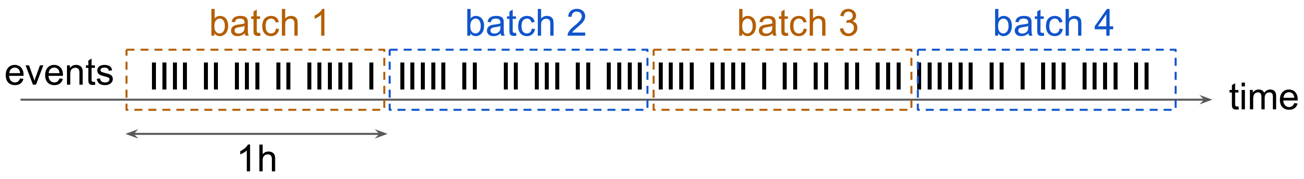
Stream Processing

dr. Rok Piltaver, siječanj 2024

FAKULTET INFORMATIKE I DIGITALNIH TEHNOLOGIJA
SVEUČILIŠTE U RIJECI

Why do we need stream processing?

- **Batch** processing requires **finite** sized **inputs** so it runs **daily** or **hourly**.
- Many data sources are **unbounded** and produce data continuously.
- Some use cases require more **frequent updates**.
- Stream processing is like taking batch processing to the limit:
  each event is it's own "batch".



Why do we need stream processing?

- **Batch** processing requires **finite** sized **inputs** so it runs **daily** or **hourly**.
- Many data sources are **unbounded** and produce data continuously.
- Some use cases require more **frequent updates**.
- Stream processing is like taking batch processing to the limit:
  each event is it's own "batch".

Stream refers to data that is incrementally made available over time:

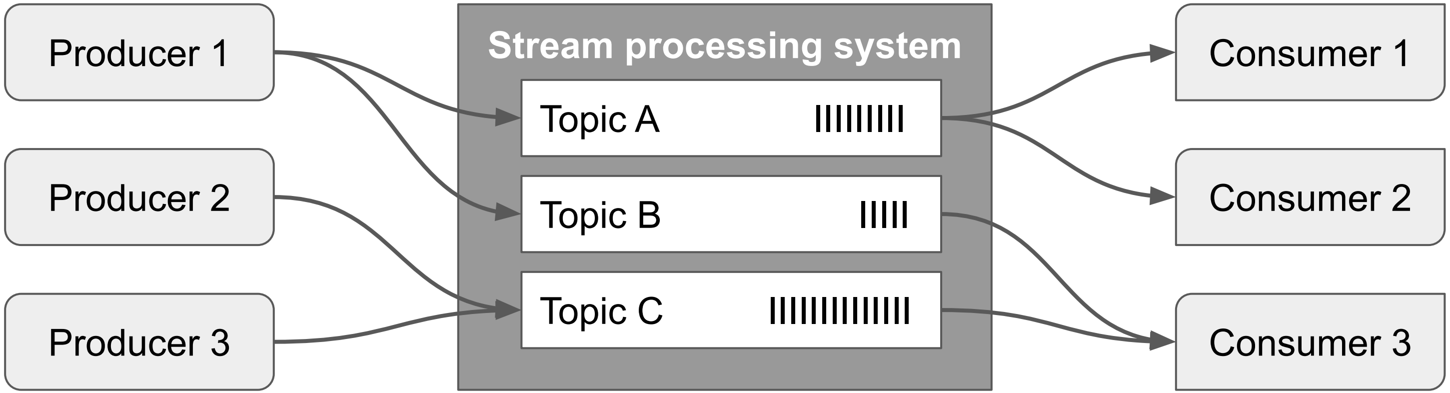
- **stdin** and **stdout** in Unix
- filesystem APIs, e.g. **FileInputStream** in Java
- **TCP** connections
- delivering **audio** and **video** over the internet

Event streams: data management mechanism for unbounded, incrementally processed data.

