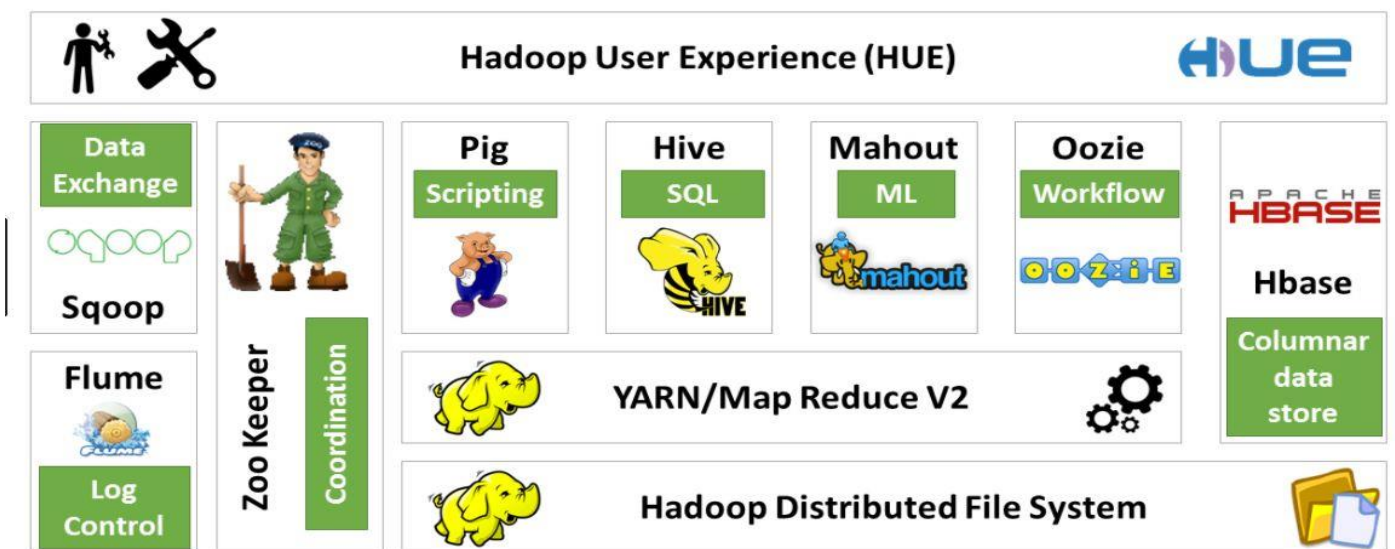


A file system to manage the storage of data

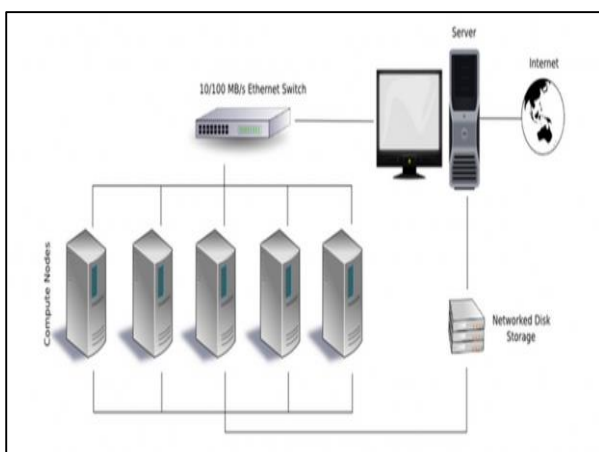
A framework to process data across multiple servers

## [ Hadoop & Eco System ]

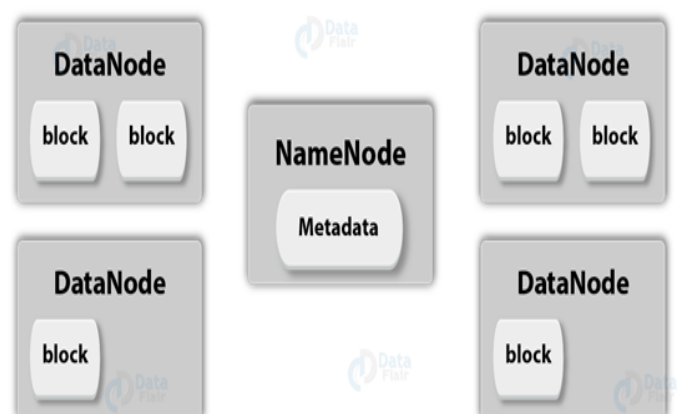


## [ 클러스터 ]

여러 대의 컴퓨터들이 연결되어 하나의 시스템처럼 동작하는 컴퓨터들의 집합



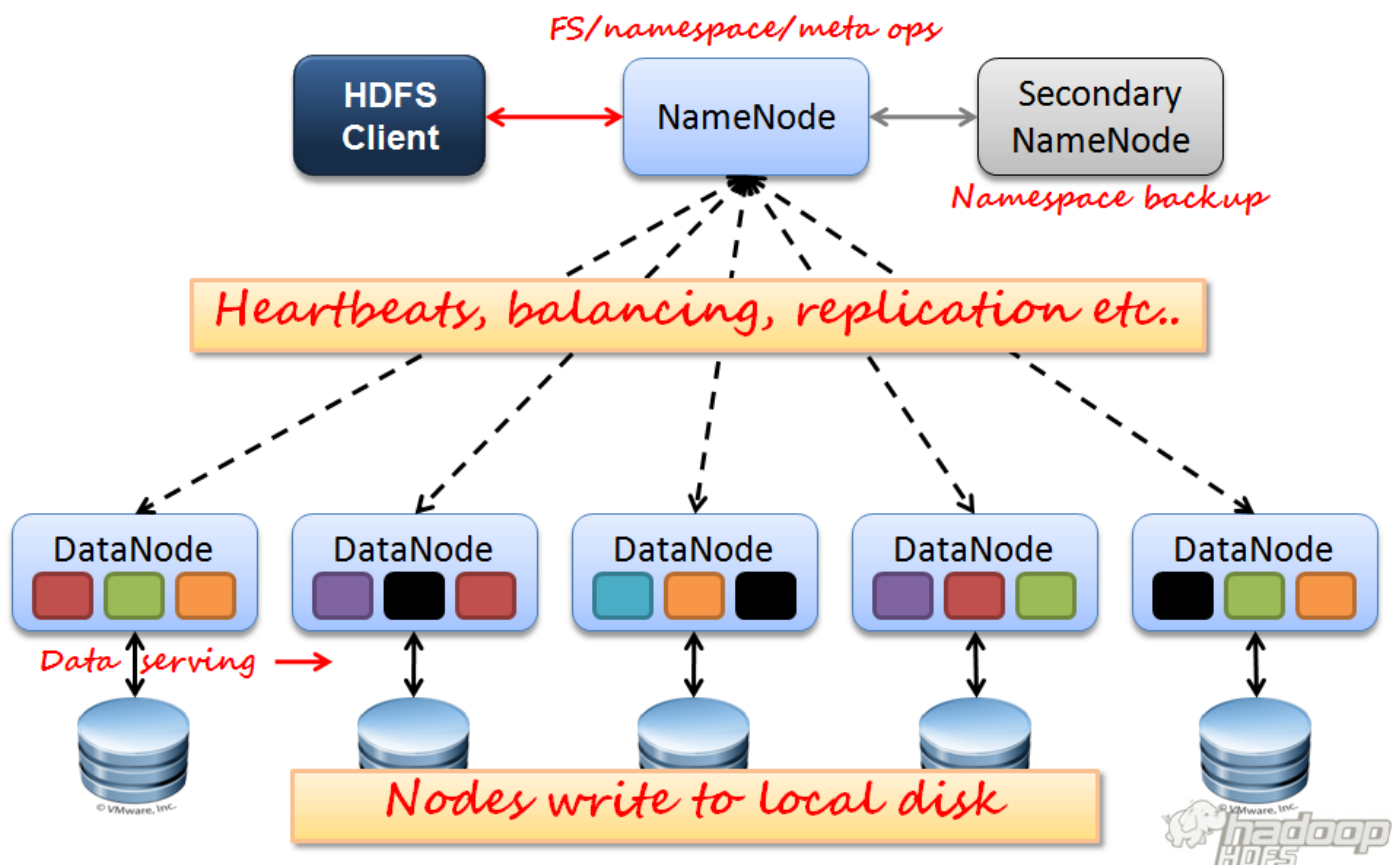
## Hadoop Cluster



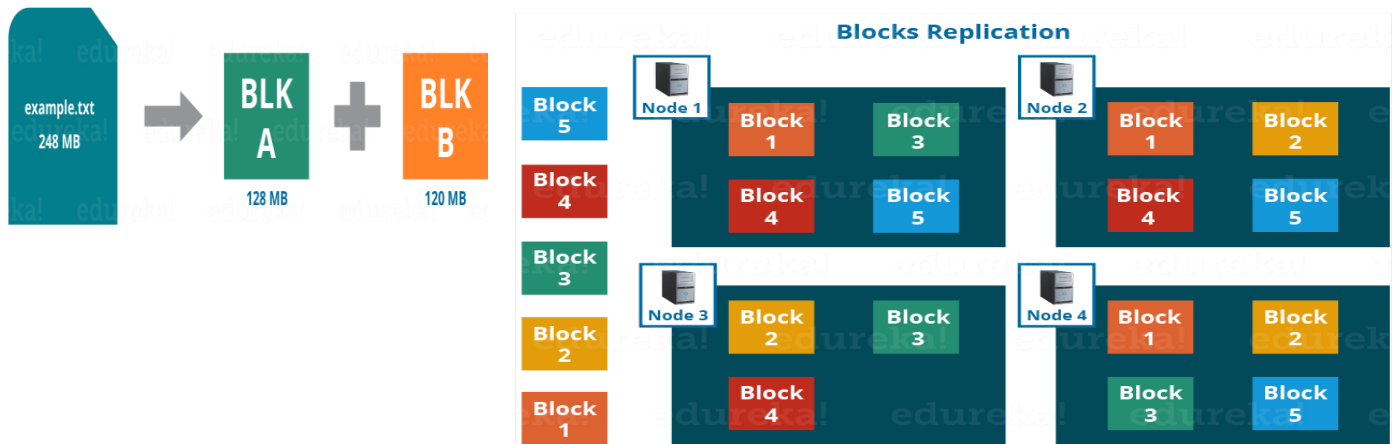
## [ 분산 컴퓨팅 시스템 ]



