UMD DATA605 - Big Data Systems

12.1: Streaming and Real-time Analytics

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

- ***Data Streams***
- Streaming Concepts
- Apache Streaming Zoo
- Processing Styles

Data Streams: Motivation

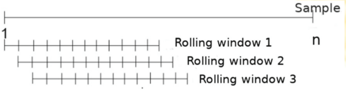
- Big Data is generated as a **continuous, unbounded stream**
- Applications generate data at **high velocity**
  - Financial transactions and market feeds
  - Sensor instrumentation, RFID, IoT telemetry
  - Network and system monitoring
  - Continuous media (video, audio)
- A **data stream** is a time-ordered 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

Data Streams: Examples

- 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



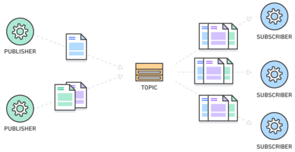
$$m_n = \frac{1}{n}\sum_{i=1}^{n} a_i$$

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

- Data Streams
- ***Streaming Concepts***
- Apache Streaming Zoo
- Processing Styles

Pub-Sub Systems: Motivation

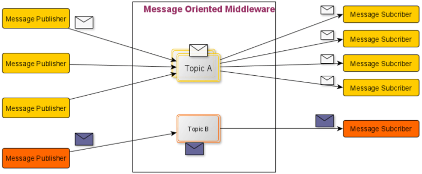
- Modern distributed systems use **small, independent components**
  - E.g., serverless architectures, microservices (e.g., Uber)
  - Easier evolution, isolation, scalability
- **Publish-subscribe (pub-sub) systems**
  - Aka "message queues", "message brokers"
  - Connect producers and consumers for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



SCIENCE ACADEMY

Pub-Sub Systems: Architecture

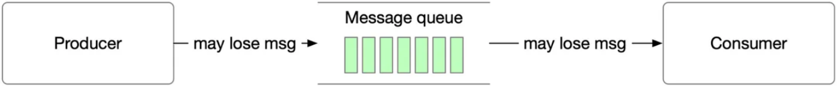
- **Publishers**
  - Send messages or events
- **Subscribers**
  - Consume messages
- **Message broker**
  - Message broker routes event flow between publishers and subscribers, based on topics and subscriptions



- **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: At-most once

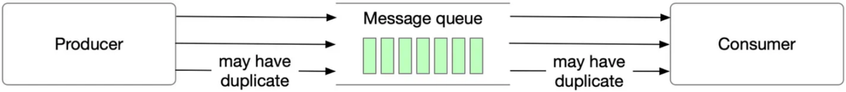
- **At-most once**: *message may be lost, not redelivered*



- **Pros**
  - 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: At-least once

- **At-least once**: messages retried until acknowledged

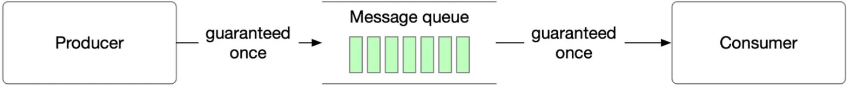


- **Pros**
  - Ensures no loss but duplicates possible
- **Cons**
  - Requires idempotent operations or deduplication

SCIENCE ACADEMY

Delivery Semantics: Exactly once

- **Exactly once**: each message processed once globally

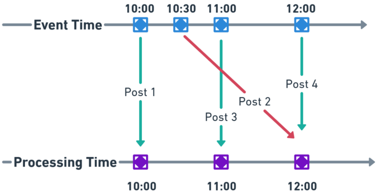


- Most consumer-friendly but hardest to guarantee
  - Complicated by distributed coordination limits (e.g., "Two Generals' Problem")
- Used in financial and mission-critical systems
  - E.g., mission-critical systems (e.g., payment, trading, accounting)

SCIENCE ACADEMY

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 re-computation

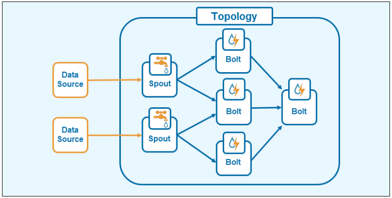
- Data Streams
- Streaming Concepts
- ***Apache Streaming Zoo***
- Processing Styles

