메타 데이터 정보와 파일 정보를 묶어서 관리

- 데이터 노드 간 데이터 복제

- 파일이나 디스크 위치 이름을 지정하는 이름 지정 시스템 등의 조율 활동 수행

- 블록이 어느 데이터 노드에 있는지 정보 저장

- 하나의 클러스트에 하나의 활동 네임노드만 있을 수 있음

- 파일 시스템의 접근과 HDFS 파일을 생성, 열기, 수정, 삭제하는 HDFS 기반의 API 사용을 제어

- 네임노드는 멀티 스레드 프로세서, 여러 클라이언트 요청을 한 번에 처리 가능

# Hadoop 분산 파일 시스템[HDFS]

- 메타데이터 파일 종류
  - 데이터 손실 방지를 위한 추가 기능



edits log

HDFS의 모든 변경 이력을 저장

Client가 파일을 저장하거나,  
삭제, 파일을 이동등에 대한 트랜잭션  
로그

주기적으로 생성



fsimages

메모리에 저장된 메타데이터의  
파일 시스템 이미지를 저장한 파일

네임스페이스와 블록 정보

# Hadoop 분산 파일 시스템[HDFS]

- editslog
  - HDFS의 모든 변경 이력을 저장
  - Client가 파일을 저장하거나, 삭제, 파일을 이동하는 경우 editslog와 메모리에 로딩돼 있는 메타데이터에 기록

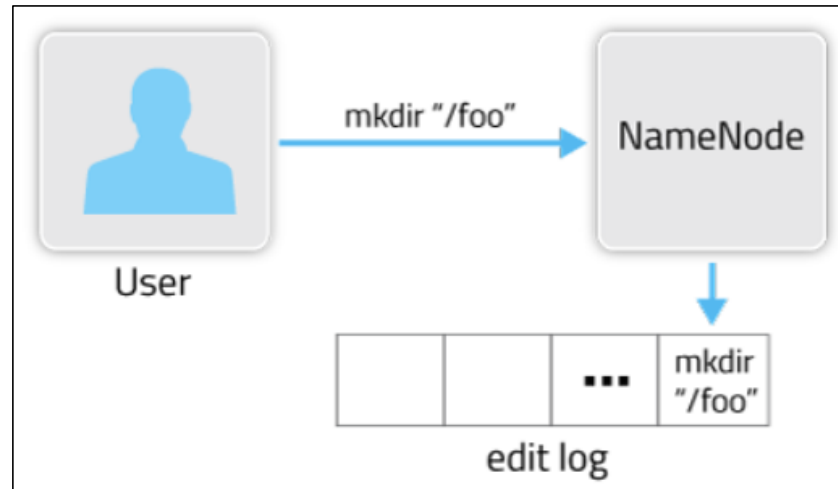


그림 출처 : <http://blog.cloudera.com/blog/2014/03/a-guide-to-checkpointing-in-hadoop>



# Hadoop 분산 파일 시스템[HDFS]

- editlog 와 fsimage의 사용 단계

1단계 : 네임노드가 구동되면 로컬에 저장된 fsmiage와 editslog를 조회

2단계 : 메모리에 fsimage를 로딩해서 파일 시스템 이미지를 생성

3단계 : 메모리에 로딩된 파일 시스템 이미지에 editslog에 기록된 변경 이력을 적용

4단계 : 메모리에 로딩된 파일 시스템 이미지를 이용하여 fsimage 파일을 갱신

5단계 : editslog 초기화

6단계 : 데이터노드가 전송한 블록 리포트를 메모리에 로딩된 파일 시스템 이미지에 적용



edits log



fsimages

# Hadoop 분산 파일 시스템[HDFS]

- Check pointing
  - 기존 fsimage와 editlog를 기반으로 새로운 fsimage를 생성

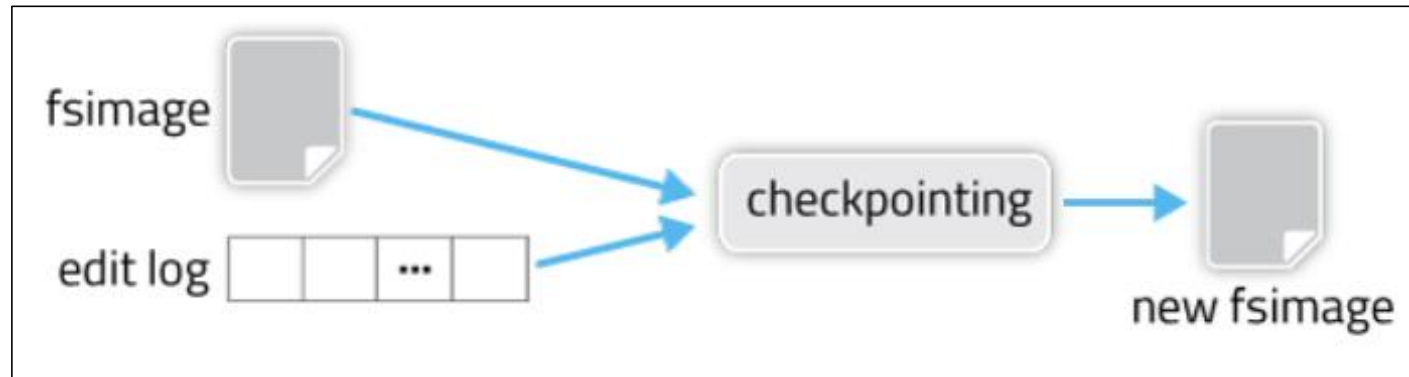


그림 출처 : <http://blog.cloudera.com/blog/2014/03/a-guide-to-checkpointing-in-hadoop>

# Hadoop 분산 파일 시스템[HDFS]

Hadoop				Overview	Datanodes	Datanode Volume Failures	Snapshot	Startup Progress	Utilities ▾
Startup Progress				Elapsed Time: 6 sec, Percent Complete: 100%					
Phase				Completion		Elapsed Time			
Loading fsimage /var/lib/hadoop-hdfs/cache/hdfs/dfs/name/current/fsimage_0000000000000018771 91.83 KB				100%		1 sec			
inodes (1121/1121)				100%					
delegation tokens (0/0)				100%					
cache pools (0/0)				100%					
Loading edits				100%		0 sec			
/var/lib/hadoop-hdfs/cache/hdfs/dfs/name/current/edits_0000000000000018772-0000000000000019509 1 MB (738/738)				100%					
Saving checkpoint				100%		0 sec			
inodes /var/lib/hadoop-hdfs/cache/hdfs/dfs/name/current/fsimage.ckpt_0000000000000019509 (0/0)				100%					
delegation tokens /var/lib/hadoop-hdfs/cache/hdfs/dfs/name/current/fsimage.ckpt_0000000000000019509 (0/0)				100%					
cache pools /var/lib/hadoop-hdfs/cache/hdfs/dfs/name/current/fsimage.ckpt_0000000000000019509 (0/0)				100%					
Safe mode				100%		3 sec			
awaiting reported blocks (975/976)				100%					

# Hadoop 분산 파일 시스템[HDFS]

- editlog 와 fsimage의 사용 단계

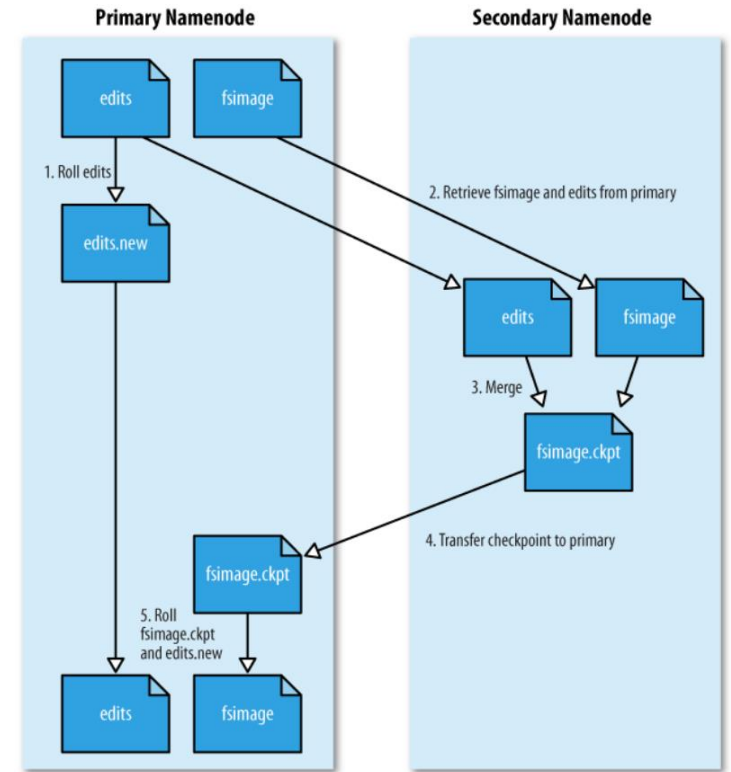
```
[root@quickstart current]# ls
edits_00000000000000000001-000000000000000005340  edits_000000000000000018772-000000000000000019509
edits_000000000000000005341-00000000000000006467  edits_000000000000000019510-000000000000000020594
edits_000000000000000006468-00000000000000008334  edits_000000000000000020595-000000000000000022157
edits_000000000000000008335-00000000000000009469  edits_000000000000000022158-000000000000000023263
edits_000000000000000009470-000000000000000010234  edits_000000000000000023264-000000000000000024369
edits_000000000000000010235-000000000000000011336  edits_inprogress_000000000000000024370
edits_000000000000000011337-000000000000000011771  fsimage_000000000000000023263
edits_000000000000000011772-000000000000000012207  fsimage_000000000000000023263.md5
edits_000000000000000012208-000000000000000013473  fsimage_000000000000000024369
edits_000000000000000013474-000000000000000015488  fsimage_000000000000000024369.md5
edits_000000000000000015489-000000000000000016570  seen_txid
edits_000000000000000016571-000000000000000017666  VERSION
edits_000000000000000017667-000000000000000018771
[root@quickstart current]# pwd
/var/lib/hadoop-hdfs/cache/hdfs/dfs/name/current
```

# Hadoop 분산 파일 시스템[HDFS]

- HDFS Architecture

- 세컨더리 네임노드

- Hadoop은 단일 네임노드로 구동
    - 네임노드가 클러스터의 단일 고장점(Single point of failure)
    - 네임노드에 문제 발생시 해결책
    - 네임 노드의 최신 메모리 스냅샷 복사본 유지
      - 체크포인트(checkpoint) : 메모리 스냅샷 복사본 의미
      - CheckPoint 기능 보유(체크포인팅 서버라고도 함)
      - 주기적으로 name node의 fsimage를 갱신
      - fsimage와 Edits 파일을 주기적으로 머지하여 최적 블록의 상태로 파일 생성
        - » 머지하면서 Edits 파일 삭제 따라서 디스크 부족 문제 해결 가능
- 권장 구조
  - 네임노드의 견고성 확보를 위해 일반적으로 네임노드와 다른 노드에 구동



<http://kbnithesh.blogspot.com/2015/04/secondary-namenode-check-pointing.html>

# Hadoop 분산 파일 시스템[HDFS]

---

- 하트비트
  - 하트비트에 디스크 가용 공간 정보와 데이터 이동, 적재량 등의 정보 있음
  - 네임노드와 데이터노드 간의 핸드셰이킹(handshaking)에 사용
  - 하트 비트 정보를 기초로 네임노드는 다음 블록 저장소의 우선 순위를 정해서 클러스터의 적재 균형을 맞춤
  - 네임노드는 하트비트 응답을 데이터 노드와 그 밖의 데이터 노드들과 통신해서 블록 복제를 관리, 블록 삭제, 블록 리포트 등을 하는데 이용

# Hadoop 분산 파일 시스템[HDFS]

---

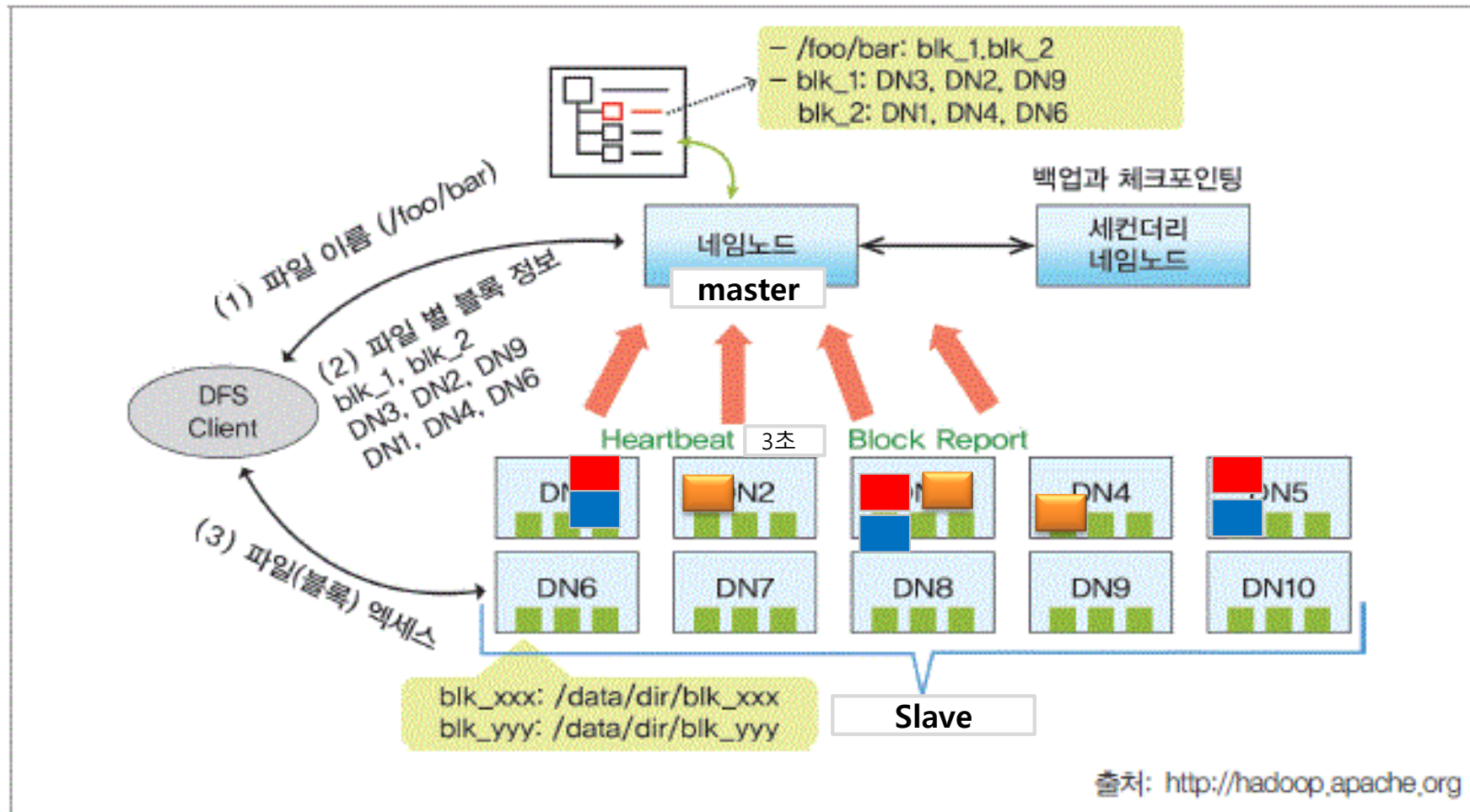
- HDFS Architecture

- DataNode

- 모든 Hadoop 클러스터 노드에 배치하는 슬레이브 역할만 함
    - 애플리케이션의 데이터를 저장하는 기능
    - HDFS에 있는 데이터 파일 각각은 여러 블록으로 쪼개져서 여러 데이터 노드에 저장
    - Hadoop 파일 블록은 데이터 노드에 두개의 파일로 대응
    - 하나의 파일은 블록 데이터, 나머지는 앞의 파일의 체크섬(checksum)

