

# Chapter12. 하둡2 소개

---

[12.1 하둡 2의 등장 배경](#)

[12.2 하둡2 특징](#)

[12.3.1 안의 등장 배경](#)

[12.3.2 안의 특징](#)

[12.3.3 안 아키텍처](#)

[12.4 네임노드 HA](#)

[12.6 HDFS 스냅샷](#)

[12.6.1 스냅샷 Directory](#)

[12.6.2 스냅샷 관리](#)

[12.7 쇼트 서킷 조회](#)

[12.8 헤테로지니어스 스토리지](#)

[12.8.1 스토리지 시장의 변화](#)

[12.8.2 헤테로지니어스 스토리지란?](#)

[12.8.3 스토리지 정책](#)

[12.8.4 무버\(Mover\)](#)

---

## 12.1 하둡 2의 등장 배경

### 1. 리소스 관리

- 하나의 클러스터에서 다양한 하둡 에코시스템이 적절히 시스템 자원을 할당받고, 할당된 자원이 모니터링되고 해제되어야 하는데, 기존 하둡에는 이러한 체계가 잡혀있지 않음

### 2. 안정성 문제

- 하둡의 SPOF(single point of failure, 단일 고장점)인 네임노드의 이중화 문제
- 데이터노드 블록들이 하나의 네임스페이스만 사용하는 데 따른 단점과 성능 개선의 필요

## 12.2 하둡2 특징

### • 포함된 주요 기능

1. YARN
2. 네임노드 고가용성
3. HDFS 페더레이션
4. HDFS 스냅샷
5. NFSv3 파일 시스템 지원
6. 성능 개선

### • 하둡2의 가장 큰 변화는 YARN의 등장

- 하둡1의 맵리듀스 프레임워크는 반드시 맵리듀스 API로 개발된 애플리케이션만 실행할 수 있었음
- 하둡2의 YARN은 맵리듀스 외에 다른 종류의 애플리케이션도 실행할 수 있는 구조. 안이라는 시스템 안에서 다양한 데이터 처리 애플리케이션을 실행할 수 있음

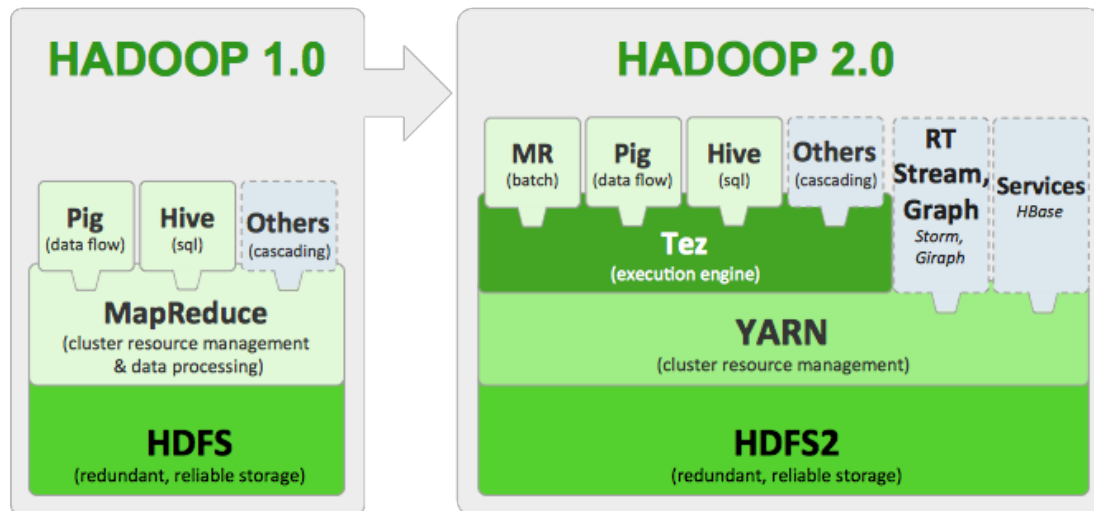
- HDFS는 네임노드의 고가용성을 지원. 데이터노드의 블록을 Pool 방식으로 사용할 수 있는 HDFS 페더레이션 외에 다양한 기능이 추가됨

### 12.3.1 양의 등장 배경

- 하둡1은 데이터 저장소인 HDFS와 데이터 저장소에 저장된 데이터를 배치 처리할 수 있는 맵리듀스라는 두 개의 시스템으로 구성됨. 맵리듀스는 마스터 역할을 하는 잡트래커와 슬레이브 서버 역할을 하는 태스크트래커로 구성됨
  - 맵리듀스의 단일 고장점(Single Point of Failure, SPOF)
    - 잡트래커는 모든 맵리듀스의 잡의 실행 요청을 받고, 전체 잡의 스케줄링 관리와 리소스 관리를 담당.
    - 클라이언트가 맵리듀스 잡을 실행하려면 반드시 잡트래커가 실행중이어야 하며, 태스크트래커가 실행중이더라도 잡트래커가 돌아가고 있지 않다면 맵리듀스 잡 실행이 불가능
  - 잡트래커의 메모리 이슈
    - 잡트래커는 메모리 상에 전체 잡의 실행 정보를 유지하고, 이를 맵리듀스 잡 관리에 활용. 이렇게 메모리에 많은 정보를 유지하기 때문에, 잡트래커도 자연스럽게 많은 메모리가 필요. 실제로 하둡 클러스터를 운영할 때 잡트래커에 힙 메모리를 여유 있게 할당해주는 이유도 바로 이러한 특징 때문.
    - 하지만 잡트래커의 메모리가 부족하다면 잡의 상태를 모니터링할 수도 없고, 새로운 잡의 실행을 요청할 수도 없음. 이러한 특징 때문에 잡트래커는 맵리듀스의 SPOF가 됐고, 이에 대해 해결방안이 활발하게 논의됨
  - 맵리듀스 리소스 관리 방식
    - 맵리듀스는 슬롯이라는 개념으로 클러스터에서 실행할 수 있는 태스크 개수를 관리. 이때 슬롯은 맵 태스크를 실행하기 위한 맵 슬롯과 리듀스 태스크를 실행하기 위한 리듀스 슬롯으로 구분됨.
    - 설정한 맵 슬롯과 리듀스 슬롯을 모두 사용하고 있을 때는 문제가 없으나 실행 중인 잡이 맵 슬롯만 사용하고 있거나, 혹은 리듀스 슬롯만 사용하고 있다면 다른 슬롯은 잉여 자원이 되기 때문에 전체 클러스터 입장에서 리소스가 낭비.
    - 또한, 맵리듀스의 자원을 다른 에코시스템과 공유하는 데도 문제가 있음. 하이브나 피그 같은 맵리듀스 기반 시스템은 상관이 없지만, 타조, 임팔라, 지프라처럼 맵리듀스 기반이 아닌 시스템은 자원을 공유할 수가 없음
  - 클러스터 확장성
    - 최대 단일 클러스터는 4,000여대, 최대 동시 실행 태스크는 40,000개가 한계
    - 맵리듀스는 맵과 리듀스라는 정해진 구조 외에 다른 알고리즘을 지원하는 데 한계가 있음. 예를 들어, K-means / PageRank 알고리즘의 경우 Pregel(구글에서 개발한 분산 그래프 분석 오픈소스 프레임워크)에서는 맵리듀스 대비 10배 이상의 성능을 보여줌
  - 버전 통일성
    - 맵리듀스 잡 실행을 요청하는 클라이언트와 맵리듀스 클러스터 버전이 반드시 동일해야 한다는 부분도 문제점 중 하나

