Distributed Data Management
Stream Processing

Thorsten Papenbrock
F-2.04, Campus II
Hasso Plattner Institut
## Services (online systems)
- Accept requests and send responses
- Performance measure: response time and availability
- Expected runtime: milliseconds to seconds

## Batch processing systems (offline systems)
- Take (large amounts of) data; run (complex) jobs; produce some output
- Performance measure: throughput (i.e., data per time)
- Expected runtime: minutes to days

## Stream processing systems (near-real-time systems)
- Consume volatile inputs; operate stream jobs; produce some output
- Performance measure: throughput and precision
- Expected runtime: near-real-time (i.e., as data arrives)
Distributed Data Management

Types of Systems

Batch processing systems (offline systems)

Stream processing systems (near-real-time systems)

- **Batch processing systems (offline systems)**
  - Map
  - Map
  - Reduce
  - Map
  - Map
  - Reduce
  - Map
  - Reduce
- **Stream processing systems (near-real-time systems)**
  - Map
  - Map
  - Reduce
  - Map
  - Map
  - Reduce
  - Map
  - Reduce
Distributed Data Management

Types of Systems

Batch processing systems (offline systems)

Stream processing systems (near-real-time systems)
Distributed Data Management

Use Cases for Streaming Data

Sensor Processing
- Continuous and endless readings by nature

Process Monitoring
- Side effects of processes that are continuously observed

Location Tracking
- Continuous location updates of certain devices

Log Analysis
- Digital footprints of applications that grow continuously

User Interaction
- Continuous and oftentimes bursty click- and call-events

Market and Climate Prediction
- Changing stock market prices and weather characteristics
Spark Streaming (Recap)

**Batched Stream Processing**

- **Reasons:**
  - **Incremental processing:** start processing data that is still being written to
  - **Latency reduction:** pipeline data to maximizing resource utilization

Stream processing reads the **data exactly once** and still guarantees **fault-tolerance** through **checkpointing** and **write ahead logs** (WAL).
Data Stream

- Any data that is incrementally made available over time
- Examples:
  - Unix stdin and stdout
  - Filesystem APIs (e.g. Java’s FileInputStream)
  - Online media delivery (audio/video streaming)
- Creation from …
  - static data: files or databases (read records line-wise)
  - dynamic data: sensor readings, service calls, transmitted data, logs, …

Event

- = an immutable record in a stream (often with timestamp)
- “Something that happened”
- Encoded in Json, XML, CSV, … maybe in binary format
Distributed Data Management

Batch vs. Stream

- **Batches**: Write once, read often
- **Streams**: Send once, receive once

maybe multiple receivers

Distributed Data Management
Stream Processing

ThorstenPapenbrock
Slide 8
Overview
Stream Processing

Transmitting Event Streams
Databases and Streams
Processing Streams

Distributed Data Management
Stream Processing
ThorstenPapenbrock
Slide 9
Dataflow Through Databases

- Process 2 needs to poll the database for updates
  - bad performance
  - slow event propagation

Dataflow Through Services

- Working speed of process 2 determines stream speed
  - maybe bad performance
  - ok-ish event propagation

Message-Passing Dataflow

- Asynchronous messaging and notification about new events
  - good performance
  - fast event propagation
Communication
- Objects send messages to other objects via queues.

Message
- Container for data (= events)
  - Often carries metadata (sender, receiver, timestamp, ...)

Message queue
- Data structure (queue or list) assigned to communicating object(s)
  - Enqueues messages in order of arrival
  - Buffers incoming messages for being processed
  - Notifies subscribers if new messages are available
What if the stream producer is faster than the stream consumer(s)?

a) Drop messages
   - Delete messages that cannot be accepted.
   - Ok for use cases where timeliness is more important than completeness (e.g. for processing of sensor readings)

b) Buffer messages
   - Store messages in a cache until resources are available.
   - Ok to capture load spikes and if there is no constant overload that fills up buffers permanently (e.g. for user activity event streams)

c) Apply backpressure
   - Block the sender until resources are available.
   - Ok if the sender can be blocked and if the stream is not generated from outside (e.g. for reading a file as a stream from disk)
What if nodes crash or temporarily go offline?

a) Fault ignorance
   - Failed messages are lost.
   - Ensures optimal throughput and latency

b) Fault tolerance
   - Failed messages are recovered from checkpoints (disk or replicas).
   - Ensures messaging reliability

