2017년 정보처리학회 단기강좌

빅데이터 플랫폼과 Spark

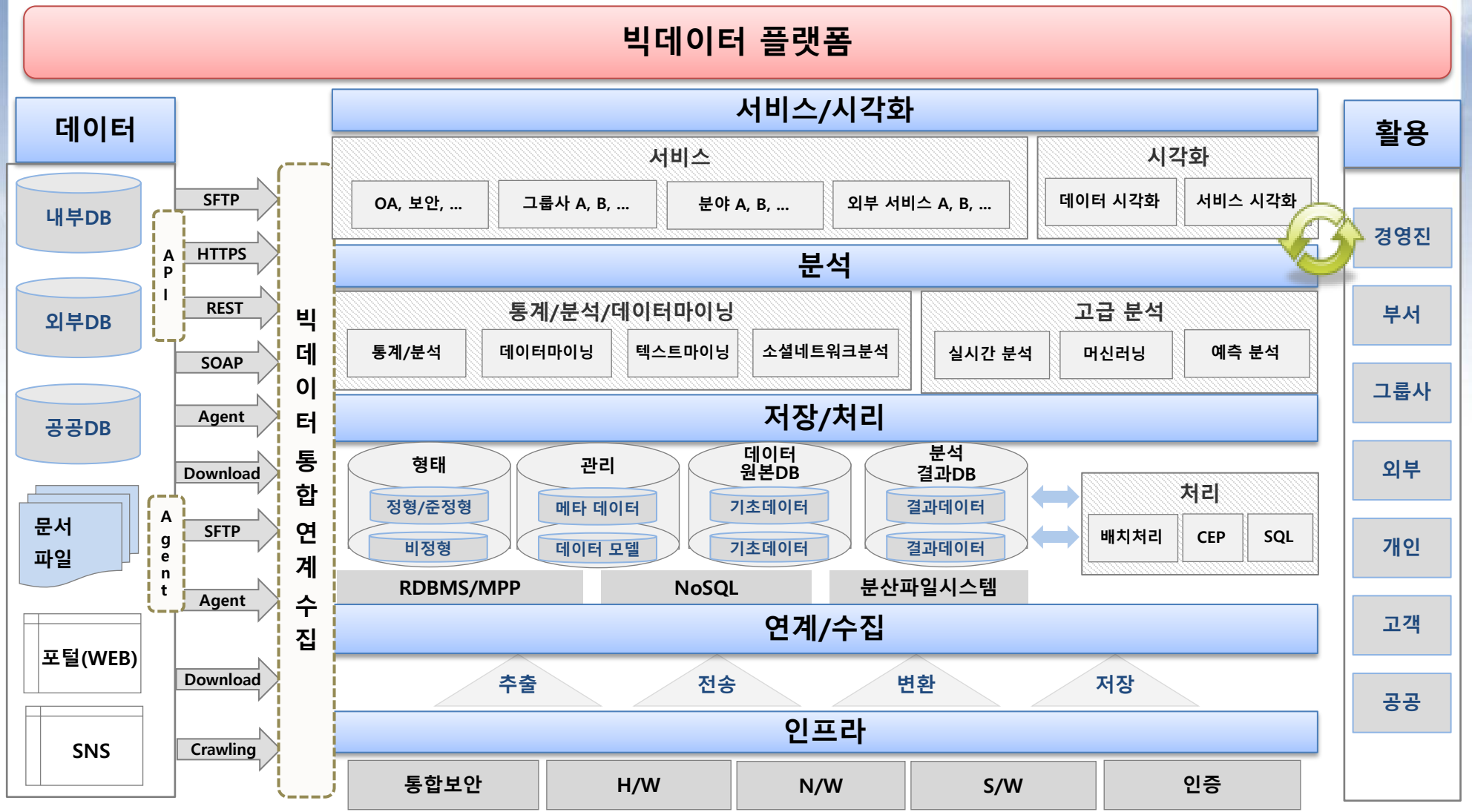
- 장형석
- 국민대학교 빅데이터경영MBA과정 교수
- chjang1204@nate.com

Part I

빅데이터 플랫폼

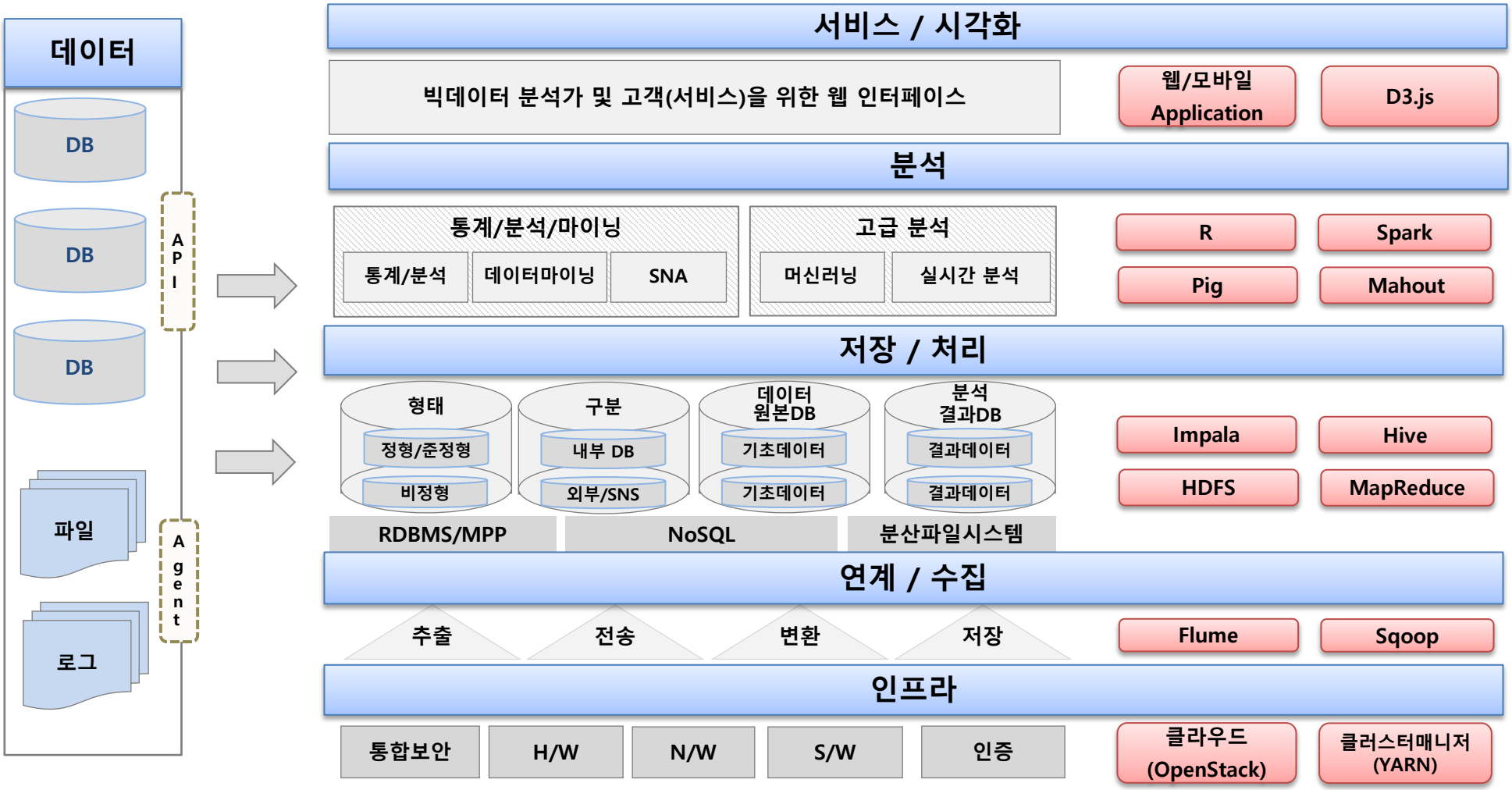
1. 빅데이터 플랫폼

- 빅데이터 수집/분석/서비스를 위한 목표 플랫폼



1. 빅데이터 플랫폼

- 빅데이터 플랫폼과 소프트웨어



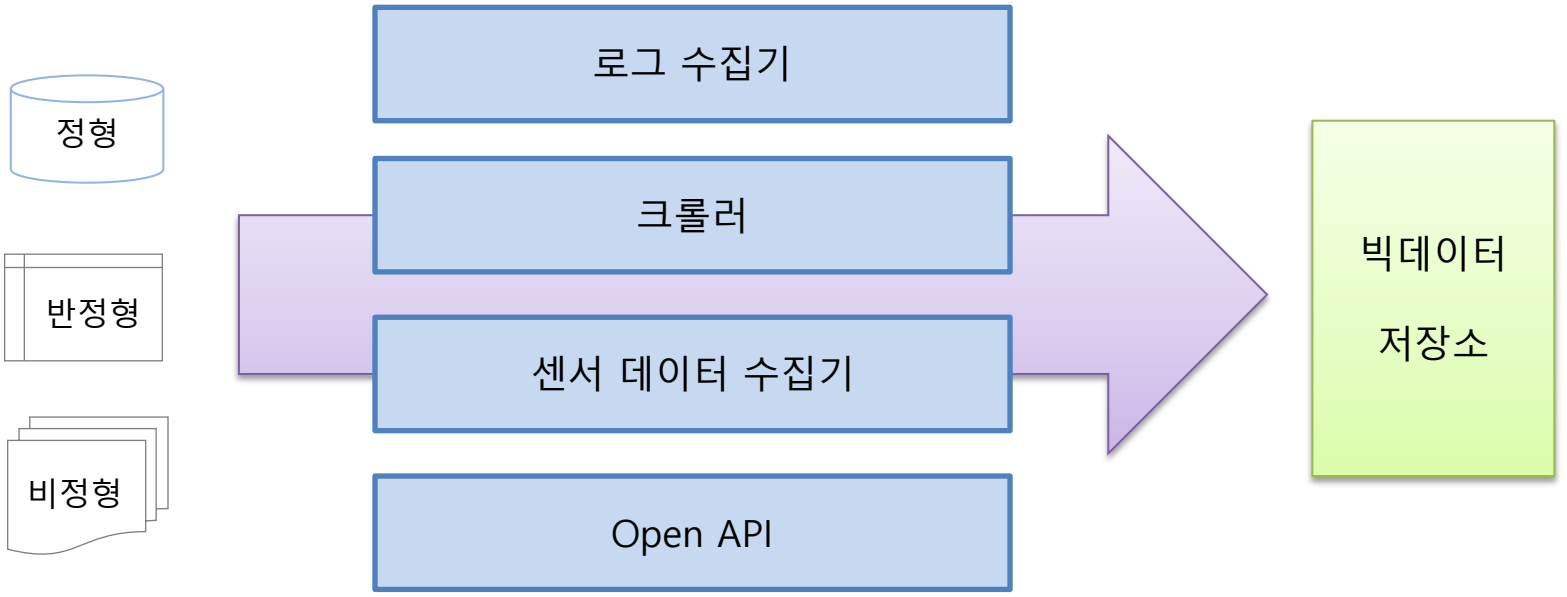
2. 빅데이터 프로세스 및 기술

- **빅데이터 프로세스**

데이터소스	수집	저장	처리	분석	표현
내부 데이터	수동	- 정형 - 비정형 - 준정형	배치처리 (맵리듀스)	전처리	서비스(활용) 시각화
외부 데이터	자동 - 로그수집기 - 크롤러 - 센서수집기	분산파일시스템 **NoSQL**	**SQL**분석	통계 데이터마이닝 머신러닝	데이터(분석) 시각화
사물인터넷 (센서)			실시간처리		

2. 빅데이터 프로세스 및 기술

- **수집 개요 및 기술**



<빅데이터의 주요 수집 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

2. 빅데이터 프로세스 및 기술

● 저장 기술

구분	기술	최초 개발	주요 기능 및 특징
분산파일시스템	HDFS	아파치	대표적인 오픈소스 분산파일시스템
	Hive	페이스북	HDFS기반의 DataWarehouse
	S3	아마존	아마존의 클라우드 기반 분산 스토리지 서비스
NoSQL	HBase	아파치	HDFS기반의 NoSQL
	Cassandra	A. Lakshman	ACID 속성을 유지한 분산 데이터베이스
	Mongo DB	10gen	DB의 수평 확장 및 범위 질의 지원, 자체 맵리듀스

● 처리 기술

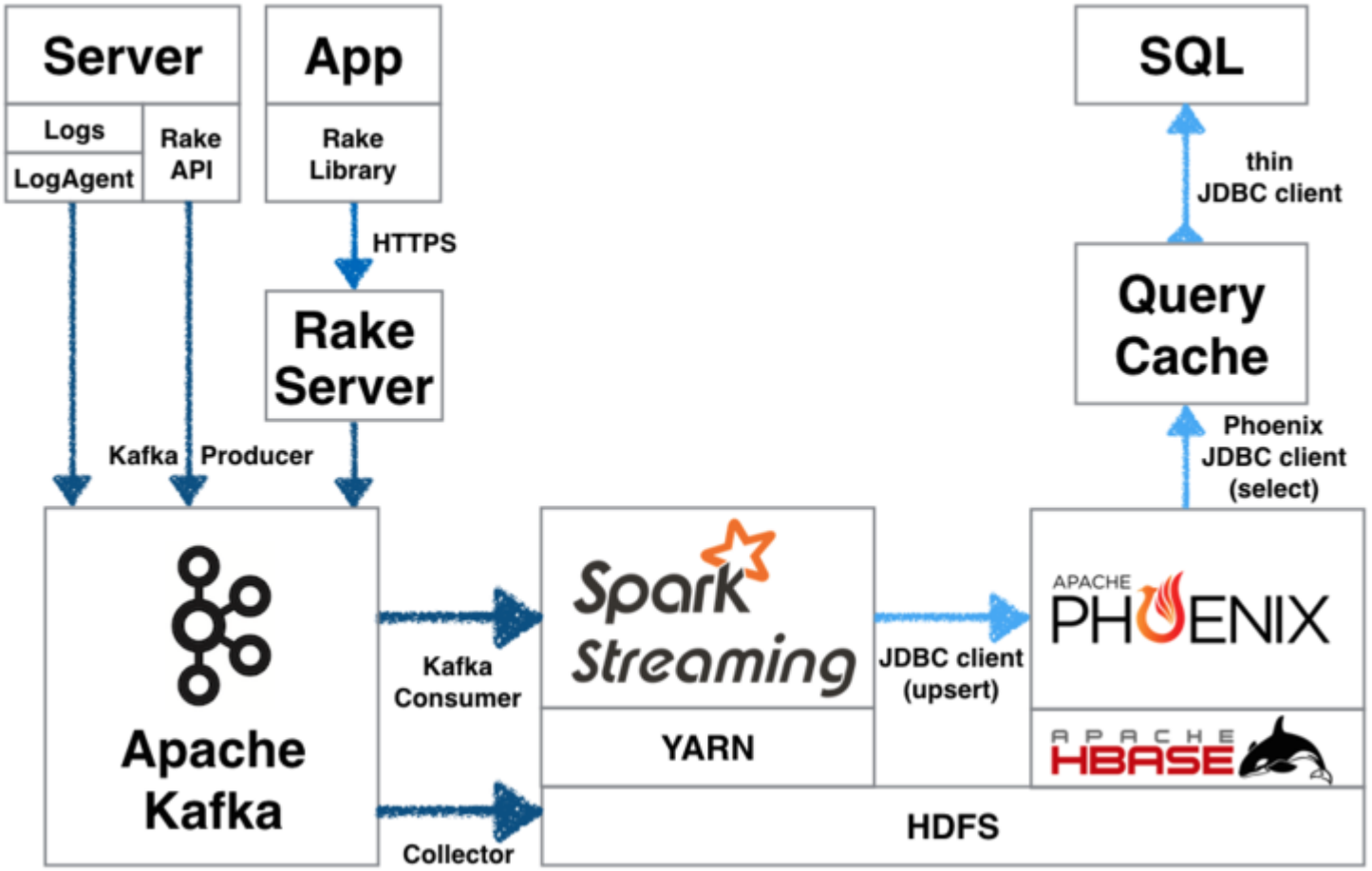
- 배치 처리 : 하둡 맵리듀스, Pig, Hive
  - 분산 병렬 데이터 처리 기술의 표준, 일반 범용 서버로 구성된 군집화 시스템을 기반으로 <키,값> 입력 데이터 분할 처리 및 처리 결과 통합 기술, job 스케줄링 기술, 작업 분배 기술, 태스크 재수행 기술이 통합된 분산 컴퓨팅 기술
- SQL on Hadoop : Hive on Tez, Impala, Presto, Shark(SparkSQL)
  - 배치처리 중심의 맵리듀스의 한계를 넘기 위해 만들어진 SQL 기반의 자체 쿼리 실행 엔진.

2. 빅데이터 프로세스 및 기술

- 분석 기술
  - R
    - 오픈소스 통계분석 소프트웨어. 기본적인 통계분석부터 최신 머신러닝까지 다양한 패키지를 지원.
    - 하둡 및 스팍과 연동이 가능
    - R, RStudio, RHadoop, RHive , SparkR
  - Python
    - 데이터마이닝과 머신러닝을 지원하는 다양한 패키지
    - 통계학과의 기본적인 프로그래밍 언어로 정착
    - 스크립트 기반의 대화형 분석 환경 지원
    - iPython Notebook
  - Mahout
    - 추천시스템, 분류, 군집 등 머신러닝 기능을 지원
    - 하둡 기반의 머신러닝 Java 라이브러리
    - 현재 Spark의 MLlib로 발전함
  - Spark
    - 대화형 분석, 머신러닝, SQL, 그래프알고리즘 등 다양한 분석 가능
    - Spark Core, Spark MLlib, SparkSQL, GraphX, Spark Streaming

2. 빅데이터 프로세스 및 기술

- 실시간 수집 및 처리 사례



• 출처 : 엄태욱, 2015/12/18, SK플래닛 기술 블로그 http://readme.skplanet.com/?p=12465

3. 빅데이터 인프라

< 개발/파일럿 시스템 구성 >

수집 서버

x86
2CPU, 64G RAM
36TB SATA HDD
Linux
웹크롤러
수집 애플리케이션

분석 서버
(하둡/스팍 클라이언트)

x86
2CPU, 64G RAM
36TB SATA HDD
Linux
Hadoop Ecosystem Clients(Hive,Spark등)

분석 서버
(RStudio Server)

x86
2CPU, 64G RAM
36TB SATA HDD
Linux
R(Linux Server Version)

마스터 노드

마스터 노드 1

x86
2CPU, 96G RAM
4TB SAS HDD
Linux
NameNode(Acitve)
Metastore(Mysql Server)

마스터 노드 2

x86
2CPU, 96G RAM
4TB SAS HDD
Linux
2st NameNode(Stanby)
YARN/Spark Master

데이터 노드

데이터 노드 1

x86
2CPU, 64G RAM
36TB SATA HDD
Linux
DataNode
TaskTracker
Spark Worker

데이터 노드 2

x86
2CPU, 64G RAM
36TB SATA HDD
Linux
DataNode
TaskTracker
Spark Worker

데이터 노드 3

x86
2CPU, 64G RAM
36TB SATA HDD
Linux
DataNode
TaskTracker
Spark Worker

3. 빅데이터 인프라

하둡 클러스터

마스터 노드

[마스터 노드 1] **[마스터 노드 2]** **마스터 노드 3**

x86	x86	x86
2CPU, 96G RAM	2CPU, 96G RAM	2CPU, 96G RAM
4TB SAS HDD	4TB SAS HDD	4TB SAS HDD
Linux	Linux	Linux
NameNode(active)	2st NameNode(Stanby)	Cluster Manager
Metastore(Mysql)	YARN/Spark Master	Backup

데이터 노드

[데이터 노드 1] **[데이터 노드 2]** **[데이터 노드 3]**

x86
2CPU, 64G RAM
36TB SATA HDD
Linux
DataNode
TaskTracker
Spark Worker

데이터 노드 4 **데이터 노드 5**

데이터 노드 6 **데이터 노드 7**

데이터 노드 8 **데이터 노드 9** **데이터 노드 10**

수집용

[수집 서버 1]

x86
2CPU, 64G RAM
36TB SATA HDD
ETL
Linux

수집 서버 2

x86
2CPU, 64G RAM
36TB SATA HDD
ETL
Linux

크롤링 서버

x86
2CPU, 64G RAM
36TB SATA HDD
Crawler
LInux

분석용

[분석 서버 1]

x86
2CPU, 64G RAM
36TB SATA HDD
Hadoop Clients(Hive등)
Linux

[분석 서버 2]

x86
2CPU, 64G RAM
36TB SATA HDD
RStudio Server
Linux

시각화 서버

x86
2CPU, 64G RAM
36TB SATA HDD
Tableau Server
Linux

서비스용

웹 서버 1

x86
2CPU, 64G RAM
8TB SATA HDD
WebServer/WAS

웹 서버 2

x86
2CPU, 64G RAM
8TB SATA HDD
WebServer/WAS

Part II

하둡 에코시스템

1. 분산병렬처리

● 처리 속도의 발전

- 1단계 : CPU의 Clock Frequency 늘리기
  - 무어의 법칙 : 동일한 면적의 반도체 직접회로의 트랜지스터의 개수가 **18개월에 2배씩 증가**
  - CPU Clock 물리적인 한계 : **최대 4Ghz**

- 2단계 : Single CPU에서 Multi CPU & Core
  - 1 CPU -> **2 CPU(최적)** -> 4 CPU -> 8 CPU
  - 1 CPU : 1 Core -> 2 Core -> 4 Core -> 6 Core -> **8 Core**

2배가 아닌
10배씩 증가

- 3단계 : Single Machine -> Cluster(N대의 Machine을 Network로 연결)
  - 수퍼컴퓨터
  - 하둡 클러스터 -> N대의 범용 컴퓨터를 묶은 클러스터(하둡 HDFS와 맵리듀스)

