December 10, 2025

▣ **강의명:** CSCI E-103: 재현 가능한 머신러닝

▣ **주차:** Lecture 03

▣ **교수명:** Anindita Mahapatra

Eric Gieseke

▣ **목적:** Lecture 03의 핵심 개념 학습

Contents

1 개요 (Overview)

이 문서는 데이터 엔지니어링의 핵심 구성 요소인 데이터 파이프라인, 특히 ETL(추출, 변환, 적재) 프로세스에 대해 다룹니다.

데이터 처리의 두 가지 주요 패러다임인 **배치(Batch) 처리**와 **스트리밍(Streaming) 처리**의 개념을 비교하고, 이 두 가지를 통합하려는 시도에서 발전한 **람다(Lambda) 아키텍처**와 **카파(Kappa) 아키텍처**의 차이점을 중점적으로 학습합니다.

이 문서를 통해 데이터를 효율적으로 수집하고 처리하는 방법론을 이해하고, 다양한 비즈니스 요구사항에 맞는 적절한 데이터 아키텍처를 선택하는 기준을 확립하는 것을 목표로 합니다. 또한 Spark Structured Streaming을 사용한 실습 예제를 포함하여 이론적 개념이 실제 코드에서 어떻게 구현되는지 살펴봅니다.

2 지난 강의 핵심 복습

데이터 파이프라인과 스트리밍을 배우기에 앞서, 데이터 모델링 및 저장소에 대한 이전 강의의 핵심 용어들을 복습합니다.

Table 1: 데이터 모델링 및 저장소 핵심 용어 복습

개념 (Concept)	핵심 설명 (Explanation)	예시 (Example)
3NF (정규화)	**데이터 중복을 최소화**하고 데이터 무결성을 보장하기 위한 관계형 데이터베이스 모델링 기법입니다.	고객 테이블, 주문 테이블, 상품 테이블을 분리하여 관리합니다.
비정규화 (Denormalization)	데이터 중복을 감수하는 대신, **빠른 조회 속도**를 얻기 위해 테이블을 통합하거나 중복 필드를 추가하는 기법입니다. (주로 NoSQL에서 사용)	고객 테이블에 '최근 주문일'을 중복 저장하여 조인(Join)을 피합니다.
ETL / ELT	**ETL** (Extract, Transform, Load): 추출 → **변환** → 적재. 전통적인 방식. **ELT** (Extract, Load, Transform): 추출 → 적재 → **변환**. 데이터 레이크하우스에서 주로 사용.	ETL: CRM 데이터를 정제하여 DW에 저장. ELT: 원본 로그를 S3에 먼저 올리고 Spark로 변환.
Star / Snowflake Schema	데이터 웨어하우스(DW)의 대표적인 스키마 디자인입니다. **Star (성형)**: 하나의 **팩트(Fact) 테이블**이 여러 **차원(Dimension) 테이블**에 직접 연결됩니다. (비정규화) **Snowflake (눈송이)**: 차원 테이블이 추가로 정규화되어 여러 단계로 나뉩니다.	Star: '판매' 팩트가 '날짜', '고객', '상품' 차원에 바로 연결됨.Snowflake: '상품' 차원이 '카테고리' 차원으로 추가 분리됨.
키-값 스토어	고유한 키(Key)와 값(Value)의 쌍으로만 데이터를 저장하는 가장 단순한 NoSQL 모델입니다.	**Amazon S3**, Redis
도큐먼트 스토어	JSON이나 XML 같은 유연한 **문서(Document) 형식**으로 데이터를 저장합니다. 스키마가 유연합니다.	**MongoDB**, CouchDB
컬럼형 스토어	데이터를 행(Row) 단위가 아닌 **열(Column) 단위**로 저장하여, 분석 쿼리(OLAP)에 매우 높은 성능을 보입니다.	**Cassandra**, DynamoDB
바이너리 포맷	텍스트(CSV, JSON)보다 압축률이 높고 **성능이 뛰어난** 이진 파일 형식입니다. **Parquet**: 컬럼(열) 기반 저장 방식으로 분석에 최적화되어 있습니다.	Avro, **Parquet**
델타 레이크 (Delta Lake)	데이터 레이크(S3 등) 위에 **ACID 트랜잭션, 스키마 진화, 타임 트래블** (버전 관리) 기능을 제공하는 오픈소스 스토리지 포맷입니다.	Parquet 파일과 _delta_log(트랜잭션 로그)로 구성됩니다.
메타데이터 (Metadata)	**데이터에 대한 데이터**로, 데이터의 구조, 유형, 이력 등 맥락을 설명하는 정보입니다.	테이블 스키마, 델타 레이크의 트랜잭션 로그.