# Hadoop 분산 파일 시스템[HDFS]

- 동작 방식





# Hadoop 분산 파일 시스템[HDFS]

- 동작 방식

- Hadoop 시작
- 데이터 노드 각각은 네임 노드에 접속해 네임 노드에게 자신이 사용 가능함을 알림
- 시스템이 시작되면 네임 노드는 네임스페이스 아이디와 소프트웨어 버전 확인
- 데이터 노드는 데이터 노드 시작할 때 가지고 있던 모든 데이터 블록을 기술하는 블록 리포트를 보냄
- 데이터 노드 구동 중에 각 데이터 노드는 주기적으로 네임노드에 데이터 노드가 이용 가능함을 알리는 하트비트(heartbeat) 신호를 보냄
- 기본 하트비트 주기값은 3초
- 10초 이상 받지 못할 경우 해당 네임노드가 사용할 수 없는 상태로 여김
- 이 경우 기본 설정으로 네임 노드는 해당 데이터 노드의 데이터 블록을 다른 데이터 노드에 복제

# Hadoop 분산 파일 시스템 파일 읽기/쓰기

- 파일 읽기

1단계

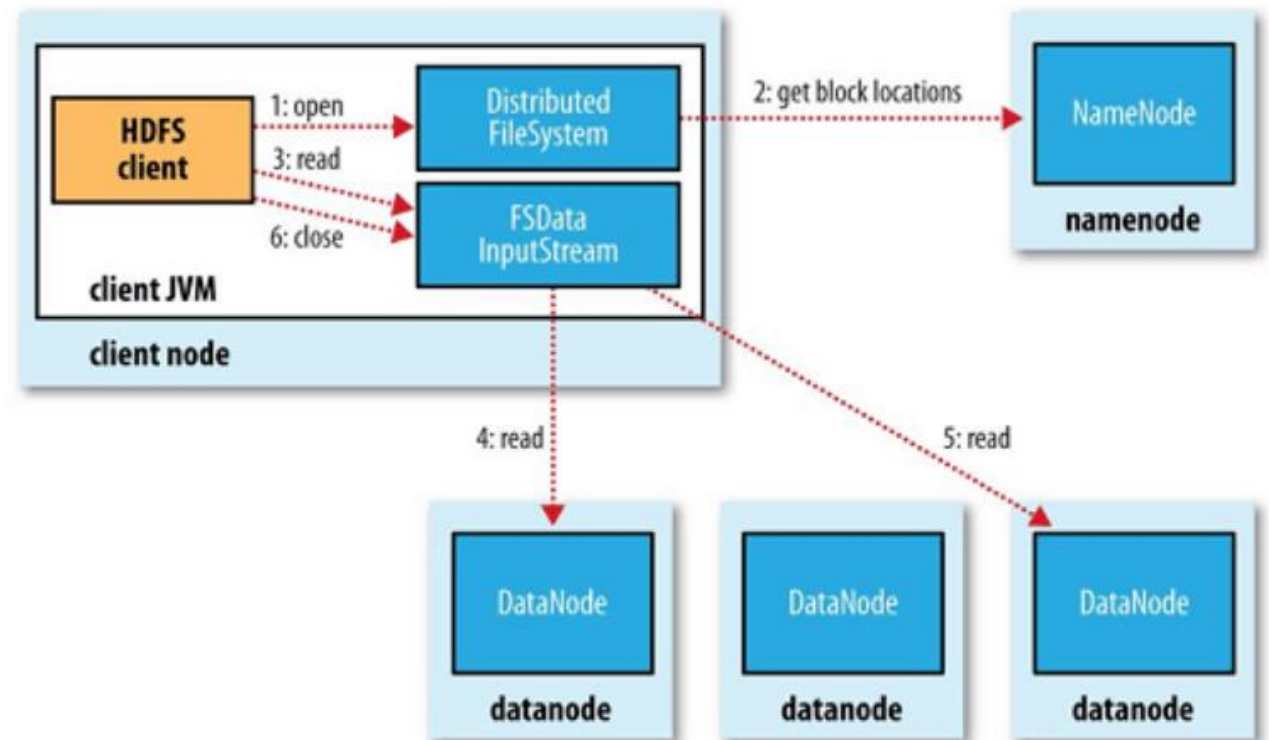
: 네임 노드에 파일이 보관된 블록 위치 요청

2단계

: 네임 노드가 블록 위치 반환

3단계

: 각 데이터 노드에 파일 블록을 요청



# Hadoop 분산 파일 시스템 파일 읽기/쓰기

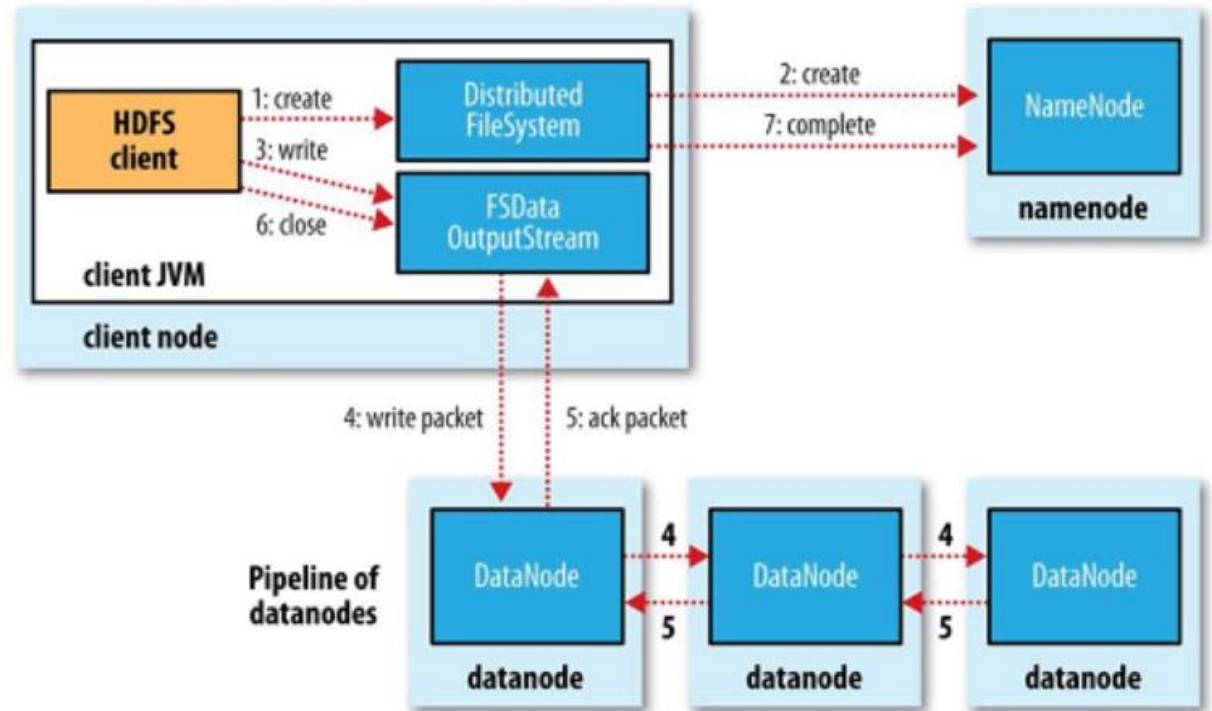
- 파일 쓰기

1단계 : 네임 노드에 파일 정보를 전송하고, 파일의 블록을 서야할 노드 목록을 요청

2단계 : 네임 노드가 파일을 저장할 목록 반환

3단계 : 데이터 노드에 파일 쓰기 요청

- 데이터 노드간 복제가 진행



# | MapReduce

# 맵리듀스 필요성

---

- 대용량 데이터를 빠르고 안전하게 처리하기 위해 만들어짐
- Hadoop 파일 시스템(HDFS)
  - 대규모 분산 파일 시스템 구축의 성능과 안전성 제시
- 맵리듀스
  - HDFS에 저장된 대규모 분산 파일에 대한 로그분석, 색인 구축, 검색에 탁월한 능력 발휘
  - 간단한 단위 작업을 반복하여 처리할 때 사용하는 프로그래밍 모델
  - 간단한 작업 처리하는 **맵(Map)단계** -> 맵 작업의 결과물을 모아서 집계하는 **리듀스(Reduce) 단계**

# 맵리듀스 필요성

---

- 기존 프로그래밍 방식

데이터를 가져온다(Fetch Data)

가져온 데이터를 처리한다(Process Data)

처리한 데이터를 저장한다(Save Data)

- 데이터를 가져와서 중앙에서 처리하고 다시 저장하는 구조

- 기존 프로그래밍 방식인
  - 데이터를 가져와서 중앙에서 처리하고 다시 저장하는 구조의 단점?

## 대용량 데이터인 경우?

애써 분산 환경에 저장했는데, 처리를 위해 대용량 데이터를 다시 가져와야 한다면?

데이터를 가져오는 비용이 많이 든다

해결책 : 데이터를 가져오지 않고 처리 할 수 있는 '무엇' 인가가 필요

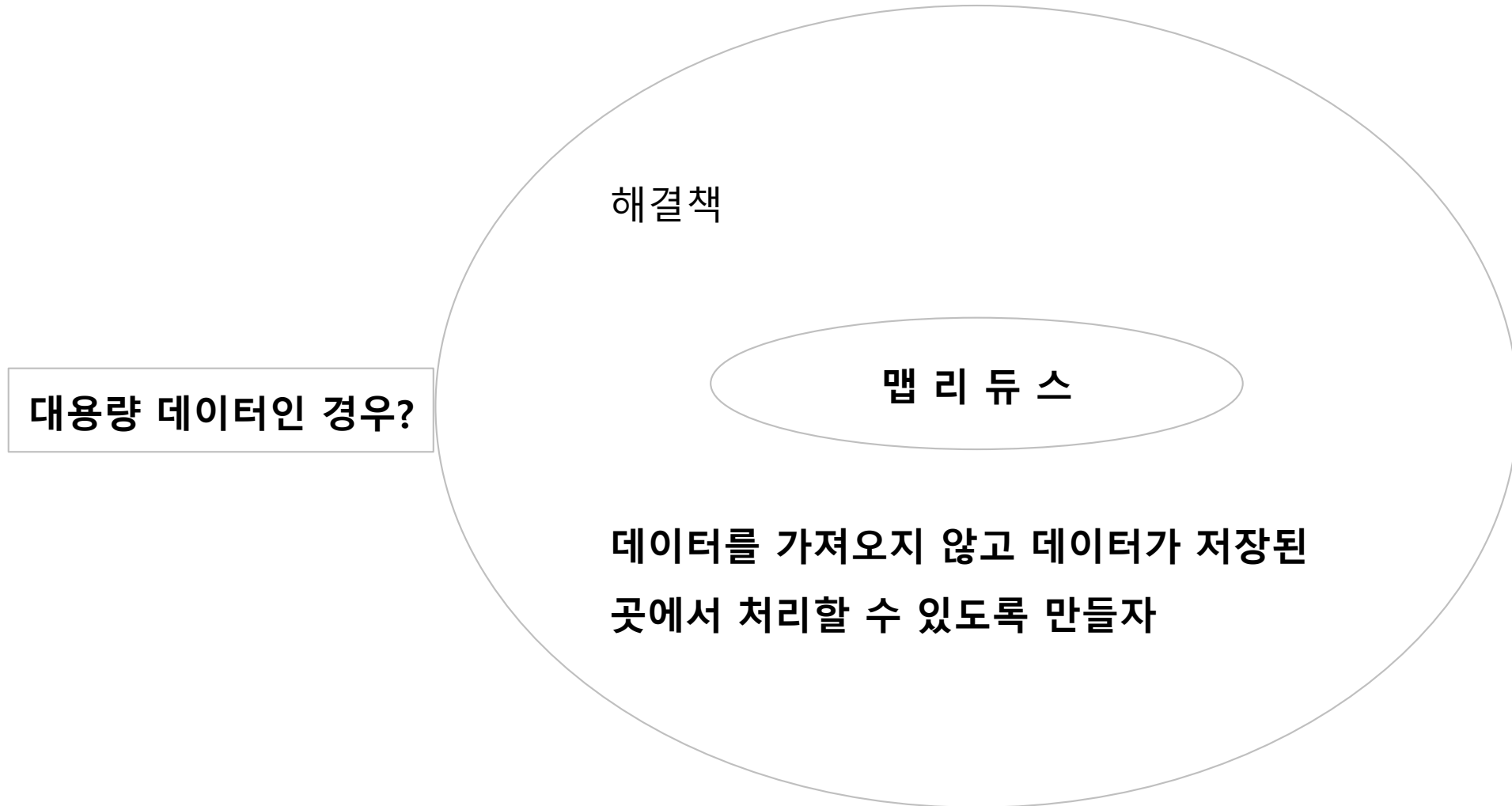
대용량 데이터인 경우?

애써 분산 환경에 저장했는데, 처리를 위해 대용량 데이터를 다시 가져와야 한다면?

