December 10, 2025

▣ **강의명:** CSCI E-103: 재현 가능한 머신러닝

▣ **주차:** Lecture 04

▣ **교수명:** Anindita Mahapatra

Eric Gieseke

▣ **목적:** Lecture 04의 핵심 개념 학습

Contents

1 개요 (Overview)

이번 4강에서는 데이터 엔지니어링의 핵심 작업인 '데이터 변환'에 대해 배웁니다.

데이터 파이프라인을 설계할 때 자주 사용되는 검증된 해결책인 **디자인 패턴**을 학습합니다. 또한, 데이터를 다룰 때 반드시 지켜야 할 법적 규제(GDPR, CCPA)인 **컴플라이언스**의 중요성을 이해합니다.

Spark의 내부 동작 원리인 **파티션, Z-Ordering, 조인 전략**을 살펴보고, **CRUD, CDC, SCD**와 같은 데이터 변경 및 이력 관리 기법을 배웁니다. 마지막으로, **Autoloader**를 사용해 클라우드 스토리지의 파일을 증분 처리하는 실습을 진행합니다.

□ 핵심 요약

이번 세션의 핵심 목표

- 데이터 아키텍처의 주요 디자인 패턴(Lambda, Kappa, Data Mesh 등)을 이해합니다.
- GDPR과 같은 데이터 규제가 엔지니어링에 미치는 영향을 파악합니다.
- Spark의 성능 최적화 핵심인 파티션(테이블 vs Spark)과 조인 전략을 구분합니다.
- CDC(변경 데이터 캡처)와 SCD(느리게 변하는 차원)의 차이와 구현(Type 1, 2)을 이해합니다.
- Autoloader를 사용하여 스키마 진화 및 불량 데이터를 처리하는 방법을 실습합니다.

2 지난 강의 복습 (Review of Lecture 3)

데이터 변환을 배우기에 앞서, 지난 강의에서 다룬 스트리밍과 데이터 아키텍처의 핵심 용어들을 복습합니다.

- **스트리밍 유형 (2가지):**
  - **파일 기반 (File-based):** 파일이 도착하는 것을 이벤트로 감지하여 처리합니다.
  - **이벤트 기반 (Event-based):** Kafka, Pulsar 등 메시징 큐를 통해 개별 이벤트(메시지)가 발생하는 즉시 처리합니다.
- **체크포인팅 (Checkpointing):** 스트리밍 작업이 실패했을 때를 대비하여, '어디까지 처리했는지' 마지막으로 성공한 위치(오프셋)를 저장하는 것입니다. 작업이 재시작되면 이 저장된 지점부터 다시 시작하여 데이터 유실이나 중복 처리를 방지합니다.
- **워터마킹 (Watermarking):** '얼마나 늦게 도착하는 데이터까지 허용할 것인가'를 정의하는 기준선입니다. 예를 들어, 워터마크를 10분으로 설정하면, 특정 윈도우(예: 10:00 10:10)가 닫힌 후 10분(즉, 10:20)이 지나면, 10:00 10:10에 속하는 데이터가 늦게 도착하더라도 무시(삭제)합니다. 집계의 정확성과 속도 사이의 트레이드오프입니다.
- **스트리밍 출력 모드 (Streaming Output Modes):**
  - **Append (추가):** 처리된 새로운 행만 출력합니다. (워터마크 없이는 집계에 사용 불가)
  - **Update (업데이트):** 변경된 행만 출력합니다.
  - **Complete (전체):** 전체 결과 테이블을 매번 다시 계산하여 출력합니다.
- **처리 트리거 (Processing Trigger):** 마이크로배치(Micro-batch) 스트리밍에서 '얼마나 자주' 배치를 실행할지 결정하는 간격입니다. (예: 1초마다)
- **Lambda vs. Kappa 아키텍처:**

  - **Lambda (람다):** 속도가 중요한 실시간 처리(Speed Layer)와 정확도가 중요한 배치 처리(Batch Layer)를 **두 개의 별도 파이프라인**으로 운영한 뒤, 서빙 레이어에서 합치는 구조입니다. (복잡하지만 안정적)
  - **Kappa (카파): 하나의 스트리밍 파이프라인**으로 모든 것을 처리합니다. 배치가 필요하면 스트림을 처음부터 다시 읽어옵니다. (단순하지만 스트리밍 시스템에 대한 의존도가 높음)
- **데이터 레이크하우스 (Lakehouse):** 데이터 웨어하우스(정형 데이터, BI)의 장점과 데이터 레이크(원시 데이터, ML)의 장점을 결합한 현대적인 아키텍처입니다.
- **관리형 vs. 외부 테이블 (Managed vs. External):**
  - **관리형 (Managed):** Spark가 데이터와 메타데이터(스키마 정보)를 모두 관리합니다. 'DROP TABLE' 시 데이터와 메타데이터가 **모두 삭제**됩니다.
  - **외부 (External):** Spark는 메타데이터만 관리하고, 실제 데이터는 외부(예: S3)에 있습니다. 'DROP TABLE' 시 메타데이터만 삭제되고 원본 데이터는 **유지**됩니다.
- **메타데이터 (Metadata):** 데이터에 대한 데이터(스키마, 소유자, 위치 등)입니다. 메타데이터가 잘 관리되면 데이터 검색 가능성(Discoverability)과 데이터 거버넌스(Governance, 통제 및 관리)가 향상됩니다.
- **구체화된 뷰 (Materialized View):** 일반적인 뷰(View)는 쿼리 시점에 계산되는 '가상 테이블'이지만, 구체화된 뷰는 쿼리 결과를 '미리 계산하여 물리적으로 저장'해 둔 테이블입니다.
  - **장점:** 쿼리 속도가 매우 빠릅니다.
  - **단점:** 원본 데이터가 변경되어도 즉시 반영되지 않아 데이터가 오래될(stale) 수 있으며, 별도의 저장 공간이 필요합니다. (주기적인 새로고침(Refresh) 필요)

3 데이터 엔지니어링 디자인 패턴 (Design Patterns)

3.1 디자인 패턴이란? (What is a Design Pattern?)

디자인 패턴은 소프트웨어 엔지니어링이나 데이터 엔지니어링에서 **자주 발생하는 공통적인 문제에 대한 검증된 해결책(템플릿)**입니다.

