

# ACID의 의미

**ACID**는 데이터베이스 트랜잭션이 “정확하고 일관성 있게 수행되도록 보장”하는 핵심 원칙을 말합니다.

ACID는 다음 네 가지 속성의 약자입니다.

## 1. ACID의 의미

항목	영어	설명
A	<b>Atomicity</b> (원자성)	트랜잭션은 모두 실행되거나, 전혀 실행되지 않아야 함. 일부만 수행된 상태로 남지 않음.
C	<b>Consistency</b> (일관성)	트랜잭션이 완료된 후에도 데이터베이스는 항상 일관된 상태를 유지해야 함. 모든 제약조건을 만족해야 함.
I	<b>Isolation</b> (고립성)	동시에 여러 트랜잭션이 실행되더라도, 각 트랜잭션은 서로 간섭하지 않아야 함. 마치 혼자 실행되는 것처럼 보여야 함.
D	<b>Durability</b> (지속성)	트랜잭션이 성공적으로 완료되면, 시스템 장애가 발생하더라도 그 결과는 영구적으로 보존되어야 함.

## 2. 쉽게 이해하는 예시

예를 들어, “A 계좌에서 B 계좌로 10만 원을 송금”하는 트랜잭션을 생각해봅시다.

1. A 계좌에서 10만 원 출금
2. B 계좌에 10만 원 입금

이 두 단계가 하나의 트랜잭션입니다.

ACID 속성	의미	예시에서의 적용
<b>Atomicity</b>	두 단계가 모두 성공하거나, 하나라도 실패하면 전체를 되돌림 (rollback)	A의 돈이 빠져나가고 B에 안 들어가는 상황은 없음
<b>Consistency</b>	트랜잭션 후에도 “A+B의 총액은 동일”해야 함	송금 전후 총 잔액 불일치 발생 X
<b>Isolation</b>	다른 사용자가 동시에 송금하더라도, 각 송금 결과가 서로 간섭하지 않음	동시에 여러 송금이 발생해도 계산 오류 없음
<b>Durability</b>	송금 완료 후 서버가 재시작되어도 거래 내용은 그대로 보존	트랜잭션 커밋 후 데이터 손실 없음

## 3. 왜 중요한가?

- 데이터베이스의 신뢰성과 정확성을 보장하기 때문입니다.

- 특히 금융, 결제, 재고관리처럼 “데이터의 무결성(Integrity)”이 중요한 시스템에서는 필수적입니다.
  - 반면, 일부 빅데이터 플랫폼(Hadoop, S3, NoSQL 등)은 처리 속도와 확장성을 위해 ACID를 완전하게 지원하지 않는 경우가 있습니다.
- 

## 4. 확장 맥락 — 데이터 레이크와 **ACID**

- 전통적인 Hadoop 기반 데이터 레이크(HDFS)는 ACID를 지원하지 않아, 중간에 작업이 실패하면 데이터가 손상되거나 중복될 수 있었습니다.
- 이에 반해 **Delta Lake**, **Apache Iceberg**, **Apache Hudi** 같은 최신 오픈 테이블 포맷은 오브젝트 스토리지 상에서도 **ACID** 트랜잭션을 지원합니다.

즉, 현대의 “데이터 레이크하우스(Lakehouse)”는 ACID를 구현해

\*\*“데이터 웨어하우스의 신뢰성과 데이터 레이크의 확장성”\*\*을 동시에 제공합니다.

---

### 요약

#### **ACID란:**

데이터베이스 트랜잭션이 “정확하고, 일관되며, 고립되고, 영구적”으로 처리되도록 보장하는 원칙.

**핵심 목적:** 데이터 무결성(Data Integrity) 확보.

**현대적 의미:** Delta Lake나 Iceberg 같은 최신 데이터 플랫폼에서도  
오브젝트 스토리지 위에 ACID를 적용하여 안정성과 신뢰성을 확보하고 있습니다.

# 데이터 웨어하우스와 데이터레이크의 차이

## 1. 개념 비교

구분	데이터 웨어하우스 ( <b>Data Warehouse</b> )	데이터 레이크 ( <b>Data Lake</b> )
정의	정제되고 구조화된 데이터를 저장하고 분석하는 시스템	원본(raw) 상태의 모든 형태의 데이터를 저장하는 중앙 저장소
데이터 형태	정형 데이터 (예: 매출, 거래, 고객 정보 등)	정형, 반정형, 비정형 데이터 (예: 로그, 이미지, 음성, 센서 데이터 등)
데이터 저장 방식	ETL(Extract-Transform-Load): 저장 전 변환 및 정제	ELT(Extract-Load-Transform): 원본 데이터 저장 후 필요 시 변환
스키마 방식	Schema-on-Write (저장 시 스키마 정의)	Schema-on-Read (분석 시 스키마 적용)
주요 사용자	비즈니스 애널리스트, 경영진 (BI 중심)	데이터 엔지니어, 데이터 사이언티스트 (AI/ML 중심)
처리 성능	빠른 쿼리 처리 및 요약 분석에 최적화	대용량 배치 및 실시간 데이터 처리에 강점
비용 구조	고성능 DB 중심으로 상대적으로 비용이 높음	클라우드 오브젝트 스토리지 기반으로 비용 효율적
데이터 품질	품질과 신뢰성이 높음	품질 관리 미흡 시 Data Swamp 위험 존재
대표 기술	Teradata, Snowflake, BigQuery, Redshift 등	Hadoop, Spark, S3, Azure Data Lake, Delta Lake 등

## 2. 용도 비교

활용 목적	데이터 웨어하우스	데이터 레이크
비즈니스 인텔리전스(BI)	KPI 분석, 대시보드, 보고서 작성에 적합	지원 가능하지만 비효율적
정형 데이터 리포트	사전 정의된 질의 중심, 최적	비정형 데이터 중심이므로 부적합
머신러닝 및 AI 분석	데이터 제한으로 한계 있음	AI 학습용 대규모 데이터 처리에 최적
실시간 분석	제한적	스트리밍 데이터 분석에 강점
데이터 탐색 및 실험적 분석	구조적 제약으로 비효율적	자유로운 탐색과 실험적 분석에 적합

## 3. 핵심 요약

- 데이터 웨어하우스는 “정제된 데이터를 기반으로 안정적인 분석과 의사결정을 지원하는 시스템”이다.
- 데이터 레이크는 “정형, 반정형, 비정형 데이터를 모두 저장하고 AI 및 머신러닝 분석까지 확장 가능한 유연한 플랫폼”이다.

## 4. 최근 발전 방향: 데이터 레이크하우스 (**Data Lakehouse**)

최근에는 두 구조의 장점을 결합한 데이터 레이크하우스(**Data Lakehouse**) 개념이 등장했습니다.

이는 웨어하우스의 정확성·거버넌스와 레이크의 유연성·확장성을 동시에 제공하는 통합형 데이터 플랫폼입니다.

대표적인 예가 **Databricks Lakehouse**로, 모든 데이터 유형을 하나의 오픈 포맷에서 관리하며 BI, 실시간 분석, AI 워크로드를 모두 지원합니다.

정리하자면,

데이터 웨어하우스는 정형 데이터 기반의 비즈니스 분석 중심 플랫폼,

데이터 레이크는 다양한 원시 데이터를 저장하고 **AI/ML** 분석까지 가능한 확장형 플랫폼,

그리고 데이터 레이크하우스는 이 둘을 결합한 차세대 통합 데이터 아키텍처입니다.

# 데이터 레이크 & 데이터 웨어하우스 활용

데이터 웨어하우스와 데이터 레이크를 같이 사용한다면, 먼저 모든 원본 소스를 저장하기 위한 데이터 레이크를 만들고, 데이터 레이크 안의 데이터를 정제하여 데이터 웨어하우스를 만드는 것이 순서인가요?

정확하게 이해하셨습니다.

일반적으로 \*\*데이터 레이크(Data Lake)\*\*와 \*\*데이터 웨어하우스(Data Warehouse)\*\*를 함께 사용하는 경우, 말씀하신 순서 — “데이터 레이크 → 데이터 웨어하우스” — 가 올바른 접근입니다.

아래에 그 이유와 실제 데이터 파이프라인의 흐름을 단계별로 설명드리겠습니다.

## 1. 전체 흐름 개요

데이터 플랫폼의 기본적인 흐름은 다음과 같습니다:

데이터 수집 → 저장(**Data Lake**) → 정제/가공 → 분석(**Data Warehouse**) → 활용(AI, BI 등)

이 구조를 흔히 \*\*“Lake to Warehouse Architecture”\*\*라고 부릅니다.

## 2. 단계별 구성

### (1) 데이터 레이크 (**Data Lake**) – “원본 저장소”

- 역할:

모든 형태의 원시(raw) 데이터를 저장하는 중앙 저장소입니다.

로그, 이미지, IoT 데이터, 트랜잭션, 외부 API 데이터 등 다양한 소스가 들어옵니다.

