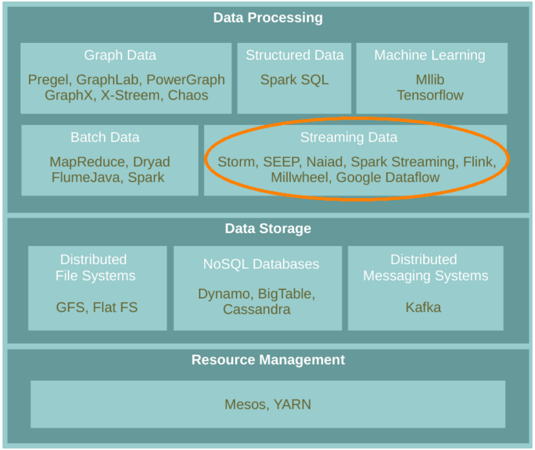
Data Stream Processing

Amir H. Payberah
payberah@kth.se
2025-09-16

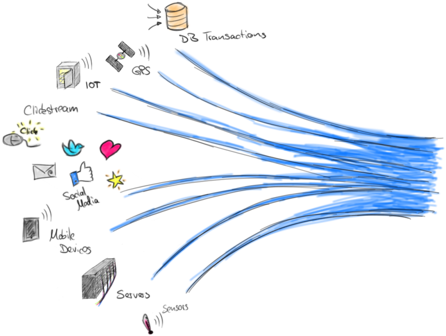
Where Are We?



Data Processing
Graph Data
Pregel, GraphLab, PowerGraph
GraphX, X-Streem, Chaos
Structured Data
Spark SQL
Machine Learning
Mllib
Tensorflow
Batch Data
MapReduce, Dryad
FlumeJava, Spark
Streaming Data
Storm, SEEP, Naiad, Spark Streaming, Flink,
Millwheel, Google Dataflow
Data Storage
Distributed
File Systems
GFS, Flat FS
NoSQL Databases
Dynamo, BigTable,
Cassandra
Distributed
Messaging Systems
Kafka
Resource Management
Mesos, YARN

Stream Processing

- Stream processing is the real-time computation of continuously incoming data.
- The input data is unbounded: a series of events, no predetermined beginning or end.



Streaming Data

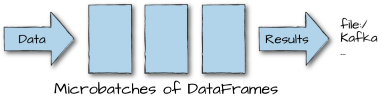
- Data stream is unbound data, which is broken into a sequence of individual tuples.
- A data tuple is the atomic data item in a data stream.

Streaming Data

- Data stream is unbound data, which is broken into a sequence of individual tuples.
- A data tuple is the atomic data item in a data stream.
- Can be structured, semi-structured, and unstructured.

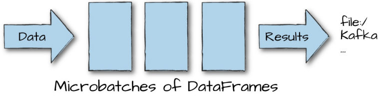
Streaming Processing Patterns

- Micro-batch systems
  - Batch engines
  - Slicing up the unbounded data into a sets of bounded data, then process each batch.

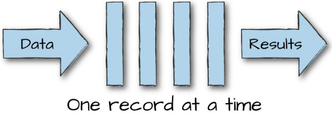


Streaming Processing Patterns

- Micro-batch systems
  - Batch engines
  - Slicing up the unbounded data into a sets of bounded data, then process each batch.



- Continuous processing-based systems
  - Each node in the system continually listens to messages from other nodes and outputs new updates to its child nodes.



Event and Processing Time

Event Time vs. Processing Time

- Event time: the time at which events actually occurred.
  - Timestamps inserted into each record at the source.

- Prcosseing time: the time when the record is received at the streaming application.

Windowing and Triggering

- Windowing: dividing a continuous stream of data into discrete chunks (windows).

Windowing and Triggering

- **Windowing**: dividing a continuous stream of data into discrete chunks (windows).
  - Count-based, Time-based, etc.

Windowing and Triggering

- Windowing: dividing a continuous stream of data into discrete chunks (windows).
  - Count-based, Time-based, etc.

- Triggering: defines when a computation within a window should be performed.

Windowing and Triggering

- Windowing: dividing a continuous stream of data into discrete chunks (windows).
  - Count-based, Time-based, etc.

- Triggering: defines when a computation within a window should be performed.
  - Count-based policy: the maximum number of tuples a window buffer can hold

Windowing and Triggering

- Windowing: dividing a continuous stream of data into discrete chunks (windows).
  - Count-based, Time-based, etc.
- Triggering: defines when a computation within a window should be performed.
  - Count-based policy: the maximum number of tuples a window buffer can hold
  - Time-based policy: based on processing or event time period

Windows After Triggering

- Two possibilites: tumbling and sliding

Windows After Triggering

- Two possibilites: tumbling and sliding
- Tumbling window: when the buffer fills up, all the tuples are evicted.

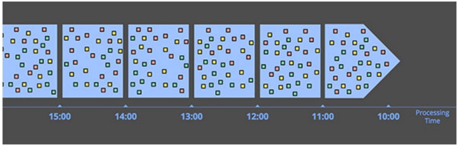
Windows After Triggering

- Two possibilites: tumbling and sliding
- Tumbling window: when the buffer fills up, all the tuples are evicted.

- Sliding window: when the buffer fills up, older tuples are evicted.

