UMD DATA605 - Big Data Systems

12.1: Streaming and Real-time Analytics

- **Instructor**: Dr. GP Saggese, gsaggese@umd.edu

Motivation

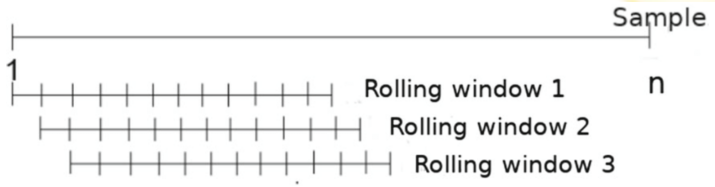
- Big Data is generated as a **continuous, unbounded stream**
- Applications generate data at **high velocity**
  - Financial transactions and market feeds
  - Sensor instrumentation, RFID, IoT telemetry
  - Network and system monitoring
  - Continuous media (video, audio)
- A **data stream** is a time-ordered sequence of events
  - Stream processing treats streams as first-class computational objects
- **Requirements**
  - Ingest and handle high-throughput event streams
  - Low-latency, near-real-time operations (e.g., time-series analytics)
  - Efficient dissemination of relevant subsets to consumers
  - Distributed processing to scale beyond a single machine

Examples of Data Stream Tasks

- Continuous queries
  - Any SQL query can be continuous
  - E.g., *"compute moving average over last hour every 10 mins"*
- Anomaly detection, pattern recognition
  - E.g., *"alert me when A occurs and then B within 10 mins"*
  - Correlate events from different streams
- Statistical tasks
  - E.g., de-noising measured readings
  - Build an online machine learning model
- Process multimedia data
  - E.g., online object detection, activity detection

Why Not Using Standard Solutions?

- **Example**
  - "*Report moving average of XYZ over last hour every 10 minutes*"
- **Solution**
  - Insert arriving items into a relational table
  - Re-run query repeatedly
- **Problems**
- Re-executes full query, not leveraging incremental updates
- Many streaming computations are recursive
- Complex computations may not be easily expressed incrementally
- Real systems may run thousands of continuous queries

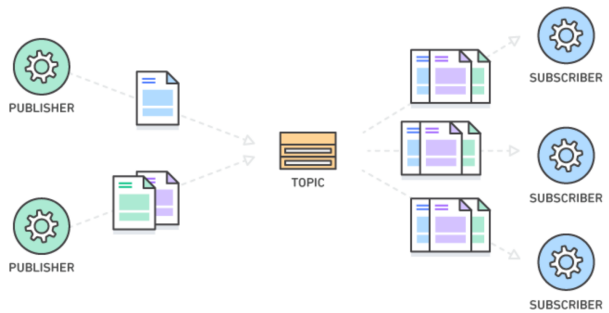


$$m_n = \frac{1}{n} \sum_{i=1}^{n} a_i$$

$$m_n = m_{n-1} + \frac{a_n - m_{n-1}}{n}$$

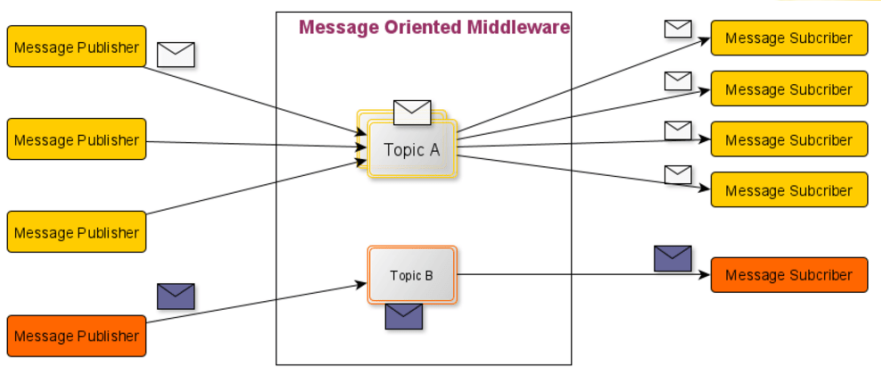
Pub-Sub Systems

- Modern distributed systems use **small, independent components**
  - E.g., serverless architectures, microservices (e.g., Uber)
  - Easier evolution, isolation, scalability
- **Publish-subscribe (pub-sub) systems**
  - Aka "message queues", "message brokers"
  - Connect producers and consumers for event distribution
  - Topics cluster related messages
  - Typically provide lightweight dissemination rather than complex queries
  - Examples: AWS SQS, Kinesis, Kafka, RabbitMQ, Redis Streams, Celery, JBoss



