CSCI E-103: Data Engineering for Analytics
Lecture 03: Data Pipelines, ETL, and Streaming

Harvard Extension School

Fall 2024

- **Course:** CSCI E-103: Data Engineering for Analytics
- **Lecture:** Lecture 03: Data Pipelines & Streaming
- **Instructor:** Anindita Mahapatra & Eric Gieseke
- **Objective:** Understand data pipelines, ETL/ELT processes, batch vs streaming processing, and Lambda/Kappa architectures

Key Summary

This lecture covers the backbone of data engineering: **data pipelines**. We explore different pipeline types (batch, streaming, ETL, ELT), dive deep into streaming concepts (triggers, checkpoints, watermarks), and compare the **Lambda Architecture** (separate batch and streaming layers) with the modern **Kappa Architecture** (unified streaming). We also examine popular streaming frameworks (Kafka, Flink, Spark Structured Streaming) and discuss real-world trade-offs in pipeline design.

Contents

1 Review: Key Concepts from Lecture 02

Table 1: *Data Modeling and Storage Review*

Concept	Key Explanation	Example
3NF (Normalization)	Minimize data redundancy, ensure data integrity	Separate Customer, Order, Product tables
Denormalization	Accept redundancy for faster query performance	Store "last_order_date" in Customer table
ETL / ELT	Extract → Transform → Load vs Extract → Load → Transform	Traditional DWH vs Data Lake
Star Schema	Fact table surrounded by dimension tables (denormalized)	Sales fact + Date, Customer, Product dims
Key-Value Store	Simplest NoSQL: key + value pairs	Amazon S3, Redis
Document Store	Flexible JSON/XML document storage	MongoDB, CouchDB
Columnar Store	Column-oriented for analytical workloads	Cassandra, DynamoDB
Parquet	Binary columnar format, optimized for analytics	Hadoop/Spark ecosystem
Delta Lake	ACID transactions on data lakes (Parquet + transaction log)	Databricks lakehouse
Metadata	Data about data (schema, lineage, history)	Delta transaction log

2 Table Types in Data Platforms

Before diving into pipelines, let's understand the different types of tables you'll encounter:

2.1 Permanent vs Temporary Tables

- **Permanent Tables**: Persist data durably—data survives session/cluster restarts
- **Temporary Tables**: Exist only for a limited scope
  - **Local**: Visible only within the current session
  - **Global**: Shared across sessions within the same cluster

2.2 Views and Materialized Views

Definition:

View (Virtual Table) A stored query definition—not the data itself. Every time you query a view, it re-executes the underlying query.
Pros: Always current data, no storage overhead
Cons: Slower (recomputes every time)

Definition:

Materialized View A view where the query results are **cached/stored**. Results are precomputed and persisted.
Pros: Much faster queries (no recomputation)
Cons: Can return stale data if not refreshed; good for aggregations and BI reports

2.3 Streaming Tables

Definition:

Streaming Table A special table type for streaming pipelines that automatically updates as new data arrives. The table maintains state and incrementally processes incoming events.

2.4 Managed vs External Tables

Table 2: *Managed vs External Tables*

Aspect	Managed Table	External Table
Location	Platform-managed (warehouse path)	User-specified external path
Data ownership	Platform owns data + metadata	Platform owns only metadata
DROP behavior	**Deletes data + metadata**	Deletes only metadata
Sharing	Less portable	More portable/shareable
Risk of loss	Higher (DROP = data gone)	Lower (data remains after DROP)

3 What is a Data Pipeline?

Definition:

Data Pipeline A data pipeline is the complete process of moving data from one point (source) to another (destination). Think of it as ”plumbing” for data—the infrastructure that enables data flow through an organization.

Pipelines can range from simple (copy file A to location B) to complex (ingest from multiple sources, apply multiple transformations, write to multiple destinations).

3.1 ETL Pipelines: A Special Purpose

Definition:

ETL Pipeline A specialized type of data pipeline designed to make data ”analytics-ready.” ETL (Extract, Transform, Load) pipelines:

- **Extract**: Pull data from source systems
- **Transform**: Clean, validate, enrich, aggregate
- **Load**: Write to destination (data warehouse, data mart, ML system)

Key Information

ETL vs ELT

- **ETL**: Transform *before* loading—traditional approach for structured data warehouses
- **ELT**: Load raw data first, transform *later*—modern approach for data lakes where schema may not be known upfront

3.2 Pipeline Lifecycle

Building a data pipeline is like building software:

1. **Design**: Define sources, transformations, destinations
2. **Implement**: Write the pipeline code
3. **Test**: Validate correctness with sample data
4. **Deploy**: Move to production environment
5. **Monitor**: Track execution, detect failures, measure performance
6. **Iterate**: Handle changes, update, redeploy

3.3 Pipeline Triggers

How does a pipeline know when to run?