3 데이터 파이프라인 (Data Pipelines)

3.1 데이터 파이프라인이란?

데이터 파이프라인(Data Pipeline)은 데이터를 한 지점(소스)에서 다른 지점(대상)으로 이동시키는 전체 프로세스를 의미합니다. 이는 데이터 이동을 위한 일반적인 "배관(Plumbing)" 작업에 비유할 수 있습니다.

데이터 파이프라인은 단순히 데이터를 A에서 B로 복사하는 간단한 작업일 수도 있고, 여러 소스에서 데이터를 수집하고, 복잡하게 변환하며, 여러 대상에 저장하는 정교한 프로세스일 수도 있습니다.

3.2 ETL 파이프라인: 특별한 목적

ETL(Extract, Transform, Load) 파이프라인은 데이터 파이프라인의 **특수한 유형**입니다. 이는 원시(Raw) 데이터를 수집하여 비즈니스 분석, 애플리케이션, 머신러닝(ML) 시스템에서 즉시 사용할 수 있는 '준비된 데이터'로 만드는 데 특화되어 있습니다.

□ 핵심 요약

데이터 파이프라인 흐름 예시 **데이터 소스 → 수집 (Ingestion) → Raw/Landing (원본 저장) → 변환 (ELT/ETL) → Curated (정제됨) → 데이터 목적지**

데이터 파이프라인 vs. ETL 파이프라인

데이터 파이프라인 (Data Pipeline)	ETL 파이프라인 (ETL Pipeline)

ETL 파이프라인은 데이터 분석을 목적으로 하는, 데이터 파이프라인의 전문화된 하위 집합입니다.

3.3 파이프라인 구축 프로세스 및 자동화

데이터 파이프라인은 일회성 작업이 아니라 지속적으로 관리되어야 하는 소프트웨어 애플리케이션과 같습니다.

- **구축 프로세스**: 설계 → 구현 → 테스트 → 배포 → 모니터링
- **변경 관리**: 비즈니스 요구사항이나 데이터 소스가 변경되면 파이프라인을 수정하고 다시 배포해야 합니다.
- **자동화 (Automation)**: 파이프라인의 핵심 가치입니다. 수동 개입 없이 자동으로 실행되어야 합니다.

파이프라인이 실행되는 시점을 결정하는 것을 **트리거(Trigger)**라고 하며, 주요 유형은 다음과 같습니다.

- **파일 도착 (File Arrival)**: 특정 위치에 새 파일이 도착하면 실행됩니다.
- **스케줄 (Schedule)**: 매일 오전 6시, 매주 월요일 등 정해진 일정(Cron)에 따라 실행됩니다.
- **수동 (Manual)**: 사용자가 직접 실행을 명령합니다.

3.4 장점과 도전과제

□ 핵심 요약

파이프라인의 장점 (Pros)

- **셀프 서비스 및 IT 의존도 감소**: 현업 사용자가 IT 팀의 개입 없이도 자동화된 채널을 통해 필요한 데이터에 접근할 수 있습니다.
- **클라우드 자원 활용**: 데이터 양에 따라 동적으로 컴퓨팅 자원을 확장(Elasticity)하여 비용 효율적으로 처리할 수 있습니다.
- **실시간 의사결정 지원**: 실시간 분석 및 애플리케이션을 지원하여 비즈니스의 빠른 의사결정을 돕습니다.

주의사항

파이프라인의 도전과제 (Cons)

- **데이터 드리프트 (Data Drift)**: 가장 큰 문제입니다. 원본 데이터의 **스키마** (열 추가/삭제/변경)나 **품질** (값의 형식 변경)이 예고 없이 변경되면 파이프라인이 실패할 수 있습니다.
- **인프라 종속성**: 파이프라인은 특정 기술, 인프라, 프로세스에 강하게 결합되어 있어, 해당 환경이 변경되면 파이프라인도 영향을 받습니다.

4 파이프라인 유형 비교

데이터 파이프라인은 처리 방식, 목적, 사용 도구에 따라 다양하게 분류할 수 있습니다.