## [ Hadoop Cluster의 구조 ]



## [ Data Block ] – 블록 사이즈( 디폴트 : 128M)



## [ Hadoop HDFS 명령어 ]

<https://hadoop.apache.org/docs/r2.7.7/hadoop-project-dist/hadoop-common/FileSystemShell.html>

`hdfs dfs -ls /[디렉토리명 또는 파일명]`

지정된 디렉토리의 파일리스트 또는 지정된 파일의 정보를 보여준다.

`hdfs dfs -ls -R /[디렉토리명]`

지정된 디렉토리의 파일리스트 및 서브디렉토리들의 파일 리스트도 보여준다.

`hdfs dfs -mkdir /디렉토리명`

지정된 디렉토리를 생성한다.

`hdfs dfs -cat /[디렉토리/]파일`

지정된 파일의 내용을 화면에 출력한다.

`hdfs dfs -put 사용자계정로컬파일 HDFS디렉터리[/파일]`

지정된 사용자계정 로컬 파일시스템의 파일을 HDFS 상 디렉터리의 파일로 복사한다.

`hdfs dfs -get HDFS디렉터리의파일 사용자계정로컬 디렉터리[/파일]`

지정된 HDFS상의 파일을 사용자계정 로컬 파일시스템의 디렉터리나 파일로 복사한다.

`hdfs dfs -rm /[디렉토리]/파일`

지정된 파일을 삭제한다.

`hdfs dfs -rm -r /디렉토리`

지정된 디렉토리를 삭제. 비어 있지않은 디렉터리도 삭제하며 서브 디렉터리도 삭제한다.

`hdfs dfs -tail /[디렉토리]/파일`

지정된 파일의 마지막 1kb 내용을 화면에 출력한다.

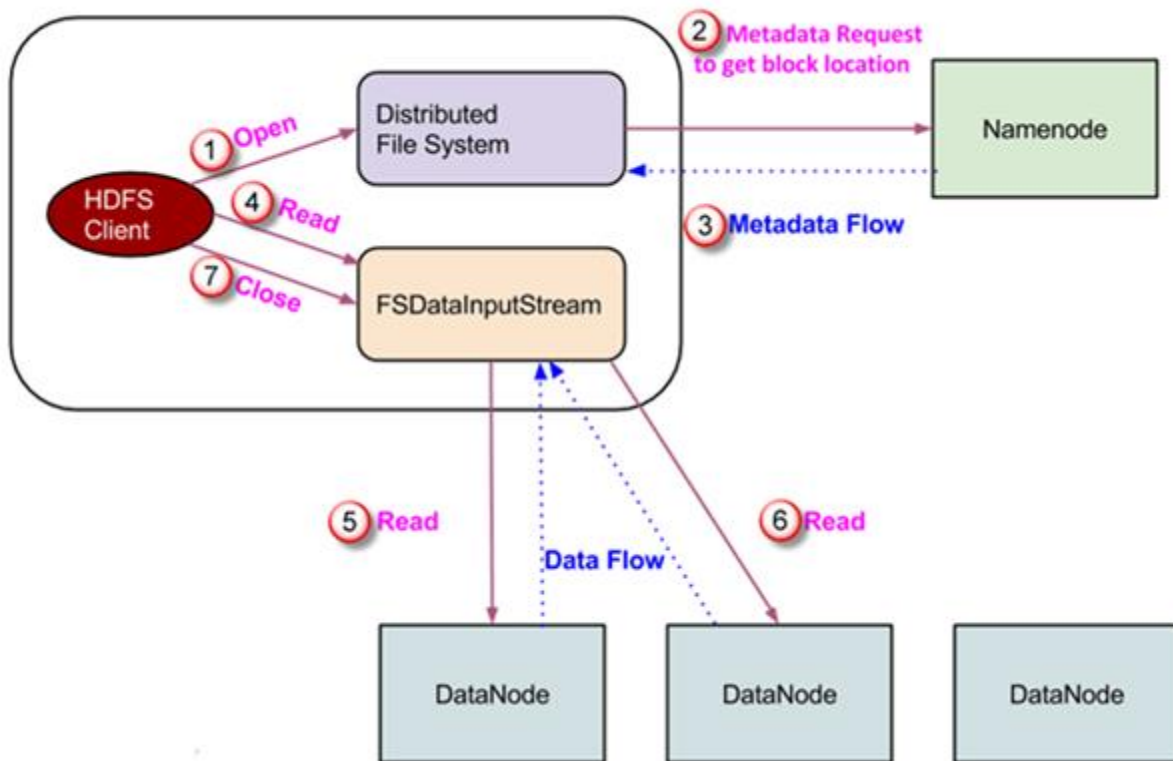
`hdfs dfs -chmod 사용자허가모드 /[디렉토리명 또는 파일명]`

지정된 디렉토리 또는 파일의 사용자 허가 모드를 변경한다.

`hdfs dfs -mv /[디렉토리]/old파일 /[디렉토리]/new파일`

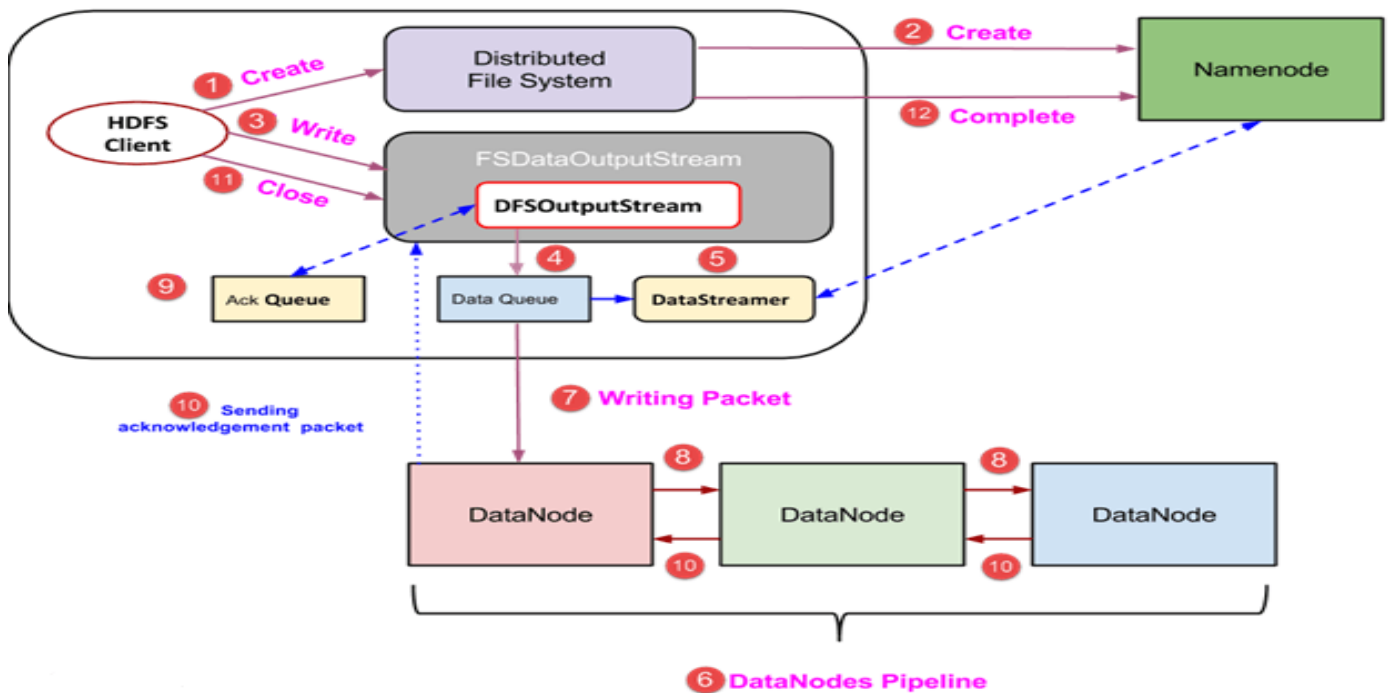
지정된 디렉토리의 파일을 다른 이름으로 변경하거나 다른 폴더로 이동한다.