Triggering According to Processing Time

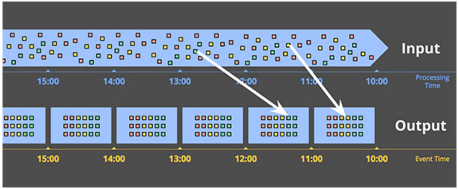
- The system buffers up incoming data into windows until some amount of processing time has passed.
- E.g., five-minute fixed windows



[https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101]

Triggering According to Event Time (1/2)

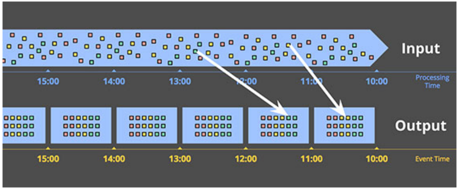
- Reflect the times at which events actually happened.



[https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101]

Triggering According to Event Time (1/2)

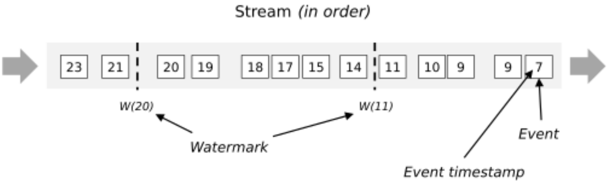
- Reflect the times at which events actually happened.
- Handling out-of-order evnets.



[https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101]

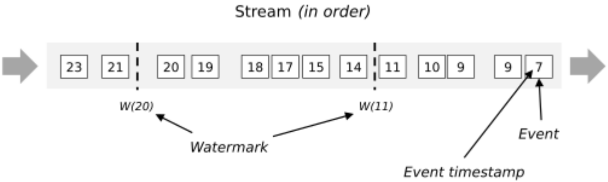
Triggering According to Event Time (2/2)

- Watermarking helps a stream processing system to deal with lateness.



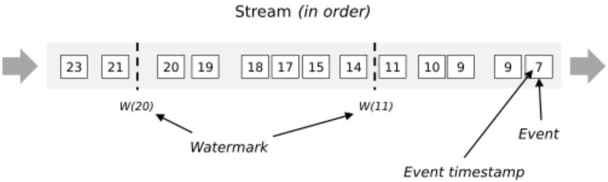
Triggering According to Event Time (2/2)

- Watermarking helps a stream processing system to deal with lateness.
- Watermarks flow as part of the data stream and carry a timestamp `t`.



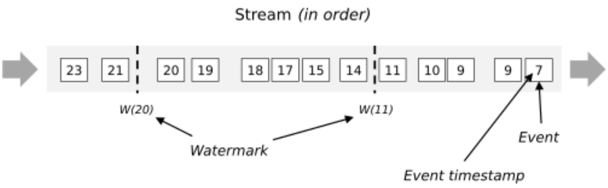
Triggering According to Event Time (2/2)

- Watermarking helps a stream processing system to deal with lateness.
- Watermarks flow as part of the data stream and carry a timestamp `t`.
- A `W(t)` declares that event time has reached time `t` in that stream



Triggering According to Event Time (2/2)

- Watermarking helps a stream processing system to deal with lateness.
- Watermarks flow as part of the data stream and carry a timestamp `t`.
- A `W(t)` declares that event time has reached time `t` in that stream
- A watermark is a threshold to specify how long the system waits for late events.



Spark Streaming

Spark Streaming

- Run a streaming computation as a series of very small, deterministic batch jobs.

Spark Streaming

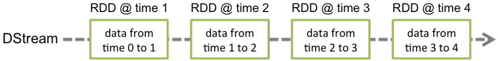
- Run a streaming computation as a series of very small, deterministic batch jobs.
  - Chops up the live stream into batches of X seconds.
  - Treats each batch as RDDs and processes them using RDD operations.

Spark Streaming

- Run a streaming computation as a series of very small, deterministic batch jobs.
  - Chops up the live stream into batches of X seconds.
  - Treats each batch as RDDs and processes them using RDD operations.
  - Discretized Stream Processing (DStream)

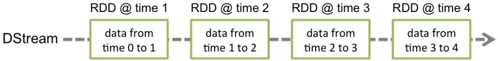
DStream (1/2)

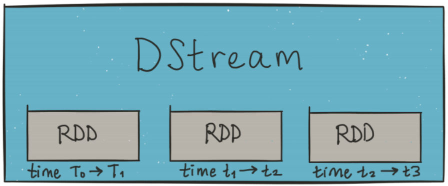
- DStream: sequence of RDDs representing a stream of data.



DStream (1/2)

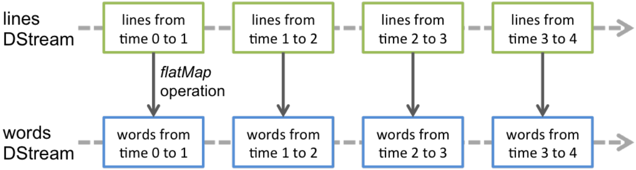
- DStream: sequence of RDDs representing a stream of data.





DStream (2/2)

- Any operation applied on a DStream translates to operations on the underlying RDDs.



StreamingContext

- StreamingContext is the main entry point of all Spark Streaming functionality.

```
val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))
```

- The second parameter, Seconds(1), represents the time interval at which streaming data will be divided into batches.