Table 2: 주요 데이터 파이프라인 유형 비교

파이프라인 유형	처리 스타일	주요 도구	대표 사용 사례
배치 (Batch)	대량의 데이터를 **주기적**으로 처리 (시간별, 일별, 주별)	Apache Spark, AWS Glue, Talend	일일 리포트 생성, 급여 정산, 월간 빌링
스트리밍 (Real-Time)	**이벤트 중심**의 **연속적**인 데이터 처리	Apache Kafka, Flink, AWS Kinesis, Databricks Streaming	사기 탐지, 실시간 대시보드, IoT 센서 데이터
ETL	추출 → 변환 → 적재. (주로 배치 방식)	Informatica, Talend, Matillion, dbt	CRM/ERP 데이터를 정제하여 데이터 웨어하우스로 로드
ELT	추출 → 적재 → 변환. (배치 또는 실시간)	dbt, Snowflake, Databricks	원본 로그를 DW에 먼저 적재 후 나중에 변환
데이터 복제	데이터를 거의 실시간 또는 스케줄에 따라 동기화	Fivetran, Airbyte, AWS DMS	운영 DB(OLTP)를 분석용 DB(OLAP)로 복제
ML 파이프라인	모델 훈련 또는 배포를 위한 데이터 처리 (배치/실시간)	MLflow, Kubeflow, Vertex AI	고객 이탈 예측 모델 훈련 및 서빙
워크플로우 / 오케스트레이션	**파이프라인 간의 의존성**을 관리하는 '메타 파이프라인'	Apache Airflow, Prefect, Dagster	ETL 작업 실행 → 완료 시 모델 훈련 → 완료 시 대시보드 업데이트

5 스트리밍 (Streaming)

5.1 왜 스트리밍이 필요한가? (Why Streaming?)

"It's all about SPEED." (핵심은 속도입니다.)

스트리밍의 목적은 이벤트 스트림(연속적인 데이터 흐름)을 **더 빠르게** 분석 가능한 데이터로 전환하여 인사이트를 얻는 것입니다. 이는 비즈니스 중심의 가치(예: 사기 탐지)를 IT 중심의 코딩 프로세스보다 우선시하는 현대 데이터 플랫폼의 핵심입니다.

모든 작업에 실시간(초 단위) 처리가 필요한 것은 아닙니다. 하지만 비즈니스 요구사항에 따라 "배치(일 단위)"에서 "실시간(초 단위)"까지 다양한 속도 스펙트럼이 존재하며, 스트리밍 기술은 이 스펙트럼 전반을 다룰 수 있습니다.

5.2 스트리밍 핵심 용어 정리

스트리밍 파이프라인을 구성하는 핵심 개념들은 다음과 같습니다.

Table 3: 스트리밍 핵심 용어

용어 (Term)	설명 (Description)
소스 (Source) & 싱크 (Sink)	모든 파이프라인은 데이터를 읽어오는 **소스**(예: Kafka)와 데이터를 저장하는 **싱크**(예: Delta Table)로 구성됩니다.
파일 기반 vs. 이벤트 기반	**파일 기반**: 데이터가 디스크(파일)에 저장된 후 처리됩니다. (상대적 느림) **이벤트 기반**: 데이터가 메모리(이벤트)에서 디스크를 거치지 않고 바로 처리됩니다. (매우 빠름, 예: Kafka)
마이크로배치 vs. 연속	**마이크로배치 (Micro-batch)**: 데이터를 아주 작은 묶음(예: 1초)으로 나누어 배치처럼 처리합니다. (Spark Structured Streaming의 기본값) **연속 (Continuous)**: 이벤트가 도착하는 즉시 처리합니다. (Flink의 방식)
처리 트리거 (Trigger)	마이크로배치가 실행되는 간격을 의미합니다. (예: 30초마다)
출력 모드 (Output Modes)	싱크에 데이터를 쓰는 방식입니다. (Append: 추가, Overwrite: 덮어쓰기, Update: 갱신)
체크포인트 (Checkpoint)	**[매우 중요]** 파이프라인이 어디까지 처리했는지 기록하는 "게임 저장 지점"입니다. 장애 발생 시, 체크포인트부터 복구하여 데이터 유실이나 중복 없이 처리를 재개할 수 있습니다.
윈도우 (Window)	집계(Aggregation)를 위한 시간 간격입니다. (예: "지난 5분간"의 평균 클릭 수)
워터마크 (Watermark)	**[매우 중요]** "지연 도착 데이터"를 얼마나 기다려줄지 정의하는 허용 시간입니다. (예: "버스가 10분 더 기다려줌") 이는 집계 연산 시 **무한정 상태(State)를 저장하는 것을 방지**하여 메모리 초과(OOM)를 막는 핵심 기능입니다.

5.3 스트리밍 사용 사례 스펙트럼

모든 비즈니스가 초(second) 단위의 응답을 요구하지는 않습니다. 요구사항에 맞춰 적절한 속도를 선택해야 합니다.