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

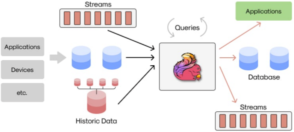
- Open-source distributed real-time computation system
  - Acquired and open-sourced by Twitter
- **Horizontal scalability**: add machines to handle increasing data
- **Directed acyclic graph (DAG)**:
  - Spouts as data sources (as source nodes)
  - Bolts as processing units (as nodes)
  - Data streams (as edges)



- **Fault tolerance**:
  - At-least-once processing
  - Automatic task restarts
  - Workload redistribution
- **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

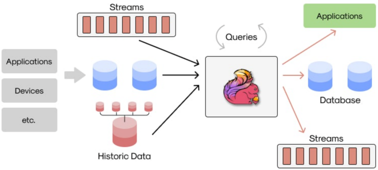


- **Delivery**: at-least-once, at-most-once, exactly-once
- High throughput, low latency
  - Persistent, replicated log storage
- *Kafka Connect* for integration with external systems
- *Kafka Streams* for native stream processing

Apache Flink

- Open-source, distributed data processing framework

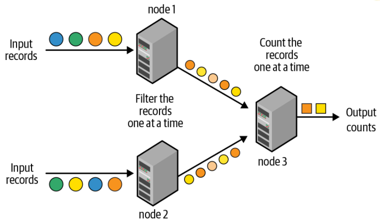
- Distributed processing engine with strong support for stateful streaming
- Exactly-once semantics via checkpointing and robust state management
- Unified API for batch and streaming
- Rich windowing functions
- Runs on standalone clusters, YARN, Mesos, Kubernetes, and cloud



- Data Streams
- Streaming Concepts
- Apache Streaming Zoo
- ***Processing Styles***

Record-at-a-time Processing

- **Designed to handle infinite data streams**
  - Implemented in Apache Kafka
- **Distributed processing over multiple nodes**
  - Nodes organized in a DAG
  - Each node continuously:
    - Receives a single record
    - Processes the record immediately
    - Forwards the output to the next node

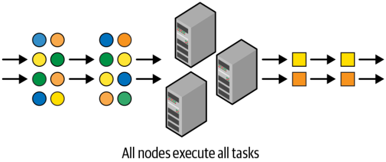


- **Pros**
  - Achieves extremely low latency
    - Example: sub-millisecond response times
- **Cons**
  - Poor fault tolerance
    - Requires extra nodes or redundant paths for failover
  - Sensitive to stragglers
    - Slow nodes can delay the entire pipeline

SCIENCE ACADEMY

Micro-Batch Stream Processing

- Break continuous stream into **small batches** (e.g., 1-second windows)
  - Implemented in Spark Streaming (aka "DStreams")

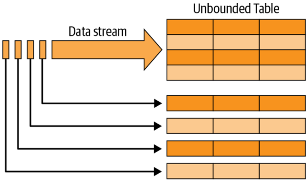


- **Pros**
  - Recover from failures and stragglers with task scheduling
    - Schedule same task multiple times
  - Deterministic tasks
    - Exactly-once processing
    - Consistent API: same semantics as RDDs
    - Fault-tolerance
- **Cons**
  - Higher latency
    - E.g., seconds

SCIENCE ACADEMY

Spark Structured Streaming

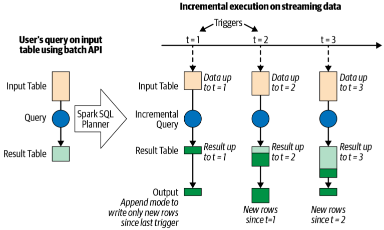
- Unified DataFrame/SQL-based model for both batch and streaming
- System manages state, faults, incremental computation, and late data
- Streaming table abstraction
  - Conceptually an unbounded table continuously appended with new rows
  - At time T, equivalent to a static DataFrame of all rows up to T



Data stream as an unbound input table

Incrementalization

- Framework identifies necessary state across micro-batches
- Uses DAG analysis to compute updated results from prior state
- Developers specify trigger conditions for updates
- Results updated incrementally as events arrive



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 Modes

- Indicate when to save results and where
  - 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` is 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
- `select()`, `filter()` are stateless transformations
- `count()` is stateful transformation

```python
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()
```

- Configuration
  - 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 exactly-once guarantees, failure recovery
- `start()` processing (non-blocking)
  - `awaitTermination()` blocks until data is available