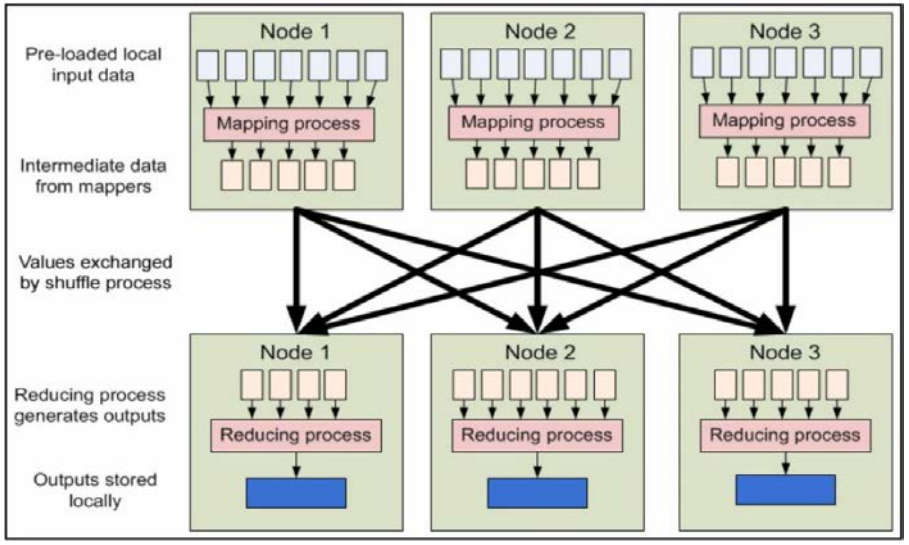
1. 분산병렬처리

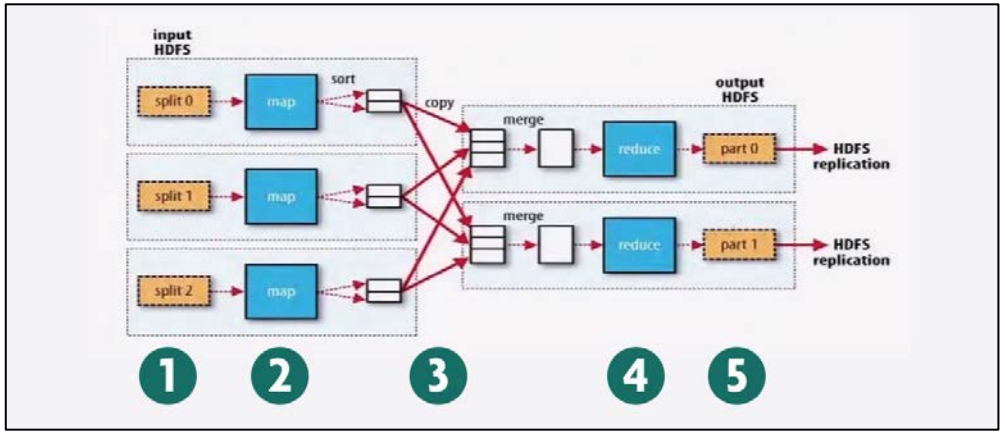
- **분산병렬처리를 위한 맵리듀스 프레임워크**
  - 구글
    - 2003년 GFS : 구글 분산 파일 시스템 아키텍처 -> 논문 공개
    - 2004년 MapReduce : 구글 맵리듀스 아키텍처 -> 논문 공개
  - 하둡
    - 2006년 2월 더그커팅 : 오픈소스로 하둡(HDFS,MapReduce) 공개
    - 2008년 야후 : 900노드에서 1테라바이트 정렬 : 209초(세계최고기록)
    - **Hadoop**, Pig, Hive, HBase, Mahout, Zookeeper .... **Spark** : 하둡에코시스템 및 스팍으로 발전

오픈소스 | 범용 컴퓨터 | 수집/저장/처리/분석/시각화 기술

1. 분산병렬처리

- 하둡 MapReduce





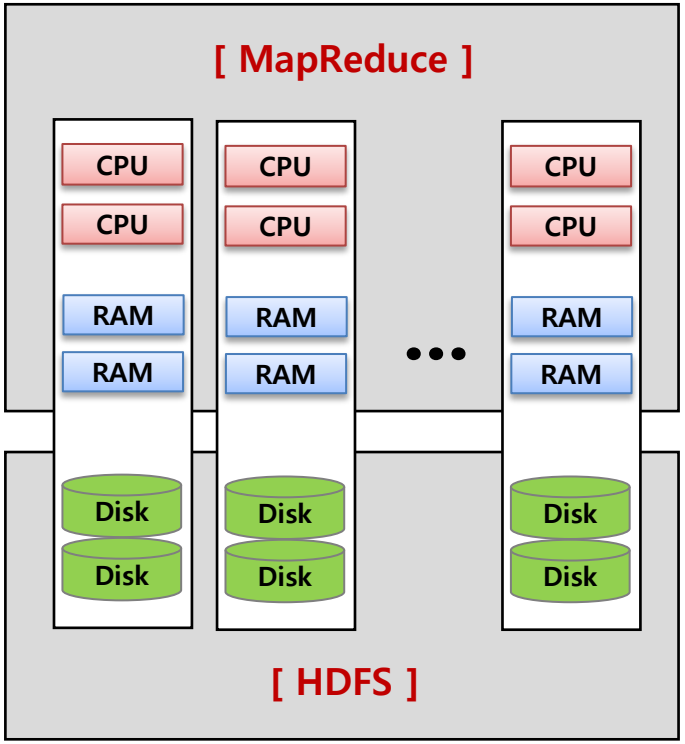
•출처 : https://developer.yahoo.com/hadoop/tutorial/module4.html

https://blog.pivotal.io/pivotal/products/hadoop-101-programming-mapreduce-with-native-libraries-hive-pig-and-cascading

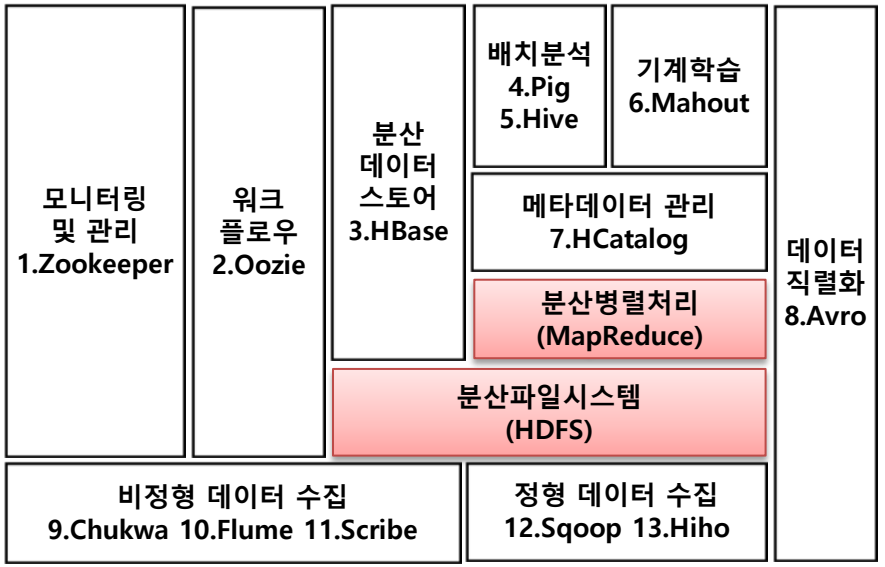
2. Hadoop Ecosystem

빅데이터 핵심 기술 – 하둡

< 분산파일시스템 + 분산병렬처리 >

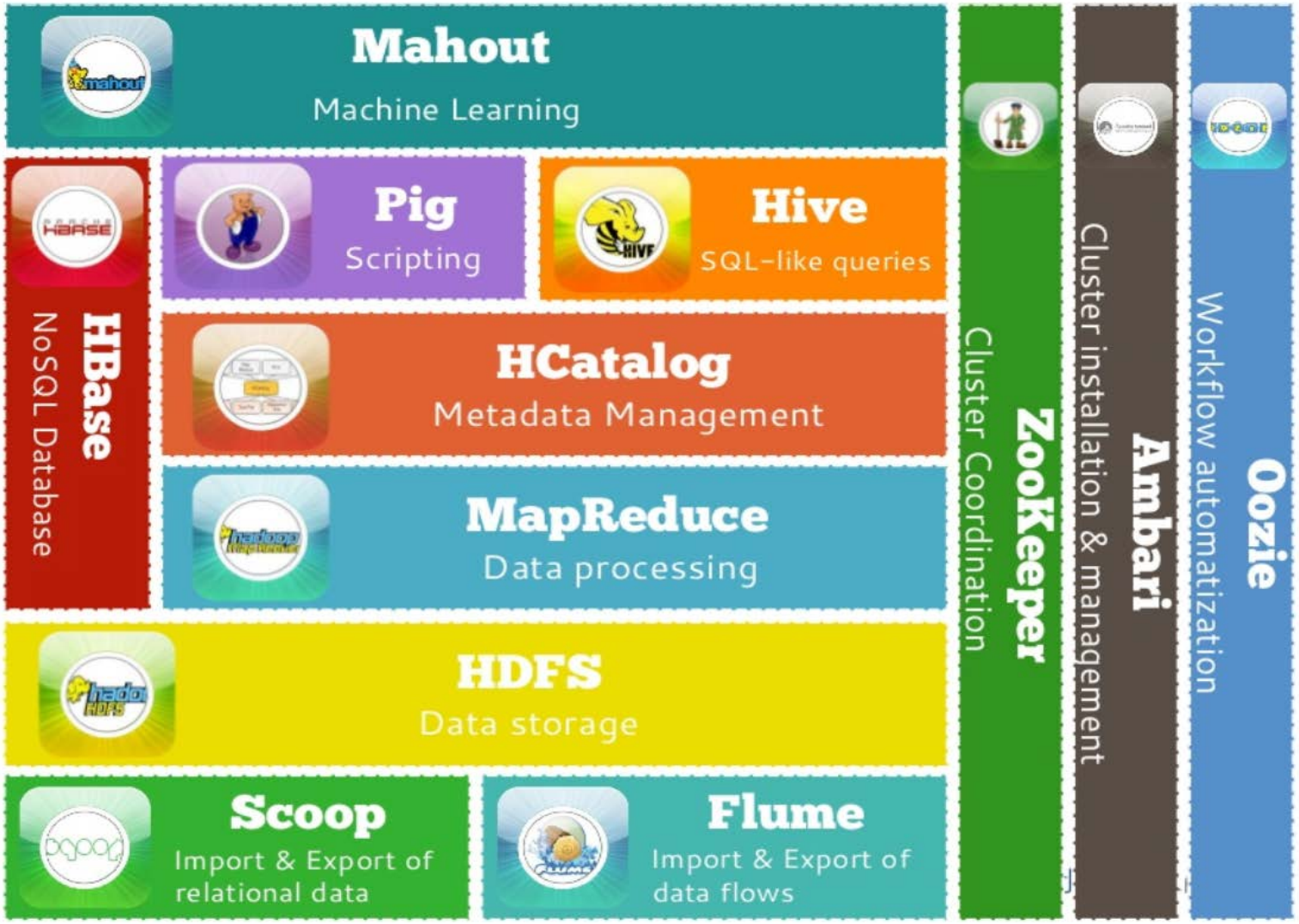


하둡 에코시스템



1. Zookeeper : 분산 환경에서 서버들간에 상호 조정이 필요한 다양한 서비스 제공
2. Oozie : 하둡 작업을 관리하는 워크플로우 및 코디네이터 시스템
3. Hbase : HDFS 기반의 컬럼 NoSQL
4. Pig : 복잡한 MapReduce 프로그래밍을 대체할 Pig Latin 언어 제공
5. Hive : 하둡 기반의 데이터웨어하우스, 테이블단위의 데이터 저장과 SQL 쿼리를 지원
6. Mahout : 하둡 기반으로 데이터 마이닝 알고리즘을 구현한 오픈 소스 라이브러리
7. Hcatalog : 하둡 기반의 테이블 및 스토리지 관리
8. Avro : RPC(Remote Procedure Call)과 데이터 직렬화를 지원하는 프레임워크
9. Chukwa : 분산 환경에서 생성되는 데이터를 HDFS에 안정적으로 저장시키는 플랫폼
10. Flume : 소스서버에 에이전트가 설치,에이전트로부터 데이터를 전달받는 콜랙터로 구성
11. Scribe : 페이스북에서 개발, 데이터 수집 플랫폼, Chukwa와 달리 중앙집중서버로 전송
12. Sqoop : 대용량 데이터 전송 솔루션이며, HDFS, RDBMS, DW,NoSQL등 다양한 저장소에 대용량 데이터를 신속하게 전송할 수 있는 방법 제공
13. Hiho : Sqoop과 같은 대용량 데이터 전송 솔루션이며, 하둡에서 데이터를 가져오기 위한 SQL을 지정할 수 있으며, JDBC 인터페이스를지원

2. Hadoop Ecosystem



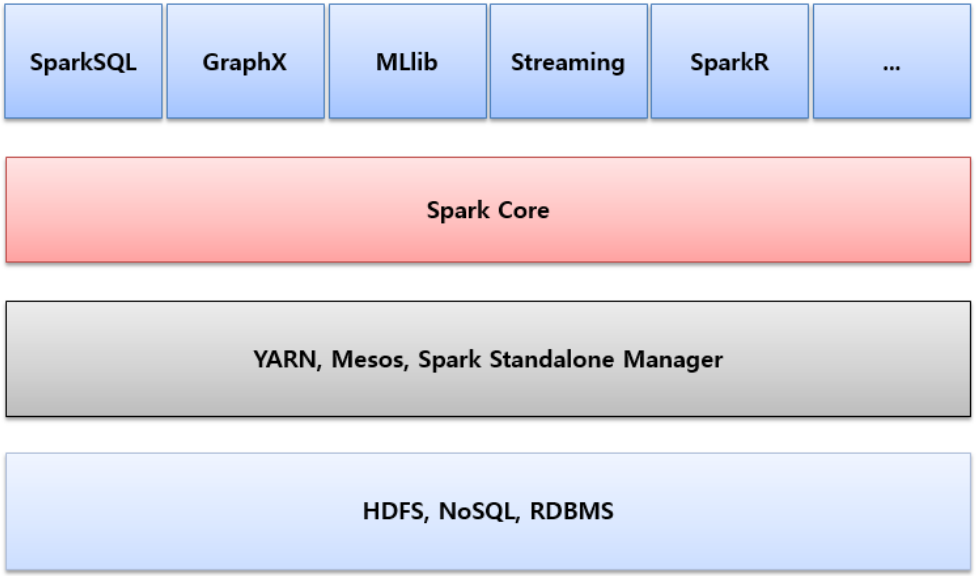
•출처 : 호튼웍스 http://hortonworks.com/

Part III

Spark의 이해

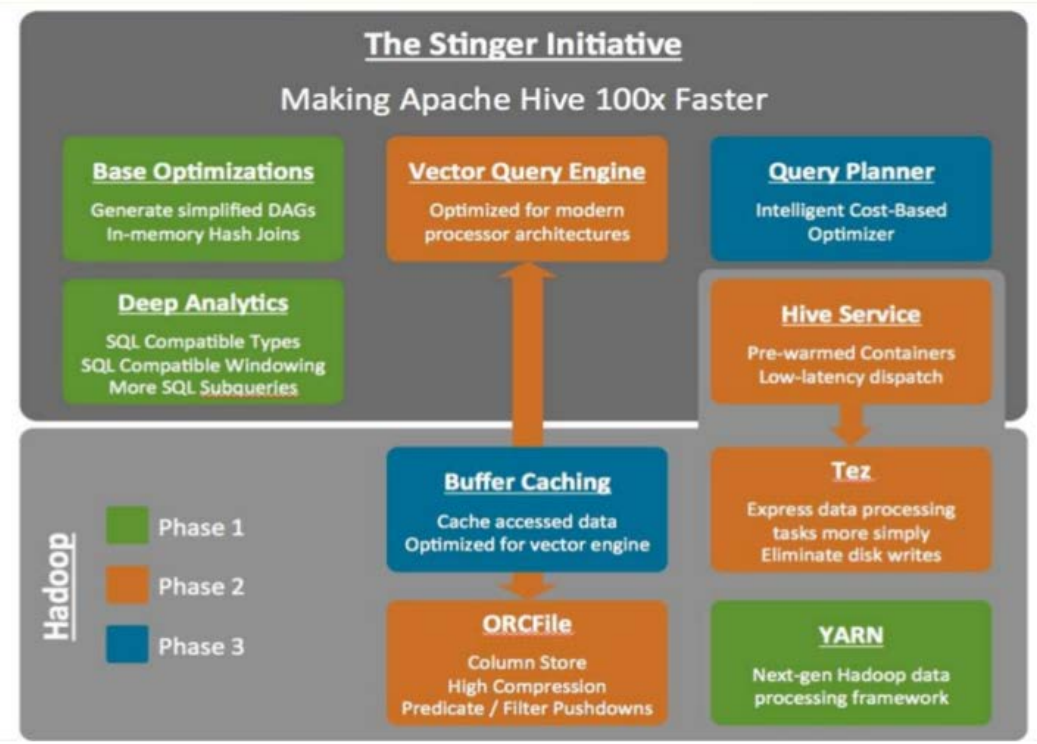
Spark 개요

- **Spark 개요**
  - 하둡 MapReduce 보다 발전된 새로운 분산병렬처리 Framework
    - 저장소는 로컬파일시스템, 하둡 HDFS, NoSQL(Hbase, Redis), RDBMS(오라클,MSSQL)
    - Spark는 분산병렬처리 엔진
  - 기존 Hive, Pig, Mahout, R, Storm 등을 모두 대체 가능
    - 분산병렬처리 엔진인 Spark Core를 기반으로
    - 다양한 내장 패키지를 지원



Spark 개요

- **Spark의 특징**
  - 하둡 에코시스템 발전의 최고봉
    - 구현 원리 및 아키텍처는 Stinger Initiative와 유사
    - 레코드 기반과 컬럼 기반 데이터 저장이 모두 가능
    - YARN, Hive, Sqoop, Kafka 등 다양한 하둡 에코시스템과 통합



Spark 개요

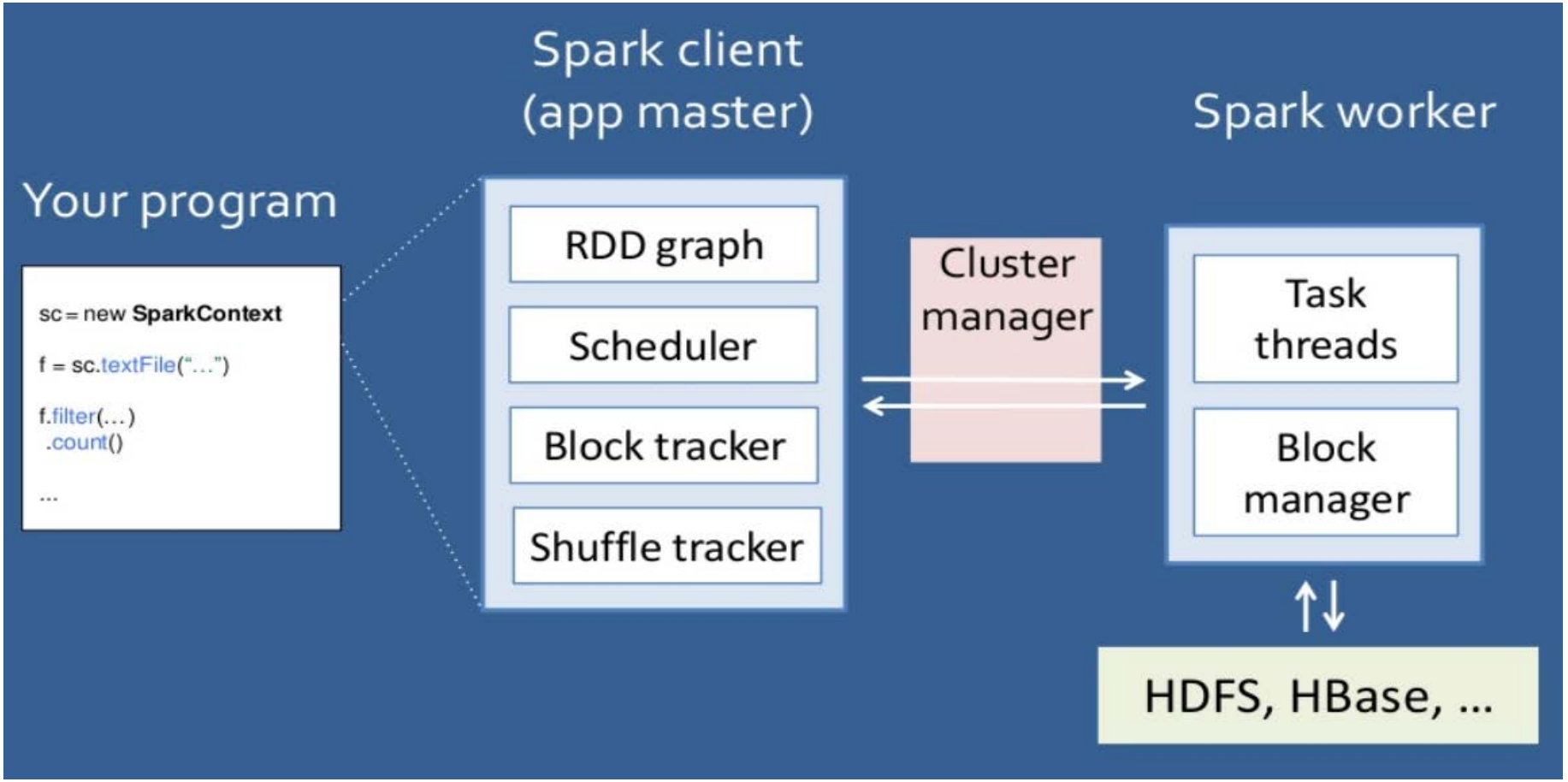
- **Spark vs MapReduce : Word Count**

```
file = spark.textFile("hdfs://...")

file.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)
```

```
package org.myorg;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordCount {

 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable>
{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
 }

 public static class Reduce extends Reducer<Text, IntWritable, Text,
IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
 }

 public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
 }

}
```

