

Chapter03. 하둡 분산 파일 시스템 (HDFS)

3.1 HDFS 기초

3.2 HDFS 아키텍처

3.2.1 블록 구조 파일 시스템

3.2.2 네임노드와 데이터노드

3.2.2.1 네임노드

3.2.2.2 데이터노드

3.2.3 HDFS의 파일저장

3.2.3.1 파일 저장 요청

3.2.3.2 패킷 전송

3.2.3.3 파일 닫기

3.2.4 HDFS의 파일 읽기

3.2.4.1 파일 조회 요청

3.2.4.2 블록 조회


3.2.4.3 입력 스트림 닫기

3.2.5 보조네임노드

3.1 HDFS 기초

- HDFS(Hadoop Distributed File System)은 수십 테라바이트 또는 페타바이트 이상의 파일을 분산된 서버에 저장하고, 많은 클라이언트가 저장된 데이터를 빠르게 처리할 수 있게 설계된 파일 시스템

범용적으로 사용하는 대용량 파일 시스템

 이름	 특징
<u>DAS</u> (<u>Direct-Attached Storage</u>).	서버에 직접 연결된 storage. 외장형 HDD. 여러 개의 HDD를 장착할 수 이쁜 외장 케이스를 이용하는 방식
<u>NAS</u> (<u>network-Attached Storage</u>).	일종의 파일 서버. 별도의 운영체제를 사용. 파일 시스템을 안정적으로 공유할 수 있음. 주로 첨부 파일이나 이미지 같은 데이터를 저장하는 데 많이 사용

Aa 이름	≡ 특징
<u>SAN (Storage Area Network)</u>	수십에서 수백 대의 SAN 스토리지를 데이터 서버에 연결해 총괄적으로 관리해주는 네트워크. DAS의 단점을 극복하기 위해 개발됐으며, 현재 SAN 기법이 시장의 절반 이상을 차지. DBMS와 같이 안정적이고 빠른 접근이 필요한 데이터를 저장하는 데 사용

- HDFS와 기존 대용량 파일 시스템의 가장 큰 차이점

- 기존 대용량 파일 시스템은 고성능 서버를 사용해야하는 반면, 저사양 서버를 이용해 스토리지를 구성할 수 있음
- 수십 혹은 수백 대의 웹 서버급 서버를 묶어서 하나의 스토리지 처럼 사용 가능
- HDFS에 저장하는 데이터는 물리적으로는 분산된 서버의 로컬 디스크에 저장돼 있음.
- 파일의 읽기 및 저장과 같은 제어는 HDFS에서 제공하는 API를 이용해 처리
- DBMS처럼 고성능과 고가용성이 필요한 경우에는 SAN을, 안정적인 파일 저장이 필요한 경우에는 NAS를 사용. 전자상거래처럼 트랜잭션이 중요한 경우 HDFS가 적합하지 않으며, 대규모 데이터를 저장하거나, 배치로 처리를 하는 경우에 HDFS를 이용

- HDFS 설계 목표

1. 장애 복구

- HDFS를 구성하는 분산 서버에는 다양한 장애가 발생할 수 있음. 예를 들어, 하드디스크에 오류가 생겨서 데이터를 저장할 때 실패할 수 있고, 디스크 복구가 불가능한 경우 데이터가 유실되는 심각한 상황이 발생할 수 있음. 또한, 네트워크에 문제가 생겨서 특정 분산 서버에 대한 네트워크가 차단될 수도 있음
- HDFS는 장애를 빠른 시간에 감지하고, 대처할 수 있게 설계. HDFS에 데이터를 저장하면, 복제 데이터도 함께 저장되어 데이터 유실을 방지. 분산 서버 간에는 주기적으로 상태를 체크해 빠른 시간에 장애를 인지하고, 대처

2. 스트리밍 방식의 데이터 접근

- HDFS는 클라이언트의 요청을 빠른 시간 내에 처리하는 것보다는 동일한 시간 내에 더 많은 데이터를 처리하는 것을 목표. HDFS는 이를 위해 랜덤 방식의 데이터 접근을 고려하지 않음. 따라서 인터넷 뱅킹, 인터넷 쇼핑몰과 같은 서비스에서 기존 파일 시스템 대신 HDFS를 사용하는 것은 적합하지 않음
- HDFS는 랜덤 접근 방식 대신 스트리밍 방식으로 데이터에 접근. 따라서 클라이언트는 끊임없이 연속된 흐름으로 데이터에 접근 가능

3. 대용량 데이터 저장

- HDFS는 하나의 파일이 기가바이트에서 테라바이트 이상의 크기로 저장될 수 있게 설계. 따라서 높은 데이터 전송 대역폭과 하나의 클러스터에서 수백 대의 노드를 지원할 수 있음. 또한 하나의 인스턴스에서는 수백만개 이상의 파일을 지원