데이터를 가져오는 비용이 많이 든다

해결책 : 데이터를 가져오지 않고 처리할 수 있는 '무엇' 인가가 필요





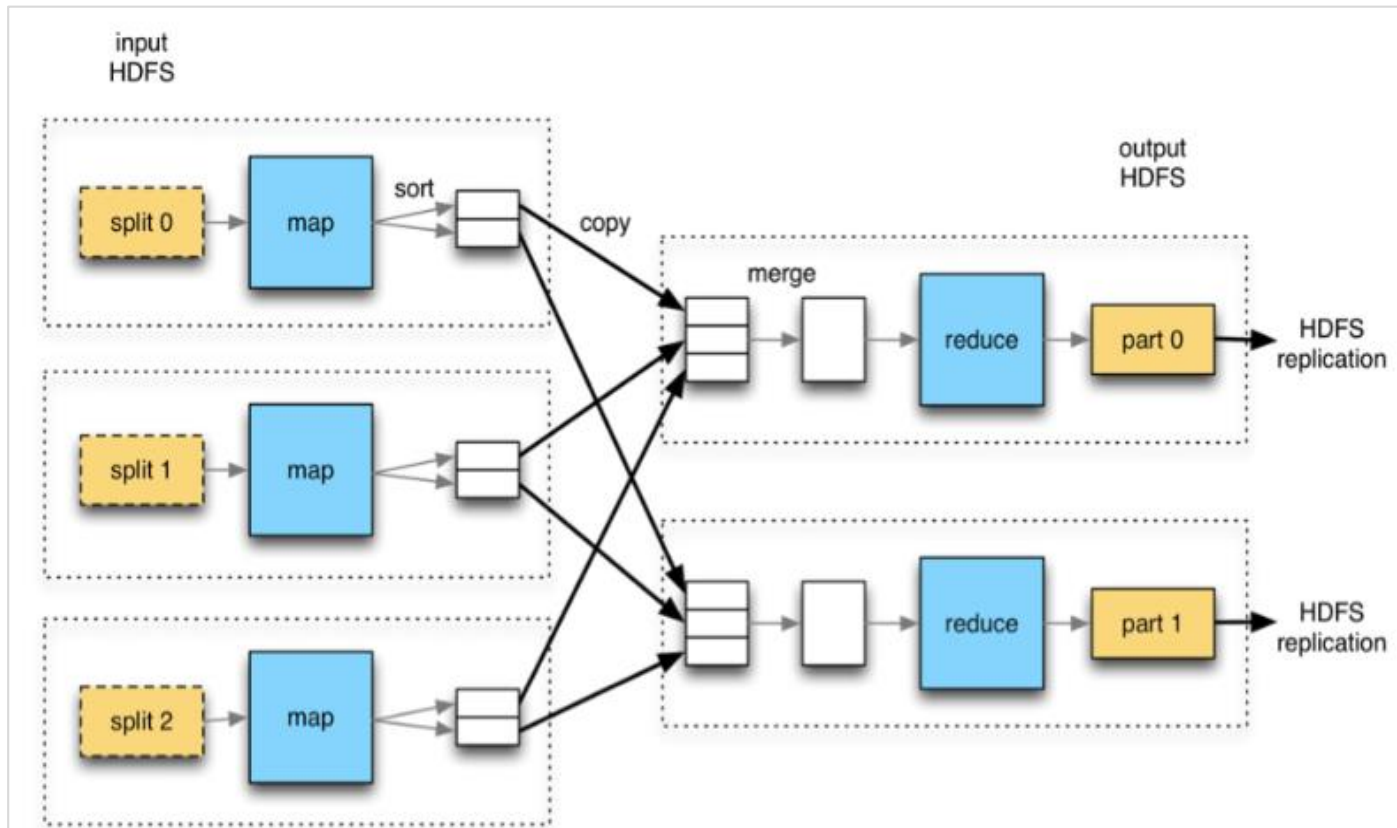
# 맵리듀스 필요성

---

- 맵리듀스 단계
  - 맵(Map) 단계와 리듀스(Reduce) 단계로 구분
  - 맵(Map) 단계
    - 데이터가 저장된 로컬에서 동작
    - 간단한 단위작업을 처리
  - 리듀스(Reduce) 단계
    - 맵 작업의 결과물을 모아서 집계하는 단계

# 맵리듀스

- 분산 처리를 담당하는 맵리듀스 작업은 맵과 리듀스로 구분되어 처리
- 병렬 처리 가능, 여러 컴퓨터에서 동시에 작업을 처리하여 속도 향상이 가능



맵의 입력은 스플릿(InputSplit)단위로 분할

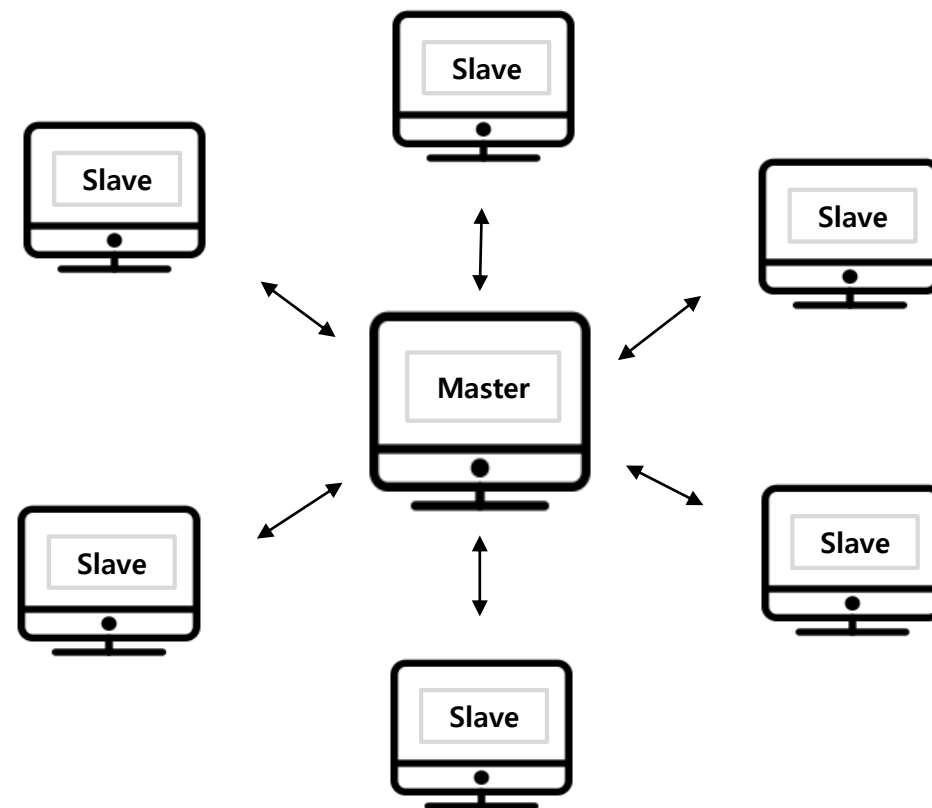
맵작업은 큰 데이터를 하나의 노드에서 처리하지 않고, 분할하여 동시에 병렬 처리하여 작업 시간을 단축

스플릿이 작으면 작업 부하가 분산되어 성능이 향상  
단, 스플릿의 크기가 너무 작으면 맵 작업의 개수가 증가하고 맵 작업 생성을 위한 오버헤드가 증가하여 작업이 느려질 수 있음  
따라서 작업에 따라 적절한 개수의 맵 작업을 생성해야 함

일반적인 맵 작업의 적절한 스플릿 크기 :  
데이터 지역성의 이점을 얻을 수 있는 HDFS 블록의 기본 크기(128MB)

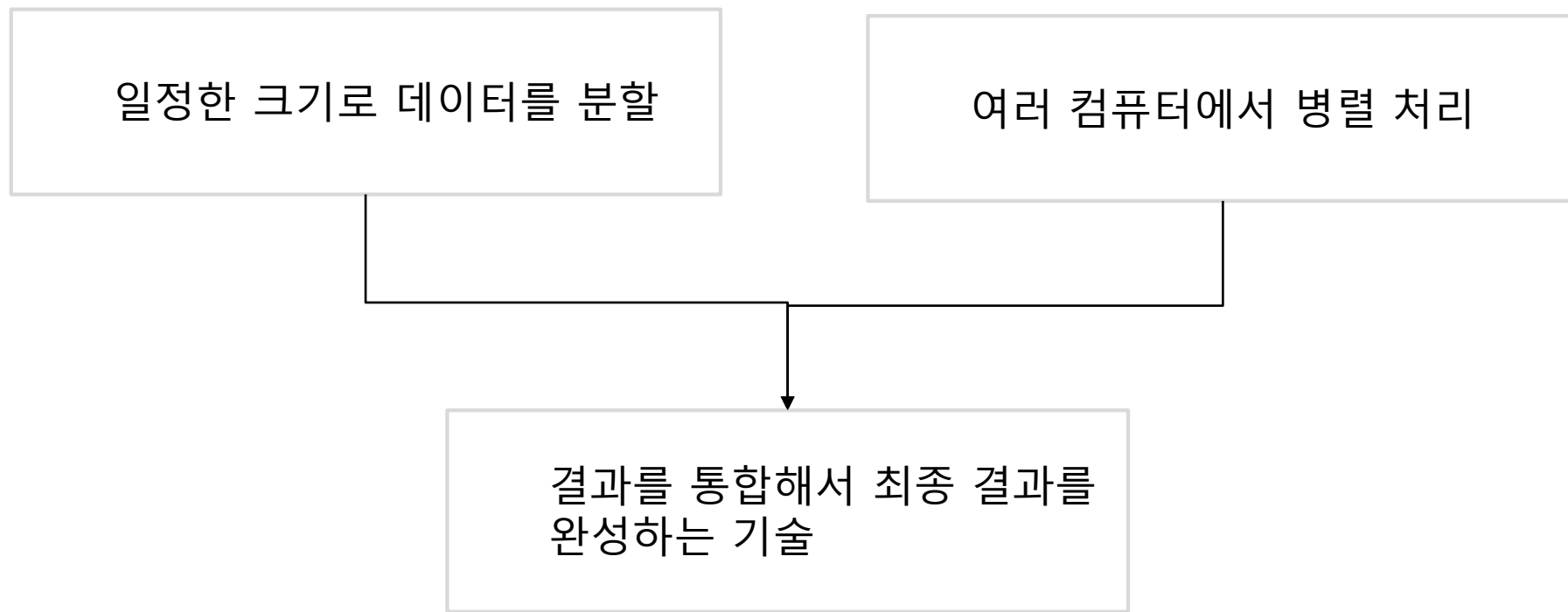
# 맵리듀스

- 개요
  - HDFS에 저장된 파일을 분산 배치  
분석 할 수 있게 도와주는  
프레임워크
  - 데이터에 처리에 대한 부분을 분산 병렬  
처리 방식으로 여러 개의 작업 노드에 분  
산하여 병렬 수행 할 수 있는 프레임워크
  - Master 와 Slave의 협업을 통한 대용량 빅  
데이터 처리



# 맵리듀스

- 기능



# 맵리듀스

- 맵리듀스 분산 병렬 처리 방식

## 맵 단계

1. 입력 파일을 한 줄씩 read해서 데이터를 변형
2. 분산된 데이터를 Key, Value의 리스트로 모으는 단계
3. 개발자가 정의
4. 데이터를 어떻게 처리할 것인가에 대한 로직 보유

## 셔플 단계

1. 맵 단계에서 나온 결과 중간 결과를 해당 리듀스로 전달하는 단계
2. Hadoop이 자동으로 처리

## 리듀스 단계

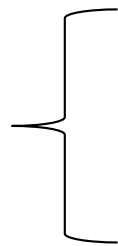
1. 맵의 출력 데이터를 찾아서 집계하는 단계
2. 개발자가 정의

# 맵리듀스

- 맵리듀스 분산 병렬 처리 방식



개발자 정의



맵(Map) 함수

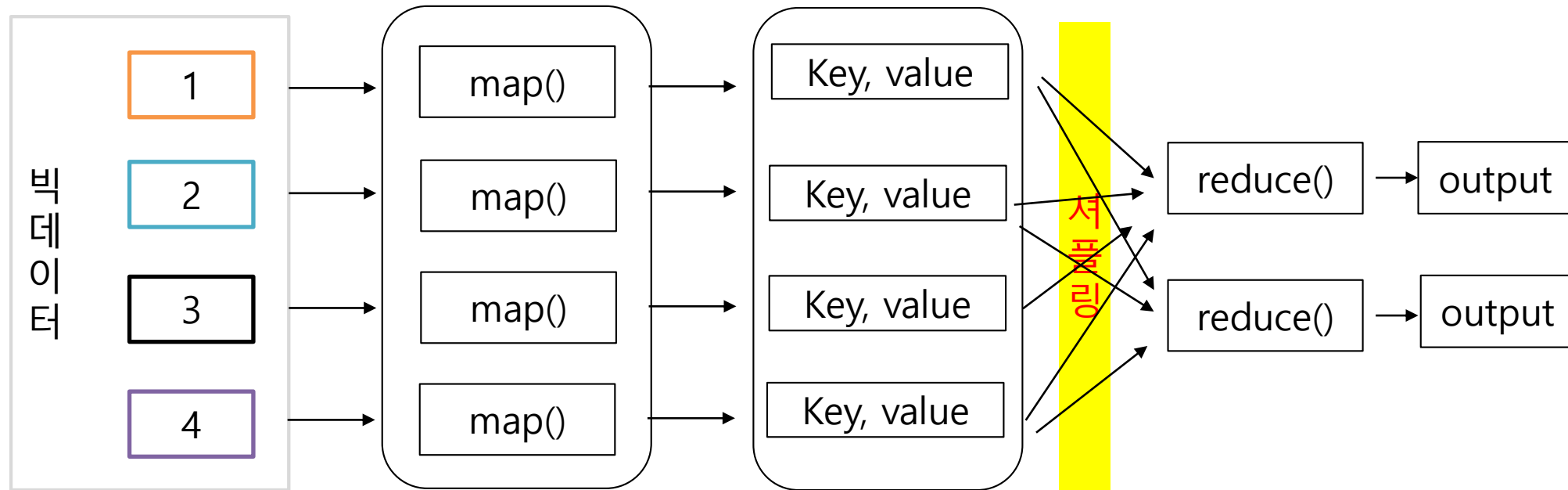
- 입력 파일을 한 줄씩 읽어서 데이터를 변형 (transformation)

리듀스(Reduce) 함수

- 맵의 결과 데이터를 집계(aggregation)