### 12.3.2 양의 특징

- 안이 등장한 이후로 달라진 하둡1과 하둡2 아키텍처 비교



- 안의 설계 목표

#### 1. 잡트래커의 주요 기능 추상화

- 원래 잡트래커는 클러스터 자원 관리와 애플리케이션 라이프 사이클 관리라는 두 가지 핵심 기능을 수행. 안은 이 두 가지 기능을 분리하고 새로운 추상화 레이어를 만들

#### 2. 다양한 데이터 처리 애플리케이션의 수용

- 기존 맵리듀스는 반드시 맵리듀스 API로 구현된 프로그램만 실행할 수 있음. 하지만 안은 맵리듀스도 안에서 실행되는 애플리케이션의 하나로 인식

- 안의 특징

#### 1. 확장성

- 안은 수용 가능한 단일 클러스터 규모를 10,000 노드까지 확대했으며, 클러스터 규모 확대에 의해 처리 가능한 데이터 처리 작업의 개수도 증가.

#### 2. 클러스터 활용 개선

- 안은 자원 관리를 위해 Resourcemanager라는 새로운 컴포넌트를 개발.
- 기존 맵리듀스는 맵 슬롯과 리듀스 슬롯이라는 개념으로 자원을 관리했지만, 리소스 매니저는 CPU, 메모리, 디스크, 네트워크 등 실제 가용한 단위로 자원을 관리하고, 안에서 실행되는 애플리케이션에 자원을 배분. 따라서 하둡 클러스터 운영자는 개별 에코시스템들의 자원 배분을 고민하지 않아도 됨.

#### 3. 워크로드 확장

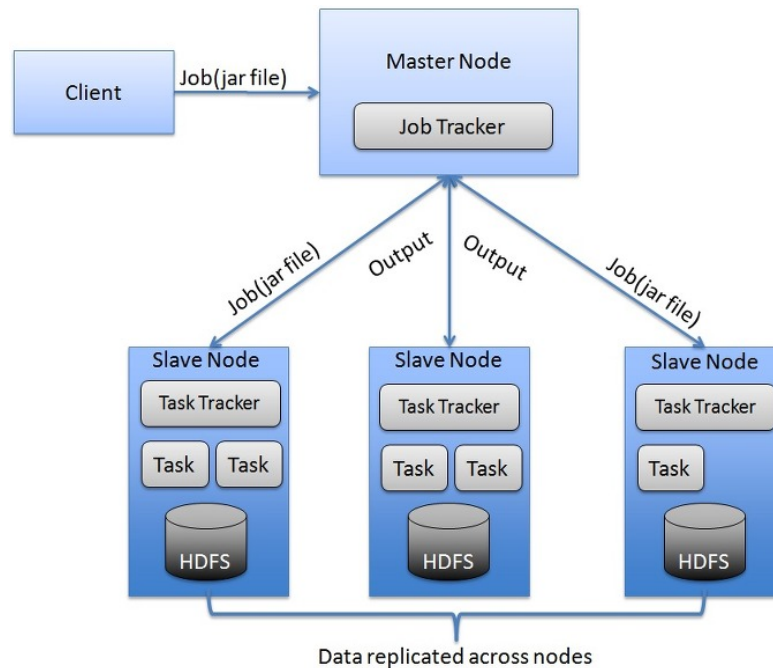
- 하둡1은 맵리듀스, 하이브, 피크 등 맵리듀스 기반의 애플리케이션에서만 실행할 수 있었음. 하지만 안은 맵리듀스 외에도 인터랙티브 질의, 실시간 처리, 그래프 알고리즘 등 다양한 형태의 워크로드 확장이 가능. 심지어 슬라이더 같은 시스템을 이용하면 H베이스도 안을 기반으로 이용할 수 있음.
- 슬라이더(Slider) : 다양한 애플리케이션을 안에서 구동시켜주는 프레임워크이며, 현재 H베이스 Accumulo, Twill 같은 애플리케이션의 안 기반 실행을 지원

#### 4. 맵리듀스 호환성

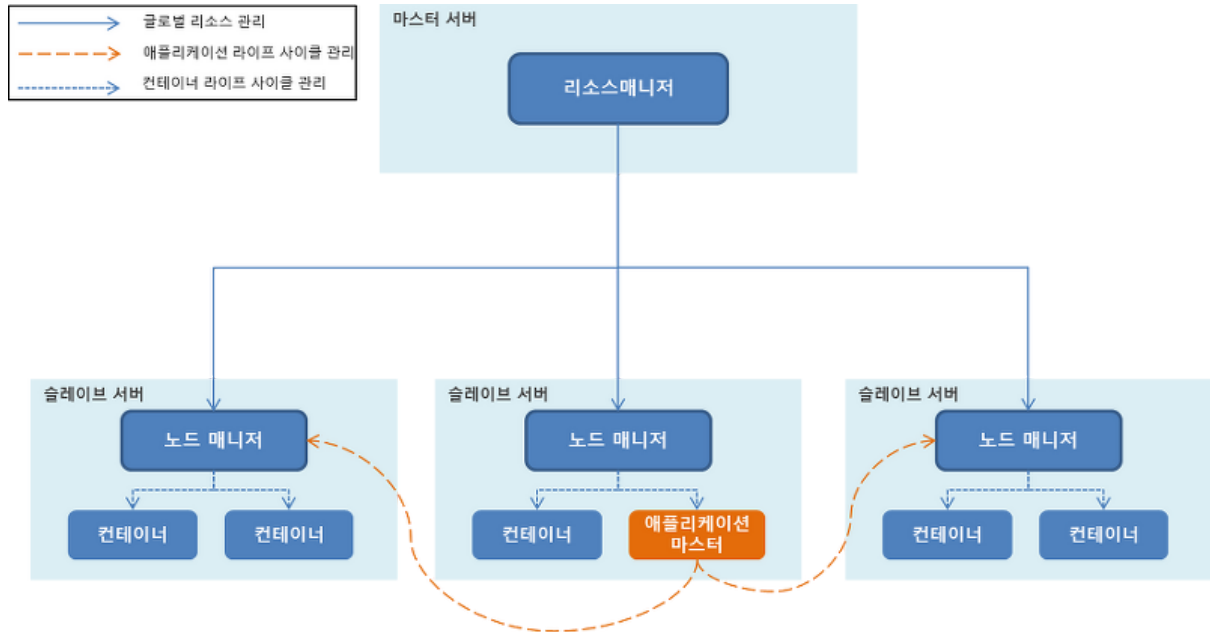
- 얀은 기존 맵리듀스 프로그램 코드를 수정하지 않고도 얀에서 실행할 수 있게 해줌