## [ HDFS Read Process ]



- (1) 클라이언트는 FileSystem(DistributedFileSystem) 객체의 'open()' 메소드를 호출하여 읽기 요청을 시작한다.
- (2) FileSystem(DistributedFileSystem) 객체는 RPC를 사용하여 NameNode에 연결하고 파일 블록의 위치 정보를 담고 있는 메타 데이터 정보를 가져온다.
- (3) 이 메타 데이터에는 읽고자 하는 파일의 블록에 대한 사본이 저장되어 있는 DataNode 의 주소가 담겨 있다.
- (4, 5) DataNode의 주소를 받으면 FSDDataInputStream 유형의 객체가 클라이언트에 리턴된다. FSDDataInputStream 객체는 DFSInputStream 을 포함하고 있으며 읽기 관련 메서드가 호출되면 가장 우선순위가 높은 데이터 노드에서 해당 블록을 읽게 된다.
- (6) 블록의 끝 부분에 도달하면 DFSInputStream은 연결을 닫고 다음 블록에 대한 다음 DataNode를 찾는다.
- (7) 클라이언트가 읽기를 완료 하면 close() 메소드를 호출한다 .

## [ HDFS Write Process ]



- (1) 클라이언트는 새로운 파일을 생성하는 DistributedFileSystem 객체의 'create ()'메소드를 호출하여 쓰기 작업을 시작한다.
- (2) DistributedFileSystem 객체는 RPC 호출을 사용하여 NameNode에 연결하고 새 파일 만들기를 시작한다. NameNode의 책임은 생성되는 파일이 이미 존재하지 않고 클라이언트가 새 파일을 생성 할 수 있는 올바른 권한을 가지고 있는지 확인하는 것이다. 파일이 이미 있거나 클라이언트에 새 파일을 작성할 수 있는 충분한 권한이 없는 경우 IOException 을 클라이언트에서 발생시키지만 문제가 없는 경우 NameNode에 의해 파일에 대한 새 레코드가 만들어진다.
- (3) NameNode에 의해 새 레코드가 만들어지면 FSDataOutputStream 유형의 객체가 클라이언트에 반환된다. 클라이언트는 이 객체를 사용하여 HDFS에 데이터를 쓰기 위하여 write() 메소드를 호출한다..
- (4) FSDataOutputStream은 DataNodes 및 NameNode와의 통신을 모니터링하는 역할의 DFSOutputStream 객체를 포함한다. 클라이언트가 계속해서 데이터를 쓰는 동안 DFSOutputStream 은 이 데이터를 가지고 패킷을 계속 작성하고 DataQueue 라고 하는 대기열에 저장한다. .
- (5) DataStreamer는 NameNode에 새로운 블록 할당을 요청하여 복제에 사용할 바람직한 DataNode를 선택한다.
- (6) DataNode를 사용하여 파이프 라인을 생성해서 복제 프로세스를 시작한다. 여기서는 복제 수준을 3으로 선택했으므로 파이프 라인에 3 개의 DataNode가 존재하게 된다.
- (7) DataStreamer는 패킷을 파이프 라인의 첫 번째 DataNode에 집어 넣는다.
- (8) 파이프 라인의 모든 DataNode는 받은 패킷을 저장하고 파이프 라인의 두 번째 DataNode로 전달한다.
- (9) 또 다른 큐인 'Ack Queue'는 DFSOutputStream에 의해 유지되어 DataNode로부터 확인을 기다리는 패킷을 저장한다.
- (10) 대기열에 있는 패킷에 대한 수신 확인을 파이프 라인의 모든 DataNode로 부터 수신하게 되면 'Ack 대기열'에서 제거된다. DataNode가 실패 할 경우 이 큐의 패킷을 사용하여 작업을 다시 시작한다.
- (11) 클라이언트가 데이터 쓰기를 완료하면 close() 메서드를 호출하여 남아있는 데이터 패킷을 파이프 라인으로 플러시하고 최종 승인을 기다린다.
- (12) 최종 승인을 받으면 NameNode에 연락하여 파일 쓰기 작업이 완료되었음을 알린다.



## 하둡 설치를 위한 linux 시스템 4개 준비

1. VMWare Player 에서 VM을 m1 이라고 만든다.

2. LINUX 를 설치한다.

설치도중에 나오는 네트워크 설정 화면에서 호스트명을 master 지정한다.

시스템 관리자 계정은 root 이다.(암호 : password), 설치 중간에 centos 계정을 생성한다.(암호 : centos)

설정사항 : 메모리 4G, 고정 IP 주소 설정 : 192.168.111.120

ifconfig 로 IP 주소가 변경된 것을 확인한다.

선택적으로 추가 작업을 수행한다.

3. JDK 를 설치한다. (tools 디렉토리에 강사컴에서 가져간 LINUX용 JDK 설치 파일을 복사한다.)

```
cd tools
```

```
tar xvfz jdk-8u131-linux-x64.tar.gz
```

```
mv jdk1.8.0_131 /usr/local
```

```
cd /usr/local
```

```
ln -s jdk1.8.0_131 java
```

```
ls -l
```

```
vi /etc/profile
```

```
source /etc/profile
```

```
java -version
```

제일 아래에 밑에 다음 행들을 추가한다.  
export JAVA\_HOME=/usr/local/java  
export PATH=\$JAVA\_HOME/bin:\$PATH

4. Hadoop을 설치한다.

tools 디렉토리에서 다음 명령을 수행하고 하둡 설치 프로그램을 다운로드한다. (hadoop-2.x.x-bin.tar.gz) :

```
wget https://archive.apache.org/dist/hadoop/common/hadoop-2.7.7/hadoop-2.7.7.tar.gz
```

다운로드 폴더에 있는 hadoop-2.x.x-bin.tar.gz 을 tar xvfz xxxx.gz 명령으로 풀고 root 계정의

홈디렉토리로 hadoop-2.x.x 폴더를 이동한다. ls 명령을 수행시켜서 하둡이 다음 디렉토리에 설치된 것을 확인한다.  
/root/hadoop-2.x.x

5. 방화벽 중단

```
systemctl stop firewalld
```

```
systemctl disable firewalld
```

6. 홈디렉토리의 .bashrc 파일에 다음 내용을 추가한다.

```
export HADOOP_HOME=/root/hadoop-2.7.7
```

```
export PATH=$HADOOP_HOME/bin:$HADOOP_HOME/sbin:$PATH
```

7. Hadoop의 설정파일 디렉토리로 옮겨간다.

```
cd $HADOOP_HOME/etc/hadoop
```

8. hadoop-env.sh 파일 끝에 다음 내용을 추가한다.

```
export JAVA_HOME=/usr/local/java
```

```
export HADOOP_HOME=/root/hadoop-2.7.7
```

```
export HADOOP_HEAPSIZE=500
```

9. mapred-env.sh 파일 끝에 다음 내용을 추가한다.

```
export JAVA_HOME=/usr/local/java
```

```
export HADOOP_JOB_HISTORYSERVER_HEAPSIZE=500
```

```
export HADOOP_HOME=/root/hadoop-2.7.7
```

10. yarn-env.sh 파일 끝에 다음 내용을 추가한다.

```
export JAVA_HOME=/usr/local/java
export YARN_HEAPSIZE=500
export HADOOP_HOME=/root/hadoop-2.7.7
```

11. core-site.xml 에 다음 내용을 편집한다.

```
<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://master:9000/</value>
  </property>
</configuration>
```

12. hdfs-site.xml 에 다음 내용을 편집한다.

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>3</value>
  </property>
  <property>
    <name>dfs.name.dir</name>
    <value>/root/hadoop-2.7.7/hdfs/name</value>
  </property>
  <property>
    <name>dfs.data.dir</name>
    <value>/root/hadoop-2.7.7/hdfs/data</value>
  </property>
  <property>
    <name>dfs.support.append</name>
    <value>true</value>
  </property>
  <property>
    <name>dfs.namenode.secondary.http-address</name>
    <value>slave1:50090</value>
  </property>
  <property>
    <name>dfs.namenode.secondary.https-address</name>
    <value>slave1:50091</value>
  </property>
</configuration>
```

13. mapred-site.xml 에 다음 내용을 편집한다.(이 파일은 [mapred-site.xml.template](#) 을 복사해서 만든다.)

```
<configuration>
  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
```

} MapReduce 관련 설정

</configuration>

14. yarn-site.xml 에 다음 내용을 편집한다.