More on fault tolerance later!
Message Broker

- Also called message queue or message-oriented middleware
- Part of the message-passing framework that delivers messages from their sender to the receiver(s)
- Maintains queues that sender can post messages to
- Notifies subscribers on new messages
- Resolves sender and receiver addresses
- Applies binary encoding when necessary
- Define the ...
  - message congestion strategy
  - messaging fault strategy

If it blocks and persists, then it is a database, right?
Transmitting Event Streams
Message Brokers vs. Databases

**Message Broker**
- Short lived messages
  - Delete messages once successfully transmitted
- Small working set
  - If the number of pending messages increases, the performance drops (disk!)
- Subscription-based retrieval
  - Deliver messages to all subscribers of a queue
- Push client communication
  - Knows clients and initiates communications

**Database**
- Long-term persisted records
  - Store records until explicitly deleted
- Large working set
  - If the number of records increases, the performance is hardly affected
- Query-based retrieval
  - Read records upon client query using indexes
- Pull client communication
  - Clients are unknown and initiate communications
Routing

- Producer send messages to queues.
- Message Broker notifies one or many consumers about such deliveries.
- Routing strategies:
  a) One-to-one messages (Load Balancing)
     - Messages are routed to one subscriber
     - For data parallelism
  b) One-to-many messages (Fan-out)
     - Messages are routed to all subscribers
     - For task parallelism
Fault tolerance

- **Acknowledgement:**
  - Consumer send an acknowledgement to the Message Broker when they successfully received/completed a message.
  - Message Broker removes any completed message from its queues.

- **Redelivery:**
  - If acknowledgement fails to appear, the Message Broker redelivers it (perhaps to a different consumer).
m3 failed at consumer 2 and is redelivered to consumer 3.

m3 is preserved but stream at consumer 3 is now out-of-order!
Persist

- Keep entire message stream (until reaching size or time limit)
- No need to track consumers
- Let consumers go back in time
  - Database-like
- Log-based Message Broker (e.g. Kafka, Kinesis or DistributedLog)

Forget

- Remove processed messages from stream (immediately after acknowledgement)
- Track consumers to forget old content
- The past is past
  - Volatile, light-weight
- Queue-based Message Brokers (e.g. RabbitMQ, ActiveMQ or HornetQ)
Transmitting Event Streams
Log-based Message Broker

- Message broker that persist messages as logs on disk (distributed, replicated)
- Logs are immutable and append-only
  - Excellent sequential read performance
  - Support parallel, conflict-free reading by multiple clients
- Uncontrolled one-to-many messaging (we do not know who will read a message)
- Replicated Logs
  - For fault tolerance and better parallel read performance
  - Leader-based (to avoid complex replication protocols)
- Partitioned Logs
  - For parallel writes
  - Message ordering guaranteed only within a partition (not between partitions)
  - Partitioning strategies:
    - round-robin, load, partition size, semantic keys, ...
Transmitting Event Streams

Queue-based Message Broker

- Message broker that store messages in queues (distributed, replicated)
- Queues are mutable (usually in-memory) FIFO list data structures
  - Append messages at the end
  - Remove messages from the top
- Controlled one-to-one or one-to-many messaging (usually via JMS\(^1\) or AMQP\(^2\) protocols)
- Replicated/Mirrored Queues
  - For fault tolerance and availability only
    - (no performance gain, because all replicas need to do all appends/removes)
  - Leader-based (to avoid complex replication protocols)
- No partitioning for queues
  - Create multiple queues manually if needed
- Reliability:
  - Send-and-acknowledge handshake with clients
    - (keep messages until successfully acknowledged)

\(^1\)Java Message Service (JMS) 2.0 Specification
\(^2\)Advanced Message Queuing Protocol (AMQP) Specification
Transmitting Event Streams
Message Brokers: Persist or Forget

https://content.pivotal.io/blog/understanding-when-to-use-rabbitmq-or-apache-kafka

Transmitting Event Streams
Message Brokers: Persist or Forget

Persist

Keep entire message stream (until reaching size or time limit)
No need to track consumers
Let consumers go back in time
- Database-like
Log-based Message Broker (e.g. Kafka, Kinesis or DistributedLog)

Forget

Remove processed messages from stream (immediately after acknowledgement)
Track consumers to forget old content
The past is past
- Volatile, light-weight
Queue-based Message Brokers (e.g. RabbitMQ, ActiveMQ or HornetQ)
Topics and Partitions

- **Topics** are logical groupings for event streams.
  - e.g. click-events, temperature-readings, location-signals
  - Every topic is created with a fixed number of partitions.