Table 4: 데이터 처리 속도별 사용 사례

시간 단위 (Latency)	유형 (Type)	사용 사례 (Use Cases)
일 (Day) - 시간 (Hours)	배치 (Batch)	ETL, 미터링 및 빌링(정산), 임시 분석, BI 리포트
분 (Minutes)	준 실시간 (Near Real-Time)	모바일 및 IoT 데이터 수집, 서비스 모니터링 및 알림, 로그 수집, 클릭스트림 분석
초 (Sec) - 밀리초 (Sub-Sec)	실시간 (Real-Time)	**사기 탐지 (Fraud Detection)**, 금융 자산 거래, 멀티플레이어 게임, **ML 추론 (Inference)**

6 람다(Lambda) vs. 카파(Kappa) 아키텍처

주의사항

[용어 혼동 주의] AWS Lambda ≠ 람다 아키텍처 여기서 다루는 **람다(Lambda) 아키텍처**는 데이터 처리 파이프라인을 구성하는 **업계 표준 아키텍처 패턴(Pattern)**입니다.
이는 AWS의 서버리스(Serverless) 컴퓨팅 서비스인 **"AWS Lambda"와는 전혀 다른 개념**입니다.
두 용어가 우연히 이름이 같을 뿐이므로 혼동하지 않아야 합니다.

6.1 람다 아키텍처 (Lambda Architecture) - 구세대

람다 아키텍처는 스트리밍 기술이 성숙하지 않았던 시기에 **배치의 신뢰성**과 **스트리밍의 속도**라는 두 마리 토끼를 잡기 위해 고안된 방식입니다.

□ 핵심 요약

람다 아키텍처 흐름

- 데이터가 들어오면 **두 개의 파이프라인**으로 동시에 복제되어 흐릅니다.
- **1. 배치 레이어 (Batch Layer)**: (느리지만 정확함)
  - 모든 원본 데이터를 저장하고 주기적으로 배치 처리하여 '정확한 과거 데이터 뷰(Batch View)'를 생성합니다.
- **2. 스트리밍 레이어 (Streaming Layer)**: (빠르지만 근사치일 수 있음)
  - 실시간 데이터를 빠르게 처리하여 '최신 데이터 뷰(Real-time View)'를 생성합니다.
- **3. 서빙 레이어 (Serving Layer)**:
  - 사용자가 쿼리를 보내면, 'Batch View'와 'Real-time View'의 결과를 **조합(Reconciliation)**하여 최종 결과를 반환합니다.

- **장점**: 속도와 신뢰성의 균형을 맞출 수 있습니다.
- **단점 (치명적)**:
  - **코드 중복**: 동일한 비즈니스 로직을 배치용과 스트리밍용으로 두 번 개발해야 합니다.
  - **복잡성 및 비용**: 두 개의 파이프라인을 유지보수해야 하므로 매우 복잡하고 비용이 많이 듭니다.
  - **결과 일치 문제 (Reconciliation Headache)**: 배치 결과와 스트리밍 결과가 정확히 일치하도록 보장하는 것이 매우 어렵습니다.

6.2 카파 아키텍처 (Kappa Architecture) - 신세대

카파 아키텍처는 람다 아키텍처의 복잡성을 해결하기 위해 등장했으며, 현대 스트리밍 프레임워크가 성숙하면서 표준으로 자리 잡았습니다.

카파 아키텍처의 핵심 아이디어: "모든 것을 스트림으로 취급한다."

□ 핵심 요약

카파 아키텍처 흐름

- 데이터 파이프라인이 **오직 하나(스트리밍 레이어)**만 존재합니다.
- **1. 스트리밍 레이어 (Streaming Layer):**
  - 모든 데이터(실시간이든 배치든)를 스트림으로 간주하고 처리합니다.
- **2. 서빙 레이어 (Serving Layer):**
  - 스트리밍 레이어에서 처리된 결과를 받아 사용자에게 제공합니다.

- **장점:**
  - **단순함:** 단 하나의 코드베이스와 파이프라인만 관리하면 됩니다.
  - **확장성:** 수평 확장이 용이합니다.
  - **결과 일치성:** 결과 일치 문제(Reconciliation)가 원천적으로 발생하지 않습니다.

6.3 FAQ: 카파 아키텍처에서 배치는 어떻게 처리하나요?

□ 핵심 요약

Q: 카파 아키텍처는 스트리밍 전용인가요? 일(Day) 단위 배치 작업은 어떻게 하나요? A: 카파 아키텍처에서도 배치 작업을 완벽하게 수행할 수 있습니다.
배치(Batch)는 그저 "처리 간격(Trigger Interval)이 매우 긴 스트림"일 뿐입니다.
예를 들어, Spark에서 `spark.read`(배치 API) 대신 `spark.readStream`(스트리밍 API)을 사용하되, 트리거 옵션을 `.trigger(once=True)` 또는 `.trigger(processingTime='24 hours')`로 설정하면 됩니다.
이렇게 하면 **디자인은 스트리밍**이지만(`readStream`), **동작은 배치**처럼(하루 한 번 실행) 됩니다.
이점: 스트리밍 API를 사용하면 **체크포인트(Checkpoint)** 기능이 활성화됩니다. 플랫폼이 알아서 "어떤 파일을 처리했고, 어떤 파일을 처리하지 않았는지" 상태를 관리해 줍니다. 따라서 개발자가 수동으로 파일 처리 여부를 추적하는 로직을 만들 필요가 없어집니다.

