# Introduction to Data Stream Processing

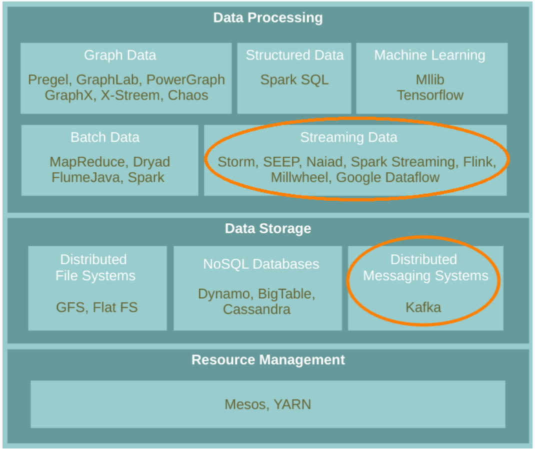
Amir H. Payberah
payberah@kth.se
2020-09-16

# The Course Web Page

https://id2221kth.github.io

https://tinyurl.com/y4qph82u

# Where Are We?

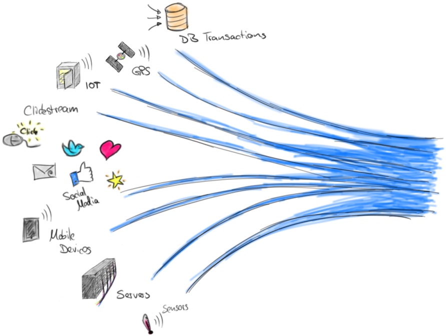

Data Processing
Graph Data
Pregel, GraphLab, PowerGraph
GraphX, X-Streem, Chaos
Structured Data
Spark SQL
Machine Learning
Mllib
Tensorflow
Batch Data
MapReduce, Dryad
FlumeJava, Spark
Streaming Data
Storm, SEEP, Naiad, Spark Streaming, Flink,
Millwheel, Google Dataflow
Data Storage
Distributed
File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

# Stream Processing (1/4)

- Stream processing is the act of continuously incorporating new data to compute a result.

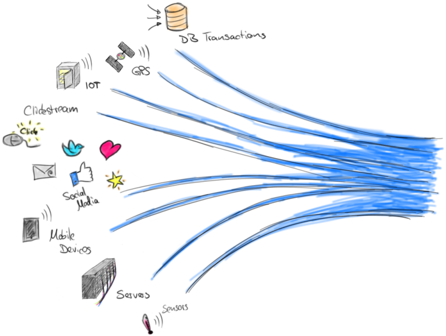
# Stream Processing (2/4)

- The input data is unbounded.
    - A series of events, no predetermined beginning or end.



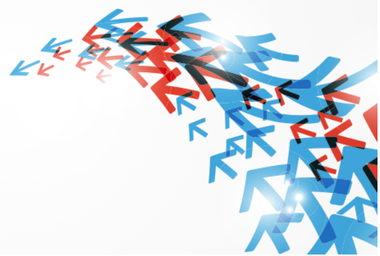
# Stream Processing (2/4)

- The input data is unbounded.
  - A series of events, no predetermined beginning or end.
  - E.g., credit card transactions, clicks on a website, or sensor readings from IoT devices.



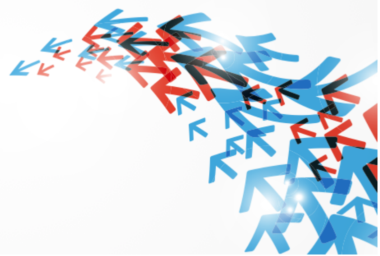
# Stream Processing (3/4)

- User applications can then compute various queries over this stream of events.



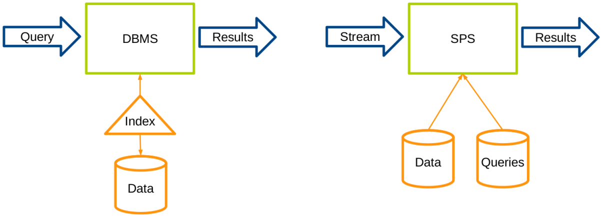
# Stream Processing (3/4)

- User applications can then compute various queries over this stream of events.
  - E.g., tracking a running count of each type of event, or aggregating them into hourly windows.



# Stream Processing (4/4)

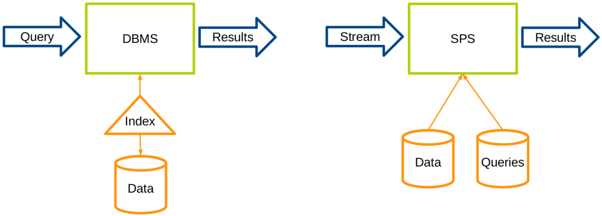
- Database Management Systems (DBMS): data-at-rest analytics
  - Store and index data before processing it.
  - Process data only when explicitly asked by the users.



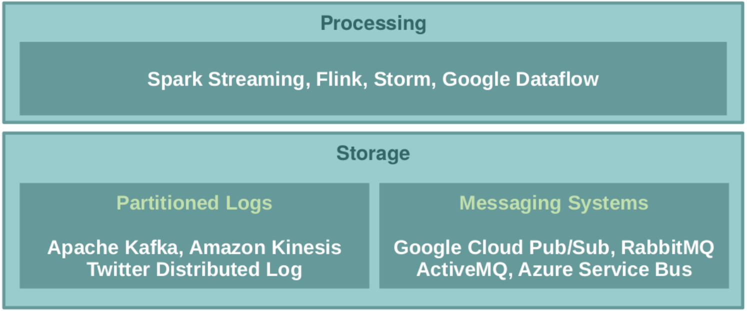
# Stream Processing (4/4)

- Database Management Systems (DBMS): data-at-rest analytics
  - Store and index data before processing it.
  - Process data only when explicitly asked by the users.