Pub-Sub Systems: Architecture

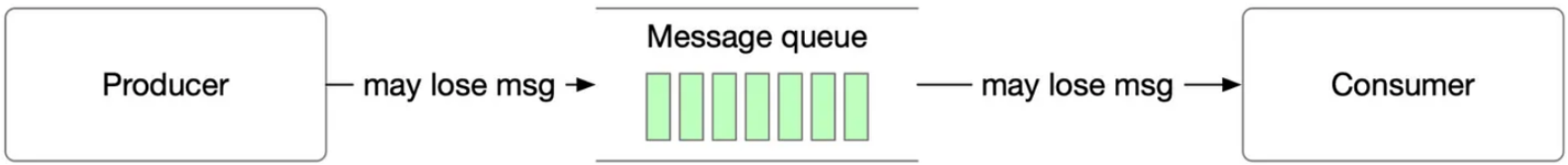
- **Publishers**
  - Send messages or events
- **Subscribers**
  - Consume messages
- **Message broker**
  - Message broker routes event flow between publishers and subscribers, based on topics and subscriptions



- **Design parameters**
  - Event distribution model (topics, filters)
  - Push vs pull consumption
  - Subscriber interest patterns
  - Delivery guarantees
    - At-most-once
    - At-least-once
    - Exactly-once

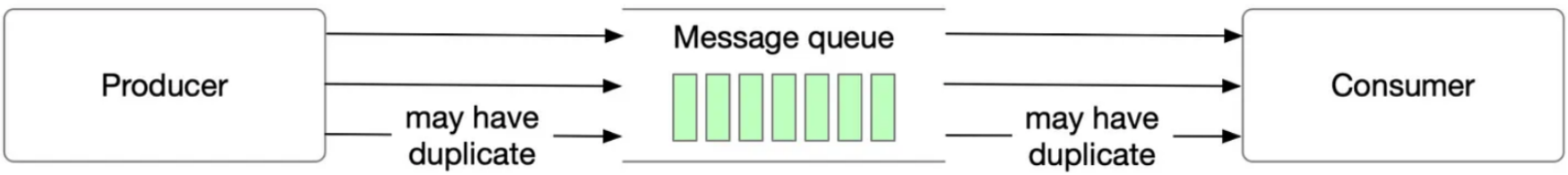
Delivery Semantics: 1 / 3

- **At-most once**
  - Message may be lost, not redelivered
  - Small implementation overhead, high-performance
  - Easy to implement: "fire-and-forget"
  - Works when occasional loss is acceptable
  - E.g., monitoring metrics of website



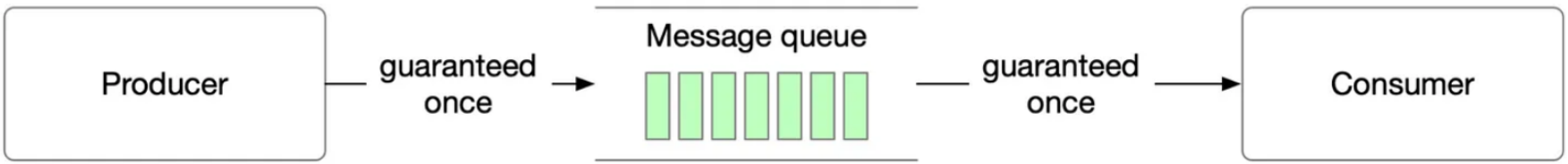
Delivery Semantics: 2 / 3

- **At-least once**
  - Messages retried until acknowledged
  - Ensures no loss but duplicates possible
  - Requires idempotent operations or deduplication



