# Hadoop Eco-system

CM1-2팀 박주원

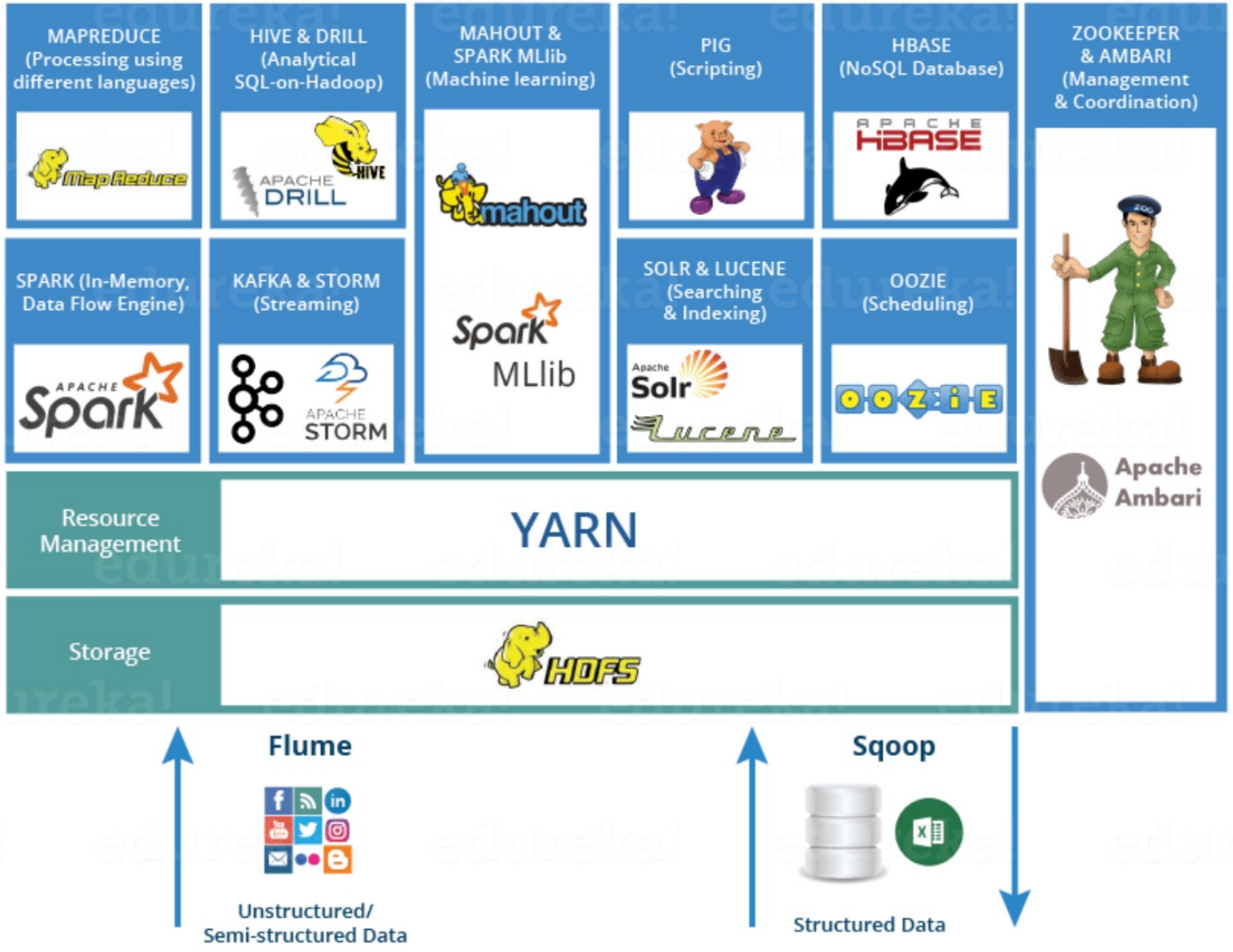
# Hadoop

- 대용량 데이터를 분산 저장 및 처리할 수 있는 자바 기반의 오픈소스 프레임워크
- 단일 노드의 성능을 Scale-up 하기에는 한계가 있음 (비용, 성능) → Scale-out하여 여러 노드를 활용하여 분산 처리
- 분산 파일 시스템인 HDFS (Hadoop Distributed File System)에 데이터를 저장하고 분산 처리 시스템인 MapReduce를 이용해 데이터를 처리

# Hadoop

- 2003년 구글파일시스템 (GFS), 2004년 구글의 MapReduce 논문을 토대로 야후 개발자 Doug Cutting이 Hadoop 프로젝트 수행 – 2006년
- Hadoop 프로젝트 하위의 모듈
  - Hadoop Common
  - Hadoop Distributed File System (HDFS)
  - Hadoop MapReduce
  - Hadoop YARN
  - Hadoop Ozone

# Hadoop Eco-system



# HDFS

- GFS (Google File System)을 기반으로 개발된 분산 파일 시스템
- GFS
  - Master-slave Architecture
- 구글 플랫폼 철학
  - 한대의 고가 장비보다 여래 대의 저가 장비 (Scale up < Scale out)
  - 데이터 분산 저장
    - Parallel Computing
    - Distributed Computing
  - System (H/W)은 언제든 죽을 수 있다.
    - Fault Tolerance 설계 필요
  - 시스템 확장 용이
    - 노드 수를 늘려도 평소처럼 작동