Input Operations

- Socket connection
  - Creates a DStream from text data received over a TCP socket connection.

```
ssc.socketTextStream("localhost", 9999)
```

Input Operations

- Socket connection
  - Creates a DStream from text data received over a TCP socket connection.

```
ssc.socketTextStream("localhost", 9999)
```

- File stream
  - Reads data from files.

```
streamingContext.fileStream[KeyClass, ValueClass, InputFormatClass](dataDirectory)

streamingContext.textFileStream(dataDirectory)
```

Input Operations

- Socket connection
  - Creates a DStream from text data received over a TCP socket connection.

```
ssc.socketTextStream("localhost", 9999)
```

- File stream
  - Reads data from files.

```
streamingContext.fileStream[KeyClass, ValueClass, InputFormatClass](dataDirectory)

streamingContext.textFileStream(dataDirectory)
```

- Connectors with external sources, e.g., Twitter, Kafka, Flume, Kinesis, ...

Transformations (1/2)

- Transformations on DStreams are still lazy!
- DStreams support many of the transformations available on normal Spark RDDs.

Transformations (1/2)

- Transformations on DStreams are still lazy!
- DStreams support many of the transformations available on normal Spark RDDs.
- Computation is kicked off explicitly by a call to the `start()` method.

Transformations (2/2)

- `map`: a new DStream by passing each element of the source DStream through a given function.

Transformations (2/2)

- `map`: a new DStream by passing each element of the source DStream through a given function.
- `reduce`: a new DStream of single-element RDDs by aggregating the elements in each RDD using a given function.

Transformations (2/2)

- `map`: a new DStream by passing each element of the source DStream through a given function.

- `reduce`: a new DStream of single-element RDDs by aggregating the elements in each RDD using a given function.

- `reduceByKey`: a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function.

Example - Word Count (1/6)

- First we create a `StreamingContex`

```
import org.apache.spark._
import org.apache.spark.streaming._

// Create a local StreamingContext with two working threads and batch interval of 1 second.
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
```

Example - Word Count (2/6)

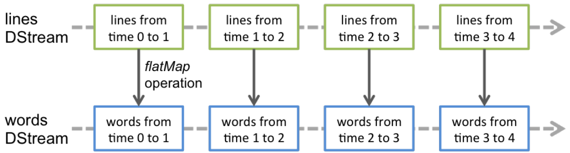
- Create a DStream that represents streaming data from a TCP source.
- Specified as hostname (e.g., localhost) and port (e.g., 9999).

```
val lines = ssc.socketTextStream("localhost", 9999)
```

Example - Word Count (3/6)

- Use `flatMap` on the stream to split the records text to words.
- It creates a new DStream.

```
val words = lines.flatMap(_.split(" "))
```



Example - Word Count (4/6)

- Map the `words` DStream to a DStream of `(word, 1)`.
- Get the frequency of words in each batch of data.
- Finally, print the result.

```
val pairs = words.map(word => (word, 1))

val wordCounts = pairs.reduceByKey(_ + _)

wordCounts.print()
```

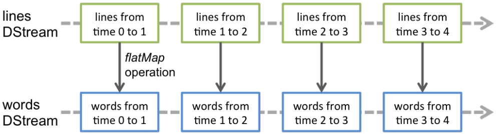
Example - Word Count (5/6)

- Start the computation and wait for it to terminate.

```
// Start the computation
ssc.start()

// Wait for the computation to terminate
ssc.awaitTermination()
```

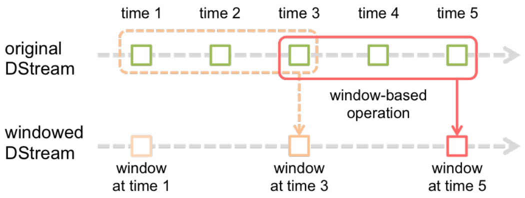
Example - Word Count (6/6)

```
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))

val lines = ssc.socketTextStream("localhost", 9999)
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
wordCounts.print()

ssc.start()
ssc.awaitTermination()
```



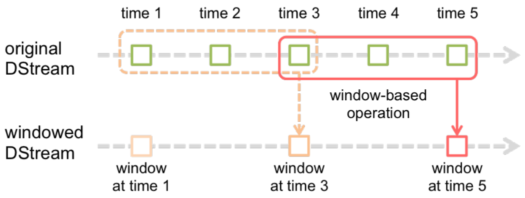
Window Operations (1/2)

- Spark provides a set of transformations that apply to a over a sliding window of data.



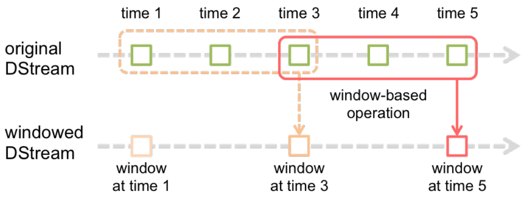
Window Operations (1/2)

- Spark provides a set of transformations that apply to a over a sliding window of data.
- A window is defined by two parameters: window length and slide interval.



Window Operations (1/2)

- Spark provides a set of transformations that apply to a over a sliding window of data.
- A window is defined by two parameters: window length and slide interval.
- A tumbling window effect can be achieved by making slide interval = window length



Window Operations (2/2)

- `reduceByWindow(func, windowLength, slideInterval)`
  - Returns a new single-element DStream, created by aggregating elements in the stream over a sliding interval using func.

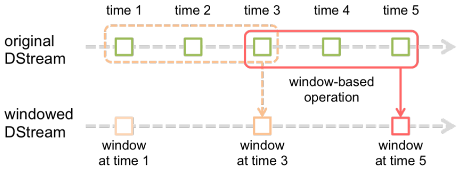
Window Operations (2/2)

- `reduceByWindow(func, windowLength, slideInterval)`
  - Returns a new single-element DStream, created by aggregating elements in the stream over a sliding interval using func.