```
<configuration>
  <!-- Site specific YARN configuration properties -->
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
  <property>
    <name>yarn.resourcemanager.hostname</name>
    <value>master</value>
  </property>
  <property>
    <name>yarn.resourcemanager.webapp.address</name>
    <value>${yarn.resourcemanager.hostname}:8088</value>
  </property>
  <property>
    <name>yarn.nodemanager.resource.memory-mb</name>
    <value>4096</value>
  </property>
  <property>
    <name>yarn.scheduler.minimum-allocation-mb</name>
    <value>2048</value>
  </property>
</configuration>
```

MapReduce 관련 설정

MapReduce 관련 설정

15. slaves 파일을 다음 내용으로 편집한다.

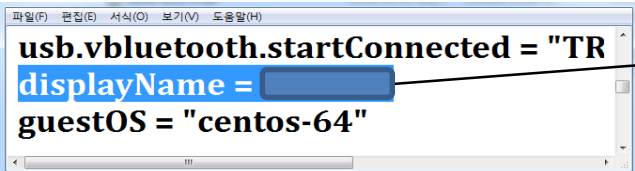
slave1  
slave2  
slave3

----- 머신 1개 완성

16. 생성 완료된 VM을 복사하여 총 4개를 만든다.

m1, m2, m3, m4

m1을 제외하고 각 폴더의 xxxxx.vmx파일을 메모장으로 열어서 수정한다. 다음과 같이 수정한다.



m1을 각각 m2, m3, m4 로 수정

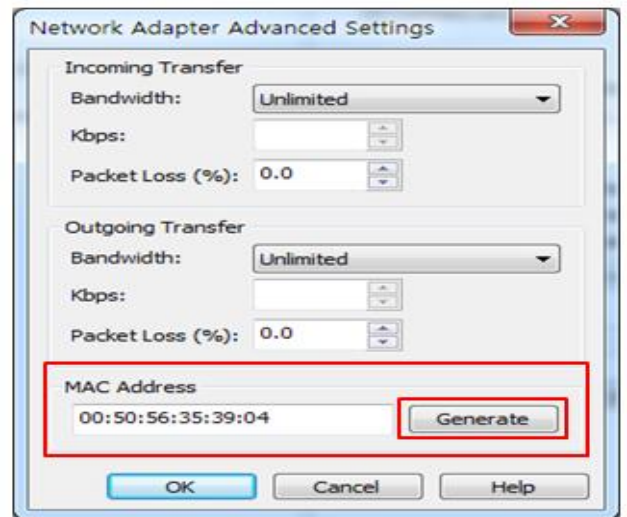
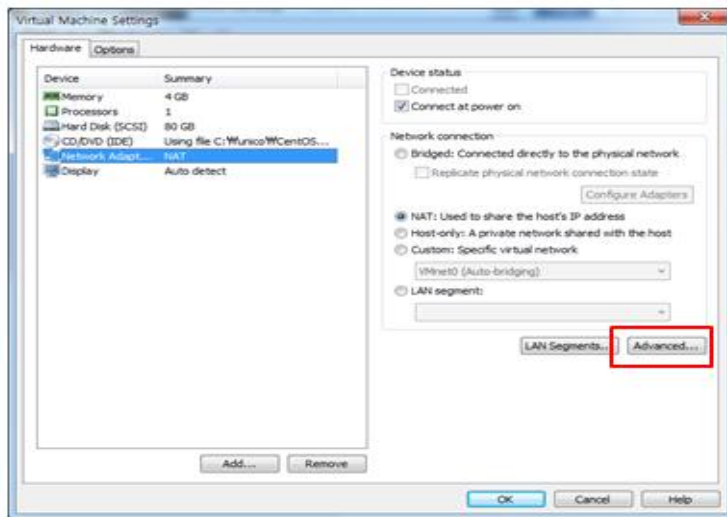
displayName = "m1" --> displayName = "m2"  
displayName = "m1" --> displayName = "m3"  
displayName = "m1" --> displayName = "m4"

각각 수정한다.

17. VMWare Player 에서 각각의 VM을 오픈하고 Edit virtual machine settings를 선택하여 slave1, slave2, slave3 의 MAC 어드레스를 재할당 받기 위해 VM을 선택하고 Edit virtual machine settings를 선택한다.

18. 네트워크 어댑터를 선택하고 삭제 후에 새로이 추가한 후에 Advanced 버튼을 누르고 MAC 어드레스를 새로이 생성한다. → 이거 엄청 중요!!!! 이 때의 MAC 어드레스를 각각 메모한다.





19. 리눅스를 각각 기동시킨다. 옆의 윈도우가 출력되면 반드시 **I moved it** 을 선택한다.

root 로 로그인한 후에 각 리눅스의 호스트명을 수정한다.

```
hostnamectl set-hostname slave1
```

```
hostnamectl set-hostname slave2
```

```
hostnamectl set-hostname slave3
```

수정 후에 hostname 이라는 명령을 수행시켜서 변경된 것을 확인한다.

20. 각 머신의 고정 IP를 설정한다. (리눅스 교재 117~118 참조)

IP 주소는 각각

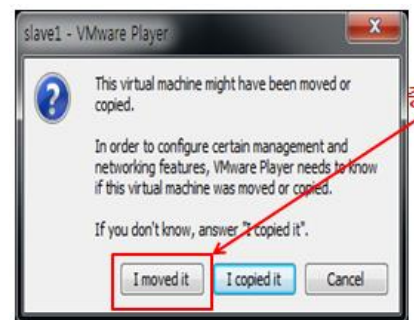
slave1 : 192.168.111.130

slave2 : 192.168.111.140

slave3 : 192.168.111.150 으로 하여 각각 진행한다.

**이 때 첫 번째 행의 HWADDR="MAC 주소" 를 각각 메모한 MAC 주소로 수정한다.**

ifconfig 명령으로 수정된 사항을 체크한다.



21. /etc/hosts 파일을 오픈하여 다음 내용을 추가한다.

```
192.168.111.120 master
```

```
192.168.111.130 slave1
```

```
192.168.111.140 slave2
```

```
192.168.111.150 slave3
```

서로 통신이 잘 되는지 ping 명령어를 서로에게 수행시켜 본다. : ping slave1, ping slave2, ping slave3,

22. SSH Key 를 생성하여 다른 시스템과 공유한다.

비밀번호 없이 로그인이 가능하도록 하는 것으로 다음 명령을 master 시스템에서 수행시켜 공개키를 만든다.

(참고 : 시큐어 셸(Secure Shell, SSH)은 네트워크 상의 다른 컴퓨터에 로그인하거나 원격 시스템에서 명령을 실행하고 다른 시스템으로 파일을 복사할 수 있도록 해 주는 응용 프로그램 또는 그 프로토콜을 가리킨다.)

```
[hadoop@master ~]$ ssh-keygen -t rsa
```

Generating public/private rsa key pair.

Enter file in which to save the key (/root/.ssh/id\_rsa): **그냥엔터**

Created directory '/root/.ssh'.

Enter passphrase (empty for no passphrase) : **그냥엔터**

Enter same passphrase again: 그냥엔터

공개키를 생성한 다음에는 다음 명령을 수행시켜서 공개키를 나머지 리눅스 시스템과 공유한다.

```
[root@master ~]$ ssh-copy-id -i /root/.ssh/id_rsa.pub root@slave1
[root@master ~]$ ssh-copy-id -i /root/.ssh/id_rsa.pub root@slave2
[root@master ~]$ ssh-copy-id -i /root/.ssh/id_rsa.pub root@slave3
[root@master ~]$ ssh-copy-id -i /root/.ssh/id_rsa.pub root@master
```

다음 단계들은 master 에서만 수행한다.

23. 하둡 파일시스템 포맷한다.

```
hdfs namenode -format
```

24. Hadoop의 HDFS 데몬들을 기동시킨다.

```
start-dfs.sh
```

25. 데몬이 제대로 수행되었지 확인한다. - jps 사용

```
master : NameNode
slave1, slave2, slave3 : DataNode
slave1 : SecondaryNameNode
```

26. 하둡 HDFS 영역에 폴더를 구축한다.

강사시스템에서 파일 세개를 가져가서 리눅스의 sampledata 디렉토리에 넣고 압축파일은 압축을 푼다.

( 압축해제 명령 : bzip2 -kd 2008.csv.bz2 )

```
hdfs dfs -ls /
hdfs dfs -mkdir /edudata
hdfs dfs -chmod 777 /edudata
hdfs dfs -put /root/sampledata/파일명 /edudata
hdfs dfs -ls /edudata
hdfs dfs -chmod 777 /
```

27. 웹 페이지에서 저장된 블록 체크

```
http://master:50070/
```

28. HDFS 데몬 종료

```
stop-dfs.sh
```

29. yarn 기동(Yet Another Resource Negotiator : 분산 컴퓨팅 환경)