□ 예제: title

요리를 할 때 '재료를 기름에 볶는다'(볶음), '물에 오래 끓인다'(찜) 같은 검증된 조리법이 있습니다. 데이터 엔지니어링에서도 '대용량 데이터를 실시간과 배치로 동시에 처리하는 문제'에 대해 '람다 아키텍처'라는 검증된 레시피(패턴)가 있습니다.

디자인 패턴은 다음과 같은 이점을 제공합니다.

- **좋은 설계 원칙 증진:** 추상화(Abstraction), 관심사 분리(Separation of Concerns), 문제 분할(Divide and Conquer) 등 좋은 설계 원칙을 자연스럽게 따르도록 유도합니다.
- **공통 어휘 제공:** 팀원 간에 "이 부분은 람다 패턴을 적용하죠"라고 말하면, 복잡한 설계를 매번 설명할 필요 없이 효율적으로 소통할 수 있습니다.
- **재사용성 및 안정성:** 이미 많은 사람이 사용하고 검증한 해결책이므로, 처음부터 설계를 고민하는 것보다 안정적이고 효율적입니다.

주의: 맹목적인 적용 금지

디자인 패턴은 만병통치약이 아닙니다. 내가 해결하려는 문제의 맥락(데이터 크기, 속도, 비용 등)에 맞는지 신중하게 검토하고 적용해야 합니다.

3.2 빅데이터 디자인 패턴 분류

빅데이터 환경(대용량, 고속도, 다양성)에서는 전통적인 소프트웨어 디자인 패턴(GoF 패턴)과는 다른, 데이터 처리에 특화된 패턴들이 사용됩니다.

데이터 파이프라인의 각 단계별로 적용할 수 있는 주요 디자인 패턴은 다음과 같습니다.

단계	주요 패턴	설명 (해결하려는 문제)
모델링	Star/Snowflake Schema	분석(BI)에 최적화된 데이터 웨어하우스 구조 (사실 테이블 + 차원 테이블)
(Modeling)	Vault Modeling	데이터의 모든 변경 이력을 원본 그대로 저장하는 모델링 기법
수집	Connectors	다양한 데이터 소스(DB, API, 파일)에 일관된 방식으로 연결
(Ingestion)	Lambda / Kappa	실시간 및 배치 데이터를 처리하는 아키텍처 (앞서 복습)
	Compute/Storage Separation	계산 자원과 저장 자원을 분리하여 각각 독립적으로 확장 (비용 효율화)
변환	Schema on Read / Write	스키마(데이터 구조)를 언제 정의할지 결정
(Transformation)	ACID Transactions	대용량 데이터 환경에서 원자성, 일관성, 고립성, 지속성을 보장 (예: Delta Lake)
	Multi-Hop Pipeline	데이터를 여러 단계(Bronze → Silver → Gold)로 정제 (Medallion 아키텍처)
저장	Columnar Storage	데이터를 행(Row) 단위가 아닌 열(Column) 단위로 저장 (예: Parquet). 분석 쿼리 속도 향상.
(Persist)	Denormalized Tables	정규화를 낮추고(중복 허용) 테이블을 넓게 만들어, 비싼 조인(Join) 연산을 피함.
분석	In-Stream Analytics	데이터가 저장되기 전, 스트리밍 중에 실시간으로 분석 (예: 실시간 사기 탐지)
(Analytics)		

데이터 파이프라인 단계별 디자인 패턴

3.3 주요 아키텍처 및 스토리지 패턴

- **이벤트 기반 아키텍처 (EDA, Event-Driven Architecture):** 시스템의 모든 동작이 '이벤트'(상태 변경 또는 데이터 입력)에 대한 반응으로 설계된 구조입니다.
  - **구성:** 이벤트 생산자 (Producer) → 이벤트 스트리밍 (예: Kafka) → 이벤트 소비자 (Consumer)
  - **용도:** 마이크로서비스, 실시간 전자상거래 주문 처리 등
- **CQRS (Command Query Responsibility Segregation):** '관심사 분리' 원칙의 극단적인 예로, 시스템의 명령(Write/Update)과 조회(Read/Query) 책임을 완전히 분리하는 패턴입니다.
  - **목적:** 쓰기 작업(Command)에 최적화된 모델과 읽기 작업(Query)에 최적화된 모델을 따로 설계하여, 각각 독립적으로 확장하고 성능을 극대화합니다.
  - **예시:** 쓰기는 빠르지만 조회가 느린 DB에 데이터를 쓰고, 이 데이터를 읽기에 최적화된 다른 DB(예: 검색 엔진)로 복제하여 사용자가 조회하게 만듭니다.
- **폴리글랏 퍼시스턴스 (Polyglot Persistence):** 'Polyglot'은 '여러 언어를 사용하는'이란 뜻입니다. 이 패턴은 **"망치(하나의 DB)를 들고 모든 것을 못으로 보지 말라"**는 철학입니다.
  - **목적:** 단 하나의 데이터베이스 기술로 모든 문제를 해결하려 하지 않고, 각 기능에 가장 적합한 DB

를 여러 개 조합하여 사용하는 것입니다.

  - **예시:** 트랜잭션 데이터는 RDBMS(관계형DB)에, 비정형 문서는 Document DB(예: MongoDB)에, 친구 관계망은 Graph DB(예: Neo4j)에 저장합니다.

- **데이터 레이크 패턴 (Data Lake Pattern):** 모든 데이터를(정형, 비정형) 원본(Raw) 형식 그대로 저장하는 거대한 저장소입니다.
  - **목적:** "일단 저장하고, 나중에 어떻게 쓸지 결정하자"는 접근입니다. 데이터가 어떻게 사용될지 모르는 미래의 분석이나 ML을 위해 원본을 보존합니다.
  - **특징:** 스키마 온 리드(Schema on Read) 방식을 사용합니다. (데이터를 읽는 시점에 스키마 정의)

3.4 기타 주요 처리 및 서비스 패턴

- **CAP 정리 (CAP Theorem):** 분산 시스템(여러 대의 컴퓨터가 협력하는 시스템)은 세 가지 속성(일관성, 가용성, 분할 내성) 중 최대 두 가지만 동시에 만족시킬 수 있다는 이론입니다.
  - **Consistency (일관성):** 모든 노드가 동시에 같은 데이터를 보여줌.
  - **Availability (가용성):** 모든 요청에 대해 (오래된 데이터일지라도) 항상 응답함.
  - **Partition Tolerance (분할 내성):** 노드 간 통신이 끊겨도 시스템이 계속 동작함.