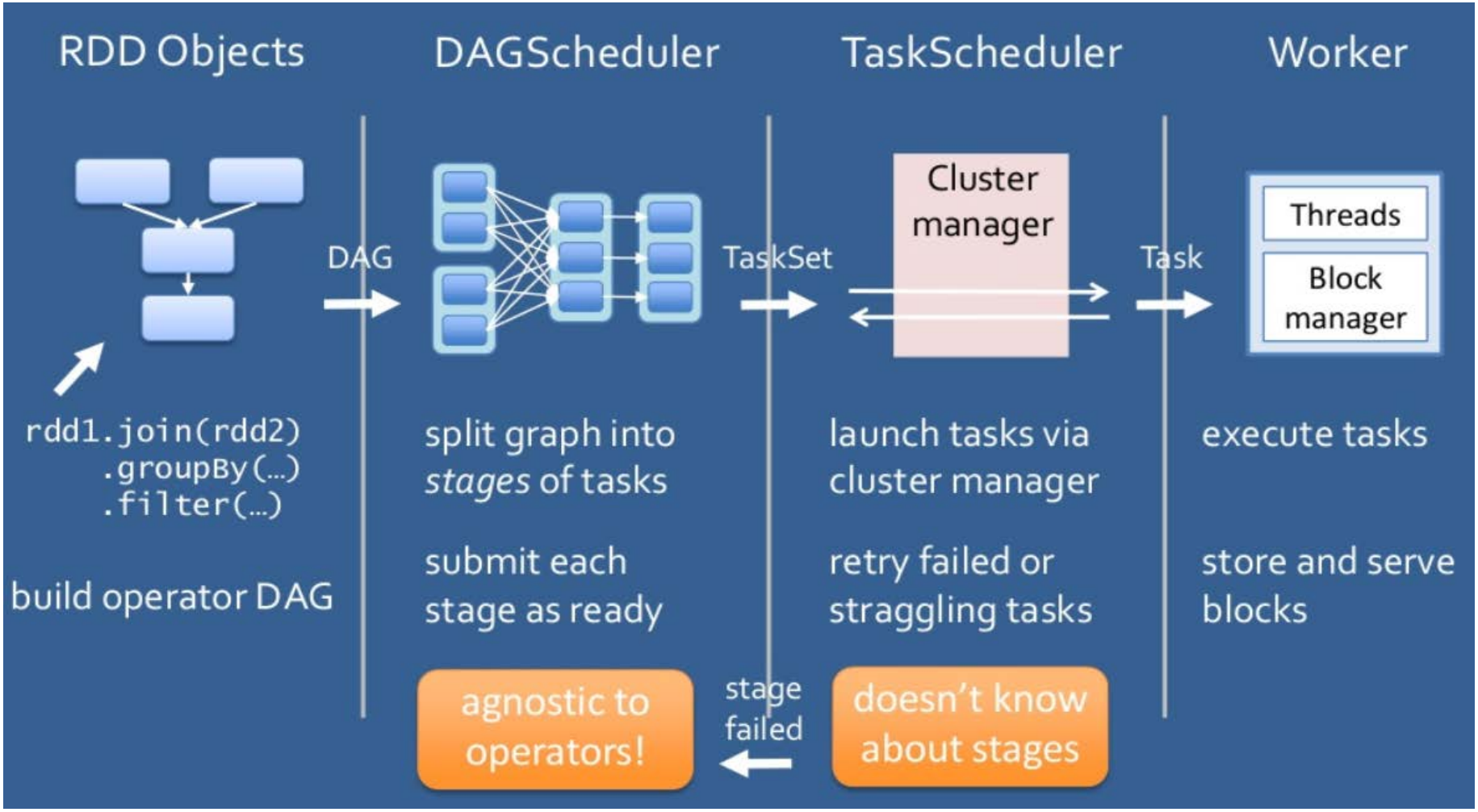
Spark 개요

- 구성요소



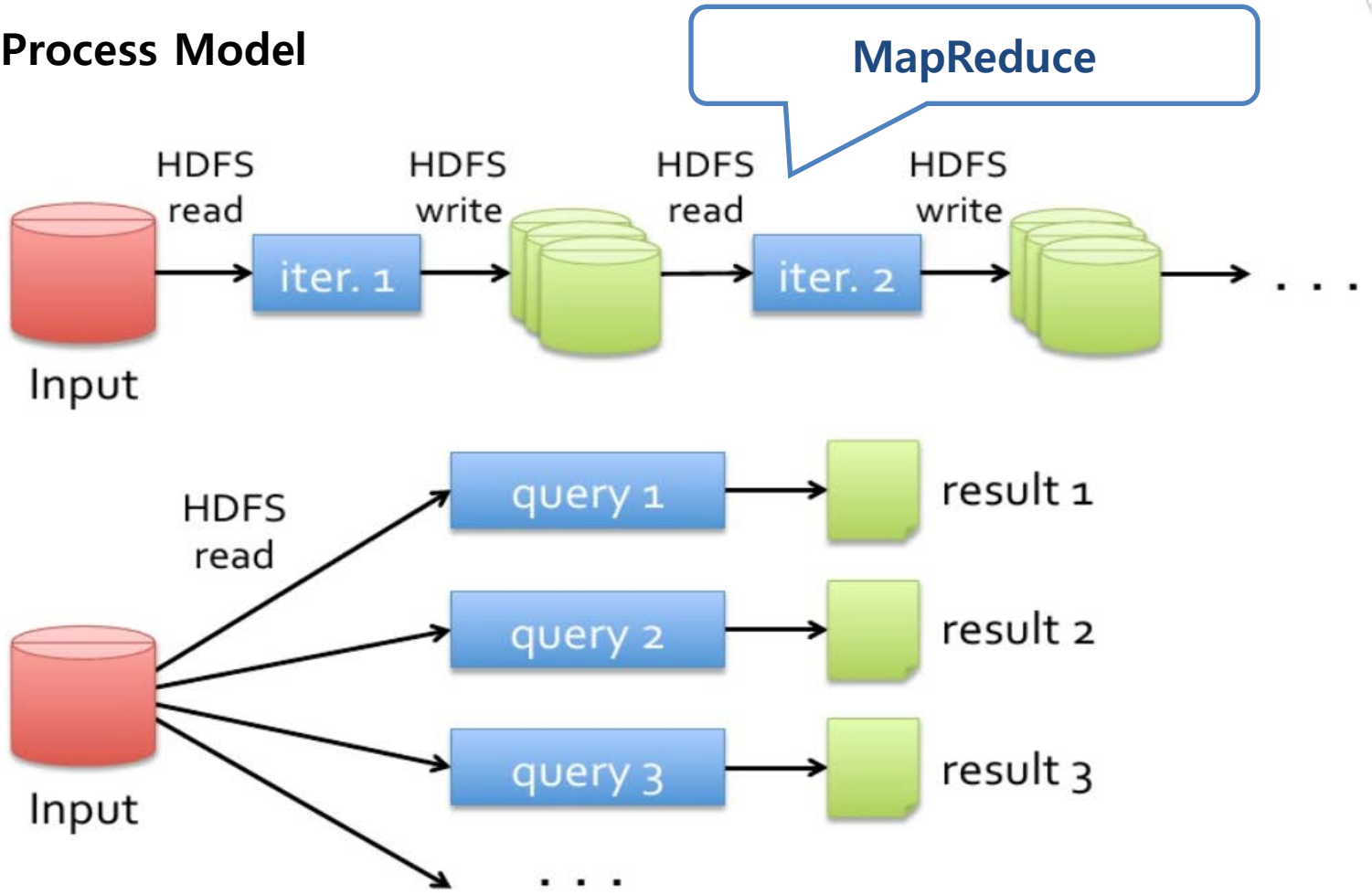
Spark 개요

- **Process & Scheduling**



Spark 개요

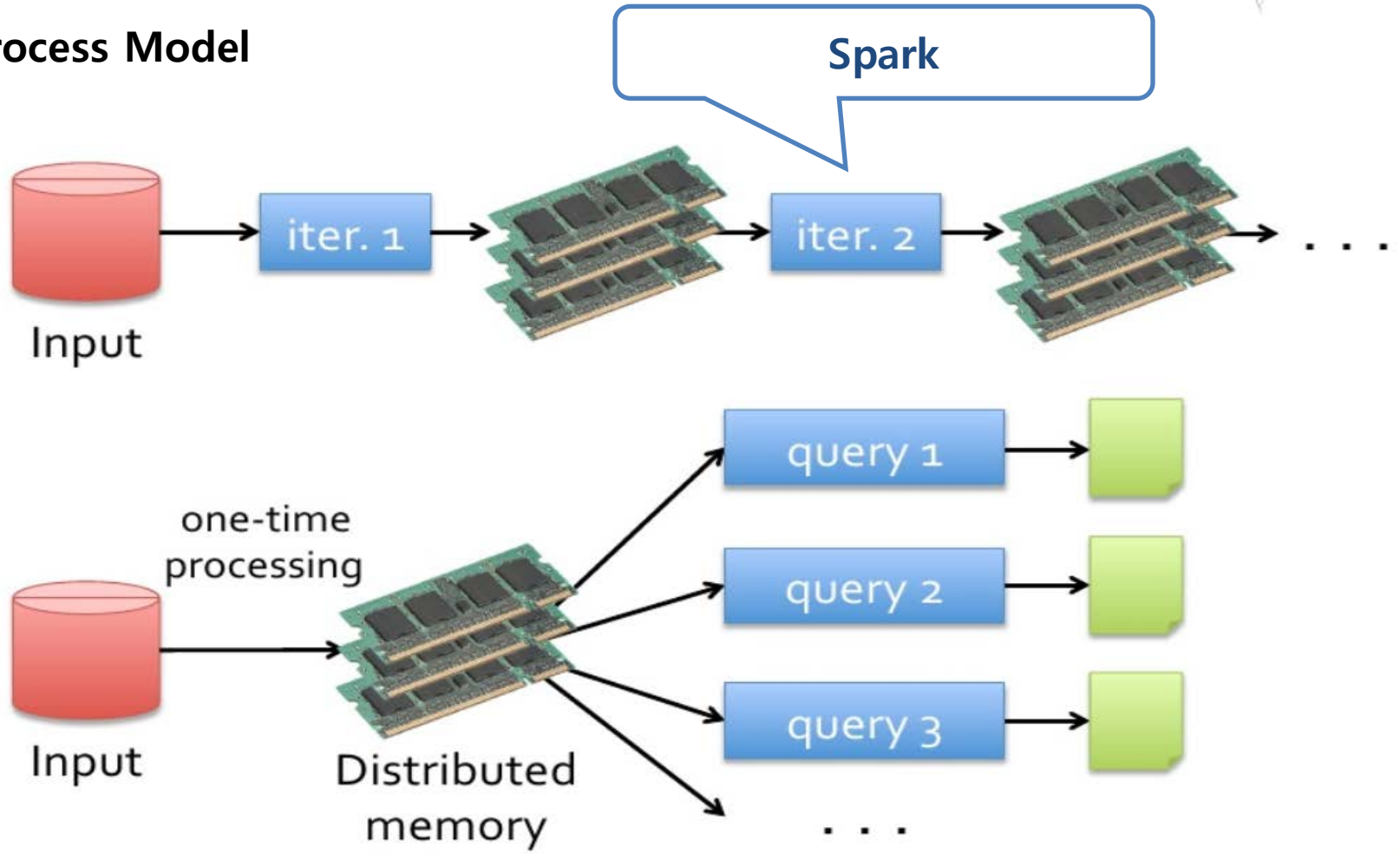
- **Data Process Model**



Slow due to replication, serialization, and disk IO

Spark 개요

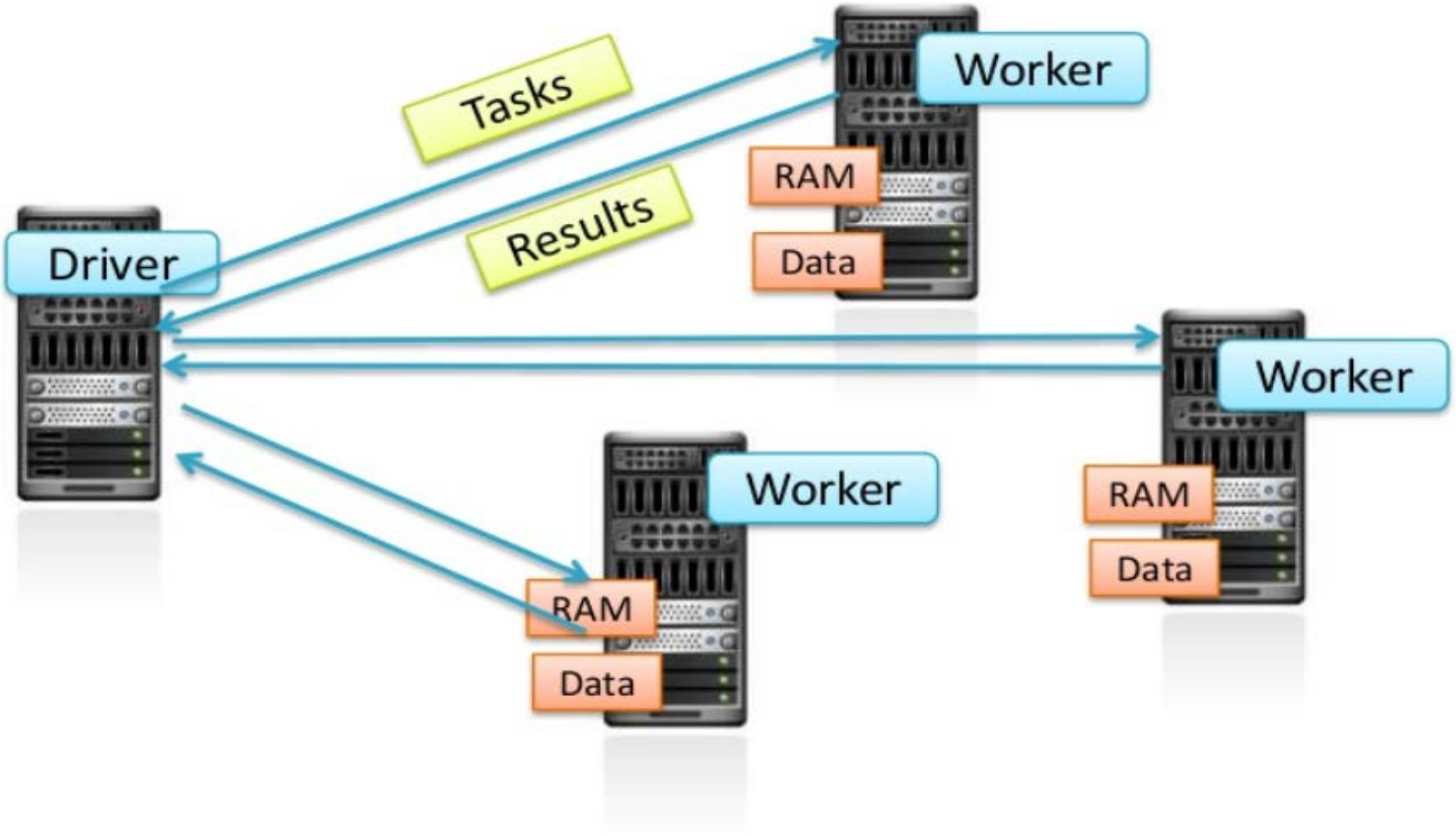
- **Data Process Model**



10-100× faster than network and disk

Spark 개요

- **Spark runs on Hadoop Cluster**



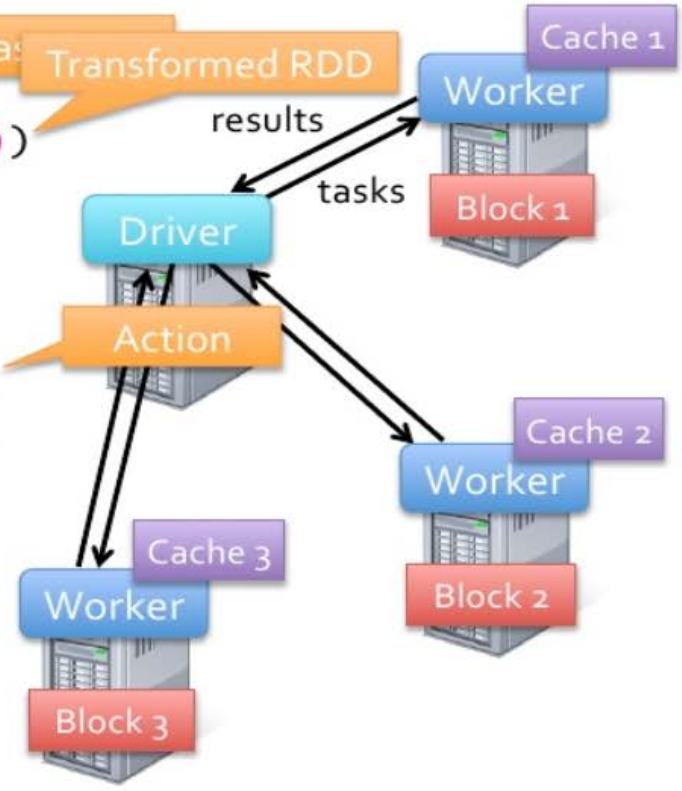
Spark 개요

- **Example : Log Mining**

Load error messages from a log into memory, then interactively search for various patterns