```
start-yarn.sh
```

30. yarn 종료

```
stop-yarn.sh
```

31. 웹에서 Resource Manager 체크

```
http://master:8088/
```

## [ HADOOP HDFS 주요 API 정리 ]

<http://hadoop.apache.org/docs/r2.7.7/api/index.html?org/apache/hadoop/fs/package-summary.html>

### org.apache.hadoop.conf.Configuration

void addResource(String name)                      설정 리소스를 추가한다.  
void set(String name, String value)              key와 value 값을 설정한다.

### org.apache.hadoop.fs.Path

FileSystem getFileSystem(Configuration conf)  
NameNode에 의해 관리되는 FileSystem 객체를 리턴한다.

### org.apache.hadoop.fs.FileSystem

FSDatInputStream open(Path f)  
지정된 Path 객체에 대한 FSDatInputStream 객체를 리턴한다.  
FSDatOutputStream create(Path f)  
지정된 Path 객체에 대한 FSDatOutputStream 객체를 리턴한다.  
FSDatOutputStream create(Path f, boolean overwrite)  
지정된 Path 객체에 대한 FSDatOutputStream 객체를 리턴한다.  
FSDatOutputStream append(Path f)  
지정된 Path 객체에 대한 FSDatOutputStream 객체를 리턴한다.  
boolean delete(Path f, boolean recursive)  
경로에 해당되는 파일을 삭제한다.  
boolean exists(Path f)  
경로가 존재하는지 여부를 반환한다.

### static FileSystem get(Configuration conf)

주어진 Configuration에 대한 FileSystem 객체를 리턴한다.

FileStatus getFileStatus(Path f)  