### 12.3.3 얀 아키텍처

- 하둡1 맵리듀스 아키텍처
  - 마스터 역할을 하는 잡트래커와 슬레이브 역할을 하는 태스크트래커로 구성됨



- 잡트래커의 역할
  - 리소스 관리
    - 맵리듀스는 리소스를 슬롯 단위로 관리하며, 슬롯 단위로 맵과 리듀스 태스크를 실행할 수 있음. 잡트래커는 클러스터에서 실행되는 맵리듀스의 잡에 대한 슬롯 할당 및 회수와 같은 리소스 관리를 담당
  - 스케줄링 관리
    - 잡트래커는 클러스터에서 실행되는 맵과 리듀스 태스크의 진행 상태를 모니터링. 그래서 오류가 발생하거나 진행 상태가 느린 태스크는 재실행하고, 문제가 있는 태스크를 강제 종료하기도 함
  - 태스크트래커 관리
    - 리소스와 스케줄링을 관리하기 위해서는 태스크트래커가 정상적으로 실행되고 있는지 알고 있어야 함. 이를 위해 잡트래커는 태스크트래커와 하트비트를 주고받는 식으로 태스크트래커의 상태를 확인하고, 문제가 있는 태스크트래커는 클러스터에서 제외
    - 태스크트래커는 잡트래커가 실행 요청한 잡을 태스크 단위로 실행하고, 각 태스크의 실행 상태를 주기적으로 잡트래커에 알리는 역할을 수행
- 하둡2 얀 아키텍처
  - 얀은 맵리듀스와 같이 마스터/슬레이브 구조로 구성되며, 크게 네 종류의 컴포넌트로 구성되어 있음.



#### • 각 컴포넌트의 주요 기능

##### 1. 리소스매니저(ResourceManager)

- 글로벌 스케줄러. 전체 클러스터에서 가용한 모든 시스템 자원을 관리. 양 클러스터에서 실행되는 애플리케이션이 리소스를 요청하면 이를 적절하게 분배, 리소스 사용 상태를 모니터링. 또한 리소스 매니저는 양의 마스터 서버에 해당

##### 2. 노드매니저(NodeManager)

- 맵리듀스의 태스크트래커의 기능을 담당. 태스크트래커가 각 슬레이브 서버마다 하나의 데몬이 실행된 것처럼 노드매니저도 각 슬레이브에서 하나의 데몬이 실행됨. 노드매니저는 컨테이너(Container)를 실행하고, 컨테이너의 라이프 사이클을 모니터링

##### 3. 컨테이너(Container)

- 노드매니저가 실행되는 서버의 시스템 자원을 표현, CPU, 메모리, 디스크, 네트워크 같은 다양한 시스템 자원을 표현. 컨테이너는 리소스매니저의 요청에 의해 실행되며, 하나의 서버에는 시스템 자원 현황에 따라 여러 개의 컨테이너가 실행될 수 있음. 맵리듀스의 태스크트래커가 태스크 단위로 잡을 실행했다면 노드매니저는 컨테이너 단위로 애플리케이션을 실행하고 각 상태를 스케줄링

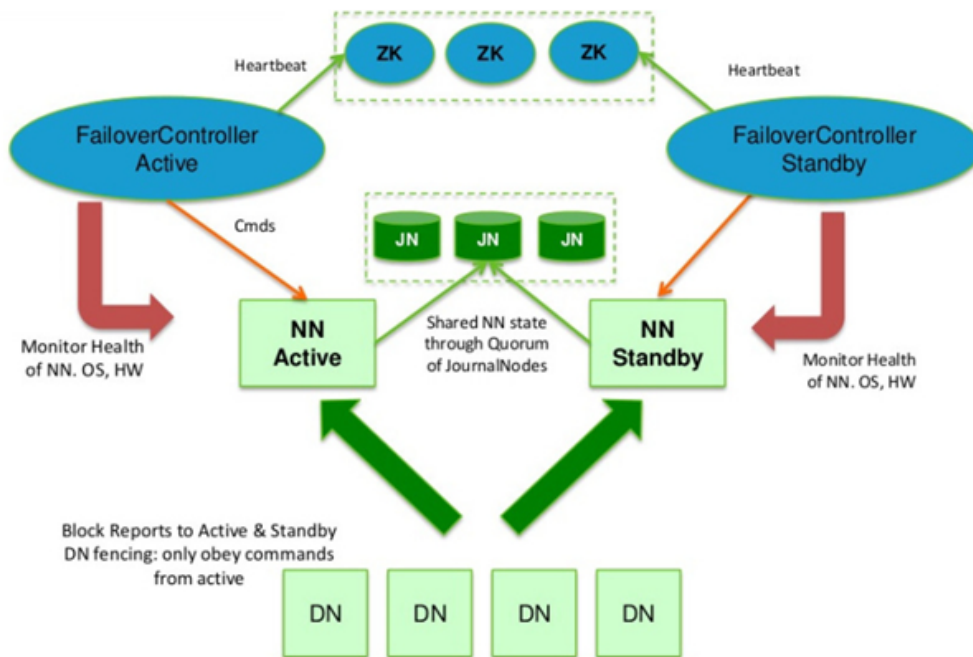
##### 4. 애플리케이션마스터(ApplicationMaster)

- 하나의 애플리케이션을 관리하는 마스터 서버. 클라이언트가 양에 애플리케이션 실행을 요청하면 양은 하나의 애플리케이션에 하나의 애플리케이션마스터를 할당. 애플리케이션마스터는 애플리케이션에 필요한 리소스를 스케줄링하고, 노드매니저에게 애플리케이션이 필요한 컨테이너를 실행할 것을 요청. 참고로 애플리케이션마스터는 컨테이너에서 실행됨
- 예를 들어, 양 클러스터에 하나의 맵리듀스 잡과 하나의 스톱 애플리케이션 실행을 요청했다면 두 개의 애플리케이션 마스터가 실행됨.