- `reduceByKeyAndWindow(func, windowLength, slideInterval)`
  - Called on a DStream of (K, V) pairs.
  - Returns a new DStream of (K, V) pairs where the values for each key are aggregated using function func over batches in a sliding window.

Example - Word Count with Window

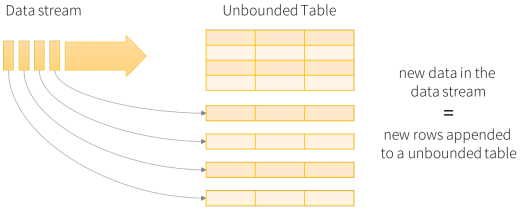
```
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))

val lines = ssc.socketTextStream("localhost", 9999)
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))
val windowedWordCounts = pairs.reduceByKeyAndWindow(_ + _, Seconds(30), Seconds(10))
windowedWordCounts.print()

ssc.start()
ssc.awaitTermination()
```



Structured Streaming

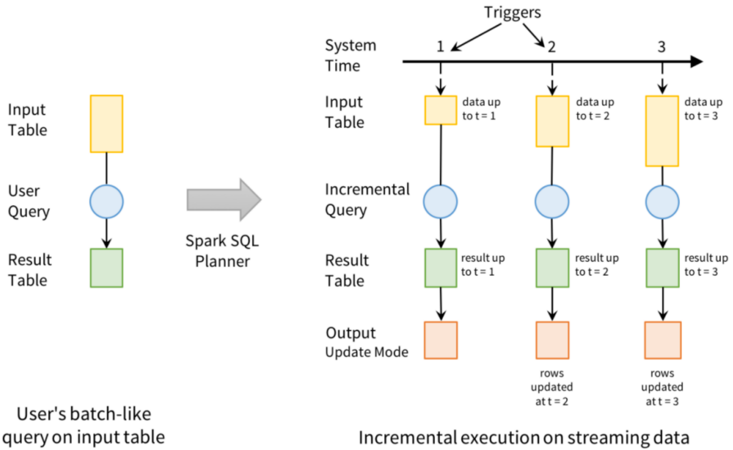
Structured Streaming

- Treating a live data stream as a table that is being continuously appended.



Data stream as an unbounded table

Programming Model



Output Modes

- Three output modes:

1. Append: only the new rows appended to the result table since the last trigger will be written to the external storage.

Output Modes

- Three output modes:

1. Append: only the new rows appended to the result table since the last trigger will be written to the external storage.

2. Complete: the entire updated result table will be written to external storage.

Output Modes

- Three output modes:

1. Append: only the new rows appended to the result table since the last trigger will be written to the external storage.
2. Complete: the entire updated result table will be written to external storage.
3. Update: only the rows that were updated in the result table since the last trigger will be changed in the external storage.

Steps to Define a Streaming Query (1/4)

- Define input sources.
- Use spark.readStream to create a DataStreamReader.

```
val spark = SparkSession.builder.master("local[2]").appName("appname").getOrCreate()

val lines = spark.readStream.format("socket")
    .option("host", "localhost")
    .option("port", 9999)
    .load()
```

Steps to Define a Streaming Query (2/4)

- Transform data.
- E.g., below counts is a streaming DataFrame that represents the running word counts.

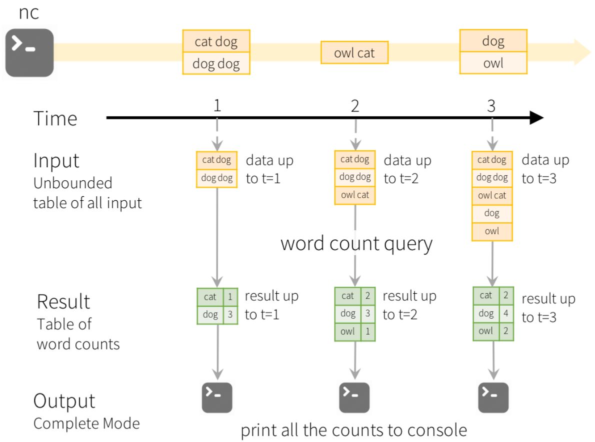
```
import org.apache.spark.sql.functions._

val words = lines.select(split(col("value"), " ").as("word"))

val wordCounts = words.groupBy("word").count()
```

Steps to Define a Streaming Query (3/4)

- Define output sink and output mode.
- Use DataFrame.writeStream to define how to write the processed output data.
- Start the query.

```
val query = wordCounts.writeStream.format("console").outputMode("complete").start()

query.awaitTermination()
```

Steps to Define a Streaming Query (4/4)

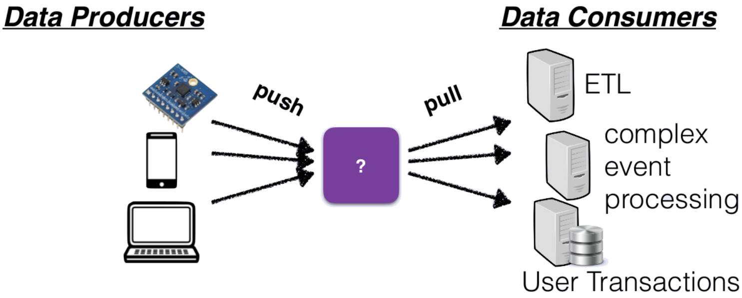


[https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html]

