TYPES OF DATA PROCESSING PIPELINES

LATENCY + ENGINEERING COMPLEXITY -

BATCH

ie. A factory processing all day's work at once

Features:

Processed in scheduled intervals

- Full dataset, partition-based processing
- Schedule-driven execution
- Strong DQ capabilities

Latency components:

- **Job scheduling: minutes**
- **Full dataset processing: hours**
- **Typical total: 4-6 hours daily**
- **Usually runs at midnight UTC**

Use when:

- **Daily refresh req, complex DQ needs, resource optimization**

Avoid when:

- **Intraday data needs, real-time, customer-face live analytics**

MICROBATCH

ie. Traditional assembly line, larger batch sizes

Features:

Processed in in shorter scheduled intervals

- Larger batch windows
- Better suited for DQ checks
- Similar tech stack to near real-time

Latency components:

- Job scheduling: minutes
- Full dataset processing: hours
- Typical total: 4-6 hrs
- Usually runs midnight UTC

Use when:

- Robust DQ checks, complex event correlation, high data vol process

Avoid when:

- **Real-time alerting needs, simple tranformations**

NEAR REAL-TIME

ie. A rapid batch assembly line

Features:

Processed in small batches every few minutes

- Small batch collection windows
- Micro-batch processing model
- Better DQ capability vs. pure streaming

Latency components:

- Collection window: 2-5 min
- Processing time: variable
- Writing time: depends on sink

Use when:

- Compute throughout day, complex transformation

Avoid when:

- **ms latency requirements, simple pass-though scenarios**

REAL-TIME

ie. A fast conveyor belt with minimal stops

Features:

Instant processing with inherent latencies

- Near-immediate processing
- Event-time processing
- State management requirements

Latency components:

- **Network time: 100-200ms**
- **Processing overhead: 1-2sec**
- **Total: several minutes**

Use when:

- Customer-facing apps, critical business operations

Avoid when:

- Complex DQ needs, limited team expertise, heavy transformations

STREAMING

ie. A river (water keeps flowing continuously)

Features:

Immediate processing upon generation

- Event-by-event processing
- Continuous state management
- Native watermarking support

Latency components:

- **Event capture: ~150ms**
- **Flink processing: ~100ms-2sec**
- **Sink writing: variable, based on destination**

Use when:

- **critical real-time decision, fraud/security monitor, customer-facing real-time features.**

Avoid when:

- **large state/ complex DQ req**

Spark Spark Structured Stream Flink Kafka

STREAMING DATA PIPELINES

📌 How to consider the use of streaming pipelines

❓ Does the team have the skill set?

Limited team member knowledge limits troubleshooting efforts to a few.

ie. Netflix used to run mostly on batch/ microbatch pipelines. Requirements to have streaming pipelines by the security team to reduce 15min latency to lower latency (2-3min) arguably not justifiable enough

❓ What is the incremental benefit?

If you cannot make it a strong use case (someone key user of your solution, ie. security), avoid streaming.

❓ Homogeneity of your pipelines

Pipeline hetegeneity (a blend of batch, microbatch & streaming) increases maintenance burden & on call overhead.

Going streaming might be a business model decision (ie. Uber uses streaming pipelines on Kappa architecture)

❓ Trade-off between daily, hourly, micro batch & streaming

Review the latency vs. complexity (+ vs. DQ) continuum

❓ How should DQ be inserted?

The lower the latency, the higher the DQ complexity (how to implement DQ on an ever-running pipeline?)

📌 Streaming use cases

Justified cases:

- **Fraud detection**: real-time pattern matching & anomaly detection across multiple transactions within milliseconds using stateful processing to maintain user behavior profiles.
- **Bad behavior prevention**: immediate analysis of user actions against behavioral patterns and rules, with the ability to maintain short-term state for session-based analysis.
- **High-frequency trading**: Sub-millisecond latency processing of market data feeds with complex event processing enables immediate reaction to market conditions and price movements.
- **Live event processing**: real-time aggregation and processing of millions of concurrent events (clicks, views, interactions) with minimal latency, supporting dynamic content adaptation.

Gray area cases:

- **Customer-Facing Data**: Micro-batch offers better data quality with acceptable latency trade-off for customer-facing services, where consistency matters more than absolute real-time delivery
- **Upstream Master Data Processing:** Micro-batch prevents compute spikes in DWH (especially at midnight UTC) and enables better resource distribution & amortization throughout the day. Real case: reduced notification processing from 9h to 1h latency while maintaining stability).

  Notification System Case at 

  While streaming seemed ideal for processing notification events (sent, delivered, generated, clicks), the need to handle duplicates throughout the day required maintaining 20TB of data in memory. With Spark executors requiring 16GB RAM each, pure streaming failed due to OOM errors. Micro-batch solved this memory constraint while maintaining acceptable latency.