7 스트리밍 처리 프레임워크

다양한 스트리밍 처리 프레임워크가 있으며, 각기 다른 장단점을 가집니다.

Table 5: 주요 스트리밍 처리 프레임워크 비교

프레임워크	처리 모델	지연 시간(Latency)	상태 관리	내결함성	비고 / 최고 장점
Kafka Streams	이벤트 단위(Event-at-a-time)	매우 낮음 (<10ms)	RocksDB (로컬)	Kafka 로그	**Kafka와 완벽 통합**, 경량 라이브러리
Apache Flink	**진짜 스트리밍 (True streaming)**	**매우 낮음 (ms)**	고급 (대용량 상태)	**강력함 (Exactly-once)**	**초저지연, 복잡한 상태 처리(CEP)**에 최강
Spark Structured Streaming	**마이크로배치 (기본값)**	높음 (100ms 초)	체크포인트	강력함 (Exactly-once)	**배치+스트리밍 통합 API**, Spark 생태계
Apache Storm	이벤트 단위(Event-at-a-time)	매우 낮음 (ms)	제한적	기본 (Acks)	레거시(Legacy), 현재는 거의 사용 안 함
Apache Samza	이벤트 단위(Event-at-a-time)	낮음	RocksDB	Kafka/YARN	LinkedIn에서 개발, 현재는 Niche
KSQLDB	Kafka 기반 연속 SQL	낮음	내부 관리	Kafka 상속	Kafka 데이터를 **SQL로 쉽게 스트리밍**
Amazon Kinesis	마이크로배치	중간	관리형	관리형	AWS 네이티브 앱, IoT
Google Dataflow (Beam)	통합 배치 + 스트리밍	낮음 중간	고급 (Stateful)	강력함	GCP 네이티브, Apache Beam 기반 (이식성)
Azure Stream Analytics	SQL 기반 스트리밍	중간	제한적	관리형	Azure 네이티브, 로우코드(Low-code)

□ 핵심 요약

프레임워크 선택 가이드 (Rule of Thumb)

- **초저지연** 및 복잡한 이벤트 처리가 필요하다면? → **Apache Flink**
- **배치와 스트리밍을 통합**하고 Spark/Lakehouse 생태계를 활용한다면? → **Spark Structured Streaming**
- 이미 **Kafka**를 사용 중이고 간단한 처리가 필요하다면? → **Kafka Streams** 또는 **KSQLDB**
- 특정 클라우드(AWS, GCP, Azure)에 종속적이어도 된다면? → Kinesis, Dataflow, Azure Stream Analytics

8 스트리밍 아키텍처의 트레이드오프

8.1 균형잡기: 반복적 프로세스

스트리밍 파이프라인 설계는 "한 번에 완벽하게"가 아니라, 요구사항과 비용 사이의 균형을 맞추는 **반복적인 프로세스**입니다.

1. **목표(Goals) 이해**: 비즈니스 요구사항(SLA)을 명확히 합니다. (얼마나 빨라야 하는가? 데이터 정확도는?)
2. **전략(Strategy) 정의**: 목표 달성을 위한 기술적 접근법을 정의합니다. (Flink? Spark? Watermark 10초? 10분?)
3. **자원(Resources) 정의**: 전략에 필요한 자원을 산출합니다. (VM 스펙, 클러스터 규모)
4. **비용(Cost) 계산**: 해당 자원의 비용을 계산합니다.
5. **트레이드오프(Tradeoffs) 재검토**: 비용이 너무 높다면 목표를 조정하거나(예: 실시간 → 준실시간), 전략을 변경합니다. (1번으로 복귀)

결국 스트리밍 설계는 **성능(Performance)**, **품질(Quality)**, **확장성(Scalability)**이라는 목표와 **컴퓨팅(Compute)**, **스토리지(Storage)**, **개발/운영 노력(Effort)**이라는 자원 간의 줄다리기입니다.

8.2 실전 시나리오와 디버깅

스트리밍 파이프라인 운영 시 흔히 겪는 문제와 해결책입니다.