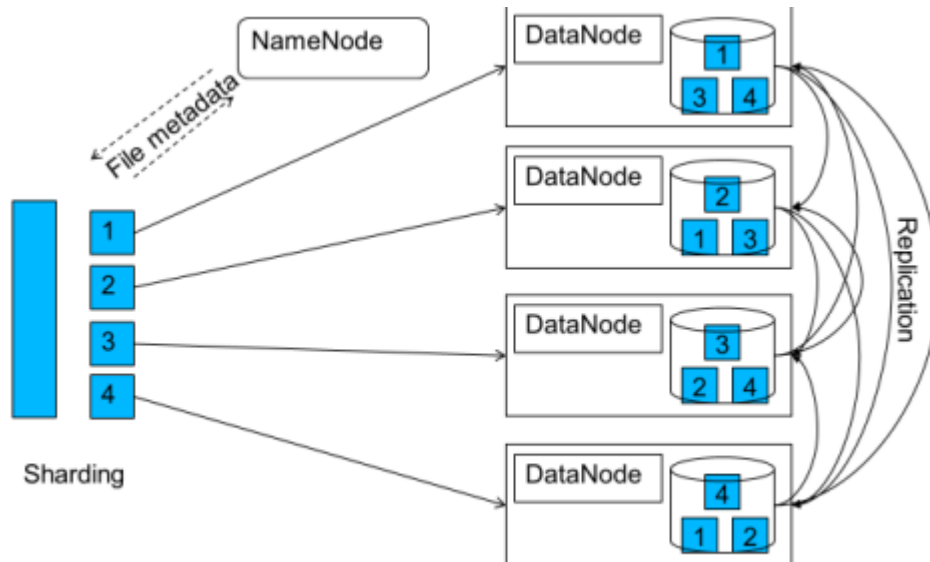
4. 데이터 무결성

- 데이터 무결성은 데이터의 일관성을 의미. 즉, 데이터의 입력이나 변경 등을 제한해 데이터의 안정성을 저해하는 요소를 막음. HDFS에서는 한 번 저장한 데이터는 더는 수정할 수 없고, 읽기만 가능하게 해서 데이터 무결성을 유지. 데이터 수정은 불가능하지만 파일 이동, 삭제, 복사할 수 있는 인터페이스 제공.
- 하둡2부터는 HDFS에 저장된 파일에 append 가능