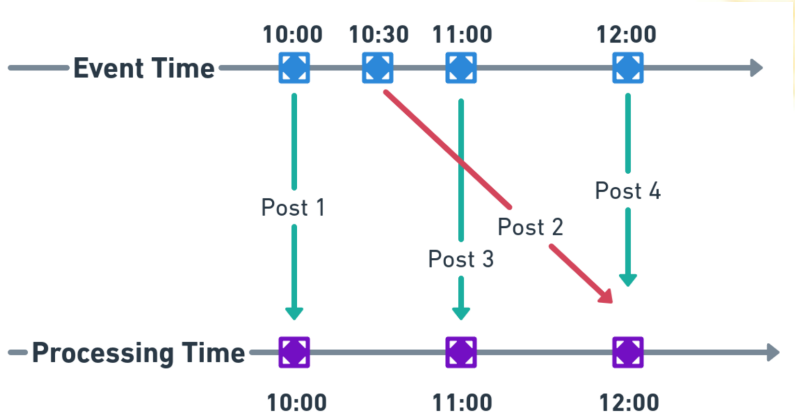
Delivery Semantics: 2 / 3

- **Exactly once**
  - Each message processed once globally
  - Most consumer-friendly but hardest to guarantee
  - Used in financial and mission-critical systems
  - Complicated by distributed coordination limits (e.g., "Two Generals' Problem")
  - E.g., mission-critical systems (e.g., payment, trading, accounting)



Event vs Processing Time

- In both streaming and pub-sub architectures
  - **Event time**
    - Time when each record is generated
  - **Processing time**
    - Time when each record is received
    - Ingestion vs processing time: when events are received vs processed



- **Problems with events**
  - Events may arrive late or out of order
  - Determining how long to wait for stragglers is difficult
  - Systems set bounds on lateness; extremely late data may be dropped or trigger recomputation

Apache Streaming Zoo

- Many different streaming frameworks in the Apache family
  - E.g., Apex, Beam, Flink, Kafka, Spark, Storm, NiFi
  - Built simultaneously at different companies, then open-sourced
- **Different workloads**
  - Real-time analytics, continuous computation
  - Streaming ML, ETL pipelines
  - Messaging and log aggregation
- **Differences arise in**
  - Batch vs streaming orientation
  - Delivery semantics
  - Compute vs pub-sub roles
  - Throughput, latency, fault tolerance
  - API and language support

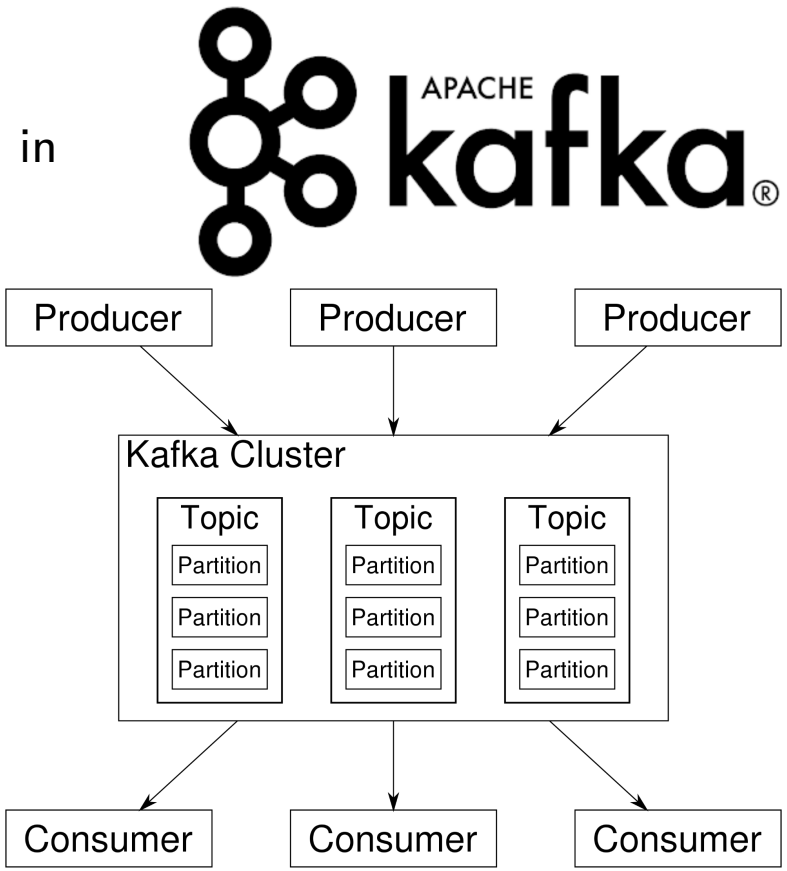
Apache Storm

- Open-source distributed real-time computation system
  - Acquired and open-sourced by Twitter