## 12.4 네임노드 HA

- 네임노드는 HDFS(데이터 저장소)에서 가장 중요한 컴포넌트

1. 네임노드가 정상적으로 동작하지 않을 경우 모든 클라이언트가 HDFS에 접근 불가하기 때문
  2. 네임노드의 파일 시스템 이미지에 문제가 생길 경우에도 HDFS에 저장된 데이터를 조회할 수 없음. 그 이유는 파일 시스템 이미지에 HDFS의 디렉터리 구조와 파일 위치가 모두 보관되어 있어서 이 정보가 유실될 경우 블록에 접근하기 위한 통로가 없어지기 때문.
  3. 네임노드의 에디트로그에 문제가 생길 경우에도 데이터가 유실될 확률이 높음. 네임노드는 HDFS에 대한 데이터 갱신 내역을 에디트로그에 저장하고, 파일 시스템 이미지를 메모리에서 관리. 보조네임노드는 체크포인팅 작업을 통해 에디트로그를 파일 시스템 이미지에 갱신. 그런데 체크포인트가 만들어지기 전에 에디트로그가 손상되고, 네임노드가 재시작된다면 손상된 에디트로그는 파일 시스템 이미지에 반영되지 않은 채로 네임노드가 구동됨
- 이러한 문제점을 없애기 위해 네임노드 HA 기능을 추가



## • 네임노드 HA의 주요 컴포넌트

### 1. 저널노드(JournalNode)

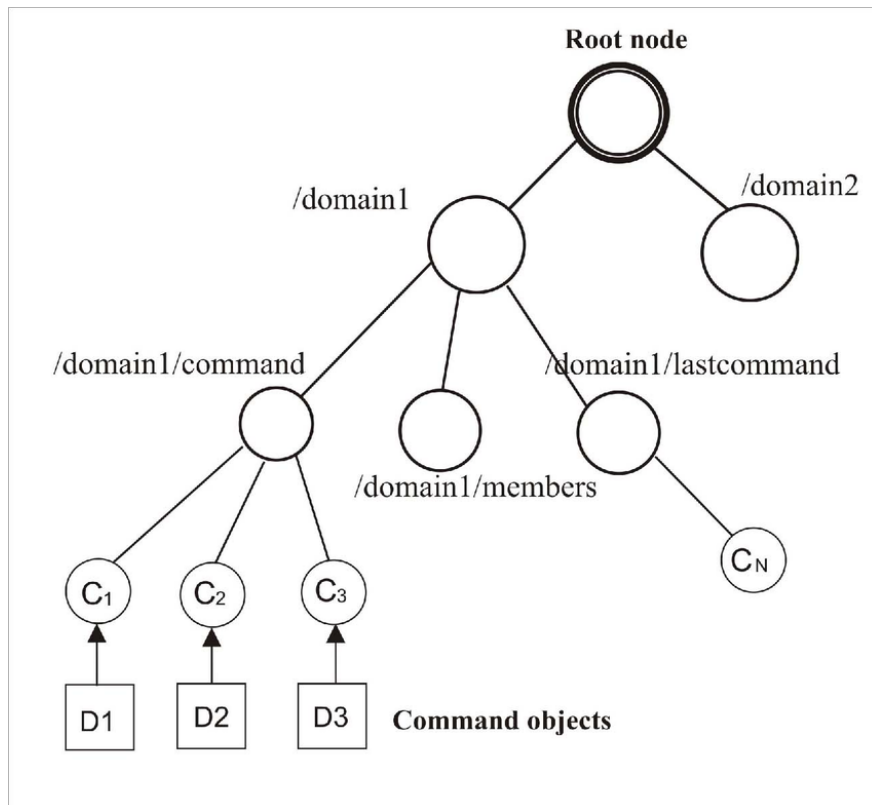
- HDFS에 저장된 파일을 수정할 경우 에디트 로그가 발생. 기존 하둡1은 네임노드만이 에디트 로그를 저장하지만, 하둡2에서는 에디트 로그를 여러 서버에 복제해서 저장. 이를 위해 하둡은 저널노드라는 새로운 컴포넌트를 제공하며, 별도의 데몬으로 실행됨. 저널노드는 에디트 로그를 자신이 실행되는 서버의 로컬 디스크에 저장. 네임노드는 저널노드에 접근하기 위한 클라이언트가 되는데, 액티브 네임노드만 에디트 로그를 저장할 권한이 있고, 스탠바이 네임노드는 조회만 요청할 수 있음.
- 저널노드는 반드시 3개 이상 실행돼야 하며, 홀수 단위로만 실행될 수 있음. 물리적으로 한 대의 서버에서 3개의 저널노드를 실행하는 것이 아니라 별도의 3대의 서버에서 각각 저널노드를 실행해야 함. N개의 저널노드는 동일한 에디트 로그를 유지하게 되며, 만약 저널노드 중 한 대에 문제가 생겨도 네임노드 입장에서는 아무런 문제 없이 에디트 로그를 저장하고 조회할 수 있음
  - 단, 위와 같이 네임노드가 저널노드의 영향을 받지 않으려면 적어도 다음 개수만큼 저널노드가 실행돼 있어야만 한다.
  - (전체 저널노드 설치 대수 // 2) + 1

- 예를 들어, 5대의 서버에 5개의 저널노드를 설치했다면 적어도 3대 이상의 저널노드가 실행돼 있어야 한다. 즉, 2대의 서버의 장애만 허용하는 것.
- 저널노드는 시스템 리소스를 적게 사용하는 데몬이기 때문에 네임 노드, 잡트래커, 리소스매니저 같은 데몬과 같은 서버에서 함께 실행돼도 문제가 되지 않음.

## 2. 주키퍼(Zookeeper)

- 네임노드 HA를 구성할 경우 어떠한 네임노드가 액티브 네임노드이고 어떠한 네임노드가 스탠바이 네임노드인지를 저장할 곳이 필요. 하둡 커뮤니티는 네임노드 HA 상태 정보를 저장하기 위한 저장소로 주키퍼(Zookeeper)를 선택
- 주키퍼는 대표적인 하둡 에코시스템 중 하나로 애플리케이션들이 잘 동작하도록 중재하는 역할. 즉, 분산시스템을 위한 코디네이터의 기능을 수행. 주키퍼는 이를 위해 네이밍 서비스, 분산 동기화, 메시지 큐, 알림 시스템과 같은 다양한 기능을 제공. 이 같은 기능은 셀 명령어로 호출하거나 클라이언트 라이브러리를 통해 이용할 수 있음
- 주키퍼의 아키텍처는 중복 서비스를 이용한고가용성을 제공하는데, 이를 위해 주키퍼 마스터를 홀수 단위로 실행. 주키퍼 클라이언트는 주키퍼 마스터가 응답하지 않을 경우 다른 주키퍼 마스터에게 요청
- 여러 대의 주키퍼 마스터를 실행할 경우 한 대는 리더가 되고, 나머지 마스터는 해당 리더를 따르는 팔로워로 설정. 각 주키퍼 마스터는 동일한 주키퍼 데이터를 복제하고 있기 때문에 클라이언트로 부터 데이터 조회 요청이 올 경우 자신이 보관하고 있는 데이터를 이용해 이에 응답. 하지만 모든 쓰기 요청은 리더로 설정된 주키퍼 마스터에게 보내짐. 리더는 해당 요청을 팔로워들에게 반영하며, 과반수 이상의 주키퍼 마스터에 적용이 완료되면 쓰기 요청을 해 클라이언트에 쓰기가 정상적으로 완료됐다는 신호를 보냄
- 주키퍼는 파일 시스템 구조와 유사한 Znode라는 트리 계층을 유지하며, znode의 각 노드에 저장할 수 있는 데이터는 1MB로 제한.