3.2 HDFS 아키텍처

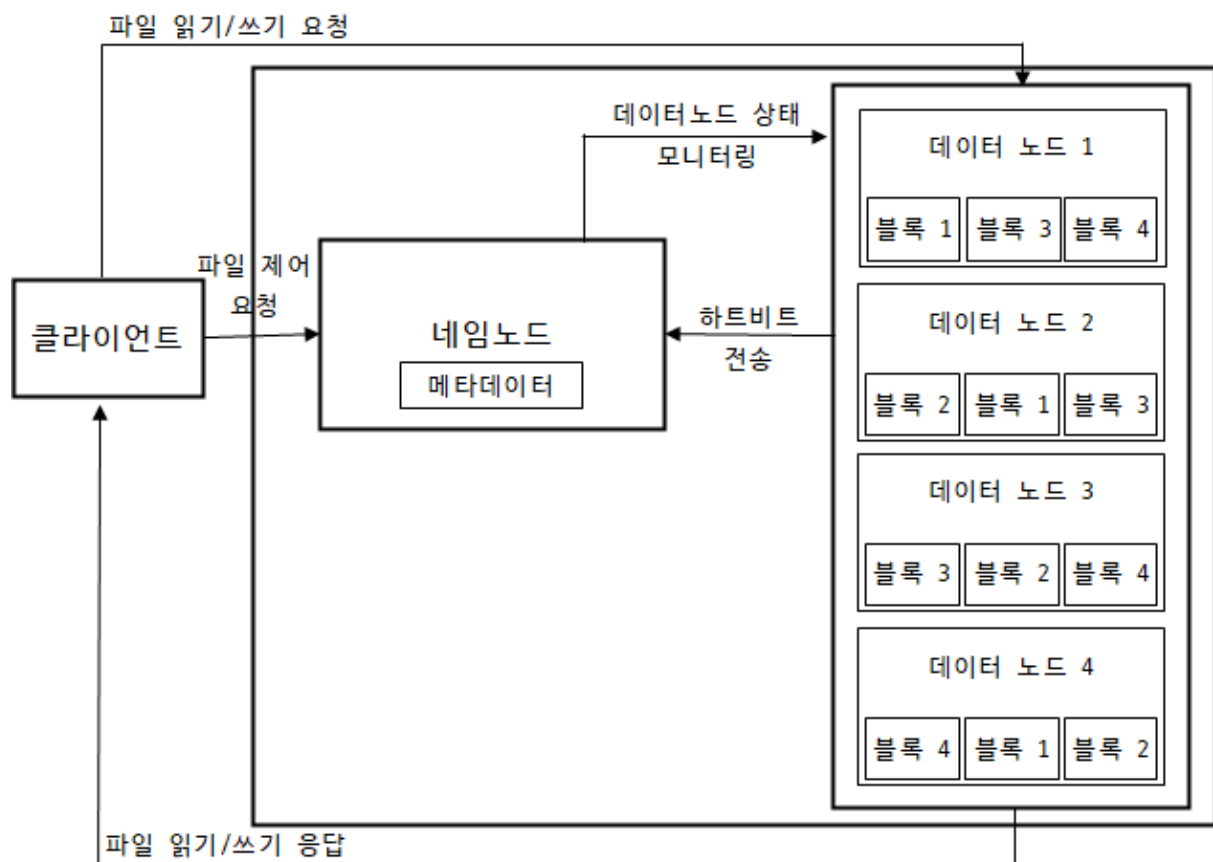
3.2.1 블록 구조 파일 시스템

- HDFS는 블록 구조의 파일 시스템. HDFS에 저장하는 파일은 특정 크기(하둡2 : 128MB, 수정 가능)의 블록으로 분산된 서버에 저장.
- 여러 개의 블록은 동일한 서버에 저장되는 것이 아니라 여러 서버에 나눠서 저장. 이 덕분에 로컬 서버의 하드디스크보다 큰 규모의 데이터를 저장할 수 있고, 그 용량을 확대할 수 있음.
- HDFS는 블록을 저장할 때 기본적으로 3개씩(수정 가능) 블록의 복제본을 저장. 이 덕분에 특정 서버의 하드디스크에 오류가 생기더라도 복제된 블록을 이용해 데이터를 조회할 수 있음.
- 기본 블록 크기보다 작은 파일도 HDFS에 저장할 수 있으며, 크기가 작은 파일이면 그 크기에 맞게 블록이 저장됨. 예를 들어, 블록 크기가 64MB인 HDFS에 10KB인 파일을 저장할 경우 해당 파일은 10KB의 디스크 공간만을 차지



3.2.2 네임노드와 데이터노드

- HDFS는 Master-Slave 아키텍처. Master server는 NameNode이며, Slave server는 DataNode



3.2.2.1 네임노드

- 네임노드 기능
 - 메타데이터 관리
 - 네임노드는 파일 시스템을 유지하기 위한 메타데이터를 관리. 메타데이터는 파일 시스템 이미지(파일명, 디렉터리, 크기, 권한)와 파일에 대한 블록 매핑 정보로 구성.
 - 네임노드는 클라이언트에게 빠르게 응답할 수 있게 메모리에 전체 메타데이터를 로딩해서 관리
 - 데이터노드 모니터링
 - 데이터노드는 네임노드에게 3초마다 하트비트(heartbeat) 메시지를 전송. 하트비트는 데이터노드 상태 정보와 데이터노드에 저장돼 있는 블록의 목록(block report)로 구성
 - 네임노드는 하트비트를 이용해 데이터노드의 실행 상태와 용량을 모니터링. 그리고 일정 기간 동안 하트비트를 전송하지 않는 데이터노드가 있을 경우 장애가 발생한 서버로 판단
 - 블록 관리
 - 네임노드는 다양한 방법으로 블록을 관리. 네임노드는 하트비트를 이용해 장애가 발생한 데이터노드를 발견하면 해당 데이터노드의 블록을 새로운 데이터노드로 복제.
 - 용량이 부족한 데이터노드가 있다면 용량에 여유가 있는 데이터노드로 블록을 이동시킴.
 - 블록의 복제본 수를 관리 만약 복제본 수와 일치하지 않는 블록이 발견될 경우 추가로 블록을 복제하거나 삭제해줌
 - 클라이언트 요청 접수
 - 클라이언트가 HDFS에 접근하려면 반드시 네임노드에 먼저 접속해야 함.
 - HDFS에 파일을 저장하는 경우 기존 파일의 저장 여부와 권한 확인의 절차를 거쳐서 저장을 승인
 - HDFS에 저장된 파일을 조회하는 경우 블록의 위치 정보를 반환

3.2.2.2 데이터노드

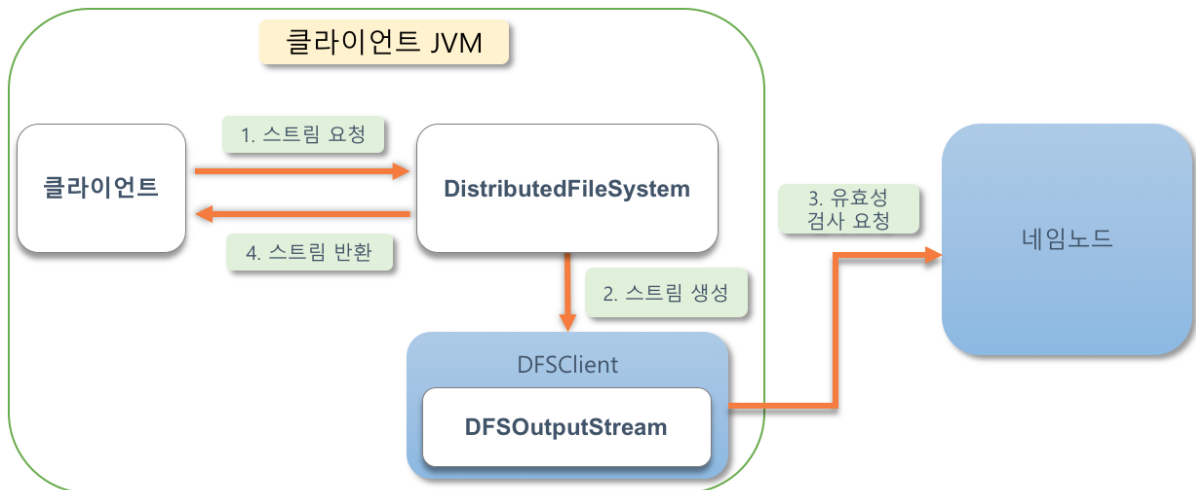
- 데이터노드는 클라이언트가 HDFS에 저장하는 파일을 로컬 디스크에 유지. 이때 로컬 디스크에 저장되는 파일은 두 종류로 구성.
 - 첫 번째 파일 : 실제 데이터가 저장돼 있는 로우 데이터

- 두 번째 파일 : 체크섬이나 파일 생성 일자와 같은 메타데이터가 설정돼 있는 파일

3.2.3 HDFS의 파일저장

3.2.3.1 파일 저장 요청

- 클라이언트가 HDFS에 파일을 저장하는 경우 파일을 저장하기 위한 스트림을 생성해야 함.