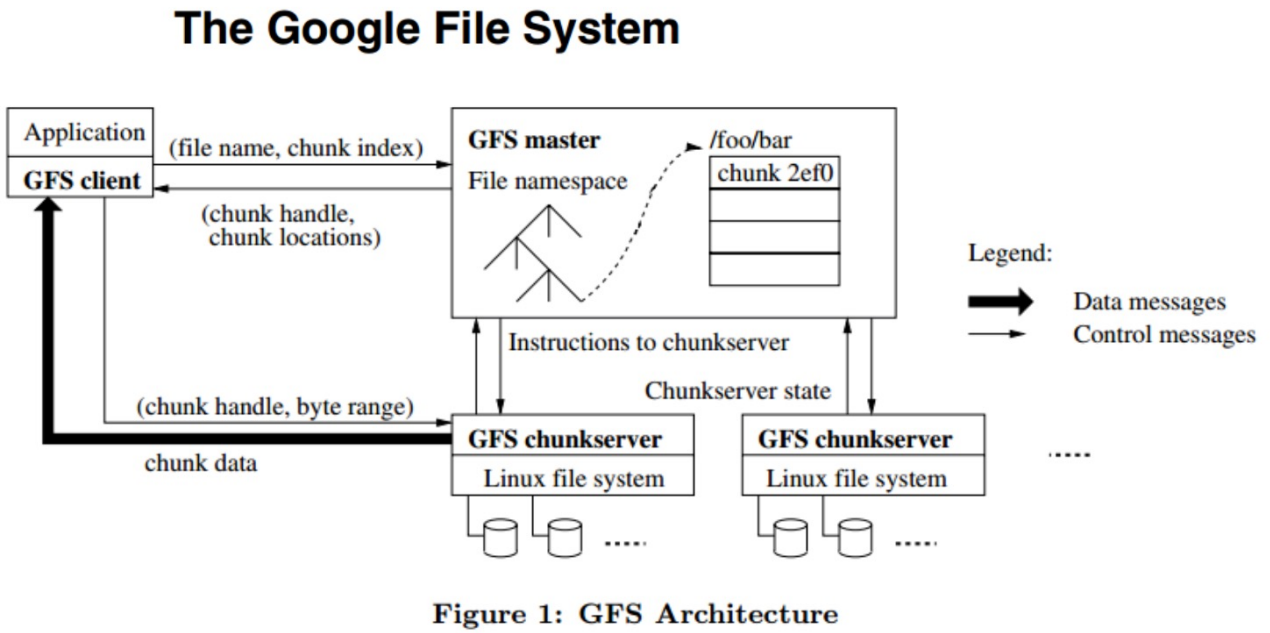
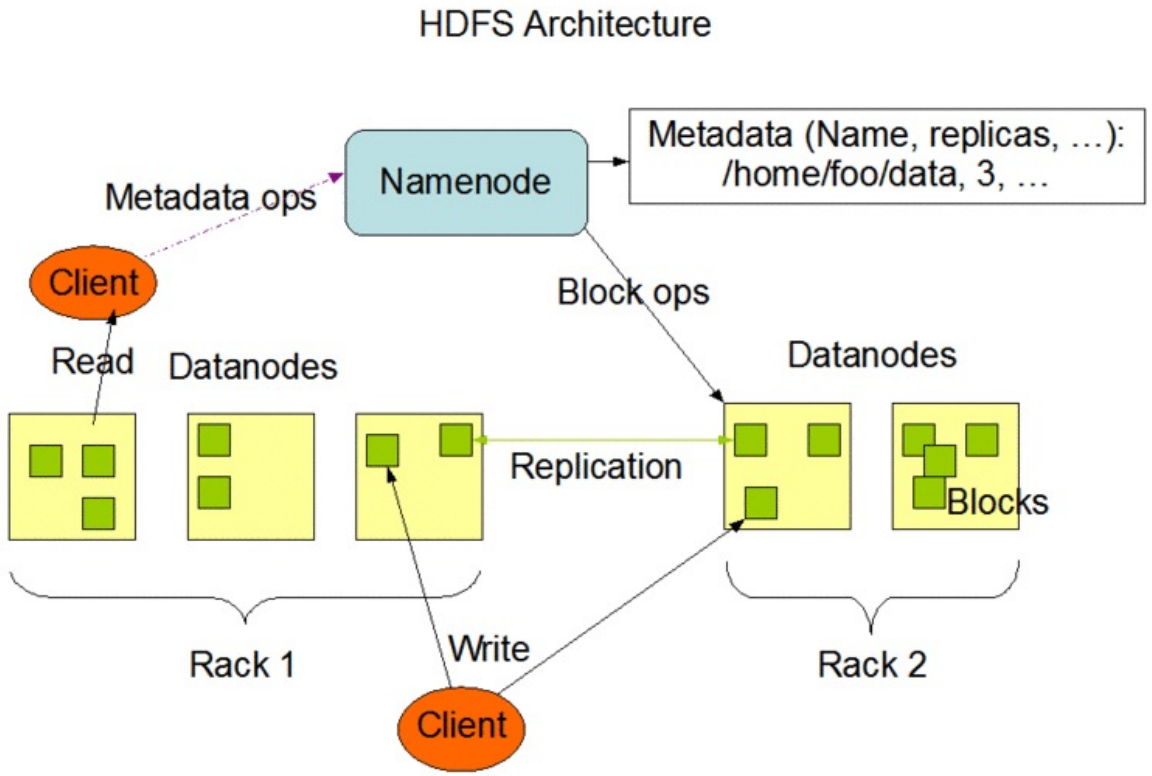



Figure 1: GFS Architecture

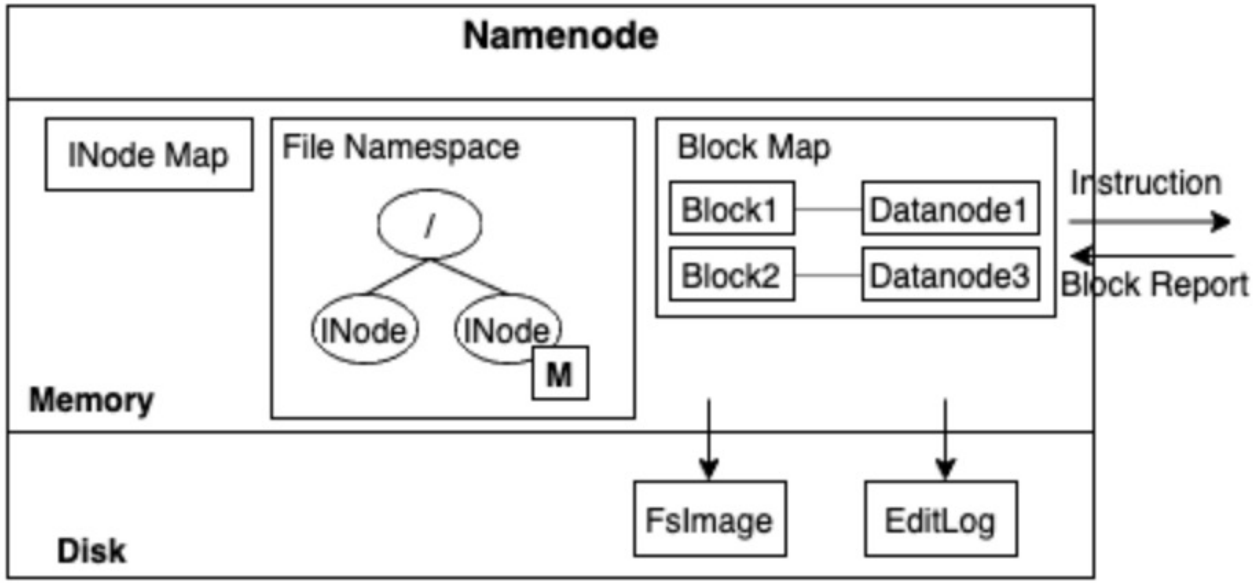
# HDFS

- HDFS 특성
  - 수천대 이상의 리눅스 기반 범용 서버들을 하나의 클러스터로 사용
  - Master-slave Architecture
    (Master – NameNode, Slave - DataNode)
  - 파일은 Block 단위로 저장
    (default 128MB)
  - 블록 데이터의 복제본 유지로 인한 신뢰성 보장
    (3 Replications, Erasure Coding)