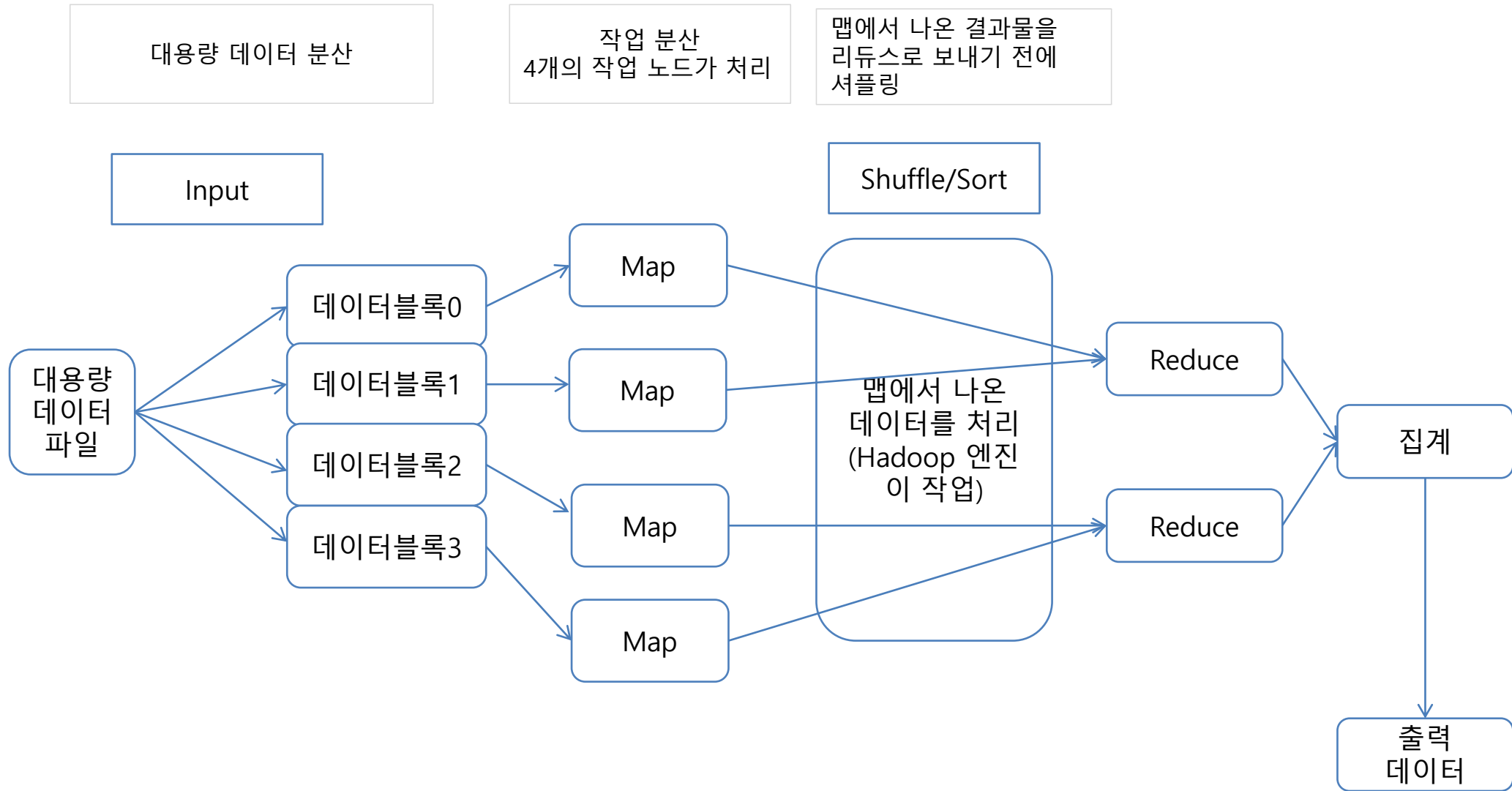
# 맵리듀스

- 맵리듀스 분산 병렬 처리 방식
  - Hadoop이 map() 함수들을 데이터에 배포



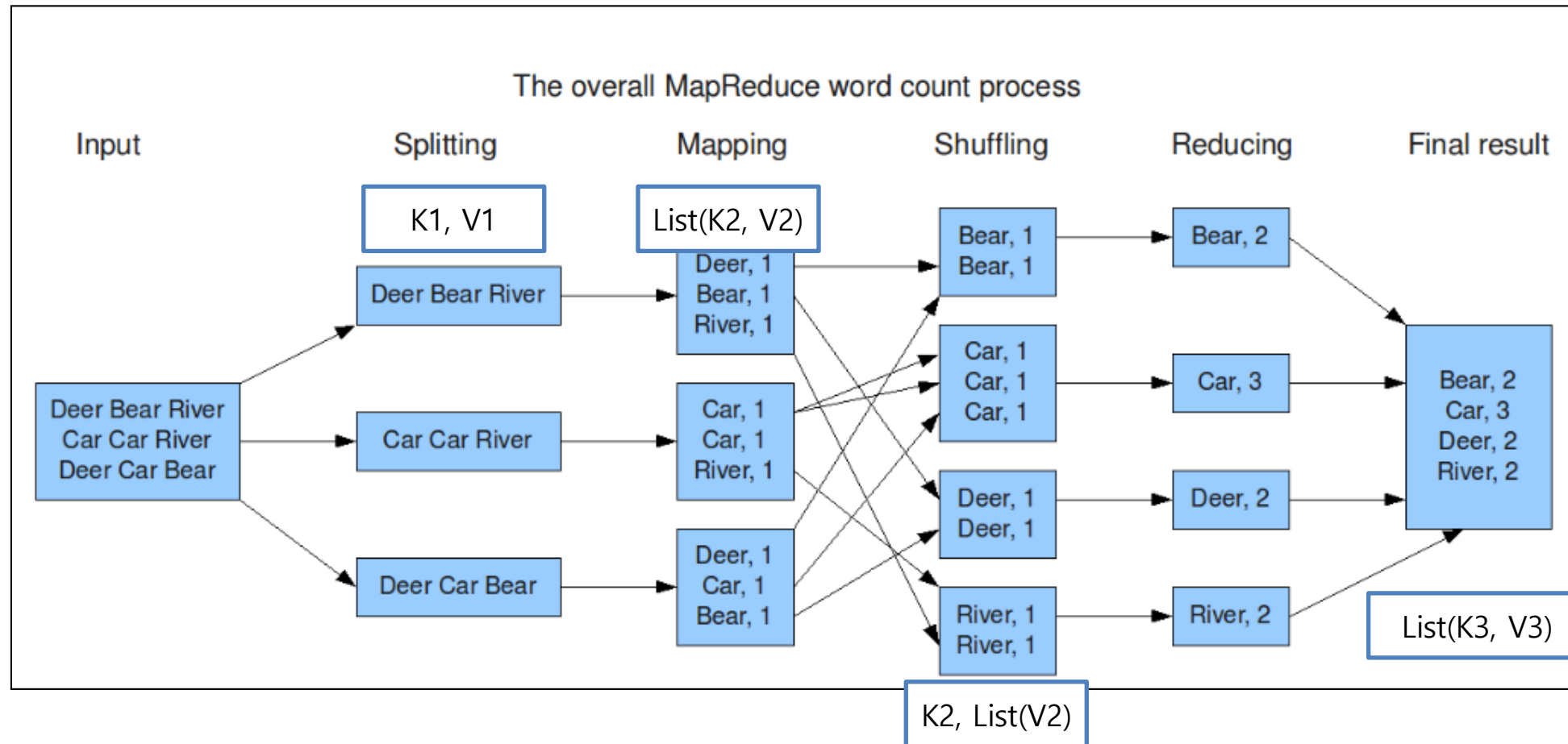


# 맵리듀스

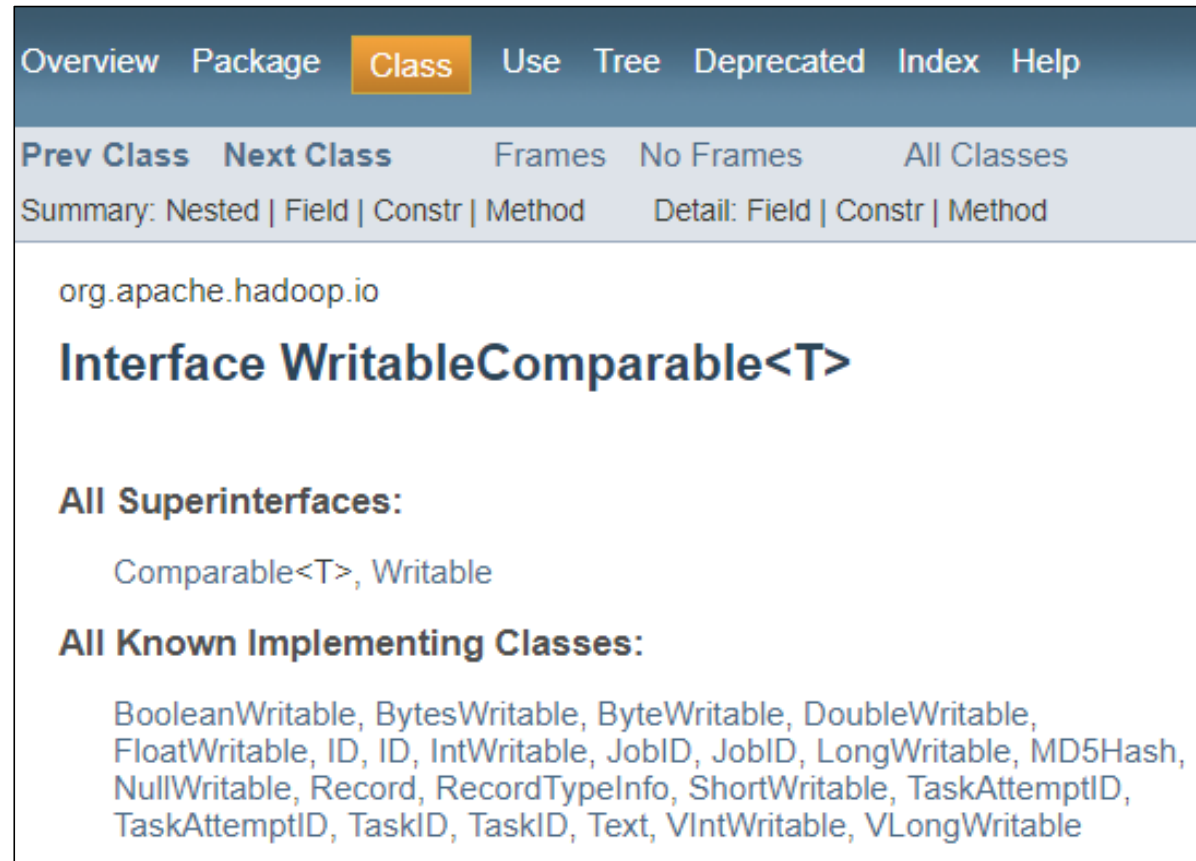


# 맵리듀스

- 맵리듀스 분산 병렬 처리 방식 예제
  - Word count 예시



- Wrapper API



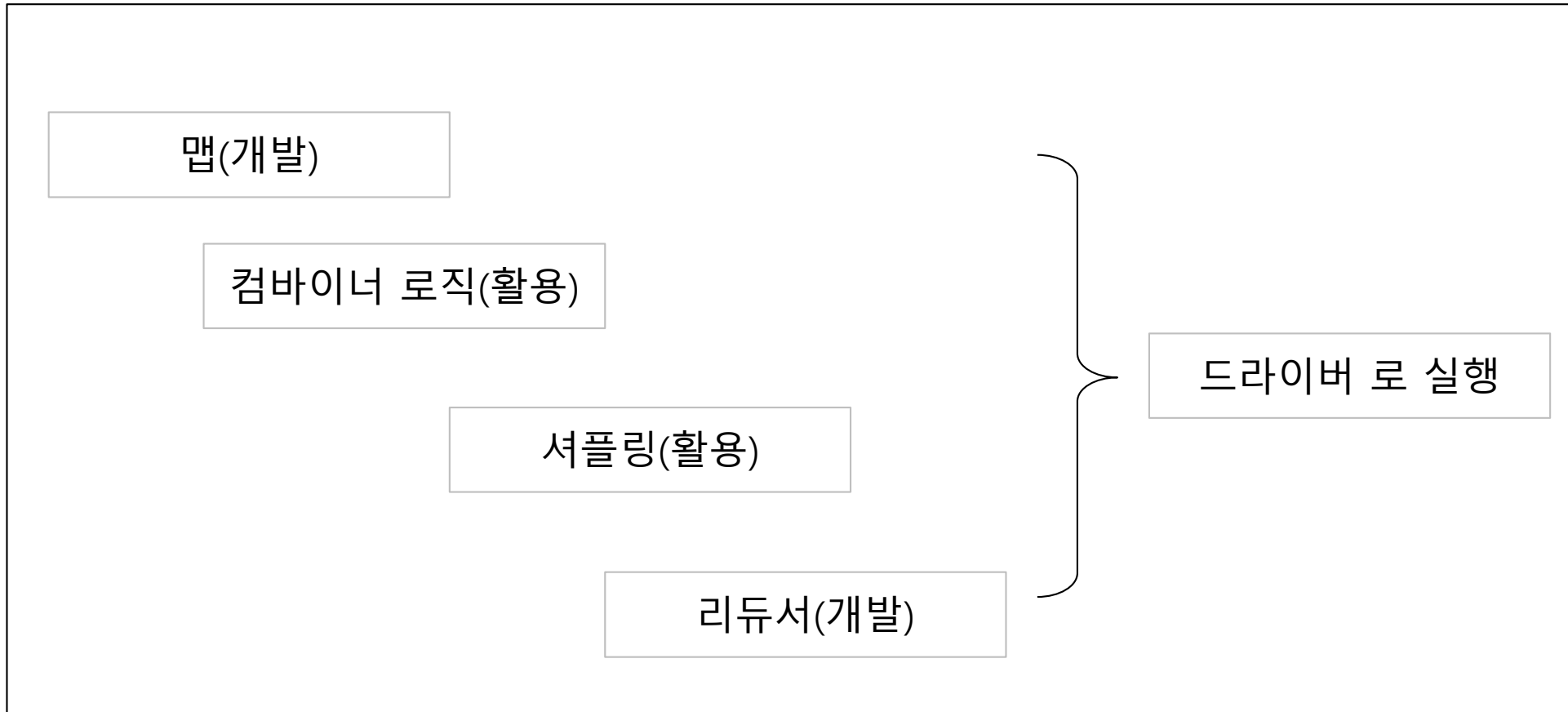
The screenshot shows the Hadoop API documentation for the `org.apache.hadoop.io.WritableComparable<T>` interface. The navigation bar at the top includes links for Overview, Package, Class (highlighted), Use, Tree, Deprecated, Index, and Help. Below this, there are links for Prev Class, Next Class, Frames, No Frames, and All Classes. A summary section lists Nested, Field, Constr, Method, and Detail: Field, Constr, Method. The main content area displays the package `org.apache.hadoop.io`, the interface name **Interface WritableComparable<T>**, and sections for All Superinterfaces (Comparable<T>, Writable) and All Known Implementing Classes (BooleanWritable, BytesWritable, ByteWritable, DoubleWritable, FloatWritable, ID, ID, IntWritable, JobID, JobID, LongWritable, MD5Hash, NullWritable, Record, RecordTypeInfo, ShortWritable, TaskAttemptID, TaskAttemptID, TaskID, TaskID, Text, VIntWritable, VLongWritable).

그림 출처 : <https://hadoop.apache.org/docs/r2.8.0/api/org/apache/hadoop/io/WritableComparable.html>

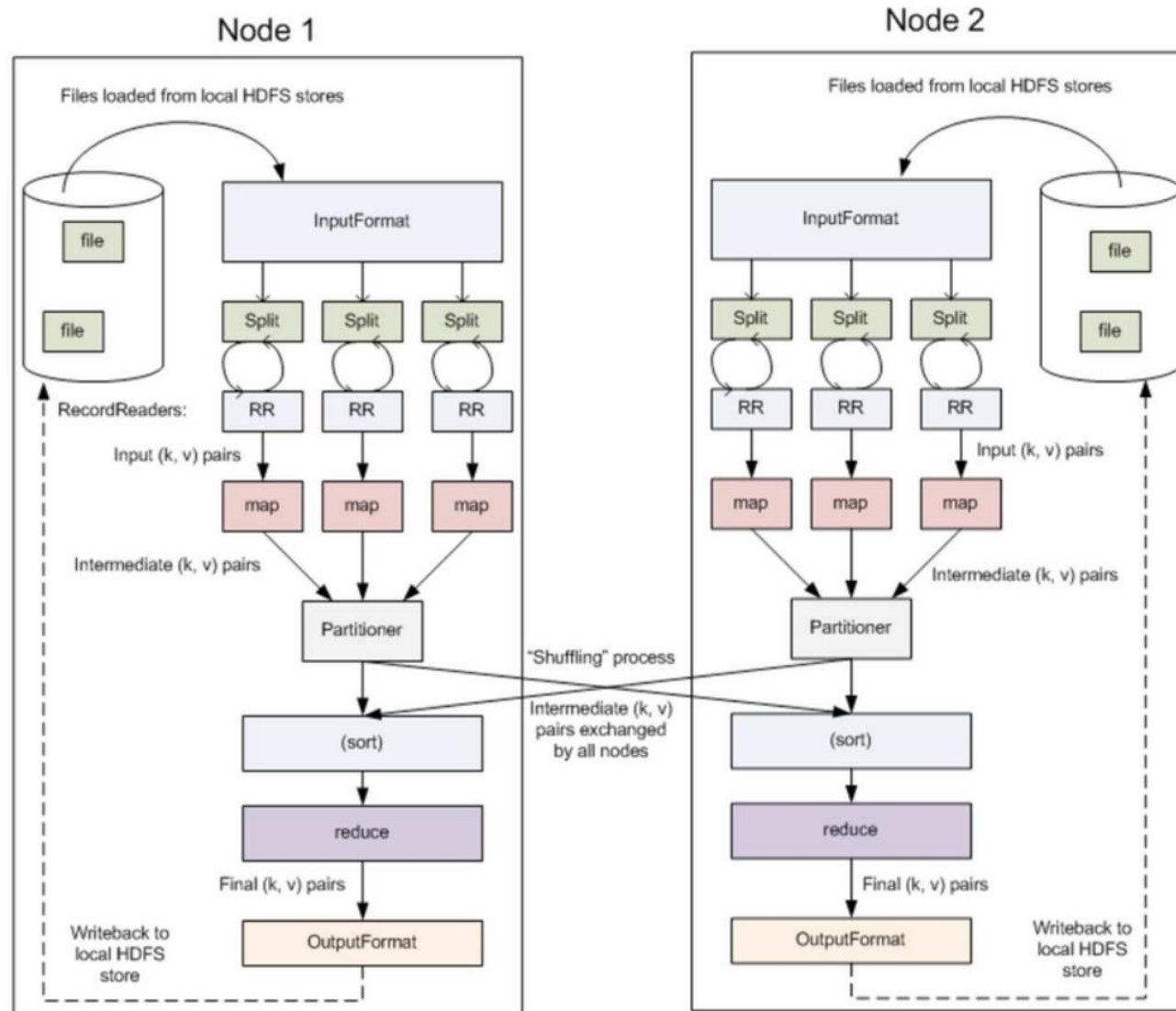
- 콤바이너 클래스
  - 맵 태스크의 출력 데이터는 네트워크를 통해 리듀스 태스크로 전달되며, 맵태스크와 리듀스 태스크 사이의 데이터 전달은 셔플
  - 네트워크를 통해 데이터가 전송되기 때문에 전송할 데이터의 크기를 줄일수록 전체 잡의 성능은 향상
  - **셔플 할 데이터의 크기를 줄이는 효과 발생**
  - 동작
    - 매퍼의 출력 데이터를 입력 데이터로 전달받아 연산을 수행
    - 로컬 노드에 출력 데이터를 생성 한 후 리듀스 태스크에 네트워크로 전달
    - 따라서 출력 데이터는 기존 매퍼의 출력 데이터보다 크기가 감소되었기 때문에 네트워크 비용 감소 효과