- Stream Processing Systems (SPS): data-in-motion analytics
  - Processing information as it flows, without storing them persistently.



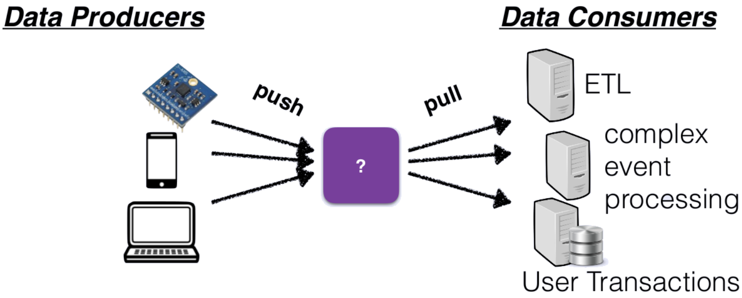
# Stream Processing Systems Stack



# Data Stream Storage

# The Problem

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



# Example

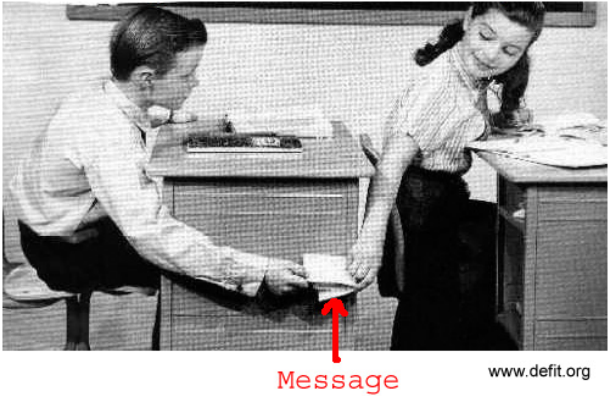
- Suppose you have a website, and every time someone loads a page, you send a viewed page event to consumers.

# Example

- Suppose you have a website, and every time someone loads a page, you send a viewed page event to consumers.
- The consumers may do any of the following:
  - Store the message in HDFS for future analysis
  - Count page views and update a dashboard
  - Trigger an alert if a page view fails
  - Send an email notification to another user

# Possible Solution?

- Messaging systems



# What is Messaging System?

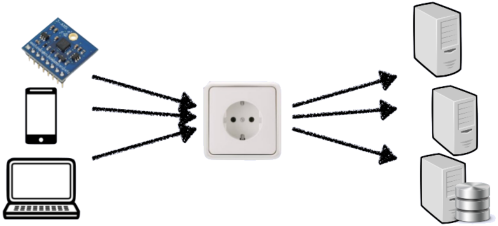
- Messaging system is an approach to notify consumers about new events.

# What is Messaging System?

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

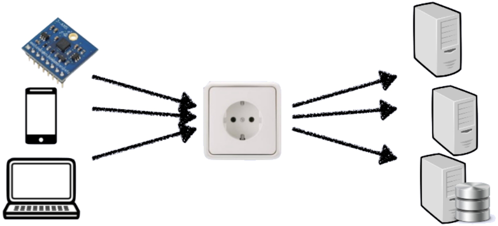
# Direct Messaging (1/2)

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



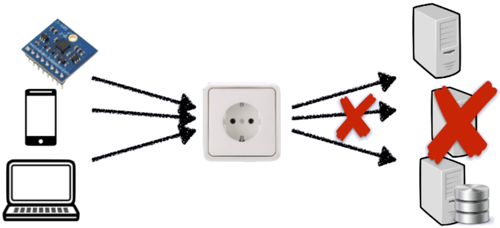
# Direct Messaging (1/2)

- 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.



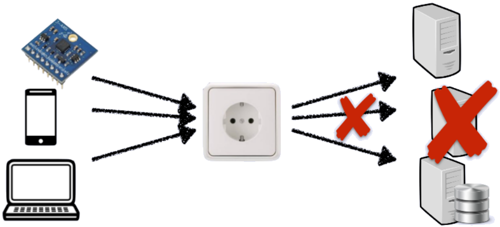
# Direct Messaging (2/2)

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



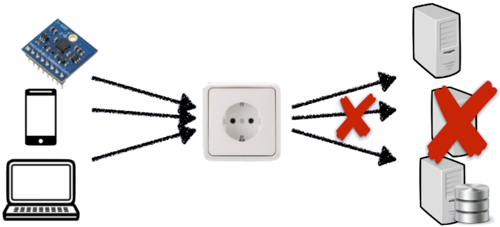
# Direct Messaging (2/2)

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



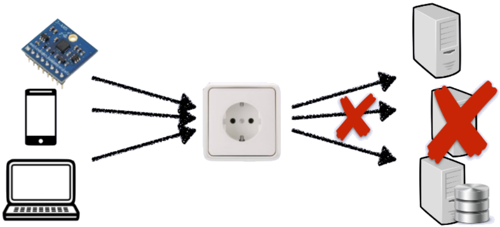
# Direct Messaging (2/2)

- 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

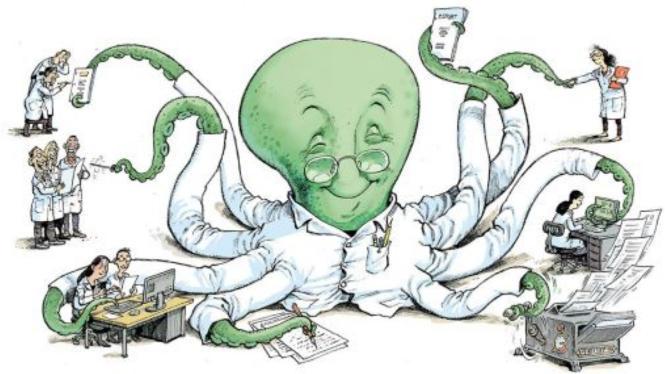


# Direct Messaging (2/2)

- 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.