<네임노드 ha 주키퍼의 Znode 구조>



### 3. ZKFC(Zookeeper Failover Controller)

- 주키퍼에 네임노드의 HA 상태를 저장하려면 주키퍼를 제어하기 위한 주키퍼 클라이언트가 필요. 또한 액티브 네임노드에 장애가 발생할 경우 대기 상태에 있던 네임노드(스텐바이 네임노드)는 액티브 네임노드로 전환되고, 문제가 발생한 네임노드는 HDFS에서 제외돼야 함. 하둡은 위와 같은 기능을 수행하기 위해 ZKFC(ZookeeperFailoverController)를 제공. ZKFC는 네임노드가 실행되고 있는 서버에서 별도의 데몬으로 실행되며 다음과 같은 기능을 제공

1. 로컬 네임노드의 상태를 모니터링
2. 주키퍼 세션 관리. 네임노드 상태가 정상일 경우 주키퍼 마스터에 대한 세션을 유지. 그리고 액티브 네임노드에서 실행되는 ZKFC는 Znode에 락을 걸어줌
3. 자동 장애 처리(failover) 기능. 만약 액티브 네임노드에 장애가 발생할 경우 ZKFC와 주키퍼 마스터 간의 세션은 종료될 것. 그리고 이때 스탠바이 네임노드의 ZKFC가 이러한 상태 변화를 즉시 감지하고, 스탠바이 네임노드를 액티브 네임노드로 전환. 그런 다음 HDFS 클러스터에서 기존 액티브 네임노드를 제거

### 4. 네임노드(Name Node)

- 네임노드 내부에서는 QJM(Quorum Journal Manager)이 저널노드에 에디트 로그를 출력. 이때 반드시 절반 이상의 저널노드에서 정상적으로 저장했다는 응답을 받아야 해당 에디트 로그를 파일 시스템 이미지에 반영
- 즉, 서버 3대에서 3개의 저널노드를 실행했다면 적어도 2대 이상은 가동되고 있어야만 네임노드에서 에디트로고를 활용할 수 있음.
- 스탠바이 네임노드는 일정 주기로 저널노드에서 에디트 로그를 조회해 스탠바이 네임노드가 보관하고 있는 파일 시스템 이미지를 갱신. 참고로 네임노드 HA를 구성할 경우 보조네임노드



를 실행할 필요가 없음. 왜냐하면 위와 같이 스탠바이 네임노드가 체크포인팅과 유사하게 동작하고 있기 때문

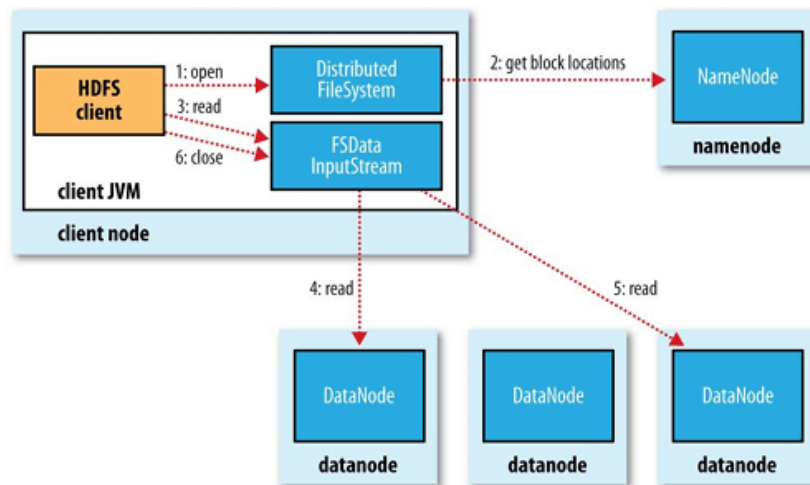
#### 5. 데이터노드

- 데이터노드는 액티브 네임노드와 스탠바이 네임노드에 모두 블록 리포트를 전송

## 12.5 HDFS 페더레이션

- HDFS 페더레이션은 기존 HDFS가 네임노드에서 기능이 집중되는 것을 막기 위해 제안된 시스템.

### 12.5.1 기존 HDFS 아키텍처

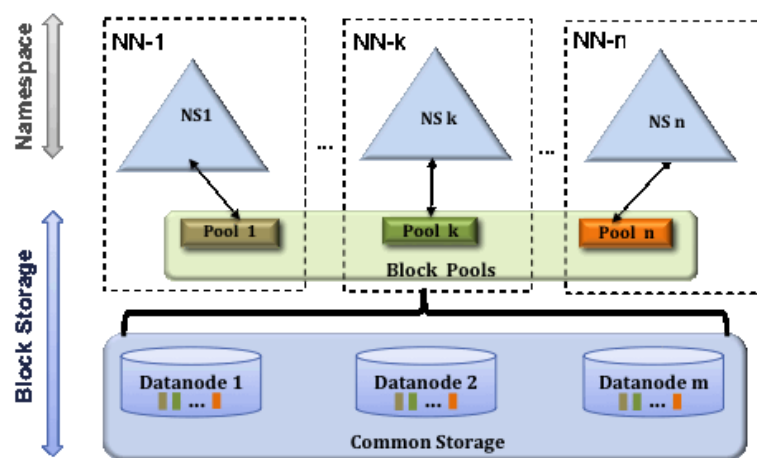


- 네임노드
  - 모든 파일과 디렉터리에 대한 메타데이터 정보인 네임스페이스를 관리. 파일시스템 이미지 파일과 에디트 로그를 이용해 네임스페이스가 유지되게 해줌.
  - 블록 관리. 데이터노드와 하트비트를 유지하며, 데이터노드의 상태를 파악하고, 블록 리프트를 이용해 블록의 위치도 파악
  - HDFS에 데이터를 읽고, 쓰고, 삭제하는 모든 트랜잭션과 복제 데이터에 대한 배치도 네임노드에서 관리
- 데이터노드
  - 블록 저장소 역할. 데이터를 로컬 파일 시스템에 저장하고, 데이터에 대한 제어 요청이 왔을 때만 데이터를 전송
- HDFS 아키텍처의 문제점
  - 네임노드 확장
    - 데이터노드는 서버만 추가하면 확장이 가능하지만, 네임노드는 수평적인 확장이 불가능. 네임노드가 구동될 때 전체 파일 시스템 이미지를 메모리에 생성하기 때문에 메모리 크기의 제한을 받을 수 밖에 없음.
  - 동일 네임노드 사용

- 모든 하둡 클라이언트가 동일한 네임노드를 사용. 서로 다른 종류의 잡이 하나의 네임노드에서 수행되기 때문에 다양한 문제가 발생할 수 있음.
- 성능에 문제가 있는 맵리듀스 잡이 수행된다면 다른 정상적인 잡이 수행되는데 영향을 줌
- 디스크 I/O를 지나치게 일으키거나 네트워크 트래픽을 점유해버리고, CPU 사용률이 올라가게 하는 등 다양한 문제를 일으킬 수 있음
- 네임스페이스와 블록 관리의 과도한 코드 공유
  - 네임노드가 제공하는 네임스페이스와 블록 관리의 서로 다른 기능. 하지만 현재 두 기능은 네임노드 관련 코드로 연결돼 있어서 복잡도가 높고 해당 기능을 확장하기가 어려운 구조