- **Partitions** are ordered lists of logically dependent events in a topic.
  - e.g. click-events by user, temperature-readings by sensor, location-signals by car
  - Provide “happens-before semantic” for these events
  - Order is valid within each partition, not across different partitions.
  - Are accessed sequentially
    - Producers write new events sequentially.
    - Consumers read events sequentially.

- **Purpose:**
  - Parallelism: to read a topic in parallel
  - Load-balancing: to store the events of one topic on multiple nodes
Topics and Partitions

Every partition has a leader that accepts all writes to that partition and forwards them to its follower replicas.

Leaders for different partitions are distributed in the cluster to allow parallel writes to one topic.

A producer can ask any broker to locate the leader of a partition that it wants to write (done via ZooKeeper).

Distributed Data Management
Stream Processing
ThorstenPapenbrock
Slide 25
Producers and Consumers

- **Producers**
  - Post to *concrete partitions* within a topic (only one leader can take these posts).
  - Define a Partitioner-strategy (on the producer side) to decide which partition is next.
    - Round-Robin Partitioner-strategy is used by default.
    - Custom Partitioner-strategies let producers define semantic grouping functions.

- **Consumers**
  - Read *concrete partitions* within a topic (all broker with that partition can take these reads).
  - Hold an offset pointer for every partition that they read (on consumer side).
  - **Poll and wait** (no callback registration)

“Kafka does not track acknowledgments from consumers [...]. Instead, it *allows* consumers to use Kafka to track their position (offset) in each partition.”

(Book: Kafka - The Definite Guide)
Producers and Consumers

- **Producers**
  - Post to **concrete partitions** within a topic (only one leader can take these posts)
  - Define a **Partitioner-strategy** on the producer side
    - Round-Robin Partitioner-strategy is used by default
    - Custom Partitioner-strategies let producers define semantic grouping functions

- **Consumers**
  - Read **concrete partitions** within a topic (all brokers with that partition can take these reads)
  - Hold an offset pointer for every partition that they read (on consumer side)
  - **Poll and wait** (no callback registration)
Producers and Consumers

- Consumer Groups
  - A group of consumers that processes all events of one topic in parallel.
  - The offsets for a consumer group can be managed by Kafka on server side.
    - A dedicated group coordinator manages offsets, membership, scheduling etc.
    - Consumer commit successfully processed offsets to the group coordinator so that the coordinator can re-assign partitions to consumers.

And in this way, Kafka kind of knows its consumers ...
Producers and Consumers

- **#partitions > #consumer**
  - Consumer take multiple partitions and process them alternatingly.

- **#partitions = #consumer**
  - Every consumer takes one partition; maximum parallelism.

- **#partitions < #consumer**
  - Some consumers idle, because the group reads every partition exactly once.
Different consumers that read the same partition in parallel and at different locations.

Different consumer groups that read same partitions in parallel (and at different locations).
Transmitting Event Streams
Kafka

Log-based Message Broker

send message by appending to log

sequence offsets to ensure ordering

send message by appending to log

receive message by reading log sequentially; when reaching the end, wait and poll again

Only one-to-many messaging!

partitioning (and replication)

= Stream B

offset for B.0 = 4
offset for B.1 = 5
offset for B.2 = 9

Distributed Data Management
Stream Processing

Thorsten Papenbrock
Slide 31
Transmitting Event Streams

Kafka

Log-based Message Broker

Example:
6 TB of disk capacity (= log size)
150 MB/s write throughput
11 h until an event is forgotten (at maximum event throughput!)

No one-to-one scheduling:
Max parallelism bound by number of partitions in a topic!

Events with high processing costs block all subsequent events

Storing a history for events costs memory
Kafka APIs

- Communication with Kafka happens via a specific APIs.
- The API can manage the specifics of the reading/writing process transparently.
  - e.g. offset-tracking (consumers) and partition-scheduling (producers)
- Two options:
  - A rich API that offers high abstraction, but limited control functions.
  - A low-level API that provides access to offsets and allows consumers to rewind them as the need.

Event lifetime

- Configurable:
  - By time of event
  - Max partition size
Transmitting Event Streams
Kafka

Optimizations that make Kafka fast:

- Sequential I/O:
  - Sequential writes avoid disk seek times.
  - Exclusive write access to logs avoids blocking (one writer per log).
  - Sequential reads enable pre-fetching and caching of messages.

- Minimal serialization/deserialization:
  - Standardized binary formats let producers, brokers and consumers use the same data representations without individual modification.