주의사항

시나리오 1: 비용 문제 (네트워크 위협 탐지)

- **증상**: VM(컴퓨팅) 비용(30%)보다 **스토리지 비용(70%)이 비정상적으로 높게** 나옴.
- **원인 1 (작은 파일 문제)**: 스트리밍 볼륨이 너무 낮아(Low frequency), **아주 작은 파일(Small files)**이 대량으로 생성됨. 클라우드 스토리지(S3, ADLS)는 파일 개수가 많으면 API 호출 비용(LIST, GET 등)이 급증합니다.
- **원인 2 (스토리지 티어)**: 데이터를 'Cool' 스토리지(저장 비용은 싸지만 **접근 비용은 비쌈**)에 보관하고, 이 데이터를 자주 접근하는 파이프라인을 실행함. (활성 데이터는 'Hot' 티어에 있어야 함)
- **해결**: 처리 트리거 간격을 늘려 파일 크기를 키우고, 활성 데이터는 Hot/Warm 스토리지에 보관.

시나리오 2: 멀티 테넌시 (Multi-Tenancy)

- **목표**: 여러 고객사(Tenant)의 데이터를 격리하여 처리.
- **잘못된 전략**: 고객사별로 별도의 클러스터와 잡(Job)을 생성함.
- **문제**: 고객사가 100개면 클러스터 100개가 필요. **비용이 폭증**하고 리소스 활용률이 낮으며 운영이 복잡해짐.
- **권장 전략**: **하나의 대형 클러스터**에서 모든 고객 데이터를 처리하되, 데이터를 저장할 때 **'partitionBy("client$_id$")'** . (DB) $(View)$.

주의사항

시나리오 3: 중복 제거(Deduplication)와 OOM (메모리 초과)

- **증상:** '$\text{stream}_df.dropDuplicates(\text{"}user_id\text{"})$' , **Out-Of-Memory(OOM) 오류** .
- **원인:** `dropDuplicates` 는 중복을 확인하기 위해 **지금까지 본 모든** $user_id$ $(State)$. $(Statestore)$.
- **해결 (워터마크 사용): 워터마크(Watermark)를 사용하여 상태를 제한해야 합니다.**

```
# Watermark: "ET" 컬럼기준분  10 지연까지허용
# 분이10 지난데이터의상태  (state)는 정리(clean up)됨
uniqueVisitors = stream_df \
  .withWatermark("ET", "10 minutes") \
  .dropDuplicates("ET", "uid")
```

- **설명: 워터마크는 "10분 늦게 오는 데이터까지만 중복을 체크하고, 그보다 더 늦으면 무시하겠다"는 의미입니다. 이를 통해 Spark는 10분이 지난 상태(State) 정보를 메모리에서 삭제할 수 있게 되어 OOM을 방지합니다.**
- **대안 (더 안정적): `foreachBatch`와 `MERGE INTO` (Delta Lake 기능)를 사용하면 속도는 조금 느릴 수 있으나, 상태 관리가 필요 없어 더 안정적으로 중복 제거(Idempotent writes)가 가능합니다.**

8.3 스트리밍 모범 사례 (Best Practices)

- **SLA 및 요구사항 이해**: 가장 중요합니다. TPS(초당 트랜잭션 수), E2E 지연 시간 요구사항을 명확히 합니다.
- **항상 체크포인트(Checkpoint) 사용**: 내결함성(Fault Tolerance)과 복구를 위해 필수입니다.
- **처리 트리거 간격(Processing Trigger Interval) 조절**: 비용 최적화를 위해 사용합니다. (예: 1분마다, 1시간마다). 실시간이 필요 없다면 24/7 실행할 필요가 없습니다.
- **쓰기 최적화(Optimized Writes)**: 작은 파일 문제를 피하기 위해 `delta.autoOptimize.optimizeWrite = true` 같은 옵션을 사용하여 파일 압축(Compaction)을 수행합니다.
- **용량 계획 (Capacity Planning)**: Mux-Demux 아키텍처(모든 데이터를 Bronze에 모으고, 이후에 분리) 등을 고려하여 클러스터 용량을 계획합니다.

9 실습: 네트워크 로그 분석 (Lab 02: Network Analysis)

이 실습에서는 원시(Raw) Apache 네트워크 로그 파일을 Spark Structured Streaming을 사용하여 파싱, 정제, 보강(Enrich)하고 Delta Lake 테이블로 저장하는 전 과정을 다룹니다.

9.1 실습 목표 및 준비

- **목표**: 비정형(Unstructured) 텍스트 로그를 정형(Structured) 데이터로 변환하고, 외부 메타데이터와 조인하여 분석 가능한 델타 테이블을 생성합니다.
- **준비**: 실습을 위한 카탈로그, 스키마, 볼륨을 생성합니다.