1. 스트림 요청

- 하둡은 FileSystem이라는 추상 클래스에 일반적인 파일 시스템을 관리하기 위한 메서드를 정의. 이 추상 클래스를 상속받아 각 시스템에 맞게 구현된 파일 시스템에 클래스를 제공.
- HDFS에 파일을 저장하는 경우에는 파일 시스템 클래스 중 DistributedFileSystem를 사용. 클라이언트는 DistributedFileSystem의 create 메서드를 호출해 스트림 객체를 생성

2. 스트림 생성

- DistributedFileSystem은 클라이언트에게 반환할 스트림 객체로, FSDataOutputStream을 생성. FSDataOutputStream은 데이터노드와 네임노드의 통신을 관리하는 DFSOutputStream을 래핑하는 Wrapper class. DFSOutputStream을 생성하기 위해 DFSClient의 create 메서드를 호출
 - Wrapper class : 기본형 변수를 객체로 감싸서 객체로 사용할수 있게끔 하는 class

3. 유효성 검사 요청

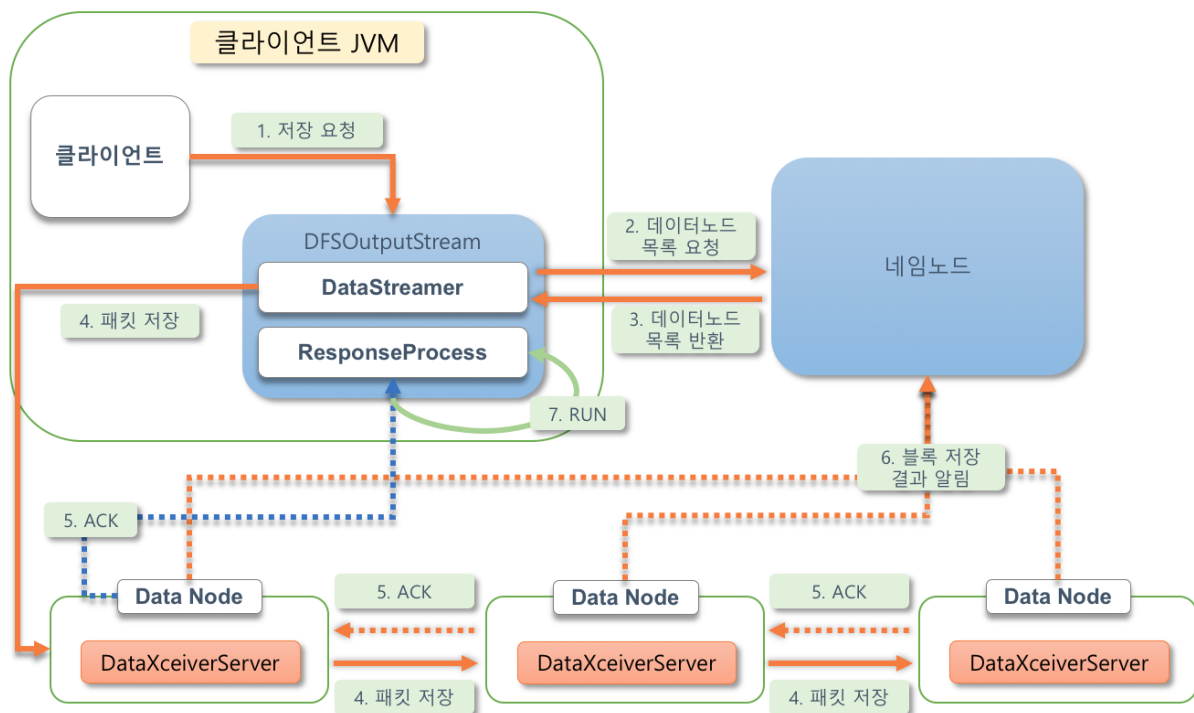
- DFSClient는 DFSOutputStream을 생성. 이때 DFSOutputStream은 RPC 통신으로, 네임노드의 create 메서드를 호출. 네임노드는 클라이언트의 요청이 유효한지 검사를 진행. 이미 생성된 파일이거나, 권한에 문제가 있거나, 현재 파일 시스템의 용량을 초과한다면 오류가 발생. 네임노드는 파일 유효성 검사가 정상일 경우 파일 시스템 이미지에 해당 파일의 엔트리를 추가. 마지막으로 네임노드는 클라이언트에게 파일을 저장할 수 있는 제어권을 부과

4. 스트림 반환

- 네임노드의 유효성 검사를 통과했다면 DFSOutputStream객체가 정상적으로 생성됨. DistributedFileSystem은 DFSOutputStream을 래핑한 FSDataOutputStream을 클라이언트에게 반환

3.2.3.2 패킷 전송

- 클라이언트가 네임노드에게서 파일 제어권을 얻게 되면 파일 저장을 진행. 이때 클라이언트는 파일을 네임노드에게 저장하지 않고, 각 데이터노드에 전송. 그리고 저장한 파일을 패킷 단위로 나눠서 전송.