- Zero-copy policy:
  - Data exchange completely in kernel space without copying it to user space avoids costly kernel-space to/from user-space copy processes (due to standardized formats, there is no need to copy messages into user space).

- Batch processing:
  - Batching of data reduces network calls and improves sequential writes.
  - Compression of batches (with LZ4, SNAPPY or GZIP) leads to better compression ratios.
Further reading

- Kafka: The Definitive Guide

Transmitting Event Streams
Message Brokers: Persist or Forget

Persist

- Keep entire message stream (until reaching size or time limit)
- No need to track consumers
- Let consumers go back in time
  - Database-like
- Log-based Message Broker (e.g. Kafka, Kinesis or DistributedLog)

Forget

- Remove processed messages from stream (immediately after acknowledgement)
- Track consumers to forget old content
- The past is past
  - Volatile, light-weight
- Queue-based Message Brokers (e.g. RabbitMQ, ActiveMQ or HornetQ)

Use if **throughput** matters, event processing costs are similar and the **order of messages** is important

Use if **one-to-one scheduling** is needed, **event processing costs differ** and the **order of messages** is insignificant
Transmitting Event Streams
Message Brokers: Persist or Forget

- **Persist**
  - Keep entire message stream (until reaching size or time limit)
  - No need to track consumers
  - Let consumers go back in time
    - Database-like
  - Log-based Message Broker (e.g. Kafka, Kinesis or DistributedLog)

- **Forget**
  - Remove processed messages from stream (immediately after acknowledgement)
  - Track consumers to forget old content
  - The past is past
    - Volatile, light-weight
  - Queue-based Message Brokers (e.g. RabbitMQ, ActiveMQ or HornetQ)

Use if **throughput** matters, event processing costs are similar and the **order of messages** is important

**Wait throughput?**
Yes, because ...
- dumping events to storage instead of routing them to consumers is faster.
- broker does not need to track acknowledgements for every event (only consumers track their queue offset).
- broker can utilize batching and pipelining internally.

**Distributed Data Management**
Stream Processing
ThorstenPapenbrock
Slide 37
Overview
Stream Processing

Transmitting Event Streams

Databases and Streams

Processing Streams

Distributed Data Management
Stream Processing

ThorstenPapenbrock
Slide 38
Databases and Streams
Data Storage – Keeping Systems in Sync

Producer

Events

Volatile write/delete instructions

OLAP System
OLTP System
Search Index
Caches

Distributed Data Management
Stream Processing
ThorstenPapenbrock
Slide 39
Write conflict:
Database and search index are inconsistent, because they don’t share a common leader (that implements e.g. 2PC or MVCC).
Persisting Message Broker

Enables:
- Global ordering of events (→ eventual consistency)
- Fault-safe event delivery
- Backpressure on high load

Producer

Events

OLAP System
OLTP System
Search Index
Caches

Distributed Data Management
Stream Processing
ThorstenPapenbrock
Slide 41
Databases and Streams

Data Storage – Keeping Systems in Sync

Producer

Events

Persisting Message Broker

OLAP System

OLTP System

Search Index

Caches

Distributed Data Management
Stream Processing

Thorsten Papenbrock
Slide 42
Data Change Event Streams

- If events are change operations (writes/deletes) to individual objects (records) it suffices to store only the most recent log entry for each object to rebuild a database.

- **Log Compaction:**
  - Periodically removes outdated log entries from the log
  - Lets the log grow linearly with the data

Message Broker → Database

- If the broker knows what the events mean (e.g. key-value mappings) it can apply log compaction.
  - Event log does not outgrow the maximum buffer size.
  - Message broker becomes a database.

- Implemented by e.g. Apache Kafka
Message Broker as a Database

- **Advantages:**
  - **Data Provenance/Auditability:**
    - The line of events describes the history of every value.
    - Allows to follow a value back in time (e.g. the balance history of a bank account)
      - Fraud protection, temporal analytics, data recovery, ...
  - **Command Query Responsibility Segregation (CQRS):**
    - Events describe what happened (= facts) not their implications.
    - Allows consumers to read/interpret events differently (= different views)
      - Multi-tenant systems, system evolution, data analytics, ...