- **Horizontal scalability**: add machines to handle increasing data
- **Fault tolerance**:
  - At-least-once** processing
  - Automatic task restarts
  - Workload redistribution
- **Directed acyclic graph (DAG)** with:
  - Spouts as data sources (as source nodes)
  - Bolts as processing units (as nodes)
  - Data streams (as edges)
- Suitable for complex data processing workflows with multiple stages and parallelism

Apache Kafka

- Open-source distributed streaming platform
  - Developed at LinkedIn, open-sourced in 2011
- Core components: producers, brokers, consumers, topics, partitions
- Persistent, replicated log storage
- High throughput, low latency
- Delivery: at-least-once, at-most-once, exactly-once
- Kafka Connect for integration with external systems
- Kafka Streams for native stream processing

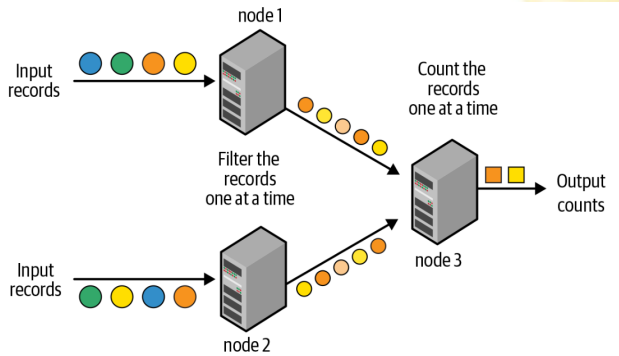


Apache Flink

- Open-source, distributed data processing framework
- Focus on stateful computations over data streams
- Scalability: Horizontal scaling across large clusters
- Fault tolerance:
  - **Exactly-once** processing semantics, checkpointing, state management
- Batch processing support: **unified API for stream and batch processing**
- Flexible windowing: time-based, count-based, session windows
- Deployment options: standalone, YARN, Mesos, Kubernetes, cloud environments

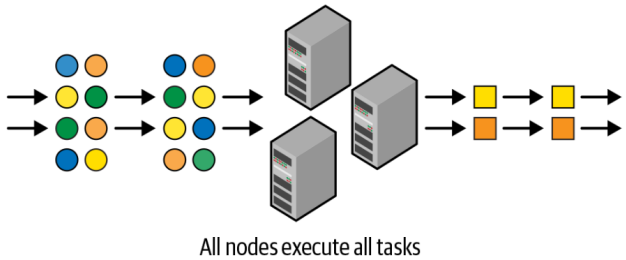
Record-at-a-time Processing

- Implemented in Apache Kafka
- Goal: handle endless data stream
- **Multiple-node distributed processing engine**
  - Map computation on DAG of nodes
  - Each node continuously
    - Receives one record
    - Processes it
    - Forwards to next node
- **Pros**
  - Very low latencies
    - E.g., less than msecs
- **Cons**
  - Inefficient node failure recovery
    - E.g., need failover resources/redundancy
  - Straggler nodes (slower than others)



Micro-Batch Stream Processing

- Aka DStreams
- Implemented in Spark Streaming
- **Computation as a continuous series of batch jobs on small chunks of stream data**
  - E.g., 1 second
  - Process each batch in the Spark cluster in a distributed manner
- **Pros**
  - Recover from failures and stragglers using task scheduling
    - E.g., schedule same task multiple times
  - Deterministic tasks
    - Exactly-once processing guarantees
    - Consistent API: same functional semantics as RDDs
    - Fault-tolerance
- **Cons**
  - Higher latency
    - E.g., seconds

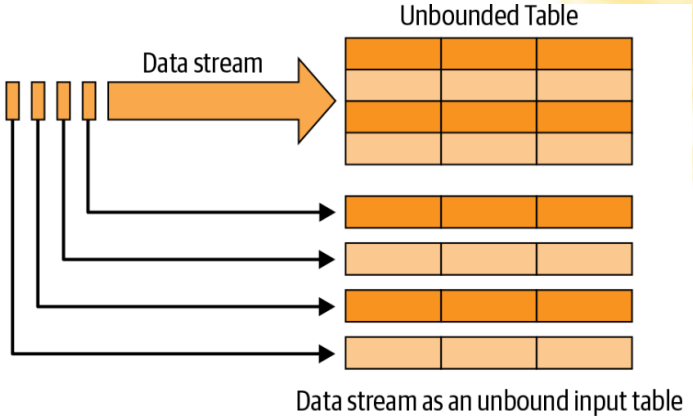


Spark Micro-Batch Processing: Cons

- Line between real-time and batch processing blurred
  - Application computing data hourly: stream or batch?
- No single API for batch and stream processing
  - Same abstractions (RDD) and operations
  - Rewrite code with different classes
- No support for event-time windows
  - Operations defined by processing time
  - No support for tardy data
- Spark replaced DStreams with Structured Streaming in v3
  - Supports micro-batch and continuous streaming
  - Closer batch vs streaming API