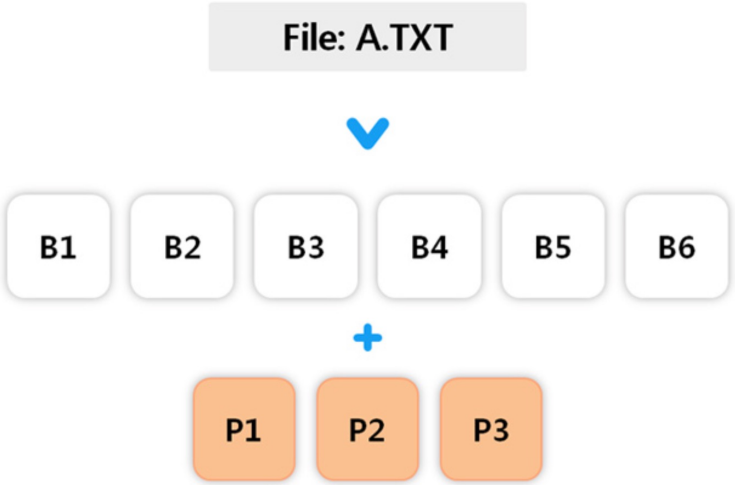
# HDFS – Master-Slave Architecture

- NameNode
  - Namespace Management
    - Namespace
    - BlockMap
    - InodeMap
  - Replica Placement
  - Checkpoint
- DataNode
  - Block Data
  - Block Report
  - Heartbeat



# HDFS – Fault Tolerance

- 3 Replication
  - 3개의 Replica를 만들어 같은 랙에 있는 다른 노드와 다른 랙에 있는 다른 노드에 분산하여 저장
  - 데이터 저장 공간 비용이 큼
- Erasure Coding
  - Hadoop 3 부터 추가
  - Reed-Solomon 알고리즘 활용
  - 3 replication → 3N 만큼의 스토리지 필요, Erasure Coding → 1.5N 만큼의 스토리지 필요




| | | | |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |
| x1 | x2 | x3 | x4 |
| x5 | x6 | x7 | x8 |

X

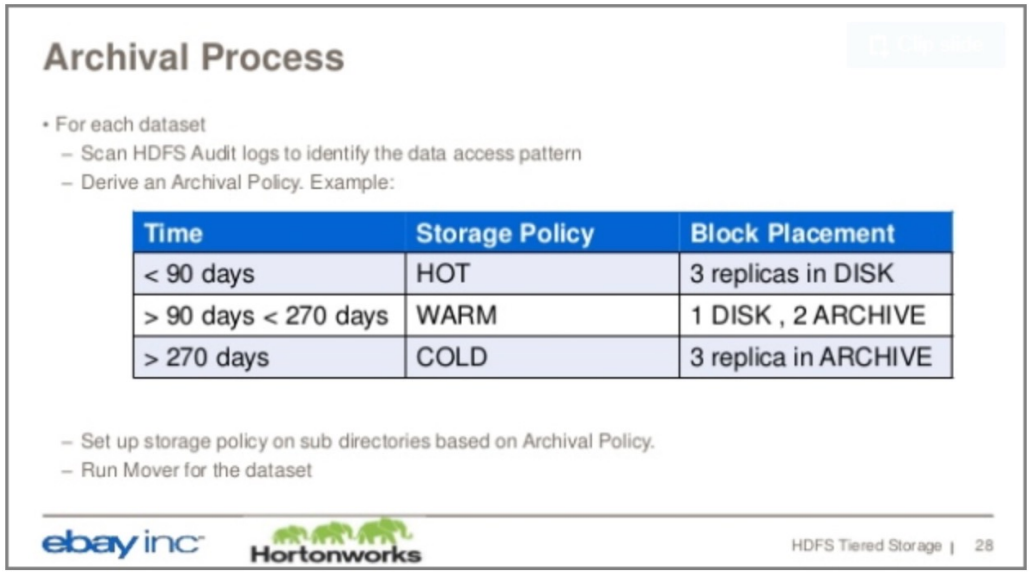
| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

=

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | |
| 5 | 6 | 7 | 8 | |
| 9 | 10 | 11 | 12 | |
| 13 | 14 | 15 | 16 | |
| c1 | c2 | c3 | c4 | Parity Block 1 |
| c5 | c6 | c7 | c8 | Parity Block 2 |

# HDFS – Heterogeneous Storage Policy

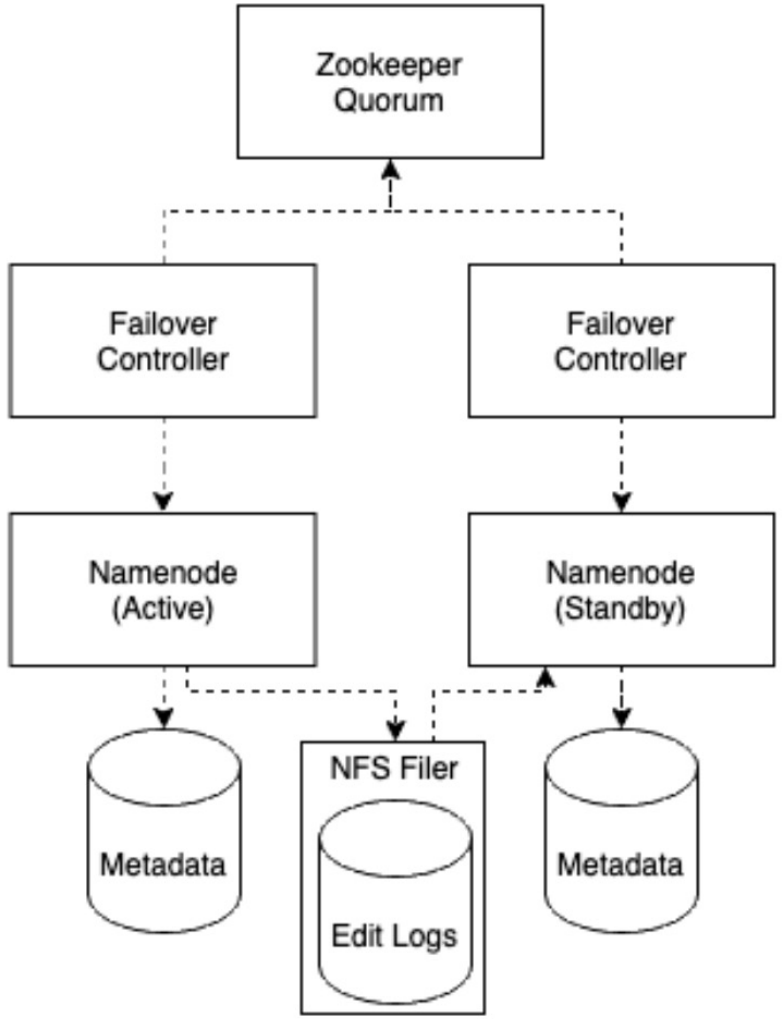
- Hadoop은 모든 노드가 Homogeneous하다고 여기고 동작
  - Data Access 빈도에 따른 차별화된 저장 불가
- Hadoop 3에서 부터 Heterogeneous Storage Policy 추가
  - Hot: 모든 복제본이 DISK에 저장
  - Cold: 모든 복제본이 ARCHIVE에 저장
  - Warm: 하나의 복제본은 HDD에 저장되고, 나머지 복제본은 ARCHIVE에 저장
  - All_SSD: 모든 복제본이 SSD에 저장
  - One_SSD: 하나의 복제본은 SSD에 저장되고, 나머지 복제본은 DISK에 저장
  - Lazy_Persist: 복제본이 RAM_DISK에 저장된 다음 추후 DISK에 저장