- 특징:

- 정형, 반정형, 비정형 데이터를 모두 수용
- 빠른 적재(ELT 구조) 가능
- 필요할 때 스키마를 적용 (Schema-on-Read)

- 예시 기술:

- Hadoop HDFS, Amazon S3, Azure Data Lake, Google Cloud Storage

### (2) 데이터 정제 및 처리 (**Data Transformation**)

- 역할:

데이터 레이크에 쓰인 데이터를 분석 가능한 형태로 가공하는 과정입니다.

품질이 낮은 데이터, 중복 데이터, 불완전한 데이터를 정리하고 스키마를 적용합니다.

- 처리 기술:

Spark, Databricks Delta Live Tables, Airflow, Glue 등

---

### (3) 데이터 웨어하우스 (**Data Warehouse**) – “분석용 저장소”

- 역할:

정제된 데이터를 기반으로 분석, 보고, 의사결정 지원을 담당합니다.

주로 BI 대시보드, 리포팅, 집계 분석에 사용됩니다.

- 특징:

- 구조화된 데이터 중심 (Schema-on-Write)
- 트랜잭션 일관성 보장
- SQL 기반 분석에 최적화

- 예시 기술:

- Snowflake, BigQuery, Redshift, Databricks SQL Warehouse

---

## 3. 통합 사용 시의 장점

구분	데이터 레이크	데이터 웨어하우스
저장 목적	모든 원본 데이터를 저장	분석 및 비즈니스 인사이트 제공
유형	비정형·반정형 중심	정형 중심
유연성	매우 높음 (AI/ML 활용 가능)	낮음 (정제된 데이터 중심)
거버넌스/품질	낮음 (추가 관리 필요)	높음 (데이터 품질 관리됨)
비용 효율성	높음	상대적으로 낮음

데이터 레이크에서 데이터를 전처리하고 필요한 데이터만 웨어하우스로 옮기면,

비용과 성능을 모두 최적화할 수 있습니다.

## 4. Databricks Lakehouse 접근 방식

Databricks의 **Lakehouse Architecture**는 위 두 단계를 하나로 통합한 형태입니다.

즉, 데이터를 **Data Lake**처럼 유연하게 저장하면서도,

**Warehouse**처럼 신뢰성과 성능을 보장하는 구조입니다.

- 원시 데이터는 Delta Lake 포맷으로 저장
  - 정제 및 분석은 Databricks SQL, Delta Live Tables 등에서 수행
  - 별도의 데이터 이동 없이 동일한 플랫폼 내에서 처리
- 

### 결론

네, 맞습니다.

일반적인 순서는 데이터 레이크 → 데이터 정제 → 데이터 웨어하우스입니다.

그러나 Databricks Lakehouse처럼 최신 플랫폼에서는

두 단계를 통합해 데이터 저장, 정제, 분석을 한 환경에서 수행할 수 있도록 진화하고 있습니다.

# 데이터 레이크 기술 발전

“하둡(Hadoop)”은 데이터 레이크의 최신 기술이 아닙니다.

하둡은 데이터 레이크의 초기 세대를 대표하는 기술이지만,

현재는 그 한계를 극복한 클라우드 네이티브(**open format** 기반) 기술들이 주류로 자리 잡았습니다.

아래에 시대별 발전 흐름과 기술 비교를 정리하겠습니다.

## 1. 데이터 레이크 기술의 발전 단계

세대	시기	대표 기술	특징	한계
<b>1세대: Hadoop 기반 레이크</b>	2010년 대 초반	Hadoop HDFS, Hive, MapReduce	대용량 데이터 저장과 배치 처리에 강점	비효율적 자원 관리, 실시간 분석 한계, 복잡한 운영
<b>2세대: 클라우드 오브젝트 스토리지 기반 레이크</b>	2015년 이후	AWS S3, Azure Data Lake, GCS, Spark	클라우드 스토리지와 분산 처리 결합, 탄력적 확장	데이터 무결성·거버넌스 부족 (Data Swamp 문제)
<b>3세대: 오픈 포맷 기반 레이크하우스</b>	2020년 이후	<b>Delta Lake, Apache Iceberg, Apache Hudi, Databricks</b>	트랜잭션, 스키마 관리, 거버넌스 통합 지원	현재 주류 기술로 발전 중

## 2. Hadoop의 역할과 한계

\*\*하둡(Hadoop)\*\*은 2000년대 후반부터 2010년대 초반까지 데이터 레이크의 기반 기술이었습니다.

대량의 데이터를 여러 노드에 분산 저장(HDFS)하고,

MapReduce를 통해 병렬 처리하는 방식으로 빅데이터 분석을 가능하게 했습니다.

장점

- 당시로서는 대규모 데이터 처리의 혁신적인 해결책
- 비용 효율적인 분산 저장 및 배치 처리
- 오픈소스 생태계(Hive, Pig, HBase, Oozie 등)로 확장성 높음

단점

- 실시간 처리 어려움 (MapReduce는 배치 중심)
- 복잡한 운영 및 높은 관리 비용

- 클러스터 장애 및 자원 관리의 어려움
- 데이터 품질 관리와 보안 거버넌스 미흡
- 클라우드 환경과의 궁합이 낮음 (온프레미스 중심 구조)

즉, 하둡은 데이터 레이크의 “기초를 닦은 기술”이지만,

현재는 클라우드 기반 오브젝트 스토리지 + 오픈 포맷 구조로 대체되었습니다.

---

### 3. 최신 데이터 레이크 기술 트렌드

#### (1)

클라우드 네이티브 스토리지

- Amazon S3, Azure Data Lake Storage, Google Cloud Storage 등
- 저비용, 무제한 확장성, 높은 내구성

#### (2)

오픈 테이블 포맷(**Open Table Format**)

- **Delta Lake (Databricks), Apache Iceberg, Apache Hudi**
- ACID 트랜잭션 지원
- 데이터 버저닝(Time Travel), 스키마 진화(Schema Evolution) 지원
- BI와 AI를 모두 지원하는 핵심 기술

#### (3)

**Lakehouse Architecture**

- 데이터 레이크 + 웨어하우스의 통합 구조
  - 대표 예: **Databricks Lakehouse, Snowflake Unistore**
  - 실시간 분석, ML, BI 통합 가능
- 

### 4. 결론

- \*\*하둡(Hadoop)\*\*은 데이터 레이크의 초기 기반 기술로 역사적으로 중요하지만,

더 이상 “최신” 기술은 아닙니다.

- 현재는 **Delta Lake**, **Apache Iceberg**, **Apache Hudi** 같은 트랜잭션 지원형 오픈 포맷 기반 레이크하우스 기술이 최신 표준으로 자리 잡았습니다.
  - 특히 **Databricks Lakehouse Platform**은 Delta Lake를 중심으로 데이터 저장, 분석, 거버넌스, AI를 통합한 차세대 데이터 인텔리전스 플랫폼으로 평가받고 있습니다.
- 

요약하자면,

Hadoop은 “데이터 레이크의 시작점”이었고,  
Delta Lake·Iceberg·Hudi는 “데이터 레이크의 현재이자 미래”입니다.

# 하둡의 한계

하둡(Hadoop)은 과거에는 빅데이터 처리의 핵심 기술이었지만, 오늘날에는 최신 기술로 간주되지 않습니다.

그 이유는 기술적 구조적 한계와 운영상의 비효율성 때문입니다.

아래에 하둡이 기술적으로 더 이상 최신 기술이 아닌 이유를 세부적으로 설명하겠습니다.

---

## 1.

### 하둡의 기본 구조

하둡은 2000년대 후반 빅데이터 시대의 개척자 기술입니다.

핵심 구성은 다음 세 가지입니다.

- **HDFS (Hadoop Distributed File System)**: 데이터를 여러 서버에 분산 저장
- **MapReduce**: 데이터를 병렬로 처리하는 분산 컴퓨팅 프레임워크
- **YARN**: 자원 관리 및 작업 스케줄링 담당

이 구조는 당시 대량의 데이터를 값싸게 저장하고 처리하는 데 매우 혁신적이었지만, 현재의 데이터 환경에는 여러 한계가 드러납니다.

---

## 2.

### 하둡이 최신 기술이 아닌 이유

#### (1)

##### 배치 중심 처리 (**Batch-only Architecture**)

- 하둡의 핵심 엔진인 **MapReduce**는 실시간 처리를 지원하지 않습니다.
- 한 번의 작업(job)이 끝나야 다음 작업이 시작되는 순차 처리 구조로 되어 있습니다.
- 오늘날 데이터 분석은 실시간(**Streaming**)과 인터랙티브 쿼리를 요구하지만, 하둡은 이에 부적합합니다.

예: Kafka, Spark Structured Streaming, Flink 같은 실시간 처리 기술이 하둡을 대체함.