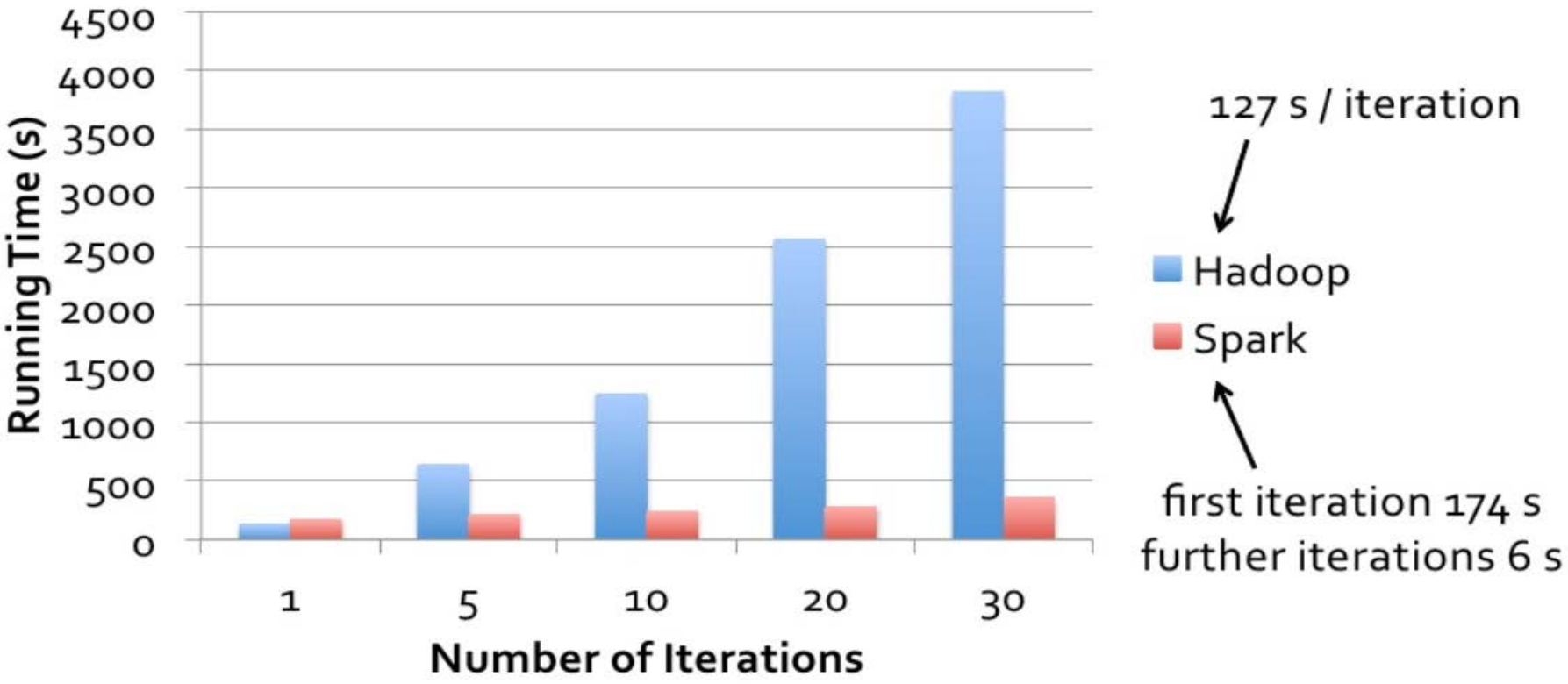
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
. . .
```

Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)



Spark 개요

- 성능 : Logistic Regression



Part IV

Spark 아키텍처

Spark 아키텍처와 구현원리

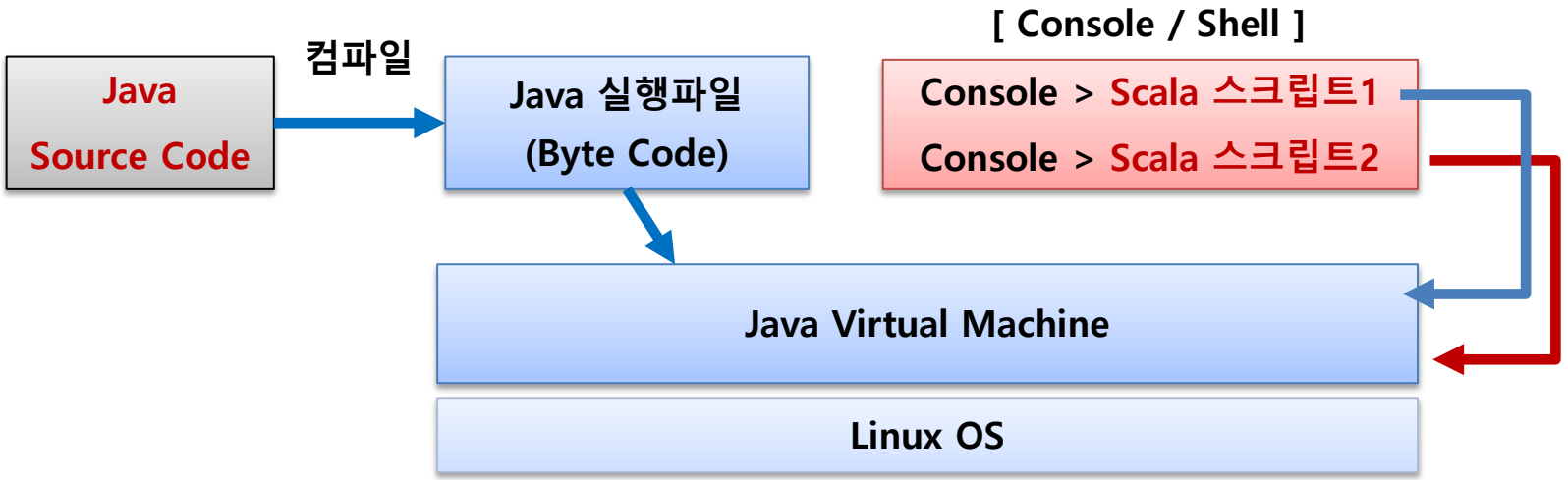
- **함수형 언어 : Scala**
  - 분산병렬처리에 최적화된 함수형 언어의 필요성
    - 구글 MapReduce 프레임워크 : C/C++ 언어
    - 하둡 MapReduce 프레임워크 : Java 6 / Java 7
    - 하둡 Tez 프레임워크 : Java 7 / Java 8
    - Spark Core 프레임워크 : Scala / Java 8
  - 함수형 언어
    - 함수형 언어 vs 절차형 언어 또는 명령형 언어
    - 대표적인 함수형 언어로는 SQL
      # MapReduce의 구현 목표는 분산병렬처리가 가능한 데이터 집계 기능(SQL)
      # HiveQL, SQL on Hadoop(임팔라, 타조, 프레스토 등)
    - Data Work Flow는 DAG(Directed Acyclic Graph, 방향성 비사이클 그래프)
      # Start -> 입력(Input) -> 처리 -> 분기 -> 처리 -> 병합 -> 처리 -> 결과(Output) -> End
    - 입력이 같으면 결과도 같아야 함
    - 단일 머신에서 병렬처리 & 클러스터에서 분산병렬처리가 가능
    - 명령형 언어이면서 함수형 언어의 특징을 가진 대표적인 언어 : R, Java Script, Python

Spark 아키텍처와 구현원리

- **함수형 언어 : Scala**
  - 프로그래밍 아키텍처
    - OS : Linux
    - JVM(자바 가상 머신) : 자바 애플리케이션을 실행하기 위한 가상 머신
    - Java 8 / Scala 프로그래밍 언어 : JVM 기반의 애플리케이션 개발 언어
    - 컴파일 vs 스크립트

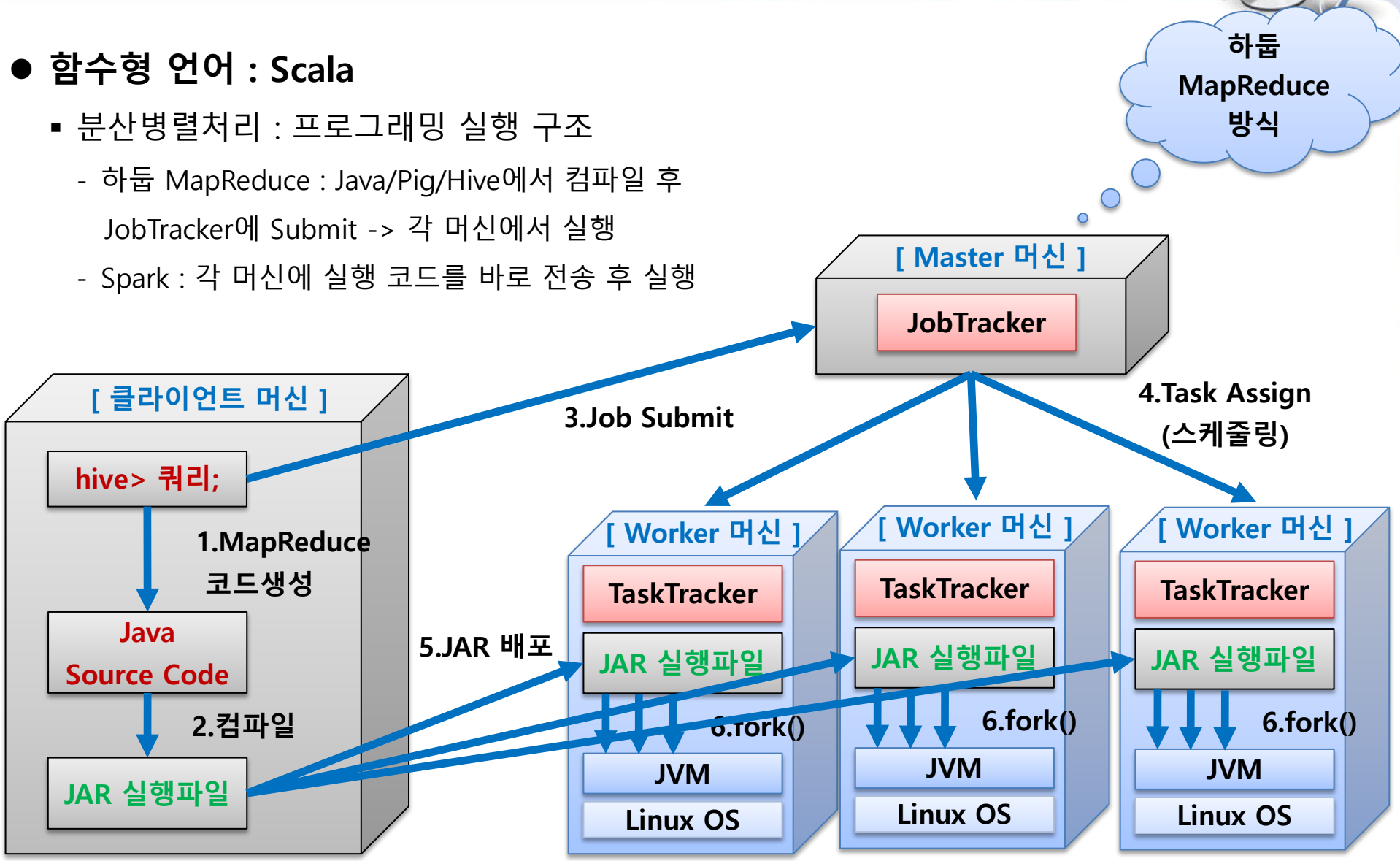
      컴파일 방식 : 컴파일 후 실행파일(JAR)을 각 머신에 배포해야 함(Compile -> Job Submit)

      스크립트 방식 : 컴파일 과정 없이 스크립트를 그 때마다 해석하여 JVM에서 바로 실행(Console)



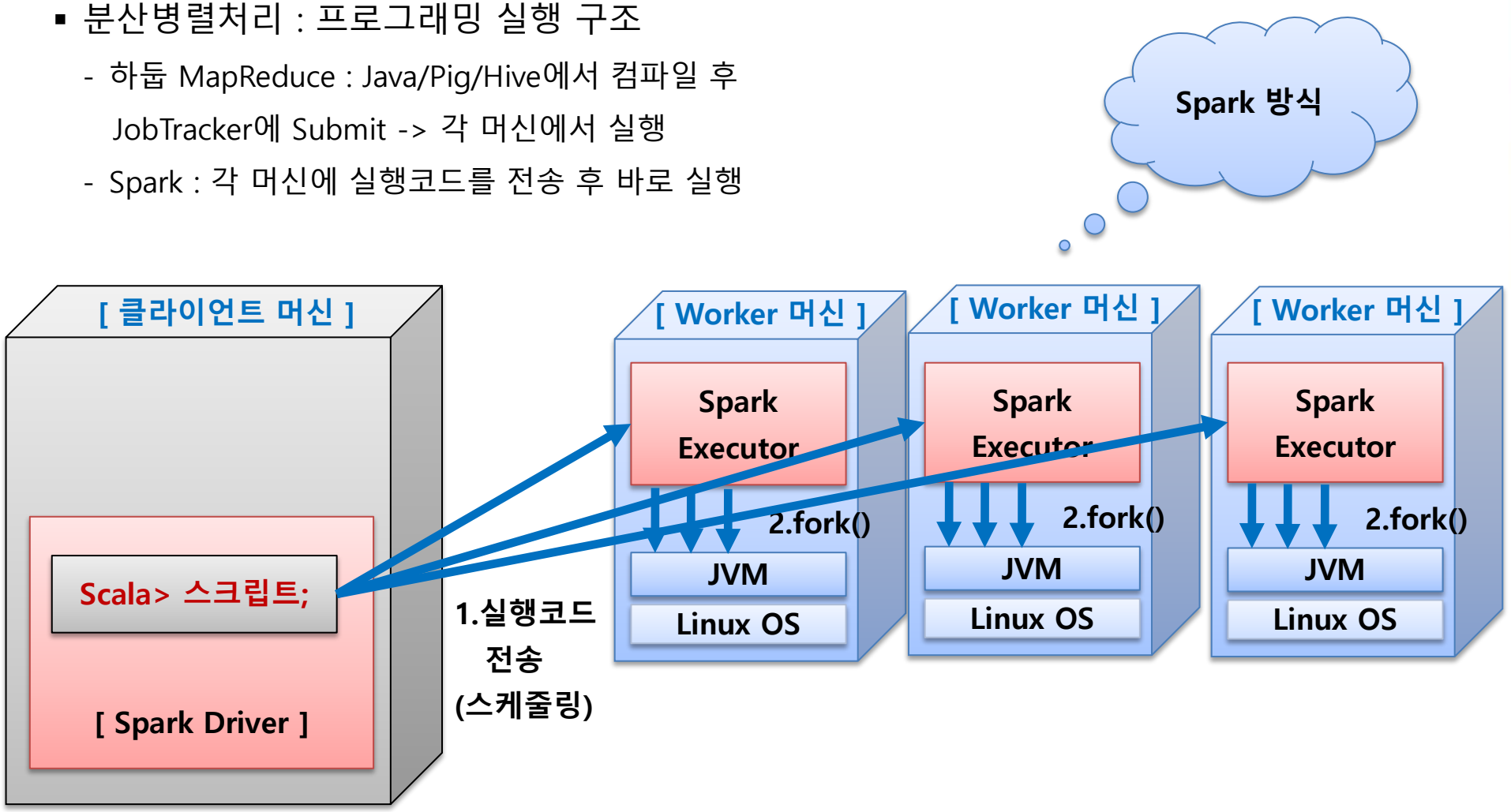
Spark 아키텍처와 구현원리

- **함수형 언어 : Scala**
  - 분산병렬처리 : 프로그래밍 실행 구조
    - 하둡 MapReduce : Java/Pig/Hive에서 컴파일 후 JobTracker에 Submit -> 각 머신에서 실행
    - Spark : 각 머신에 실행 코드를 바로 전송 후 실행



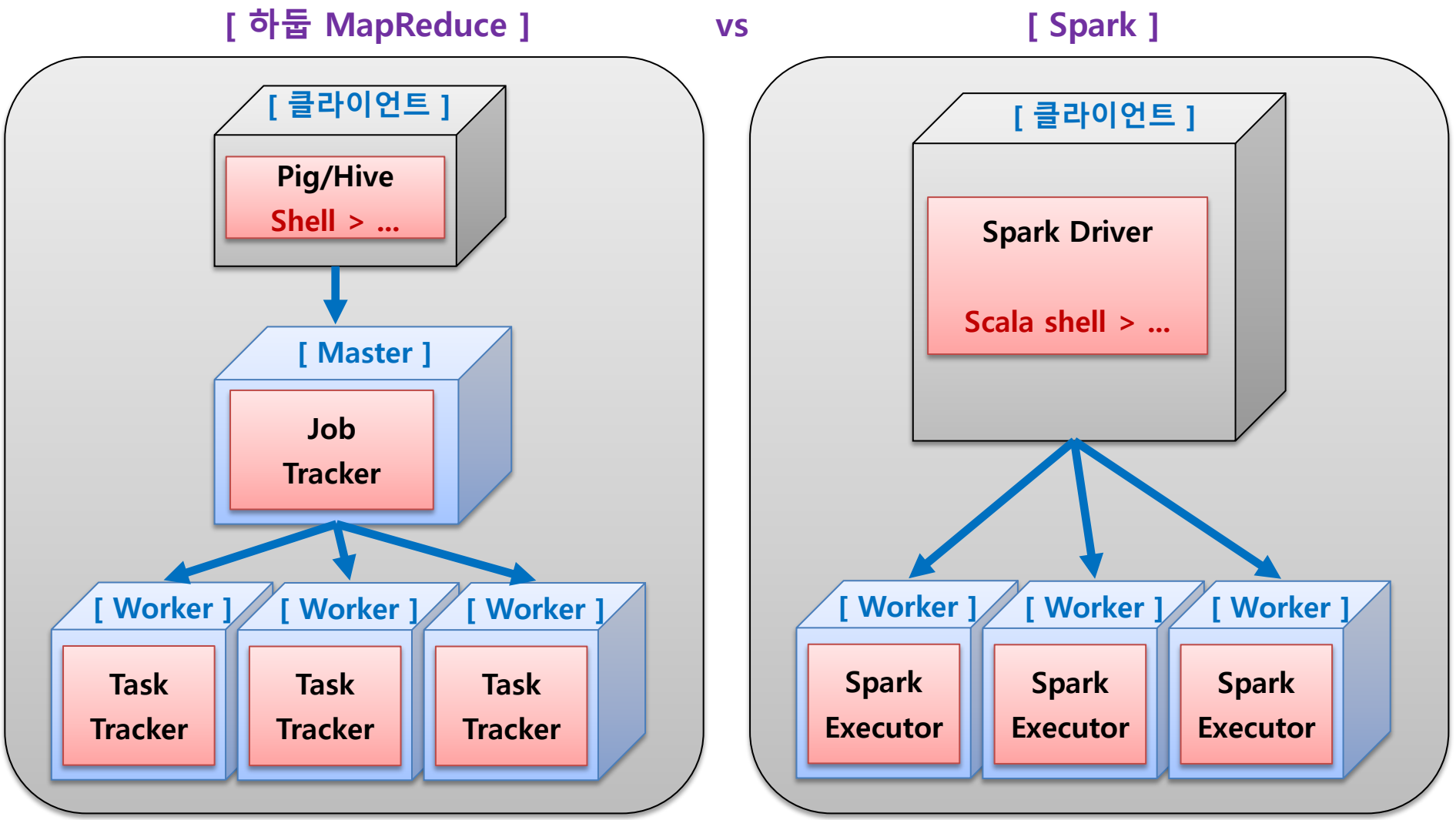
Spark 아키텍처와 구현원리

- **함수형 언어 : Scala**
  - 분산병렬처리 : 프로그래밍 실행 구조
    - 하둡 MapReduce : Java/Pig/Hive에서 컴파일 후 JobTracker에 Submit -> 각 머신에서 실행
    - Spark : 각 머신에 실행코드를 전송 후 바로 실행



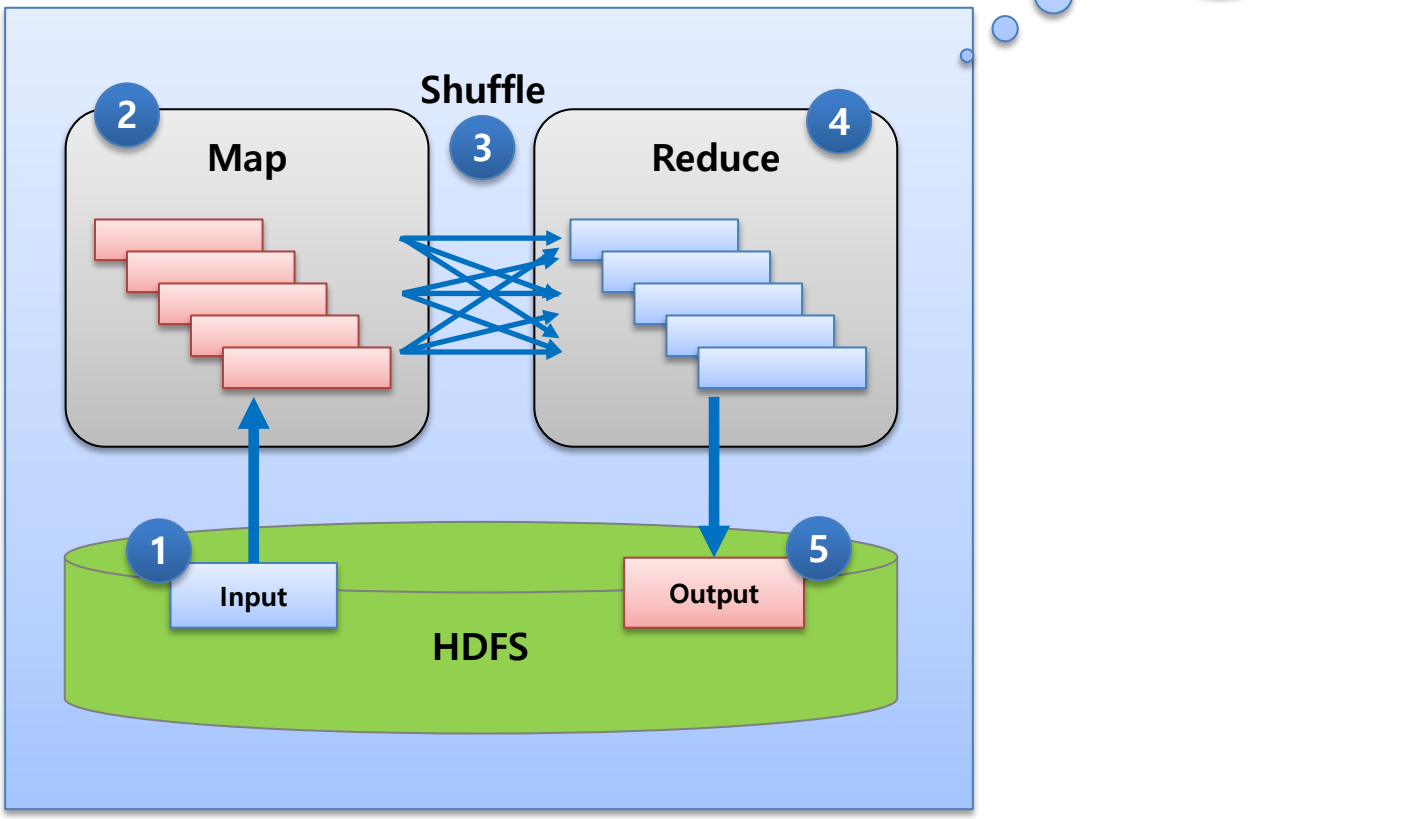
Spark 아키텍처와 구현원리

- 함수형 언어 : Scala



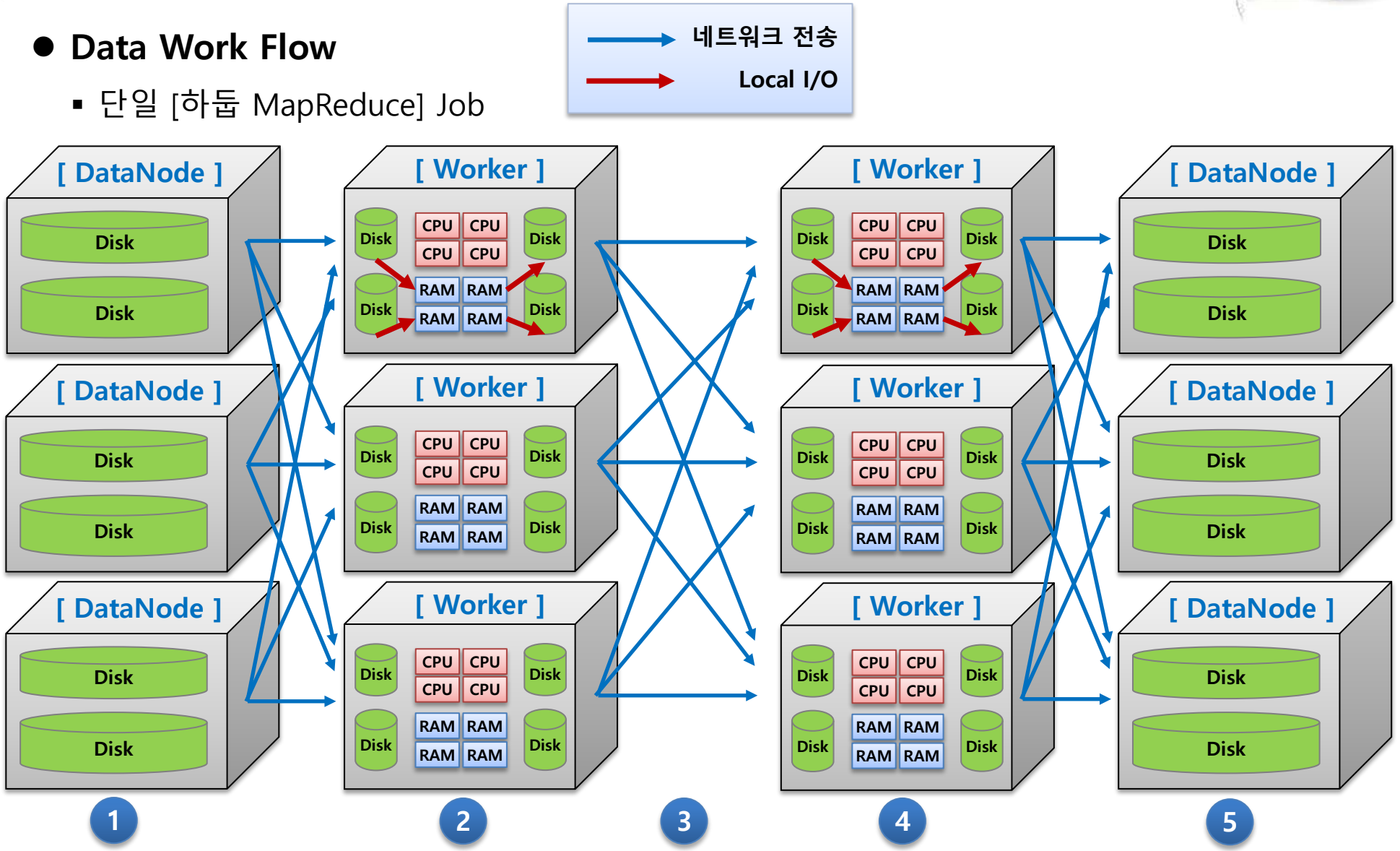
Spark 아키텍처와 구현원리

- **Data Work Flow**
  - Job DAG의 이해
    - [하둡 MapReduce] Job : Input -> Map -> Shuffle -> Reduce -> Output
    - Job DAG : 1번째 Job -> 2번째 Job -> ... -> N번째 Job



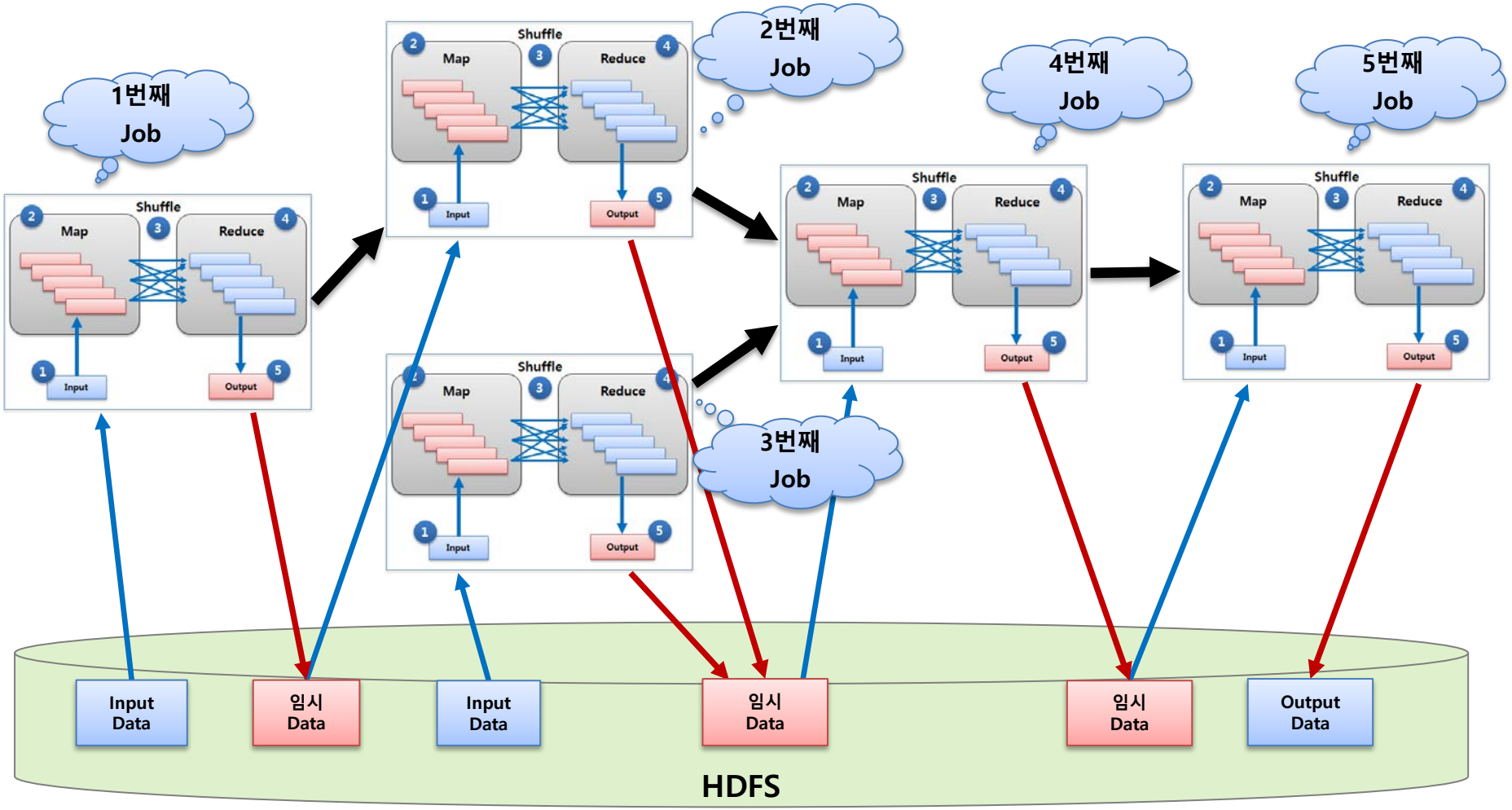
Spark 아키텍처와 구현원리

- **Data Work Flow**
  - 단일 [하둡 MapReduce] Job



Spark 아키텍처와 구현원리

- **Data Work Flow**
  - [하둡 MapReduce] Job DAG : 1번째 Job -> 2번째 Job -> ... -> N번째 Job

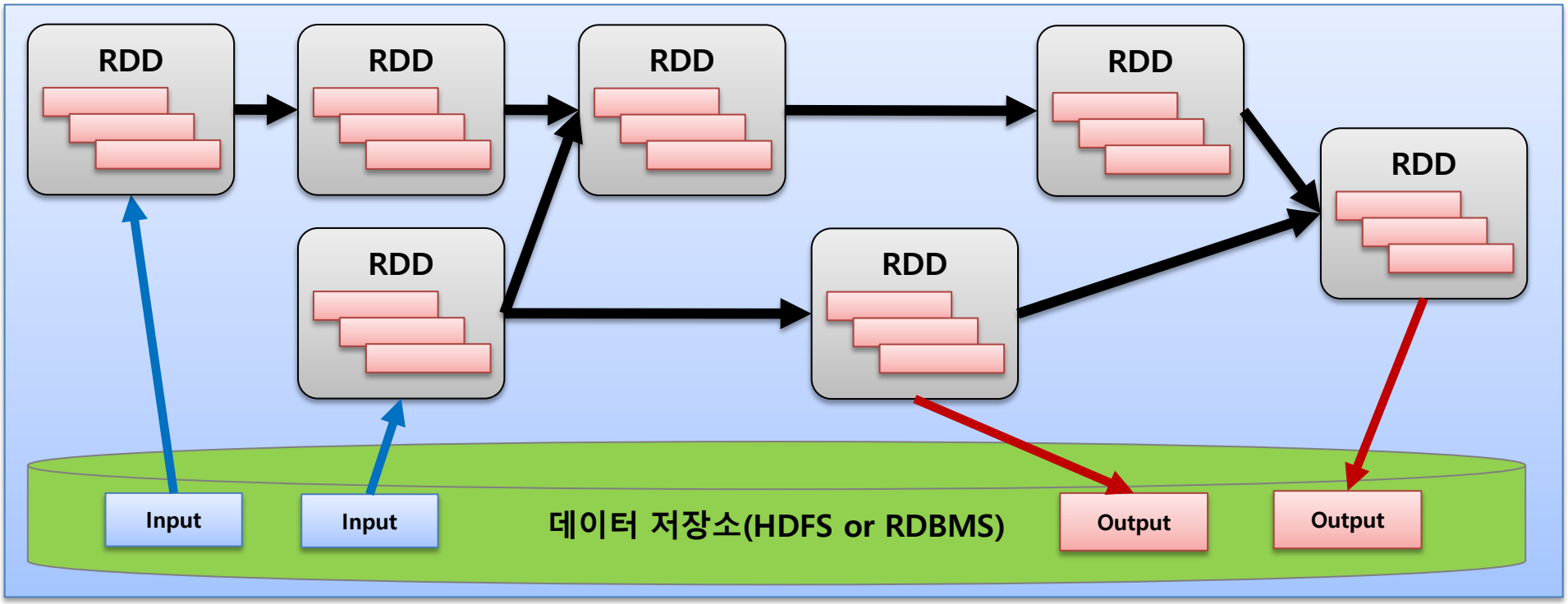


Spark 아키텍처와 구현원리

- **Data Work Flow**
  - 하둡 MapReduce의 단점
    - Disk I/O가 크다.
    - 네트워크 부하가 크다.
    - 전체 Work을 여러 단계의 Job으로 분리하는 것은 굉장히 어렵고 이해하기도 힘들다.
  - 대안 => 스파크
    - 입출력 사이의 중간 과정은 Disk대신 메모리에서 처리.
    - 작업은 단일 Job : 대신 여러 단계의 Stage로 구분해서 내부적으로 실행됨.
    - 배치성 작업뿐만 아니라 대화형 분석이 가능
    - Data Work Flow : RDD(Resilient Distributed Dataset, 탄력적인 분산 데이터셋) Graph로 유지관리.
    - Input, Map, Shuffle, Reduce, Output을 위한 체계적인 Operator 제공.
    - Operator는 Transformation과 Action로 크게 구분.
    - Action 연산시에만 실제 실행 : 지연 실행(Lazy Evaluation)
    - Persist( )를 요청하면 특정 RDD를 인메모리에 상주시킴.
