



UMD DATA605 - Big Data Systems

12.1: Streaming and Real-time Analytics

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

Motivation

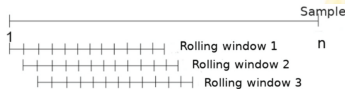
- **Often Big Data is generated continuously**
- Applications grow daily, e.g.,
 - Financial data
 - Sensors, RFID, IoT
 - Network/systems monitoring
 - Video/audio data
- **Data stream is a sequence of events in time**
- **Stream processing is a paradigm for data streams as computation objects**
- Needs to support
 - High data rates
 - Real-time processing with low latencies
 - Time-series operations
 - Data dissemination
 - Send data subset to interested users
 - Distributed operation
 - Single node often insufficient for real-time processing

Examples of Data Stream Tasks

- Any SQL query can be continuous
 - E.g., “compute moving average over last hour every 10 mins”
 - StreamSQL extends SQL to support windows over streams
- Pattern recognition
 - E.g., “alert me when A occurs and then B within 10 mins”
- Surveillance
- Fraud detection and prevention
 - Anti-money laundering
 - RegNMS, MiFID
- Financial modeling
- 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 the Usual RDBM?

- Simple case: “*report moving average over last hour every 10 minutes*”
 - Insert items into DB table as they arrive
 - Execute query every 10 minutes
- One query: slow and inefficient
 - Doesn't reuse previous work
 - Can describe computation incrementally with recursion
- Computations
 - Can be complex (e.g., train LLM model with 1,000 GPUs)
 - Not always easily rewritten recursively
- Typically thousands of continuous queries



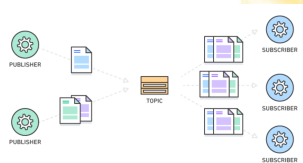
$$m_n = \frac{1}{n} \sum_{i=1}^n a_i$$

Computation

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

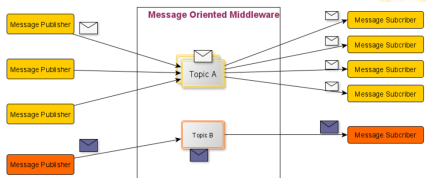
Pub-Sub Systems

- In modern architecture, complex distributed systems are built with **small and independent building blocks**
 - E.g., serverless, micro-services
 - E.g., Uber
 - Facilitate decoupling and allow systems to evolve independently
 - Allow scalability and flexibility
- **Publish-subscribe systems**
 - Aka pub-sub, message queues, message brokers
 - Provide communication and coordination between building blocks
 - Focus on data dissemination
 - Typically no complex queries
 - Topics categorize messages and events
 - E.g., AWS SQS, AWS Kinesis, Apache Kafka, RabbitMQ, Redis, Celery, JBoss



Pub-Sub Systems

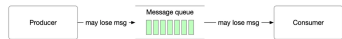
- **Publishers**
 - Send messages or events
- **Subscribers**
 - Consume messages
- **Message broker**
 - Manage message flow between publishers and subscribers
- **Design parameters**
 - Events distributed
 - Topics, by subscription
 - Event distribution method
 - Push, pull
 - Subscriber interest in topics
 - Delivery semantics
 - At-most once
 - At-least once
 - Exactly once



Delivery Semantics

- **At-most once**

- Message may be lost, not redelivered
- Pros
 - High-performance
 - Small implementation overhead
 - Easy to implement: “fire-and-forget”
- E.g., monitoring metrics allow small data loss



- **At-least once**

- Deliver message more than once, no message lost
- Handle transport message loss
 - Keep state at sender
 - Acknowledge state at receiver
- Works if
 - Data duplication is acceptable
 - Deduplication (e.g., storing key-value)
 - Idempotency



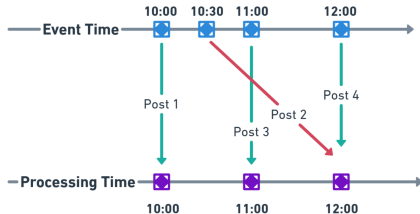
- **Exactly once**

- Every message sent once
- Friendly for downstream consumers
- Difficult to implement
- Two Generals' Problem



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**
 - Out of order
 - Tardiness
 - How long to wait for late data?
 - In an asynchronous system, never sure all data has arrived
 - Use bounds on delay
 - Assume data arrives every minute
 - Recompute once late data arrives? If not, drop late data?



Apache Streaming Zoo

- Many different streaming frameworks
 - Apache Apex, Apache Beam, Apache Flink, Apache Kafka, Apache Spark, Apache Storm, Apache NiFi
- Use cases
 - Real-time analytics
 - Online machine learning
 - Continuous computation
 - ETL processes
 - Data pipeline processing
 - Messaging
 - Log aggregation
- Different solutions to the same problem
 - Batch vs streaming
 - Delivery semantic type
 - Computing vs messaging/pub-sub
 - Throughput vs fault-tolerance
 - Supported languages
- Built simultaneously at different companies, then open-sourced

Apache Storm

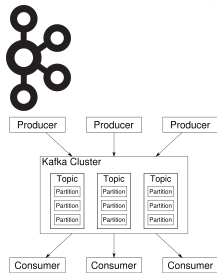
- Open-source distributed real-time computation system
 - Acquired and open-sourced by Twitter
- Horizontal scalability: add machines to handle increasing data
- Fault tolerance: **at-least-once** processing, automatic task restarts, workload redistribution
- Directed acyclic graph (DAG) with:
 - spouts (data sources)
 - bolts (processing units) as vertices
 - data streams as edges
- Suitable for complex data processing workflows with multiple stages and parallelism



APACHE
STORM™

Apache Kafka

- Open-source distributed streaming platform
 - Developed at LinkedIn, open-sourced in 2011
- Producers, brokers, consumers, topics, partitions
- Persistent storage, data replication across brokers
- High-throughput, low-latency messaging
- Fault tolerance: broker replication, automatic recovery
- Message delivery semantics:
 - **At-least-once, at-most-once, exactly-once**
- Kafka Connect: Integrate with data sources and sinks
- Kafka Streams: Stream processing library on Kafka



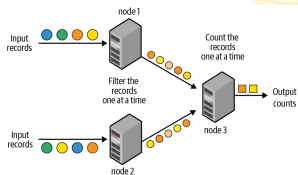
Apache Flink

- 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



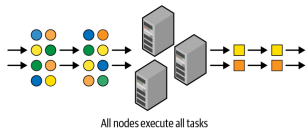
Record-at-a-time Processing

- 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 msec
- **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

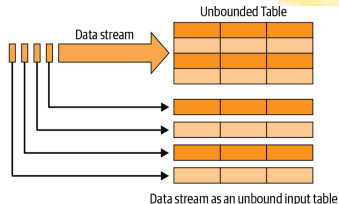


Spark Micro-Batch Processing: Cons

- 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

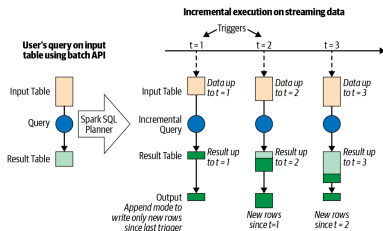
Spark Structured Streaming

- 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()
```