# HDFS – High Availability

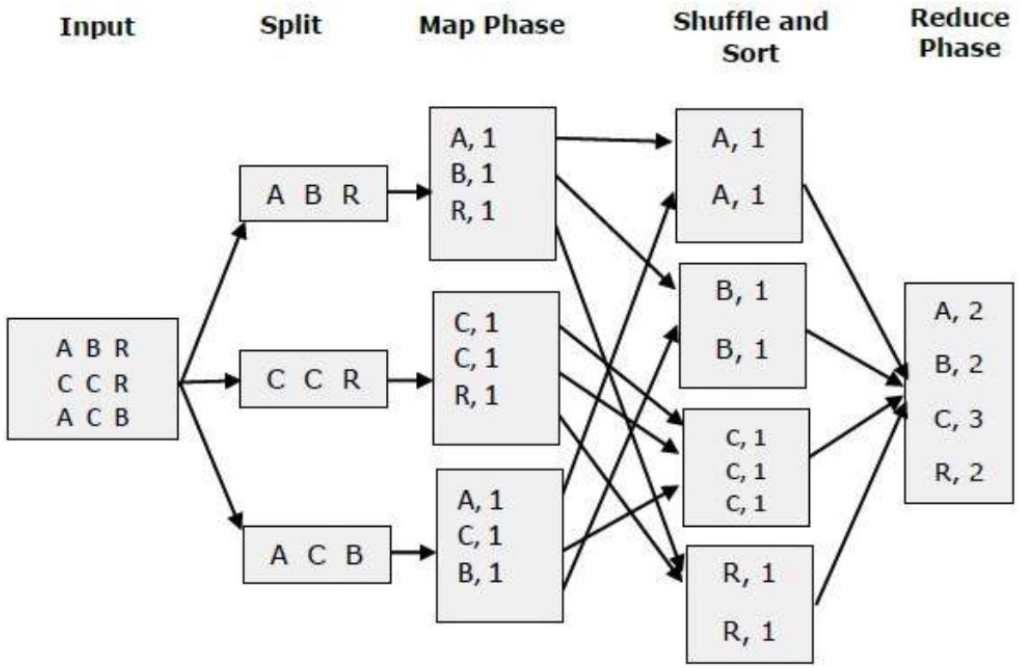
- NameNode가 고장나도 HDFS를 문제없이 사용할 수 있는 아키텍처
- Active NameNode가 고장나면, Standby NameNode가 Active NameNode로 동작
- NameNode HA Architecture
  - Active NameNode / Standby NameNode
  - Journal Node
  - Zookeeper
  - ZKFC (Zookeeper Failover Controller)

| | Active Namenode | Standby Namenode |
|---|---|---|
| 클라이언트 요청 받기 | O | X |
| 체크포인팅 수행 | X | O |
| 저널노드에 edits log 쓰기 | O | X (읽기만 함) |
| 데이터노드에게 heartbeat 받기 | O | O |
| 데이터노드에게 blockreport 받기 | O | |



# Hadoop MapReduce

- 구글의 Jeffrey Dean의 MapReduce 논문을 기반으로 한 분산 처리 프로그래밍 모델
- MapReduce는 프로그래밍 모델임과 동시에 구현체를 부르는 말
- 분산 처리 엔진 역할을 하는 Hadoop의 중심 모듈 중 하나
- Map 함수를 설정하여 중간 결과물 형태의 key/value 쌍 데이터 생성 →
  Reduce 함수를 설정하여 같은 key를 가진 값들을 합쳐서 최종 결과물 생성