    - 물리적인 파티션을 직접 관리할 수 있음.

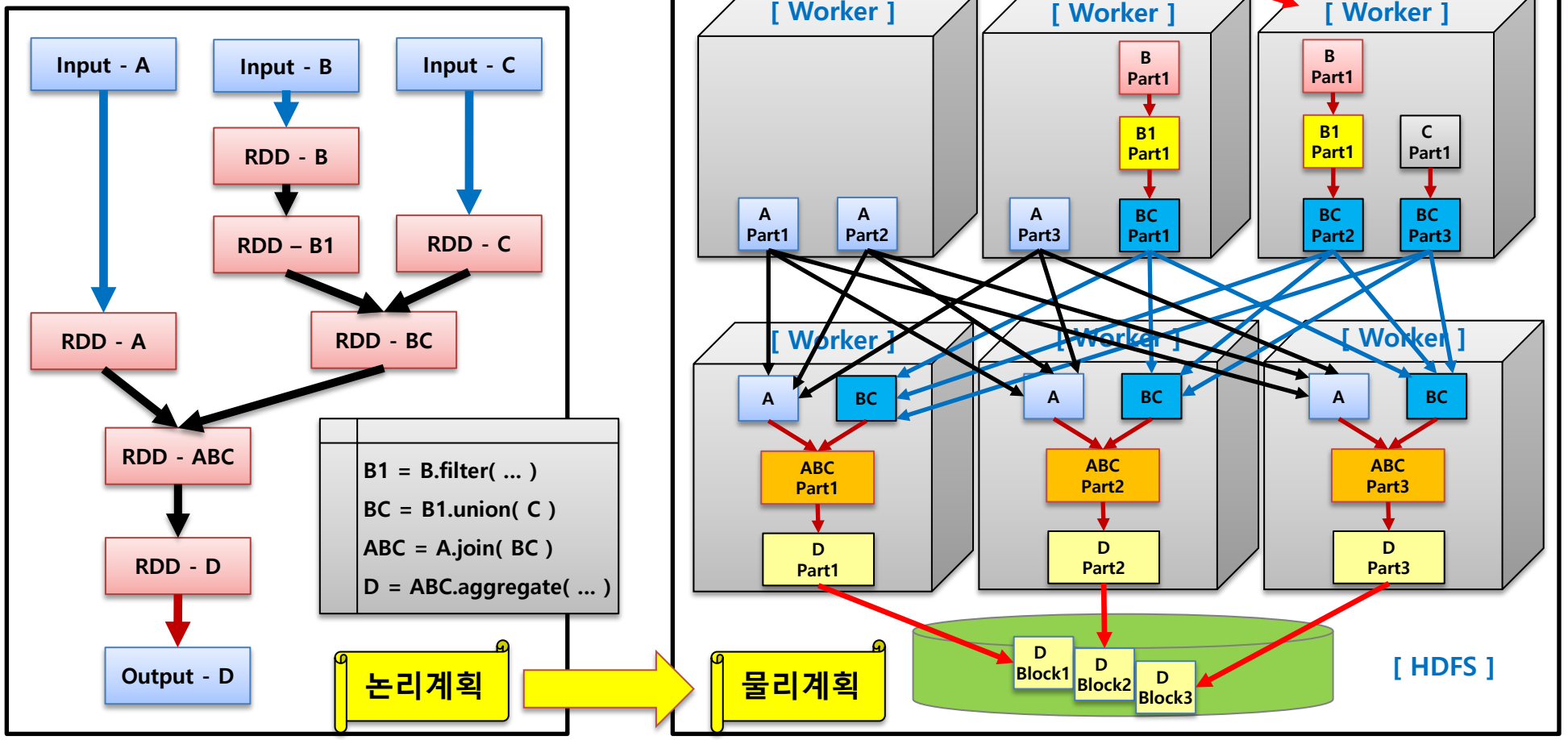
Spark 아키텍처와 구현원리

- **Spark RDD와 인메모리 방식**
  - RDD 개요
    - Resilient Distributed Dataset : 탄력적인 분산 데이터셋
    - Resilient : 처리과정에서 일부 데이터가 손상되어서 복구가 가능(부분 손상 -> 부분 복구)
    - Distributed : 처리과정에서 데이터를 여러 머신에 분산 저장 => 파티션(Partition)
    - RDD Graph : [Input] -> RDD -> RDD -> ... -> RDD -> [Output]



Spark 아키텍처와 구현원리

- **Spark RDD와 인메모리 방식**
  - RDD 논리 및 물리 계획



Spark 아키텍처와 구현원리

- **Spark 연산 및 대화형 분석**
  - Transformation과 Action
    - Transformation은 입력 데이터셋을 단계별 RDD로 변형하는 연산자
    - Action은 결과를 콘솔에서 보거나 HDFS등에 저장하는 연산자
    - Action 연산자를 실행하면 입력 소스에서 데이터를 불러와서 처리하고 그 결과를 보여주거나 저장

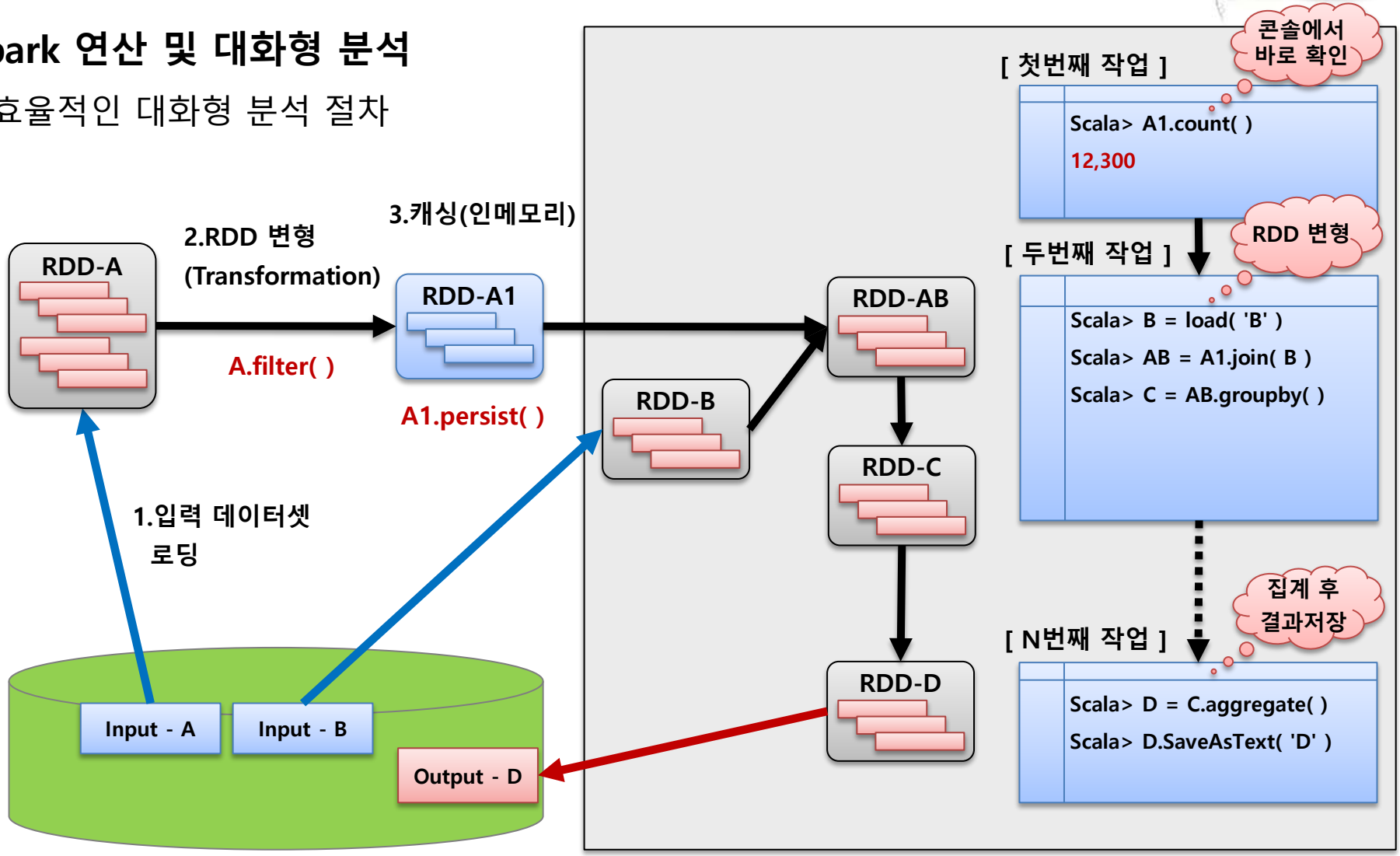
      => 즉 Transformation 연산은 실제 실행되지 않고 Action 연산을 요청할 때에만 실행되므로 이런 방식을 Lazy Evaluation(지연 실행)이라고 부름

  - 특징
    - Pig, Hive 등이 제공하는 연산자보다 직관적이고 효율적인 다수의 연산자를 지원함
    - 주의할 점은 Action을 요청할 때마다 데이터를 입력 소스에서 불러와서 처리함

      => 원하는 RDD에 persist( ) 메소드로 분산캐싱을 명시하는 방법을 제공
    - 대화형으로 동일한 입력 데이터셋을 처리하고 요약(집계)할 때 효율적임
    - Spark 연산자를 이용한 다양한 라이브러리를 제공

Spark 아키텍처와 구현원리

- **Spark 연산 및 대화형 분석**
  - 효율적인 대화형 분석 절차



Spark 아키텍처와 구현원리

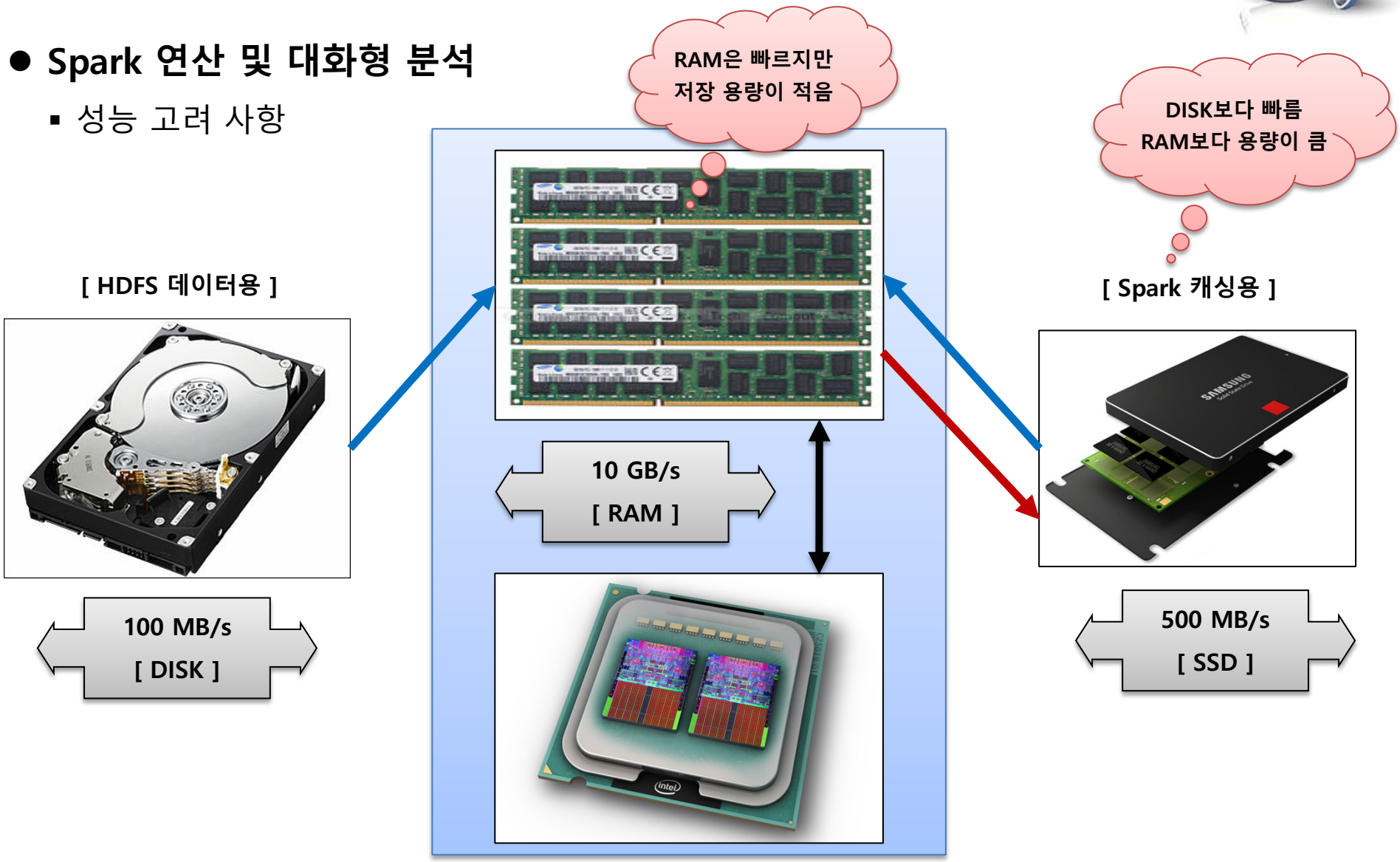
- **Spark 연산 및 대화형 분석**
  - 분산캐싱(인메모리)
    - 매번 HDFS나 RDBMS에서 데이터를 불러오는 것은 비효율적이고 시간이 오래 걸림
    - 원하는 특정 RDD를 분산캐싱(다수의 머신에 있는 메모리에 데이터를 상주시킴)할 수 있음
    - 캐싱은 메모리(우선)와 Disk 모두 가능.
    - 직렬화(압축, 컬럼기반) 옵션을 지원.

SSD 권장

옵션	필요공간	CPU시간	메모리저장	디스크저장	비고
MEMORY_ONLY	많음	낮음	YES	X	
MEMORY_ONLY_SER	적음	높음	YES	X	직렬화
MEMORY_AND_DISK	많음	중간	일부(우선)	일부	
MEMORY_AND_DISK_SER	적음	높음	일부(우선)	일부	직렬화
DISK_ONLY	적음	낮음	X	YES	

Spark 아키텍처와 구현원리

- **Spark 연산 및 대화형 분석**
  - 성능 고려 사항



Spark을 활용한 대화형 분석

- **실행 방식**
  - 로컬 모드

    $ bin/pyspark --master **local**

    - Shell을 실행하는 로컬 머신에서, 단일 프로세서로 Spark이 실행됨
    - Spark/PySpark Shell 모두 가능
  - 로컬 [N] 모드

    $ bin/pyspark --master local**[N]**

    - Shell을 실행하는 로컬 머신에서, [N]개의 프로세서로 Spark이 병렬로 실행됨
    - Spark/PySpark Shell 모두 가능
  - 분산 모드

    $ bin/spark-shell --master **spark://master.host:7077**

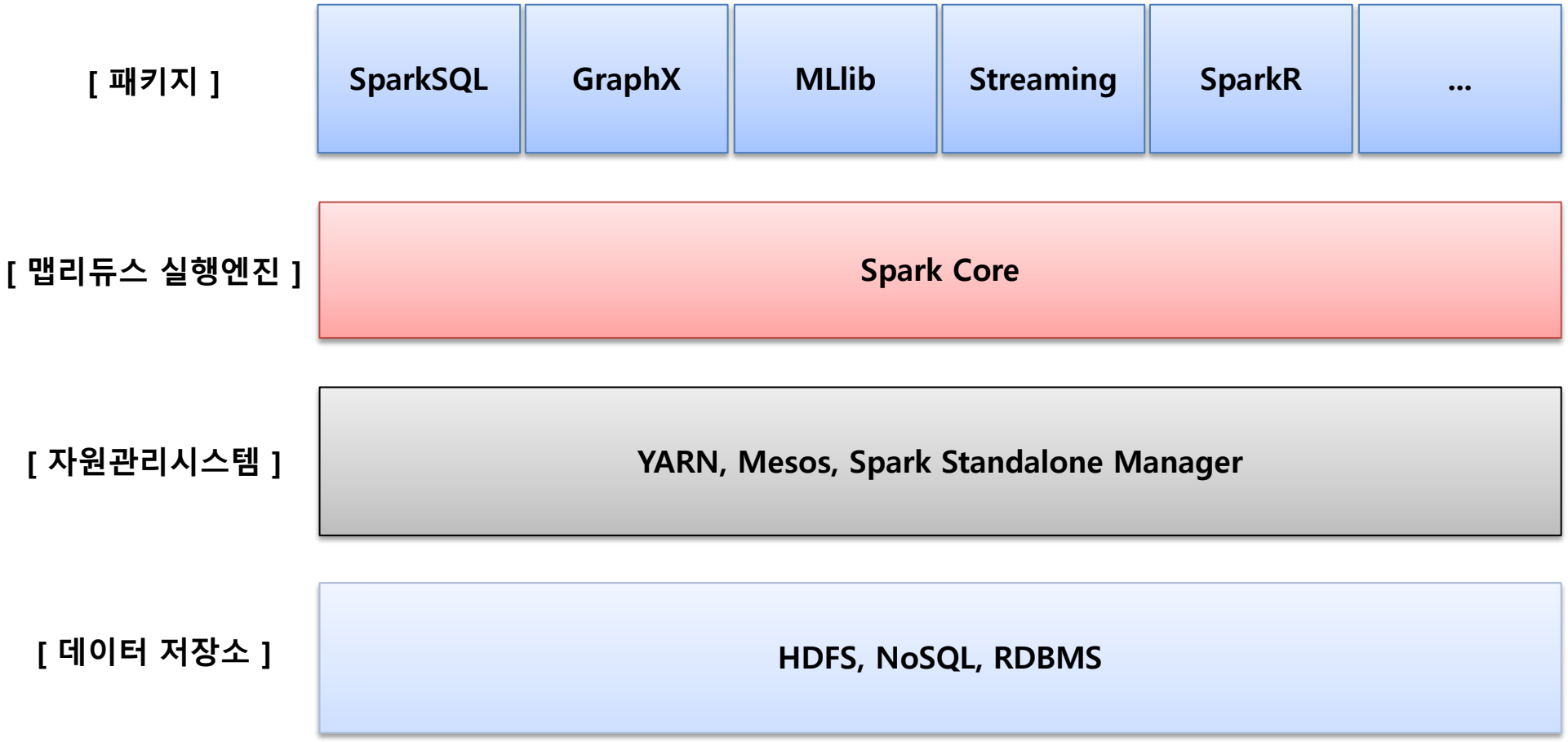
    - Spark Master/Worker 데몬이 구동되고 있는 클러스터 환경에서, 분산+병렬로 실행됨
    - Spark Shell 만 가능

    -> 클러스터 매니저를 Mesos를 사용할 경우

    $ bin/spark-shell --master mesos://**mesos.host:Port**

Spark을 활용한 대화형 분석

- Spark 기능 구조도



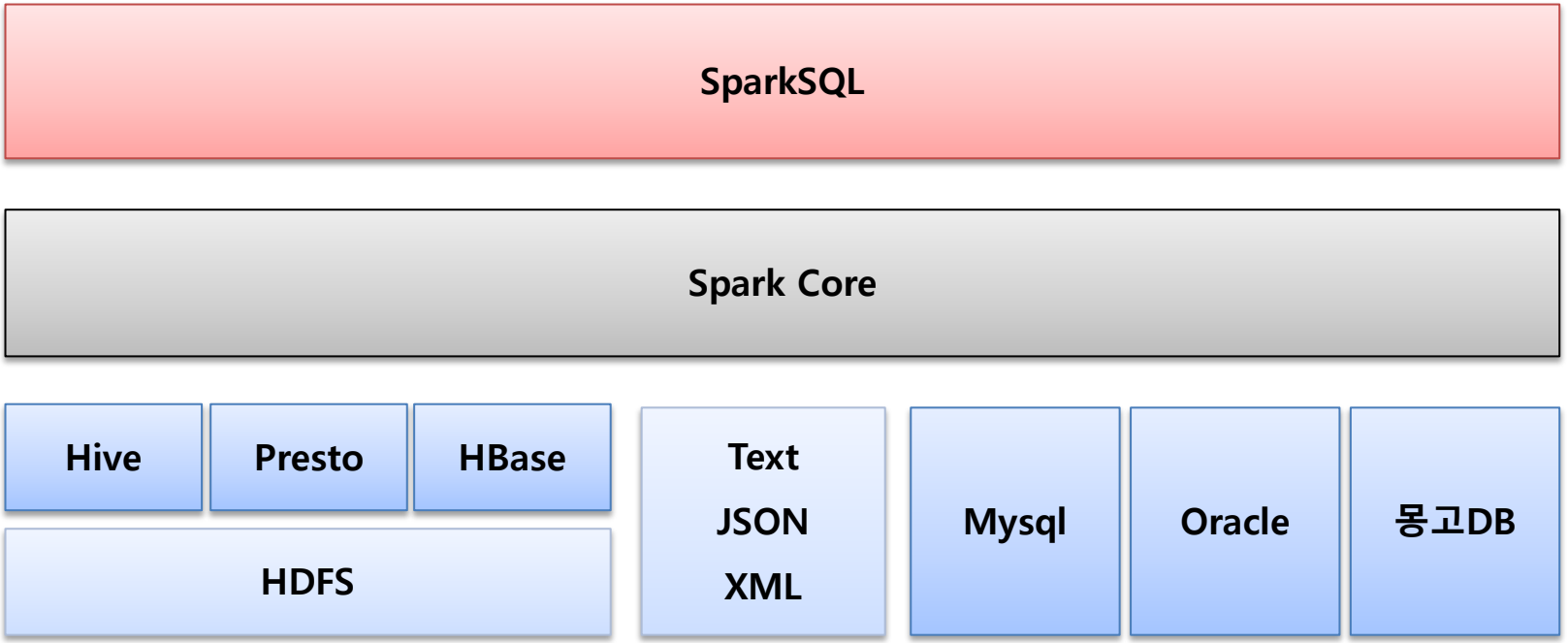
Spark을 활용한 대화형 분석

- **Spark의 주요 기능**

구분	모듈	기능	분야
핵심	Spark Core	맵리듀스와 같은 병렬처리 및 반복 연산	분산병렬처리
주요 패키지	Spark SQL	하이브와 같은 SQL 분석	데이터웨어하우스
	MLlib	마훗과 같은 머신러닝라이브러리	데이터 마이닝
	GraphX	네트워크 분석	SNA 등 네트워크 분석
	Spark Streaming	스톰과 같은 실시간 스트리밍 분석	스트리밍 처리 및 분석
확장 프로젝트	BlinkDB	빠른 응답속도를 가진 SQL 쿼리 분석	Ad-Hoc 분석
	SparkR	통계 패키지인 R과의 통합	통계 분석

Spark을 활용한 대화형 분석

- **SparkSQL**
  - SQL 쿼리 분석
    - 입력 : HDFS/Hive, RDBMS, NoSQL, Text File(Plain Text, JSON, XML 등)
    - 데이터셋을 DB Table로 처리, 내부적으로는 DataFrame 형식

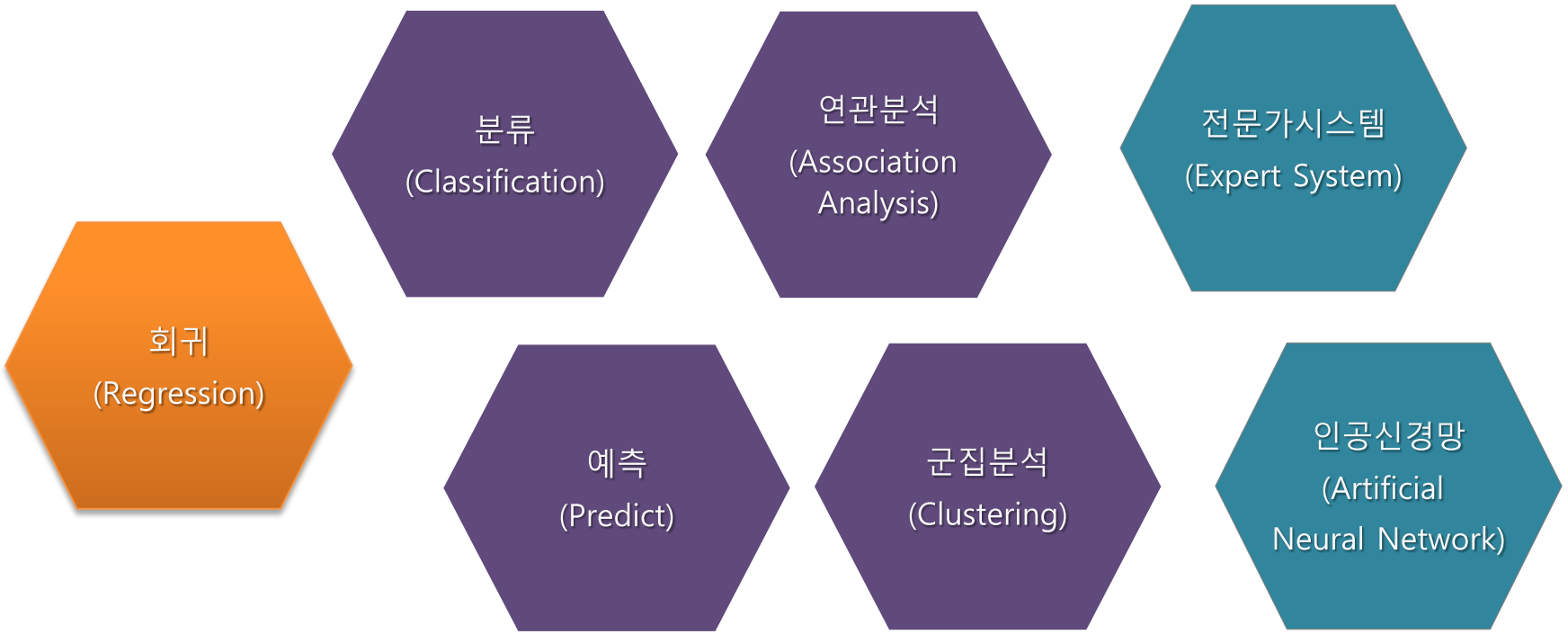


Part V

머신러닝의 이해

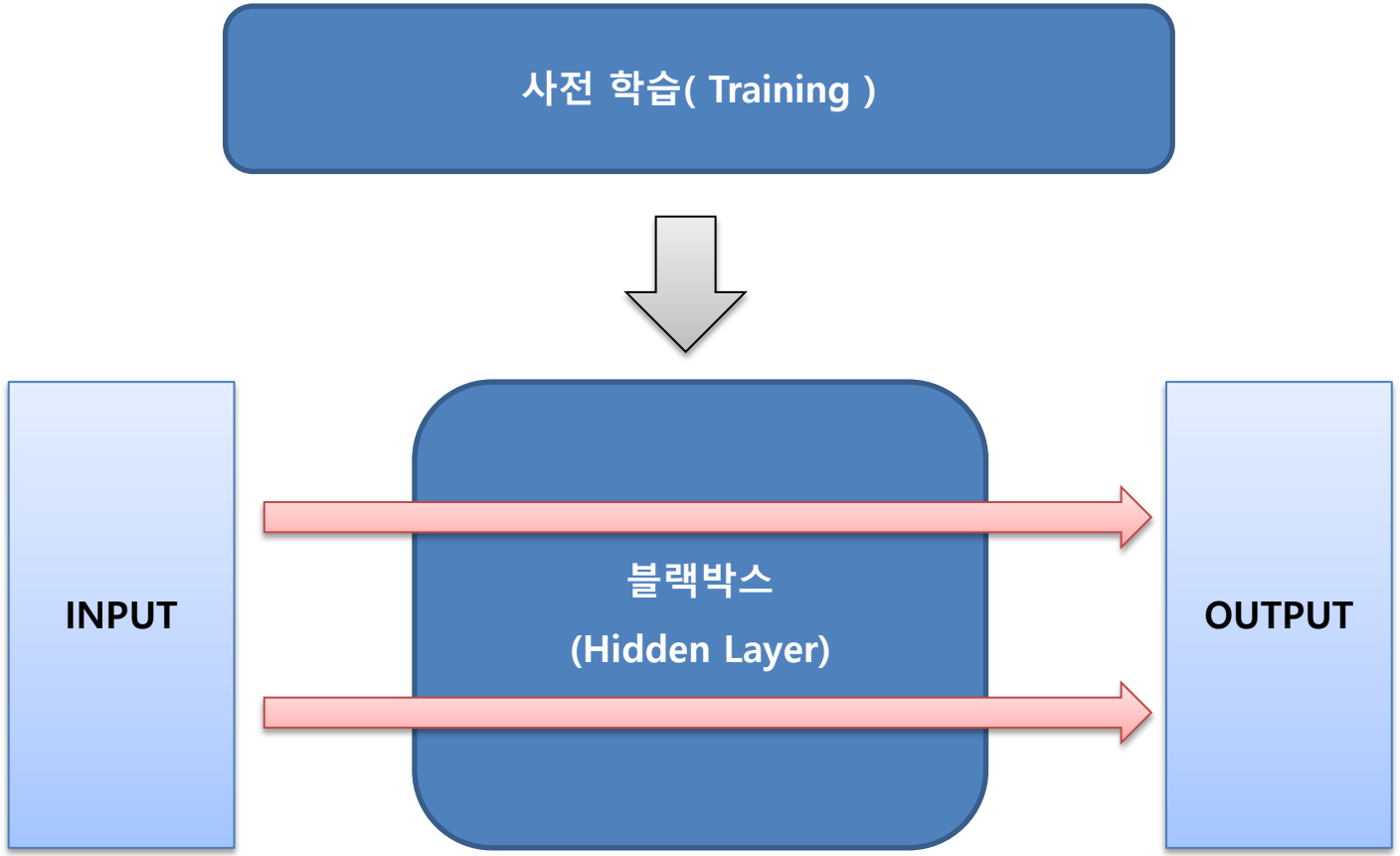
머신러닝 개요