1. 저장 요청

- 클라이언트는 스트림 객체의 write 메서드를 호출해 파일 저장을 시작.
DFSOutputStream은 클라이언트가 저장하는 파일을 64K 크기의 패킷으로 분할

2. 데이터노드 목록 요청

- DFSOutputStream은 저장할 패킷을 내부 큐인 데이터큐(dataQueue)에 등록. DFSOutputStream의 내부스레드가 데이터큐에 패킷이 등록된 것을 확인하면 DFSOutputStream의 내장 클래스인 DataStreamer는 네임노드의 addBlock 메서드를 호출

3. 데이터노드 목록 반환

- 네임노드는 DataStreamer에게 블록을 저장할 데이터노드 목록을 반환. 이 목록은 복제본 수와 동일한 수의 데이터노드를 연결한 파이프라인을 형성.
- 예를들어 복제본 수가 3개로 형성되어 있다면, 데이터노드 3개가 파이프라인을 형성

4. 패킷 저장

- DataStreamer는 파이프라인의 첫 번째 데이터노드로 패킷 전송을 시작. 데이터노드는 클라이언트와 다른 데이터노드와 패킷을 주고받기 위해 DataXceiverServer 데몬을 실행.
- DataXceiverServer는 클라이언트 및 다른 데이터노드와 패킷 교환 기능을 제공
- 첫 번째 데이터노드는 패킷을 저장하면서, 두 번째 데이터노드에게 패킷 저장을 요청. 두 번째 데이터노드에 패킷을 저장하면서, 세 번째 데이터노드에게 패킷 저장을 요청. 마지막으로 세 번째 데이터노드가 패킷을 저장.
- 첫 번째 데이터노드에 패킷을 저장할 때 DFSOutputStream은 내부 큐인 승인큐(ackQueue)에 패킷을 등록. 승인큐는 패킷 전송이 완료됐다는 응답을 기다리는 패킷이 등록되어 있으며, 모든 데이터노드로부터 응답을 받았을 때만 해당 패킷이 제거

5. ACK

- 각 데이터노드는 패킷이 정상적으로 저장되면 자신에게 패킷을 전송한 데이터노드에게 ACK 메시지를 전송. ACK 메시지는 패킷 수신이 정상적으로 완료됐다는 승인 메시지. 승인 메시지는 파이프라인을 통해 DFSOutputStream에게까지 전달.

6. 블록 저장 결과 알림

- 각 데이터노드는 패킷저장이 완료되면 네임노드의 blockReceived메서드를 호출. 이를 통해 네임노드는 해당 블록이 정상적으로 저장됐다는 것을 인지

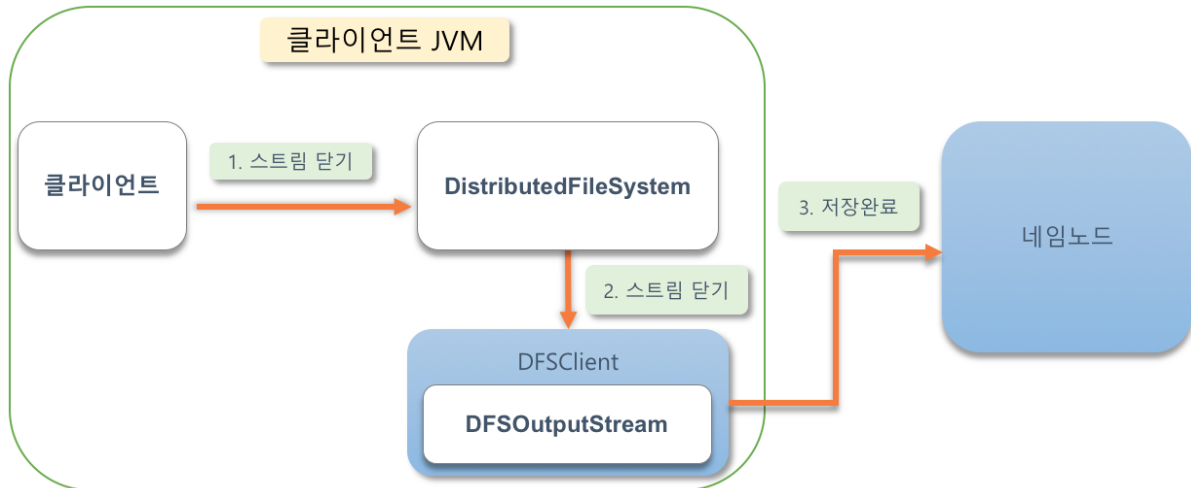
7. RUN

- DFSOutputStream의 내부 스레드인 ResponseProcessor는 파이프라인에 있는 모든 데이터노드로부터 승인 메시지를 받게 되면, 해당 패킷을 승인큐에서 제거.