1. 준비: 스키마 및 볼륨 생성 (Idempotent)

```
-- 이명령은여러번실행해도안전합니다멱등성 ()
CREATE CATALOG IF NOT EXISTS csci_e103;
USE CATALOG csci_e103;

CREATE SCHEMA IF NOT EXISTS lab02;
USE SCHEMA lab02;

CREATE VOLUME IF NOT EXISTS lab02_input;
CREATE VOLUME IF NOT EXISTS lab02_output;
```

9.2 1단계: 원시 데이터 읽기 (Read Raw Data)

먼저, 텍스트 파일을 그대로 읽어옵니다.

2. 원시 로그 파일 읽기

```
# databricks-에datasets 있는샘플로그경로
log_files_path = "/databricks-datasets/structured-streaming/events/"

# 텍스트파일로읽기
raw_log_files_df = spark.read.text(log_files_path)

# raw_log_files_df.printSchema()
# root
#  |-- value: string (nullable = true)
```

결과: DataFrame은 `value`라는 단 하나의 문자열(String) 컬럼을 가지며, 각 행에는 로그 한 줄이 통째로 들어가 있습니다.

9.3 2단계: 데이터 파싱 (Parse Data)

정규식(Regular Expression)을 사용하여 비정형 문자열을 여러 개의 구조화된 컬럼으로 분리합니다.

3. 정규식을 사용한 로그 파싱

```
from pyspark.sql.functions import regexp_extract

```

```
# Apache 로그형식을파싱하기위한정규식
log_pattern = r'(\S+) (\S+) (\S+) \[(.*?)\] "(\S+) (\S+) (\S+)" (\d{3}) (\d+)'

# 를regexp_extract 사용하여각그룹을컬럼으로추출
parsed_df = raw_log_files_df.select(
    regexp_extract('value', log_pattern, 1).alias('ip'),
    regexp_extract('value', log_pattern, 4).alias('timestamp'),
    regexp_extract('value', log_pattern, 5).alias('method'),
    regexp_extract('value', log_pattern, 6).alias('endpoint'),
    regexp_extract('value', log_pattern, 8).alias('response_code'),
    regexp_extract('value', log_pattern, 9).alias('content_size')
)
```

9.4 3단계: 타임스탬프 변환 (UDF 사용)

주의사항

문제 발생: 비표준 타임스탬프 파싱된 `timestamp` 컬럼 (예: `28/Sep/2025:10:00:00 +0000`)은 표준 SQL 형식이 아닙니다.
`CAST(timestamp AS TIMESTAMP)`를 실행하면 변환에 실패하여 **NULL**을 반환합니다.

□ 핵심 요약

해결: UDF (User-Defined Function) 사용 표준 함수로 처리할 수 없는 복잡한 로직을 위해 **사용자 정의 함수(UDF)**를 만듭니다.

1. 먼저 Python 함수를 정의합니다.
2. 이 함수를 Spark SQL에서 사용할 수 있도록 **SQL UDF로 등록**합니다.

4. UDF 정의 및 등록

```
import datetime

# 1. 비표준형식을파싱하는 Python 함수정의
def parse_datestr(datestr):
    dt = datetime.datetime.strptime(datestr, '%d/%b/%Y:%H:%M:%S %z')
    return dt.timestamp() # Unix timestamp (float)로 반환

# 2. Python 함수를 SQL 로UDF 등록
# 이름 : "parse_state", 반환타입 : "float"
spark.udf.register("parse_state", parse_datestr, "float")
```

이제 `parse_state`라는 함수를 SQL 쿼리 내에서 자유롭게 사용할 수 있습니다.

9.5 4단계: 데이터 보강 (Join Dimension)

로그 데이터의 `response_code` (예: 200, 404)는 숫자일 뿐 의미를 알 수 없습니다. 이 코드의 의미(예: "OK", "Not Found")가 담긴 외부 '차원 테이블(Dimension Table)'과 조인하여 데이터를 보강(Enrich)합니다.

5. 차원 테이블(JSON) 로드 및 조인

```
# 1. HTTP 상태코드메타데이터 (JSON 파일) 로드
status_codes_path = "..." # (JSON 파일경로 )
status_codes_df = spark.read.json(status_codes_path, multiLine=True)
# status_codes_df.printSchema()
#  |-- code: string
#  |-- description: string
#  |-- ...

# 2. 를UDF 사용하여로그데이터정리
clean_logs_df = spark.sql("""
  SELECT
    ip,
    CAST(parse_state(timestamp) AS TIMESTAMP) AS event_time,
    method,
    endpoint,
    response_code,
    CAST(content_size AS int)
  FROM log_data_view -- (를parsed_df 미리 Temp 로View 생성)
""")

# 3. 로그데이터와상태코드조인
network_df = clean_logs_df.join(
  status_codes_df,
  #
  # [!!! 중요] 데이터타입일치시키기
  #
  # clean_logs_df.response_code (int)
  # status_codes_df.code (string)
  # -> 이대로조인하면실패하므로 , 타입을맞춰야함 .
  #
  clean_logs_df.response_code.cast("string") == status_codes_df.code,
  "left_outer"
)
```