- **File Arrival**: Triggered when new files appear in a directory
- **Schedule**: Cron-based (e.g., ”every day at 6 AM”, ”every Monday”)

- **Manual**: User explicitly initiates execution
- **Event-Based**: Triggered by external events (API call, message queue)

4 Pipeline Types

Table 3: *Data Pipeline Types*

Type	Processing Style	Tools	Use Cases
Batch	Periodic bulk processing	Spark, AWS Glue	Daily reports, billing
Streaming	Continuous event processing	Kafka, Flink, Kinesis	Fraud detection, IoT
ETL	Extract → Transform → Load	Informatica, Talend, dbt	DWH loading
ELT	Extract → Load → Transform	dbt, Snowflake	Data lake processing
Replication	Sync data between systems	Fivetran, AWS DMS	OLTP → OLAP sync
ML Pipeline	Data prep for ML training/inference	MLflow, Kubeflow	Model training
Orchestration	Coordinate multiple pipelines	Airflow, Prefect	Complex workflows

5 Why Streaming?

Important:

Speed **"It's all about SPEED."**
The goal of streaming is to transform event streams into actionable insights **faster**. When business decisions depend on real-time data, batch processing (hours/days) isn't acceptable.

5.1 The Speed Spectrum

Not every use case requires sub-second latency. Choose the right speed for your business needs:

Table 4: *Latency Requirements by Use Case*

Latency	Type	Use Cases
Hours to Days	**Batch**	ETL, billing, BI reports, ad-hoc analytics
Minutes	**Near Real-Time**	Mobile/IoT ingestion, log aggregation, clickstream
Seconds/Sub-second	**Real-Time**	Fraud detection, trading, gaming, ML inference

Warning

Don't Over-Engineer
Real-time processing is expensive (more compute, more complexity). Ask: "Do we **really** need this in real-time? What action will be taken with the insight?" If the answer is "generate a weekly report," batch is fine.

6 Streaming Concepts

6.1 Core Terminology

Table 5: *Essential Streaming Concepts*

Term	Description
Source & Sink	Every pipeline has a source (where data comes from) and sink (where data goes)
File-Based vs Event-Based	File-based: data lands on disk, then processed (slower). Event-based: data processed from memory/queue (faster, e.g., Kafka)
Micro-batch vs Continuous	Micro-batch: process small chunks periodically (Spark default). Continuous: process each event immediately (Flink)
Trigger	The interval at which micro-batches execute (e.g., every 30 seconds)
Output Modes	How to write to sink: Append (add new rows), Complete (overwrite all), Update (modify changed rows)
Checkpoint	**[Critical]** "Game save point"—records processing progress for fault recovery
Window	Time interval for aggregations (e.g., "sum over last 5 minutes")
Watermark	**[Critical]** How long to wait for late-arriving data before closing a window

6.2 Checkpoints: The Game Save Point

Definition:

Checkpoint A checkpoint records the exact position in the data stream that has been successfully processed. If the pipeline fails, it can restart from the checkpoint rather than reprocessing everything from the beginning.

Why checkpoints matter:

- **Exactly-once semantics**: Ensures data isn't processed twice or dropped
- **Fault tolerance**: Recover gracefully from failures
- **No manual tracking**: The platform manages "what's been processed" automatically

6.3 Watermarks: Handling Late Data

Definition:

Watermark A watermark defines the maximum allowed lateness for data. Data arriving after the watermark threshold is considered "too late" and may be dropped or handled separately.

Example:

Watermark Analogy Imagine a bus that waits 10 minutes past the scheduled departure time for late passengers. After 10 minutes, the bus leaves regardless.
Similarly, a 10-minute watermark means: "Wait up to 10 minutes for late-arriving events. After that, close the aggregation window and move on."

Warning

Watermarks Prevent OOM
Without watermarks, aggregation operations would need to keep state **forever** (in case late data arrives). Watermarks bound the state, preventing memory exhaustion.

7 Lambda vs Kappa Architecture

Warning

Terminology Alert: Lambda Architecture ≠ AWS Lambda
The **Lambda Architecture** discussed here is an **industry-standard architectural pattern** for data pipelines. It has nothing to do with AWS Lambda (Amazon's serverless compute service). The naming is coincidental.

7.1 Lambda Architecture (Old School)

Lambda Architecture was designed when streaming technology was immature. It tried to get the best of both worlds: **batch reliability** and **streaming speed**.