- 만약 패킷을 전송하는 중에 장애가 발생하면 승인 큐에 있는 모든 패킷을 데이터큐로 이동. 그리고 네임노드에게서 장애가 발생한 데이터노드가 제거된 새로운 데이터노드 목록을 내려받음. 마지막으로 새로운 파이프라인을 생성한 후 다시 패킷 전송 작업을 시작

3.2.3.3 파일 닫기

- 스트림을 닫고 파일 저장을 완료



1. 스트림 닫기

- 클라이언트는 DistributedFileSystem의 close 메서드를 호출해 파일 닫기를 요청

2. 스트림 닫기

- DistributedFileSystem은 DFSOutputStream의 close 메서드를 호출. 이 메서드는 DFSOutputStream에 남아있는 모든 패킷을 파이프라인으로 flush 함.
 - flush : 현재 버퍼에 저장되어 있는 내용을 클라이언트로 전송하고 버퍼를 비움

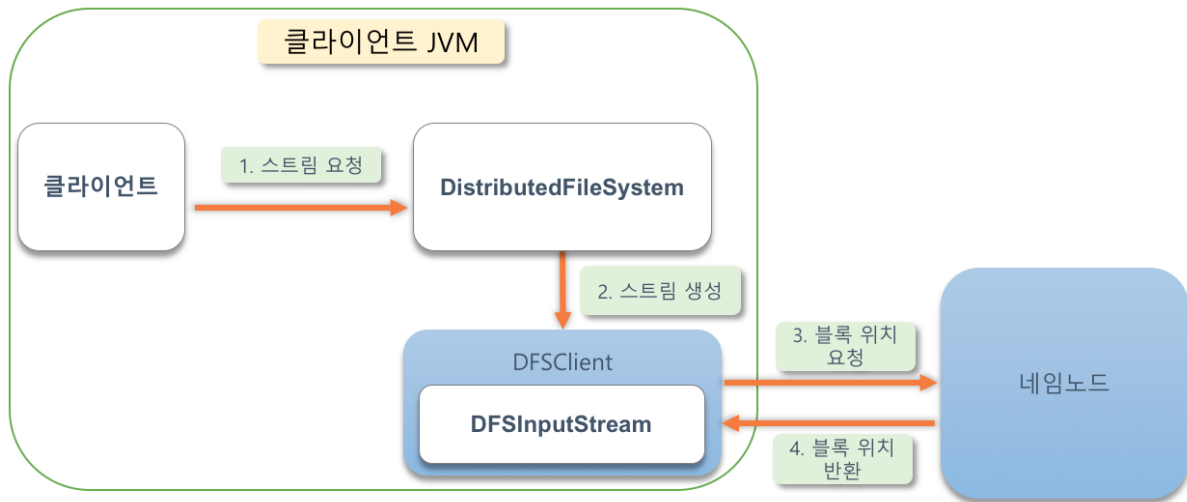
3. 저장 완료

- DFSOutputStream은 네임노드의 complete 메서드를 호출해 패킷이 정상적으로 저장됐는지 확인. 네임노드의 최소 블록 복제본 수만 저장됐다면 complete 메서드는 true를 반환, DFSOutputStream은 true를 반환받으면 파일 저장이 완료된 것으로 설정.

3.2.4 HDFS의 파일 읽기

3.2.4.1 파일 조회 요청

- 클라이언트가 HDFS에 파일을 조회하는 경우 파일을 조회하기 위한 입력 스트림 객체를 생성해야 함.



1. 스트림 요청

- 클라이언트는 DistributedFileSystem의 open메서드를 호출해 스트림 객체 생성을 요청

2. 스트림 생성

- DistributedFileSystem은 FSDataInputStream객체를 생성. 이때 FSDataInputStream은 DFSDataInputStream과 DFSInputStream을 차례로 래핑. DistributedFileSystem은 마지막으로 래핑이 되는 DFSInputStream을 생성하기 위해 DFSCient의 open 메서드를 호출