# 맵리듀스 프로그래밍

- 개발 및 실행 프로세스



# 맵리듀스 처리 단계



[https://www.researchgate.net/figure/Detailed-Hadoop-MapReduce-Data-Flow-14\\_fig1\\_224255467](https://www.researchgate.net/figure/Detailed-Hadoop-MapReduce-Data-Flow-14_fig1_224255467)

입력

맵(Map)

컴파이너(Combiner)

파티셔너(Partitioner)

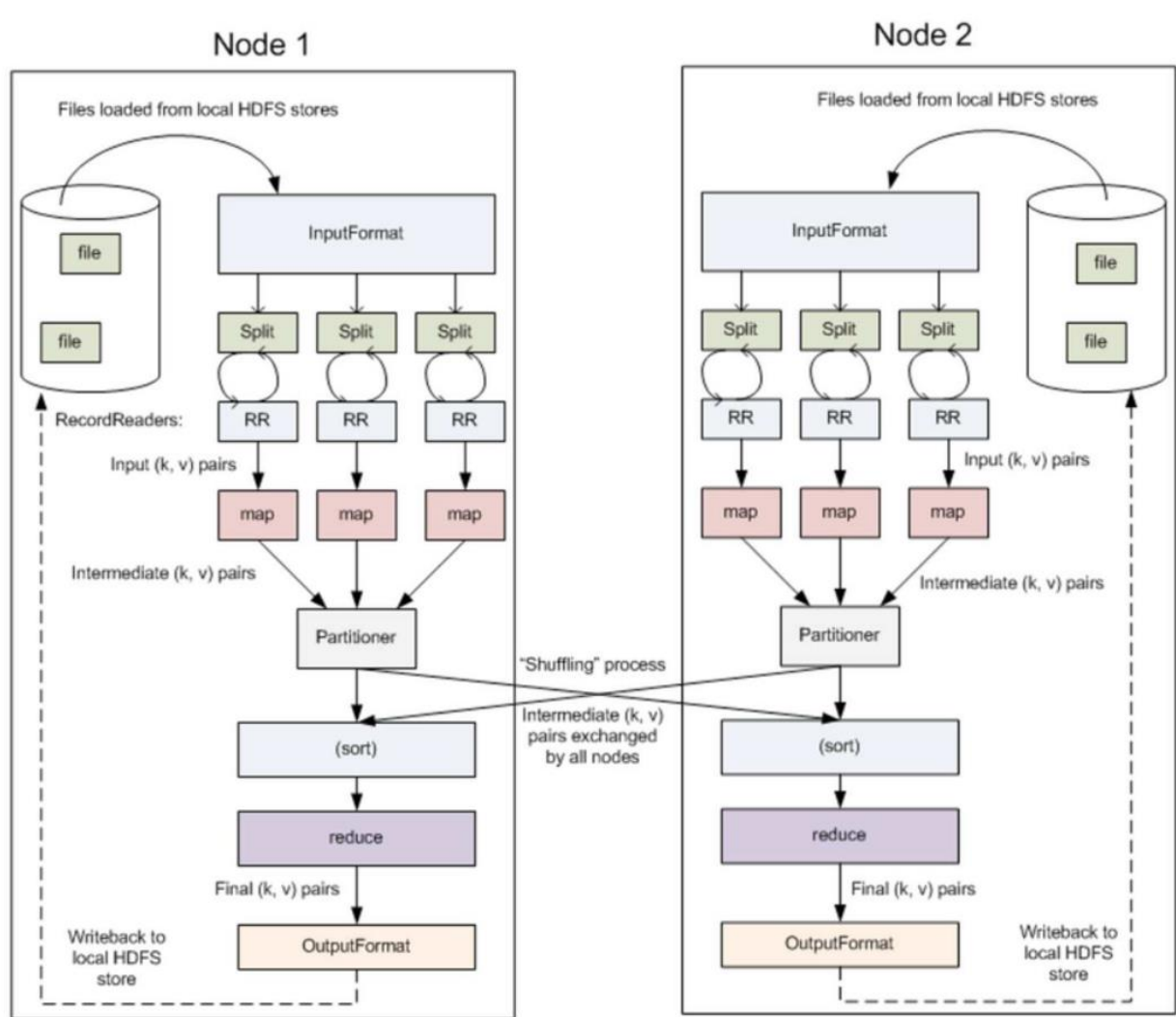
셔플(Shuffle)

정렬(Sort)

리듀서(Reduce)

출력

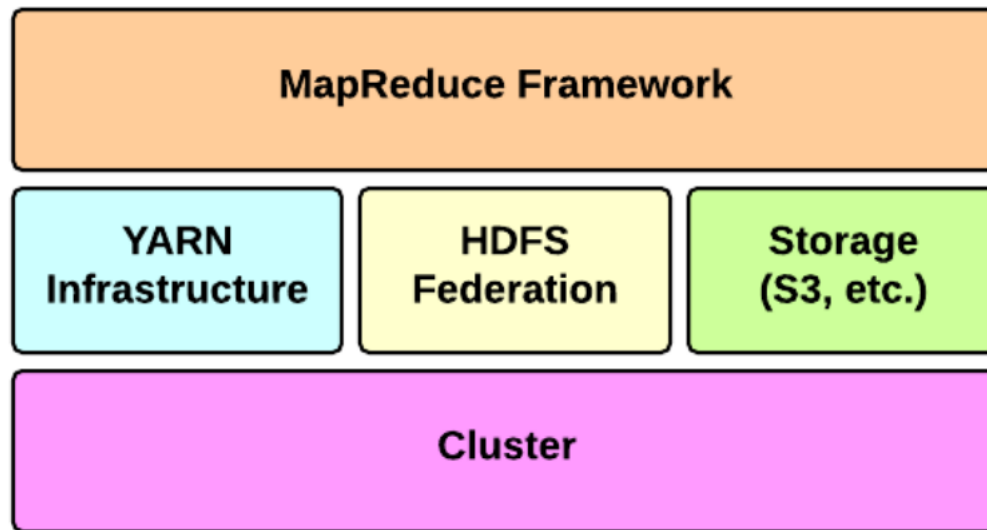
# 맵리듀스 처리 단계



입력	데이터를 text, csv, gzip형태의 데이터를 읽어서 맵으로 전달
맵(Map)	입력을 분할하여 키별로 데이터를 처리
컴파이너(Combiner)	네트워크를 타고 전송되는 데이터를 줄이기 위하여 맵의 결과를 정리, 로컬 리듀서라고도 함, 옵션
파티셔너(Partitioner)	맵의 출력 결과 키 값을 해쉬 처리하여 어떤 리듀서로 넘길지 결정
셔플(Shuffle)	각 리듀서로 데이터 이동
정렬(Sort)	리듀서로 전달되는 데이터를 키 값 기준으로 정렬
리듀서(Reduce)	리듀서로 데이터를 처리하고 결과를 저장
출력	리듀서의 결과를 정의된 형태로 저장

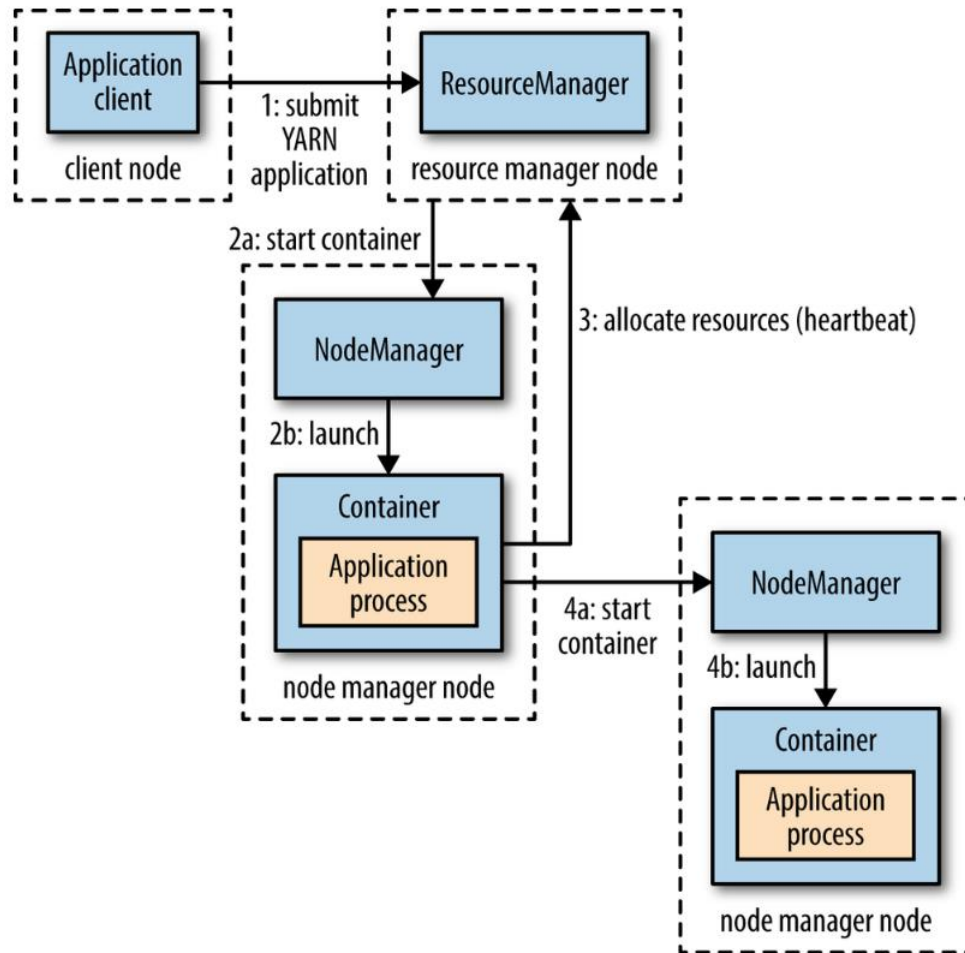
## 맵리듀스 응용 프로그램

- MapReduce 프레임 워크는 YARN을 기반으로 실행되는 많은 가능한 프레임 워크 중 하나





# 맵리듀스 응용 프로그램과 Yarn



- 애플리케이션의 라이프 사이클 관리는 애플리케이션 마스터 (Application Master)와 컨테이너(Container)를 이용하여 처리
  - 클라이언트 : 리소스 매니저에 애플리케이션을 제출
  - 리소스 매니저 : 비어 있는 노드에서 애플리케이션 마스터를 실행
  - 애플리케이션 마스터 : 작업 실행을 위한 자원을 리소스 매니저에 요청하고, 자원을 할당 받아서 각 노드에 컨테이너를 실행하고, 실제 작업을 진행
  - 컨테이너 : 실제 작업이 실행되는 단위
- 컨테이너에서 작업이 종료되면 결과를 애플리케이션 마스터에게 알리고 애플리케이션 마스터는 모든 작업이 종료되면 리소스매니저에 알리고 자원을 해제

## | 실습을 위한 환경 구축

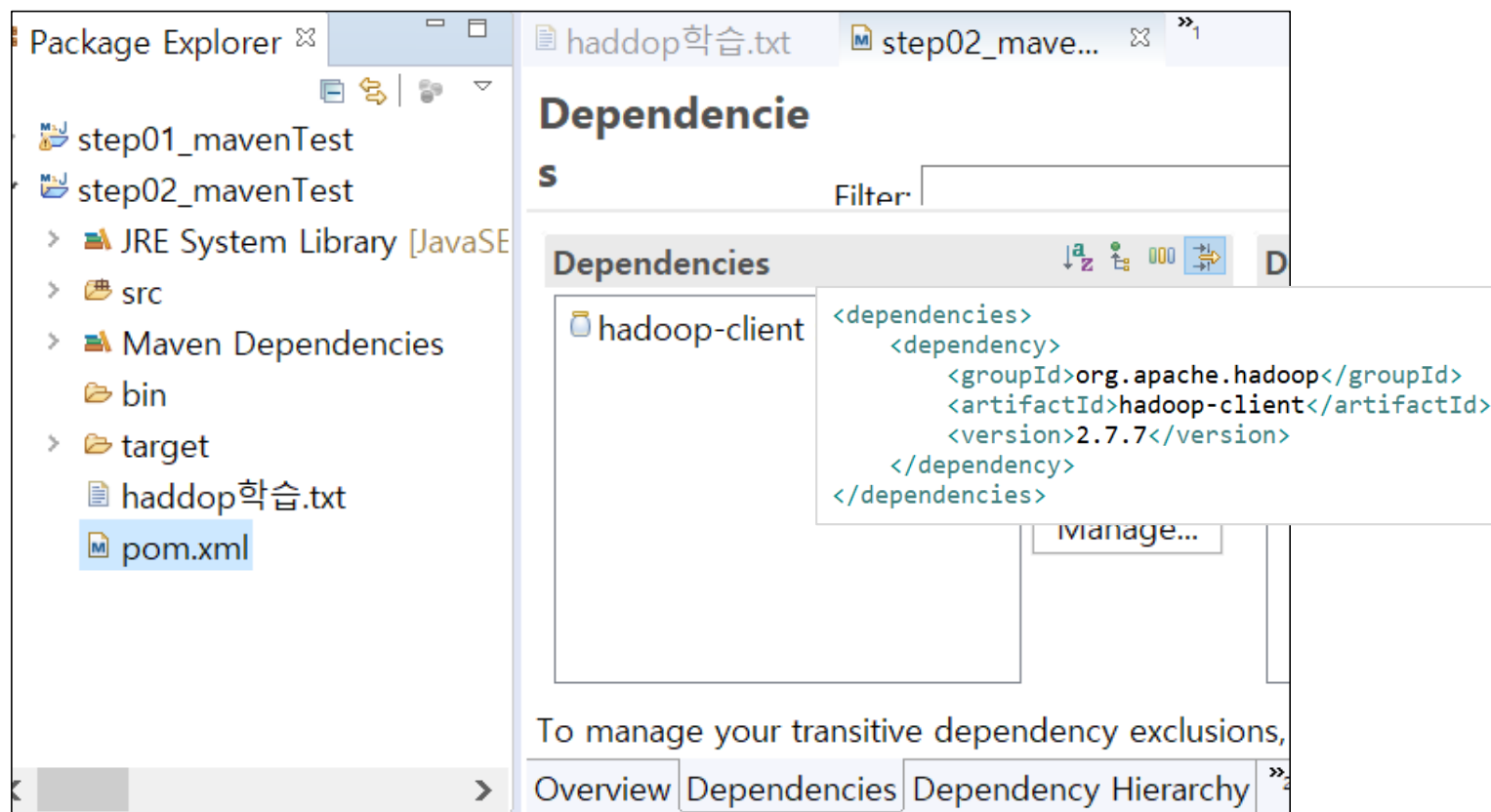
# 맵리듀스 프레임워크

- 맵리듀스 프로그램 개발 및 실행 단계

구분	OS	작업 단계
개발	window	이클립스로 개발 Maven을 활용한 Hadoop library 설정
[FileZilla 등으로 파일 전송]		
실행	Linux	1. HDFS에 데이터 저장 2. MapReduce 프로그램 실행 1. 컴파일 2. jar 로 압축 3. yarn 기반으로 실행