Transmitting Event Streams

An **event** represents something that happened at some point in time (like a record):

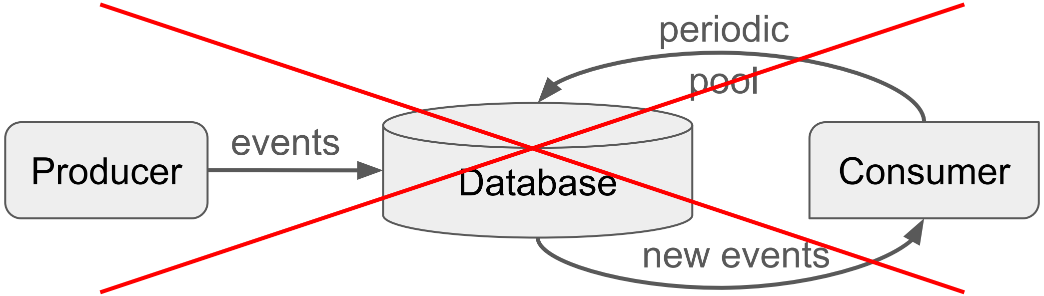
- Represents the basic atomic **unit of data** in stream processing.
- Contains **timestamp** - when it happened according to time-of-day clock.
- Event is **generated** once **by a producer** (publisher or sender).
- Event can be **processed by** multiple **consumers** (subscribers or recipients).
- Related events are usually grouped together into a **topic** or a stream.



Transmitting Event Streams

A file or a **database** would sufficient to connect producers and consumers:

- **A producer writes** every event that it generates to the datastore.
- Each **consumer periodically polls** the datastore to check for new events.
- Polling for changes is **too expensive** at high frequencies (continuous proc.).
- It is better for **consumers** to be **notified** when **new events** appear.
- Databases do offer **triggers** but they are limited.
- **Specialized tools** are used to handle streaming data and efficiently delivering event notifications.



Messaging Systems

Messaging Systems

A **publish/subscribe model** allows multiple producers and/or consumers.

Two questions differentiate messaging implementations:

- What if producers send messages faster than consumers can process them?
  - **Drop** messages
  - **Buffer** the messages in a queue
  - Apply **backpressure** (blocking the producer from sending more messages)
- What happens if nodes crash or temporarily go offline?
  - Some **messages get lost**
  - **Durability** may require some combination of writing to disk and/or replication.

Direct Communication Between Producers & Consumers

Messaging systems that use direct communication without intermediary nodes:

- **UDP multicast**: for low latency apps., application-level protocols can recover lost packets
- **Brokerless messaging libraries** (ZeroMQ): pub/sub messaging over TCP or IP multicast
- **StatsD** & **Brubeck** use UDP messaging for collecting metrics from machines for monitoring
- **Webhooks**: cons. registers callback URL, prod. makes direct HTTP/RPC request for each new event

Require application to be aware of message failure and implement fault tolerance support:

- Such systems assume that producers and consumers are constantly online.
- If a consumer if offline, it **may miss messages**.
- Some protocols allow producer to **retry failed message** deliveries.
- It **may still break down** if producer crashes losing the buffer or messages.

Message Brokers

Message broker (or message queue): a “DB” optimised for message streams:

- Runs as a **server,** producers and consumers connect to it as **clients**.
- **Producers write** messages **to** the **broker.**
- **Consumers** receive them by **reading** them **from** the **broker**.
- Data is centralised so it can easily **tolerate clients that come and go**.
- Question of **durability** is moved to the broker instead:
  - some brokers only keep messages **in memory**,
  - others write them to **disk** so that they are not lost in case of a broker crash.
- A consequence of queueing is that **consumers** are generally **asynchronous**:
  - producer only **waits** for broker to confirm that it has **buffered the message**
  - **does not wait** for the message to be **processed by consumers**.
  - a return message can notify the producer (if it can afford to wait for acknowledgement).

Message Brokers

Some brokers can participate in **two-phase commit protocols** using XA and JTA.