- **Disadvantages:**
  - Non-standing reads are slow (need to scan and interpret the entire event history).
  - Deleting data means declaring it deleted (actually deleting data is hard).
Complex Event Processing (CEP)

- “Check a stream for patterns; whenever something special happens, raise a flag.”
- Similar to pattern matching with regular expressions (often SQL-dialects)
- Implementations: Esper, IBM InfoSphere, Apama, TIBICO StreamBase, SQLstream

Stream Analytics

- “Transform or aggregate a stream; continuously output current results.”
- Often uses statistical metrics and probabilistic algorithms:
  - Bloom filters (set membership)
  - HyperLogLog (cardinality estimation)
  - HDHistogram, t-digest, decay (percentile approximation)
- Implementations: Storm, Flink, Spark Streaming, Concord, Samza, Kafka Streams, Google Cloud Dataflow, Azure Stream Analytics

Approximation is often used for optimization, but Stream Processing is not inherently approximate!

Bounded memory consumption

Thorsten Papenbrock
Maintaining Materialized Views

- “Serve materialized views with up-to-date data from a stream.”
- Views are also caches, search indexes, data warehouses, and any derived data system
- Implementations: Samza, Kafka Streams (but also works with Flink, Spark, and co.)

Search on Streams

- “Search for events in the stream; emit any event that matches the query.”
- Similar to CEP but the standing queries are indexed, less complex, and more in number
- Implementations: Elasticsearch

Message Passing

- “Use the stream for event communication; actors/processes consume and produce events.”
- Requires non-blocking one-to-many communication
- Implementations: Any message broker; RPC systems with one-to-many support
Spark Streaming (Recap)

Batched Stream Processing

- Reasons:
  - Incremental processing: start processing data that is still being written to
  - Latency reduction: pipeline data to maximizing resource utilization

Stream processing reads the data exactly once and still guarantees fault-tolerance through check pointing and write ahead logs (WAL).
Processing Streams

Spark Streaming (Recap)

```scala
val articles = spark.read.text("/mnt/data/articles/*.csv")

val words = articles.as[String].flatMap(_.split(" "))
val urls = words.filter(_.startsWith("http"))
val occurrences = urls.groupBy("value").count()

occurrences.show()
```

**Streaming input sources:**
- Files: text, csv, json, parquet
- Kafka: Apache Kafka message broker
- Socket: UTF8 text data from a socket
- Rate: Generated data for testing

**Streaming output sinks:**
- Files: "parquet", "orc", "json", "csv", etc.
- Kafka: "kafka" pointing to a Kafka topic
- Foreach: .foreach(...)
- Console: "console"
- Memory: "memory" with .queryName("...")
Storm

- A free and open source distributed real-time computation system (stream processor)
- Competes with Apache Flink in stream processing speed
- Creates a directed acyclic graph (DAG) of “spout” and “bolt” vertices
  - Spout = streaming data source
  - Bolt = data transformation operator

- Designed for:
  - real-time analytics
  - online machine learning
  - continuous computation
  - distributed RPC
  - ETL

- Guarantees:
  - scalability
  - fault-tolerance
  - “best effort”, “at least once”, and “exactly once” processing capabilities
  - ease to set up and operate
A source that streams some text lines

```java
public class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;

    @Override
    public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
        _collector = collector;
        _rand = new Random();
    }

    @Override
    public void nextTuple() {
        Utilities.sleep(100);
        String[] sentences = new String[] {"the cow jumped over the moon", "an apple a day keeps the doctor away", "four score and seven years ago", "snow white and the seven dwarfs", "i am at two with nature"};
        String sentence = sentences[_rand.nextInt(sentences.length)];
        _collector.emit(new Values(sentence));
    }

    @Override
    public void ack(Object id) {
    }

    @Override
    public void fail(Object id) {
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

Text to be streamed

Output format
Processing Streams

Examples

Storm bolds implement UDFs

A flatMap() implementation

```
public class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;

    public static class SplitSentence extends BaseBasicBolt {
        @Override
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declare(new Fields("word"));
        }

        @Override
        public Map<String, Object> getComponentConfiguration() {
            return null;
        }

        public void execute(Tuple tuple, BasicOutputCollector basicOutputCollector) {
            String sentence = tuple.getStringByField("sentence");
            String words[] = sentence.split(" ");
            for (String w : words) {
                basicOutputCollector.emit(new Values(w));
            }
        }