---

#### (2)

## 운영 복잡성 및 유지비용 증가

- 하둡은 온프레미스 서버 중심으로 설계되어 있어, 노드 추가·장애 대응·클러스터 관리가 어렵습니다.
  - 관리자가 직접 자원(YARN, HDFS 노드)을 튜닝해야 하고, 장애 처리도 수동으로 해야 합니다.
  - 확장성은 뛰어나지만, 운영 복잡도와 유지보수 비용이 급격히 증가합니다.
  - 반면, 최신 클라우드 기술은 서비스(**Serverless**) 및 오토스케일링으로 자동 관리가 가능합니다.
- 

## (3)

### 비효율적인 자원 활용

- 하둡은 컴퓨팅과 스토리지가 결합된 구조입니다.
  - 데이터를 저장하려면 반드시 동일 노드에 컴퓨팅 자원(CPU, 메모리)도 할당되어야 합니다.
  - 데이터 저장과 처리가 분리되지 않아 비용 비효율적이며, 클라우드 시대의 탄력적 확장에 불리합니다.
  - 최신 기술은 **Compute-Storage** 분리형 구조를 사용해 필요할 때만 자원을 할당합니다.  
(예: Databricks, Snowflake, BigQuery)
- 

## (4)

### 데이터 품질과 거버넌스 관리 한계

- HDFS 기반 시스템은 메타데이터 관리, 보안, 접근 제어, 데이터 계보(**Lineage**) 가 분산되어 있습니다.
  - 데이터의 신뢰성(ACID 트랜잭션) 보장 기능이 없어서 데이터 정합성(**Data Consistency**) 문제가 자주 발생합니다.
  - 반면, 최신 오픈 포맷(Delta Lake, Iceberg, Hudi)은 ACID 트랜잭션, 스키마 진화, Time Travel 등을 지원하여 훨씬 안정적입니다.
- 

## (5)

### AI 및 머신러닝 환경과의 통합 어려움

- 하둡은 기본적으로 분석용 시스템(**OLAP**) 이 아니라 저장 및 배치 처리용 시스템입니다.
- AI나 ML 모델 학습을 위한 인터랙티브 환경(Jupyter, Notebook, MLflow 등)과의 통합이 어렵습니다.

- 최신 플랫폼(Databricks, Google Vertex AI 등)은 데이터 처리와 ML/AI 워크플로우를 한 환경에서 수행할 수 있습니다.
- 

### 3. 대체 및 진화 기술

기능 영역	하둡 대체 기술
데이터 저장	Amazon S3, Azure Data Lake, Google Cloud Storage
분산 처리	Apache Spark, Databricks, Flink, Presto/Trino
테이블 포맷	Delta Lake, Apache Iceberg, Apache Hudi
워크플로우 관리	Databricks LakeFlow, Airflow, dbt
거버넌스/보안	Unity Catalog, Apache Ranger, AWS Lake Formation

### 4. 결론

하둡은 빅데이터 시대의 혁신을 이끈 기반 기술이었지만,

현재는 운영 복잡성, 실시간 처리 한계, 클라우드 비효율성으로 인해

더 이상 현대적인 데이터 인프라 요구사항을 충족하지 못합니다.

오늘날의 표준은 다음과 같습니다.

- Compute-Storage 분리 구조
- 오픈 포맷 기반의 **Lakehouse** 아키텍처 (**Delta Lake, Iceberg, Hudi**)
- Serverless & Auto-scaling** 환경
- 통합 거버넌스 및 **AI** 연계성 지원

즉, 하둡은 여전히 일부 온프레미스 환경에서 사용되지만,

데이터 플랫폼의 방향은 이미 **Databricks**와 같은 클라우드 네이티브, 오픈 포맷 기반 통합 플랫폼으로 이동했습니다.

# 하둡 비정형 데이터 저장

데이터 레이크의 경우, 비정형 데이터 (예: 로그, 이미지, 음성, 센서 데이터 등)도 저장이 가능하다고 하였습니다.

하둡도 이미지, 음성 데이터도 저장 가능한가요?

아주 좋은 질문입니다.

결론부터 말씀드리면, 하둡(**Hadoop**)도 이미지나 음성 같은 비정형 데이터 저장은 가능합니다.

하지만 “저장 가능하다”와 “효율적으로 처리·활용할 수 있다”는 완전히 다른 문제입니다.

하둡은 비정형 데이터를 단순히 파일 형태로 저장하는 것은 가능하지만,

이를 분석·활용하기 위한 효율적인 지원이 부족합니다.

아래에 이유를 단계별로 설명드리겠습니다.

---

## 1. 하둡이 비정형 데이터를 저장할 수 있는 이유

- 하둡의 저장소인 **HDFS (Hadoop Distributed File System)** 은  
파일 단위로 데이터를 분산 저장하는 시스템입니다.
- 즉, 텍스트, 로그, 이미지, 음성, 영상 등 파일 형태로 존재하는 데이터라면 모두 저장이 가능합니다.
- 예를 들어 다음과 같은 형태의 데이터 저장이 가능합니다:
  - 웹 서버 로그 파일 (txt, csv, json)
  - 이미지 (jpg, png 등)
  - 음성 (mp3, wav 등)
  - IoT 센서 데이터 (json, avro 등)

따라서 “물리적으로 저장 가능하다”는 측면에서는 하둡이 충분히 대응할 수 있습니다.

---

## 2. 하지만 하둡은 비정형 데이터 “활용”에는 부적합함

하둡은 저장만 가능할 뿐, 비정형 데이터를 구조화하여 분석하거나 처리하는 기능은 매우 제한적입니다.

그 이유는 다음과 같습니다.

### (1) HDFS는 단순 파일 저장소일 뿐, 콘텐츠 이해 기능이 없음

- HDFS는 파일 단위를 블록으로 쪼개 저장하지만,

파일 내부의 데이터(예: 이미지의 픽셀, 음성의 주파수 정보)는 “내용”으로 인식하지 못합니다.

- 즉, 하둡은 파일을 저장하고 읽는 데는 강하지만,  
파일 내용을 기반으로 분석하거나 특징(feature)을 추출하는 기능은 없습니다.

예:

- “이 이미지에서 사람 얼굴을 찾아라”
  - “이 음성에서 특정 단어를 인식하라”
- 이런 작업은 하둡 자체로는 불가능합니다.
- 

## (2) 비정형 데이터를 분석하려면 추가 프레임워크가 필요

- 이미지, 음성 같은 데이터는 **Spark**, **TensorFlow**, **PyTorch** 같은 프레임워크를 통해 별도로 처리해야 합니다.
- 하둡은 단순한 저장소 역할만 수행하고, 분석 로직은 외부 엔진에서 실행해야 합니다.
- 이 때문에 데이터 이동과 변환 비용이 크고, 처리 파이프라인이 복잡해집니다.

예를 들어,

- HDFS에 저장된 이미지를 Spark에서 로드하고
  - TensorFlow로 전송하여 AI 모델을 학습시키는 식으로 복잡한 연결이 필요합니다.
- 

## (3) 데이터 검색·거버넌스 기능의 부재

- 하둡은 데이터 카탈로그나 메타데이터 관리 기능이 매우 약합니다.
  - 이미지나 음성 데이터를 태그나 속성 기반으로 검색하거나  
접근 제어를 세밀하게 설정하는 기능이 없습니다.
  - 최신 데이터 레이크(Lakehouse) 시스템은 메타데이터, 카탈로그, 보안 정책을 통합 관리합니다.  
(예: Databricks Unity Catalog, AWS Glue Data Catalog 등)
- 

## 3. 비교: 하둡 **vs** 최신 데이터 레이크(**Lakehouse**)

구분	하둡 ( <b>HDFS</b> 기반)	최신 데이터 레이크 / 레이크하우스
데이터 저장	가능 (파일 단위 저장)	가능 (파일 + 오브젝트 단위 관리)
데이터 형식 지원	정형/비정형 모두 가능	정형/비정형 + 반정형 (JSON, Parquet 등)
비정형 데이터 처리	별도 도구 필요 (Spark, TensorFlow 등)	통합 처리 가능 (Databricks, Snowflake 등)
메타데이터 관리	약함 (수동 관리 필요)	강력함 (Unity Catalog, Glue, Atlas 등)
<b>AI/ML</b> 통합	직접 통합 어려움	내장 또는 네이티브 통합 지원
거버넌스/보안	Apache Ranger 수준	통합 거버넌스 지원 (IAM, RBAC, Audit 등)

## 4. 정리

- **하둡(HDFS)**은 이미지, 음성, 영상 등 비정형 데이터를 “파일”로 저장할 수 있습니다.
- 그러나 내용 기반의 분석, 검색, 모델 학습을 직접 수행할 수는 없습니다.
- 최신 데이터 레이크(**Lakehouse**) 기술은 저장뿐 아니라 **AI** 분석, 스키마 관리, 메타데이터 거버넌스, 접근 제어를 통합적으로 지원합니다.

따라서,

하둡은 “비정형 데이터를 저장할 수는 있지만,  
그것을 ‘이해하고 활용’할 수 있는 수준에는 이르지 못한 기술”입니다.

요약하자면,

- 저장: 하둡 가능
- 처리 및 분석: 별도 도구 필요
- 최신 기술과 비교: 비효율적, 운영 복잡

→ 따라서 현재는 **Databricks, Iceberg, Delta Lake, Snowflake** 등이 비정형 데이터 관리의 주류 기술로 자리 잡고 있습니다.