Definition:

Lambda Architecture A data processing architecture with **two parallel paths**:

1. **Batch Layer**: Processes all data periodically (slow but accurate)
2. **Streaming Layer**: Processes real-time data (fast but approximate)
3. **Serving Layer**: Merges results from both layers for queries

Pros:

- Balances speed and reliability

Cons (significant):

- **Code duplication**: Same logic implemented twice (batch + streaming)
- **Complexity**: Two pipelines to maintain
- **Reconciliation nightmare**: Ensuring batch and streaming results match is extremely difficult

7.2 Kappa Architecture (Modern)

Kappa Architecture emerged as streaming frameworks matured and could handle both real-time and batch workloads.

Definition:

Kappa Architecture A simplified architecture with **one unified streaming layer** that handles all data—whether real-time or batch.
Core idea: "Treat everything as a stream."

Pros:

- **Simplicity**: One codebase, one pipeline
- **Scalability**: Horizontal scaling
- **No reconciliation**: Results are inherently consistent

7.3 FAQ: How Does Kappa Handle Batch?

Key Information

Q: If Kappa is streaming-only, how do I run daily batch jobs?

A: Batch is just "streaming with a very long trigger interval."

Instead of `spark.read` (batch API), use `spark.readStream` (streaming API) with:

```
.trigger(processingTime='24 hours')  # Wake up once per day
# or
.trigger(once=True)  # Run once, process all available data
```

The **design** is streaming (using `readStream`), but the **behavior** is batch-like. You still get checkpoint benefits—the platform tracks what's been processed.

7.4 Comparison Table

Table 6: *Lambda vs Kappa Architecture*

Aspect	Lambda	Kappa
Pipelines	Two (batch + streaming)	One (streaming only)
Code duplication	Yes (same logic twice)	No
Complexity	High	Low
Reconciliation	Required (difficult)	Not needed
Batch support	Separate batch layer	Streaming with long trigger
Modern preference	Legacy	**Recommended**

8 Streaming Processing Frameworks

Table 7: *Streaming Framework Comparison*

Framework	Processing Model	Latency	Best For	Notes
Kafka Streams	Event-at-a-time	Very low (<10ms)	Kafka integration	Lightweight Java library
Apache Flink	True streaming	Very low (ms)	Ultra-low latency, CEP	Complex stateful processing
Spark Structured Streaming	Micro-batch (default)	Medium (100ms+)	Unified batch+stream	Spark ecosystem
Apache Storm	Event-at-a-time	Very low	Legacy	Mostly obsolete
Apache Samza	Event-at-a-time	Low	LinkedIn use cases	Niche
KSQLDB	Continuous SQL	Low	SQL over Kafka	Easy Kafka streaming
Amazon Kinesis	Managed streaming	Medium	AWS native	Serverless option
Google Dataflow	Unified batch+stream	Low-Medium	GCP native	Apache Beam based
Azure Stream Analytics	SQL-based streaming	Medium	Azure native	Low-code

8.1 Framework Selection Guide

Key Summary

Rule of Thumb

- **Ultra-low latency + complex event processing?** → Apache Flink
- **Unified batch + streaming with Spark ecosystem?** → Spark Structured Streaming
- **Already using Kafka + simple processing?** → Kafka Streams or KSQLDB
- **Cloud-native, serverless preferred?** → Kinesis, Dataflow, or Azure Stream Analytics

8.2 File-Based vs Event-Based Streaming

- **File-Based**: Data lands on disk (S3, ADLS), then processed
  - Slower (disk I/O)
  - Simpler to implement
  - Spark autoloader, Delta Live Tables

- **Event-Based**: Data flows through message queue, processed in-memory
  - Faster (no disk)
  - More complex
  - Kafka, Kinesis, Event Hubs

9 Spark Structured Streaming

Spark Structured Streaming is the foundation for streaming in the Databricks/Lakehouse ecosystem.

9.1 Basic Streaming Pattern

```python
# Read from stream (note: readStream, not read)
df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "host:9092") \
    .option("subscribe", "topic_name") \
    .load()

# Transform the data
transformed = df \
    .selectExpr("CAST(value AS STRING) as json_data") \
    .select(from_json("json_data", schema).alias("data")) \
    .select("data.*")

# Write to stream
query = transformed.writeStream \
    .format("delta") \
    .option("checkpointLocation", "/path/to/checkpoint") \
    .trigger(processingTime="30 seconds") \
    .start("/path/to/output")
```