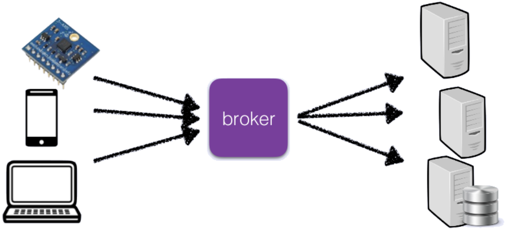
# Message Broker



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

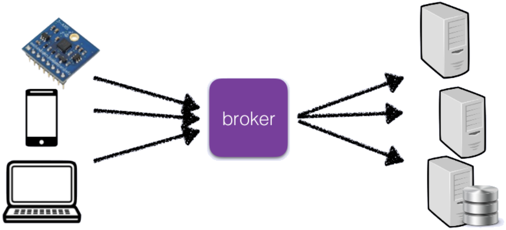
# Message Broker

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



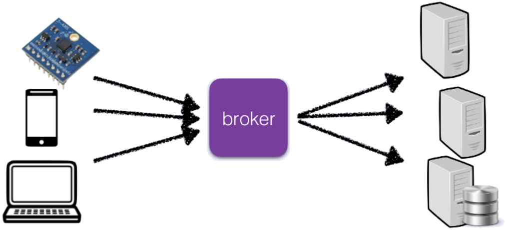
# Message Broker

- 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.

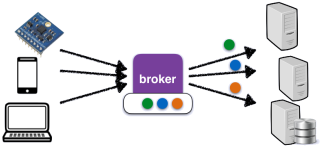


# Message Broker (2/2)

- When multiple consumers read messages in the same topic.

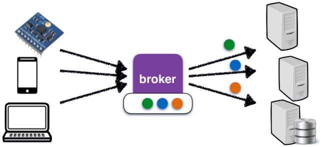
# Message Broker (2/2)

- When multiple consumers read messages in the same topic.
- Load balancing: each message is delivered to one of the consumers.

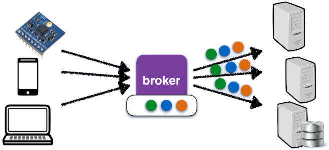


# Message Broker (2/2)

- When multiple consumers read messages in the same topic.
- Load balancing: each message is delivered to one of the consumers.




- Fan-out: each message is delivered to all of the consumers.



# Partitioned Logs (1/2)

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

# Partitioned Logs (1/2)

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

# Partitioned Logs (1/2)

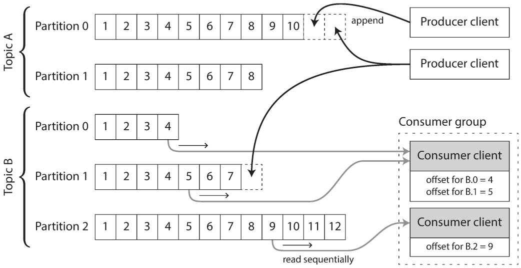
- In typical message brokers, once a message is consumed, it is deleted.
- Log-based message brokers durably store all events in a sequential log.
- A log is an append-only sequence of records on disk.

# Partitioned Logs (1/2)

- 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.

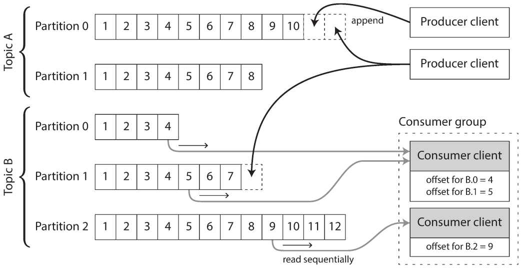
# Partitioned Logs (2/2)

- To scale up the system, logs can be partitioned hosted on different machines.



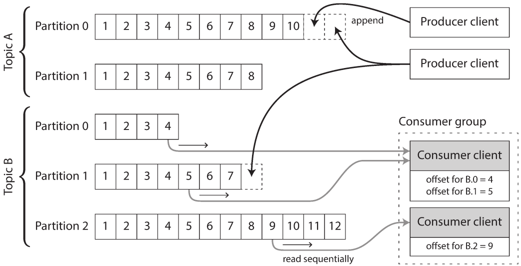
# Partitioned Logs (2/2)

- To scale up the system, logs can be partitioned hosted on different machines.
- Each partition can be read and written independently of others.



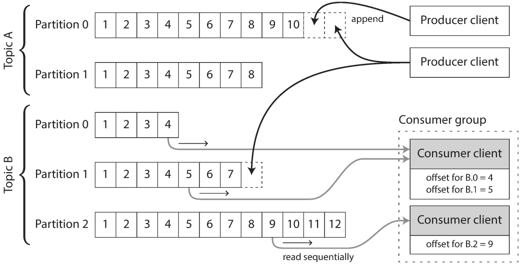
# Partitioned Logs (2/2)

- To scale up the system, logs can be partitioned hosted on different machines.
- Each partition can be read and written independently of others.
- A topic is a group of partitions that all carry messages of the same type.



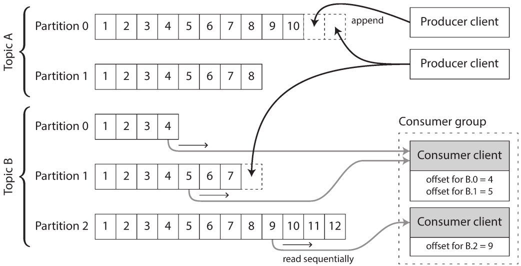
# Partitioned Logs (2/2)

- To scale up the system, logs can be partitioned hosted on different machines.
- Each partition can be read and written independently of others.
- A topic is a group of partitions that all carry messages of the same type.
- Within each partition, the broker assigns a monotonically increasing sequence number (offset) to every message



# Partitioned Logs (2/2)

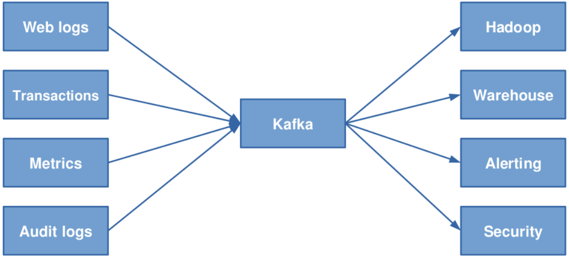
- To scale up the system, logs can be partitioned hosted on different machines.
- Each partition can be read and written independently of others.
- A topic is a group of partitions that all carry messages of the same type.
- Within each partition, the broker assigns a monotonically increasing sequence number (offset) to every message
- No ordering guarantee across partitions.