# 하둡 vs RDB

하둡도 RDB 처럼 데이터 스키마 기준으로 관리하고, 스키마 기준의 데이터를 분산 저장하는 줄 알았습니다.  
아닌가요?

아주 정확한 부분을 짚으셨습니다.

많은 분들이 하둡(Hadoop)을 “대규모 분산 데이터베이스처럼” 이해하지만, 실제로는 그렇지 않습니다.

하둡은 \*\*데이터베이스가 아니라 ‘파일 시스템’\*\*입니다 — 즉, 데이터를 스키마 기반으로 관리하지 않습니다.

이 차이를 명확히 이해하시면 데이터 아키텍처의 핵심 구조를 훨씬 잘 보실 수 있습니다.

아래에 자세히 설명드리겠습니다.

---

## 1. 하둡은 “파일 시스템”, RDB는 “데이터베이스 시스템”

항목	Hadoop (HDFS 기반)	RDB (관계형 데이터베이스)
기본 구조	분산 파일 시스템 (HDFS)	테이블 기반 데이터베이스
데이터 저장 단위	파일 (block 단위로 분산 저장)	행(row)과 열(column)로 구성된 테이블
스키마 관리	없음 (Schema-free, Schema-on-read)	저장 시점에 스키마 필수 (Schema-on-write)
데이터 형식	CSV, JSON, Parquet 등 다양한 파일 형식	테이블 구조로 고정
저장 방식	단순히 파일을 여러 노드에 나누어 저장	스키마에 따라 구조적으로 저장
메타데이터 관리	외부 도구(Hive Metastore 등)가 필요	DB 엔진이 자체 관리
쿼리 지원	별도 엔진 필요 (Hive, Impala, Trino 등)	SQL 내장
트랜잭션(ACID)	기본적으로 미지원	완전 지원
데이터 일관성	eventual consistency (최종적 일관성)	strong consistency (즉시 일관성)

## 2. 핵심 차이 — “하둡은 스키마를 모른다”

RDB는 스키마(데이터 구조)를 저장 전에 반드시 정의해야 합니다.

예를 들어 고객 정보를 저장하려면 다음과 같이 정의합니다.

즉, RDB는 저장할 때부터 구조를 강제하고,

데이터가 정의된 스키마에 맞지 않으면 저장 자체가 불가능합니다.

이를 **Schema-on-write**라고 합니다.

반면, 하둡(HDFS)은 다음과 같은 특징을 가집니다.

- 스키마를 정의하지 않습니다.
- CSV, JSON, 이미지, 로그 파일 등 그 어떤 형식이든 그대로 저장합니다.
- 즉, 데이터를 “읽을 때” 스키마를 적용합니다.

이를 **Schema-on-read**라고 합니다.

---

### 3. 예시로 보는 차이

예를 들어 아래와 같은 로그 데이터를 저장한다고 해보겠습니다.

- **RDBMS:**

데이터를 저장하기 전에 테이블 구조를 정의해야 하고,

데이터가 해당 스키마에 맞지 않으면 저장 불가입니다.

→ 구조적 일관성 보장, 하지만 유연성 부족.

- **HDFS:**

스키마 없이 파일 형태로 저장합니다.

나중에 Spark, Hive, Trino 등을 통해 “이 필드를 user\_id, event, timestamp로 읽자”라고 정의합니다.

→ 유연하지만, 구조적 일관성과 품질 보장은 어렵습니다.

### 4. 하둡이 “스키마 기반처럼 보이게” 하는 기술들

하둡 생태계에는 RDB처럼 스키마를 흉내내는 상위 계층 기술이 존재합니다.

이것들이 하둡을 RDB처럼 보이게 만들어 혼동을 주곤 합니다.

도구	역할	특징
Hive	SQL 인터페이스 제공	HDFS 상의 데이터를 테이블처럼 조회 가능 (Hive Metastore에 스키마 저장)

<b>Impala / Trino / Presto</b>	대화형 SQL 쿼리 엔진	Hive의 스키마를 참조해 빠른 질의 지원
<b>HBase</b>	NoSQL 데이터베이스	HDFS 위에서 키-값 형태로 데이터 저장 (스키마 단순함)
<b>Atlas / Ranger</b>	메타데이터 및 보안 관리	스키마 및 접근 권한을 별도로 관리

즉, 하둡 자체는 스키마를 모르는 파일 시스템이지만,

**Hive** 같은 도구가 스키마를 “추상적으로” 덧씌워주는 것입니다.

이로 인해 “하둡이 RDB처럼 스키마 기반이다”라는 착각이 생길 수 있습니다.

---

## 5. 결론

요약하면 다음과 같습니다.

1. 하둡(**HDFS**)은 데이터베이스가 아니라 분산 파일 저장소이다.
  2. 하둡은 스키마 없이(**raw** 형태로) 데이터를 저장하고,  
나중에 분석 시점에 스키마를 적용한다.
  3. RDB는 스키마를 기반으로 데이터를 구조적으로 저장하고,  
저장 시점부터 일관성과 품질을 강제한다.
  4. Hive, Impala, Trino 등은 하둡 위에 “SQL형 스키마 레이어”를 덧붙이는 도구일 뿐,  
하둡 자체의 구조는 여전히 파일 단위 비구조적 저장이다.
- 

따라서 요약하면,

하둡은 데이터를 “스키마 기반으로 저장”하지 않습니다.

다만 “읽을 때 스키마를 적용”하는 방식이며,

Hive 같은 상위 도구가 이를 데이터베이스처럼 보이게 만드는 것입니다.

# 최근 기술 동향 : 하둡 대신 오브젝트스토리지

최근에는 하둡대신 오브젝트스토리지를 사용하는 추세인가요?

정확합니다.

현재 전 세계적으로 하둡(**Hadoop**) 대신 클라우드 오브젝트 스토리지(**Object Storage**)를 사용하는 것이 명확한 추세입니다.

하둡은 2010년대 초·중반까지 빅데이터 플랫폼의 핵심이었지만,

2020년대 이후 대부분의 기업은 “하둡 → 오브젝트 스토리지 기반 아키텍처”로 전환했습니다.

이 변화는 단순한 기술 교체가 아니라,

데이터 저장 구조, 운영 방식, 비용 모델, 클라우드 전략이 모두 바뀌었다는 의미입니다.

아래에서 그 이유와 구조적 차이를 자세히 설명드리겠습니다.

---

## 1. 하둡(HDFS)에서 오브젝트 스토리지로 전환되는 이유

구분	Hadoop (HDFS)	Object Storage (S3, ADLS, GCS 등)
기반 구조	온프레미스 분산 파일 시스템	클라우드 네이티브 오브젝트 저장소
스토리지 구조	파일(block) 단위로 저장	오브젝트(key-value) 단위로 저장
컴퓨팅과 스토리지	결합 (Coupled)	분리 (Decoupled)
확장성	클러스터 단위로 물리 확장 필요	사실상 무한 확장 (클라우드 기반)
비용 구조	CAPEX 중심 (서버·디스크 직접 구매)	OPEX 중심 (사용량 기반 과금)
관리 방식	직접 운영, 장애 대응, 확장 수동	완전 관리형 (서비스·자동 복구)
데이터 접근성	내부 네트워크 기반	API 기반 글로벌 접근 (HTTP/S3 API)
데이터 내구성	노드 장애 발생 시 수동 복제	자동 3중 이상 복제 (99.99999999% 내구성)
AI/ML 연계성	외부 연동 필요 (비효율적)	클라우드 AI 서비스와 직접 통합 가능

요약하자면,

하둡은 “직접 서버를 관리하면서 데이터를 저장”하는 구조이고,

오브젝트 스토리지는 “클라우드에서 자동으로 확장·복제·관리되는 저장소”입니다.

---

## 2. 전환이 일어난 기술적 배경

### (1)

#### Compute-Storage 분리(Decoupled Architecture)

- 하둡은 HDFS와 컴퓨팅(Spark, MapReduce)이 같은 노드에 결합되어 있습니다.
- 따라서 저장 용량을 늘리려면 컴퓨팅 노드도 함께 늘려야 합니다. (비용 비효율)
- 오브젝트 스토리지는 저장소와 컴퓨팅을 완전히 분리하여,  
데이터는 S3 같은 스토리지에 두고, 필요한 시점에만 Spark·Databricks·Trino 등으로 연산합니다.

→ 즉, 저장소는 항상 켜져 있지만, 컴퓨팅은 필요할 때만 사용 → 비용 절감.

---

### (2)

#### 운영 복잡성 제거

- 하둡은 노드 장애, 네임노드 이중화, 데이터 리밸런싱 등 운영 난이도가 매우 높습니다.
- 반면, 오브젝트 스토리지는 운영이 완전 자동화되어 있습니다.
  - 하드웨어 교체 없음
  - 장애 복구 자동
  - 버전 관리 및 백업 내장

→ 따라서 데이터 엔지니어가 “운영”보다 “활용”에 집중할 수 있습니다.