FileStatus 객체를 반환한다.

boolean isDirectory(Path f)  
경로가 디렉토리인지의 여부를 리턴한다.

boolean isFile(Path f)  
경로가 파일인지의 여부를 리턴한다.

static FileSystem newInstance(Configuration conf)  
주어진 Configuration에 대한 FileSystem 객체를 리턴한다.

FSDatInputStream open(Path f)  
주어진 경로에 대한 FSDatInputStream 객체를 오픈한다.

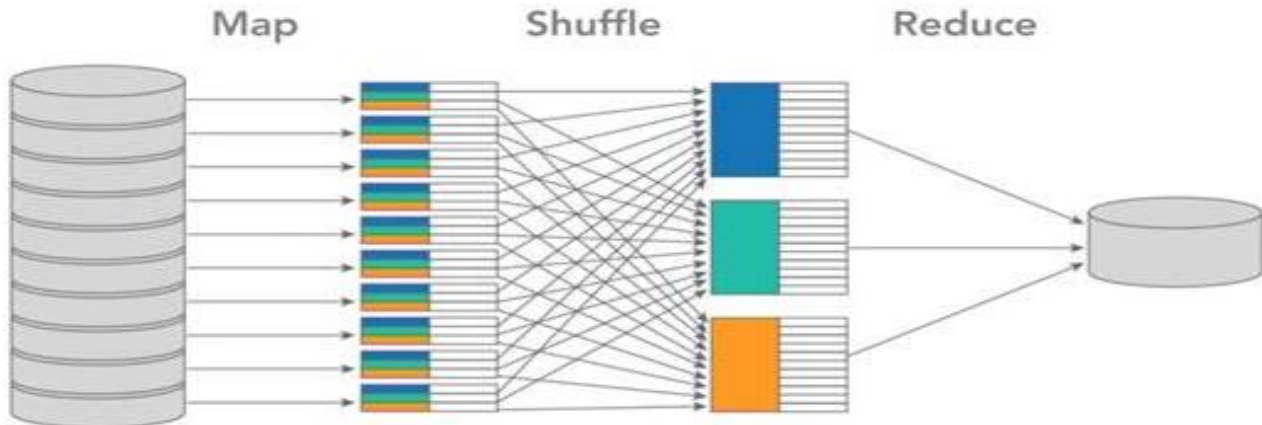
### org.apache.hadoop.fs.FileStatus

int getBlockSize()  
파일의 블록사이즈를 반환한다.

long getLength()  
파일의 길이(바이트 단위)를 리턴한다.

## [ Hadoop MapReduce ]

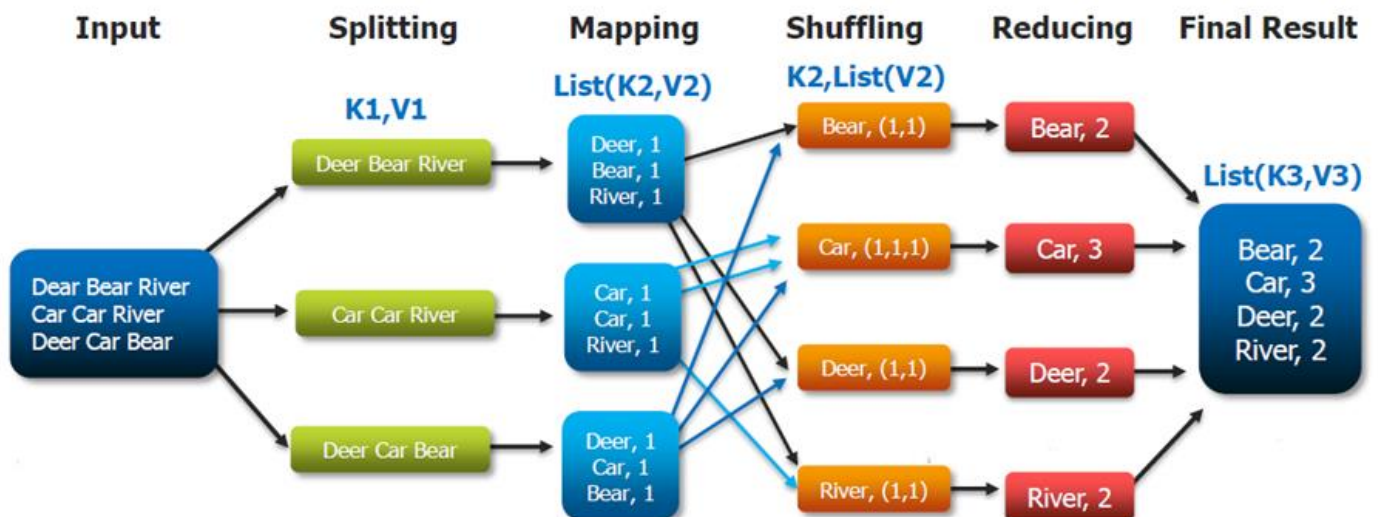
MapReduce는 구글에서 대용량 데이터 처리를 분산 병렬 컴퓨팅에서 처리하기 위한 목적으로 제작하여 2004년 발표한 소프트웨어 프레임워크다. 이 프레임워크는 페타바이트 이상의 대용량 데이터를 신뢰도가 낮은 컴퓨터로 구성된 클러스터 환경에서 병렬 처리를 지원하기 위해서 개발되었다.



MapReduce는 Hadoop 클러스터의 데이터를 처리하기 위한 시스템으로 총 2개(Map, Reduce)의 단계로 구성된다. Map과 Reduce 사이에는 shuffle과 sort라는 단계가 존재한다. 각 Map task는 전체 데이터 셋에 대해서 별개의 부분에 대한 작업을 수행하게 되는데, 기본적으로 하나의 HDFS block을 대상으로 수행하게 된다. 모든 Map 태스크가 종료되면, MapReduce 시스템은 중간(intermediate) 데이터를 Reduce 단계를 수행할 노드로 분산하여 전송한다.

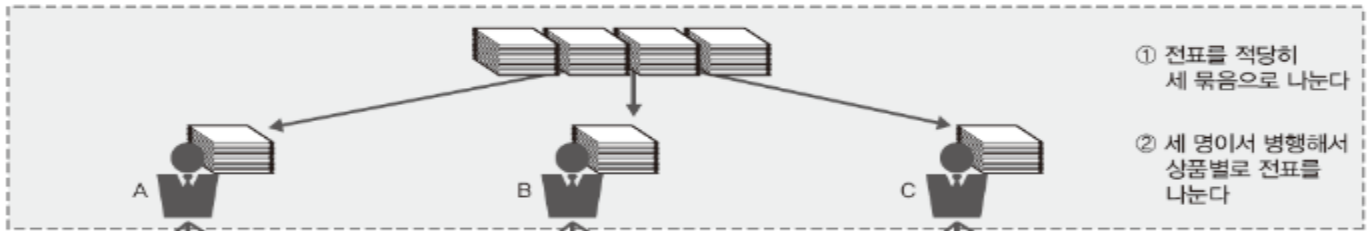
Distributed File System에서 수행되는 MapReduce 작업이 끝나면 HDFS에 파일이 써지고, MapReduce 작업이 시작할 때는 HDFS로 부터 파일을 가져오는 작업이 수행된다.