  빅데이터 시스템은 대부분 P(분할 내성)를 기본으로 선택하고, C(일관성)와 A(가용성) 사이에서 타협합니다.
  - **CP 시스템 (일관성 우선):** RDBMS, HBase. 데이터가 틀릴 바에는 차라리 응답하지 않습니다.
  - **AP 시스템 (가용성 우선):** DynamoDB, Cassandra. 데이터가 조금 틀리더라도(최종적 일관성) 항상 응답합니다.
- **데이터 메시 (Data Mesh):** 최신 아키텍처 패턴으로, 중앙화된 데이터 팀(병목 현상) 대신, 각 도메인(현업 부서) 팀이 자신의 데이터를 '제품(Data as a Product)'처럼 직접 소유하고 관리/제공하는 **분산형 아키텍처**입니다.
- **멀티 홉 (Multi-Hop) / 메달리온 (Medallion) 아키텍처:** 데이터를 여러 단계에 걸쳐 점진적으로 정제하는 가장 일반적인 패턴입니다. '관심사 분리'와 '문제 분할' 원칙을 따릅니다.
  - **Bronze (Landing):** 원본(Raw) 데이터. 최소한의 변환만 거쳐 저장. (데이터 엔지니어용)
  - **Silver (Refined):** 비즈니스 로직 적용. 정제되고(Cleaned), 필터링되고, 보강된 데이터. (데이터 사이언티스트용)
  - **Gold (Aggregated):** 최종 집계 데이터. BI 대시보드나 리포트를 위한 '사용 준비 완료' 상태. (비즈니스 분석가용)

  이 구조의 장점은 파이프라인이 모듈화되어 디버깅이 쉽고, 데이터 품질은 Gold로 갈수록 높아지지만, 데이터 가용성(속도)은 Bronze가 가장 빠르다는 것입니다.
- **데이터 이동 최소화 (Minimize Data Movement):** 대용량 데이터를 네트워크로 복사하는 것은 비용과 시간이 많이 듭니다. 이를 피하기 위한 패턴입니다.
  - **Time Travel (시간 여행):** 데이터의 변경 이력을 버전별로 관리하여, 데이터를 복사하지 않고도 과거 특정 시점의 스냅샷을 조회할 수 있게 합니다. (예: Delta Lake)
  - **Zero Copy Clone (제로 카피 복제):** 실제 데이터를 복사하지 않고, 메타데이터만 복제하여 새로운 테이블을 즉시 생성합니다. 스토리지 비용 없이 테스트 환경을 만들 수 있습니다.

4 데이터 컴플라이언스 (Data Compliance)

데이터 엔지니어는 데이터를 기술적으로 처리할 뿐만 아니라, 법적/윤리적 책임을 가지고 데이터를 관리해야 합니다.

4.1 주요 규제: GDPR과 CCPA

- **GDPR (General Data Protection Regulation):** 유럽 연합(EU)의 일반 데이터 보호 규정입니다. EU 시민의 개인정보를 처리하는 전 세계 모든 기업에 적용됩니다.
- **CCPA (California Consumer Privacy Act):** 미국 캘리포니아주의 소비자 개인정보 보호법입니다.

이러한 규제들은 조직이 개인 데이터를 합법적이고 엄격한 조건 하에 수집/관리하도록 강제하며, 데이터 소유자(소비자)의 권리를 존중할 의무를 부여합니다. 위반 시 막대한 과징금이 부과될 수 있습니다.

4.2 데이터 엔지니어의 과제: 핵심 권리

데이터 엔지니어링 관점에서 컴플라이언스는 다음 두 가지 핵심 권리를 기술적으로 구현해야 하는 과제를 의미합니다.

1. **삭제권 (Right of Erasure) / 잊힐 권리 (Right to be Forgotten):** 소비자가 자신의 개인정보 삭제를 요청할 경우, 시스템에서 해당 데이터를 완전히 삭제해야 합니다.

   기술적 난제

   데이터 레이크에 저장된 수백 TB의 데이터 속에서 특정 사용자 1명의 데이터를 찾아내어 정확하게 삭제하는 것은 매우 어렵습니다. 특히 Parquet 같은 불변(immutable) 파일 포맷에서는 특정 행만 삭제하기가 거의 불가능합니다. (Delta Lake 같은 기술이 이 문제를 해결해줍니다.)

2. **이동권 (Right of Portability):** 소비자가 자신의 데이터를 (경쟁사 등) 다른 서비스로 이전할 수 있도록, 식별 가능하고 재사용 가능한 형태로 추출하여 제공해야 합니다.

4.3 기술적 해결 방안

- **세분화된 업데이트/삭제 (Fine-grained Updates/Deletes):** Delta Lake, Hudi, Iceberg와 같은 최신 데이터 레이크 포맷은 대용량 데이터셋에 대해 특정 행을 업데이트하거나 삭제하는 기능을 지원하여 GDPR의 삭제권 준수를 돕습니다.
- **가명화 (Pseudonymization):** 개인정보(식별자)를 외부에서 식별 불가능한 키(가명, 토큰)로 대체(토큰화)하는 기법입니다.
  - **작동 방식:** 원본 [사용자 ID ↔ 토큰] 매핑 테이블을 안전한 곳에 별도로 보관하고, 모든 분석 시스템에서는 '토큰'만 사용합니다.
  - **삭제 요청 시:** 매핑 테이블에서 해당 사용자의 [ID ↔ 토큰] 연결 고리만 파괴하면, 시스템에 데이터가 남아있더라도 더 이상 해당 사용자와 연결(추적)할 수 없게 됩니다.

5 Spark 핵심 개념: 실행 구조와 파티션

5.1 Spark 실행 아키텍처 (Jobs → Stages → Tasks)

우리가 작성한 Spark 코드(예: PySpark, SQL)는 클러스터에서 복잡한 계층 구조로 나뉘어 실행됩니다.

Spark 작업 실행 계층

1. **Job (작업):** Spark 드라이버(Driver) 프로그램에 제출된 '액션(Action)'(예: 'save()', 'collect()') 하나당 하나의 Job이 생성됩니다.
2. **Stage (스테이지):** 하나의 Job은 **셔플(Shuffle)**이 발생하는 지점을 기준으로 여러 개의 Stage로 나뉩니다.
   - **셔플(Shuffle)이란?** 'groupBy', 'join' 등 데이터 재분배가 필요한 작업 시, 여러 작업자(Worker) 노드 간에 데이터가 네트워크를 통해 교환되는 비싼 작업입니다.
   - Stage는 셔플 없이 한 번에 실행될 수 있는 작업들의 묶음입니다.
3. **Task (태스크):** Stage 내에서 실행되는 가장 작은 작업 단위입니다.
   - 하나의 Task는 하나의 파티션(데이터 조각)에 대해 동일한 연산을 수행합니다.
   - Task는 실제 작업자인 Executor의 코어에서 병렬로 실행됩니다.