Practical **differences compared to databases**:

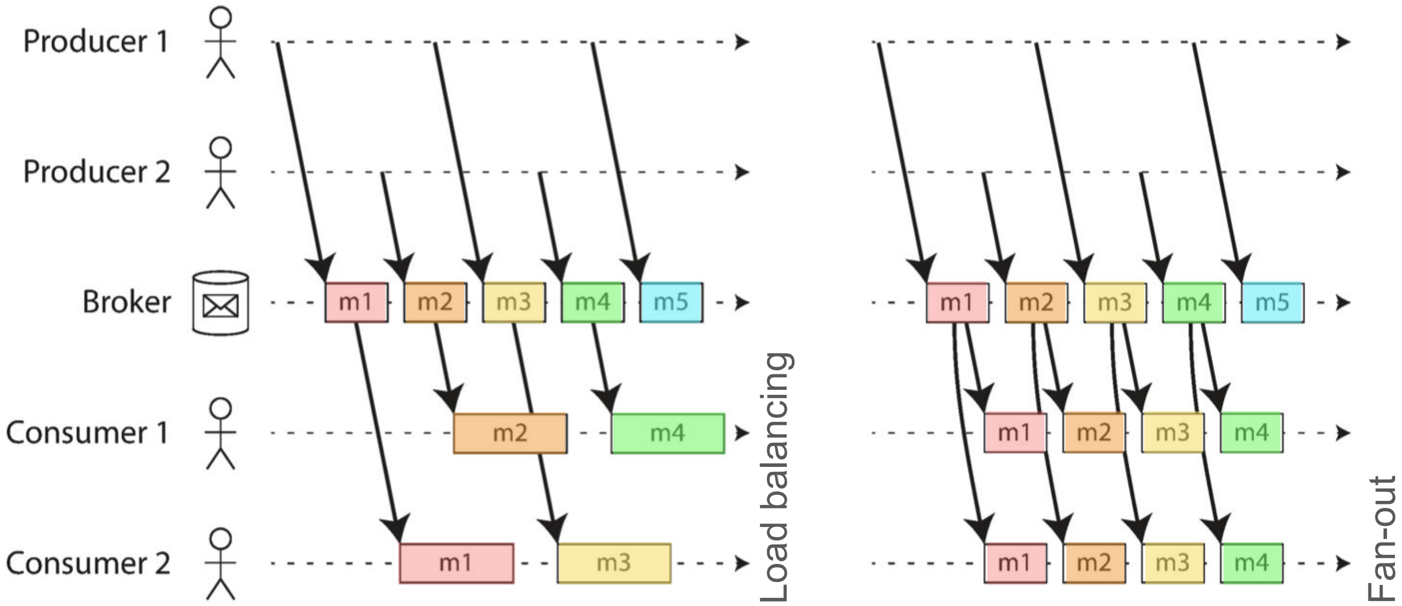
- Broker **automatically deletes a message** when it has been successfully delivered to its consumers. This makes brokers not suitable for long-term storage.
- If broker needs to **buffer a lot of messages**, each individual message takes longer to process, and the overall **throughput may degrade** (assume small working set).
- Brokers often support **subscribing to a subset of topics** matching some pattern.
- Brokers **do not support arbitrary queries**, they notify clients when data changes.

Traditional message brokers (JMS, AMQP standards): RabbitMQ, ActiveMQ, HornetQ, Qpid, IBM MQ, Azure Service Bus, Google Cloud Pub/Sub.

Multiple Consumers

2 patterns for 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.



Multiple Consumers

2 patterns for multiple consumers read messages in the same topic:

- **Load balancing**: each message is delivered **to one** of the consumers.
  - consumers can **share the work of processing** the messages
  - useful when the messages are **expensive to process** (parallelized processing)
  - broker may **assign messages** to consumers **arbitrarily**
  - AMQP: set multiple clients to consume the same queue; JMS: shared subscription
- **Fan-out**: each message is delivered **to all** of the consumers.
  - **independent consumers** receive same messages, without affecting each other
  - streaming **equivalent of several batch jobs** that read the same input file
  - AMQP: exchange bindings, JMS: topic subscriptions
- **Combined**: two groups of consumers subscribe to a topic, **each group** collectively **receives all messages**, but within each group only **one node receives each message**.