## 12.5.2 HDFS 페더레이션의 아키텍처

- 하나의 하둡 클러스터에 여러 개의 네임노드를 구동



- 네임노드
  - 다수의 네임노드로 구성되며, 각 네임노드별로 네임스페이스를 보유. 각 네임노드는 독립적
  - 각 데이터노드는 모든 네임노드에 등록. 기존의 HDFS에서처럼 하트비트와 블록 리포트를 사용해 네임노드가 데이터 읽기/쓰기를 가능하게 해줌
  - 각 네임노드는 블록풀을 추가로 저장. 블록풀이란 네임스페이스에 속하는 블록 정보를 의미. 데이터노드는 모든 블록 풀의 블록을 저장.
  - 각 블록 풀은 다른 블록 풀과 독립적으로 관리. 다른 네임노드의 협조 없이 자신의 네임스페이스만을 이용해 블록을 생성할 수도 있음. 네임노드 간에 연관성이 없기 때문에 한 네임노드에 문제가 생겨도 다른 네임노드는 정상적으로 서비스됨
- 데이터노드
  - 모든 네임노드가 사용하는 공통 블록 저장소.

## 12.5.3 네임스페이스 관리

- 네임노드는 자신만의 네임스페이스를 갖고, 다른 네임노드의 네임스페이스와는 별도로 동작. 즉, HDFS 페더레이션은 통합된 네임스페이스를 제공하지 않음. 따라서 클라이언트에 전체 네임노드의 디렉토리가 설정돼 있는 마운트 테이블을 생성해야 함. 이러한 마운트 테이블을 공유해서 사용한다면 통합된 네임스페이스처럼 모든 클라이언트가 동일한 네임스페이스를 참조할 수 있음

- 마운트 테이블은 ViewFs를 이용해 구현할 수 있음. ViewFs는 AbstractFileSystem을 확장한 형태이며, 클라이언트의 메모리에 파일 시스템을 생성. 클라이언트는 ViewFs를 이용해 다양한 파일 시스템의 네임스페이스에 접근 가능. 이때 클라이언트는 링크를 설정해 네임스페이스에 접근. 예를 들어, 클라이언트는 아래처럼 서로 다른 네 종류의 네임스페이스를 마운트 테이블로 사용할 수 있음.
- 마운트 테이블 사용 예
  - 이때, 클라이언트가 마운트 테이블에 접근하려면 URI를 반드시 "viewfs://"로 설정해야 함. 예를 들어 첫 번째 네임스페이스에 접근하려는 경우 "viewfs://user"로 URI를 지정하면 됨

```
/user -> hdfs://nnContainUserDir/user
/project/foo -> hdfs://nnProject1/projects/foo
/project/bar -> hdfs://nnProject2.projects/bar
/tmp -> hdfs://nnTmp/privateTmpForUserXXX
```

- 기본 마운트 테이블 설정
  - 모든 마운트 테이블 설정은 fs.viewfs.mounttable라는 접두어를 사용해 설정할 수 있음

```
fs.viewfs.mounttable.default.link./user= hdfs://nnContainingUserDir/user
fs.viewfs.mounttable.default.link./project/foo= hdfs://nnProject1/projects/foo
fs.viewfs.mounttable.default.link./project/bar= hdfs://nnProject2/projects/bar
fs.viewfs.mounttable.default.link./tmp= hdfs://nnTmp/privateTmpForUserXXX
```

- 이름이 다른 마운트 테이블을 정의할 수도 있음.
  - 추가 마운트 테이블 설정 예

```
fs.viewfs.mounttable.sanjayMountable.*=hdfs://sanjayMountable/
```

- 두 개 이상의 디렉터리를 병합해 새로운 마운트 테이블을 만들 수도 있음.
  - 이 때 디렉터리는 콤마(,)를 사용해서 구분

```
fs.viewfs.mounttable.default.linkMerge./user=hdfs://nnUser1/user, hdfs://nnUser2/user
```

## 12.6 HDFS 스냅샷

- 하둡2는 스냅샷 기능을 제공함으로써 스냅샷을 이용해 언제든지 복원 가능한 시점으로 HDFS를 복원할 수 있음. 스냅샷은 루트 디렉터리(/) 이하의 전체 디렉터리 혹은 특정 디렉터리 이하의 하위 디렉터리 전체를 대상으로 생성할 수 있음
- 스냅샷에 저장된 전체 파일 시스템의 상태를 복원하거나, 스냅샷의 특정 데이터만 선택해서 복원할 수도 있음.
- 관리자만이 스냅샷을 생성할 수 있으며, 일반 사용자의 경우 생성된 스냅샷을 조회만 할 수 있음
- 스냅샷은 기존 HDFS 운영에 지장을 주지 않도록 효율적으로 구현돼 있음
  - 스냅샷 생성을 요청하는 즉시 스냅샷이 생성됨
  - 메모리 사용을 최적화

- 데이터노드에 있는 블록을 복사하지 않고 스냅샷 파일에 블록 목록과 파일 크기가 저장됨
- 스냅샷을 생성하는 동안 일반적인 HDFS 동작에 영향을 주지 않음. HDFS 클라이언트가 현재 데이터에 접근할 수 있게 시간 역순으로 스냅샷을 생성

## 12.6.1 스냅샷 Directory

- 하둡2는 스냅샷을 효율적으로 관리하기 위해 스냅샷의 대상으로 설정된 Directory를 스냅샷 (snapshottable) directory라고 정의.
- 스냅샷 설정
  1. 관리자가 루트 혹은 특정 디렉터리에 대해 스냅샷을 설정할 경우, 해당 디렉터리는 스냅샷 디렉터리가 됨.
  2. 스냅샷 디렉터리 내부에 임의의 디렉터리가 생성되고, 이 디렉터리 내에 스냅샷 파일이 저장됨. 이때 스냅샷 디렉터리는 스냅샷 대상 디렉터리 아래에 있는 모든 디렉터리에 대한 스냅샷이 생성될 때까지 삭제할 수 없음. (참고: 스냅샷 디렉터리 내부에는 65,536개의 스냅샷을 동시에 생성할 수 있으며, 생성되는 디렉터리의 개수는 무제한)
    - 스냅샷 디렉터리에 생성되는 임의의 디렉터리는 ".snapshot/s<순번>"이라는 이름으로 생성됨. 순번은 최초로 생성된 스냅샷일 경우 s0, 해당 디렉터리에 대한 스냅샷을 추가로 생성할 때마다 s1,s2 같은 식으로 순번이 증가. 스냅샷을 복구할 때는 s 뒤의 숫자가 가장 큰 디렉터리가 최신 스냅샷.
    - ".snapshot"디렉터리는 HDFS의 예약어이며, 일반 사용자는 ".snapshot"디렉터리를 생성할 수 없음
    - 예를들어, /wikibooks 디렉터리에 대한 최초의 스냅샷을 생성할 경우 "/wikibooks/.snapshot/s0" 이름으로 스냅샷 디렉터리가 만들어짐.
    - 만약, /wikibooks 디렉터리에 data.txt라는 파일이 있다면 "/wikibooks/.snapshot/s0/data.txt"와 같이 스냅샷이 생성됨.