# Kafka - A Log-Based Message Broker

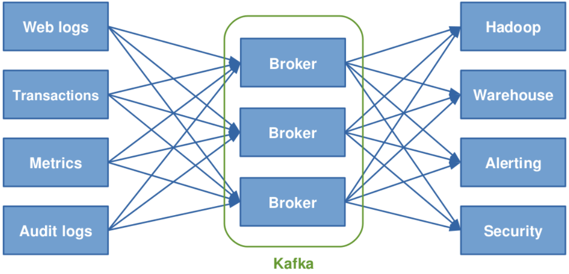
# Kafka (1/5)

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



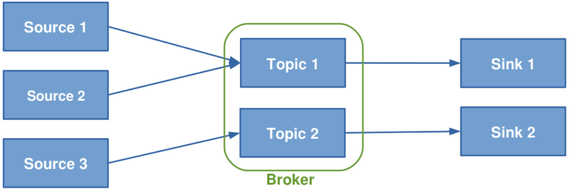
# Kafka (2/5)

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



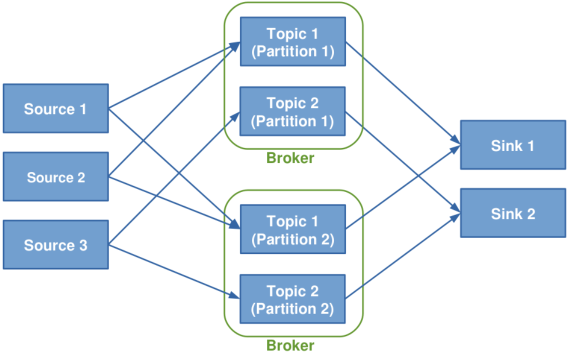
# Kafka (3/5)

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



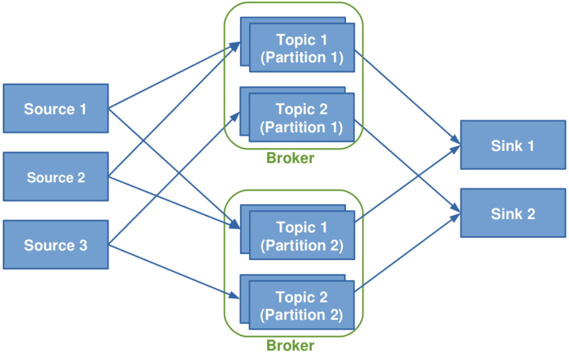
# Kafka (4/5)

- 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.



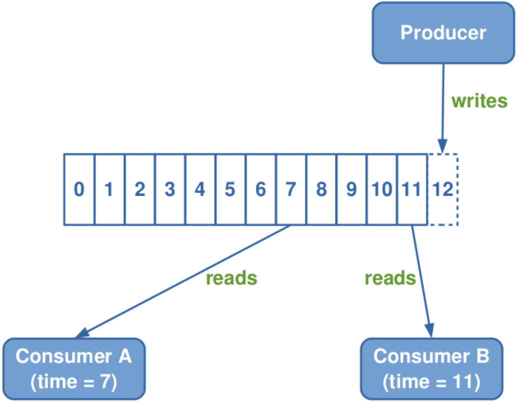
# Logs, Topics and Partition (1/5)