        @Override
        public void fail(Object id) {
        }

        @Override
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declare(new Fields("word"));
        }
    }
```

http://admicloud.github.io/www/storm.html

ThorstenPapenbrock
Slide 52
public class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;
}

public static class SplitSentence extends BaseBasicBolt {
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
    @Override
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String word = tuple.getString(0);
        Integer count = counts.get(word);
        if (count == null) {
            count = 0;
            count++;
            counts.put(word, count);
            collector.emit(new Values(word, count));
        }
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}

public static class WordCount extends BaseBasicBolt {
    Map<String, Integer> counts = new HashMap<String, Integer>();
    @Override
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String word = tuple.getString(0);
        Integer count = counts.get(word);
        if (count == null) {
            count = 0;
            count++;
            counts.put(word, count);
            collector.emit(new Values(word, count));
        }
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}

Another flatMap() implementation
Streaming output: emit every update
Parallelism hint for spouts/bolts

Define the grouping for the input of each bolt:
- **shuffle**: assign randomly
- **field**: assign by field value

Execute on cluster

Execute locally

Runs until explicitly stopped

More on Apache Storm @ [http://storm.apache.org/](http://storm.apache.org/)
In-memory data structure that grows indefinitely large

Implemented as a narrow flatMap() and not as a wide groupBy() to avoid blocking of the pipeline
Goal
- Query and analyze streaming data in real-time (i.e. as data passes by).

Challenges
- **Limited memory resources** (but endlessly large volumes of data)
  - Only a fixed-size window of the stream is accessible at a time.
- **Old data is permanently gone** (and not accessible any more)
  - Only one-pass algorithms can be used.
- **Endlessness contradicts certain operations**
  - E.g. sorting makes no sense, i.e., no sort-merge-joins or groupings (on the entire stream!).
- **Input cannot be re-read or easily back-traced**
  - Fault tolerance must be ensured differently.
Windows

- A continuous segment of the stream usually implemented as a buffer
  - New events oust the oldest events from the window.
- Events within the window can be accessed arbitrarly often.
- Bounded in size usually using a time interval or a maximum number of events
Standing queries

- Persisted queries that are served with volatile event data (reversed DBMS principle)
- Produce a streaming output of “complex events”
- Apply event checking, pattern matching, correlation analysis, aggregation, ...
- Operate on windows
Processing Streams

Windows

**Tumbling Windows**
- Fixed-length, non-overlapping windows
  - New window starts when previous window ended (e.g. successive intervals of 3 seconds or 100 events)

**Hopping Windows**
- Fixed-length, overlapping windows with fixed steps
  - Defined by window length and hop width (e.g. intervals of 3 seconds starting every 2 seconds)

**Sliding Windows**
- Fixed-length, overlapping windows with event-dependent steps
  - Either new events oust old events or events stay for a certain amount of time

**Session Windows**
- Arbitrary-length, overlapping windows
  - Fix start- and end-event (e.g. user logs in; user logs out or session times out)

File-based micro-batching!
How does parallelization happen?

- Different windows can be processed in parallel, but how do we parallelize one window?

- One input stream of events; not pre-partitioned by e.g. HDFS

- Process sequences of logically related events

- The framework does not automatically know which elements belong together and which can be processed in parallel.

- We expect a repartition() here, but for streaming scenarios and overlapping windows, this should be a stable operation in accordance with event/ingestion/processing time and order.

- Different windows can be processed in parallel, but how do we parallelize one window?
Non-Keyed Windows
- Partition a stream into another stream of buckets
- For parallel processing, events need to be replicated (not supported by all streaming frameworks)
  - Usually no parallelization without keying

Keyed Windows
- Partition a stream into multiple other streams of buckets (one per key value)
- Output streams can naturally be processed in parallel without replication
  - Default stream parallelization technique

Also called **partitioned windows**
Non-Keyed Windows

```java
stream
  .windowAll(...) <-- required: "assigner"
  [.trigger(...)] <-- optional: "trigger" (else default trigger)
  [.evictor(...)] <-- optional: "evictor" (else no evictor)
  [.allowedLateness(...)] <-- optional: "lateness" (else zero)
  [.sideOutputLateData(...)] <-- optional: "output tag" (else no side output for late data)
  .reduce/aggregate/fold/apply() <-- required: "function"
  [.getSideOutput(...)] <-- optional: "output tag"
```

Keyed Windows