Data Stream Storage

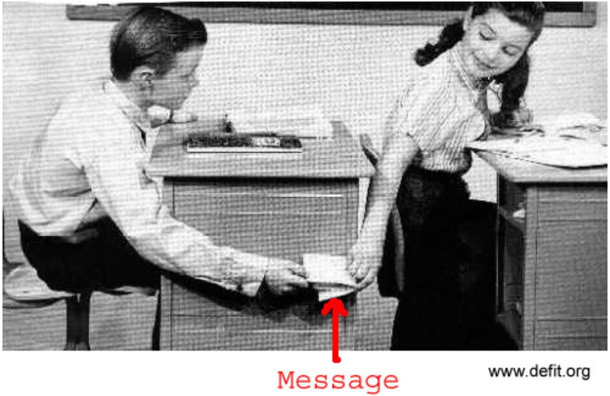
The Problem

- We need disseminate streams of events from various producers to various consumers.



Possible Solution?

- Messaging systems



What is Messaging System?

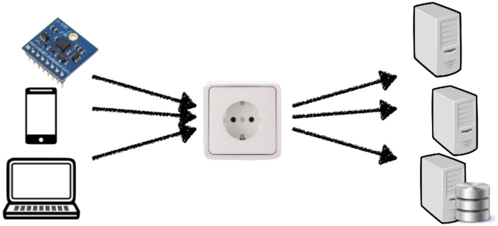
- Messaging system is an approach to notify consumers about new events.

What is Messaging System?

- Messaging system is an approach to notify consumers about new events.
- Messaging systems
  - Direct messaging
  - Message brokers

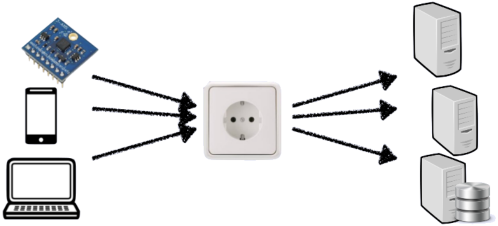
Direct Messaging (1/2)

- Necessary in latency critical applications (e.g., remote surgery).
- A producer sends a message containing the event, which is pushed to consumers.



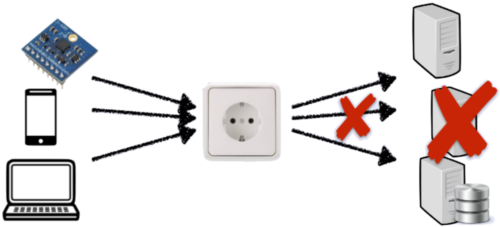
Direct Messaging (1/2)

- Necessary in latency critical applications (e.g., remote surgery).
- A producer sends a message containing the event, which is pushed to consumers.
- Both consumers and producers have to be online at the same time.



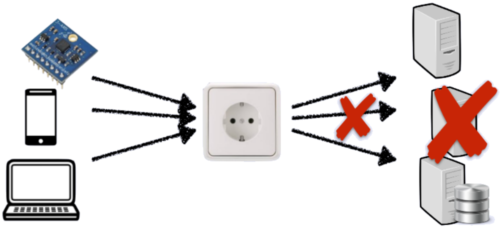
Direct Messaging (2/2)

- What happens if a consumer crashes or temporarily goes offline? (not durable)



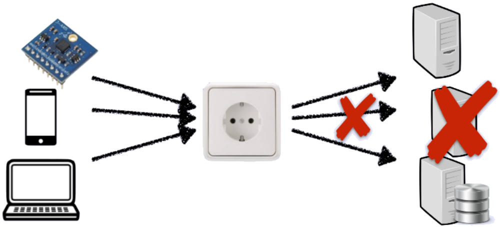
Direct Messaging (2/2)

- What happens if a consumer crashes or temporarily goes offline? (not durable)
- What happens if producers send messages faster than the consumers can process?



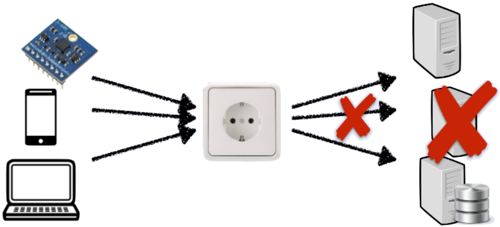
Direct Messaging (2/2)

- What happens if a consumer crashes or temporarily goes offline? (not durable)
- What happens if producers send messages faster than the consumers can process?
    - Dropping messages
    - Backpressure

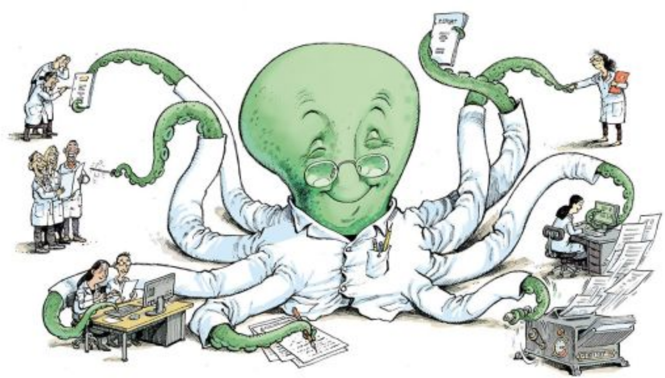


Direct Messaging (2/2)

- What happens if a consumer crashes or temporarily goes offline? (not durable)
- What happens if producers send messages faster than the consumers can process?
  - Dropping messages
  - Backpressure
- We need message brokers that can log events to process at a later time.