- Kafka is about logs.
- 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 8
::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" 200 268
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/log_compaction.png HTTP/1.1" 200 4141
::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
```

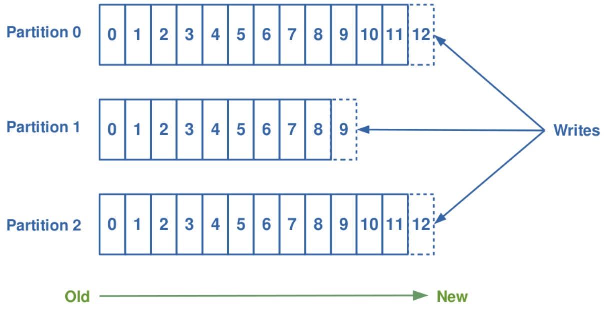
# Logs, Topics and Partition (2/5)

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



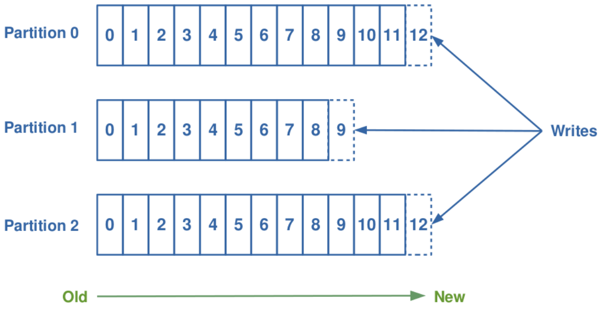
# Logs, Topics and Partition (3/5)

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



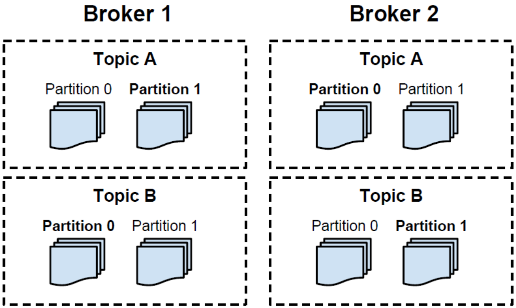
# Logs, Topics and Partition (4/5)

- 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.

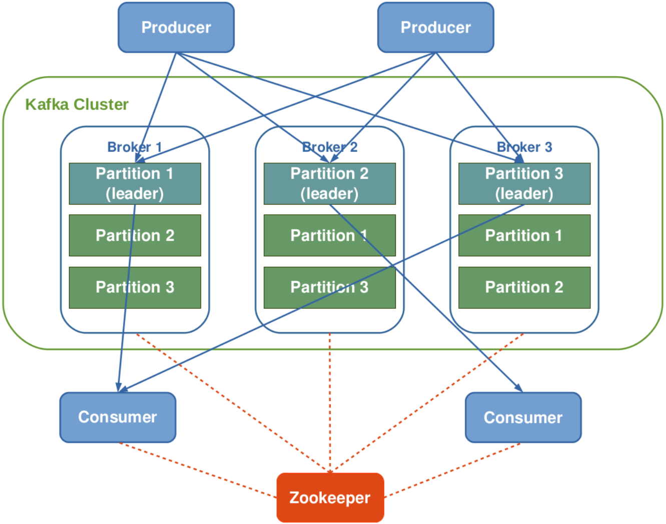


# Logs, Topics and Partition (5/5)

- 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.



# Kafka Architecture



# Coordination

- Kafka uses Zookeeper for the following tasks:

# Coordination

- Kafka uses Zookeeper for the following tasks:
- Detecting the addition and the removal of brokers and consumers.
- Keeping track of the consumed offset of each partition.

# State in Kafka

- Brokers are sateless: no metadata for consumers-producers in brokers.

# State in Kafka

- Brokers are sateless: no metadata for consumers-producers in brokers.
- Consumers are responsible for keeping track of offsets.

# State in Kafka

- Brokers are sateless: no metadata for consumers-producers in brokers.
- Consumers are responsible for keeping track of offsets.
- Messages in queues expire based on pre-configured time periods (e.g., once a day).

# Delivery Guarantees

- Kafka guarantees that messages from a single partition are delivered to a consumer in order.

# Delivery Guarantees

- Kafka guarantees that messages from a single partition are delivered to a consumer in order.
- There is no guarantee on the ordering of messages coming from different partitions.

# Delivery Guarantees

- Kafka guarantees that messages from a single partition are delivered to a consumer in order.
- There is no guarantee on the ordering of messages coming from different partitions.
- Kafka only guarantees at-least-once delivery.

# Start and Work With Kafka

```
# Start the ZooKeeper
zookeeper-server-start.sh config/zookeeper.properties
```

# Start and Work With Kafka

```
# Start the ZooKeeper
zookeeper-server-start.sh config/zookeeper.properties
```

```
# Start the Kafka server
kafka-server-start.sh config/server.properties
```

# Start and Work With Kafka

```
# Start the ZooKeeper
zookeeper-server-start.sh config/zookeeper.properties
```

```
# Start the Kafka server
kafka-server-start.sh config/server.properties
```

```
# Create a topic, called "avg"
kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1
                --topic avg
```

# Start and Work With Kafka

```
# Start the ZooKeeper
zookeeper-server-start.sh config/zookeeper.properties
```

```
# Start the Kafka server
kafka-server-start.sh config/server.properties
```

```
# Create a topic, called "avg"
kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1
                --topic avg
```

```
# Produce messages and send them to the topic "avg"
kafka-console-producer.sh --broker-list localhost:9092 --topic avg
```

# Start and Work With Kafka

```
# Start the ZooKeeper
zookeeper-server-start.sh config/zookeeper.properties
```

```
# Start the Kafka server
kafka-server-start.sh config/server.properties
```

```
# Create a topic, called "avg"
kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1
                --topic avg
```

```
# Produce messages and send them to the topic "avg"
kafka-console-producer.sh --broker-list localhost:9092 --topic avg
```