```java
stream
  .keyBy(...) <-- keyed versus non-keyed windows
  .window(...) <-- required: "assigner"
  [.trigger(...)] <-- optional: "trigger" (else default trigger)
  [.evictor(...)] <-- optional: "evictor" (else no evictor)
  [.allowedLateness(...)] <-- optional: "lateness" (else zero)
  [.sideOutputLateData(...)] <-- optional: "output tag" (else no side...
  .reduce/aggregate/fold/apply() <-- required: "function"
  [.getSideOutput(...)] <-- optional: "output tag"
```

val env = StreamExecutionEnvironment.getExecutionEnvironment
val text = env.socketTextStream("localhost", 4242, "\n")

val windowCounts = text
  .flatMap { w => w.split("\s") }
  .map { w => WordWithCount(w, 1) }
  .keyBy("word")
  .timeWindow(Time.seconds(5), Time.seconds(1))
  .sum("count")

windowCounts.print().setParallelism(1)
env.execute("Socket Window WordCount")

 caso class WordWithCount(word: String, count: Long)
Continuous Query Language

- Developed at Stanford University: [http://www-db.stanford.edu/stream](http://www-db.stanford.edu/stream)
- Used to define standing queries for windows of a stream

```
SELECT count(*)
FROM Requests R [RANGE 1 Day PRECEDING]
WHERE R.domain = 'stanford.edu'
```

“Count the number of requests to stanford.edu for the last 1 day.”