---

### (3)

#### 클라우드 데이터 레이크 / 레이크하우스 아키텍처 등장

- AWS S3, Azure Data Lake Storage, Google Cloud Storage는 단순 저장소를 넘어서 “데이터 레이크의 표준”이 되었습니다.
- Delta Lake, Apache Iceberg, Apache Hudi 같은 오픈 포맷이 오브젝트 스토리지 위에서 작동하면서 **ACID** 트랜잭션 + 거버넌스를 제공하게 되었죠.

즉, 오늘날의 데이터 플랫폼은

“HDFS + Hive 기반 Hadoop 시스템” → “S3 + Delta Lake 기반 Lakehouse 시스템”  
으로 완전히 전환되었습니다.

### 3. 오브젝트 스토리지를 중심으로 한 최신 아키텍처 예시

#### Databricks Lakehouse Architecture

[Object Storage: AWS S3 / ADLS / GCS]



[Delta Lake: ACID, Schema, Versioning]



[Databricks SQL / Spark / MLflow / Unity Catalog]



[BI Tools, AI/ML Models, Dashboard]

이 구조에서는:

- 데이터 저장은 **S3** 같은 오브젝트 스토리지
- 데이터 관리 및 트랜잭션은 **Delta Lake**
- 분석과 AI는 **Databricks** 엔진

이 각각 분리되어 작동합니다.

### 4. 기업들의 실제 전환 동향

기업 / 기관	과거 환경	현재 환경
Netflix	Hadoop + Hive	AWS S3 + Iceberg + Spark
Uber	Hadoop HDFS	GCS + Hudi + Presto
Airbnb	Hadoop HDFS	S3 + Databricks
카카오페이 (한국 사례)	Hadoop 기반	클라우드 레이크하우스 PoC 검토 중 (Databricks 등)
국내 금융사 다수	온프레미스 Hadoop	Private Cloud Object Storage + Spark 전환 진행 중

### 5. 결론

- 하둡(**HDFS**)은 “과거의 온프레미스 데이터 레이크 표준”이었지만, 운영 복잡성과 클라우드 호환성 한계로 인해 현재는 거의 신규 구축이 중단되었습니다.
- 오브젝트 스토리지(**S3, ADLS, GCS**)는
  - 무한 확장성

- 내구성
- 서비스 운영
- AI/ML 통합성

덕분에 현재 데이터 레이크 및 레이크하우스의 표준 저장소로 자리 잡았습니다.

- 특히 **Databricks, Snowflake, Google BigLake** 같은 플랫폼은

기본 저장소를 HDFS가 아니라 오브젝트 스토리지로 전제합니다.

# 오프젝트 스토리지 기반 데이터 플랫폼 아키텍쳐

오브젝트 스토리지도 마찬가지로 스키마가 없는 파일 저장소이고, 스키마처럼 관리하기 위해서는 하둡과 동일하게 다른 플랫폼을 사용하는 것인가요?

정확히 이해하셨습니다.

핵심만 먼저 요약하면,

오브젝트 스토리지(**Object Storage**) 자체도 하둡(HDFS)처럼 스키마를 인식하지 못하는 단순한 파일 저장소입니다.

하지만 **Delta Lake**, **Apache Iceberg**, **Apache Hudi** 같은 “테이블 포맷(Table Format)” 계층을 덧붙여 스키마 관리·버전 관리·ACID 트랜잭션 등을 지원하게 되면서 오늘날의 데이터 레이크하우스(**Lakehouse**) 개념이 완성되었습니다.

아래에서 자세히 단계별로 설명드리겠습니다.

---

## 1. 오브젝트 스토리지는 “파일 저장소”일 뿐, 스키마를 모른다

- AWS S3, Azure Data Lake Storage, Google Cloud Storage 등은 \*\*‘파일 저장소’(Object Store)\*\*입니다.
- 파일을 “객체(Object)”로 저장하며, 각 객체는
  - **Key** (파일 경로/이름)
  - **Value** (파일 내용)
  - **Metadata** (부가 속성)로 구성됩니다.

즉, 내부적으로는 단순히

```
s3://bucket-name/path/to/file1.parquet  
s3://bucket-name/path/to/image.jpg  
s3://bucket-name/path/to/event_2025_10_12.json
```

이런 식으로 “파일”을 보관하는 시스템입니다.

→ **S3**는 파일의 구조나 스키마를 전혀 인식하지 않습니다.

→ 즉, CSV 파일인지 Parquet인지, 어떤 컬럼이 있는지 “모릅니다.”

---

## 2. 하둡(HDFS)와의 공통점

항목	Hadoop (HDFS)	Object Storage (S3, ADLS, GCS 등)
역할	분산 파일 저장소	클라우드 오브젝트 저장소
스키마 관리	없음	없음
데이터 저장 단위	파일 (block 기반)	객체 (object 기반)
데이터 읽기 방식	Schema-on-Read (읽을 때 스키마 적용)	Schema-on-Read
기본 상태에서 <b>SQL</b> 질의 가능 여부	불가능	불가능

따라서 오브젝트 스토리지도 그 자체로는 “스키마 인식”이나 “데이터베이스 기능”이 없습니다.

---

### 3. 스키마처럼 관리하기 위한 계층 — “테이블 포맷(Table Format)”

이 한계를 해결하기 위해 등장한 것이 바로 오픈 테이블 포맷(**Open Table Format**) 기술입니다.

이 계층이 오브젝트 스토리지를 스키마 기반의 데이터베이스처럼 다루게 만듭니다.

기술	주요 제공자	주요 기능
<b>Delta Lake</b>	Databricks	ACID 트랜잭션, Time Travel, 스키마 관리, 데이터 버전 관리
<b>Apache Iceberg</b>	Netflix, Apple, Snowflake 등	테이블 스냅샷, 스키마 진화, 멀티엔진 지원
<b>Apache Hudi</b>	Uber	실시간 업서트/인서트, 증분 쿼리, CDC 지원

이 세 기술은 공통적으로 \*\*“데이터 파일을 테이블처럼 관리”\*\*합니다.

예를 들어 S3에 저장된 수천 개의 Parquet 파일을

단일 테이블처럼 인식하게 만들어 SQL 질의가 가능해집니다.

---

### 4. 어떻게 스키마처럼 동작하나?

이 오픈 포맷 계층은 데이터 파일 + 메타데이터 레이어(**Metadata Layer**)로 구성됩니다.

예시 (Delta Lake 구조):

```
s3://data/events/part-0000.snappy.parquet
s3://data/events/part-0001.snappy.parquet
s3://data/events/_delta_log/0000000000000000000010.json
```

- 실제 데이터: Parquet 파일들

- `_delta_log`: 테이블의 변경 이력, 스키마, 버전 정보를 저장하는 로그

즉, 오브젝트 스토리지 자체는 여전히 “스키마를 모르는 파일 저장소”이지만,

이 상위 계층이 스키마와 트랜잭션을 가상으로 구성해주는 역할을 합니다.

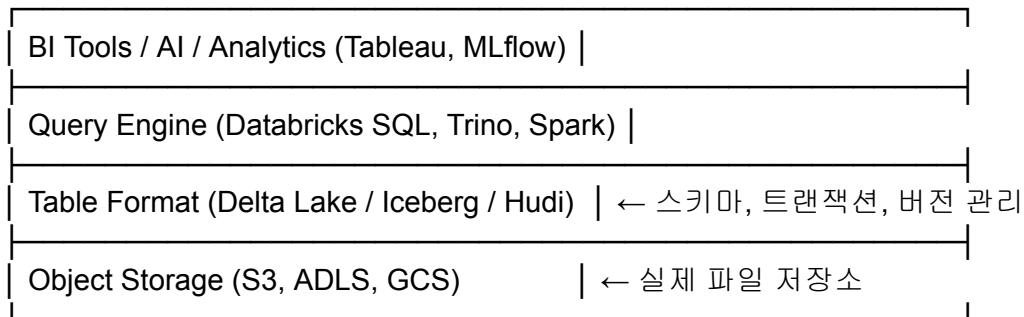
결과적으로 Spark, Databricks, Trino 같은 엔진은

이 로그를 읽어 “테이블처럼” 인식할 수 있습니다.

---

## 5. 오브젝트 스토리지 + 오픈 포맷 + 분석 엔진의 결합 구조

현대의 데이터 플랫폼은 다음과 같이 구성됩니다.



즉, 오브젝트 스토리지는

“데이터를 담는 그릇”이고,

Delta Lake나 Iceberg는 “그 데이터에 구조를 부여하는 뇌” 역할을 합니다.

---

## 6. Databricks의 예시

Databricks는 오브젝트 스토리지를 기반으로 **Delta Lake**를 기본 테이블 포맷으로 채택하고 있습니다.

- 실제 데이터는 AWS S3, Azure Data Lake, GCS 등에 저장
- Delta Lake가 메타데이터와 스키마를 관리
- Unity Catalog가 접근 제어, 데이터 계보(Lineage), 권한 관리 담당

이렇게 되면,

오브젝트 스토리지가 단순 파일 저장소가 아니라

“트랜잭션이 가능한 데이터베이스처럼 동작”하게 됩니다.