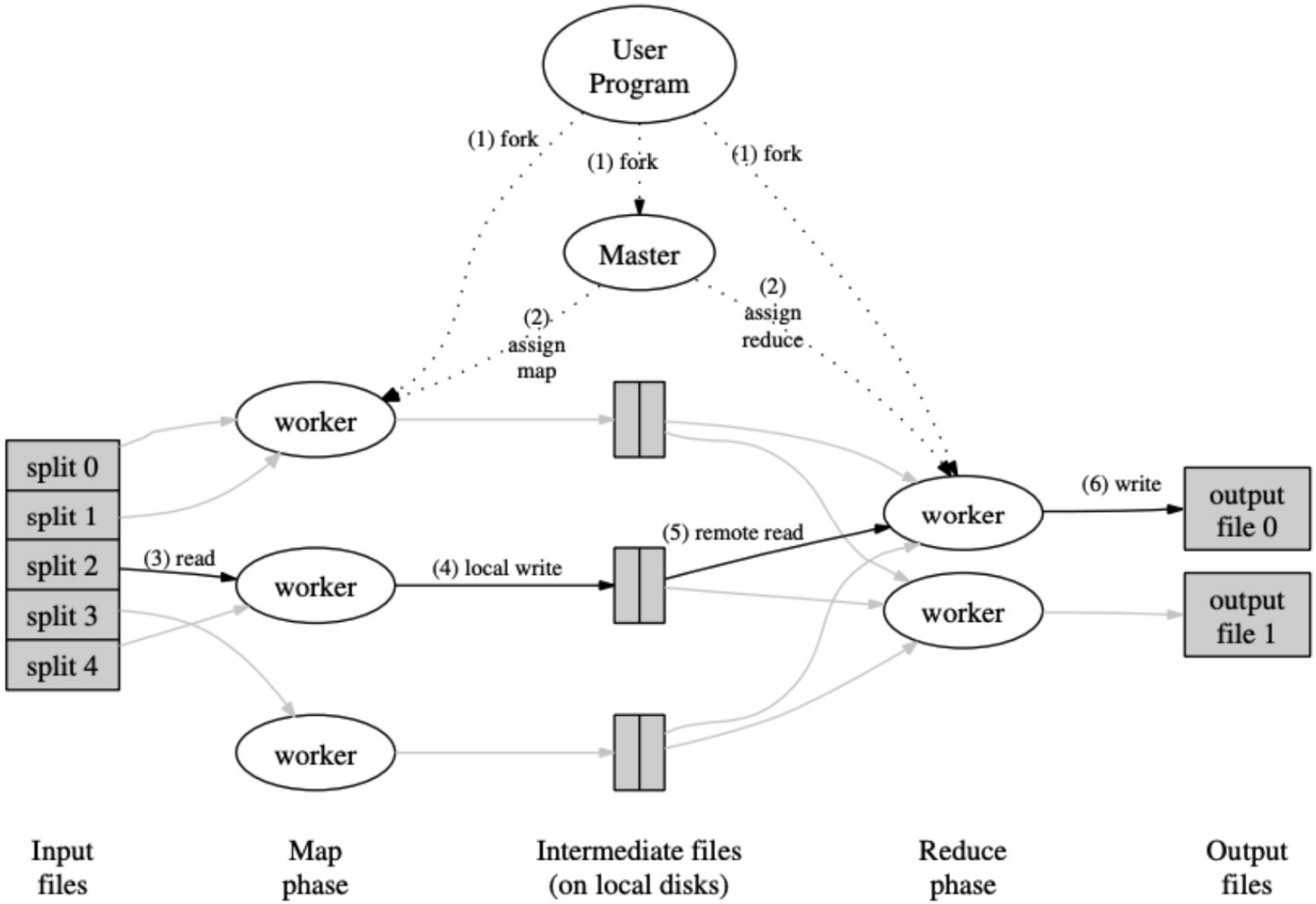
# Hadoop MapReduce

```
public static class TokenizerMapper
     extends Mapper<Object, Text, Text, IntWritable>{

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

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

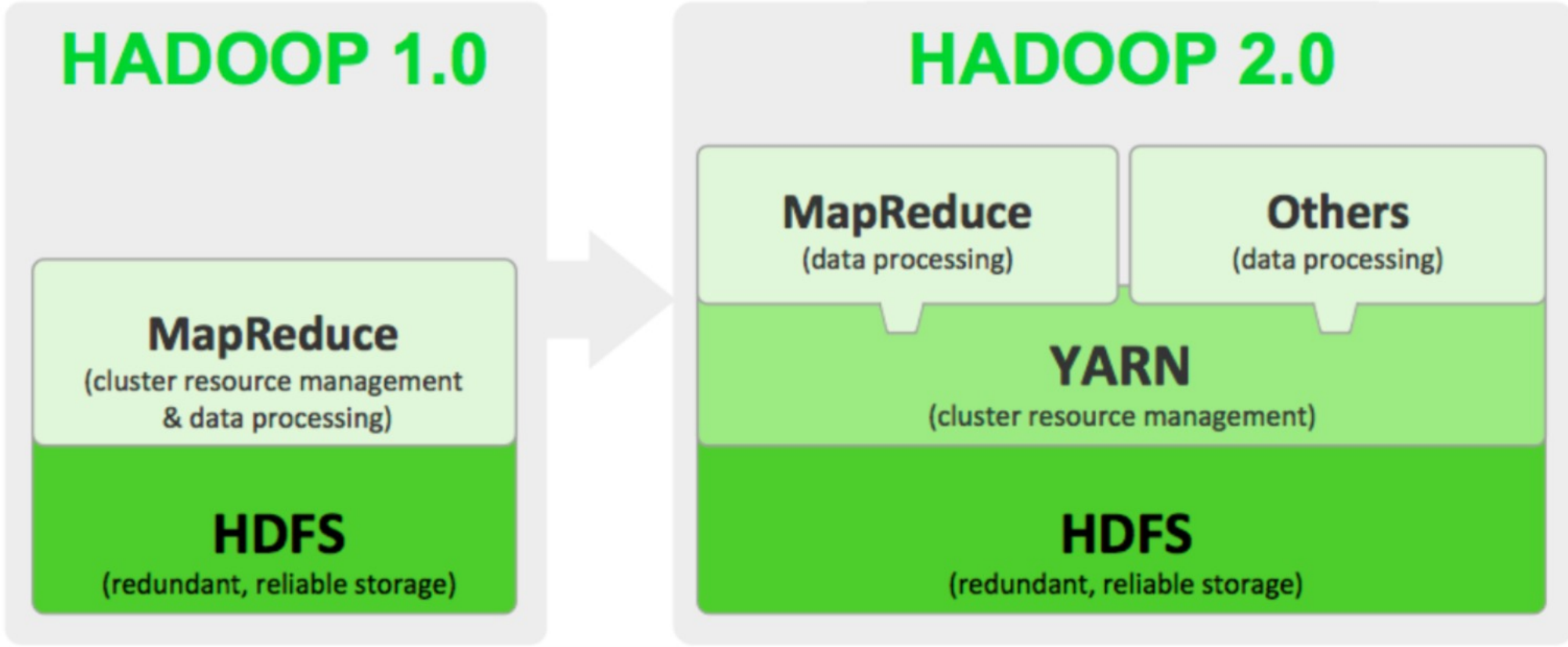
```
public static class IntSumReducer
     extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