Listing 1: Basic Spark Streaming Pipeline

9.2 Key Differences: Batch vs Streaming API

Table 8: *Batch vs Streaming API*

Batch	Streaming
`spark.read`	`spark.readStream`
`df.write`	`df.writeStream`
No checkpoint needed	**Checkpoint required**
Runs once, completes	Runs continuously (or triggered)

9.3 Trigger Options

```python
# Fixed interval micro-batch
.trigger(processingTime="10 seconds")

# Process all available data once, then stop
.trigger(once=True)

# Newer: available-now (process all, checkpoint, stop)
.trigger(availableNow=True)
```

```
9
10 # Continuous (true streaming, experimental)
11 .trigger(continuous="1 second")
```

Listing 2: Trigger Options

10 Trade-offs in Streaming Design

10.1 The Iterative Design Process

Streaming pipeline design is not "design once, build forever." It's an iterative process balancing requirements and costs:

1. **Understand Goals**: What latency does business need? What's the SLA?
2. **Define Strategy**: Which framework? What trigger interval? Watermark settings?
3. **Estimate Resources**: How many VMs? What cluster size? Storage tier?
4. **Calculate Costs**: What's the monthly bill?
5. **Re-evaluate Trade-offs**: If too expensive, can we relax latency requirements?

10.2 The Three Dimensions

- **Scalability**: How much data? What TPS (transactions per second)?
- **Processing**: How many transformations? Joins? Stateful operations?
- **Quality**: Exactly-once semantics? Deduplication? Disaster recovery?

10.3 Real-World Scenarios

Warning

Scenario 1: Storage Costs Higher Than Compute
Symptom: 70% of costs are storage, only 30% compute
Root Causes:

- **Small files problem**: Low-frequency streaming creates many tiny files. Cloud storage charges per API call (LIST, GET), so millions of small files = huge bills.
- **Wrong storage tier**: Active data stored in "Cool" tier (cheap storage, **expensive access**). Should be in "Hot" tier.

Solution: Increase trigger interval to create larger files; keep active data in hot storage.

Example:

Scenario 2: Multi-Tenancy **Goal**: Process data for 100 different customers, isolated from each other.
Bad approach: Create 100 separate clusters and jobs.
Problem: Massive cost, low resource utilization, operational nightmare.
Good approach: One large cluster processing all data, with `partitionBy("client_id")` to physically separate data in storage. Create views per customer in the serving layer.

11 Databricks Lakeflow and Jobs

11.1 Lakeflow Components

Databricks Lakeflow is the end-to-end data engineering platform:

1. **Connect**: Connectors for data ingestion (Oracle, SQL Server, Salesforce, Workday)
2. **Pipelines**: Transformation logic (Medallion architecture, Delta Live Tables)
3. **Jobs**: Workflow orchestration (scheduling, dependencies, monitoring)

11.2 Jobs Features

Table 9: *Databricks Jobs Capabilities*

Feature	**Description**
File Arrival Triggers	Start job when new files appear
Conditional Tasks	If/else branching based on previous task results
Job Parameters	Pass parameters to customize job execution
Webhooks	Notifications to Slack, email, PagerDuty
Duration Alerts	Alert if job exceeds expected runtime
Task Looping	For-each construct for repeated execution
Continuous Execution	Long-running streaming jobs
System Tables	Audit logs of all job executions
Modular Workflows	Call other jobs/workflows

12 Summary and Key Takeaways

Key Summary

Key Takeaways

1. **Data Pipelines** are the "plumbing" that moves data through an organization. ETL pipelines are specialized for analytics readiness.
2. **Pipeline Types**: Batch (periodic), Streaming (continuous), ETL/ELT, Replication, ML, Orchestration
3. **Streaming Concepts**:
   - **Checkpoint**: "Game save point" for fault recovery
   - **Watermark**: How long to wait for late data
   - **Trigger**: When to execute micro-batches
4. **Lambda Architecture**: Two pipelines (batch + streaming) merged in serving layer. Complex, code duplication, reconciliation headache. **Legacy approach.**
5. **Kappa Architecture**: Single streaming pipeline handles everything. Batch = streaming with long trigger. **Modern standard.**
6. **Framework Selection**:
   - Ultra-low latency → Flink
   - Unified batch+stream → Spark Structured Streaming
   - Kafka ecosystem → Kafka Streams or KSQLDB
7. **Trade-offs**: Balance latency, cost, and complexity. Not everything needs real-time processing.
8. **Batch API**: `spark.read` / `spark.write`
   Streaming API: `spark.readStream` / `df.writeStream`

Warning

The Golden Rule

Always ask: "Do we really need real-time?"

Real-time is expensive. If the insight drives a weekly report, batch is fine. Design for business requirements, not technical elegance.