---

## 7. 결론

- 오브젝트 스토리지는 기본적으로 스키마가 없는 비구조적 파일 저장소입니다.
  - 하둡(HDFS)처럼 **Schema-on-Read** 방식으로 데이터를 관리합니다.
  - 하지만 **Delta Lake, Iceberg, Hudi** 같은 테이블 포맷 계층을 추가하면  
스키마, 트랜잭션, 버전 관리, 거버넌스를 지원할 수 있습니다.
  - Databricks, Snowflake, BigQuery 등의 현대 플랫폼은  
모두 이 오브젝트 스토리지 + 테이블 포맷 조합을 기반으로 동작합니다.
- 

요약하자면,

오브젝트 스토리지는 여전히 “스키마 없는 저장소”이지만,  
최신 데이터 플랫폼은 “테이블 포맷 계층”을 통해  
그것을 “스키마 기반의 데이터베이스처럼” 관리하고 있습니다.

즉,

하둡(**HDFS**)는 “파일 저장소 + Hive”로 스키마를 흉내냈다면,  
현대 플랫폼은 “오브젝트 스토리지 + Delta/Iceberg/Hudi”로 스키마를 완성형으로 제공합니다.

# Hadoop vs Object Storage : 이미지 활용

HDFS는 단순 파일 저장소일 뿐, 콘텐츠 이해 기능이 없다고 했는데, 오브젝트스토리지도 동일한 것 아닌가요? 이미지나 음성과 같은 콘텐츠 이해 기능 측면에서 하둡보다 오브젝트스토리지의 장점이 있나요?

아주 탁월한 질문입니다.

핵심부터 정리하면 다음과 같습니다.

네, 오브젝트 스토리지(**Object Storage**)도 하둡(HDFS)처럼 콘텐츠(예: 이미지, 음성)를 “이해”하지는 못합니다. 즉, 둘 다 파일 수준의 저장소이기 때문에, “AI처럼 내용을 해석하는 기능”은 없습니다.

하지만 오브젝트 스토리지는 하둡보다 훨씬 더 ‘콘텐츠 이해 파이프라인에 적합한 환경’을 제공합니다. 즉, ‘저장소 자체가 데이터를 이해하진 않지만’, **AI/ML** 서비스나 분석 도구들이 데이터를 이해하기 훨씬 쉽게 접근할 수 있는 구조로 되어 있습니다.

아래에서 기술적으로 왜 그런지 비교해보겠습니다.

---

## 1. 콘텐츠 이해(예: 이미지·음성 분석)의 관점에서 본 구조적 차이

항목	Hadoop (HDFS)	Object Storage (S3, ADLS, GCS 등)
기본 성격	온프레미스 분산 파일 시스템	클라우드 네이티브 오브젝트 저장소
데이터 접근 방식	로컬 네트워크 기반 (전용 API 필요)	HTTP/HTTPS 기반 글로벌 API (REST, SDK)
<b>AI/ML</b> 서비스와의 통합성	낮음 (직접 연결 어려움)	매우 높음 (클라우드 AI 서비스와 네이티브 통합)
메타데이터 관리	제한적 (파일명, 경로 정도)	확장 가능 (커스텀 메타데이터 저장 가능)
데이터 접근 제어	OS/하둡 사용자 기반 ACL	IAM, 정책 기반 세밀한 권한 제어
<b>AI</b> 모델 접근성	수동 설정 필요 (Spark, MapReduce 등 경우)	바로 접근 가능 (Python SDK, S3 API 등)
<b>AI</b> 파이프라인 연계	별도 서버 구축 필요	SageMaker, Vertex AI, Databricks 등과 직접 통합 가능

즉, 둘 다 “이미지나 음성을 해석하는 기능”은 없지만,

오브젝트 스토리지는 **AI**가 그 데이터를 가져와서 처리하기 훨씬 쉬운 구조로 되어 있습니다.

---

## 2. 기술적 이유 — 오브젝트 스토리지는 “**AI** 친화적 접근”을 지원함

(1)

## API 기반 접근 (RESTful Interface)

- 하둡(HDFS)은 내부 네트워크에서만 접근 가능한 전용 프로토콜을 사용합니다.  
예: `hdfs://namenode:8020/path/to/image.jpg`
- 반면 오브젝트 스토리지는 HTTP 기반이기 때문에  
전 세계 어디서든 AI 모델, 애플리케이션, 서비스 함수가 데이터를 직접 접근할 수 있습니다.  
예: `https://s3.amazonaws.com/bucket-name/path/to/image.jpg`

→ 즉, \*\*AI 모델이 바로 데이터를 “읽을 수 있는 구조”\*\*입니다.

---

## (2)

### 확장 가능한 메타데이터 시스템

- HDFS는 파일 이름, 크기, 수정일 정도의 메타데이터만 저장합니다.
- 오브젝트 스토리지는 사용자 정의 메타데이터(**Custom Metadata**)를 지원합니다.

예를 들어 S3에서는

```
x-amz-meta-category: face
x-amz-meta-resolution: 1920x1080
x-amz-meta-detected: true
```

이런 식으로 **AI** 분석 결과(태그, 감정 분석 결과 등)를 메타데이터로 함께 저장할 수 있습니다.

→ 즉, AI 모델이 한 번 분석한 결과를 다시 파일에 “속성으로 기록”할 수 있습니다.

→ 하둡에서는 이런 기능이 없습니다.

---

## (3)

### AI 서비스 및 데이터 파이프라인과의 네이티브 통합

- HDFS는 AI/ML 도구(PyTorch, TensorFlow, OpenAI API 등)와 직접 연결이 어렵습니다.  
Spark를 통해 중간 단계를 거쳐야 하고, 오픈소스 자원으로 한정됩니다.
- 오브젝트 스토리지는 클라우드 플랫폼의 AI 서비스와 직접 연동됩니다.

예를 들어:

- AWS: S3 ↔ SageMaker / Rekognition / Transcribe

- GCP: GCS ↔ Vertex AI / Vision API / Speech-to-Text
- Azure: ADLS ↔ Azure Cognitive Services

→ 즉, 오브젝트 스토리지는 “AI가 데이터를 바로 가져다 쓸 수 있는 표준 인터페이스”를 제공합니다.

---

#### (4)

##### 서비스/자동 확장 환경

- 하둡은 클러스터 기반으로 고정된 자원 위에서 작동합니다.  
데이터가 많아지면 클러스터 확장을 직접 해야 하고, 처리량이 제한됩니다.
  - 오브젝트 스토리지는 서비스 기반으로 AI가 요청하는 양에 맞춰 자동 확장됩니다.  
이미지 수백만 장, 음성 수천 시간 규모의 데이터도 자동 병렬 처리 가능합니다.
- 

### 3. 예시 — 이미지/음성 AI 파이프라인 비교

단계	Hadoop (HDFS)	Object Storage (S3 예시)
데이터 저장	/data/images/img001.jpg	s3://bucket/images/img001.jpg
데이터 로딩	Spark Job 작성 필요	Python SDK (boto3) 한 줄로 로드
AI 분석	Spark + TensorFlow 연결 필요	SageMaker / Vision API 바로 연결
결과 저장	HDFS에 별도 파일로 저장	동일 오브젝트에 메타데이터 추가 가능
확장성	클러스터 노드 증설 필요	자동 확장 (무제한 저장 가능)

결국, 오브젝트 스토리지는 \*\*데이터를 “이해하지는 못하지만, 이해하는 AI와 가장 잘 연결되는 구조”\*\*입니다.

---

### 4. 결론 요약

구분	HDFS	Object Storage
콘텐츠 이해( <b>semantic understanding</b> )	불가능	불가능

<b>AI 접근 용이성</b>	낮음 (로컬, 수동 설정)	높음 (API 기반 글로벌 접근)
<b>AI 서비스 통합성</b>	별도 구축 필요	클라우드 서비스와 네이티브 통합
<b>메타데이터 확장성</b>	제한적	사용자 정의 메타데이터 가능
<b>AI 파이프라인 구성 난이도</b>	높음	낮음 (직접 연동 가능)

요약하면,

하둡과 오브젝트 스토리지는 둘 다 데이터를 “이해하지는 못하지만”,  
오브젝트 스토리지는 **AI**가 데이터를 이해할 수 있는 환경을 훨씬 더 쉽게 제공합니다.

즉, 오브젝트 스토리지는 “AI 친화적인 저장소”이고,  
하둡은 “AI와 동작하기 어려운 레거시 파일 시스템”이라고 정리할 수 있습니다.

# 오브젝트 스토리지와 일반 스토리지와의 차이

아주 핵심적인 질문입니다.

“오브젝트 스토리지(Object Storage)”와 “일반 스토리지(파일 스토리지·블록 스토리지)”는 모두 데이터를 저장하는 기술이지만,

\*\*데이터를 “어떻게 저장하고, 관리하고, 접근하는가”\*\*에서 근본적인 차이가 있습니다.