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

```
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = Job.getInstance(conf, "word count");
  job.setJarByClass(WordCount.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setCombinerClass(IntSumReducer.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

# Hadoop MapReduce

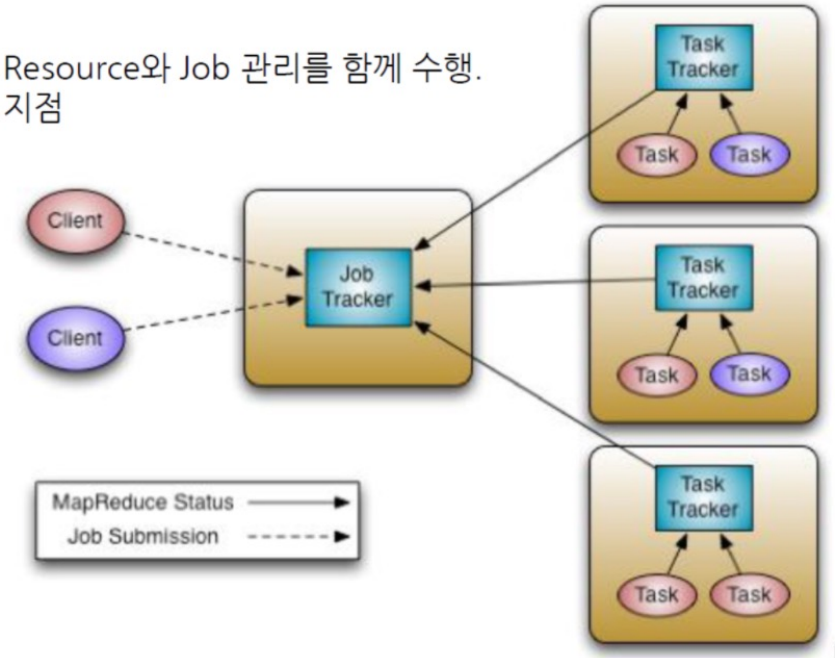


# Hadoop Yarn


HADOOP 1.0
MapReduce
(cluster resource management
& data processing)
HDFS
(redundant, reliable storage)
HADOOP 2.0
MapReduce
(data processing)
Others
(data processing)
YARN
(cluster resource management)
HDFS
(redundant, reliable storage)

## Hadoop 1 MapReduce

**Master Node**
하둡 클러스터의 Resource와 Job 관리를 함께 수행.
병목이 발생하는 지점

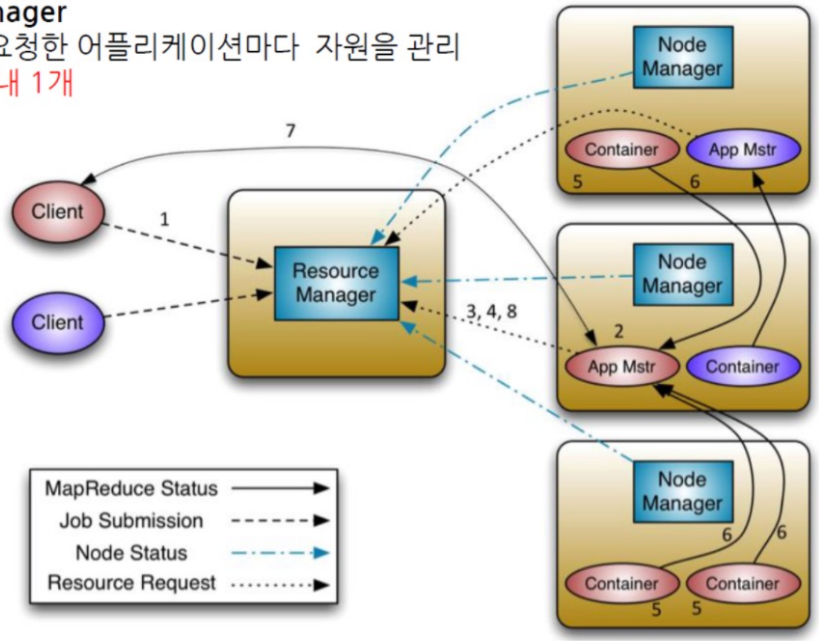



**Data Node**
한 노드에서 실행할 수 있는
Map과 Reduce Task의 개수 제한.

M/R만 처리

## Hadoop 2 MapReduce

**Resource Manager**
클라이언트가 요청한 어플리케이션마다 자원을 관리
하둡 클러스터 내 1개



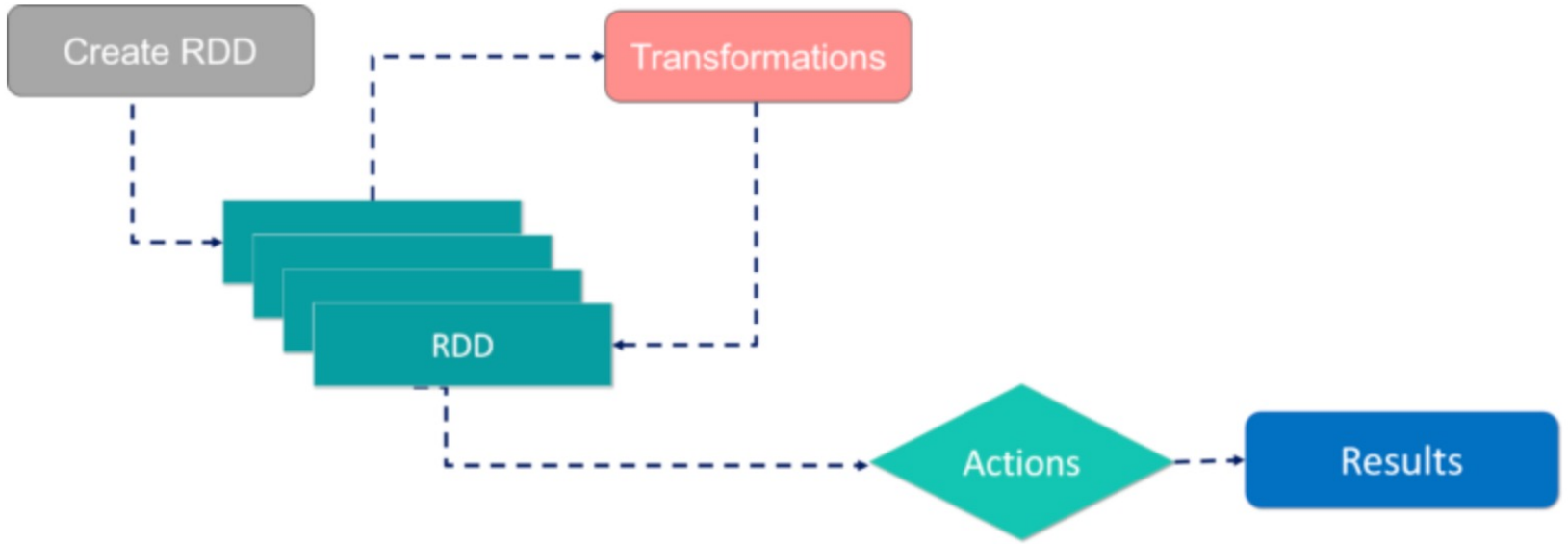

**Node Manager**
각 슬레이브 노드 마다 1개.
컨테이너와 자원의 상태를
RM에게 통지

**Application Master**
어플리케이션의 실행을 관리하고
상태를 RM에게 통지
어플리케이션마다 1개.

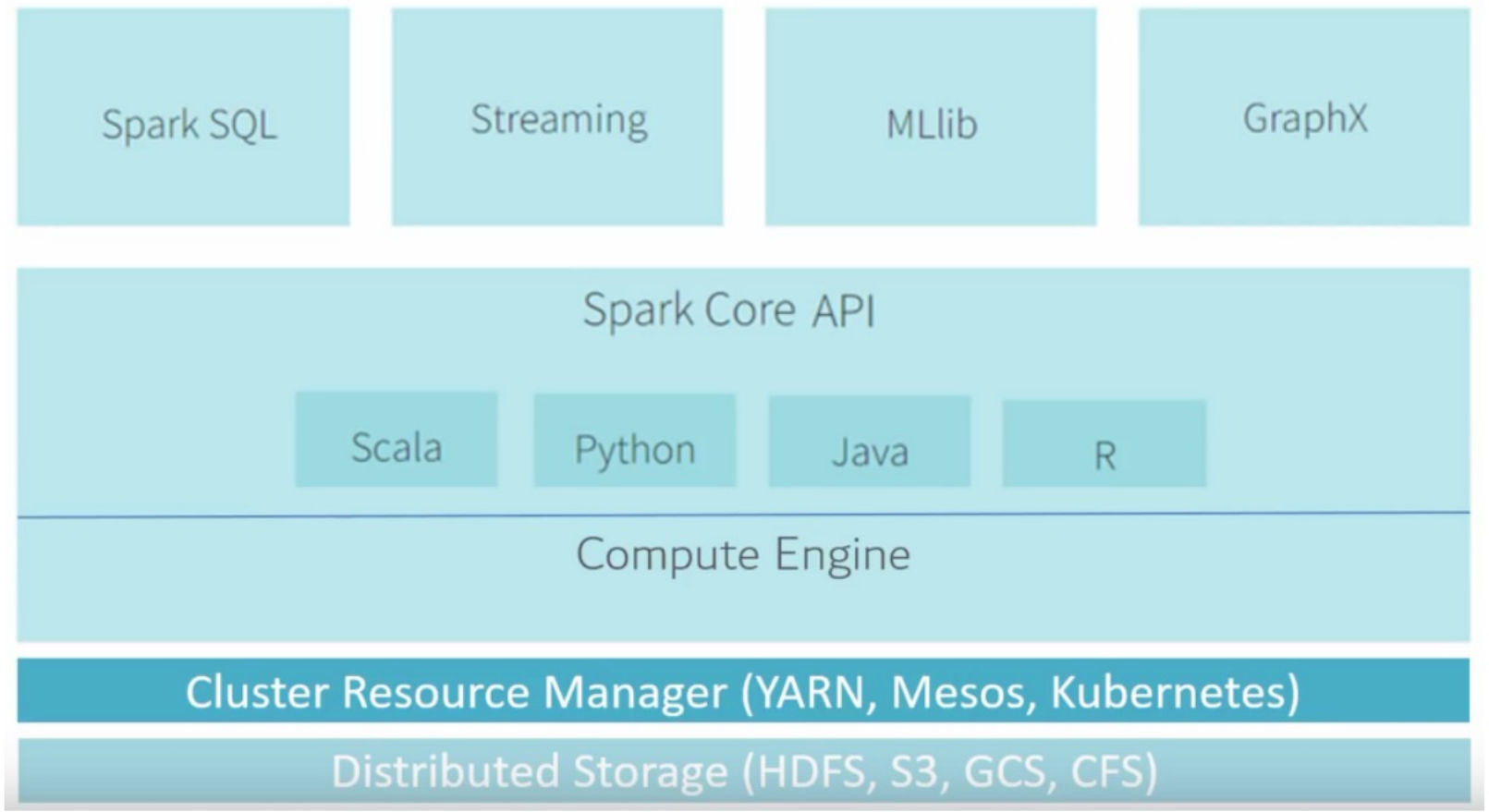
**Container**
어플리케이션을 수행하는 역할
제한된 자원을 소유하며,
상태를 AM에게 통지

# Spark

- Hadoop MapReduce의 단점을 보완 → 인-메모리 프로세싱
- Resilient Distributed Datasets (RDD) + Directed Acyclic Graph (DAG)

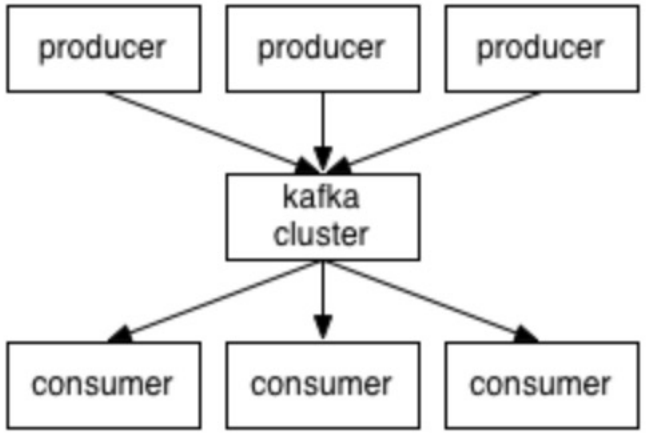


# Spark


Spark SQL
Streaming
MLlib
GraphX
Spark Core API
Scala
Python
Java
R
Compute Engine
Cluster Resource Manager (YARN, Mesos, Kubernetes)