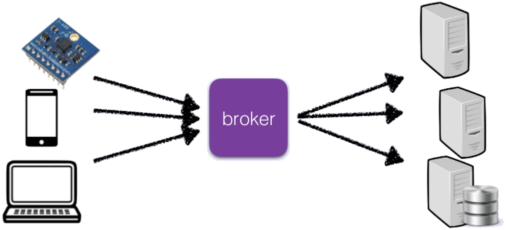
Message Broker



[https://bluesyemre.com/2018/10/16/thousands-of-scientists-publish-a-paper-every-five-days]

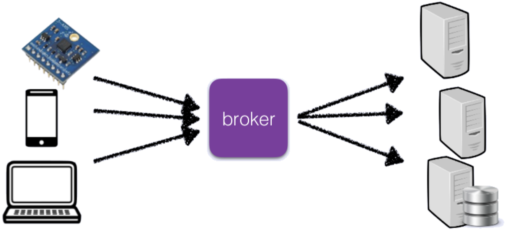
Message Broker

- A message broker decouples the producer-consumer interaction.
- It runs as a server, with producers and consumers connecting to it as clients.



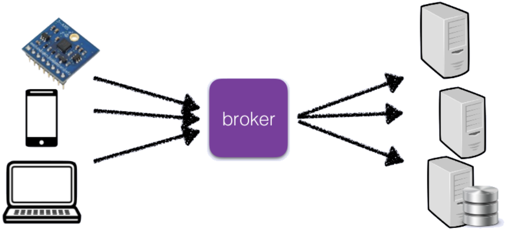
Message Broker

- A message broker decouples the producer-consumer interaction.
- It runs as a server, with producers and consumers connecting to it as clients.
- Producers write messages to the broker, and consumers receive them by reading them from the broker.



Message Broker

- A message broker decouples the producer-consumer interaction.
- It runs as a server, with producers and consumers connecting to it as clients.
- Producers write messages to the broker, and consumers receive them by reading them from the broker.
- Consumers are generally asynchronous.



Logs-Based Message Broker

- In typical message brokers, once a message is consumed, it is deleted.

Logs-Based Message Broker

- In typical message brokers, once a message is consumed, it is deleted.
- Log-based message brokers durably store all events in a sequential log.

Logs-Based Message Broker

- In typical message brokers, once a message is consumed, it is deleted.
- Log-based message brokers durably store all events in a sequential log.
- A log is an append-only sequence of records on disk.

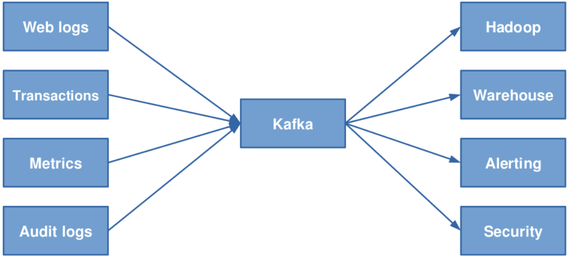
Logs-Based Message Broker

- In typical message brokers, once a message is consumed, it is deleted.
- Log-based message brokers durably store all events in a sequential log.
- A log is an append-only sequence of records on disk.
- A producer sends a message by appending it to the end of the log.
- A consumer receives messages by reading the log sequentially.

Kafka - A Log-Based Message Broker

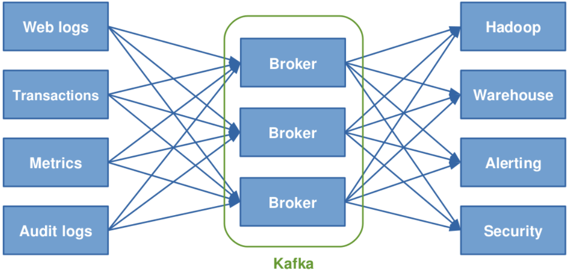
Kafka (1/5)

- Kafka is a distributed, topic oriented, partitioned, replicated commit log service.



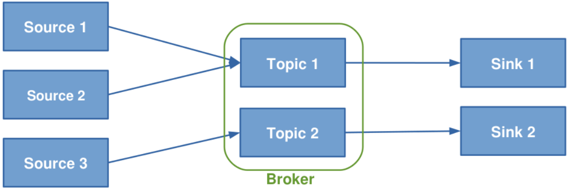
Kafka (2/5)

- Kafka is a distributed, topic oriented, partitioned, replicated commit log service.



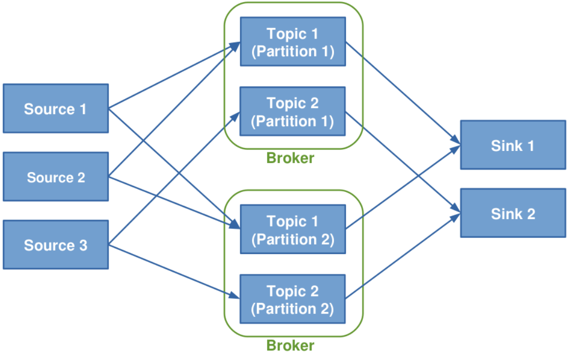
Kafka (3/5)

- Kafka is a distributed, topic oriented, partitioned, replicated commit log service.