요약: Job (액션 기준) → Stages (셔플 기준) → Tasks (파티션 기준)

5.2 테이블 파티션 vs. Spark 파티션 (매우 중요)

초심자가 가장 혼동하기 쉬운 개념으로, 두 '파티션'은 목적과 작동 방식이 완전히 다릅니다.

□ 핵심 비유: 도서관

- **테이블 파티션 (물리적 책장):** 도서관에서 책을 '국가별'(예: 한국, 미국)로 다른 책장에 나눠 꽂는 것입니다. **디스크 I/O**를 줄이는 것이 목적입니다.
- **Spark 파티션 (논리적 작업자):** 도서관의 모든 책을 정리하기 위해 '몇 명의 사서'에게 작업을 나눠줄지 정하는 것입니다. **CPU 병렬성**을 높이는 것이 목적입니다.

아래 표는 두 파티션의 차이점을 자세히 비교합니다.

특징	테이블 파티셔닝 (Table Partitioning)	Spark 파티셔닝 (Spark Partitioning)
개념/범위	(DB/테이블 수준) 대용량 테이블을 특정 컬럼 값(예: 날짜, 국가)에 따라 더 작은 조각(파일 디렉토리)으로 분할.	(Spark 처리 수준) 데이터를 클러스터 노드 간에 어떻게 분산시켜 병렬 처리할지 정하는 논리적 단위.
목적	**데이터 스캔 최소화 (Partition Pruning)** 쿼리 성능 향상 (디스크 I/O 감소)	**병렬 처리 (Parallelism) 최적화** 리소스 활용률, 내결함성 (Fault Tolerance)
관리 주체	**사용자가 명시적으로 정의** (예: 'CREATE TABLE ... PARTITIONED BY (date)')	**Spark가 동적으로 결정** (입력 파일 크기, 클러스터 설정, 셔플 등에 따라)
최적화	**파티션 프루닝 (Partition Pruning):** 'WHERE date='2025-01-01'' 쿼리 시, 다른 날짜의 파티션(디렉토리)은 스캔조차 안 함.	**셔플 (Shuffle) 최소화:** 'join', 'groupBy' 시 데이터 재분배(셔플) 방식에 영향을 주어 네트워크 오버헤드를 줄임.

테이블 파티션과 Spark 파티션 비교

테이블 파티셔닝 모범 사례:

- 'WHERE' 절에서 자주 사용되는 컬럼을 선택합니다. (예: 날짜, 국가, 지역)
- 카디널리티(Cardinality, 고유값의 개수)가 너무 높은 컬럼(예: 사용자 ID)은 피해야 합니다. 파티션이 수백만 개로 잘게 쪼개져 오히려 성능이 저하됩니다.
- 각 파티션의 데이터 크기는 최소 1GB 이상이 되도록 하는 것이 좋습니다. (작은 파일 문제 방지)
- (Databricks 기준) 최근에는 데이터가 1TB 미만이면 굳이 파티셔닝하지 않는 것을 권장합니다. 내부 최적화 엔진(Delta Lake)이 더 잘 처리할 수 있습니다.

5.3 Z-Ordering (Z-순서 지정)

Z-Ordering은 테이블 파티셔닝을 보완하는 Delta Lake의 데이터 레이아웃 최적화 기법입니다.

- **목적:** 파티션 키로 사용하지 않았지만 **자주 함께 조회되는 여러 컬럼**의 데이터들을 물리적으로 가깝게 모아(데이터 건너뛰기, Data Skipping) 쿼리 속도를 향상시킵니다.
- **작동 방식:** 다차원(여러 컬럼)의 값들을 Z-커브라는 알고리즘을 사용해 1차원의 값으로 변환한 뒤, 이 값을 기준으로 데이터를 정렬합니다.
- **예시:** '(IP 주소, 포트)'처럼 자주 같이 필터링되지만 파티션 키로는 부적절한 컬럼들에 'ZORDER BY ($\text{ip}_address, port$)' , .

```sql
-- 파일들을압축 (Bin-packing)하고
-- eventType 컬럼을기준으로데이터를재정렬 (Z-Ordering)
OPTIMIZE events
ZORDER BY (eventType)
```

Listing 1: OPTIMIZE와 Z-Ordering 사용 예시

6 Spark 조인 전략 (Spark Join Strategies)

Spark SQL은 두 테이블을 조인할 때, 데이터의 크기, 분포, 힌트 등을 고려하여 가장 효율적인 조인 전략을 자동으로 선택합니다. 어떤 전략이 있는지 아는 것은 성능 튜닝에 매우 중요합니다.

전략	설명	요구 조건	특징 (장/단점)
Broadcast Hash Join (BHJ)	하나의 작은 테이블을 모든 Executor에 **복제(Broadcast)**하여 메모리에 올린 뒤, 큰 테이블의 파티션과 해시 조인.	한쪽 테이블이 매우 작아야 함 (기본 10MB, 설정 가능)	**가장 빠름.** **셔플(Shuffle) 없음.** 정렬(Sort) 없음.
Shuffle Hash Join (SHJ)	두 테이블을 조인 키 기준으로 **셔플(재분배)**한 뒤, 각 Executor에서 더 작은 쪽의 해시 테이블을 만들며 조인.	한쪽 테이블이 (전체는 아니어도) 각 파티션이 메모리에 들어갈 만큼 작아야 함.	셔플 발생. (단점) 정렬(Sort) 없음. (장점) 데이터가 심하게 쏠리면(Skew) OOM(메모리 부족) 발생 가능.
Sort-Merge Join (SMJ)	두 테이블을 조인 키 기준으로 **셔플(재분배)**하고, 각 파티션 내에서 **정렬(Sort)**한 뒤, 병합(Merge)하며 조인.	조인 키가 정렬 가능해야 함.	**가장 안정적.** (기본 전략) 셔플과 정렬 모두 필요. (단점) 어떤 크기의 데이터도 처리 가능.
Cartesian Join (Cross Join)	조인 조건이 없거나 비효율적일 때 가능한 모든 행의 조합(N × M)을 생성.	(없음)	**최악의 조인. (피해야 함)** 거대한 셔플 발생.