STREAMING DATA PIPELINES

📌 Streaming vs. Batch Pipelines

- **Runtime pattern:** Streaming runs 24/7 vs batch runs ~4 hours/day.
- **Architecture behavior:** Streaming pipelines behave like web servers (continuous data flow) rather than DAGs (periodic processing).
- **Reliability concerns:** Higher probability of failures in streaming due to constant operation (analogous to car accident probability increasing with drive time).
- **Testing requirements:** Streaming demands extensive unit/integration testing and complex DQ checks, unlike batch where DQ validation is more straightforward.

📌 Structure of Streaming Pipelines

SOURCE
(Data Ingestion Layer)

1. Apache Kafka
   - High-throughput streaming
   - Acts like a "fire hose" with unidirectional data flow
   - Optimized for large-scale data streaming
   - Scenarios requiring high scalability
2. RabbitMQ
   - Lower throughput than Kafka but more versatile routing
   - Complex message broker scenarios
   - Enhanced pub/sub capabilities
   - Ideal for bi-directional communication scenarios
3. Dimensional Sources (Side Inputs)
   - Static or slowly changing ref. data
   - Sourced from data lakes (ie. Iceberg), relational db (PostgreSQL)
   - Data enrichment on streaming events
   - Refreshed on configurable cadences
   - Enables real-time fact data denormalization

COMPUTE
(Processing Layer)

1. Apache Flink
   - Specialized in stream processing
   - Advanced features for:
     - Window operations
     - Event sessionization
     - Watermarking
     - State management
     - Exactly-once processing semantics
2. Apache Spark
   - Unified engine for batch and stream processing
   - Strong SQL capabilities
   - Rich ecosystem for data analytics
   - Micro-batch processing model

DESTINATION
(Data Sink Layer)

1. Kafka Topics
   - For stream-to-stream processing
   - Enables pipeline chaining
   - Maintains data in motion
2. Data Lakes (ie. Apache Iceberg)
   - Streaming-friendly table format
   - Supports incremental data updates
   - Solves partition management issues
   - Enables seamless integration between streaming and batch workloads
   - Better alternative to traditional Hive metastore for streaming scenarios
3. PostgreSQL
   - For real-time data serving
   - Suitable for operational analytics
   - Enables immediate data accessibility
   - Good for scenarios requiring ACID compliance

STREAMING DATA PIPELINES

📌 Streaming Challenges

✅ Out of Order Events

Newer data is ahead of older data that is not in the right order. Watermarking to minimize the impact

Watermarking

A timestamp that says "I believe all events with timestamps before this watermark have arrived."

- When a watermark reaches an operator, it's safe to trigger time-based operations for all events before that watermark.
- There's usually a buffer time.
- You give a window where you consider the event can be out of order.

ie. you have a 5-minute window and set a watermark delay of 2 minutes: "wait up to 2 minutes for late events before considering the window complete". Any events arriving more than 2 minutes late will be dropped by default

✅ Recovering from failures (3 mechanisms in Apache Flink)

Offsets

Manage where to resume reading after a failure from interacting with Kafka.

Three main offset strategies:

- Earliest offset: Flink reads ALL existing data in Kafka from the beginning of all partitions.
  Useful to process all historical data.
- Latest offset: Flink only starts reading NEW messages that arrive after the job starts. Useul when you only care about new incoming data.
- Specific timestamp: Flink starts reading from a specific point in time. Useful when recovering from a known failure point or when you need to reprocess data from a specific moment.

Checkpoints

Flink's internal automatic fault tolerance mechanism.

How it works:

- Flink periodically takes a snapshot of the entire distributed pipeline's state
- You can configure checkpoint intervals (e.g., every n seconds)
- Checkpoints store the state internally in Flink's binary format
- When a failure occurs, Flink automatically restores the last successful checkpoint
- This ensures exactly-once processing semantics

Savepoints

More flexible and user-controlled than checkpoints:

Features:

- They're manually triggered (unlike automatic checkpoints)
- More agnostic in terms of storage and format
- Useful for planned maintenance, version upgrades, or A/B testing
- Can be used to restart jobs with different parallelism or on different clusters
- Serve as a more permanent backup of your job's state

✅ Late Arriving data

Closely related to but distinct from watermarking (watermark: "normal" 99% cases; late handling: exceptional 'long tail' cases)

- You need to explicitly define what "too late" means for your use case
- This is typically done by setting an allowedLateness parameter in your window configuration
- As noted in your lecture, this could be 5 minutes, 10 minutes, or any duration that makes sense for your specific use case

Handling mechanisms (in Flink)

- Drop it (default behavior)
- Include it in late firings of the window using allowedLateness()
- Route it to a side output using sideOutputLateData()