- **머신러닝 주요 기법**



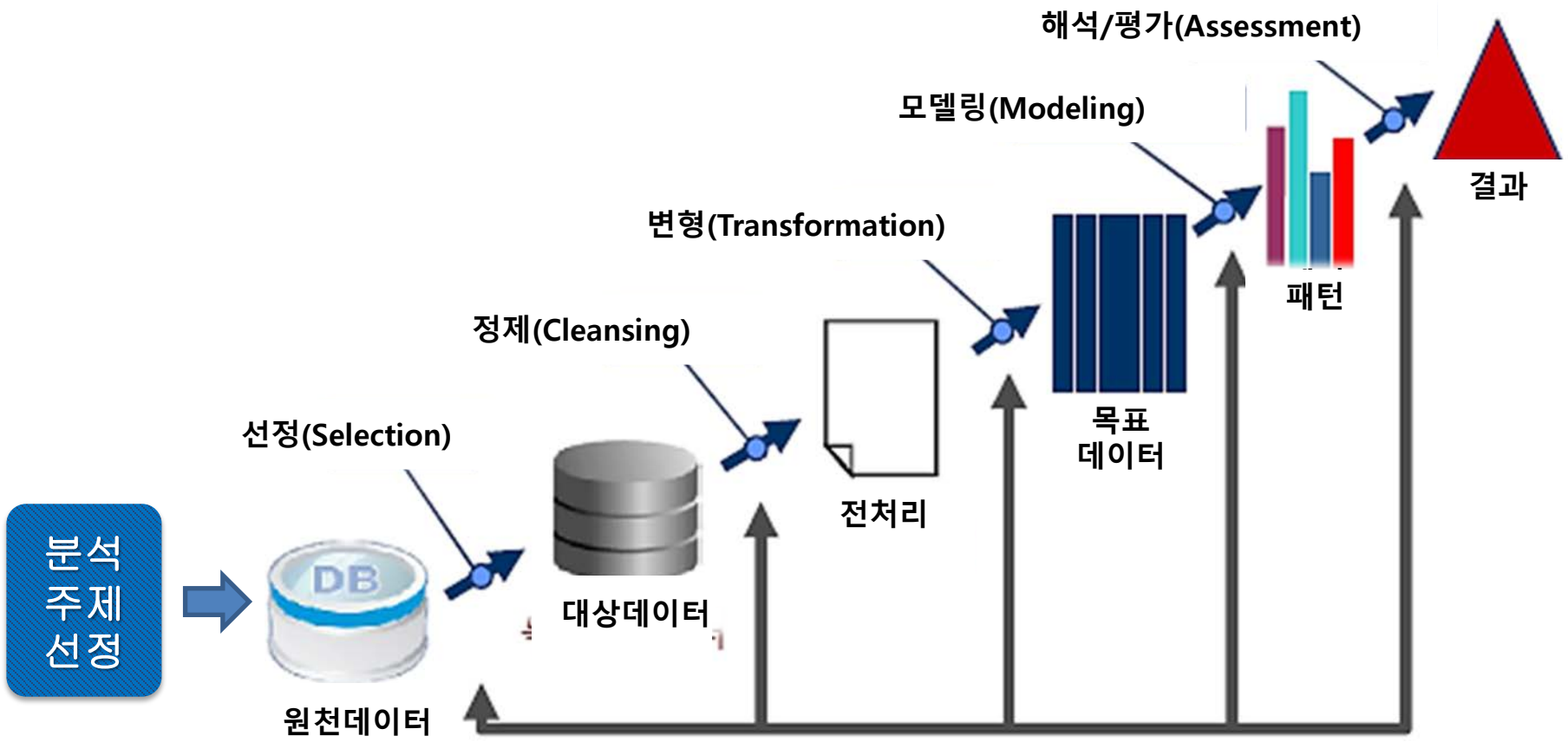
머신러닝 개요

- 머신러닝의 기본 구조



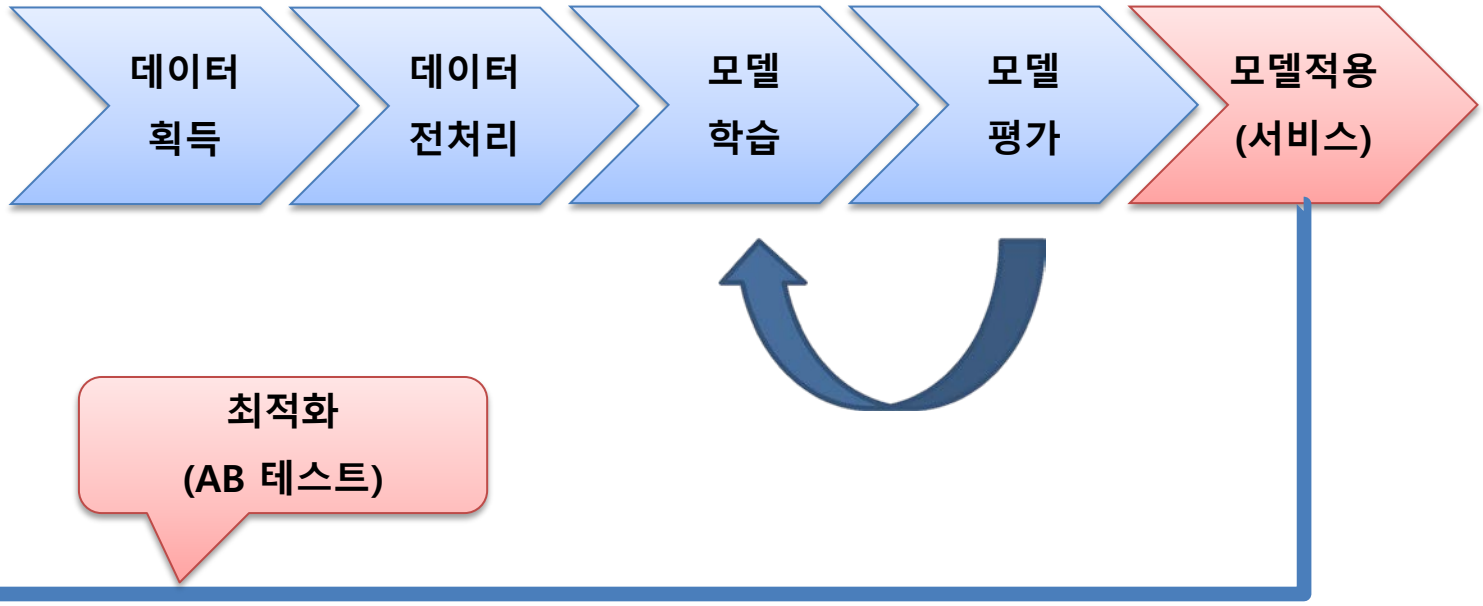
머신러닝 개요

- 통계/분석 프로세스



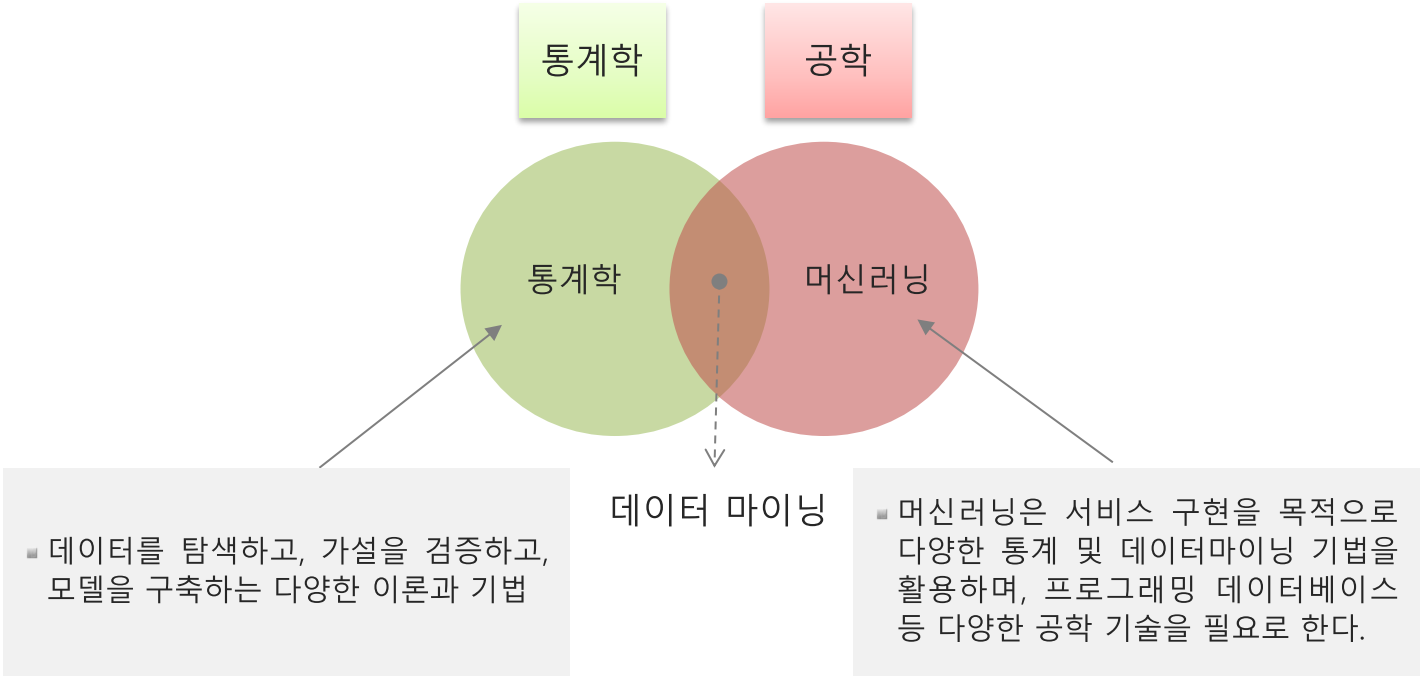
머신러닝 개요

- 데이터마이닝 & 머신러닝 프로세스



머신러닝 개요

- **통계에서 머신러닝까지**
  - **통계**
  - **데이터마이닝**
  - **머신러닝**



머신러닝 개요

■ 기초 통계

- 기술 통계, 빈도 및 추정 테이블, 가설 검정, 상관과 공분산, T-테스트 등

■ 기술통계(Descriptive Statistics)

- 자료를 요약하는 기초적인 통계
- 기술통계량 : 데이터수 개수, 평균, 분산, 표준편차, 최소값, 1분위값, 중앙값, 3분위값, 최대값, 왜도, 첨도 등
  -> 대표값 : 평균, 중앙값, 최빈치
  -> 분포 : 분산, 표준편차
  -> 대칭 : 왜도
- R 예제(표) : summary( )

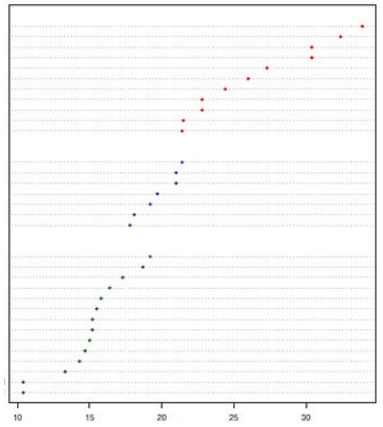
Descriptive statistics via summary( )

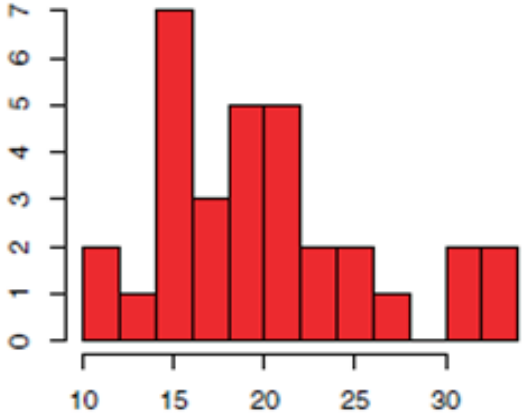
> summary(mtcars[vars])

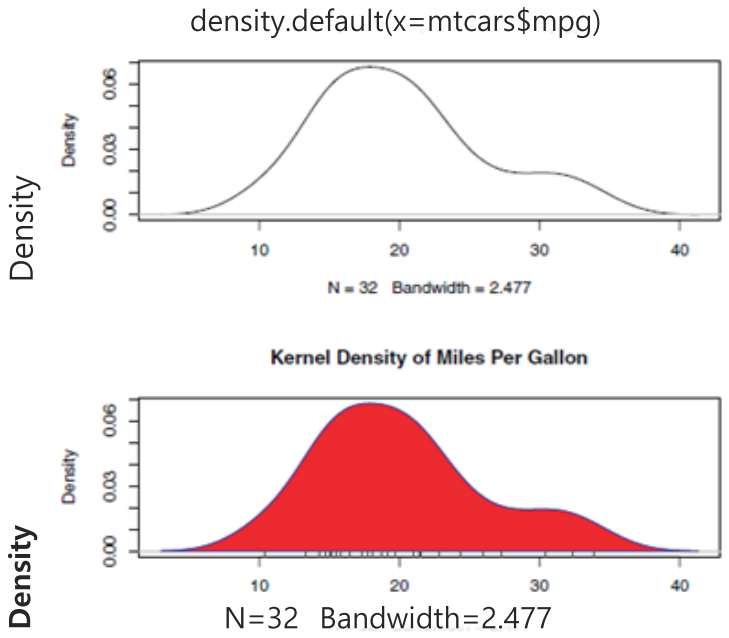
mpg		hp		wt	
Min.	:10.4	Min.	:52.0	Min.	:1.51
1st Qu.	:15.4	1st Qu.	:96.5	1st Qu.	:2.58
Median	:19.2	Median	:123.0	Median	:3.33
Mean	:20.1	Mean	:146.7	Mean	:3.22
3rd Qu.	:22.8	3rd Qu.	:180.0	3rd Qu.	:3.61
Max.	:33.9	Max.	:335.0	Max.	:5.42

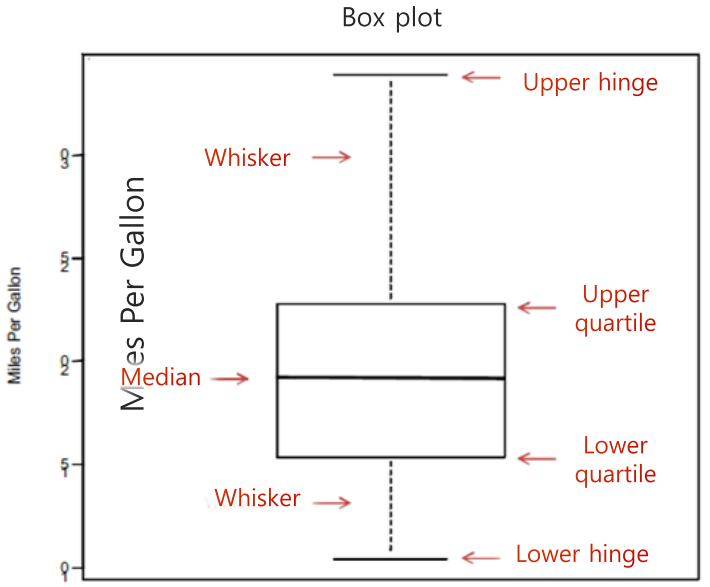
머신러닝 개요

- 분포 시각화
  - 산포도
  - 히스토그램
  - 핵밀도 그래프
  - 박스 그래프
  - 파이 차트
  - 라인 차트





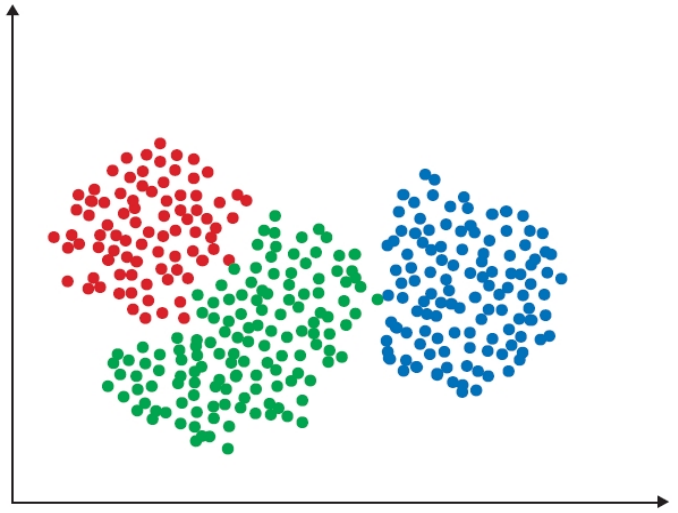




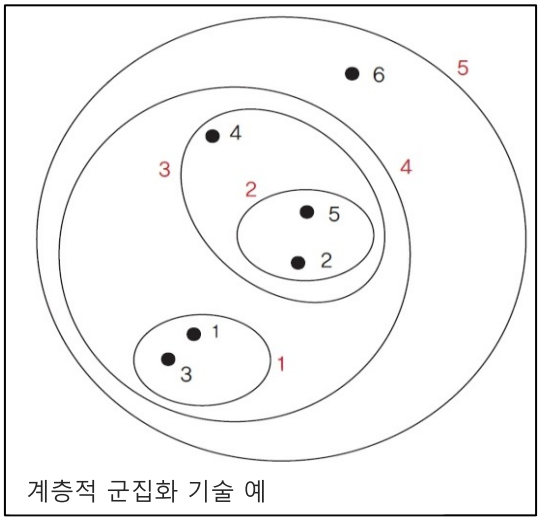
머신러닝 개요

■ 군집 : 비지도 학습

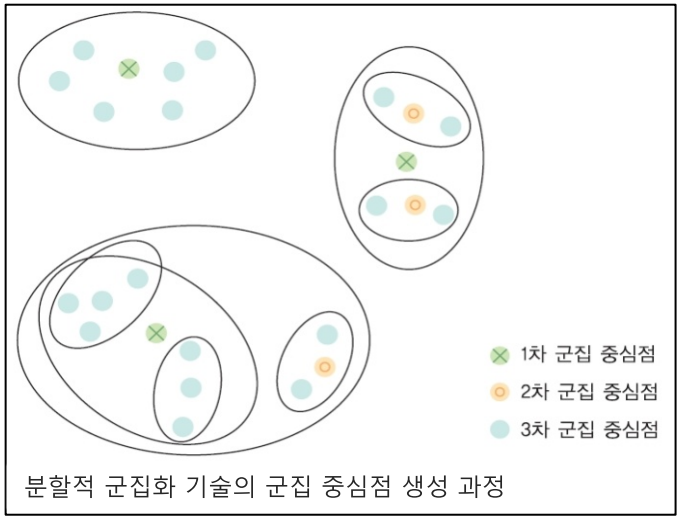
- 개요
- 군집 방법
  - 계층적 군집화 기술
  - 분할적 군집화 기술
- 군집 알고리즘
  - K-Means 알고리즘
  - EM(Expectation Maximization) 알고리즘



군집화 기술



계층적 군집화 기술 예



분할적 군집화 기술의 군집 중심점 생성 과정

머신러닝 개요

분류

- 분류 개요
  - 지도학습, 어떤 클래스에 속하는지 예측
- 의사결정나무의 장단점
  - 가장 대표적인 분류 기법. 설명력이 높다.
- 모델 구조
  - 의사결정 영역:사각형, 단말 영역
- 처리 과정(재귀 순환)
  - 2개(오른쪽 가지, 왼쪽 가지)의 유형으로 분류하는데 가장 적합한 속성과 그 분리값을 찾는다.
  - 분기된 가지(오른쪽, 왼쪽)로 간 다음, 아이템들의 라벨이 모두 같으면 그 값을 반환한다.
  - 각 가지에 속한 아이템의 라벨이 동일하지 않으면 처음 (1)로 간다

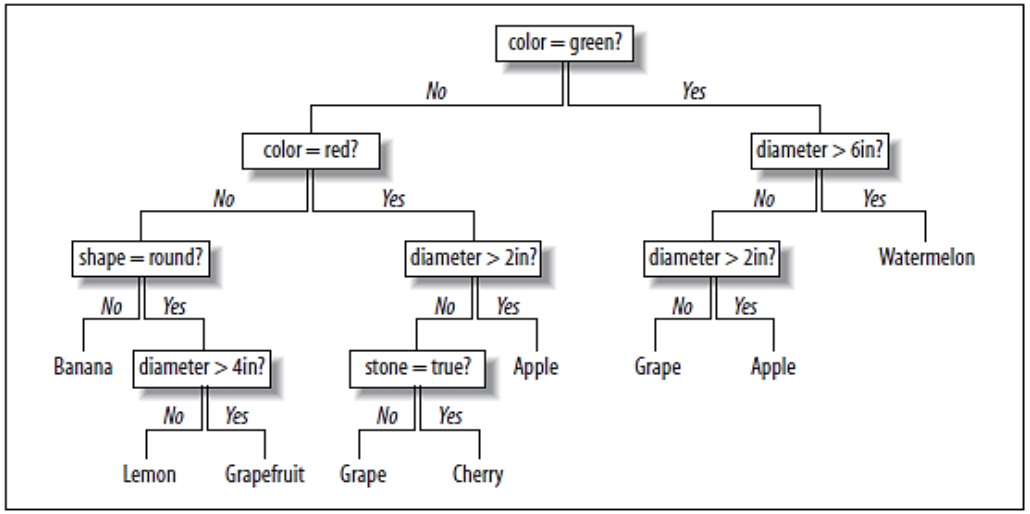
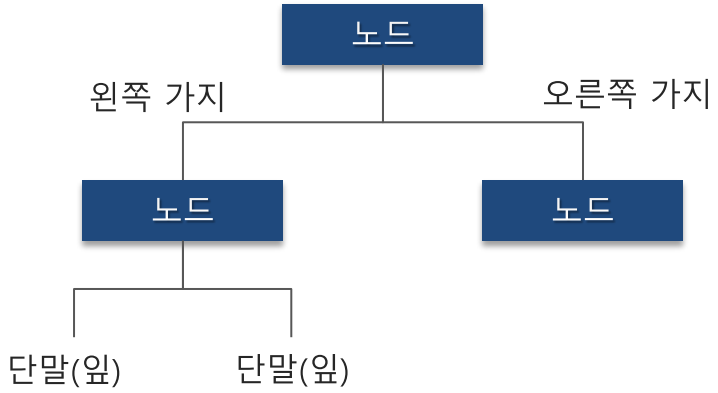


Figure 12-1. Example decision tree



머신러닝 개요

추천시스템

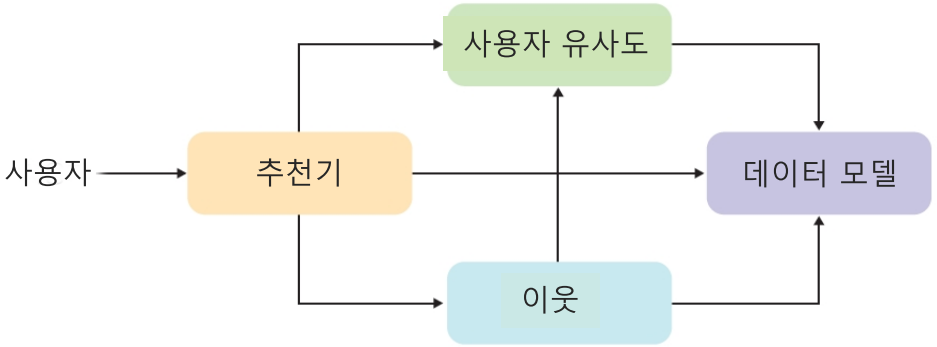
- 구매 가능성이 높은 상품을 고객에게 추천
- 대표적인 방법: 협업필터링과 연관규칙

협업필터링

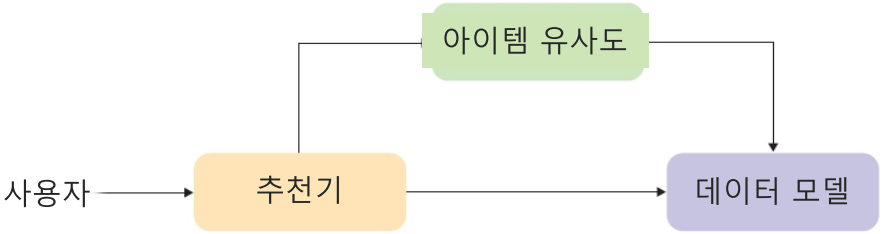
- 사용자 기반
- 아이템 기반

연관규칙

- (A->B)의 지지도 = $P(A \cap B) = N(A \cap B) / N(T)$
- (A->B)의 신뢰도 = $P(B|A) = P(A \cap B) / P(A)$
- (A->B)의 향성도 = $P(B|A) / P(B) = P(A \cap B) / P(A)P(B)$



사용자 기반 추천 엔진 구성도

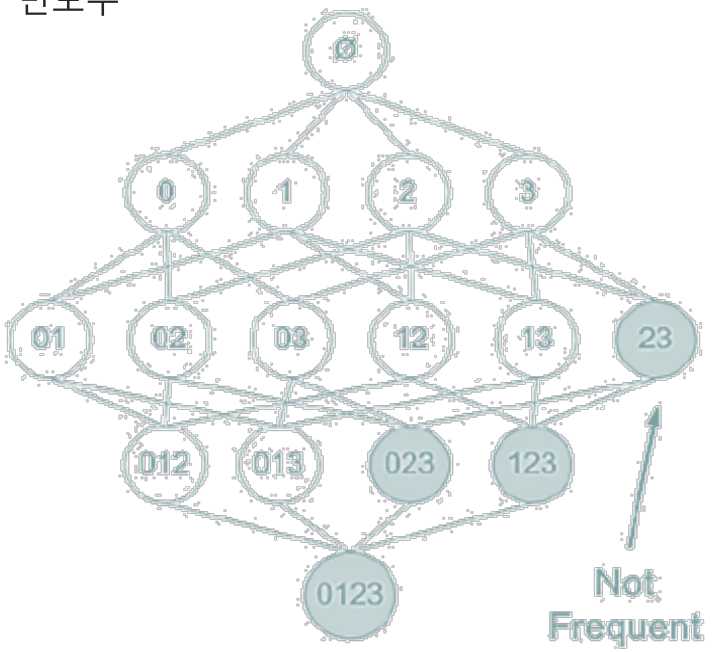


아이템 기반 추천 엔진 구성도

머신러닝 개요

연관분석

- 연관분석 개요
  - 개요
    - 연관성 : 특정 아이템과 다른 아이템의 동시 발생 현상
    - 연관규칙 또는 장바구니분석
  - Apriori 알고리즘
    - 함께 구매한 아이템의 집합을 찾는 대표적인 알고리즘
    - 아이템 수가 100개 -> 아이템 집합의 수는 $1.26X^{1030}$
  - Association Rule
    - 트랜잭션 수, 개별 상품의 빈도수, A->B 두 상품의 동시 발생 빈도수 -> 지지도, 신뢰도, 향상도 계산
  - 사례
    - 아마존 책 추천
    - 기저귀를 구매한 젊은 남성이 맥주를 많이 산다



Part VI

Spark MLlib

1. Data Model

1) Data Model

- ML을 위해 필요한 Data Type의 분류

(1) Spark Basic Type

- vector, matrix, array
- list, dataframe

(2) MLlib Type

- LabeledPoint
- Rating
- 알고리즘별 Model Class

1. Data Model

1) Data Model

- **Spark Basic Type**

(1) Vector

- 동일한 Type의 데이터 N개, 1차원, 고정길이