## 12.6.2 스냅샷 관리

- 스냅샷을 생성하려면 해당 디렉터리가 스냅샷을 허용할 수 있도록 설정해야 함

```
hdfs dfsadmin -allowSnapshot <스냅샷 디렉터리>
```

- 스냅샷 허용을 해제하고 싶다면 disallowSnapshot 옵션 이용

```
hdfs dfsadmin -disallowSnapshot <스냅샷 디렉터리>
```

- 스냅샷 허용 후, createSnapshot 옵션을 지정해 스냅샷을 생성할 수 있음

```
hdfs dfs -createSnapshot <스냅샷 디렉터리> <스냅샷 이름>
```

- 스냅샷이 생성된 모든 디렉터리 목록을 조회

```
hdfs lsSnapshottableDir
```

- 이미 생성된 스냅샷을 삭제하려면 deleteSnapshot 명령어 사용

```
hdfs dfs -deleteSnapshot <스냅샷 디렉터리> <스냅샷 이름>
```

- 필요에 따라 스냅샷의 이름도 변경 가능

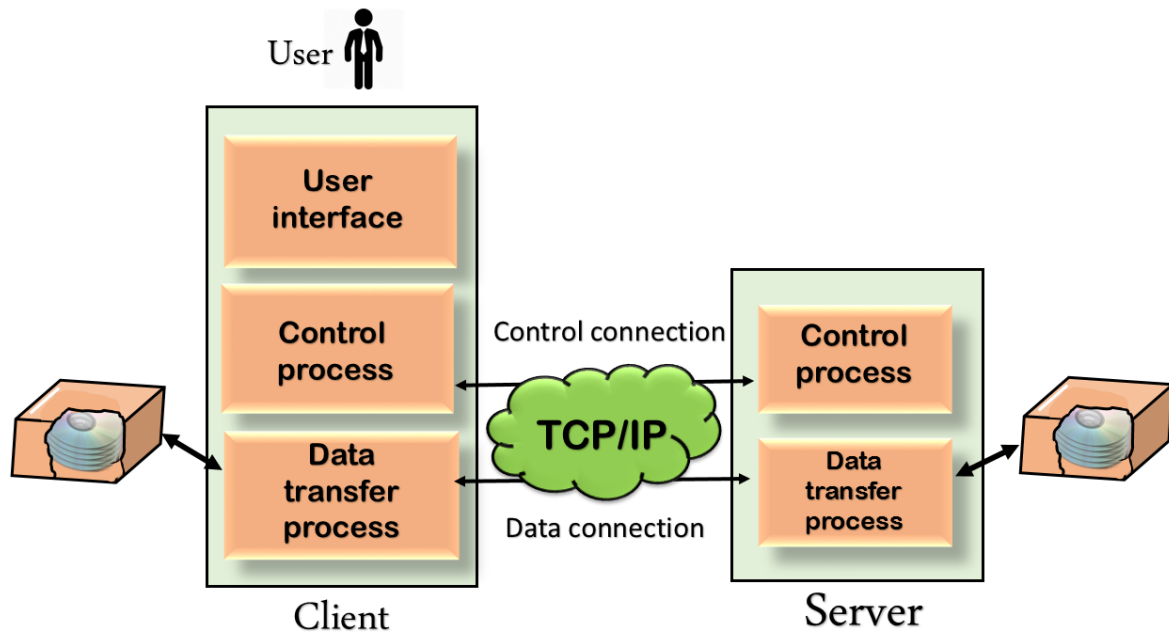
```
hdfs dfs -renameSnapshot <스냅샷 디렉터리> <기존 스냅샷 이름> <새로운 스냅샷 이름>
```

- 예제

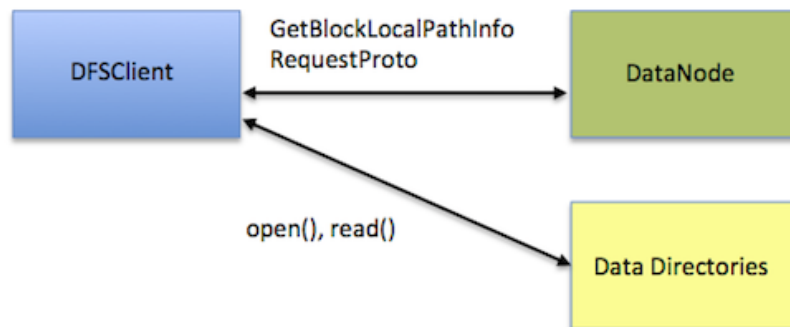
```
./bin/hdfs dfsadmin -allowSnapshot /wikibooks
./bin/hdfs dfs -createSnapshot /wikibooks wiki_snapshot
./bin/hdfs dfs -ls /wikibooks/.snapshot/
```

## 12.7 쇼트 서킷 조회

- HDFS 클러스터에 저장되는 블록은 데이터노드가 실행되는 서버의 물리적인 디스크에 저장됨. 이렇게 저장되는 블록에 대해서는 반드시 데이터노드를 통해서만 접근 가능.
- 클라이언트가 DataTransferPorocol을 이용(즉, TCP 소켓을 이용)해 데이터노드에 접속한 후, 데이터노드에서 블록의 파일을 조회한 후 로컬 파일에 접근하여 데이터노드와 데이터를 주고 받음.
  - 이러한 방식은 어떠한 HDFS 블록에 접근하더라도 TCP 소켓 통신 비용이 발생. 블록 접근이 많을수록 로컬 디스크 접근과 소켓 통신이 동시에 다량으로 발생.
  - 원격에 있는 블록 접근은 어쩔 수 없지만, 클라이언트 혹은 맵리듀스 태스크가 실행되는 서버의 로컬 디스크에 접근하는 경우에도 디스크 접근과 소켓 통신 비용이 이중으로 발생하는 것은 전체 성능을 저하시키는 원인 중 하나



- 쇼트 서킷 조회(short-circuit read)
  - 클라이언트가 로컬 파일에 저장된 블록에 접근할 경우 데이터노드와의 소켓 통신을 통하지 않고 직접 로컬 파일에 접근하는 것을 허가하는 방식.



- RequestProto 프로토콜의 GetBlockLocalPathInfo를 이용해 블록의 위치를 조회. 이 때, 한 번의 네트워크 비용 발생. 이후 블록의 위치를 통해 직접 파일을 조회

## 12.8 헤테로지니어스 스토리지

- 하둡2는 HDD(Hard Disk Drive), SSD(Solid State Drive), RAM(Random Access Memory)와 같은 다양한 종류의 스토리지를 활용할 수 있도록 헤테로지니어스(Heterogeneous) 스토리지를 지원
  - SSD : 반도체 메모리를 데이터 저장수단으로 사용. 비싼 가격과 낮은 저장 용량
  - HDD : 용량이 큰 장점이 있음.