Spark의 주요 조인 전략 비교

□ Spark 조인 힌트

Spark 옵티마이저가 잘못된 전략을 선택할 경우, SQL 쿼리 내에 힌트(‘/*+ **BROADCAST(small**$_{t}$*able*) ∗ /‘) .

7 데이터 조작(CRUD) 및 스키마 관리

7.1 CRUD (Create, Read, Update, Delete)

빅데이터 환경, 특히 Delta Lake에서는 전통적인 RDBMS와 유사하게 CRUD 작업을 수행할 수 있습니다.

- **Create (생성):**
  - **Append (추가):** 기존 데이터에 새 데이터를 추가합니다. (기본 모드)
  - **Overwrite (덮어쓰기):** 기존 데이터를 모두 삭제하고 새 데이터로 교체합니다. (‘.mode(”overwrite”)‘)
- **Read (읽기):** ‘SELECT‘, ‘.read()‘. 파티션, 키, Z-Ordering을 활용하여 효율적으로 읽습니다.
- **Update (수정):** ‘UPDATE ... SET ... WHERE ...‘. 특정 조건을 만족하는 행의 값을 수정합니다. (Delta Lake 지원)
- **Delete (삭제):** ‘DELETE FROM ... WHERE ...‘. 특정 조건을 만족하는 행을 삭제합니다. (Delta Lake 지원)

7.2 Upsert와 Merge (중요)

Upsert는 ’Update + Insert’의 합성어로, 데이터 엔지니어링에서 매우 흔하게 발생하는 작업입니다.

□ 예제: title

매일 밤 소스 시스템에서 최신 고객 목록을 가져옵니다.

- 이 고객이 타겟 테이블에 **없으면** → **INSERT** (신규 고객)
- 이 고객이 타겟 테이블에 **있으면** → **UPDATE** (기존 고객 정보 변경)

이 Upsert 로직은 Delta Lake의 ‘MERGE‘ 명령어를 통해 하나의 원자적(atomic) 명령으로 간단하게 처리할 수 있습니다.

```
MERGE INTO target_table AS t
USING source_updates AS s
ON t.id = s.id
WHEN MATCHED THEN
    -- 조건이맞으면기존 ( 고객) 업데이트
    UPDATE SET t.name = s.name, t.address = s.address
WHEN NOT MATCHED THEN
    -- 조건이안맞으면신규  ( 고객) 삽입
    INSERT (id, name, address) VALUES (s.id, s.name, s.address)
```

Listing 2: MERGE INTO를 사용한 Upsert 구현

7.3 스키마 관리 (Schema Management)

데이터 파이프라인 운영 시 스키마(데이터 구조) 변경은 피할 수 없는 문제입니다. Spark는 3가지 스키마 관리 전략을 제공합니다.

1. **스키마 추론 (Schema Inference):** Spark가 데이터(예: JSON, CSV)를 샘플링하여 스키마를 **자동으로 추측**합니다. 간편하지만, 데이터가 크거나 형식이 불일치하면(예: ‘id‘가 어떨 땐 숫자, 어떨 땐

문자열) 부정확하거나 성능이 저하될 수 있습니다.

2. **스키마 진화 (Schema Evolution):** 기존 스키마에 **새로운 컬럼이 추가**되는 것을 허용하는 옵션입니다. (예: 'df.write.option("mergeSchema", "true") ...')
   - **장점:** 파이프라인 중단 없이 소스의 스키마 변경(컬럼 추가)에 유연하게 대응할 수 있습니다.
   - **주의:** 데이터 타입이 호환되지 않게 변경(예: String → Int)되는 것은 막지 못할 수 있습니다.
3. **스키마 강제 (Schema Enforcement): 기본 동작**입니다. 저장하려는 데이터의 스키마가 타겟 테이블의 스키마와 일치하지 않으면 **에러를 발생**시키고 작업을 중단시킵니다.
   - **장점:** 데이터 품질을 엄격하게 관리할 수 있습니다. 예기치 않은 데이터(예: 'age' 컬럼에 문자열)가 유입되는 것을 막아줍니다.

☐ 권장 파이프라인 전략 (Multi-Hop)

- **Bronze (수집 단계):** 스키마 진화 (Schema Evolution) 허용. (소스에서 어떤 변경이 생길지 모르므로, 일단 모든 데이터를 받아들입니다.)
- **Silver/Gold (정제/집계 단계):** 스키마 강제 (Schema Enforcement) 적용. (데이터 컨트랙트가 확정된 단계이므로, 약속된 스키마 외의 데이터는 거부하여 품질을 보장합니다.)

8 변경 데이터 관리 (CDC & SCD)

데이터는 정지해 있지 않고 끊임없이 변경됩니다. 이러한 변경 사항을 추적하고 관리하는 기법은 매우 중요합니다.

8.1 CDC (Change Data Capture, 변경 데이터 캡처)

CDC는 **소스(Source) 시스템**에서 발생한 변경 사항(Insert, Update, Delete)을 식별하고 '캡처'하는 기술입니다.

- **목적:** 전체 테이블을 매번 복사(Full Load)하는 대신, 변경된 내용만(Incremental Load) 가져와 타겟 시스템에 효율적으로 반영하기 위함입니다.
- **예시:** 데이터베이스의 트랜잭션 로그를 읽어 'A 고객의 주소가 변경됨', 'B 고객이 삭제됨' 같은 이벤트 로그를 생성합니다.

Delta Lake의 **Change Data Feed (CDF)** 기능은 Delta 테이블 자체에 대한 변경 내역(pre-image, post-image)을 로그로 제공하여, Delta 테이블을 CDC의 소스로 활용할 수 있게 해줍니다.

8.2 SCD (Slowly Changing Dimensions, 느리게 변하는 차원)

SCD는 CDC로 캡처된 변경 사항을 **타겟(Target) 시스템**(주로 데이터 웨어하우스의 차원 테이블)에 **어떻게 반영할지 결정하는 정책**입니다.

'느리게 변하는 차원'(예: 고객, 제품, 매장)은 트랜잭션(사실) 데이터만큼 자주 변하지는 않지만, 변경 시 그 이력을 어떻게 관리할지가 중요합니다.