Distributed Storage (HDFS, S3, GCS, CFS)

# Kafka

- Linkedin에서 개발된 분산 메시징 시스템
- 대용량 실시간 로그 처리에 특화된 아키텍쳐 설계



# Kafka

- 기존 메시징 시스템 (ActiveMQ, RabbitMQ)과의 차이점
- 대용량의 실시간 로그 처리에 특화되어 설계된 메시징 시스템으로써 기존 범용 메시징 시스템대비 TPS가 매우 우수
- 분산 시스템을 기본으로 설계되었기 때문에, 분산 및 복제 구성이 쉬움
- AMQP 프로토콜이나 JMS API를 사용하지 않고 단순한 메시지 헤더를 지닌 TCP기반의 프로토콜을 사용하여 프로토콜에 의한 오버헤드를 감소
- 다수의 메시지를 batch형태로 broker에게 한 번에 전달할 수 있어 TCP/IP 라운드트립 횟수를 줄임
- 메시지를 기본적으로 메모리에 저장하는 기존 메시징 시스템과는 달리 메시지를 파일 시스템에 저장
- Kafka는 consumer가 broker로부터 직접 메시지를 가지고 가는 pull 방식으로 동작

# Kafka

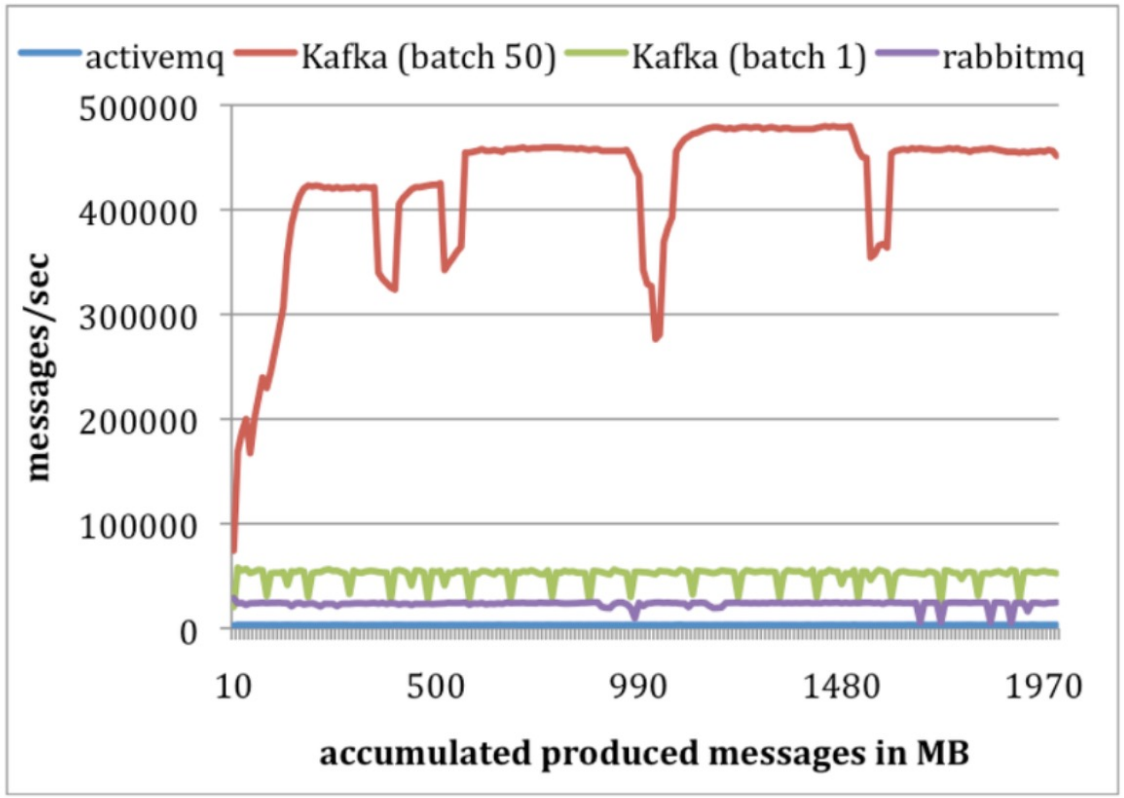



**Figure 4**. Producer Performance

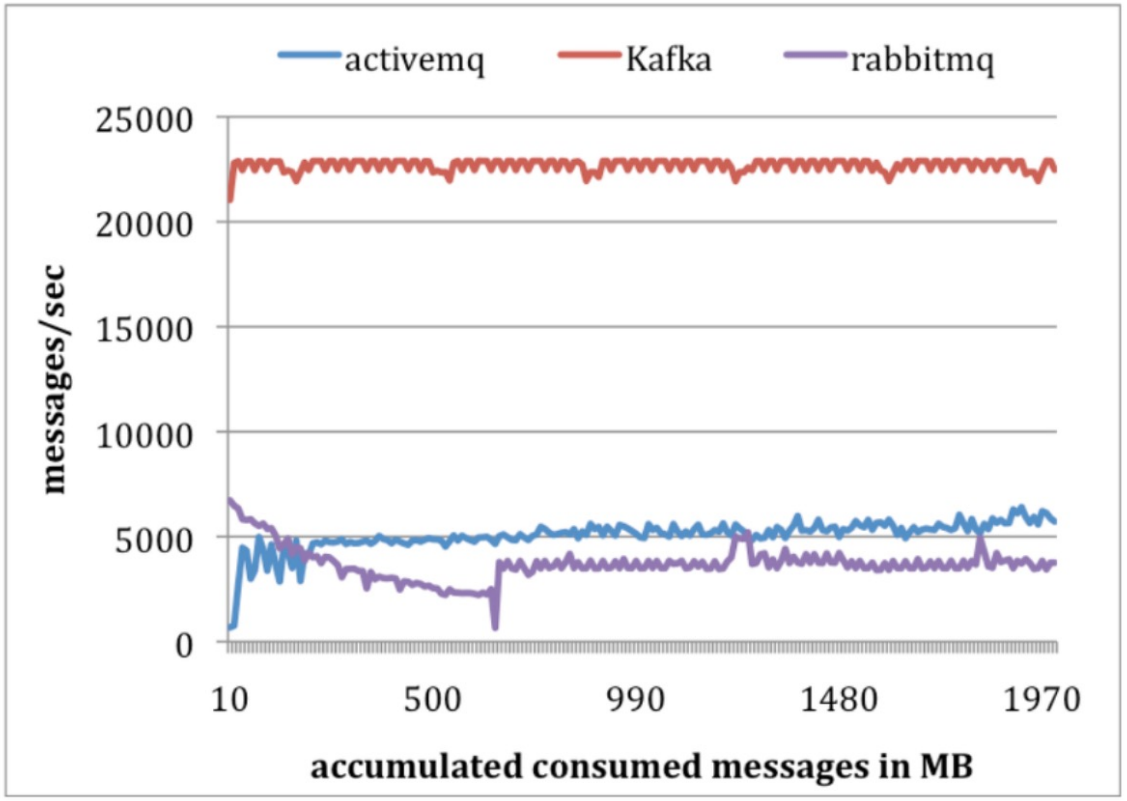



**Figure 5**. Consumer Performance

# HBase

- 구글의 Big Table을 모델로 하여 개발된 컬럼 기반 분산 데이터베이스
- 관계형 구조가 아니며, SQL을 지원하지 않음 → 비구조화된 데이터에 더욱 적합
- 수평적으로 확장성이 있어 큰 테이블에 적합
- Rowkey에 대한 인덱싱만을 지원, Zookeeper를 이용한 고가용성 보장

| rowkey | cf:name | | cf:contact | | | cf:company |
|---|---|---|---|---|---|---|
| | lastname | firstname | phone | mobile | email | company |
| 000AB | Cho | Terry | 010-123-1234 | 010-111-1111 | terry@mycom | mycom |
| 000CD | Cho | Micheal | 010-123-1234 | 010-112-1111 | mychel@mycom | mycom |
| 39AD | Kim | Hyunwoo | 010-292-1231 | 010-282-1827 | kim@yourcom | yourcom |
| 812AC | Lee | Jack | 02-567-0913 | 010-281-9928 | lee@theircom | theircom |

# Hive

- Hadoop 환경에서 데이터 웨어하우징과 같은 SQL 기반의 접근을 통한 데이터 처리
- 데이터베이스, 테이블, 파티션과 같은 논리적 레벨의 카탈로그 및 메타스토어 제공 → 데이터를 구조화