### 12.8.1 스토리지 시장의 변화

- SSD의 대중화

- SSD는 기존의 HDD보다 수 배 이상 빠른 성능을 가짐. 최근 가격이 저렴해지고, 용량이 증가하면서 HDD를 위협.
- HDD는 이에 대응하여 용량을 높임. 데이터 안정성이 SSD에 비해 높고, 비용 대비 저장 용량 역시 여전히 HDD가 SSD보다 높음
- 메모리 시장
  - 대용량 메모리의 대중화
  - 서버용 메모리의 가격이 큰 폭으로 하락. 하지만 아직 HDD에 비해서는 비쌈

## 12.8.2 헤테로지니어스 스토리지란?

- 네임노드는 모든 데이터노드가 동일한 스토리지를 사용한다고 가정. HDFS는 서로 다른 종류의 스토리지 유형을 구분할 수 없기 때문에 애플리케이션이 특정 데이터를 성능이 좋은 스토리지에 저장(즉, 용도에 맞게 저장)하는 것은 원칙적으로 불가능. 하지만, 스토리지 시장의 변화로 이것은 큰 문제. 기업에서 다양한 성능의 스토리지를 용도에 맞게 설치, 최적화하더라도 그 위의 하둡을 설치하면 무용지물이기 때문
- 헤테로지니어스 스토리지
  - 스토리지 성능과 특성에 맞게 블록 복제본이 저장될 스토리지를 선택할 수 있는 기능. dfs, datanode, data, dir 속성을 설정할 때 각 디렉터리 앞에 스토리지 유형을 태그를 다는 형식으로 사용
  - 목표
    1. 데이터노드는 스토리지 유형을 구분할 수 있다.
    2. 네임노드는 스토리지 유형을 사용
    3. 애플리케이션은 스토리지 유형을 사용할 수 있다.
  - 지원하는 스토리지 유형
    - DISK : 기본 스토리지 유형이며, HDD를 의미
    - SSD : SSD를 의미
    - RAM\_DISK : 메모리를 의미
    - ACHIEVE : 기존 데이터를 아카이빙할 때 사용할 스토리지. 데이터 기록 밀도가 높고, CPU를 적게 사용하는 스토리지를 사용



### 데이터 아카이빙이란?

현재 운영 시스템에서 사용 빈도가 낮은 데이터를 확인하여 장기간 보관 가능한 특수 아카이브 스토리지 시스템으로 옮기는 프로세스

- 예제) HDD 2개, SSD 2개, 메모리 1개 사용

```
<property>
  <name>dfs.datanode.data.dir</name>
  <value>[DISK]/data1, [DISK]/data2, [SSD]/data3, [SSD]/data4, [RAM_DISK]/data</value>
</property>
```

## 12.8.3 스토리지 정책



- HDFS에 보관하는 데이터는 데이터의 특성과 워크로드에 따라 사용 주기가 다양할 수 있음. 예를 들어 게임 로그 최근 데이터는 빈번하게 접근, 옛날 데이터는 거의 접근하지 않을 것.
- 스토리지 정책은 다양한 데이터 액세스 패턴에 따라 데이터 블록 저장 방식을 결정할 수 있음
- 스토리지 정책 종류

## Archival Process

- For each dataset
  - Scan HDFS Audit logs to identify the data access pattern
  - Derive an Archival Policy. Example:

Time	Storage Policy	Block Placement
< 90 days	HOT	3 replicas in DISK
> 90 days < 270 days	WARM	1 DISK , 2 ARCHIVE
> 270 days	COLD	3 replica in ARCHIVE

- Set up storage policy on sub directories based on Archival Policy.
- Run Mover for the dataset

HDFS Tiered Storage | 28

- HOT : 기본 스토리지 정책, 전체 블록 복제본을 하드디스크에 저장
- WARM : 1개의 블록 복제본은 하드디스크에 저장, 나머지는 ARCHIVE용 스토리지에 저장. HOT 정책 보다는 액세스 빈도가 떨어지고, 곧 아카이빙 대상이 될 데이터가 대상
- COLD : 전체 블록 복제본을 ARCHIVE용 스토리지에 저장, 별도의 풀백 스토리지가 없음
- ONE\_SSD : 1개의 블록 복제본은 SSD에 저장, 나머지는 하드디스크에 저장. 데이터를 저장한 후 로컬 접근을 자주 하는 경우에 사용
- ALL\_SSD : 전체 블록 복제본을 SSD에 저장. 데이터 접근이 자주 발생하는 경우 사용
- LAZY\_PERSIST : 1개의 블록 복제본을 메모리에 저장한 후 나머지를 하드디스크에 저장. 임시 데이터에 빠르게 접근하는 경우에 사용
- 스토리지 정책 목록 조회

```
./bin/hdfs storagepolicies -listPolicies
```

- 스토리지 정책 설정
  - 이때 설정한 디렉터리에 포함된 전체 디렉터리와 파일도 모두 동일한 정책을 수정됨. 스토리지 정책을 변경한 경우, 스토리지 정책 자체만 변경할 뿐, 실제 스토리지를 변경하지는 않음. 따라서 스토리지 정책을 변경한 경우에는 반드시 Mover를 실행해야 함

```
./bin/hdfs storagepolicies -setStoragePolicy -path <디렉터리 및 파일명> -policy <스토리지 정책 이름>
```

- 스토리지 정책 조회



```
./bin/hdfs storagepolicies -getStoragePolicy -path <디렉터리 및 파일명>
```

- 예제

1. HDFS에 SSD 전용 디렉터리를 생성
2. 스토리지 정책을 설정
3. ALL\_SSD 정책이 설정된 디렉터리에 로컬 디스크에 있는 로그 파일을 업로드

```
bin/hdfs dfs -mkdir /data/ssd_only  
bin/hdfs storagepolicies -setStoragePolicy -path /data/ssd_only -policy ALL_SSD  
bin/hdfs dfs -put /var/hadoop/logs/20160313.log /data/ssd_only
```

## 12.8.4 무버(Mover)

- 데이터 아카이빙을 위한 마이그레이션 도구. 스토리지 정책에 맞지 않게 저장된 블록을 확인한 후, 해당 블록의 정책에 맞게 스토리지를 변경.
  - 예를 들어, HOT이나 WARM 정책이 설정된 데이터가 더는 분석할 필요가 없고 백업만 해두면 된다면 관리자는 이 데이터의 정책을 COLD로 변경한 후, 무버를 실행하여 해당 데이터의 블록을 ARCHIVE 용 스토리지로 이동시킴
  - 별도 옵션을 설정하지 않을 경우 무버는 루트 디렉터리에 포함된 모든 디렉터리와 파일을 탐색

```
bin/hdfs mover [-p <마이그레이션 대상 파일 및 디렉터리 목록> : -f <마이그레이션 대상 파일 및 디렉터리 목록이 기재된 로컬 파일명>]
```