Acknowledgments and Redelivery

Even if broker delivers a message, consumer may never processes it (may crash)

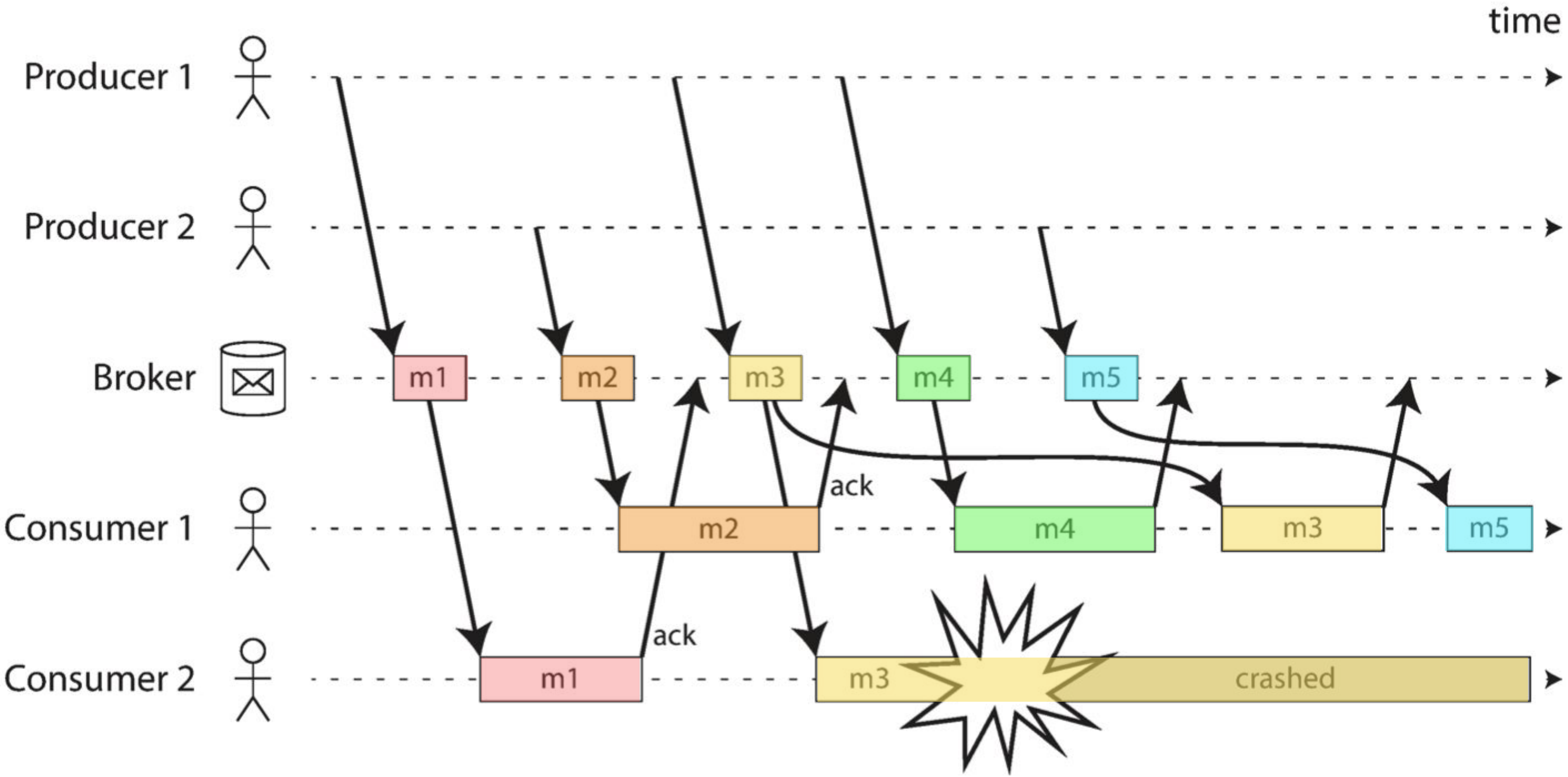
Acknowledgements are used to prevent lost message:

1. client must explicitly tell the broker when it has **finished processing** a msg.
2. then the broker can **remove** it **from** the **queue**

If connection is closed or **times out without acknowledgment**,
broker **redelivers** the message again to another consumer.