- **Type 0 (Fixed, 고정):** 변경을 허용하지 않습니다. (예: 입사일)
- **Type 1 (Overwrite, 덮어쓰기): 이력을 관리하지 않습니다.** 변경 사항이 발생하면 기존 데이터를 그냥 덮어씁니다.
- **Type 2 (New Row, 새 행 추가): 모든 변경 이력을 관리합니다.** 변경 사항이 발생하면 기존 행은 '만료' 처리하고, 새로운 행을 추가하여 이력을 쌓습니다.
- **Type 3 (New Column, 새 컬럼 추가):** 이전 값을 저장하기 위해 'previous_address' 같은 새 컬럼을 추가합니다. (이력이 몇 개 없을 때만 사용, 비추천)
- **Type 4 (History Table, 이력 테이블):** 현재 값은 원본 테이블에 유지하고, 모든 변경 이력은 별도의 이력 테이블에 저장합니다.
- **Type 6 (Hybrid):** Type 1, 2, 3을 조합한 하이브리드 방식입니다.

핵심: Type 1 vs Type 2

실무에서 가장 많이 사용되는 것은 Type 1과 Type 2입니다.

- **Type 1:** 이력이 필요 없을 때. (예: 잘못된 스펠링 수정)
- **Type 2:** 모든 변경 이력 추적이 필요할 때. (예: 고객 주소 변경에 따른 매출 지역 분석)

8.3 SCD Type 1 및 Type 2 상세 예시

8.3.1 SCD Type 1: Overwrite (덮어쓰기)

정책: 이력 없음. 변경 시 원본 데이터를 덮어씁니다.

초기 원본 테이블 (Target Dimension Table):

cust_id	name
1	A
2	B

변경 데이터 발생 (Incoming Data):

- ID 1: 이름 'A' → 'AA'로 변경 (Update)
- ID 2: 삭제 (Delete)
- ID 3: 신규 추가 (Insert)

SCD Type 1 적용 후 최종 테이블:

cust_id	name	a
1	**AA**	
3	C	

Type 1 결과 (ID 2는 삭제되고 ID 1은 덮어쓰기됨) **단점:** ID 1의 이름이 'A'였다는 사실과 ID 2가 존재했다는 이력이 사라집니다.

8.3.2 SCD Type 2: New Row (새 행 추가)

정책: 모든 이력 보관. 변경 시 새 행을 추가하고, 이력 관리를 위한 추가 컬럼(시작일, 종료일, 현재 여부 플래그)을 사용합니다.

초기 원본 테이블 (Target Dimension Table):

cust_id	name	age	s_date	
1	A	21	2021-09-29	9
2	B	31	2021-09-29	9

동일한 변경 데이터 발생 (2021-10-02):

- ID 1: 이름 'A' → 'AA'로 변경 (Update)
- ID 2: 삭제 (Delete)
- ID 3: 신규 추가 (Insert)

SCD Type 2 적용 후 최종 테이블:

cust_id	name	age	s_date	e_dat
1	A	21	2021-09-29	**2021-10**
1	**AA**	21	**2021-10-02**	9999-12-
gray!10 2	B	31	2021-09-29	**2021-10**
3	C	16	**2021-10-02**	9999-12-

Type 2 결과 (모든 이력이 보존됨) **장점:** 2021년 9월 30일 시점에서는 ID 1의 이름이 'A'였음을 정확히 추적할 수 있습니다. 'WHERE current = 'T''로 항상 최신 데이터만 조회할 수도 있습니다.

9 고급 기능 및 UDF (Advanced Features & UDFs)

9.1 Delta Lake 관리 기능

Delta Lake는 데이터 레이크의 안정성을 높이기 위해 다음과 같은 관리 명령어를 제공합니다.

- **OPTIMIZE (최적화):** 작은 파일들을 더 큰 파일로 병합(Bin-packing)하여 쿼리 성능을 향상시킵니다. (작은 파일 문제 해결) 'ZORDER BY'와 함께 사용하여 데이터 레이아웃을 최적화할 수 있습니다.
- **RESTORE (복원):** 실수로 데이터를 잘못 덮어썼을 때, Delta Lake의 트랜잭션 로그(버전 관리)를 사용하여 테이블을 특정 버전이나 타임스탬프로 되돌립니다. (Time Travel)

```
RESTORE TABLE my_table TO TIMESTAMP AS OF '2025-10-25 00:00:00'
```

Listing 3: 테이블을 어제 시간으로 복원

- **VACUUM (진공청소):** Time Travel을 위해 유지되던 오래된 버전의 데이터 파일 중, 설정된 보존 기간(기본 7일)이 지난 파일들을 실제로 삭제하여 스토리지 비용을 절감합니다.

VACUUM 주의사항

보존 기간(0일)을 너무 짧게 설정하고 'VACUUM'을 실행하면, Time Travel로 복구할 수 있는 데이터가 사라지므로 주의해야 합니다.

- **CLONE (복제):** 테이블을 복제합니다.
  - **Shallow Clone (얕은 복제):** 메타데이터만 복사합니다. (Zero Copy Clone) 원본 데이터 파일을 참조하므로 즉시 생성되며 비용이 거의 들지 않습니다. (테스트용)
  - **Deep Clone (깊은 복제):** 메타데이터와 데이터 파일 전체를 복사합니다. (백업, 마이그레이션용)

9.2 UDF (사용자 정의 함수)

Spark SQL이 제공하는 내장 함수 외에, 사용자가 직접 정의한 복잡한 로직을 실행해야 할 때 UDF를 사용합니다.

- **UDF (User Defined Function):** 가장 일반적인 형태. 행(Row) 단위로 작동합니다. (1:1 행 입력 → 1:1 값 출력)
- **Pandas UDF (Vectorized UDF): 성능이 훨씬 뛰어난 UDF**입니다. 데이터를 행 단위가 아닌, Apache Arrow를 통해 Pandas Series/DataFrame '뭉치'(Vector) 단위로 전달받아 처리합니다.
- **UDAF (User Defined Aggregate Function):** 사용자 정의 '집계' 함수. 여러 행을 입력받아 단일 값을 출력합니다. (N:1) (예: 'SUM', 'COUNT')
- **UDTF (User Defined Table Function):** 사용자 정의 '변환' 함수. 하나의 행을 입력받아 여러 행(테이블)을 출력할 수 있습니다. (1:N) (예: 'EXPLODE')

□ UDF vs Pandas UDF: 왜 Pandas UDF가 빠를까?

Spark의 핵심은 JVM(Java/Scala) 위에서 실행되고, PySpark는 Python 인터프리터에서 실행됩니다.

- **일반 Python UDF:** 데이터를 처리할 때마다 JVM ↔ Python 간에 **비싼 데이터 직렬화(Serialization)/역직렬화(Deserialization)가 발생**합니다. (마치 한 줄 한 줄 통역하는 것과 같습니다.)