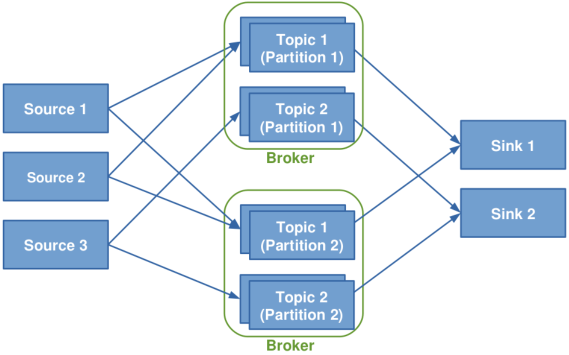
Kafka (4/5)

- Kafka is a distributed, topic oriented, partitioned, replicated commit log service.



Kafka (5/5)

- Kafka is a distributed, topic oriented, partitioned, replicated commit log service.



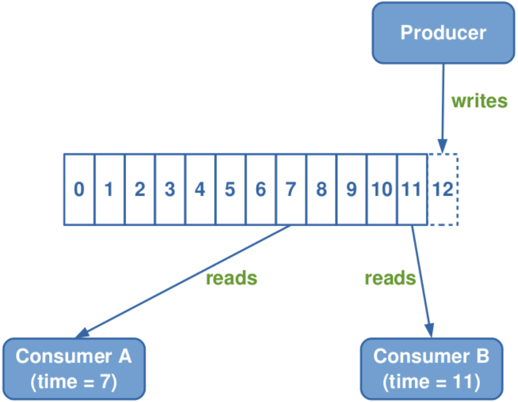
Topics and Partition (1/6)

- Topics are queues: a stream of messages of a particular type

```
jkreps-mn:~ jkreps$ tail -f -n 20 /var/log/apache2/access_log
::1 - - [23/Mar/2014:15:07:00 -0700] "GET /images/apache_feather.gif HTTP/1.1" 200 4128
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/producer_consumer.png HTTP/1.1" 200 86
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/log_anatomy.png HTTP/1.1" 200 19579
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/consumer-groups.png HTTP/1.1" 200 2682
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/log_compaction.png HTTP/1.1" 200 41414
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /documentation.html HTTP/1.1" 200 189893
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/log_cleaner_anatomy.png HTTP/1.1" 200
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/kafka_log.png HTTP/1.1" 200 134321
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/mirror-maker.png HTTP/1.1" 200 17054
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /documentation.html HTTP/1.1" 200 189937
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /styles.css HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/kafka_logo.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/producer_consumer.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/log_anatomy.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/consumer-groups.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/log_cleaner_anatomy.png HTTP/1.1" 304
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/log_compaction.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/kafka_log.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/mirror-maker.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:09:55 -0700] "GET /documentation.html HTTP/1.1" 200 195264
```

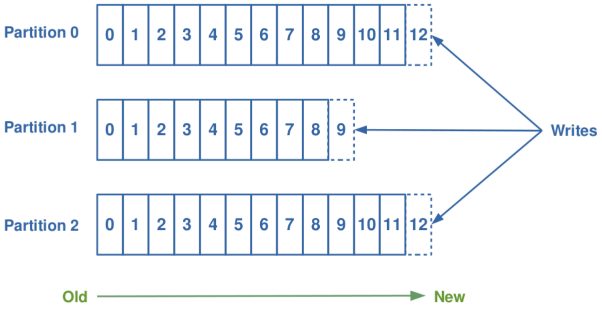
Topics and Partition (2/6)

- Each message is assigned a sequential id called an offset.



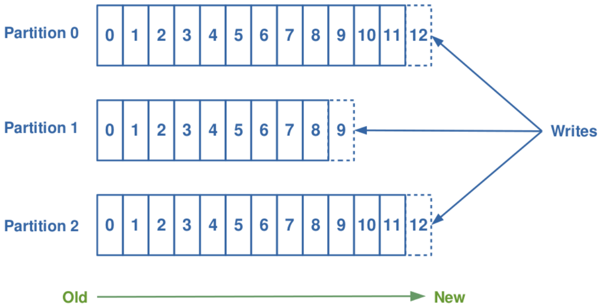
Topics and Partition (3/6)

- Topics are logical collections of partitions (the physical files).
  - Ordered
  - Append only
  - Immutable



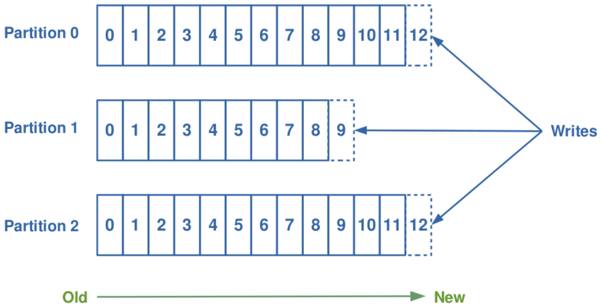
Topics and Partition (4/6)

- Ordering is only guaranteed within a partition for a topic.