# 맵리듀스 프레임워크

- 맵리듀스 프로그램 개발 단계
  - 윈도우 eclipse에 Hadoop 버전에 맞는 library 셋팅하기
    - Maven tool 사용



# | Sqoop

# 목차

---

- 스쿱(Sqoop) 개요
- 스쿱(Sqoop) 데이터 가져오기
- 스쿱(Sqoop) 데이터 내보내기

# 스쿱(Sqoop) 개요

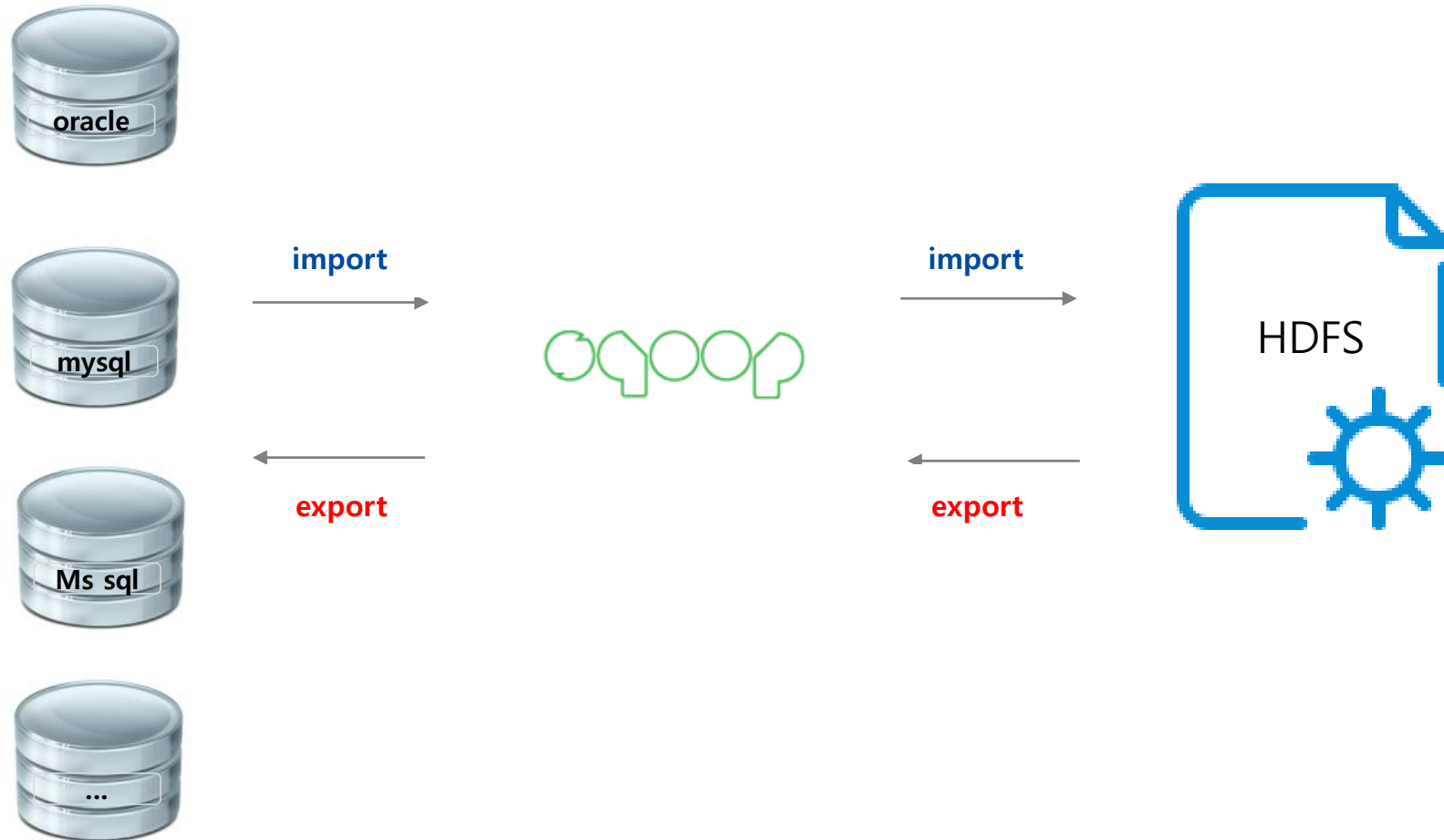
---



- RDBMS와 HDFS 간의 효율적인 대용량 데이터 전송하는 도구
- Hadoop 에코 시스템 수집 Software
- 외부 시스템의 데이터를 HDFS로 가져와서 하이브 테이블, Hbase 테이블 등 Hadoop의 다양한 파일 형태로 저장 가능
- <http://sqoop.apache.org/>

# 스쿱(Sqoop) 개요

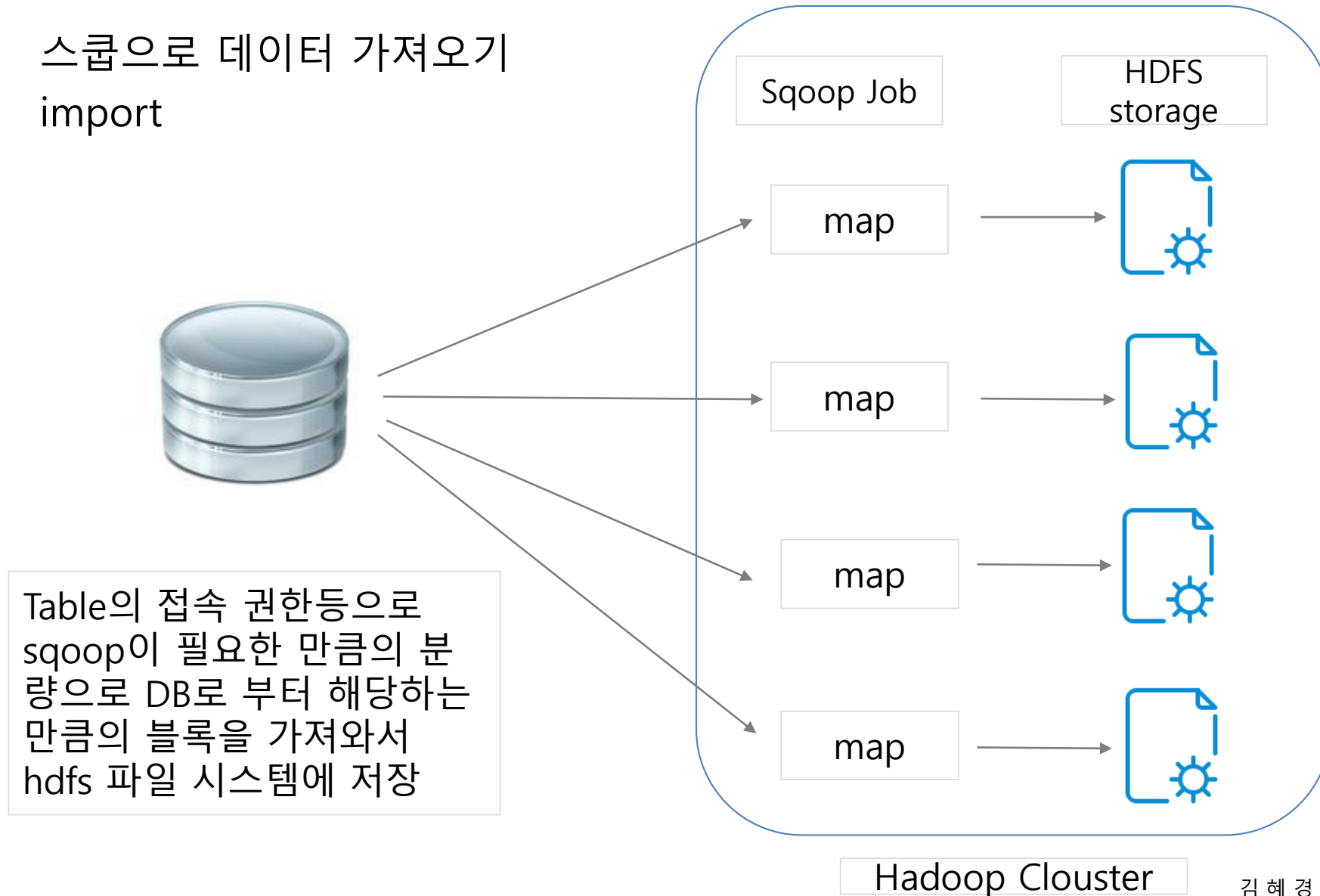
- JDBC와 호환되는 모든 RDBMS에 사용 가능





# 스쿱(Sqoop) 데이터 가져오기

- 스쿱으로 데이터 가져오기
- import



# 스쿱(Sqoop) 데이터 가져오기

- 스쿱 데이터 가져오기
  - 관계형 데이터베이스에서 메타데이터 수집
  - 필요 정보
    - 데이터베이스명, 테이블명, ID, PW등

```
sqoop list-databases
--connect jdbc:mysql://_HOST_NAME/_DATABASE_NAME_
--username ID --password PW
```

```
-----
테이블 list 검색됨
```

# 스쿱(Sqoop) 데이터 가져오기

- 스쿱 데이터 가져오기
  - 맵 적용 작업을 Hadoop에 요청
  - Sqoop import 명령어 특징
    - 리듀스 작업 없이 맵잡만 실행
  - Hadoop 클러스터의 각 노드는 데이터베이스 접근 권한을 미리 부여받아야 함
  - 옵션
    - -m : 몇 개의 map job을 만들것인지에 대한 개수
    - --target-dir : 검색해서 어느 파일 시스템에 넣을 것인가 명시
  - RDBSM의 table의 내용을 HDFS의 파일 시스템에 저장하는 명령어

```
sqoop import
--connect jdbc:mysql://localhost:3306/world
--username ID
--password PW
--table Country -m 1 --target-dir /user/sqoop/country
```

# 스쿱(Sqoop) 데이터 가져오기

- 스쿱 데이터 가져오기
  - RDBMS에서 가져온 데이터는 HDFS 디렉터리에 저장됨
    - RDBMS의 table 구조의 데이터가 HDFS에 , 구분 형태로 저장됨
  - HDFS에 저장된 파일 확인 명령어

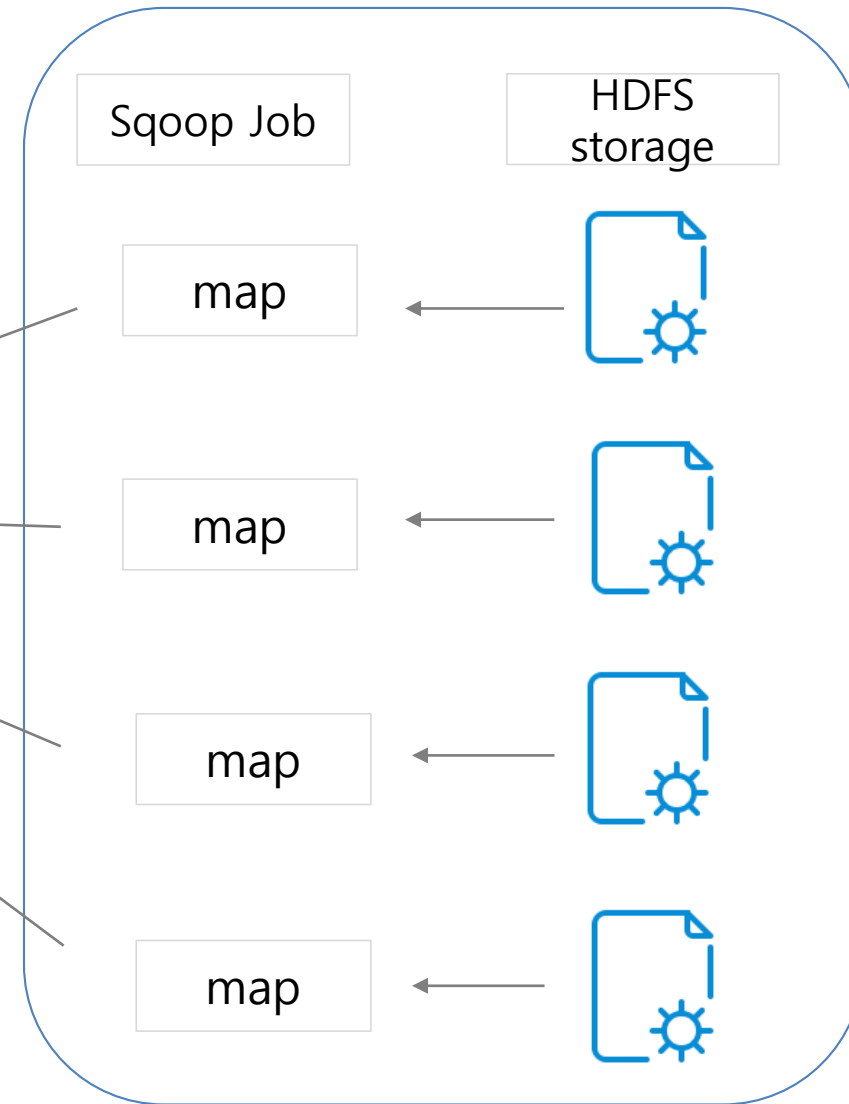
```
hdfs dfs -ls sqoop-mysql-import/country
Hdfs dfs -cat sqoop-mysql-import/country/part-m-0000
```

# 스쿱(Sqoop) 데이터 내보내기

- 스쿱으로 데이터 내보내기
- export



메타데이터에 접근해서 table의 구성 정보를 확인하고 map job을 실행하고 map job이 파일들의 정보를 읽어서 RDBMS로 넣는 구조



# 스쿱(Sqoop) 데이터 내보내기

- 스쿱 데이터 내보내기
  - 관계형 데이터베이스의 메타데이터 수집
  - 작업 단계
    - RDBMS에 Table 생성
    - Export 명령어를 사용해서 HDFS에 내장된 데이터를 RDBMS에 export
    - 단, Hadoop 클러스터의 각 노드는 RDBMS 접근 권한을 미리 부여받아야 함

```
sqoop export
--connect jdbc:mysql://localhost:3306/world
--username sqoop
--password sqoop
--table CityExport
-m 4 --export-dir /user/sqoop/city3
```