3. 블록 위치 요청

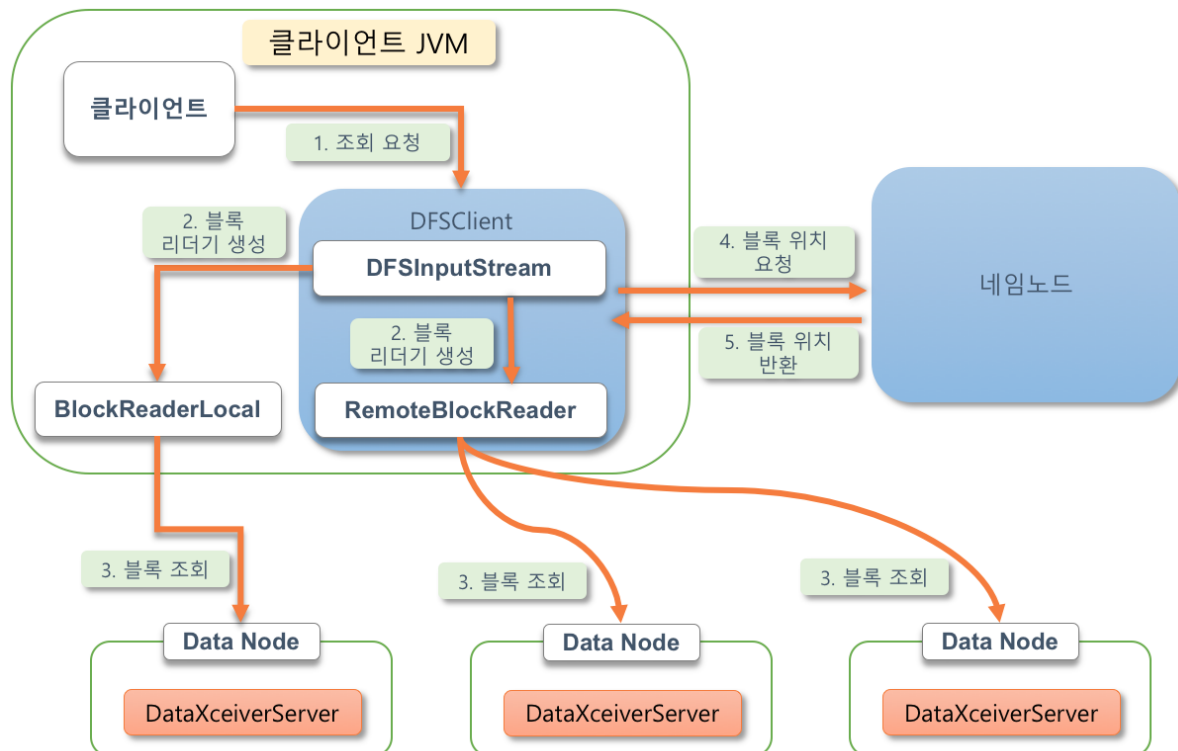
- DFSCient는 DFSInputStream을 생성. 이때 DFSInputStream은 네임노드의 getBlockLocations 메서드를 호출해 조회 대상 파일의 블록 위치 정보를 조회.
- DFSInputStream은 한 번에 모든 블록을 조회하지 않고, 기본 블록 크기의 10배 수만큼 블록을 조회. 예를 들어, HDFS의 기본 블록 크기가 64MB이면 640MB의 블록을 조회

4. 블록 위치 반환

- 네임노드는 조회 대상 파일의 블록 위치 목록을 생성한 후 목록을 클라이언트에 가까운 순으로 정렬. 정렬이 완료되면 DFSInputStream에 정렬된 블록 위치 목록을 반환. DistributedSystem은 DFSCient로부터 전달받은 DFSInputStream을 이용해 FSDataInputStream으로 생성해서 클라이언트에게 반환

3.2.4.2 블록 조회

- 클라이언트가 실제 블록을 조회하는 과정



1. 조회 요청

- 클라이언트는 입력 스트림 객체의 read 메서드를 호출해 스트림 조회를 요청

2. 블록 리더기 생성

- DFSInputStream은 첫 번째 블록과 가장 가까운 데이터노드를 조회한 후, 해당 블록을 조회하기 위한 리더기를 생성.
- 클라이언트와 블록이 저장된 데이터노드가 같은 서버에 있다면 로컬 블록 리더기인 BlockReaderLocal을 생성하고, 데이터노드가 원격에 있을 경우에는 RemoteBlockReader를 생성

3. 블록 조회

- DFSInputStream은 리더기의 read 메서드를 호출해 블록을 조회. BlockReaderLocal은 로컬 파일 시스템에 저장된 블록을 DFSInputStream에게 반환. RemoteBlockReader는 원격에 있는 데이터노드에게 블록을 요청하며, 데이터노드의 DataXceiverServer가 블록을 DFSInputStream에게 반환.
- DFSInputStream은 조회한 데이터의 체크섬을 검증하며, 체크섬에 문제가 있을 경우 다른 데이터노드에게 블록 조회를 요청

4. 블록 위치 요청

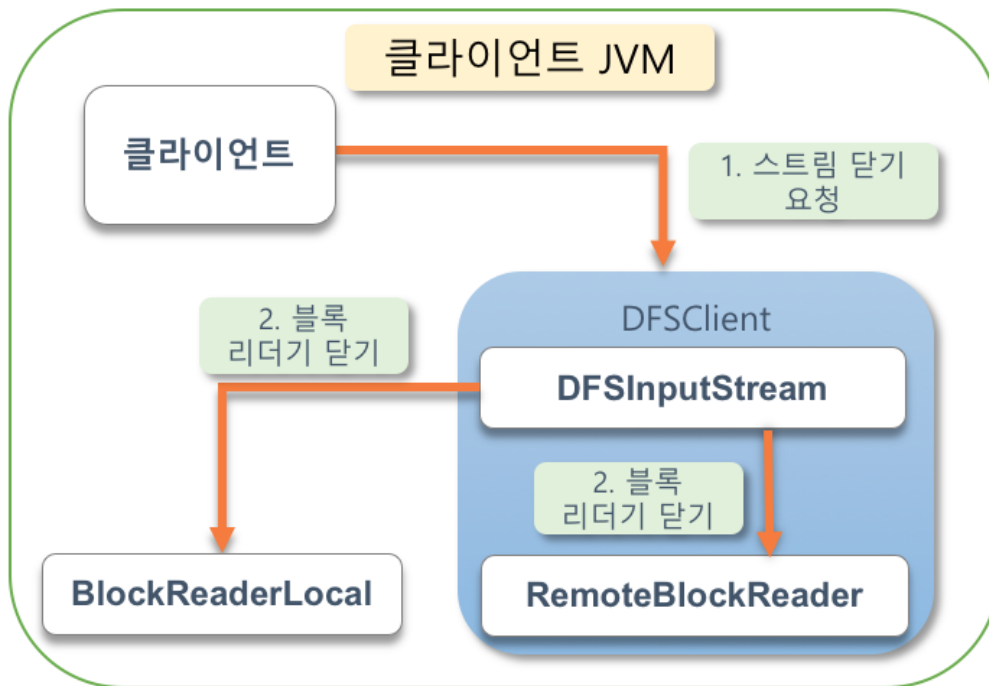
- DFSInputStream은 파일을 모두 읽을 때까지 계속해서 블록을 조회. 만약 DFSInputStream이 저장하고 있던 블록을 모두 읽었는데도 파일을 모두 읽지 못했다면 네임노드의 getBlockLocations 메서드를 호출해 필요한 블록 위치 정보를 다시 요청.
- 위와 같이 파일을 끊김 없이 연속적으로 읽기 때문에 클라이언트는 스트리밍 데이터를 읽는 것처럼 처리할 수 있음