- **Pandas UDF:** Apache Arrow라는 메모리 형식을 사용하여 JVM과 Python 간에 데이터를 '뭉치'로 복사 없이 공유합니다. 직렬화 오버헤드가 거의 없어 성능이 매우 뛰어납니다. (마치 책 한 권을 통째로 넘겨주는 것과 같습니다.)

결론: Python UDF를 사용해야 한다면, 성능을 위해 항상 Pandas UDF를 우선적으로 고려해야 합니다.

```python
from pyspark.sql.functions import pandas_udf, PandasUDFType

# Pandas UDF 정의데코레이터 ( 사용)
@pandas_udf('double', PandasUDFType.SCALAR)
def pandas_plus_one(v):
    # 입력는 v Pandas Series 객체
    return v + 1

df = spark.range(10)
df.withColumn('id_transformed', pandas_plus_one("id")).show()
```

Listing 4: Pandas UDF (Vectorized UDF) 예시

10 실습: Autoloader를 이용한 데이터 로딩

10.1 Autoloader란?

Autoloader는 클라우드 스토리지(S3, ADLS 등)에 **새로운 파일이 도착했을 때, 이를 자동으로 감지하여 증분(incrementally) 처리**해주는 Spark의 스트리밍 소스입니다.

- **작동 방식:** 파일 기반 스트리밍입니다. 별도 설정 없이 파일 도착을 효율적으로 감지합니다.
- **주요 기능:** 스키마 추론(Inference) 및 진화(Evolution)를 강력하게 지원하며, 불량 데이터 처리가 용이합니다.
- **사용법:** '.readStream.format("cloudFiles")'를 사용합니다.

10.2 Autoloader 핵심 옵션 및 코드

```
(spark.readStream
    .format("cloudFiles")  # Autoloader 사용
    .option("cloudFiles.format", "json")  # 원본파일포맷
    .option("cloudFiles.schemaLocation", "/path/to/schema/location") # 스키마저장위치
    .load("/path/to/source/data")
.writeStream
    .option("checkpointLocation", "/path/to/checkpoint/location") # 체크포인트
    .option("mergeSchema", "true") # 스키마진화허용
    .table("bronze_table")
)
```

Listing 5: Autoloader 기본 사용법

- **'cloudFiles.format':** 원본 파일 형식(JSON, CSV, Parquet 등)을 지정합니다.
- **'cloudFiles.schemaLocation':** Autoloader가 추론하거나 변경한 스키마 정보를 저장할 위치입니다. 이 위치를 기준으로 스키마 버전이 관리됩니다.
- **'mergeSchema' (Write 옵션):** 스키마 진화를 허용합니다. 소스 파일에 새 컬럼이 추가되면, 파이프라인 중단 없이 타겟 테이블(Bronze)의 스키마도 변경됩니다.

10.3 불량 데이터 처리 (Handling Bad Data)

데이터 로딩 시, 스키마가 맞지 않는(예: 'int' 컬럼에 문자열) 불량 데이터가 들어오면 파이프라인이 실패할 수 있습니다. Autoloader는 이를 우아하게 처리하는 'Rescue' 기능을 제공합니다.

- '$_{r}escued_{d}ata$' : *Autoloader* , '$_{r}escued_{d}ata$' . (*Parsing*) ( ) , '*NULL*'
- **'cloudFiles.schemaEvolutionMode = "rescue"' (옵션): 스키마 힌트와 일치하지 않는 데이터 타입을 발견했을 때, 해당 데이터를 '$_{r}escued_{d}ata$' .**

□ 예제: title

1. 파이프라인은 '$_{r}escued_{d}ata$' .2. '$WHERE_{r}escued_{d}ataISNOTNULL$' .3. '*CAST*'), (*Repair*) .

10.4 스키마 힌트 (Schema Hints)

스키마 추론은 완벽하지 않습니다. (예: ‘id‘가 ‘1‘만 있으면 ‘int‘로 추론하지만, 사실은 ‘string‘일 수 있음) **스키마 힌트**는 Autoloader에게 특정 컬럼의 데이터 타입을 강제하도록 ’힌트’를 주는 기능입니다.

```
.option("cloudFiles.schemaHints", "timestamp long, signal_strength float")
```

Listing 6: 스키마 힌트 적용

위 예시에서 ‘timestamp‘가 문자열로 추론되더라도 ‘long‘ 타입으로, ‘signal$_s$*trength*‘ ‘*string*‘ ‘*float*‘ .

10.5 스트리밍 중단 (Stopping Streams)

Jupyter(Databricks) 노트북에서 스트리밍 작업을 시작하면, 명시적으로 중단하지 않는 한 계속 실행됩니다. 모든 활성 스트림을 중단하려면 다음 코드를 사용합니다.

```
for s in spark.streams.active:
    s.stop()
```

Listing 7: 모든 활성 스트림 중단

11 용어 정리 (Glossary)