# 스쿱(Sqoop) 실습

- 사용 DB 및 Schema
  - MySql
  - world\_innodb.sql

Table	contents
City	전 세계 나라에 관한 정보
Country	각 나라 안에 있는 몇몇 도시의 정보
CountryLanguage	각 나라에서 사용되는 언어





# 목차

---

- 하이브(Hive) 개요
- 하이브 아키텍처
- 데이터 모델

# 하이프(Hive) 개요

---

- 빅데이터를 다룰 수 있는 웨어하우스 제공
- Hadoop 기반 위에 동작
- 데이터 저장 : HDFS에 저장
- Hadoop 기반의 쿼리 엔진
  - 맵리듀스 작성 없이도 쿼리 언어만으로 Hadoop의 비정형 데이터 분석 가능
- SQL과 유사한 언어
  - HiveQL
    - 구조적인 질의 언어
    - RDBMS 질의 처럼 빠르게 애드혹 질의(ad-hoc query) 가능

# 하이프(Hive) 개요

---

- 실행 원리
  - Query 실행시 맵리듀스 프로그램으로 자동 전환되어 결과 생성
    - 분산 병렬 처리 의미
- 제약조건
  - RDBMS 처럼 테이블을 이용하여 쿼리 수행, 비정형화된 입력 소스 분석에는 부적합
  - Sql과 흡사하고 table의 중요한 구조이기 때문에 정형화된 구조의 데이터 처리에 적합

# 하이프(Hive) 아키텍처

- 하이브는 Client와 Server 구조로 되어 있음
  - JDBC : Java 통신 protocol
  - Thrift : client와 server간 데이터를 전송 할수 있는 통신 protocol



JDBC 기반  
응용 프로그램 지원

Thrift 기반  
응용 프로그램 지원

ODBC 기반  
응용 프로그램 지원

하이프 client로 사용할 수 있는 응용 프로그램들

# 하이프(Hive) 아키텍처

---

- 하이브 서비스

하이브 서버

하이브 Web Interface

Metastore

Commend Line  
Interface[CLI]

Driver

Apache Derby Database

# 하이프(Hive) 아키텍처

- 하이브 서비스

하이브 서버

Commend Line  
Interface[CLI]

하이브 Web Interface

Driver

Metastore  
- table의 메타 정보 저장  
Table명, 컬럼명 등

Apache Derby Database  
-

# 하이프(Hive) 아키텍처

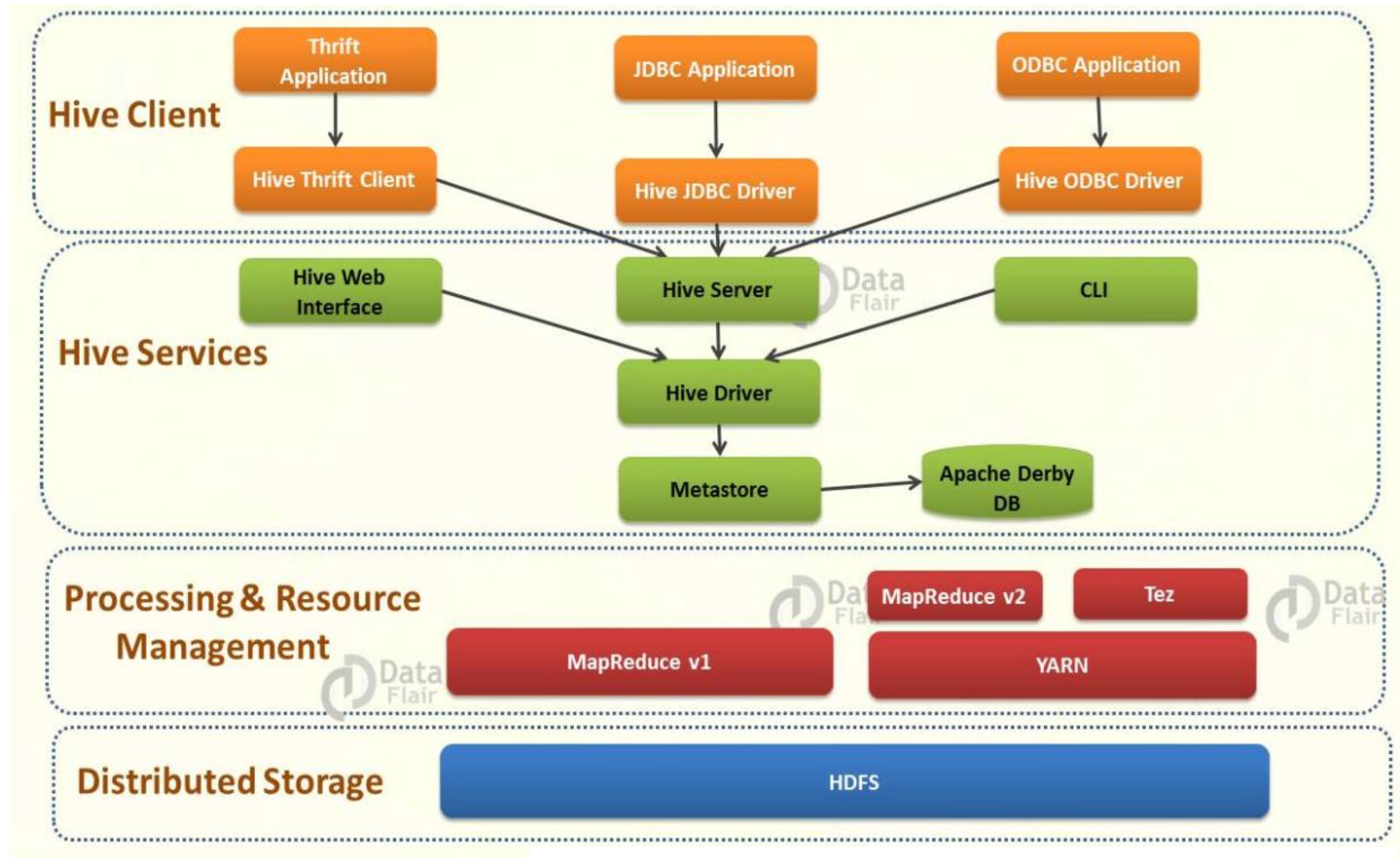


그림 출처 : <https://data-flair.training/blogs/apache-hive-architecture/>

# 하이프(Hive) 아키텍처

## ➤ 데이터 흐름

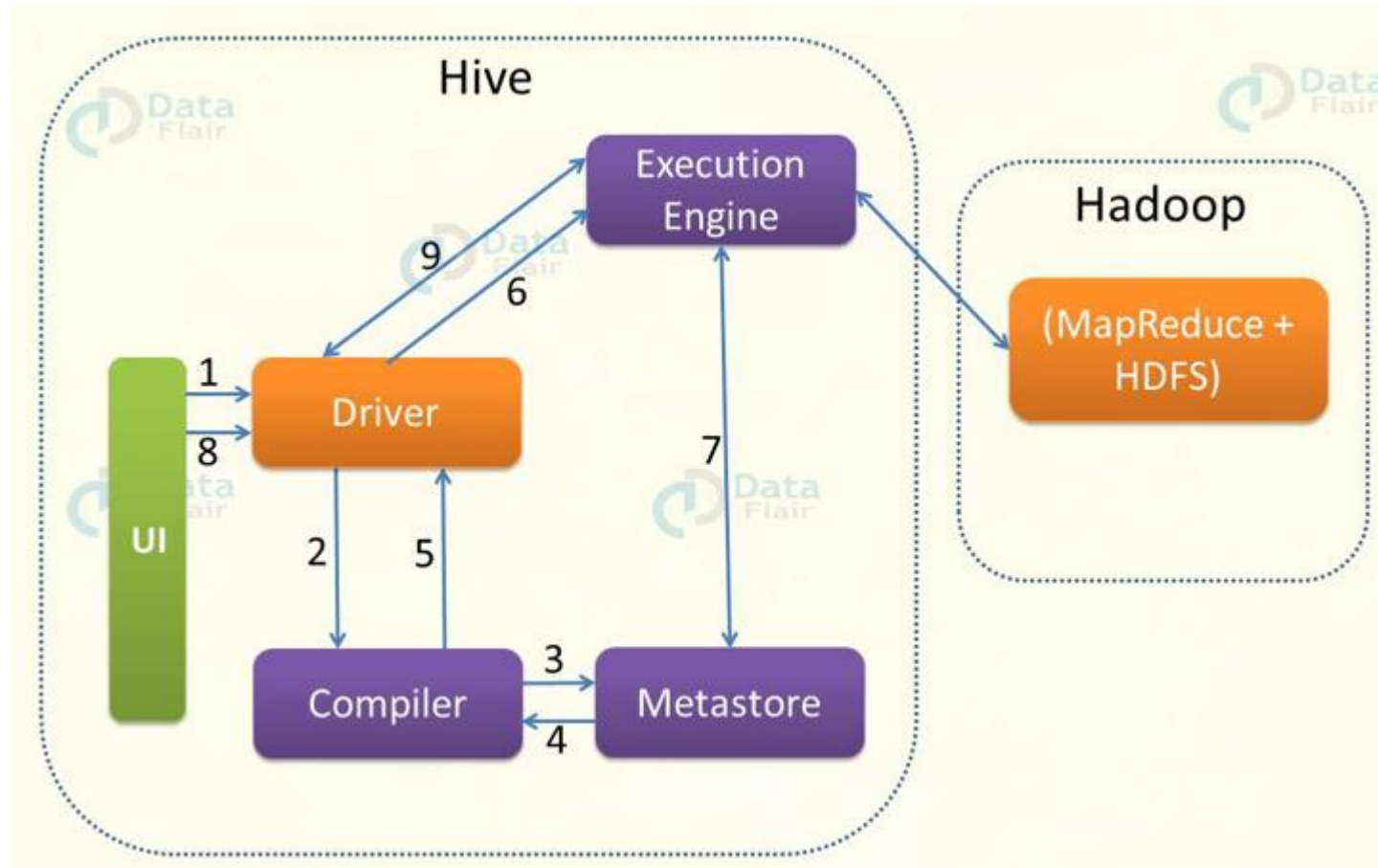


그림 출처 : <https://data-flair.training/blogs/apache-hive-architecture/>



# 하이프 아키텍처와 데이터 모델

---

- 하이브 데이터 모델
  - Hadoop 상에 구축된 정형화된 데이터를 관리하고 쿼리하는 적합한 시스템
  - 스토리지로 HDFS를 사용
  - 구축된 데이터를 처리하거나 가공, 처리, 탐색에 적합
    - 단 온라인 상의 실시간 트랜잭션 처리에는 부적합[OLTP(online transaction processing)에는 부적합]
    - 데이터를 모아 놓는 warehouse용으로 사용

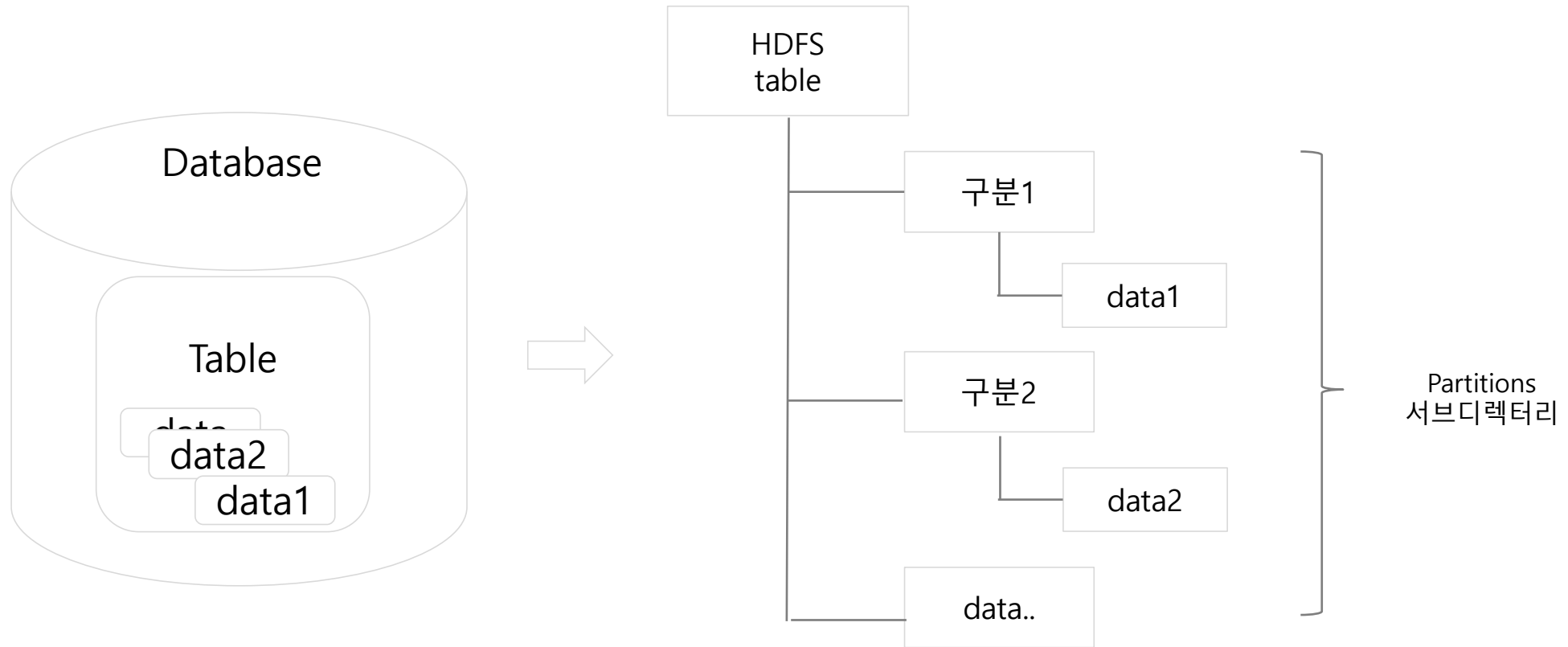
# 하이프 아키텍처와 데이터 모델

- 하이브 데이터 모델
  - 데이터 관리 방식 및 계층 구조
    - 테이블 -> HDFS의 디렉토리로 간주
    - 파티션 -> HDFS의 서브 디렉터리
    - 데이터 -> HDFS의 file