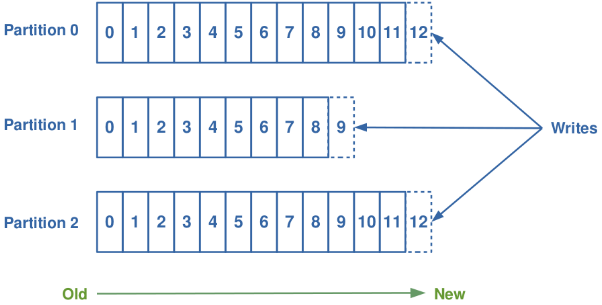
Topics and Partition (4/6)

- Ordering is only guaranteed within a partition for a topic.
- Messages sent by a producer to a particular topic partition will be appended in the order they are sent.



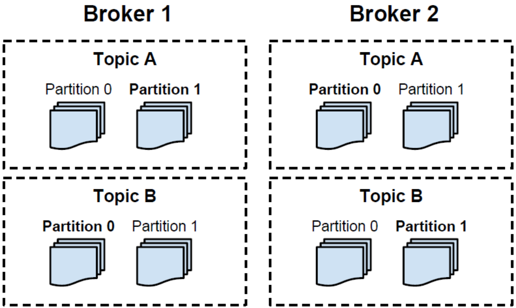
Topics and Partition (4/6)

- Ordering is only guaranteed within a partition for a topic.
- Messages sent by a producer to a particular topic partition will be appended in the order they are sent.
- A consumer instance sees messages in the order they are stored in the log.



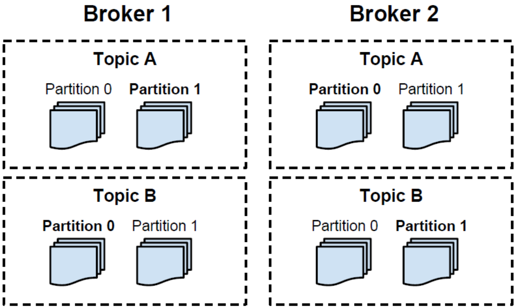
Topics and Partition (5/6)

- Partitions of a topic are replicated: fault-tolerance



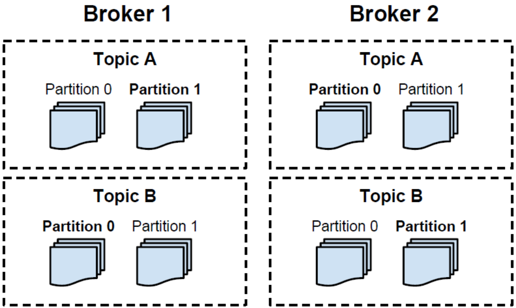
Topics and Partition (5/6)

- Partitions of a topic are replicated: fault-tolerance
- A broker contains some of the partitions for a topic.

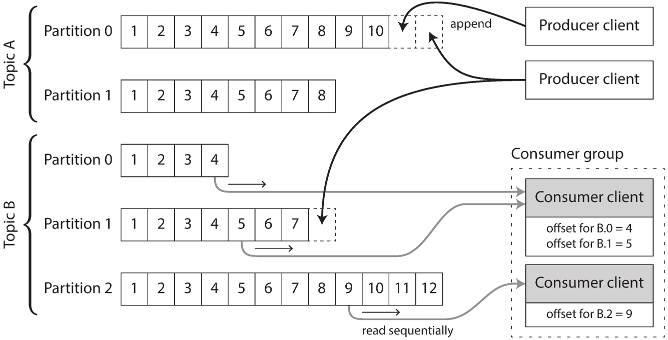


Topics and Partition (5/6)

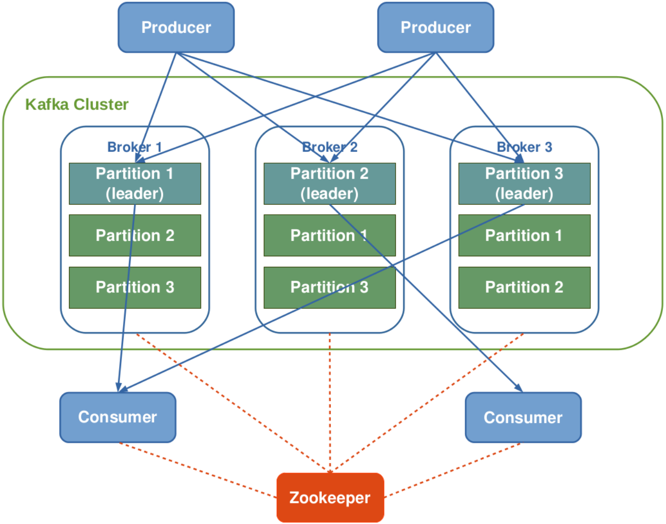
- Partitions of a topic are replicated: fault-tolerance
- A broker contains some of the partitions for a topic.
- One broker is the leader of a partition: all writes and reads must go to the leader.



Topics and Partition (6/6)



Kafka Architecture



Summary

Summary

- Data stream, unbounded data, tuples
- Event-time vs. processing time
- Windowing and triggering
- Messaging system and partitioned logs
- Kafka: distributed, topic oriented, partitioned, replicated log service
- Spark streaming and structured streaming

References

- J. Kreps et al., "Kafka: A distributed messaging system for log processing", NetDB 2011
- M. Zaharia et al., "Spark: The Definitive Guide", O'Reilly Media, 2018 - Chapter 20
- T. Akidau et al., "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing", VLDB 2015.
- M. Fragkoulis et al., "A Survey on the Evolution of Stream Processing Systems", 2020
- T. Akidau, "The world beyond batch: Streaming 101", `https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101`

Questions?