```
# Consume the messages sent to the topic "avg"
kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic avg --from-beginning
```

# Data Stream Processing

# 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 Data Processing Design Points

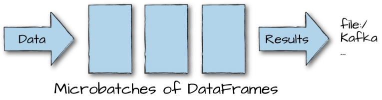
- Continuous vs. micro-batch processing
- Record-at-a-Time vs. declarative APIs
- Event time vs. processing time
- Windowing

# Streaming Data Processing Design Points

- Continuous vs. micro-batch processing
- Record-at-a-Time vs. declarative APIs
- Event time vs. processing time
- Windowing

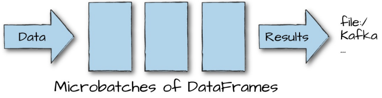
# Streaming Data Processing Patterns

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

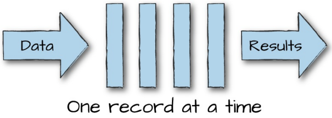


# Streaming Data Processing Patterns

- 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.



# Streaming Data Processing Design Points

- Continuous vs. micro-batch processing
- Record-at-a-Time vs. declarative APIs
- Event time vs. processing time
- Windowing

# Record-at-a-Time vs. Declarative APIs

- Record-at-a-Time API (e.g., Storm)
  - Low-level API
  - Passes each event to the application and let it react.
  - Useful when applications need full control over the processing of data.
  - Complicated factors, such as maintaining state, are governed by the application.

# Record-at-a-Time vs. Declarative APIs

- Record-at-a-Time API (e.g., Storm)
  - Low-level API
  - Passes each event to the application and let it react.
  - Useful when applications need full control over the processing of data.
  - Complicated factors, such as maintaining state, are governed by the application.
- Declarative API (e.g., Spark streaming, Flink, Google Dataflow)
  - Aapplications specify what to compute not how to compute it in response to each new event.

# Streaming Data Processing Design Points

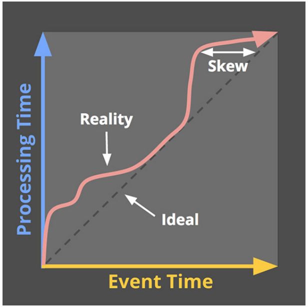
- Continuous vs. micro-batch processing
- Record-at-a-Time vs. declarative APIs
- Event time vs. processing time
- Windowing

# Event Time vs. Processing Time (1/2)

- 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.

# Event Time vs. Processing Time (2/2)

- Ideally, event time and processing time should be equal.
- Skew between event time and processing time.




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

# Streaming Data Processing Design Points

- Continuous vs. micro-batch processing
- Record-at-a-Time vs. declarative APIs
- Event time vs. processing time
- Windowing

# Windowing (1/2)

- Window: a buffer associated with an input port to retain previously received tuples.
- Four different windowing management policies.

# Windowing (1/2)

- Window: a buffer associated with an input port to retain previously received tuples.
- Four different windowing management policies.
  - Count-based policy: the maximum number of tuples a window buffer can hold

# Windowing (1/2)

- Window: a buffer associated with an input port to retain previously received tuples.
- Four different windowing management policies.
  - Count-based policy: the maximum number of tuples a window buffer can hold
  - Delta-based policy: a delta threshold in a tuple attribute

# Windowing (1/2)

- Window: a buffer associated with an input port to retain previously received tuples.
- Four different windowing management policies.
  - Count-based policy: the maximum number of tuples a window buffer can hold
  - Delta-based policy: a delta threshold in a tuple attribute
  - Punctuation-based policy: a punctuation is received

# Windowing (1/2)

- Window: a buffer associated with an input port to retain previously received tuples.
- Four different windowing management policies.
  - Count-based policy: the maximum number of tuples a window buffer can hold
  - Delta-based policy: a delta threshold in a tuple attribute
  - Punctuation-based policy: a punctuation is received
  - Time-based policy: based on processing or event time period

# Windowing (2/2)

- Two types of windows: tumbling and sliding

# Windowing (2/2)

- Two types of windows: tumbling and sliding
- Tumbling window: supports batch operations.
  - When the buffer fills up, all the tuples are evicted.

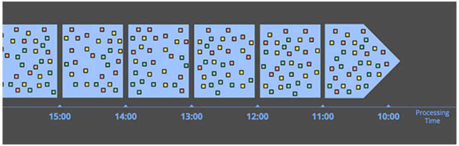
# Windowing (2/2)

- Two types of windows: tumbling and sliding
- Tumbling window: supports batch operations.
  - When the buffer fills up, all the tuples are evicted.


- Sliding window: supports incremental operations.
  - When the buffer fills up, older tuples are evicted.