```
// Create a dense vector (1.0, 0.0, 3.0).
val dv: Vector = Vectors.dense(1.0, 0.0, 3.0)
```

[Note] 수학분야의 방향과 크기를 가지고 있는 데이터를 의미하지 않음

(2) Matrix

- 동일한 Type의 데이터, 2차원, M개의 행과 N개의 열로 구성(M*N 행열)

```
// Create a dense matrix ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
val dm: Matrix = Matrices.dense(3, 2, Array(1.0, 3.0, 5.0, 2.0, 4.0, 6.0))
```

(3) Array

- 동일한 Type의 데이터, N차원(1차원, 2차원, ... N차원), 고정길이

```
var z = new Array[String](3);   z(0) = "Zara";   z(1) = "Nuha";   z(2) = "Ayan"
var z2 = Array("Zara", "Nuha", "Ayan")
```

(4) List

- 동일한 Type의 데이터, 1차원, 가변길이

(5) DataFrame

- 다양한 Type의 데이터, 2차원, 엑셀 Sheet or 데이터베이스 테이블 구조와 유사

1. Data Model

1) Data Model

- **Vector Type**

(1) Vector의 분류

- 고밀도 : Dense -> 전체 데이터를 double type의 고정길이 배열에 저장

- 저밀도 : Sparse -> 0.0이 아닌 값과 그 인덱스를 저장

(2) Example

```
// Create the dense vector <1.0, 0.0, 2.0, 0.0>;
val denseVec1 = Vectors.dense(1.0, 0.0, 2.0, 0.0)
val denseVec2 = Vectors.dense(Array(1.0, 0.0, 2.0, 0.0))

// Create the sparse vector <1.0, 0.0, 2.0, 0.0>;
val sparseVec1 = Vectors.sparse(4, Array(0, 2), Array(1.0, 2.0))  //(int size, int[] indices, double[] values)
```

(3) Vectors 메소드

apply argmax asInstanceOf compressed copy
foreachActive isInstanceOf numActives numNonzeros size
toArray toDense toJson toSparse toString

1. Data Model

1) Data Model

- **LabeledPoint Type**

(1) LabeledPoint = (Labeled : double, Features : vectors)

- Labeled ->

1.Classification : 명목형 값(classes) -> double

Binary(True, False) -> 1.0 / 0.0

MultiClass -> 0.0 1.0 2.0 .... N.0

2.Regression : 연속형 값(수치형), double

- Features -> 속성값 Vectors(double)

명목형 Feature의 값은 double type으로 변환해야 함

(2) Example

```
scala> val examples = MLUtils.loadLibSVMFile(sc, "file:///data/spark-1.6.0/data/mllib/sample_binary_classification_data.txt")
examples: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint]
scala> println(examples.take(2).mkString("₩n"))
(0.0, (692, [127,128,129, ... ,656,657], [ 51.0,159.0,253.0, ... ,141.0, 37.0] ) )
(1.0, (692, [158,159,160, ... ,682,683], [124.0,253.0,255.0, ... ,253.0,220.0] ) )
scala> val data = sc.textFile("file:///data/spark-1.6.0/data/mllib/sample_binary_classification_data.txt")
scala> println(data.take(2).mkString("₩n"))
0 128:51 129:159 130:253 ... 657:141 658:37
1 159:124 160:253 161:255 ... 683:253 684:220
```

2. Data 처리

2) 데이터 처리

- 데이터 처리 개요

1. 데이터셋 선택
2. 데이터 로딩
3. 데이터 탐색
4. 데이터 클린징
5. 데이터 변환

2. Data 처리

2) 데이터 처리

- **머신러닝 공개 데이터셋**

1. UCI 데이터 저장소

- 300여개의 데이터셋(classification,regression, clustering, recommender systems)

http://archive.ics.uci.edu/ml/

2. 아마존 공개 데이터셋

- Human Genome Project, Common Crawl web corpus, Wikipedia data, Google Books Ngrams

http://aws.amazon.com/publicdatasets/

3. 캐글(Kaggle) 데이터셋

- 머신러닝 대회, classification, regression, ranking, recommender systems, image analysis

http://www.kaggle.com/competitions

4. KDnuggets 데이터셋

- 공개 데이터셋 목록

http://www.kdnuggets.com/datasets/index.html

2. Data 처리

2) 데이터 처리

- **Spark Examles 데이터 셋**

1. 위치

$SPARK_HOME/data/mllib/

2. 파일 목록(Line 개수, Word 개수, File Size)

2000	4000	63973	gmm_data.txt
6	18	72	kmeans_data.txt
1000	11000	197105	lr_data.txt
6	12	24	pagerank_data.txt
19	57	164	pic_data.txt
100	13610	104736	sample_binary_classification_data.txt
6	34	68	sample_fpgrowth.txt
99	100	1598	sample_isotonic_regression_data.txt
12	132	264	sample_lda_data.txt
100	13610	104736	sample_libsvm_data.txt
501	5511	119069	sample_linear_regression_data.txt
1501	1501	14351	sample_movielens_data.txt
150	737	6953	sample_multiclass_classification_data.txt
11	36	95	sample_naive_bayes_data.txt
322	5474	39474	sample_svm_data.txt
569	569	115476	sample_tree_data.csv
1000	2000	42060	lr-data/random.data
67	536	10395	ridge-data/lpsa.data

2. Data 처리

2) 데이터 처리

- **데이터 로딩**

1. 데이터 소스

- 로컬 파일시스템
- 분산파일시스템 : HDFS, 아마존 S3 등
- RDBMS / NoSQL : Mysql, Oralce, 카산드라, HBase, ElasticSearch 등 -> JDBC Driver로 연결

2. 파일 포맷

- Text File
- JSON File
- CSV File(CSV is ",", TSV is "₩t")
- Key/Value(Hadoop Format) File
- Object

2. Data 처리

2) 데이터 처리

- 데이터 로딩

3. Text File 불러오기