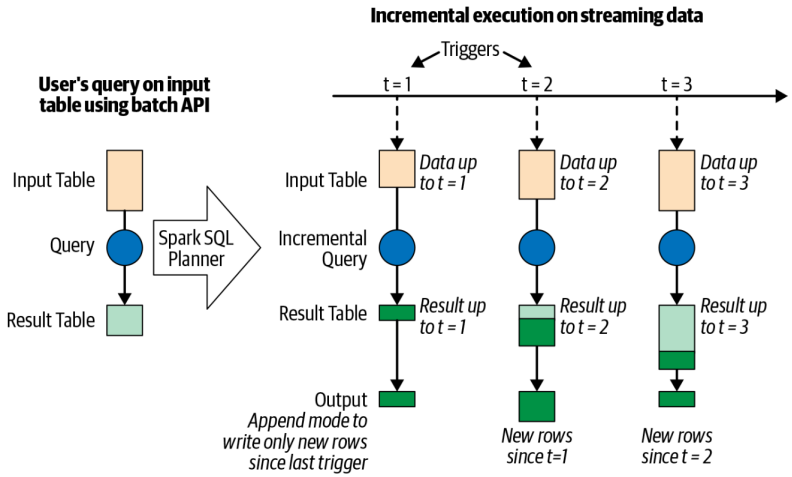
Spark Structured Streaming

- New approach used by Spark
- Goal: write stream processing as easy as writing batch pipelines
  - Single unified programming model
  - Use SQL or DataFrames on stream
- Handle automatically
  - Fault tolerance
  - Optimizations
  - Incremental computation
  - Tardy data
- **Data abstraction**
  - Batch applications: table (DataFrame) is the abstraction
  - Structured Streaming: table is an unbounded, continuously appended table
    - New record becomes a new row appended
    - At time T, it's like a static dataframe with data until T



Incrementalization

- Automatically detect state to maintain
  - Build DAG of computation
  - Express output of graph at time T in terms of graph at time T-1
  - Cache results
- Developers specify triggers to update results
- Incrementally update result with each record arrival



Triggering Modes

- Indicate when to process newly available streaming data
  - **Default**
    - Process micro-batch after previous completes
  - **Trigger interval**
    - Specify fixed interval for each micro-batch
    - E.g., "every 10 minutes"
  - **Once**
    - Wait for external trigger
    - E.g., "at end of day"
  - **Continuous (experimental)**
    - Process data continuously
    - Not all operations available
    - Lower latency

Saving Data

- Each time result table updates, write to external file system (e.g., HDFS, AWS S3) or DB (e.g., MySQL, Cassandra)
  - **Append mode**
    - Append new rows since last trigger
    - Use when existing rows don't change
  - **Update mode**
    - Write updated rows since last trigger
    - Update in place
  - **Complete mode**
    - Write entire updated result table
    - General but expensive

Spark Streaming "Hello world"

- `lines` looks like an RDD but it's a `DataStreamReader`
  - Unbounded DataFrame
  - Set up reading but doesn't start reading
- `words` split data in words
- `counts` is a streaming DataFrame
  - Running word count
- Stateless transformations don't require maintaining state
  - E.g., `select()`, `filter()`
- Stateful transformations
  - E.g., `count()`
- Define how to write processed output
  - Where to write (e.g., `console`)
  - How to write (e.g., `complete` for updated word counts)
- When to trigger computation
  - E.g., every 1 second
- Where to save metadata for:

```
from pyspark.sql.functions import *
spark = SparkSession...
lines = (spark
  .readStream.format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load())

words = lines.select(split(col("value"), "\\s").alias("word"))
counts = words.groupBy("word").count()
checkpointDir = "..."
streamingQuery = (counts
  .writeStream
  .format("console")
  .outputMode("complete")
  .trigger(processingTime="1 second")
  .option("checkpointLocation", checkpointDir)
  .start())
streamingQuery.awaitTermination()
```