### The Overall MapReduce Word Count Process

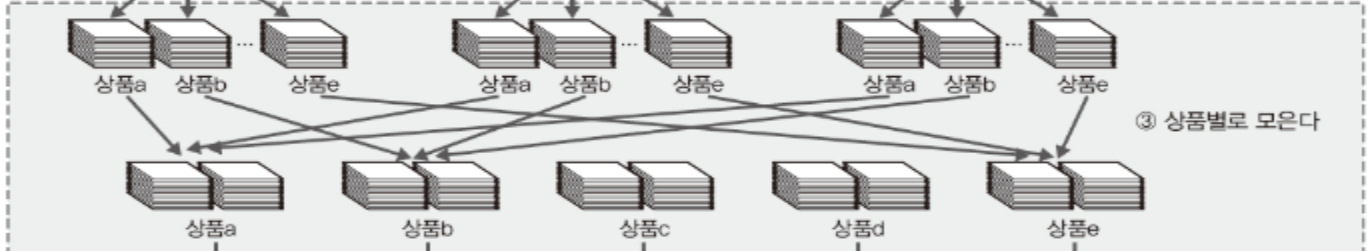


## [ MapReduce의 처리 개념 : 현실에서도 일어나는 일 ]

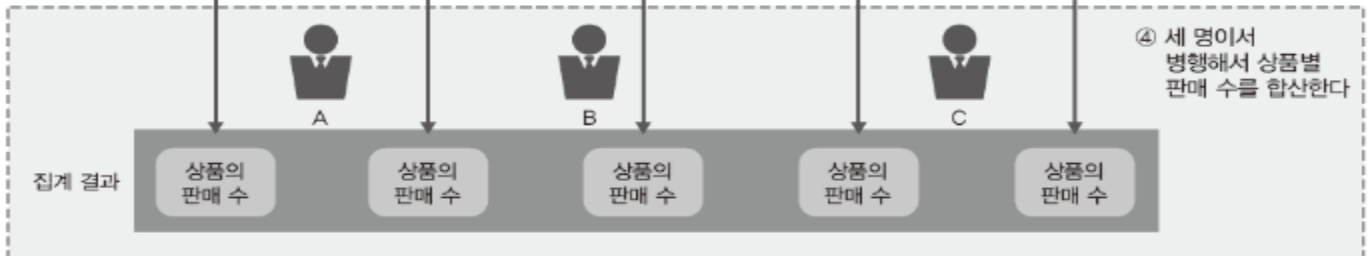
### 1단계



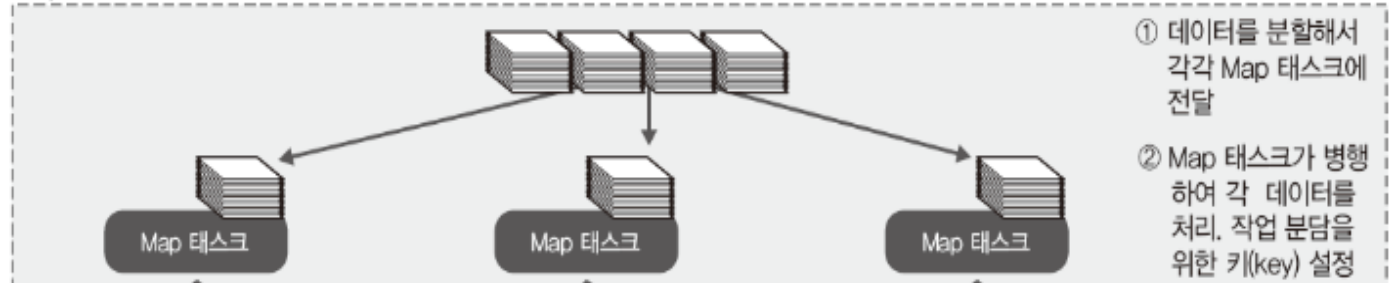
### 2단계



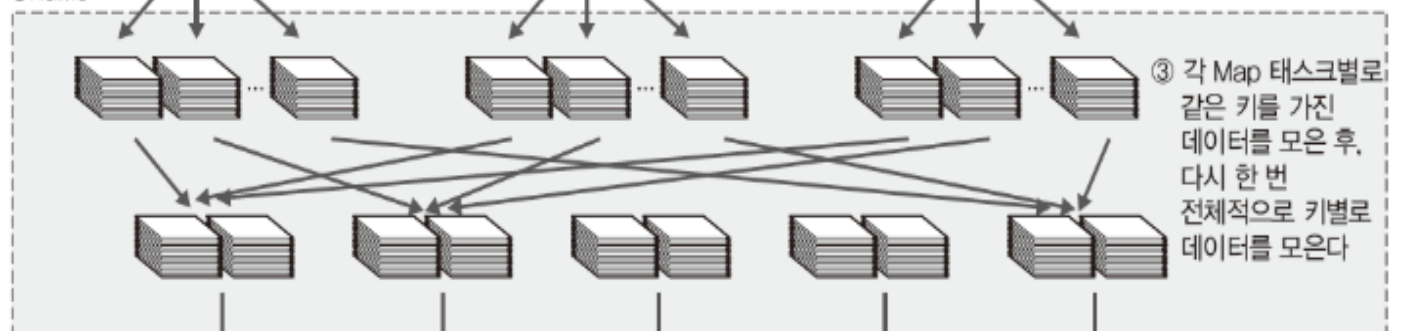
### 3단계



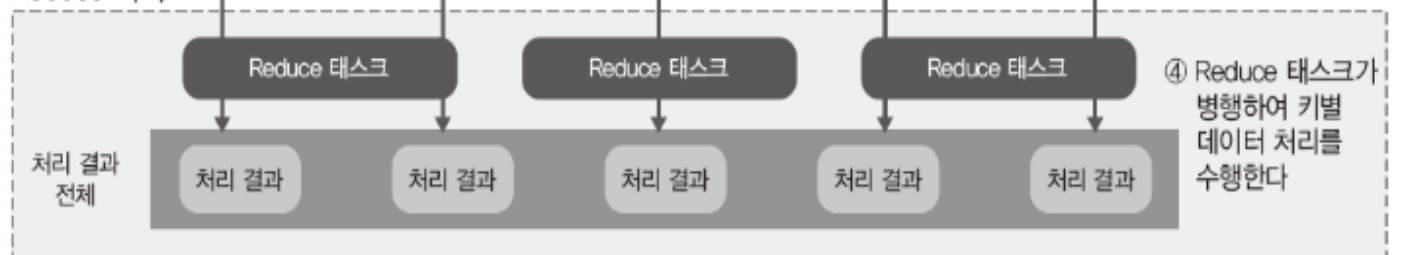
### Map 처리



### Shuffle



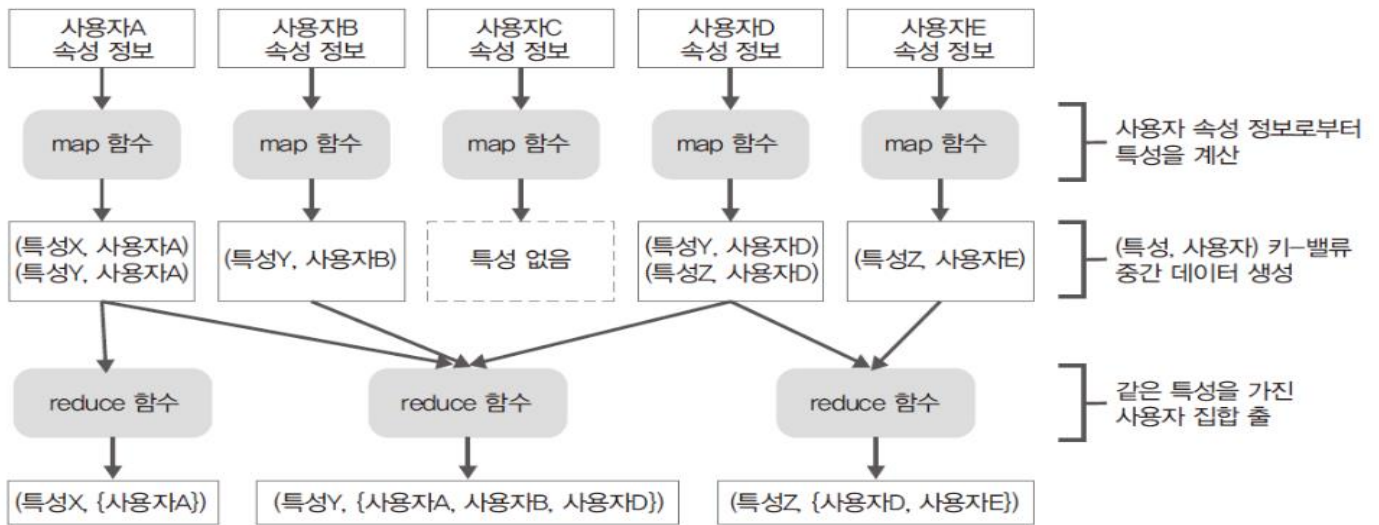
### Reduce 처리



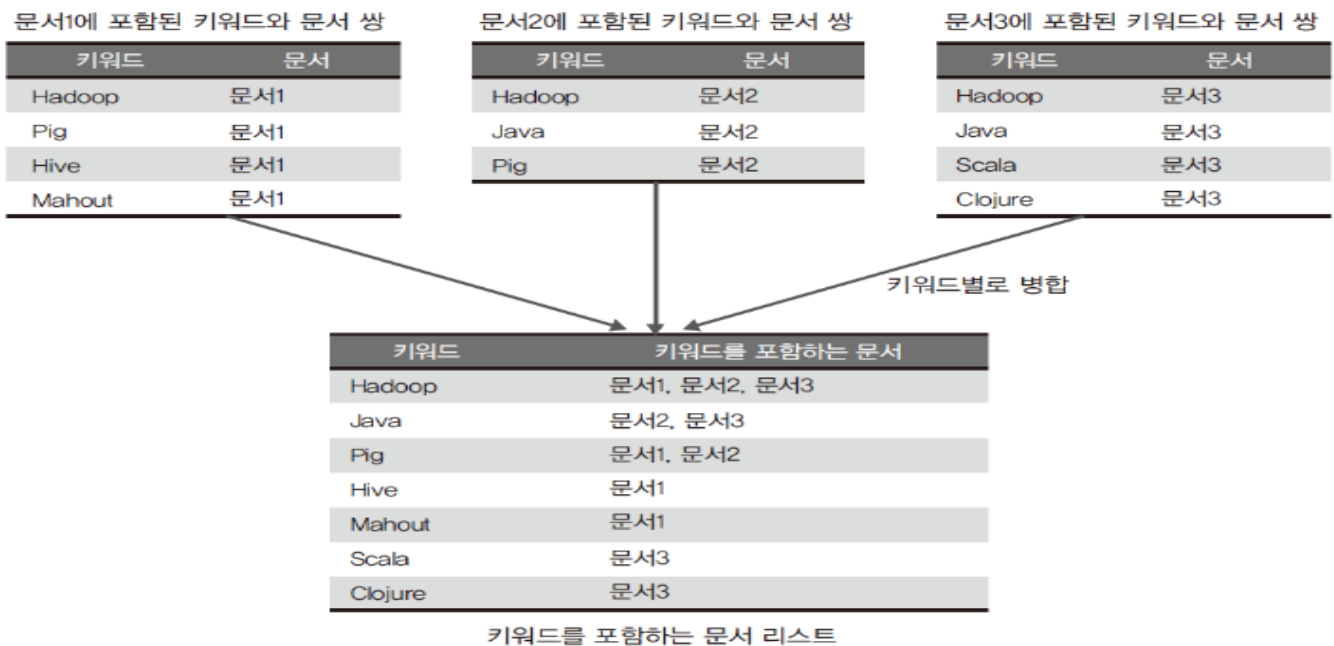


## [ MapReduce 응용 ]

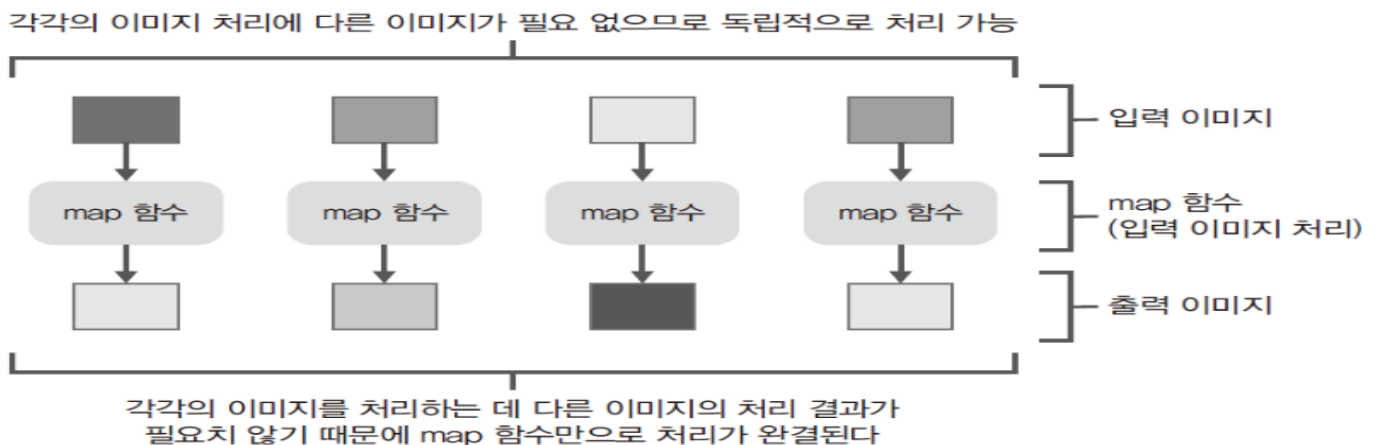
### 1. 비슷한 특성의 사람 찾기



### 2. 키워드를 추출한 문서 인덱스 작성



### 3. 이미지 데이터 분산처리(개인 정보 모자이크 처리)





## [ MapReduce 정리 ]

MapReduce는 데이터를 개별로 가공 및 필터링하거나, 어떤 키값에 기반해 데이터를 분류하거나, 분류한 데이터로 통계치를 계산하는 등, 수많은 데이터 처리에서 사용되고 있는 기법들을 일반화 하고 있다. map() 함수와 reduce() 함수는 한 번에 처리할 수 있는 데이터와 데이터 전달 방법 등이 다르다.

**map()** 함수는 처리 대상 데이터 전체를 하나씩, 하나씩 처리한다. 처리대상 데이터간에 의존관계가 없고 독립적으로 실행 가능하며 처리나 순서를 고려하지 않아도 되는 처리에 적합하다. **(전처리)**

**reduce()** 함수에는 키와 연관된 복수의 데이터가 전달된다. 또한 reduce() 함수에 전달되는 데이터는 키값으로 정렬되어 있다. 그룹화된 복수의 데이터를 필요로 하는 처리 또는 순서를 고려해야 하는 처리에 적합하다. **(그룹별 합계)**

**map : (k1, v1) -> list(k2, v2)**

**reduce : (k2, list(v2)) -> list(k3, v3)**

## [ MapReduce 소스 ]