```
val data = sc.textFile("file:///data/spark-1.6.0/data/mllib/pic_data.txt")
println(data.take(3).mkString("₩n"))
data.first()
data.count()
```

4. 컬럼 분리(BackSpace, CSV, TSV)

```
val initData = data.map { line =>
  val values = line.split(' ')map(_.toDouble)
  Vectors.dense(values.init)
}
println(initData.take(3).mkString("₩n"))

val lastData = data.map { line =>
  val values = line.split(' ')map(_.toDouble)
  Vectors.dense(values.last)
}
println(lastData.take(3).mkString("₩n"))
```

```
val firstData = data.map { line =>
  val values = line.split(' ')map(_.toDouble)
  Vectors.dense(values(0))
}
println(firstData.take(3).mkString("₩n"))

val tailData = data.map { line =>
  val values = line.split(' ')map(_.toDouble)
  Vectors.dense(values.tail)
}
println(tailData.take(3).mkString("₩n"))
```

2. Data 처리

2) 데이터 처리

- 데이터 탐색

1. RDD 기초 통계

```
scala> val second = data.map { line =>
     |   val values = line.split(' ').map(_.toDouble)
     |   (values(1))
     | }
second: org.apache.spark.rdd.RDD[Double] = MapPartitionsRDD[23] at map at <console>:36

scala> println(second.collect().mkString("₩n"))
1.0
2.0 ...

scala> second.stats
res42: org.apache.spark.util.StatCounter =
(count: 19, mean: 7.526316, stdev: 4.649487, max: 15.000000, min: 1.000000)
```

2. Data 처리

2) 데이터 처리

- 데이터 탐색

2. RDD[Vector] 기초 통계

```
scala> val features = data.map { line =>
     |   val values = line.split(' ').map(_.toDouble)
     |   Vectors.dense(values)
     | }
features: org.apache.spark.rdd.RDD[org.apache.spark.mllib.linalg.Vector] = MapPartitionsRDD[30] at map at <console>:41
scala> println(features.take(3).mkString("₩n"))
[0.0,1.0,1.0]
[0.0,2.0,1.0]
[0.0,3.0,1.0]
scala> import org.apache.spark.mllib.stat.{MultivariateStatisticalSummary, Statistics}
scala> val summary: MultivariateStatisticalSummary = Statistics.colStats(features)
scala> summary.mean
res56: org.apache.spark.mllib.linalg.Vector = [5.789473684210526,    7.526315789473685,    0.9526315789473684]
scala> summary.variance
res57: org.apache.spark.mllib.linalg.Vector = [21.953216374269005,    22.8187134502924,    0.04263157894736842]
scala> summary.numNonzeros
res58: org.apache.spark.mllib.linalg.Vector = [16.0,    19.0,    19.0]
```

 # MultivariateStatisticalSummary 메소드 목록

asInstanceOf count isInstanceOf
max mean min
normL1 normL2
numNonzeros toString variance

2. Data 처리

2) 데이터 처리

- 데이터 탐색

3. RDD[Vector] Correlations

scala> **val correlMatrix: Matrix = Statistics.corr(features, "pearson")**

correlMatrix: org.apache.spark.mllib.linalg.Matrix =

1.0	**0.8814343025316858**	**0.14417085866598092**
0.8814343025316858	**1.0**	**0.1787639407161585**
0.14417085866598092	**0.1787639407161585**	**1.0**

2. Data 처리

2) 데이터 처리

- 데이터 변환

1. 텍스트 변환 : TF(Term Frequency)

데이터셋 : http://qwone.com/~jason/20Newsgroups/ - 20개의 뉴스 그룹

```
scala> import org.apache.spark.mllib.feature.HashingTF
scala> val data = sc.wholeTextFiles("file:///data/spark-1.6.0/data/train.electronics/*")
scala> val text = data.map { case (file, text) => text }
scala> text.count
res3: Long = 591
scala> text.first()
res4: String = "From: keith@radio.nl.nuwc.navy.mil ...
scala> val tf = new HashingTF(numFeatures = 10000)
scala> val termF = text.map(txt => tf.transform(txt.split(" ")))
scala> termF.first()
res5: org.apache.spark.mllib.linalg.Vector =
(10000,[0,22,43,45,54,73,145,182,334,434,866,1015,1075,1563,1577,1666,1691,1786,1818,2266,2284,2752,2932,3159,3355,3480,3502,3521,3524
,3543,4145,4262,4304,4468,4773,4801,5240,5475,5636,5848,5853,5936,6048,6051,6216,6852,6979,6988,7118,7285,7525,7817,7865,8056,8642,8
661,8786,9011,9131,9467,9493,9792],[7.0,1.0,1.0,7.0,2.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,3.0,2.0,1.0,1.0,2.0,1.0,1.0,1.0,1.0,2.0,2.0,1.0,1.0,1.0,1.0,1.0,1
.0,1.0,1.0,2.0,1.0,2.0,1.0,1.0,1.0,1.0,1.0,1.0,2.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,2.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0])
```

2. Data 처리

2) 데이터 처리

- 데이터 변환

2. 속성값 정량화 : StandardScaler

```
scala> import org.apache.spark.mllib.feature.StandardScaler
scala> val example = sc.textFile("file:///data/spark-1.6.0/data/mllib/ridge-data/lpsa.data")
scala> val data = example.map { line =>
  val parts = line.split(',')
  LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.toDouble)))
}.cache()
scala> val scaler1 = new StandardScaler().fit(data.map(x => x.features))
scala> val scaler2 = new StandardScaler(withMean = true, withStd = true).fit(data.map(x => x.features))
scala> val data1 = data.map(x => LabeledPoint(x.label, scaler1.transform(x.features)))
scala> val data2 = data.map(x => LabeledPoint(x.label, scaler2.transform(Vectors.dense(x.features.toArray))))
scala> val summary: MultivariateStatisticalSummary = Statistics.colStats(data.map(x => x.features))
scala> val summary1: MultivariateStatisticalSummary = Statistics.colStats(data1.map(x => x.features))
scala> val summary2: MultivariateStatisticalSummary = Statistics.colStats(data2.map(x => x.features))
```

2. Data 처리

2) 데이터 처리

- **데이터 변환**

2. 속성값 정량화 : StandardScaler

scala> **summary.mean**

[-0.030983588171116905,-0.0066174110840905 3,0.11823713020843973,-0.01993077331580595,0.017840200894241987,-0.024915030703527408,-0.02940456060286869 6,0.06691288828053416]

scala> **summary1.mean**

[-0.029388629245711173,-0.00594831872742818 1,0.11733735411196436,-0.019755816802758758,0.017585582898541785,-0.024870818742510403,-0.02955039130781458,0.06440613582227006]

scala> **summary2.mean**

[**0.0**, **-2.220446049250313E-16**,1.1102230246251565E-16,1.1102230246251565E-16,5.551115123125783E-17,-1.1102230246251565E-16,1.1102230246251565E-16,2.220446049250313E-16]

scala> **summary.variance**

[1.111487960445174,1.2376212760140817,1.0153953695719211,1.0177903265464625,1.0291672192693442,1.0035584882413895,0.9635840566524871,1.079356884436503]

scala> **summary1.variance**

[1.0,1.0,1.0,0.9999999999999998,1.0,1.0000000000000004,0.9999999999999998,1.0000000000000004]

scala> **summary2.variance**

[1.0,1.0,1.0,0.9999999999999996,1.0000000000000004,1.0000000000000002,0.9999999999999996,1.0000000000000004]

3. MLlib Example

- **MLlib Process**
  - Process

    1단계 : 데이터 로딩

    2단계 : 학습 데이터/ 평가 데이터로 분리

    3단계 : 학습(Training)

    4단계 : 평가

    5단계 : 모델 저장

    6단계 : 서비스 활용

3. MLlib Example

- **Spark MLlib 예제**
  - MLlib를 활용한 분류기의 구현 예제 : 의사결정나무(Decision Tree Classifier)

클래스 Import

```
scala> import org.apache.spark.mllib.tree.DecisionTree
scala> import org.apache.spark.mllib.tree.model.DecisionTreeModel
```

Data Loading

```
scala> val data = MLUtils.loadLibSVMFile(sc, "file:///data/spark-1.6.0/data/mllib/sample_libsvm_data.txt")
scala> val splits = data.randomSplit(Array(0.7, 0.3))
scala> val (trainingData, testData) = (splits(0), splits(1))
```

학습 (Training)

```
scala> val numClasses = 2
scala> val categoricalFeaturesInfo = Map[Int, Int]()
scala> val impurity = "gini"
scala> val maxDepth = 5
scala> val maxBins = 32
scala> val model = DecisionTree.trainClassifier(trainingData,
            numClasses, categoricalFeaturesInfo, impurity, maxDepth, maxBins)
```

모델 평가

```
scala> val labelAndPreds = testData.map { point => val prediction = model.predict(point.features)
     | (point.label, prediction) }
scala> val testErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble
```

모델저장 및 로딩

```
scala> model.save(sc, "myModel_DTree")
scala> val DTModel = DecisionTreeModel.load(sc, "myModel_DTree")
```

3. MLlib Example

- **MLlib Example**
  - NaiveBayes 분류기 (1)

```
scala> import org.apache.spark.mllib.classification.{NaiveBayes, NaiveBayesModel}
scala> import org.apache.spark.mllib.linalg.Vectors
scala> import org.apache.spark.mllib.regression.LabeledPoint
scala> val data = sc.textFile("file:///data/spark-1.6.0/data/mllib/sample_naive_bayes_data.txt")
scala> println(data.collect().mkString("₩n"))
scala> val parsedData = data.map { line =>
     |   val parts = line.split(',')
     |   LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.toDouble)))
     | }
scala> val splits = parsedData.randomSplit(Array(0.6, 0.4), seed = 11L)
scala> val training = splits(0)
scala> val test = splits(1)
scala> training.count()
```

3. MLlib Example

- **MLlib Example**
  - NaiveBayes 분류기(2)

```
scala> val model = NaiveBayes.train(training, lambda = 1.0, modelType = "multinomial")
scala> val predictionAndLabel = test.map(p => (model.predict(p.features), p.label))
scala> val accuracy = 1.0 * predictionAndLabel.filter(x => x._1 == x._2).count()
scala> test.count()
scala> println(predictionAndLabel.collect().mkString("₩n"))
scala> val trainPrediction = training.map(p => (model.predict(p.features), p.label))
scala> val trainAccuracy = 1.0 * trainPrediction.filter(x => x._1 == x._2).count()
scala> training.count()
scala> println(training.collect().mkString("₩n"))
scala> println(trainPrediction.collect().mkString("₩n"))
scala> model.save(sc, "myModel_NaiveBayes")
scala> val NBModel = NaiveBayesModel.load(sc, "myModel_NaiveBayes")
```

3. MLlib Example

- **MLlib Example**
  - DecisionTree 분류기 (1)

```
scala> import org.apache.spark.mllib.tree.DecisionTree
scala> import org.apache.spark.mllib.tree.model.DecisionTreeModel
scala> import org.apache.spark.mllib.util.MLUtils
scala> val data = MLUtils.loadLibSVMFile(sc, "file:///data/spark-1.6.0/data/mllib/sample_libsvm_data.txt")
scala> println(data.collect().mkString("₩n"))
scala> val numClasses = 2
scala> val categoricalFeaturesInfo = Map[Int, Int]()
scala> val impurity = "gini"
scala> val maxDepth = 5
scala> val maxBins = 32
```

3. MLlib Example

- **MLlib Example**
  - DecisionTree 분류기 (2)

```
scala> val splits = data.randomSplit(Array(0.7, 0.3))
scala> val (trainingData, testData) = (splits(0), splits(1))
scala> val model = DecisionTree.trainClassifier(trainingData, numClasses, categoricalFeaturesInfo,
impurity, maxDepth, maxBins)
scala> val labelAndPreds = testData.map { point =>
     | val prediction = model.predict(point.features)
     | (point.label, prediction)
     | }
scala> val testErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble
scala> testData.count()
scala> model.save(sc, "myModel_DTree")
scala> val DTModel = DecisionTreeModel.load(sc, "myModel_DTree")
```

4. MLlib 알고리즘

- **Spark MLlib 알고리즘 목록**
  - logistic regression and linear support vector machine (SVM)
  - classification and regression tree
  - random forest and gradient-boosted trees
  - recommendation via alternating least squares (ALS)
  - clustering via k-means, bisecting k-means, Gaussian mixtures (GMM), and power iteration clustering
  - topic modeling via latent Dirichlet allocation (LDA)
  - survival analysis via accelerated failure time model
  - singular value decomposition (SVD) and QR decomposition
  - principal component analysis (PCA)
  - linear regression with L1, L2, and elastic-net regularization
  - isotonic regression
  - multinomial/binomial naive Bayes
  - frequent itemset mining via FP-growth and association rules
  - sequential pattern mining via PrefixSpan
  - summary statistics and hypothesis testing
  - feature transformations
  - model evaluation and hyper-parameter tuning

4. MLlib 알고리즘

Spark MLlib 알고리즘 분류

- **분류 및 회귀**

linear models (SVMs, logistic regression, linear regression)
decision trees
naive Bayes
ensembles of trees (Random Forests and Gradient-Boosted Trees)
isotonic regression

- **추천(Collaborative filtering)**

alternating least squares (ALS)

- **군집**

k-means, bisecting k-means, streaming k-means
Gaussian mixture
power iteration clustering (PIC)
latent Dirichlet allocation (LDA)

- **차원 축소**

singular value decomposition (SVD)
principal component analysis (PCA)

- **빈발 패턴 마이닝(Frequent pattern mining)**

FP-growth
association rules
PrefixSpan

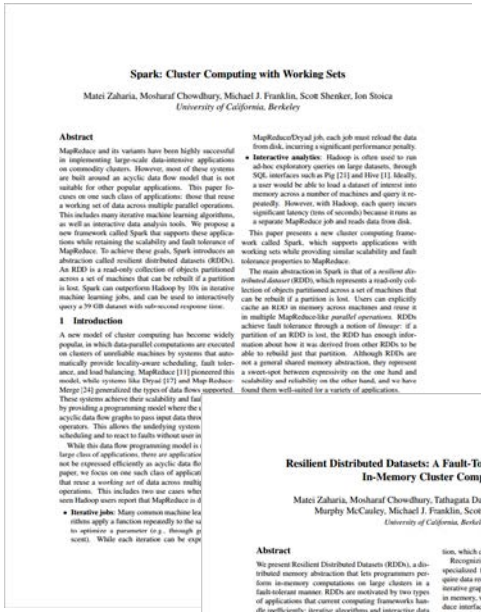
4. MLlib 알고리즘

- **분류 및 회귀(예측) 모델**

구분	모델
Binary Classification	**linear SVMs,** logistic regression, decision trees, random forests, **gradient-boosted trees,** naive Bayes
Multiclass Classification	logistic regression, decision trees, random forests, naive Bayes
Regression	linear least squares, Lasso, ridge regression, **decision trees,** **random forests,** **gradient-boosted trees,** isotonic regression

참고자료

- **Spark Papers**



Spark: Cluster Computing with Working Sets

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica

University of California, Berkeley

Abstract

MapReduce and its variants have been highly successful in implementing large-scale data-intensive applications on commodity clusters. However, most of these systems are built around an acyclic data flow model that is not suitable for other popular applications. This paper focuses on one such class of applications: those that reuse a working set of data across multiple parallel operations. This includes many iterative machine learning algorithms, as well as interactive data analysis tools. We propose a new framework called Spark that supports these applications while retaining the scalability and fault tolerance of MapReduce. To achieve these goals, Spark introduces an abstraction called resilient distributed datasets (RDDs). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Spark can outperform Hadoop by 10x in iterative machine learning jobs, and can be used to interactively query a 39 GB dataset with sub-second response time.

1 Introduction

A new model of cluster computing has become widely popular, in which data-parallel computations are executed on clusters of unreliable machines by systems that automatically provide locality-aware scheduling, fault tolerance, and load balancing. MapReduce [11] pioneered this model, while systems like Dryad [17] and Map-Reduce-Merge [24] generalized the types of data flows supported. These systems achieved their scalability and fau
by providing a programming model where the
acyclic data flow graphs to pass input data thro
operators. This allows the underlying system
scheduling and to react to faults without user in

While this data flow programming model is
large class of applications, there are applicatio
not be expressed efficiently as acyclic data flo
paper, we focus on one such class of applicat
that reuse a *working set* of data across multi
operations. This includes two use cases wh
seen Hadoop users report that MapReduce is

- **Iterative jobs**: Many common machine le
rithms apply a function repeatedly to the s
to *optimize* a *parameter* (e.g., through g
scent). While each iteration can be exp
MapReduce/Dryad job, each job must reload the data from disk, incurring a significant performance penalty.
- **Interactive analytics**: Hadoop is often used to run ad-hoc exploratory queries on large datasets, through SQL interfaces such as Pig [21] and Hive [1]. Ideally, a user would be able to load a dataset of interest into memory across a number of machines and query it repeatedly. However, with Hadoop, each query incurs significant latency (tens of seconds) because it runs as a separate MapReduce job and reads data from disk.

This paper presents a new cluster computing framework called Spark, which supports applications with working sets while providing similar scalability and fault tolerance properties to MapReduce.

The main abstraction in Spark is that of a *resilient distributed dataset* (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like *parallel operations*. RDDs achieve fault tolerance through a notion of *lineage*: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition. Although RDDs are not a general shared memory abstraction, they represent a sweet-spot between expressivity on the one hand and scalability and reliability on the other hand, and we have found them well-suited for a variety of applications.

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica

University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that *reuse* intermediate results across multiple computations. Data reuse is common in many *iterative* machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is *interactive* data mining, where a user runs multiple ad-hoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (e.g., between two MapReduce jobs) is to write it to an external stable storage system, e.g., a distributed file system. This incurs substantial overheads due to data replication, disk I/O, and serialization, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (e.g., looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, e.g., to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance efficiently. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (e.g., cells in a table). With this interface, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead.

In contrast to these systems, RDDs provide an interface based on *coarse-grained* transformations (e.g., map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data.[1] If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute

[1] Checkpointing the data in some RDDs may be useful when a lineage chain grows large, however, and we discuss how to do it in §5.4.

- **Spark RDD**

Spark: Cluster Computing with Working Sets

June 2010

http://www.cs.berkeley.edu/~matei/papers/2010/hotcloud_spark.pdf

- **Spark RDD : Fault Tolerant & In-Memory**

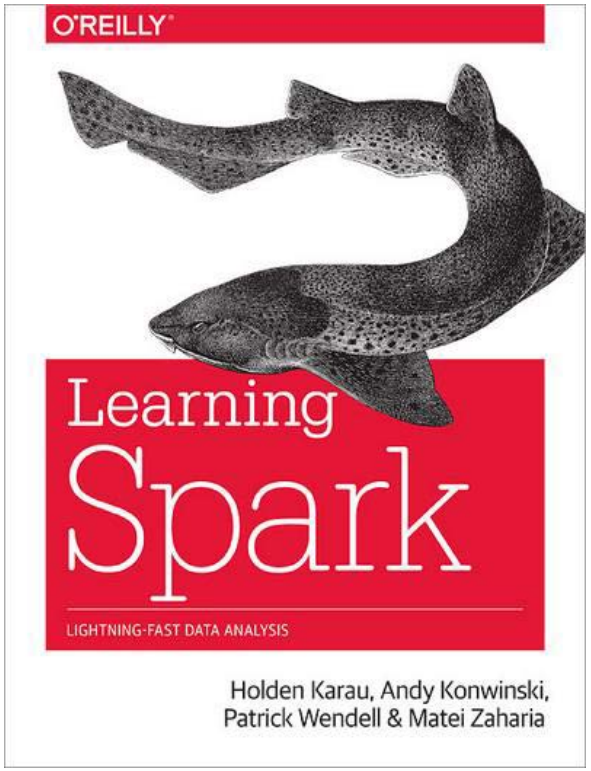
Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

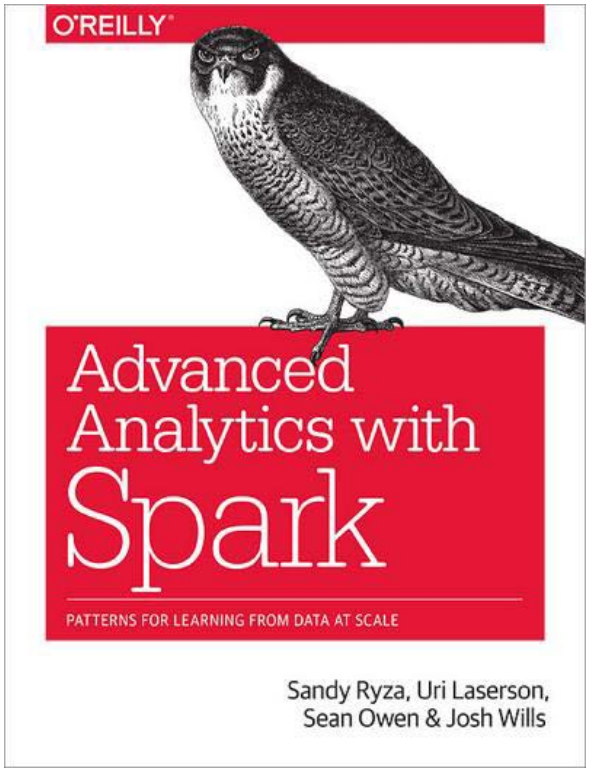
April 2012

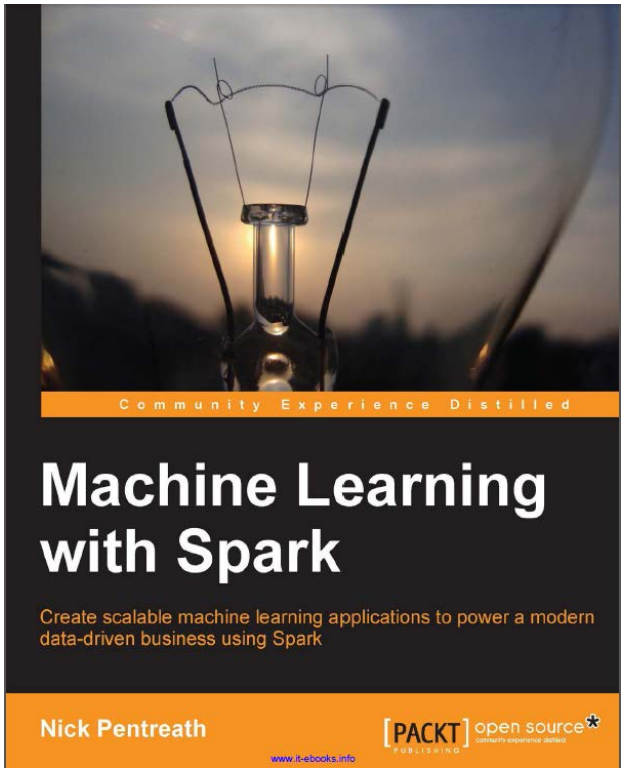
http://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

참고자료

- **Spark Books**







감사합니다.