용어	원어	쉬운 설명
디자인 패턴	Design Pattern	자주 발생하는 문제에 대한 검증된 해결책(설계도).
람다 아키텍처	Lambda Architecture	배치(Batch)와 스피드(Speed) 레이어를 분리하여 운영하는 아키텍처.
카파 아키텍처	Kappa Architecture	모든 것을 스트리밍으로 처리하는 단일 파이프라인 아키텍처.
CQRS	Command Query Responsibility Segregation	쓰기(Command)와 읽기(Query)의 책임을 완전히 분리하는 패턴.
폴리글랏 퍼시스턴스	Polyglot Persistence	하나의 DB가 아닌, 여러 종류의 DB를 목적에 맞게 조합하여 사용.
컴플라이언스	Compliance	법규(예: GDPR)를 준수하는 것.
삭제권	Right of Erasure	사용자가 본인의 데이터 삭제를 요청할 수 있는 권리.
테이블 파티션	Table Partition	디스크 I/O 감소를 위해 데이터를 물리적 디렉토리로 나누는 것. (책장 정리)
Spark 파티션	Spark Partition	CPU 병렬 처리를 위해 데이터를 논리적 작업 단위로 나누는 것. (작업자 분배)
Z-Ordering	Z-Ordering	여러 컬럼을 기준으로 데이터를 물리적으로 가깝게 정렬하는 최적화 기법.
브로드캐스트 조인	Broadcast Hash Join	작은 테이블을 모든 노드에 복제(방송)하여 셔플 없이 수행하는 가장 빠른 조인.
셔플	Shuffle	‘join‘이나 ‘groupBy‘ 시, 노드 간 데이터 재분배가 일어나는 비싼 네트워크 작업.
CRUD	Create, Read, Update, Delete	데이터의 기본 연산 (생성, 읽기, 수정, 삭제).
Upsert	Update + Insert	데이터가 있으면 수정(Update), 없으면 삽입(Insert)하는 작업. ‘MERGE‘로 구현.
스키마 진화	Schema Evolution	기존 스키마에 새로운 컬럼 추가를 허용하는 것. (‘mergeSchema=true‘)
CDC	Change Data Capture	소스 시스템의 변경(Insert, Update, Delete) 내역을 캡처하는 기술.
SCD	Slowly Changing Dimensions	변경 내역(CDC)을 타겟 차원 테이블에 어떻게 반영할지 정하는 정책(이력 관리).
SCD Type 1	Overwrite	SCD 정책 중 하나. 이력 없이 기존 데이터를 덮어씀.
SCD Type 2	New Row	SCD 정책 중 하나. 변경 시 새 행을 추가하여 모든 이력을 보관함.
Pandas UDF	Pandas UDF (Vectorized)	Python ↔ JVM 간 직렬화 없이 Arrow를 통해 데이터를 ’뭉치’로 처리하는 고성능 UDF.
Autoloader	Autoloader	클라우드 스토리지의 신규 파일을 자동으로 감지하여 증분 로딩하는 Spark 기능.
_rescued_data	_rescued_data	Autoloader가 파싱에 실패한 불량 데이터를 격리하는 컬럼.

강 핵심 용어 정리

4

12 FAQ 및 과제 Q&A (FAQ and Homework Q&A)

Q: 왜 굳이 Delta Lake 같은 복잡한 걸 쓰나요? 그냥 RDBMS(관계형 DB)를 쓰면 안 되나요?

A: 스케일(Scale) 때문입니다. RDBMS는 수십 TB(테라바이트) 수준의 데이터를 처리하고 저장하는 데는 비용과 성능의 한계가 명확합니다. Delta Lake와 같은 레이크하우스 기술은 저렴한 클라우드 스토리지(S3 등)를 기반으로 하면서도, RDBMS의 장점인 ACID 트랜잭션, 업데이트/삭제(CRUD), 안정적인 스키마 관리를 페타바이트(PB)급 대용량 데이터에서도 가능하게 해줍니다.

Q: 테이블 파티션과 Z-Ordering은 언제 같이 쓰나요?

A: 두 기술은 상호 보완적입니다.

- **1순위 (테이블 파티션):** 데이터 조회 시 **항상 사용되는 필터** (예: '날짜', '국가'). 카디널리티가 낮은(수십 수백 개) 컬럼에 적용하여 스캔할 데이터의 총량을 대폭 줄입니다.
- **2순위 (Z-Ordering):** 파티션으로 거른 데이터 안에서 **추가로 자주 사용되는 필터** (예: 'IP주소', '사용자ID'). 카디널리티가 높은(수백만 개 이상) 컬럼에 적용하여 파티션 내부의 파일들을 효율적으로 건너뛰게(Skip) 만듭니다.

(예: 'PARTITIONED BY (date)', 'ZORDER BY ($user_id$)')

과제 Q&A: ERD 다이어그램 작성 시 모든 테이블에 Primary Key(PK)가 있어야 하나요?

A: 아니요, 필수는 아닙니다. 물론 대부분의 차원 테이블(Dimension)에는 고유한 식별자(PK)가 있습니다. 하지만 트랜잭션 데이터를 담는 사실 테이블(Fact)이나, 두 테이블을 연결하는 브릿지 테이블(예: 'partsupp' 테이블)은 두 개 이상의 키를 조합해야만(Composite Key) 고유성이 보장되거나, 혹은 PK가 아예 정의되지 않을 수도 있습니다. TPC-H 데이터셋의 'lineitem' 테이블도 '($l_orderkey, l_linenumber$)' .

과제 Q&A: ERD 다이어그램 툴은 아무거나 사용해도 되나요?

A: 네, 그렇습니다. 'dbdiagram.io'와 같이 무료로 사용할 수 있고, 결과물을 스크린샷으로 찍거나 공개 링크로 공유하여 과제에 첨부할 수 있다면 어떤 툴을 사용해도 좋습니다.

13 한눈에 보기 (Quick Look Summary)

디자인 패턴 (Design Patterns)

- **Lambda:** 배치(Batch) + 스트리밍(Speed) 2개 파이프라인.
- **Kappa:** 스트리밍(Streaming) 1개 파이프라인.
- **Multi-Hop (Medallion):** 데이터 정제 단계 (Bronze → Silver → Gold).
- **CQRS:** 쓰기(Write)와 읽기(Read)를 분리.
- **Polyglot Persistence:** 여러 종류의 DB를 조합하여 사용.

Spark 파티션 (매우 중요)

- **Table Partition (테이블 파티션): 물리적** 저장. 디스크 I/O 감소 목적. (예: 'PARTITIONED BY (date)')
- **Spark Partition (스파크 파티션): 논리적** 분배. CPU 병렬 처리 목적. (셔플 단위)

Spark 조인 전략 (Join Strategies)

- **Broadcast Hash Join (빠름):** 작은 테이블 → 모든 노드에 복사. (셔플 없음)
- **Sort-Merge Join (안정적):** 큰 테이블 → 셔플 + 정렬 후 병합. (기본값)

데이터 변경 관리 (CDC vs SCD)

- **CDC (변경 데이터 캡처):** 소스(Source)의 변경 사항(Insert/Update/Delete)을 감지.
- **SCD (느리게 변하는 차원):** 타겟(Target)에 변경 이력을 어떻게 반영할지에 대한 정책.
- **SCD Type 1 (덮어쓰기):** 이력 없음. 'UPDATE'
- **SCD Type 2 (새 행 추가):** 모든 이력 보관. 'INSERT' + (기존 행 만료 처리)

Delta Lake & Autoloader

- **MERGE:** Upsert (Update + Insert)를 원자적으로 실행.
- **OPTIMIZE:** 작은 파일을 병합. (Z-Ordering과 사용)
- **Pandas UDF:** Python UDF 성능 최적화 (Arrow 사용, 직렬화↓)
- **Autoloader ('cloudFiles'):** 신규 파일 증분 로딩.
- '$_{r}escued_{d}ata$' : .