# Windowing by Processing Time

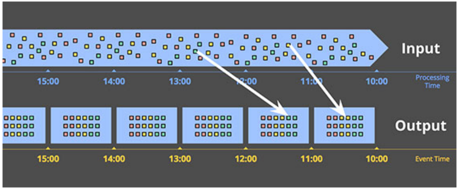
- The system buffers up incoming data into windows until some amount of processing time has passed.
- E.g., five-minute fixed windows




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

# Windowing by Event Time

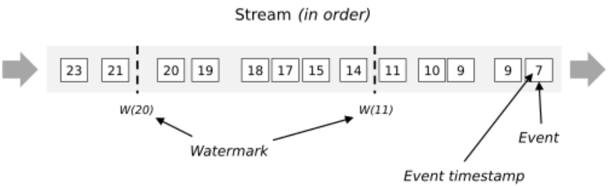
- Reflect the times at which events actually happened.
- Handling out-of-order evnets.




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

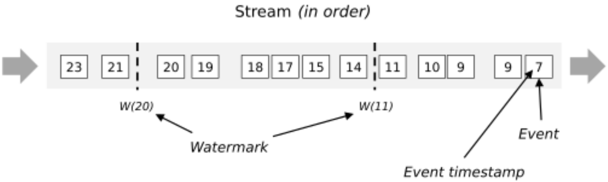
# Windowing by Event Time - Watermark (1/2)

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



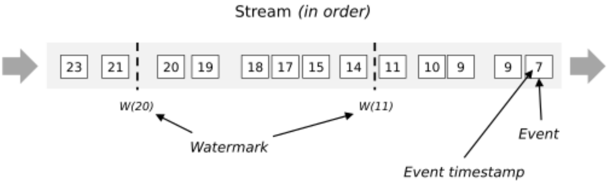
# Windowing by Event Time - Watermark (1/2)

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



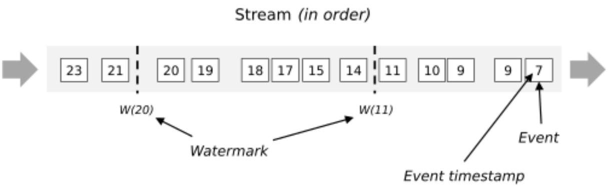
# Windowing by Event Time - Watermark (1/2)

- Watermarking helps a stream processing system to deal with lateness.
- Watermarks flow as part of the data stream and carry a timestamp `t`.
- A watermark is a threshold to specify how long the system waits for late events.



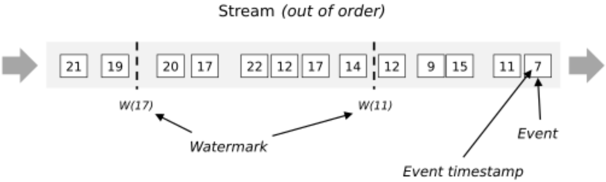
# Windowing by Event Time - Watermark (1/2)

- Watermarking helps a stream processing system to deal with lateness.
- Watermarks flow as part of the data stream and carry a timestamp `t`.
- A watermark is a threshold to specify how long the system waits for late events.
- Streaming systems uses watermarks to measure progress in event time.



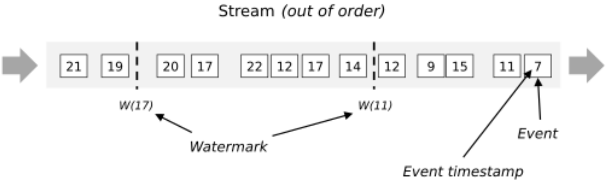
# Windowing by Event Time - Watermark (2/2)

- A `W(t)` declares that event time has reached time `t` in that stream
  - There should be no more elements from the stream with a timestamp $t' \leq t$.



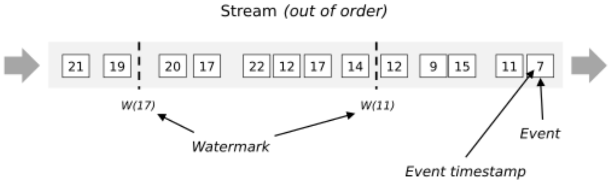
# Windowing by Event Time - Watermark (2/2)

- A `W(t)` declares that event time has reached time `t` in that stream
  - There should be no more elements from the stream with a timestamp $\mathtt{t}' \leq \mathtt{t}$.
- It is possible that certain elements will violate the watermark condition.
  - After the $\mathtt{W}(\mathtt{t})$ has occurred, more elements with timestamp $\mathtt{t}' \leq \mathtt{t}$ will occur.



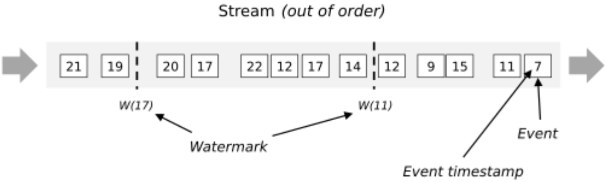
# Windowing by Event Time - Watermark (2/2)

- A `W(t)` declares that event time has reached time `t` in that stream
  - There should be no more elements from the stream with a timestamp $t' \leq t$.
- It is possible that certain elements will violate the watermark condition.
  - After the `W(t)` has occurred, more elements with timestamp $t' \leq t$ will occur.
- If an arriving event lies within the watermark, it gets used to update a query.



# Windowing by Event Time - Watermark (2/2)

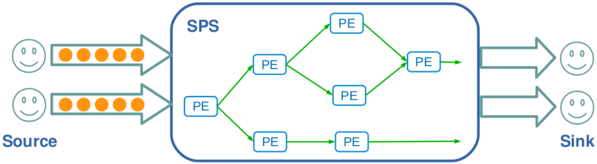
- A `W(t)` declares that event time has reached time `t` in that stream
  - There should be no more elements from the stream with a timestamp $t' \leq t$.
- It is possible that certain elements will violate the watermark condition.
  - After the `W(t)` has occurred, more elements with timestamp $t' \leq t$ will occur.
- If an arriving event lies within the watermark, it gets used to update a query.
- Streaming programs may explicitly expect some late elements.



# Streaming Data Processing Model

# Streaming Data Processing

- The tuples are processed by the application's operators or processing element (PE).
- A PE is the basic functional unit in an application.
  - A PE processes input tuples, applies a function, and outputs tuples.
  - A set of PEs and stream connections, organized into a data flow graph.



# PEs States (1/3)

- A PE can either maintain internal state across tuples while processing them, or process tuples independently of each other.
- Stateful vs. stateless tasks

# PEs States (2/3)

- Stateless tasks: do not maintain state and process each tuple independently of prior history, or even from the order of arrival of tuples.

# PEs States (2/3)

- Stateless tasks: do not maintain state and process each tuple independently of prior history, or even from the order of arrival of tuples.
- Easily parallelized.
- No synchronization.
- Restart upon failures without the need of any recovery procedure.

# PEs States (3/3)

- Stateful tasks: involves maintaining information across different tuples to detect complex patterns.

# PEs States (3/3)

- Stateful tasks: involves maintaining information across different tuples to detect complex patterns.
- A PE is usually a synopsis of the tuples received so far.