## 파티션이란?

대용량 테이블을 논리적으로 나누어 효율적인 쿼리가 가능하도록 함

# 하이프 아키텍처와 데이터 모델



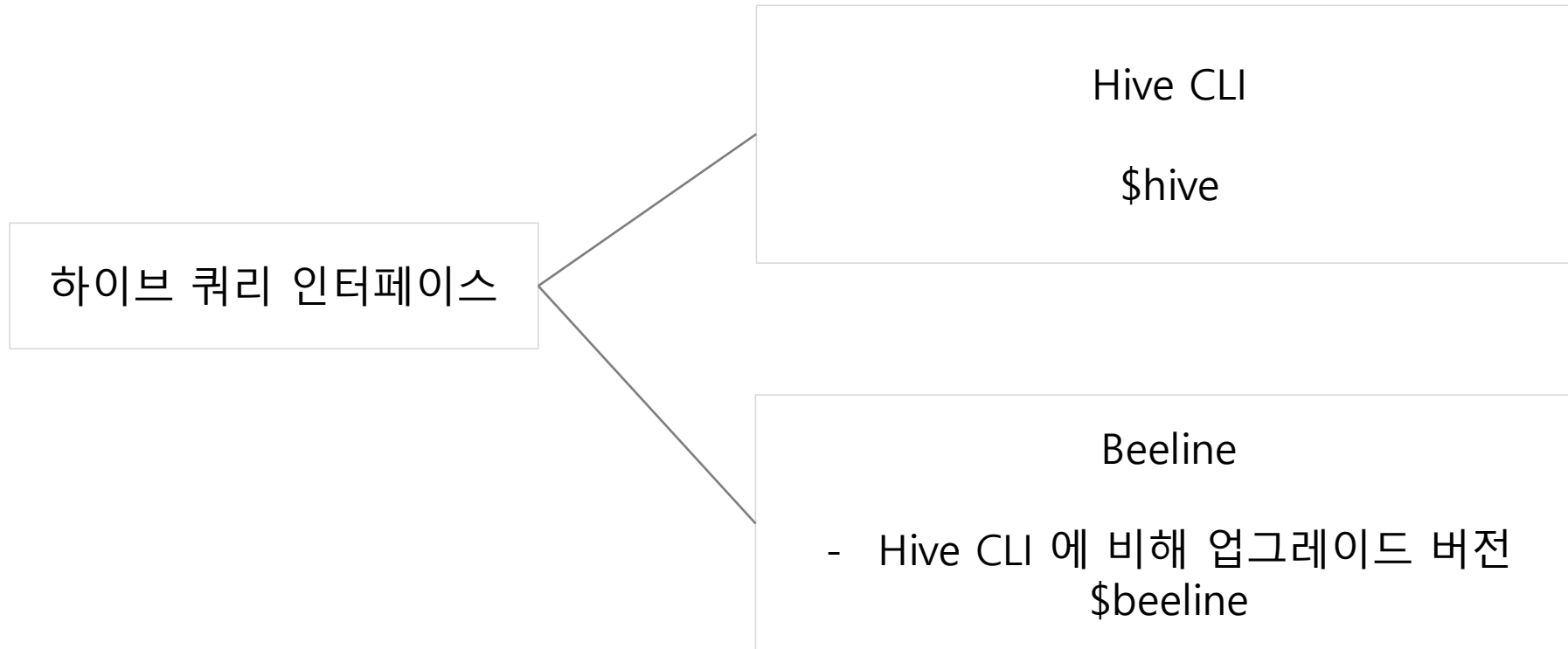
# 하이프 아키텍처와 데이터 모델

- 하이브 데이터 모델
  - 하이브 Metastore



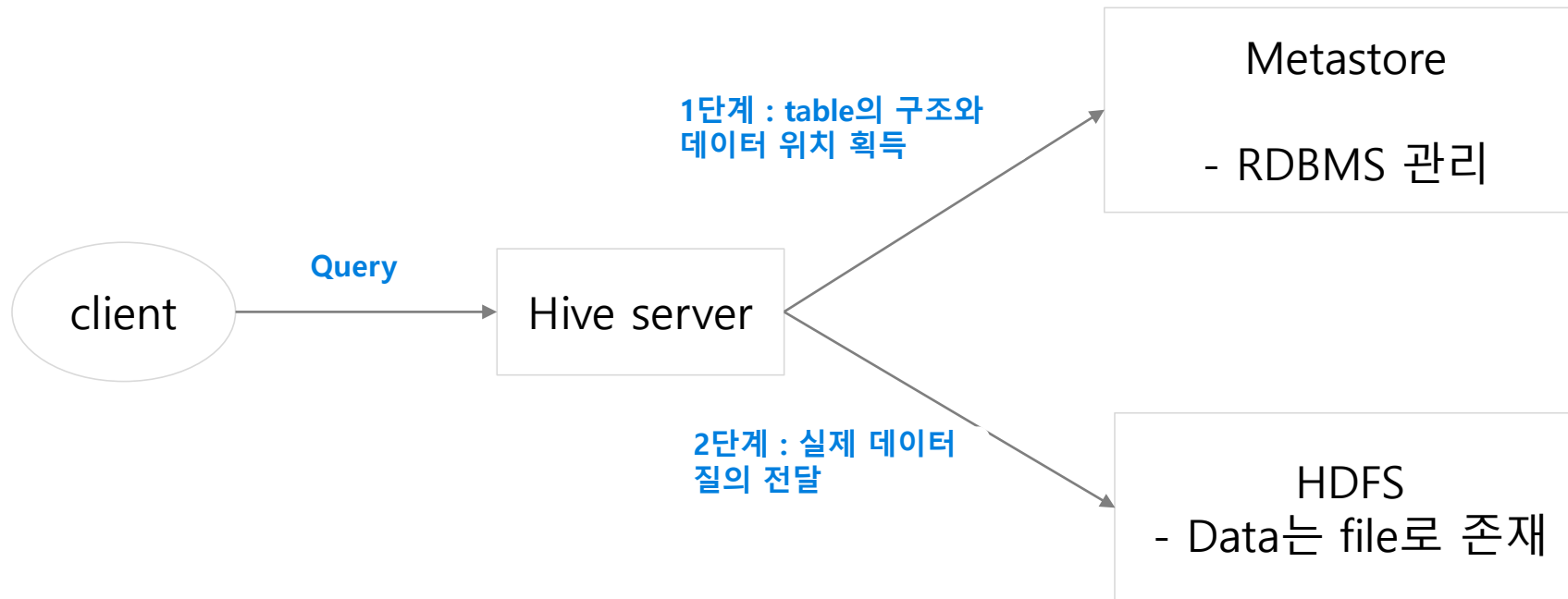
# 하이프 쿼리 인터페이스와 데이터 처리 과정

- 하이브 Query 명령어 인터페이스



# 하이프 쿼리 인터페이스와 데이터 처리 과정

- 하이브 동작 과정
  - 사용자의 HiveQL 명령어를 해석하여 맵리듀스 작업으로 변환
  - 메타스토어에서 테이블 구조와 데이터 위치를 얻음
  - 실제 데이터 질의 전달



# 하이브 테이블 관리

- 하이브 데이터 타입
  - 다양한 데이터 타입들 지원

데이터 타입	설명
TINYINT	정수형 1byte
SMALLINT	정수형 2byte
INT	정수형 4byte
BIGINT	정수형 8byte
BOOLEAN	논리형
FLOAT	부동소수형 4byte
DOUBLE	부동소수형 8byte
STRING	문자형
...	...

# 하이프 테이블 관리

- 하이프 데이터 형식

데이터 형식	설명
텍스트 파일	모든 데이터가 유니코드 표준을 사용한 원시 text로 저장됨
시퀀스 파일	데이터가 이진 키-값 쌍으로 저장됨
RC 파일	모든 데이터가 로우(행, row) 기반의 최적화 대신 컬럼기반의 최적화된 형식
ORC[Optimized Row Columnar] 형식	하이프 내부 table에 최적화된 성능 향상을 위한 파일 포맷 하이프의 성능을 크게 높임 하이프에 최적화된 구조
Parquet 형식	하이프, 임팔라, 피그등 다양한 Hadoop 도구와 호환되는 컬러 기반 형식 다양성을 제공해주는 파일 형태



# 하이프 테이블 종류

## 내부 table

하이프가 주체가 되서 관리하는 table

이 table을 삭제하면 하이브 테이블 메타 정보 삭제  
HDFS 에 저장된 실제 데이터 블록도 함께 삭제됨을  
의미

비교적 빠른 성능

## 외부 table

하이프가 HDFS의 블록을 직접 관리하지 않음  
(삭제 권한이 없음)

삭제시 하이브의 메타 정보만 삭제

원시 형태로 저장된 HDFS의 텍스트 데이터는 잔존

- 내장 table 생성 및 삭제

- a. person.tsv 파일 생성 및 hdfs에 저장
- b. hive metastore에 스키마 생성
- c. 검색 - metastore를 통한 존재 여부 확인후  
에 hdfs의 데이터가 검색
- d. 삭제 - metastore에 존재 여부 확인 후에  
schema & hdfs의 데이터 블록을 함께 삭제
- e. hdfs에 person.tsv 파일 자동 삭제됨

```
#hdfs dfs -put /user/hive/warehouse/
```

```
hive>create table person(  
name string,  
age int  
)  
row format delimited fields terminated by '\t'  
stored as textfile  
location '/user/hive/warehouse/;
```

```
#hdfs dfs -put /user/hive/warehouse/
```

```
hive>select * from person;  
hive>drop table person;
```

```
#hdfs dfs -ls
```

# 하이프 프로그래밍

- 내장 table 생성 및 삭제

```
hive> create external table person(  
  > name string,  
  > age int  
  > )  
  > row format delimited fields terminated by '\t'  
  > stored as textfile  
  > location '/user/hive/warehouse';  
OK  
Time taken: 0.149 seconds  
hive> select * from person;  
OK  
송 병 길      46  
서 동 호      33  
이 해 인      27  
이 건 훈      29  
Time taken: 0.157 seconds, Fetched: 4 row(s)  
hive> drop table person;  
OK
```

- 외장 table 생성 및 삭제

- a. person.tsv 파일 생성 및 hdfs에 저장
- b. hive metastore에 스키마 생성
- c. 검색 - metastore를 통한 존재 여부  
확인후에 hdfs의 데이터가 검색
- d. 삭제 - metastore에 삭제
- e. hdfs에 person.tsv 파일 잔존

```
#hdfs dfs -put /user/hive/warehouse/
```

```
hive>create external table person(  
name string,  
age int  
)  
row format delimited fields terminated by '\t'  
stored as textfile  
location '/user/hive/warehouse/;
```

```
#hdfs dfs -put /user/hive/warehouse
```

```
hive>select * from person;  
hive>drop table person;
```

```
#hdfs dfs -ls
```

# 하이프 프로그래밍

- 외장 table 생성 및 삭제

```
hive>  
  > create external table person(  
  > name string,  
  > age int  
  > )  
  > row format delimited fields terminated by '\t'  
  > stored as textfile  
  > location '/user/hive/warehouse';  
OK  
Time taken: 0.119 seconds  
hive> select * from person;  
OK  
송 병 길      46  
서 동 호      33  
이 해 인      27  
이 건 훈      29  
Time taken: 0.173 seconds, Fetched: 4 row(s)
```

# 하이프 프로그래밍

- 내부 table 생성 후 insert
  - Hdfs 구조 이해하기
- hive>create database datas;
- hive>show databases;
- hive>use datas;
- Table create 시  
hdfs에 자동으로 datas.db생성  
(hive의 databases명 기준으로 자동 생성)

```
hive> show databases;
OK
default
Time taken: 2.728 seconds, Fetched: 1 row(s)
hive> create database datas;
OK
Time taken: 5.516 seconds
hive> show databases;
OK
datas
default
Time taken: 0.067 seconds, Fetched: 2 row(s)
hive> use datas;
OK
Time taken: 0.249 seconds
hive> create table customers(
    > id bigint,
    > name string,
    > address string
    > );
OK
Time taken: 0.959 seconds
hive> describe customers;
OK
id                bigint
name              string
address           string
Time taken: 0.564 seconds, Fetched: 3 row(s)
hive> select * from customers;
OK
Time taken: 1.327 seconds
```

# 하이프 프로그래밍

- Insert
  - Map reduce 자동 실행

```
hive> insert into customers values
> (11, "재 석 ", "서 울 "),
> (22, "호 동 ", "서 울 ");
Query ID = root_20181101175252_28e263f1-e3a0-4e5f-bbaa-11c141427e24
Total jobs = 3
Launching Job 1 out of 3
Number of reduce tasks is set to 0 since there's no reduce operator
Starting Job = job_1541118527729_0001, Tracking URL = http://quickstart.cloudera:8088/proxy/application_1541118527729_0001/
Kill Command = /usr/lib/hadoop/bin/hadoop job -kill job_1541118527729_0001
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 0
2018-11-01 17:52:53,034 Stage-1 map = 0%, reduce = 0%
2018-11-01 17:53:25,037 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 5.8 sec
MapReduce Total cumulative CPU time: 5 seconds 800 msec
Ended Job = job_1541118527729_0001
Stage-4 is selected by condition resolver.
Stage-3 is filtered out by condition resolver.
Stage-5 is filtered out by condition resolver.
Moving data to: hdfs://quickstart.cloudera:8020/user/hive/warehouse/datas.db/customers/.hive-staging_hive_2018-11-01_17-52-01_945_4320487086912569018-1/-ext-10000
Loading data to table datas.customers
Table datas.customers stats: [numFiles=1, numRows=2, totalSize=34, rawDataSize=32]
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1 Cumulative CPU: 5.8 sec HDFS Read: 4081 HDFS Write: 105 SUCCESS
Total MapReduce CPU Time Spent: 5 seconds
OK
Time taken: 85.733 seconds
hive> select * from customers;
OK
11 재 석 서 울
22 호 동 서 울
Time taken: 0.149 seconds, Fetched: 2 row(s)
```

## Browse Directory

/user/hive/warehouse/datas.db/customers							Go!
Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rwxrwxrwx	root	supergroup	34 B	Fri Nov 02 09:53:23 +0900 2018	1	128 MB	000000_0

# 하이프 프로그래밍

- Insert 추가시

```
hive> insert into customers values  
> (33, "동업", "일산"),  
> (33, "동업", "분당"),  
> (55, "찬우", "부산");
```

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rwxrwxrwx	root	supergroup	34 B	Fri Nov 02 09:53:23 +0900 2018	1	128 MB	000000_0
-rwxrwxrwx	root	supergroup	51 B	Fri Nov 02 10:00:46 +0900 2018	1	128 MB	000000_0_copy_1

```
[root@quickstart hivetest]# hdfs dfs -cat /user/hive/warehouse/datas.db/customers/000000_0  
11재 석 서 울  
22호 동 서 울  
[root@quickstart hivetest]# hdfs dfs -cat /user/hive/warehouse/datas.db/customers/000000_0_copy_1  
33동 업 일 산  
33동 업 분 당  
55찬 우 부 산
```



# 하이프 프로그래밍

- group by

```
hive> select address, count(*) from customers group by address;
Query ID = root_20181101183535_fa0c859a-22da-410b-be2d-b3512c7e46ed
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks not specified. Estimated from input data size: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapreduce.job.reduces=<number>
Starting Job = job_1541118527729_0003, Tracking URL = http://quickstart.cloudera:3/
Kill Command = /usr/lib/hadoop/bin/hadoop job -kill job_1541118527729_0003
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
2018-11-01 18:36:06,512 Stage-1 map = 0%, reduce = 0%
2018-11-01 18:36:32,173 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 4.23 sec
2018-11-01 18:37:02,295 Stage-1 map = 100%, reduce = 100%, Cumulative CPU 9.39 sec
MapReduce Total cumulative CPU time: 9 seconds 390 msec
Ended Job = job_1541118527729_0003
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1 Reduce: 1 Cumulative CPU: 9.39 sec HDFS Read: 7844 HDFS Write: 1024
Total MapReduce CPU Time Spent: 9 seconds 390 msec
OK
부 산      1
분 당      1
서 을      2
일 산      1
Time taken: 88.551 seconds, Fetched: 4 row(s)
```