```
public class WordCountMapper 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 {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            context.write(word, one);
        }
    }
}

public class WordCountReducer 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 class WordCount {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        conf.set("fs.defaultFS", "hdfs://192.168.111.120:9000");

        Job job = Job.getInstance(conf);

        job.setJarByClass(WordCount.class);
        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

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

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

        FileInputFormat.addInputPath(job, new Path("/edudata/fruit.txt"));
        FileOutputFormat.setOutputPath(job, new Path("/output/result"));

        job.waitForCompletion(true);
        System.out.println("SUCCESS");
    }
}

```

## 맵

input : 원본데이터

output : 키/값

## 리듀스

input : 맵에서 추출한 키/[값, .....]

output : 그룹화 및 연산수행하여 최종 결과 : 키/값

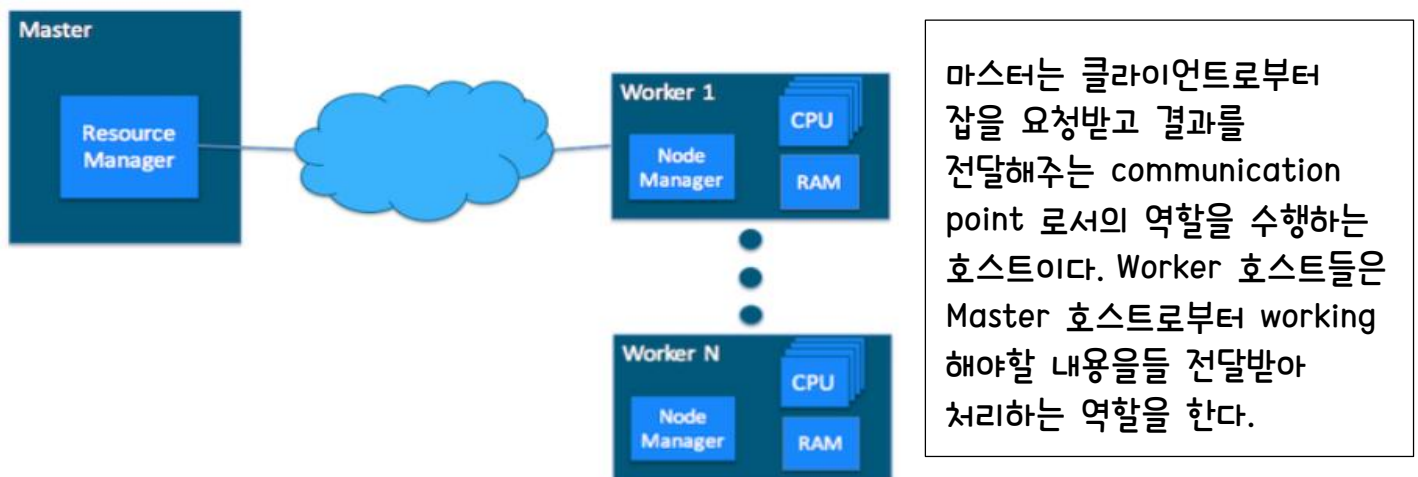
## [ YARN : Yet Another Resource Negotiator ]

Hadoop은 크게 두 가지 컴포넌트로 구성되는데 하나는 파일을 저장할 수 있는 **분산 파일 시스템**인 HDFS(Hadoop Distributed File System)와 **분산 컴퓨팅 환경**을 제공하는 YARN(Yet Another Resource Negotiator) 이다.

HDFS 클러스터 + YARN 클러스터



## [ YARN 클러스터의 구성 ]



**Resource Manager** : Master 호스트에서 뜨는 데몬으로서 client 와 통신하는 역할을 한다. 그리고 클러스터에 있는 Worker 호스트들의 Resource 를 트래킹하여 Job 실행에 필요한 Node Manager 들을 컨트롤 하는 역할을 한다.

**Node Manager** : Worker 호스트에 존재하는 데몬 으로서 실제로 job 을 처리하고 결과를 전달해주는 역할을 한다. 사용자가 요청한 프로그램을 실행하는 Container 를 fork 시키고 Container 를 모니터링 Container 장애 상황 또는 Container 가 요청한 리소스보다 많이 사용하고 있는지 감시한다.

**Container** : YARN 에서 쓰는 용어로서 리소스를 점유하는 하나의 단위이다. job 의 요청이 여러 개의 Map/Reduce Task 로 분리되어 Node Manager 에게 할당되면 Node Manager 는 그 요청을

처리하기 위해 하나의 Container 를 실행하게 된다. 이 Container 는 우측에서 보는거와 같이 CPU, RAM 등의 리소스를 점유하여 task 를 처리하게 된다.

### [ MapReduce 작업이 요청되어 처리되는 프로세스 ]

YARN 클러스터에 job 을 요청한 경우(= 클러스터에서 Process 를 Running 하는 ) 어떠한 방식으로 실행이 되는지 소개한다.

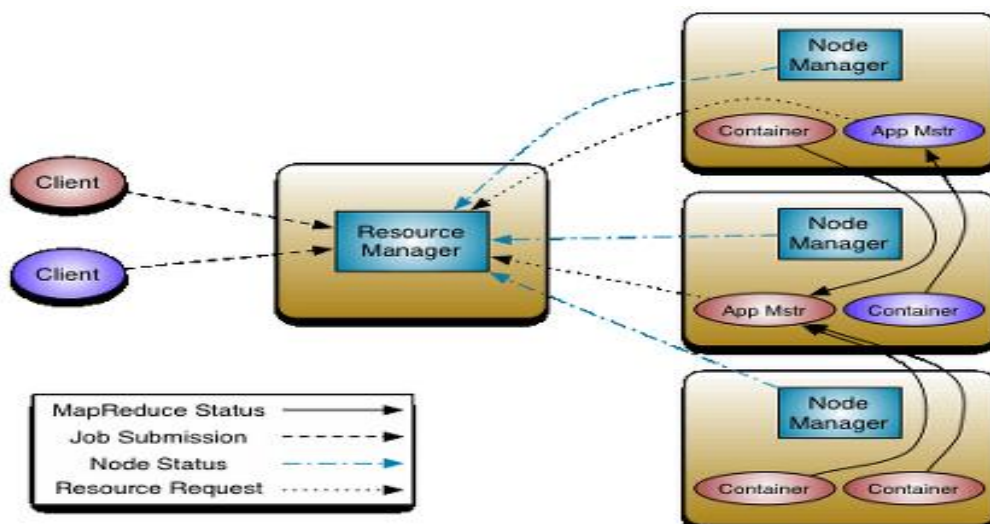
1. Client 가 Application 을 실행하고 클러스터의 Resource Manager 에게 이를 알려 준다.



2. Resource Manager 은 Worker 노드중에 하나를 골라 Container 를 생성한다. 그리고 그 Container 가 바로 Application Master 가 된다.



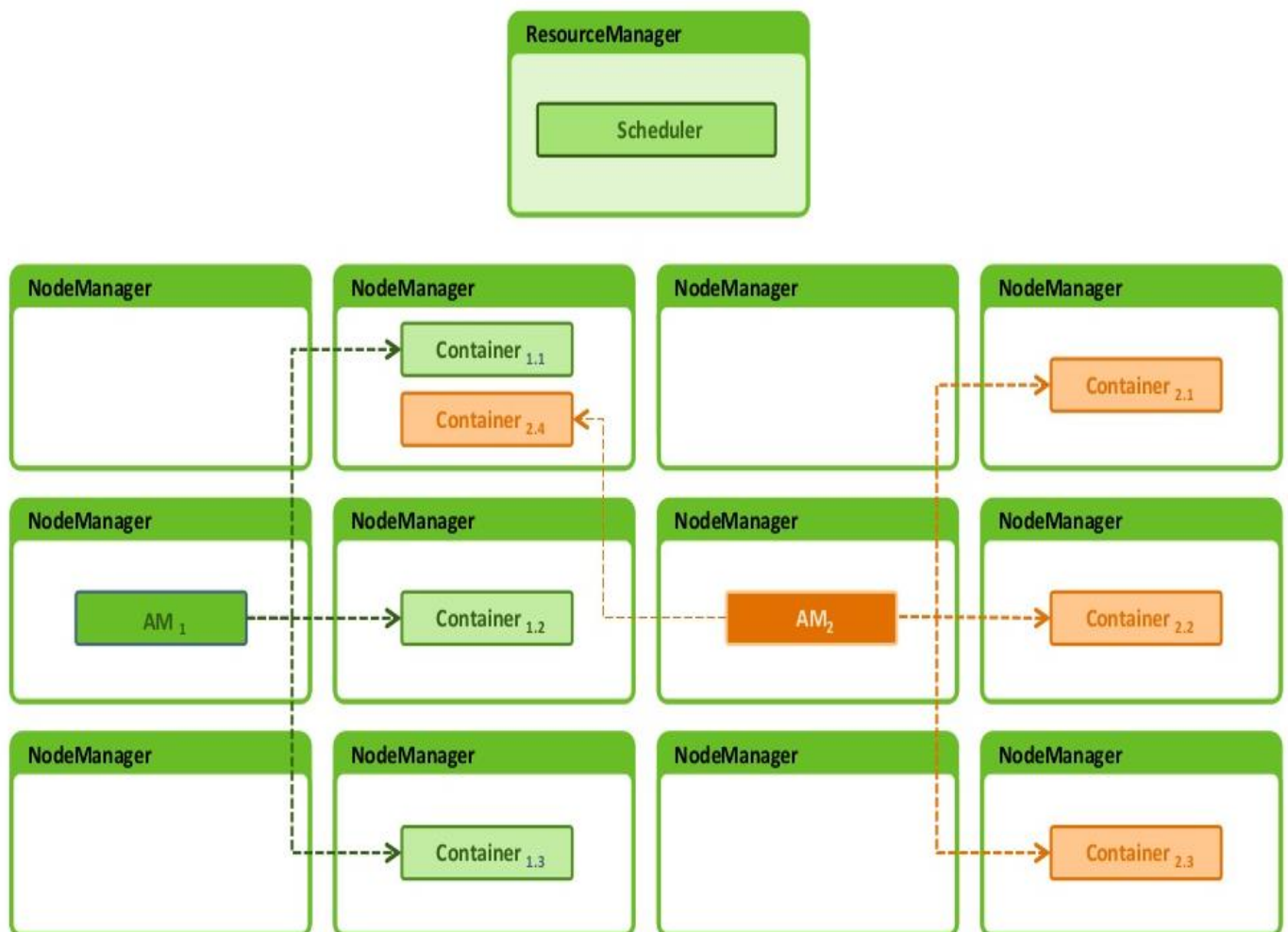
3. Application Master 는 task 를 실행할 컨테이너를 RM 에게 요청하게 된다. 그럼 Application Master 는 유효한 자원을 소유한 노드의 Node Manager 를 통해 task 를 실행할 Container 를 생성하게 되고 task 를 실행하는 Container 는 task 의 상태( status) 를 Application Master 에게 알려준다.



4. 모든 task 가 종료되면 AM 도 종료되고 클러스터에 할당된 컨테이너의 자원도 모두 de-allocated 된다. 그리고 Application client 도 종료된다.

YARN 은 Application 의 job 을 분산 환경에서 병렬 처리할 수 있도록 도와주는 클러스터링 서비스이다. Resource Manager, Application Master, Node Manager 가 YARN 에서의 주요 컴포넌트이고 하나의 잡을 처리하기 위해 여러 개의 Task 를 나누고, 이를 처리하기 위한 Container 라는 개념이 존재한다.

[ Job 을 요청 받은 YARN 클러스터의 전체적인 구성 ]



[ YARN 에서 MapReduce 동작 ]

YARN 의 실제 동작 방식을 이해하기 위해서는 MapReduce 가 YARN 클러스터에서 어떻게 동작하는지를 이해하는 것이 가장 좋은 방법이다. MapReduce 가 최초로 YARN 을 사용하는 시스템이었으며, 같은 Hadoop 프로젝트내에 엮여있기 때문에 YARN 을 가장 잘 사용하는 시스템이라고 할 수 있다. 다음은 YARN 클러스터에서 MapReduce 프로그램이 어떻게 동작하는지에 대한 설명이다.

(0) 다음 명령을 수행시켜서 YARN 클러스터를 기동한다.

`start-yarn.sh`

(1) Mapper, Reducer, Driver(메인) 클래스 작성한다.

(2) Driver(메인) 클래스 실행한다.

(3) YARN 환경에서는 Job 별로 클러스터를 구성한다.

(4) 요청된 MapReduce 를 위한 클러스터를 구성하기 위해서는 1 개의 Application Master 가 필요하며 이를 위해 YARN 의 ResourceManager 에게 리소스를 요청한다.

(5) YARN 의 ResourceManager 는 요청받은 ApplicationMaster 를 자신이 관리하는 클러스터(여러개의 NodeManager) 중 하나의 서버를 선택하여 ApplicationMaster 를 실행하고 이 서버를 클라이언트에게 알려준다. ApplicationMaster 는 요청받은 Job 의 종류에 따라서 다르며 MapReduce 에 사용되는 ApplicationMaster 는 MRAppMaster (`org.apache.hadoop.mapreduce.v2.app`) 이다.

(6) 클라이언트는 ResourceManager 로 부터 받은 ApplicationMaster 서버에 MapReduce 작업 요청을 한다.

(7) MRAppMaster 는 작업 요청을 받으면 사용자가 실행한 MapReduce 작업에 필요한 리소스를 다시 ResourceManager 에게 요청한다.

(8) ResourceManager 는 요청받은 리소스에 대해 NodeManager 를 지정하고 Container 를 실행한 후 Container 목록을 MRAppMaster 에 준다.

(9) NodeManager 에 의해 실행된 MapReduce Task 를 위한 Container 는 MRAppMaster 와 통신을 하며 주어진 Map Task 와 Reduce Task 를 실행한다.

이 과정에서 MRAppMaster 객체의 재활용, Task 를 위해 실행된 Container 객체의 재활용 등은 MapReduce 와 YARN 의 버전업에 따라 처리 방식이 조금씩 다르다.