```
CREATE TABLE docs (line STRING);

LOAD DATA INPATH 'docs' OVERWRITE INTO TABLE docs;

CREATE TABLE word_counts AS
SELECT word, count(1) AS count FROM
  (SELECT explode(split(line, '\s')) AS word FROM docs) w
GROUP BY word
ORDER BY word;
```

# Ozone

- 기존 HDFS의 단점을 보완하고, 클라우드 네이티브(Cloud-Native), 컴퓨팅과 스토리지간 분리된 환경을 지원하는 차세대 HDFS 기술로, 분산형 객체 저장소
- 기존 HDFS는 NameNode가 NameSpace와 BlockSpace 모두를 메모리 영역에서 관리하여, 물리적으로 약 3.5억개 이상의 파일처리에 한계가 존재
- HDFS는 안정성을 위해 데이터를 복제, 주기적으로 NameNode에 Block 단위로 리포트하나, 데이터가 증가하면 Block도 증가하면서 리포트 자체가 무거워져 NameNode에 부하가 가중
- HDFS의 Default Block 크기는 128MB이며, 기존 Block 크기보다 현저하게 작은 용량의 파일이 다수 존재하면, 개별 파일을 관리하기 위한 메타데이터가 폭증, 이로 인해 NameNode Memory의 비효율적 사용 및 RPC 호출, Block 스캐닝 처리성능 저하, 관련된 애플리케이션의 성능 저하 등의 Small File Problem 발생

# Ozone

- **확장성 제공**
  - HDFS NameNode에서 관리하던 NameSpace와 BlockSpace를 분리하여 수십억개의 파일과 작은 파일을 저장할 수 있도록 설계
- **일관성 제공**
  - HDFS가 Zookeeper를 활용하는 반면, 장애가 발생하는 환경에서도 데이터에 대한 강력한 일관성을 제공하기 위해 분산합의 알고리즘인 RAFT(레프트) 프로토콜을 활용
- **클라우드 네이티브(Cloud-Native) 지원**
  - 클라우드 환경에서 운영하는 것이 전제인 클라우드 네이티브 지원을 위해, YARN 및 Kubernetes와 같은 Container 환경에서 잘 작동하도록 설계
- **보안성 지원**
  - 데이터 저장시점 암호화, 데이터 통신시점 암호화, 사용자 인증을 위한 Kerberos(커버로스), 데이터 접근제어를 위한 Apache Ranger(아파치 레인저)를 지원
- **다중 프로토콜 지원**
  - Hadoop 파일 시스템 API, Amazon S3 API 등을 통해 On-Premise와 Public Cloud에 이르는 단일 Storage 아키텍처 구성이 가능
- **호환성 제공**
  - Ozone 파일시스템(Ofs, O3fs)은 Hadoop 호환 파일시스템이며, Hive, Impala, Spark 같은 기존 Hadoop Echosystem과 연동 가능
- **고가용성 제공**
  - HDDS(Hadoop Distributed Data Store)라는 고 가용성 분산복제 Block Storage 계층을 통해 여러번의 장애에도 서비스가 가능하도록 구성

# WAPL Pay with Hadoop Eco-system

- Elastic Stack을 사용해 수집한 데이터 중 보관 기간을 넘긴 데이터를 삭제하지 않고 스냅숏을 HDFS에 저장
- 주문 내역 등 사용자 데이터를 수집 활용하기 위한 데이터 파이프라인 구축
- WAPL Pay 광고 시스템 구축에 활용