



# Chapter 5. 하둡 I/O

## 5.1 데이터 무결성

완전한 수명 주기를 거치며 데이터의 정확성과 일관성을 유지하고 보증하는 것

- 체크섬?
  - 손상된 데이터를 검출하는 방법
  - 데이터 원상복구 방식이 아닌 단순한 에러 검출만 수행하며 보통 CRC-32 를 사용한다.

## 5.1.1 HDFS의 무결성

기본 설정 : 512바이트, CRC-32C (4바이트, long타입)

- 쓰기

- 클라이언트가 데이터노드 파이프라인으로 데이터를 보내면 파이프라인의 마지막 데이터노드가 해당 데이터의 체크섬을 검증
- 만일 에러 검출하면 클라이언트는 **IOException**의 서브클래스를 예외로 받고 애플리케이션 특성에 맞게 처리한다.

- 읽기

- 클라이언트가 데이터노드에 저장된 체크섬과 수신된 데이터로부터 계산된 체크섬을 검증
- 클라이언트가 검증 후 데이터노드는 체크섬 검증에 대한 로그를 갱신하며, 이는 오류 디스크 검출에 유용하다. (각 데이터노드는 체크섬 검증 로그를 영구 저장)

- **DataBlockScanner**

- **DataBlockScanner** : 각 데이터노드가 저장된 모든 블록을 주기적으로 검증
- 물리적 저장 매체에서 발생할 수도 있는 'bit rot'에 의한 데이터 손실을 피할 수 있음

- 손상된 블록 치료 방법

- 클라이언트가 읽는 과정에서 에러 검출 > 훼손된 데이터노드에 대한 정보를 네임노드에 보고, **ChecksumException** 발생시킴
- 이후, 네임노드가 블록 복제본이 손상됨을 표시하고 복사를 방지한다. 그리고 해당 블록을 다른 데이터 노드에 복제되도록 스케줄링하여 블록의 복제 계수를 원래 수준으로 복구한다.

## 5.1.2 LocalFileSystem

클라이언트 측 체크섬을 수행

- 파일 읽기 > 체크섬 검증 > 에러검출 > LocalFileSystem이 ChecksumException 발생
- RawLocalFileSystem
  - 기존 파일시스템이 자체적으로 체크섬을 지원하는 경우 LocalFileSystem의 체크섬을 비활성화하여 사용
  - 방법 1 : fs.file.impl 속성을 org.apache.hadoop.fs.RawLocalFileSystem 값으로 설정하여 파일 URI 구현을 변경
  - 방법 2 : RawLocalFileSystem 인스턴스를 직접 생성 (일부 읽기에 대해서만 검증 비활성화시 유용)

## 5.1.3 ChecksumFileSystem

체크섬이 없는 파일시스템에 체크섬 기능을 추가함, LocalFileSystem이 동작할 때 사용

- 파일 읽기 > 체크섬 검증 > 에러검출
  - > ChecksumFileSystem이 reportChecksumFailure() 메서드 호출
  - & LocalFileSystem이 문제가 되는 파일과 체크섬을 같은 디바이스의 bad\_files로 이동


∴ 관리자가 주기적으로 훼손된 파일 체크하고 적절한 처리 해야함

## 5.2 압축

- 장점
  - 저장공간 줄임
  - 네트워크 또는 디스크로부터 데이터 전송을 고속화함
- 특징
  - 공간-시간 트레이드오프 관계
- 도구
  - : -1(스피드 최적화) ~ -9(공간 최적화)
    - gzip: 일반적 목적, -5
    - bzip2: gzip보다 효율적이지만 더 느림, 압축해제 속도가 압축속도보다 빠름
    - LZO, LZ4, Snappy: 속도 최적화 (압축해제 속도: Snappy = LZ4 > LZO)

표 5-1 압축 포맷의 요약


압축 포맷	도구	알고리즘	파일 확장명	분할 가능
DEFLATE <sup>a</sup>	N/A	DEFLATE	.deflate	No
gzip	gzip	DEFLATE	.gz	No
bzip2	bzip2	bzip2	.bz2	Yes
LZO	lzop	LZO	.lzo	No <sup>b</sup>
LZ4	N/A	LZ4	.lz4	No
Snappy	N/A	Snappy	.snappy	No



## 5.2.1 코덱

압축-해제 알고리즘을 구현한 것

- **CompressionCodec을 통한 압축 및 해제 스트림**
  - `createOutputStream(OutputStream out)` 메서드  
: 출력 스트림에 쓸 데이터를 압축할 때 사용, 압축되지 않은 데이터를 압축된 형태로 내부 스트림에 쓰는 `CompressionOutputStream`을 생성
  - `createInputStream(InputStream in)` 메서드  
: 입력 스트림으로부터 읽어 들인 데이터를 압축 해제할 때 사용, 기존 스트림으로부터 비압축 데이터를 읽을 수 있는 `CompressionInputStream`을 반환
- **CompressionCodecFactory를 사용하여 CompressionCodec 유추하기**
  - `getCodec()` 메서드  
: 지정된 파일에 대한 `Path` 객체를 인자로 받아 파일 확장명에 맞는 `CompressionCodec`을 찾아줌
  - `removeSuffix()` 정적 메서드  
: 파일 접미사를 제거하여 출력 파일의 이름을 생성하는데 사용



## 5.2.1 코덱

압축-해제 알고리즘을 구현한 것

- 원시 라이브러리  
: 성능관점에서 사용하는 것이 바람직함
  - 구현체?(=규격서)
  - 모든 압축 포맷싱 원시 구현체를 가지고 있지만 모든 포맷이 자바 구현체를 가지고 있는 것은 아님
  - 하둡은 `lib/native` 디렉터리에 미리 빌드된 64비트 리눅스용 원시 압축 라이브러리인 `libhadoop.so`를 제공, 다른 플랫폼의 경우 직접 라이브러리 컴파일해야함
  - 코덱 풀 (CodecPool)|  
: 많은 압축 또는 해제 작업을 수행할 때 압축기와 해제기를 재사용하여 객체 생성 비용을 절감할 수 있다.



## 5.2.2 압축과 입력 스플릿

맵리듀스로 처리되는 데이터를 어떻게 압축할지 고민하는 시점에 압축 포맷이 분할을 지원하는지 여부를 알고 있어야 한다.

- **고려사항**
  - 파일 크기
  - 포맷
  - 처리에 사용되는 도구

## 5.2.3 맵리듀스에서 압축 사용하기

입력 파일이 압축되면 맵리듀스는 파일 확장명을 통해 사용할 코덱을 결정하고 파일을 읽을 때 자동으로 압축을 해제한다.

- 세 가지 방법