```
SELECT count(*)
FROM Requests R [PARTITION BY R.client_id
ROWS 10 PRECEDING
WHERE R.domain = 'stanford.edu']
WHERE R.url LIKE 'http://cs.stanford.edu/%'
```

“From the last 10 requests of a user to standord.edu, count all her calls to cs.”
Processing Streams

Events and Time

Event Time
- Creation time of the event on the producer (when it occurred)

Ingestion Time
- Arrival time of the event at the stream processor (when it was received)

Processing Time
- Operation time of the event on the stream processor (when it had an effect)

Stream processors (e.g. Flink) let you choose which time to use for windowing!
Processing Streams

Event Time vs. Processing Time
Processing Streams

Events and Time

Event Time
- Creation time of the event on the producer (when it occurred)

Ingestion Time
- Arrival time of the event at the stream processor (when it was received)

Processing Time
- Operation time of the event on the stream processor (when it had an effect)

Unpredictable Time Lag
- Events might be delayed due to ...
  - congestion, queuing, faults, ...
- Events might be out-of-order due to ...
  - message loss and resend, alternative routing, ...
- Event time might be measured differently due to ...
  - multiple clocks in distributed systems, clock skew and correction, ...

Recall lecture on “Distributed Systems”
Solutions

- Assign timestamps as early as possible:
  - producer > leader > time-synced worker > un-synced worker
- Assign multiple timestamps
  - creation-time, send-time, receive-time, forward-time, ...
- Solve time lag programmatically:
  - Exchange a fixed event frequency (e.g. frequency = 1 second)
  - Reasoning over events (e.g. order(X) > pay(X) > deliver(X))

Many events (e.g. sensor or log) carry timestamps naturally

Used to calculate the lag

filming order ≠ narrative order

Distributed Data Management
Stream Processing

ThorstenPapenbrock
Slide 68
Problem

- How does a stream worker know that all events for a certain window have arrived? (as events might be delayed → straggler events)

Solution

- Declare a window as completed if ...
  a) the first event for next window arrives or
  b) a timeout for this window has elapsed.
- Handle straggler events after completion of their window by ...
  a) ignoring them (maybe counting/reporting ignored stragglers) or
  b) publishing an update for their window or
  c) assigning them to the next window.
Issues

- Unbounded:
  - Jobs cannot wait making their output visible until their stream finishes

- Volatile:
  - If a fault occurs, stream data cannot be re-read

Distributed Data Management
Stream Processing

Thorsten Papenbrock
Slide 70
Processing Streams
Fault Tolerance

Microbatching and Checkpointing

- Microbatches (see Spark):
  - Tumbling windows that are treated as batches (cached, checkpointed, ...).
  - Windows represent state that is written to disk and serves to recover from faults.

- Checkpoints (see Flink):
  - Rolling checkpoints that are triggered periodically by barriers in the event stream.
  - Operator state is written to disk and serves to recover from faults.
  - Checkpoints are not tied to particular window sizes.

- Both strategies ensure that every event is processed
  - No event is lost until it produced some output.

- Still problematic:
  - Actions that recover from faults might produce redundant outputs to external event sinks (databases, message brokers, HDFS, ...).
Atomic Commit (revisited)

- Avoid **redundant outputs** using a commit protocol in conjunction with every event sink.
- Commits are logged, which helps to check whether an output happened before.
- Single event commits are cheaper than transaction commits.
- Still a research area with only a few systems supporting it:
  - Google Cloud Dataflow, VoltDB, Kafka (in development)

Idempotence

- Avoid **redundant output effects** using only idempotent output operations.
- Idempotent operation = operation that has the same effect regardless how often it is applied.
- Examples (multiple calls always replace the existing data with itself):
  - Set key to value; Create file with name; Delete resource; Overwrite content with text
- Many non-idempotent operations can be made idempotent:
  - Add an offset/identifier to each output event that identifies redundancy.
Stream-Stream Join

- Task: Join events in stream A with events in stream B.
- Problem: Joins require all events of one side to be randomly accessible, but stream is endless.
- Solution: Window Joins
  - One side of the join is kept in memory as a window (e.g. session window of logged-in users).
  - The other side of the join is probed against the events of that window (e.g. request events to an API).
  - Straggler events are dropped.

Stream-Table Join

- Task: Join events in a stream with events in a database.
- Problem: Database is too large for memory and too slow for stream checks.
- Solution: Database Partitioning/Replication
  - Forward the stream to different partitions/replica that perform different parts of the join.
The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing


Streaming 101: The world beyond batch

A high-level tour of modern data-processing concepts.
By Tyler Akidau, August 5, 2015

The call for proposals is now open for the Strata Data Conference in London, April 29-May 2, 2019.

Editor's note: This is the first post in a two-part series about the evolution of data processing, with a focus on streaming systems, unbounded data sets, and the future of big data. See part two. Also, check out "Streaming Systems," by Tyler Akidau, Slava Chernyak, and Reuven Lax.

Streaming data processing is a big deal in big data these days, and for good reasons. Amongst them:

- Businesses crave ever more timely data, and switching to streaming is a good way to achieve lower latency.
- The massive, unbounded data sets that are increasingly common in modern business are more easily tamed using a system designed for such never-ending volumes of data.
- Processing data as they arrive spreads workloads out more evenly over time, yielding more consistent and predictable consumption of resources.

Despite this business-driven surge of interest in streaming, the majority of streaming systems in existence remain relatively immature compared to their batch brethren, which has resulted in a lot of exciting, active development in the space recently.
1. Data Mining
2. Large-Scale File Systems and Map-Reduce
3. Finding Similar Items
4. Mining Data Streams
   - Sampling and Filtering
   - Counting and Aggregation
   - Estimation
   - Decaying Windows
5. Link Analysis
6. Frequent Itemsets
7. Clustering
8. Advertising on the Web
9. Recommendation Systems
Given is a stream of elements $e_1, \ldots, e_n$. The task is to select a random sample of $k$ elements ($k \leq n$) from the stream, where each element of the stream should have the same probability to be sampled. The size of the stream is not known in advance.

Give an algorithm that solves this problem with $O(k)$ memory and show that each element has the same probability to be sampled.
Homework Log Data Analytics

Distributed Data Management
Stream Processing
ThorstenPapenbrock Slide 78
Homework
Log Data Analytics

Assignment

- Task
  - Data Exploration: Find interesting insights in a log stream, such as
    - the 90th percentile response size
    - average number of requests per hour
    - most popular clients and resources
  - Don’t break the memory!

- Dataset
  - Two month's worth of all HTTP requests to the NASA Kennedy Space Center WWW server in Florida:

- Parameter
  - “java -jar YourAlgorithmName.jar --path access_log_Aug95 --cores 4”
  - Default path should be “./access_log_Aug95” and default cores 4
Homework

Inclusion Dependency Discovery - Rules

Assignment

- **Expected output**
  - Write your discoveries (text + value) to the console.
  - Use the following style for your output:
    <text> : <value>
  - **Example output:**
    - 90th percentile response size : 7265
    - average number of requests per hour : 233
    - most popular client : www.hpi.de
    - most popular resource : www.hpi.de/DDM
Homework

Inclusion Dependency Discovery - Rules

Assignment

- Submission deadline
  - 27.01.2019 23:59:59

- Submission channel
  - ftp-share that we make available via email

- Submission artifacts
  - Source code as zip (Maven project; Java or Scala)
  - Jar file as zip (fat-jar)
  - a slide with your transformation pipeline(s)

- Teams
  - Please solve the homework in teams of two students
  - Provide the names of both students in your submission (= folder name)
Team: **Most Metrics**

**Time Window**

- Daily Timers
- IP with highest number of requests
- Most requested IP
- Test number of requests
- Average duration

**Disc Writing**

- Time Window

Team: **Output Summary**

**Time Window**

- Log Data Analytics Pipeline

Team: **Client Analytics**

**Keyed Session Window**

- FastFlinkStreams: Transformation Pipeline

Team: **Nice Use Case**

**Keyed Window**