주의사항

[실무 핵심] 조인(Join) 전 데이터 타입 확인 조인가 실패하는 가장 흔한 원인은 키(Key)의 데이터 타입 불일치입니다. (예: `Integer` vs. `String`)
위 예제에서 로그의 `response_code`는 `int` 타입(예: 200)이었지만, 참조용 JSON 파일의 `code`는 `string` 타입(예: "200")이었습니다.
`.cast("string")`을 사용하여 타입을 강제로 일치시켜야만 조인이 성공합니다.

9.6 5단계: 필터링 및 저장 (Save to Delta)

최종적으로 정제되고 보강된 데이터를 분석용 델타 테이블로 저장합니다.

6. 최종 데이터 필터링 및 Delta Lake로 저장

```
# 1. 원하는조건으로데이터필터링예     (: 성공한대용량요청   )
final_df = network_df.filter(
    (network_df.response_code == 200) &
    (network_df.content_size > 100)
)

# 2. Delta Lake 테이블로저장덮어쓰기   ( 모드)
final_df.write \
    .format("delta") \
    .mode("overwrite") \
    .saveAsTable("network_filter")
```

9.7 6단계: 확인 (Verify)

저장이 완료되면, Databricks의 'Catalog Explorer'에서 `lab02` 스키마 아래에 `network_filter` 테이블이 생성된 것을 확인할 수 있으며, SQL로 즉시 조회할 수 있습니다.

```
SELECT * FROM csci_e103.lab02.network_filter LIMIT 10;

SELECT description, COUNT(*)
FROM csci_e103.lab02.network_filter
GROUP BY description;
```

Listing 1: 최종 테이블 조회

10 빠른 훑어보기 (1-Page Summary)

□ 핵심 요약

배치(Batch) vs. 스트리밍(Streaming)

- **배치 (Batch)**: "대량의 빨래를 모아서" 한 번에 처리.
- **처리 단위**: 대용량 데이터 묶음 (Bulk).
- **실행 시점**: 주기적 (Scheduled) (예: 매일 밤).
- **주요 용도**: 일일 리포트, 급여 정산.
- **스트리밍 (Streaming)**: "물이 흐르는 즉시" 처리.
- **처리 단위**: 개별 이벤트 (Event-driven).
- **실행 시점**: 연속적 (Continuous) 또는 마이크로배치 (예: 1초마다).
- **주요 용도**: 사기 탐지, 실시간 대시보드.

□ 핵심 요약

람다(Lambda) vs. 카파(Kappa) 아키텍처

- **람다 (Lambda)**: **파이프라인 2개** (배치 + 스트리밍).
- **장점**: 과거 기술로 속도와 신뢰성 동시 추구.
- **단점**: **결과 일치(Reconciliation) 문제** 발생. 코드 중복 및 복잡성.
- **카파 (Kappa)**: **파이프라인 1개** (스트리밍).
- **핵심**: "배치도 결국 트리거가 긴 스트림이다." (`spark.readStream`)
- **장점**: 단순함 (단일 코드베이스), 결과 일치 문제 원천 차단.

주의사항

스트리밍 핵심 개념 Top 2

- **체크포인트 (Checkpoint)**: "게임 저장 지점".
- **역할**: 장애 발생 시 어디부터 다시 시작할지 알려줌. 데이터 유실 및 중복 방지 (Exactly-once). **스트리밍의 필수 요소.**
- **워터마크 (Watermark)**: "지연 데이터 허용 시간" (예: 10분).
- **역할**: 집계 연산 시 **상태(State)가 무한정 커지는 것을 방지**함. 10분이 지난 상태 정보는 메모리에서 삭제하여 **OOM(메모리 초과) 오류를 막음.**

실습(Lab) 핵심 요약

- **UDF (User-Defined Function)**: `CAST` 등 표준 함수로 변환할 수 없는 비표준 형식(예: 타임스탬프)을 처리하기 위해 사용자 정의 함수를 만들어 사용.
- **Join 타입 매칭**: 조인(Join)을 수행할 때는 두 DataFrame의 키(Key) 컬럼 **데이터 타입이 일치하는지 반드시 확인**해야 함. (예: `int` 200 vs. `string` "200" → `cast("string")` 필요)