- A subset of recent tuples kept in a window buffer.

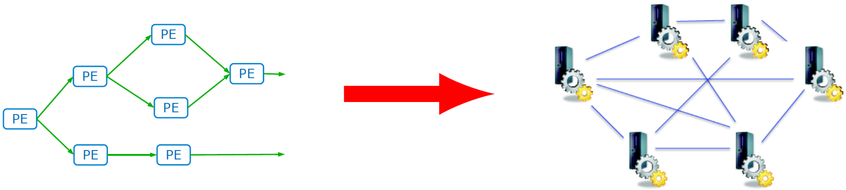
# Runtime Systems

# Job and Job Management

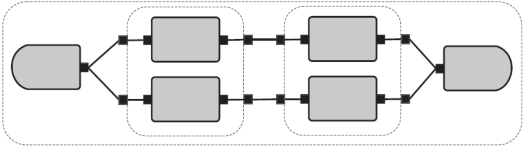
- At runtime, an application is represented by one or more jobs.
- Jobs are deployed as a collection of PEs.
- Job management component must identify and track individual PEs, the jobs they belong to, and associate them with the user that instantiated them.

# Logical Plan vs. Physical Plan (1/3)

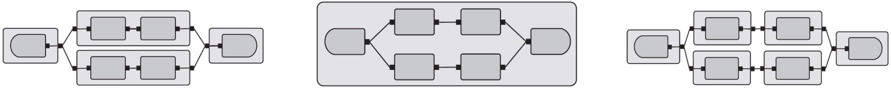
- Logical plan: a data flow graph, where the vertices correspond to PEs, and the edges to stream connections.
- Physical plan: a data flow graph, where the vertices correspond to OS processes, and the edges to transport connections.



# Logical Plan vs. Physical Plan (2/3)



Logical plan



Different physical plans

# Logical Plan vs. Physical Plan (3/3)

- How to map a network of PEs onto the physical network of nodes?
  - Parallelization
  - Fault tolerance
  - Optimization

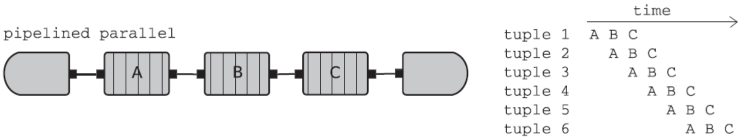
# Parallelization

# Parallelization

- How to scale with increasing the number queries and the rate of incoming events?
- Three forms of parallelisms.
  - Pipelined parallelism
  - Task parallelism
  - Data parallelism

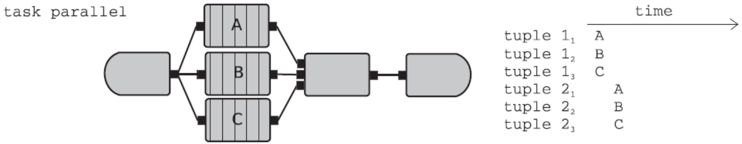
# Pipelined Parallelism

- Sequential stages of a computation execute concurrently for different data items.



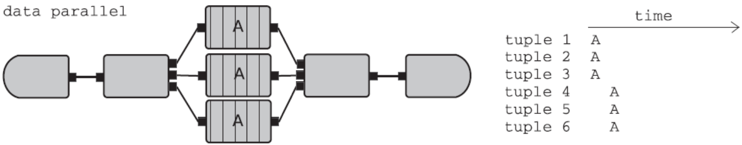
# Task Parallelism

- Independent processing stages of a larger computation are executed concurrently on the same or distinct data items.



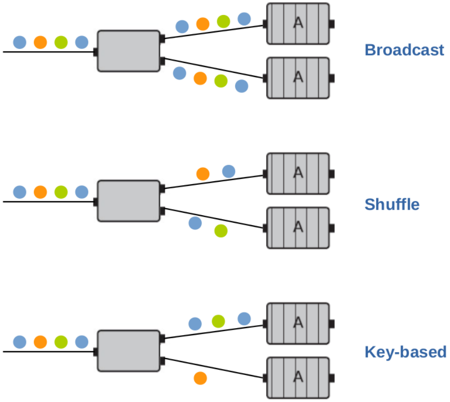
# Data Parallelism (1/2)

- The same computation takes place concurrently on different data items.



# Data Parallelism (2/2)

- How to allocate data items to each computation instance?



# Fault Tolerance

# Fault Tolerance

- The recovery methods of streaming frameworks must take:
  - Correctness, e.g., data loss and duplicates
  - Performance, e.g., low latency

# Delivery Guarantees

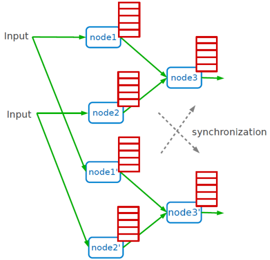
- At-least-once: might appear many times
- Exactly-once: is consumed just once

# Recovery Methods

- Active backup
- Passive backup
- Upstream backup

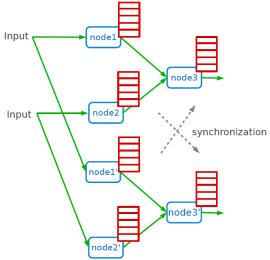
# Recovery Methods - Active Backup

- Each processing node has an associated backup node.
- Both primary and backup nodes are given the same input.



# Recovery Methods - Active Backup

- Each processing node has an associated backup node.
- Both primary and backup nodes are given the same input.
- If the primary fails, the backup takes over by sending the logged tuples to all downstream neighbors and then continuing its processing.

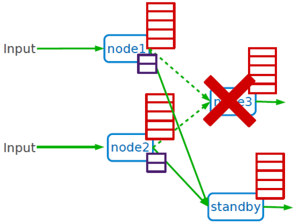


# Recovery Methods - Passive Backup

- Periodically check-points processing state to a shared storage.
- The backup node takes over from the latest checkpoint when the primary fails.

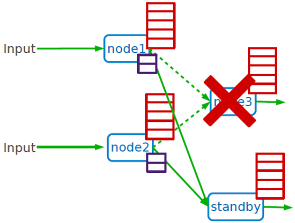
## Recovery Methods - Upstream Backup

- Upstream nodes store the tuples until the downstream nodes acknowledge them.



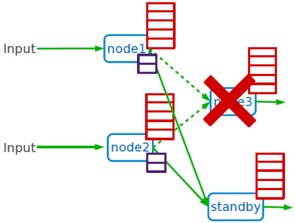
# Recovery Methods - Upstream Backup

- Upstream nodes store the tuples until the downstream nodes acknowledge them.
- If a node fails, an empty node rebuilds the latest state of the failed primary from the logs kept at the upstream server.



# Recovery Methods - Upstream Backup

- Upstream nodes store the tuples until the downstream nodes acknowledge them.
- If a node fails, an empty node rebuilds the latest state of the failed primary from the logs kept at the upstream server.
- There is no backup node in this model.



# Summary

# Summary

- Messaging system and partitioned logs
- Decoupling producers and consumers
- Kafka: distributed, topic oriented, partitioned, replicated log service
- Logs, topcs, partition
- Kafka architecture: producer, consumer, broker, coordinator

# Summary

- SPS vs. DBMS
- Data stream, unbounded data, tuples
- Event-time vs. processing time
- Micro-batch vs. continues processing (windowing)
- PEs and dataflow
- Stateless vs. Stateful PEs
- SPS runtime: parallelization, fault-tolerance

# 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
- M. Fragkoulis et al., “A Survey on the Evolution of Stream Processing Systems”, 2020
- J. Hwang et al., “High-availability algorithms for distributed stream processing”, ICDE 2005
- T. Akidau, “The world beyond batch: Streaming 101”, `https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101`

# Questions?