쉽게 말하면,

- 블록 스토리지(**Block Storage**) → “하드디스크처럼 작동하는 저장 방식”
- 파일 스토리지(**File Storage**) → “폴더와 파일로 관리하는 방식”
- 오브젝트 스토리지(**Object Storage**) → “데이터를 ‘객체(Object)’ 단위로 관리하는 클라우드 중심 방식”

아래에서 세 가지를 구조, 용도, 기술적 차이로 비교해보겠습니다.

---

## 1. 세 가지 스토리지의 개념 비교

구분	블록 스토리지 ( <b>Block Storage</b> )	파일 스토리지 ( <b>File Storage</b> )	오브젝트 스토리지 ( <b>Object Storage</b> )
기본 단위	블록(Block) 단위로 데이터 저장	파일(File) 단위로 저장	오브젝트(Object) 단위로 저장
저장 구조	디스크의 블록 주소로 직접 접근	디렉토리/폴더 구조	평면 구조 (Flat Namespace)
식별 방법	블록 주소	경로(Path)	오브젝트 ID(Key)
메타데이터	거의 없음 (파일시스템이 관리)	파일명, 크기, 권한 등 제한적	커스텀 메타데이터를 자유롭게 저장 가능
접근 방식	OS에 직접 연결 (마운트 필요)	NFS, SMB 같은 파일 프로토콜	HTTP/HTTPS API (REST 기반)
트랜잭션 성격	빠른 I/O, 저지연	공유 환경에서 적당한 성능	높은 확장성, 지연 허용
확장성	제한적 (서버·디스크 증설 필요)	제한적 (파일 시스템 용량 한계)	사실상 무한 (클라우드 기반 확장)
주요 활용 사례	데이터베이스, VM 스토리지	NAS, 파일 공유 서버	백업, 데이터 레이크, 미디어 저장, AI 학습 데이터
대표 기술 / 서비스	Amazon EBS, Azure Disk, SAN	NFS, SMB, Amazon EFS	Amazon S3, Azure Blob, Google Cloud Storage

## 2. 구조적 차이 — “데이터를 보는 관점”이 다름

### (1)

#### 파일 스토리지 (File Storage)

- 우리가 일상적으로 쓰는 컴퓨터 폴더 구조와 동일합니다.

예:

/home/data/

```
└── image1.jpg  
└── image2.jpg  
└── report.csv
```

- 장점: 사용자 친숙, 공유 폴더로 쉽게 사용 가능
  - 단점: 수백만 개 이상의 파일이 쌓이면 성능 급격히 저하됨
- 

### (2)

#### 블록 스토리지 (Block Storage)

- 데이터를 “파일”이 아니라 “고정 크기의 블록”으로 쪼개서 저장합니다.
  - 운영체제가 파일 시스템을 입히면 하드디스크처럼 인식됩니다.
  - 장점: 매우 빠르고 정밀한 I/O 가능 → 데이터베이스, 트랜잭션 시스템에 최적
  - 단점: 구조적으로 확장성이 낮고, 서버에 직접 붙여야 함
- 

### (3)

#### 오브젝트 스토리지 (Object Storage)

- 데이터를 “오브젝트(Object)” 단위로 저장합니다.
- 각 오브젝트는 ① 실제 데이터, ② 메타데이터, ③ 고유 식별자(Key)로 구성됩니다.

예:

```
{  
  "key": "images/2025/photo001.jpg",  
  "data": (binary file),  
  "metadata": {  
    "author": "haeyoung.kim",  
    "category": "profile",  
  }  
}
```

```
        "created_at": "2025-10-12"  
    }  
}
```

- 장점:

- 수십억 개 객체도 관리 가능 (확장성 거의 무한)
- HTTP 기반 API 접근 (전 세계 어디서나 사용 가능)
- 커스텀 메타데이터 저장 가능 → AI, 검색, 분류에 유리

- 단점:

- 파일 시스템처럼 폴더 탐색은 불가능
- 블록 스토리지처럼 초기지연 I/O에는 부적합

---

### 3. 오브젝트 스토리지가 각광받는 이유

#### (1)

##### 확장성 (Scalability)

- 블록/파일 스토리지는 서버 용량이 꽉 차면 물리적으로 디스크를 추가해야 함.
- 오브젝트 스토리지는 클라우드 기반으로 자동 확장 → 사실상 무한 저장 가능.

#### (2)

##### 비용 효율성

- 사용한 용량만큼 과금 (Pay-as-you-go)
- 자동 충복제거, 다중 리전 복제 지원

#### (3)

##### API 중심 설계

- HTTP 기반 API로 어디서나 접근 가능
- AI·데이터 레이크·분석 플랫폼이 직접 연동 가능 (예: Databricks, Snowflake, Vertex AI)

#### (4)

##### 메타데이터 활용성

- 단순 파일 이름 외에 추가 정보를 저장할 수 있어

“데이터 관리 자동화”나 “AI 학습 데이터 관리”에 매우 유용

---

## 4. 실제 사용 예시 비교

시나리오	적합한 스토리지	이유
데이터베이스 서버	블록 스토리지	빠른 읽기/쓰기, 저지연 필요
회사 내 파일 공유	파일 스토리지	사용자 친화적, 권한 기반 접근
AI 모델 학습용 이미지/로그 저장	오브젝트 스토리지	대규모 비정형 데이터, 메타데이터 관리 유리
백업 / 장기 보관	오브젝트 스토리지	내구성 높고 비용 저렴
고성능 연산(OLTP)	블록 스토리지	디스크 I/O 최적화
글로벌 데이터 접근	오브젝트 스토리지	HTTP API 기반, 리전 간 접근 가능

## 5. 결론 요약

비교 항목	블록 스토리지	파일 스토리지	오브젝트 스토리지
단위	블록	파일	오브젝트(데이터+메타데이터)
구조	로우레벨(디스크 기반)	폴더 계층	평면 구조 (Flat Namespace)
확장성	제한적	제한적	매우 높음
프로토콜	iSCSI, FC	NFS, SMB	HTTP, REST
AI/클라우드 통합성	낮음	중간	매우 높음
주요 용도	DB, VM, OLTP	NAS, 공유폴더	백업, 데이터 레이크, AI, 분석
대표 기술	EBS, SAN	NFS, EFS	S3, GCS, ADLS

### 한 줄 요약

오브젝트 스토리지는 “파일이나 블록보다 훨씬 유연하고 확장 가능한, 클라우드 시대의 데이터 저장 표준”입니다.

즉, “하드디스크처럼 빠르진 않지만, 무한히 넓고, AI와 데이터 분석이 바로 연결되는 저장소”라고 이해하시면 됩니다.

# 오브젝트스토리지 딕토리 개념

오브젝트 스토리지가 평면 구조라고 했는데, 폴더 관리가 안되는 건가요?

좋은 질문입니다.

정확히 말씀드리면, 오브젝트 스토리지는 “평면 구조(**Flat Namespace**)”이지만, 폴더처럼 보이는 구조를 ‘가상적으로’ 제공합니다.

즉, 실제 폴더(디렉토리)가 존재하는 것은 아니지만, 사용자가 이해하기 쉽게 ‘폴더처럼 보이도록’ 경로 규칙을 흉내내는 구조입니다.

아래에서 구체적으로 설명드리겠습니다.

---

## 1. “평면 구조”的 의미

- 평면 구조(**Flat Namespace**)라는 말은,  
    오브젝트 스토리지에는 실제 “폴더” 계층이 없고,  
    모든 오브젝트가 하나의 공간에 존재한다는 뜻입니다.

예를 들어 AWS S3에 다음과 같은 객체가 있다고 가정해봅시다.

```
photos/2025/january/image1.jpg  
photos/2025/february/image2.jpg  
docs/report.pdf
```

실제로 S3 내부에는 다음과 같은 “트리 구조”가 없습니다:

```
photos/  
└── 2025/  
    ├── january/  
    └── february/  
docs/  
└── report.pdf
```

실제로는 이 모든 객체가 단일 버킷(**bucket**) 안에 “이름(Key)”으로만 존재합니다.

즉, 다음처럼 저장된 것입니다:

```
Key: photos/2025/january/image1.jpg  
Key: photos/2025/february/image2.jpg  
Key: docs/report.pdf
```

“슬래시(/)”는 그저 문자열일 뿐이고,

오브젝트 스토리지는 그것을 폴더처럼 해석하지 않습니다.

---

## 2. 그럼 왜 폴더처럼 보이나?

- AWS S3, Azure Blob, GCS 등 대부분의 콘솔 UI가 사용자 편의를 위해 ‘접두사(**prefix**)’를 폴더처럼 표시하기 때문입니다.
- 예를 들어 photos/2025/january/image1.jpg라는 키가 있으면, 콘솔에서는 다음과 같이 계층 구조로 보여줍니다:

photos → 2025 → january → image1.jpg 그러나 이건 **UI**가 임의로 “/” 구분자를 인식해서 트리처럼 보여주는 것일 뿐,

내부적으로는 여전히 “문자열 키” 하나로 존재합니다.