5. 블록 위치 반환

- 네임노드는 DFSInputStream에게 클라이언트에게 가까운 수으로 정렬된 블록 위치 목록을 반환

3.2.4.3 입력 스트림 닫기

- 클라이언트가 모든 블록을 읽고 나면 입력 스트림 객체를 닫아야 함.



1. 스트림 닫기 요청

- 클라이언트는 입력 스트림 객체의 close 메서드를 요청해 스트림 닫기를 요청

2. 블록 리더기 닫기

- DFSInputStream은 데이터노드와 연결돼 있는 커넥션을 종료. 그리고 블록 조회용으로 사용했던 리더기도 닫아줌

3.2.5 보조네임노드

- 네임노드는 클라이언트에게 빠르게 응답할 수 있게 메모리에 전체 메타데이터를 로딩해서 관리. 하지만 메모리에만 데이터를 유지하면 서버가 재부팅될 경우 모든 메타데이터가 유실될 수 있음. HDFS는 이러한 문제점을 극복하기 위해서 editslog와 fsimage라는 두 개의 파일을 생성
- editslog
 - HDFS의 모든 변경 이력을 저장
 - HDFS는 클라이언트가 파일을 저장하거나, 삭제하거나, 혹은 이동하는 경우 editslog와 메모리에 로딩돼 있는 메타데이터에 기록.
- fsimage
 - 메모리에 저장된 메타데이터의 파일 시스템 이미지를 저장한 파일.
- 네임노드 구동시, editslog와 fsimage 파일을 사용하는 단계
 1. 네임노드가 구동되면 로컬에 저장된 editslog와 fsimage를 조회
 2. 메모리에 fsimage를 로딩해 파일 시스템 이미지를 생성
 3. 메모리에 로딩된 파일 시스템 이미지에 editslog에 기록된 변경 이력을 적용
 4. 메모리에 로딩된 파일 시스템 이미지를 이용해 fsimage 파일을 갱신
 5. editslog를 초기화
 6. 데이터노드가 전송된 블록리포트를 메모리에 로딩된 파일 시스템 이미지에 적용
- 이때, editslog의 크기가 클 경우, 3번 단계를 진행할 때 많은 시간이 소요. 이러한 문제를 해결하기 위해 보조네임노드(Secondary Name Node)라는 노드를 제공.
- 보조네임노드(Secondary Name Node)
 - 주기적으로 네임노드의 fsimage를 갱신하는 역할. 이러한 작업을 checkpoint라고 함. 따라서 흔히 보조네임노드를 checkpointing server라고도 표현.
 - 체크포인트팅이 완료되면 네임노드의 fsimage는 최신 내역으로 갱신, editslog의 크기는 축소
 - 체크포인트팅은 1시간마다(수정 가능) 한 번씩 일어남.
- 작업단계

1. 로그 롤링 요청

- 보조네임노드는 네임노드에게 editslog를 롤링할 것을 요청.
 - 로그 롤링 : 현재 로그 파일의 이름을 변경하고, 원래 이름으로 새 로그 파일을 만드는 것.

2. 로그 롤링

- 네임노드는 기존 editslog를 롤링한 후, editslog.new를 생성

3. 다운로드

- 보조네임노드는 네임노드에 저장된 롤링된 editslog와 fsimage를 다운로드

4. 병합

- 보조네임노드는 다운받은 fsimage를 메모리에 로딩하고, editslog에 있는 변경 이력을 메모리에 로딩된 파일 시스템 이미지에 적용.
- 메모리 갱신이 완료되면 새로운 fsimage를 생성하며, 이 파일을 체크포인팅할 때 사용. 이때 파일명은 fsimage.ckpt로 생성

5. 전송

- 보조네임노드는 fsimage.ckpt를 네임노드에게 전송

6. 완료

- 네임노드는 로컬에 저장돼 있던 fsimage를 보조네임노드가 전송한 fsimage.ckpt로 변경. 그리고 editslog.new 파일명을 editslog로 변경