---

## 3. 오브젝트 스토리지의 “폴더 관리” 방식

- 오브젝트 스토리지는 “디렉토리 생성”이라는 개념이 없습니다.  
대신, “폴더처럼 보이는 경로”를 키의 일부로 포함시켜 사용합니다.
- 예를 들어 다음처럼 객체를 업로드하면,

Key: projectA/data/file1.csv

Key: projectA/data/file2.csv

Key: projectB/docs/manual.pdf

- 실제로는 폴더가 자동으로 생긴 것처럼 보이지만,  
내부적으로는 **Key** 이름의 접두사(**prefix**)로 구분되는 것뿐입니다.

이 때문에 오브젝트 스토리지를 다룰 때는 “디렉토리 이동”이 아니라

“prefix 기반 필터링(list with prefix)” 개념으로 데이터를 탐색합니다.

---

## 4. 실제 예시 (**AWS S3 CLI** 기준)

예시 명령어

aws s3 ls s3://my-bucket/photos/ → 내부적으로는 prefix="photos/"로 시작하는 모든 Key를 나열하는 명령입니다.

즉, “폴더를 열었다(open folder)”가 아니라

“키 접두사로 시작하는 객체를 필터링한 것”입니다.

---

## 5. 평면 구조의 장점

항목	설명
무한 확장성	디렉토리 트리처럼 깊이 제한이 없고, 수십억 개 객체도 관리 가능
탐색 속도 향상	특정 <b>prefix</b> 기반으로 빠른 검색 가능
유연성	폴더 구조를 마음대로 설계하거나 변경 가능
글로벌 접근성	HTTP API 기반으로 전 세계 어디서든 동일한 경로로 접근 가능

## 6. 요약 정리

구분	파일 스토리지 (예: <b>NFS</b> )	오브젝트 스토리지 (예: <b>S3</b> )
폴더 구조	실제 존재 (트리 구조)	존재하지 않음 (평면 구조)
식별 방식	디렉토리 + 파일명	오브젝트 키 (문자열)
폴더 생성	실제 디렉토리 필요	단순히 접두사( <b>prefix</b> ) 문자열로 표현
파일 탐색	경로 탐색	<b>prefix</b> 기반 필터링
표시 방식	계층적 탐색	UI가 “가상 폴더”처럼 표현

### 결론

- 오브젝트 스토리지는 “평면 구조”이기 때문에 진짜 폴더 개념은 없습니다.
- 하지만 키(**Key**) 이름에 "/"를 포함시켜 “가상의 폴더 구조”를 흉내낼 수 있습니다.
- 즉, 내부적으로는 평면적이지만,

사용자에게는 트리 형태로 보이는 “가상 계층 구조(**Virtual Hierarchy**)”를 제공합니다.

한 줄로 요약하면,

오브젝트 스토리지는 \*\*“폴더처럼 보이지만, 실제로는 없는 폴더를 흉내내는 평면 저장소”\*\*입니다.

# CDC 란?

CDC(Change Data Capture)는 데이터베이스에서 데이터가 변경(삽입, 수정, 삭제)되는 내용을 실시간으로 감지하고 추적하여, 그 변경 내역을 다른 시스템이나 애플리케이션으로 전송하는 기술 또는 디자인 패턴의 집합입니다.

CDC는 단순히 주기적으로 데이터를 확인하는 기존 방식과 달리, 데이터베이스의 트랜잭션 무결성을 유지하면서 효율적으로 데이터 동기화 및 통합을 달성하는 핵심 기술입니다.

## 1. CDC의 목적 및 중요성

CDC의 주요 목적은 다음과 같습니다.

목적	설명
실시간 데이터 동기화	운영 데이터베이스(OLTP)의 변경 사항을 지역 없이 분석 데이터베이스(DW)나 다른 시스템에 반영하여 최신 데이터를 유지합니다.
운영 시스템 부하 감소	전체 데이터를 주기적으로 추출하는 대신, 변경된 데이터만 추출하여 원본 데이터베이스의 부하를 최소화합니다.
이벤트 기반 아키텍처 지원	데이터 변경 자체를 하나의 이벤트로 간주하고, 이를 메시징 시스템(예: Kafka)으로 발행하여 MSA(Microservice Architecture) 간의 느슨한 결합 및 데이터 일관성을 유지하는 데 사용됩니다.

## 2. CDC의 주요 구현 방식

CDC를 구현하는 방식은 다양하지만, 가장 효율적이고 널리 사용되는 방법은 로그 기반 방식입니다.

구현 방식	작동 원리	특징
로그 기반  <b>(Log-Based)</b>	데이터베이스의 트랜잭션 로그 (MySQL의 Binlog, PostgreSQL의 WAL 등)를 직접 읽어 변경 사항을 추출합니다.	가장 선호됨. DB에 부하를 주지 않고, 트랜잭션의 순서와 무결성을 보장하며 실시간성이 가장 높습니다.
트리거 기반 <b>(Trigger-Based)</b>	DB 테이블에 트리거를 설정하여 데이터 변경이 발생할 때마다 별도의 테이블에 변경 이력을 기록합니다.	구현이 쉽지만, 운영 DB에 상당한 부하를 줄 수 있으며, 트랜잭션 무결성 보장이 어려울 수 있습니다.
쿼리 기반 <b>(Query-Based)</b>	특정 타임스탬프 또는 버전 커럼을 주기적으로 조회(Polling)하여 변경된 레코드를 식별합니다.	구현이 간단하지만, 실시간성이 떨어지고 조회 주기마다 DB 부하가 발생합니다.

### 3. CDC와 ETL/ELT의 차이점

CDC는 데이터 통합의 한 방식이며, 전통적인 ETL/ELT와는 접근 방식에 차이가 있습니다.

구분	CDC (Change Data Capture)	ETL/ELT (Extract, Transform, Load)
데이터 추출 범위	변경된 데이터만 추출	전체 또는 대량의 배치 데이터 추출
주기	실시간 또는 준실시간 (스트리밍)	정기적인 배치 (일별, 시간별)
목표	낮은 지연 시간으로 데이터 동기화 및 이벤트 발생	대량 데이터 처리 및 복잡한 변환(정제, 통합)
사용 시나리오	실시간 데이터 분석, MSA 간 데이터 동기화, 재해 복구	정기적인 보고서, 데이터 웨어하우스 구축, 복잡한 비즈니스 로직 적용

## ODS 란?

ODS (Operational Data Store)는 우리말로 운영 데이터 스토어라고 하며, 데이터 웨어하우스(DW)와 운영 시스템(OLTP) 사이에 위치하여 단기적인 운영 보고 및 의사 결정을 지원하는 데이터 저장소입니다.

단순히 운영 시스템의 데이터를 복사해 놓는 것을 넘어, 여러 운영 시스템의 데이터를 통합하고 정제하는 중간 단계의 역할을 수행합니다.

### ODS의 주요 특징 및 역할

특징	설명
운영 데이터 통합	회사 내 여러 운영 시스템(ERP, CRM, SCM 등)에서 발생하는 데이터를 하나의 통합된 공간으로 모읍니다.
실시간/준실시간 데이터	데이터 웨어하우스처럼 장기간의 이력을 저장하기보다는, 현재 시점 또는 실시간에 가까운 최신 데이터를 보관하여 즉각적인 운영 상황 파악을 지원합니다.
단기적 의사 결정 지원	주로 '오늘' 또는 '최근'의 운영 현황에 대한 보고서나 쿼리(예: "현재 재고 수량은?", "오늘 발생한 주문 건수는?")를 지원하는데 사용됩니다.
DW의 데이터 소스 역할	ODS에서 정제되고 통합된 데이터는 최종적으로 데이터 웨어하우스(DW)로 옮겨져 장기적인 분석을 위한 이력 데이터로 활용됩니다. ODS는 DW 구축을 위한 중간 가공 및 스테이징(Staging) 영역 역할도 수행합니다.
휘발성 (Volatile)	데이터 웨어하우스와 달리 ODS의 데이터는 지속적으로 갱신되거나 최신 데이터로 대체될 수 있습니다 (일정 기간의 데이터만 보관).

구분	ODS (운영 데이터 스토어)	DW (데이터 웨어하우스)
데이터 범위	현재 시점 또는 비교적 최신 운영 데이터	장기간에 걸친 이력 데이터 및 요약 데이터
사용 목적	단기적 운영 통제, 실시간 근접 보고	장기적 전략 분석, 트렌드 파악, BI
데이터 특성	데이터가 지속적으로 갱신됨 (휘발성)	데이터가 한 번 적재되면 갱신되지 않음 (비휘발성)
세부 수준	원본에 가까운 원자성( <b>Atomic</b> ) 데이터	다양한 수준의 상세 데이터와 집계된 요약 데이터

요약하자면, ODS는 기업의 일상적인 운영 활동을 지원하기 위해 여러 시스템의 데이터를 신속하게 통합하여 현재의 운영 상황을 반영하는 "임시 작업 공간이자 최신 데이터 허브" 역할을 합니다. 이후 이 데이터는 보다 깊이 있는 분석을 위해 데이터 웨어하우스로 전달됩니다.