- Load balancing with redelivery inevitably leads to **messages being reordered**:
  use separate queue per consumer if there are causal dependencies between msgs.
- Redelivery of msg to another consumer can cause it to be **processed twice** if a message wasn't acknowledged but was actually processed.

Messages Being Reordered



Partitioned Logs

Batch Processing vs AMQP/JMS Message Brokers

Batch process:

- can be run repeatedly without the risk of damaging the input (immutable)
- a new batch process can process old data

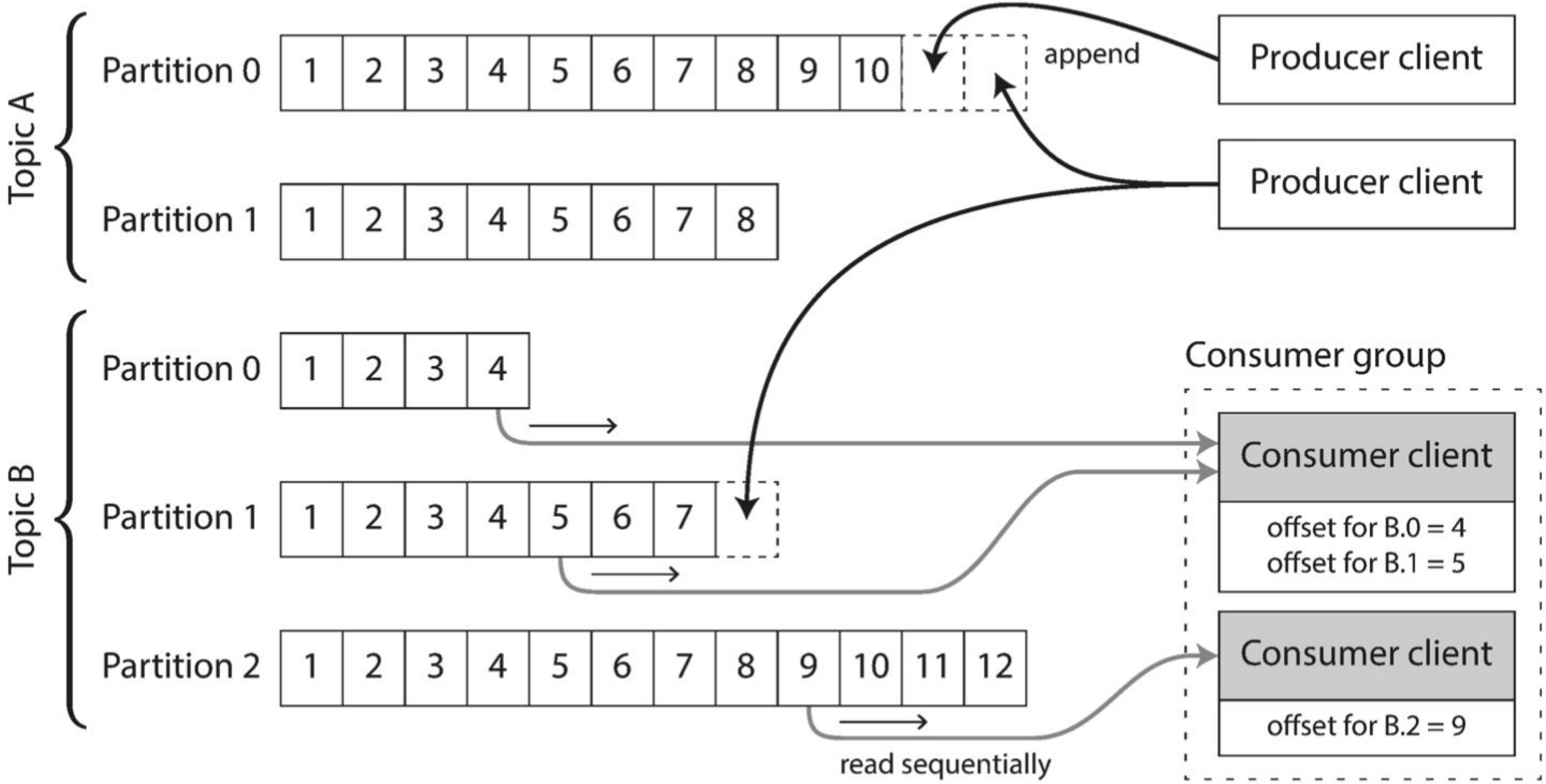
AMQP/JMS-style messaging:

- processing & acknowledging a msg is a destructive operation (deleted msg)
- a new consumer can not read messages that were sent before it was added

Log-based Message Broker

- combine **durable** storage with lightweight **low-latency msg delivery**
- **log** is an append-only sequence of records on disk:
  - producers **append** messages to the log
  - consumers receive msg by **reading log sequentially** or
    wait for notification about new msg if end of the log is reached (like `tail -f`)
  - **reading** a message **does not delete** it from the log
  - **partition** log to scale to higher throughput than a single disk can offer
  - different **partitions hosted on different machines**
  - a **topic** is defined as a **group of partitions** with the same messages type
  - **replicate** log for fault tolerance
- Examples: Apache Kafka, Amazon Kinesis Streams, DistributedLog

Log-based Message Brokers



Fan-out and Load Balancing

Log-based approach trivially supports **fan-out** messaging:
several consumers can independently read the log without affecting each other.

For **load balancing** the broker assigns entire partitions to nodes in the consumer group:

- each **client consumes all messages** in the assigned partition
- **number of nodes** sharing work on a topic ≤ number of log partitions in that topic
- single **slow to process msg** holds up processing of subsequent msgs in the partition

Log-based approach works well if:

- each message is **fast to process** and
- messages **order is important**

JMS/AMQP message broker is preferable if:

- you want to **parallelize** processing on a **message-by-message** basis, and
- where message **ordering is not so important**

Offset

Broker **assigns offset** (monot. inc. seq. num.) **to every message** within each partition

- all **msg** with offset **less than** consumer current **offset** have already been **processed**
- broker **periodically track offset for each consumer** instead of individual ack.

Similar to log sequence number in single-leader DB: broker ~ leader, consumer ~ follower.

If **consumer node fails**:

- **another node** in the consumer group starts consuming msgs **at last recorded offset**
- msgs are **processed twice** if cons. processed msgs but **did not recorded the offset**

Offset is **under** the **consumer's control**, so it can be set backward to an earlier value:

- can experimentally consume production log for **development**, **testing**, or **debugging**
- **replaying** old messages allows **easier recovery** from errors and bugs,
- making it a **good** tool **for integrating dataflows** within an organization

Disk Space

- Occasionally **old segments** are **deleted** or moved to archive to prevent running out of disk space (circular buffer with fixed max storage size).
- Can typically keep a buffer of several days’ or weeks’ worth of messages.
- **Slow consumer** that cannot keep up fall behind the offset of a deleted segment and **loose messages**.
- **Monitor and raise an alert** if a consumer falls behind significantly.
- If a consumer start missing messages, **only that consumer is affected**.
- A side benefit: consumers which are shut down don’t cause runaway large message queues.
- **Throughput remains constant**, since every message is written to disk anyway unlike systems that keep msgs in memory & only write too long queues to disk.

Databases and Streams

Keeping Systems in Sync

Replication log is stream of DB writes, produced by leader as it processes trans. Followers apply that stream of writes to their copy of data (to get accurate copy).

Multiple systems with derived data need to be **kept in sync** with system of record:

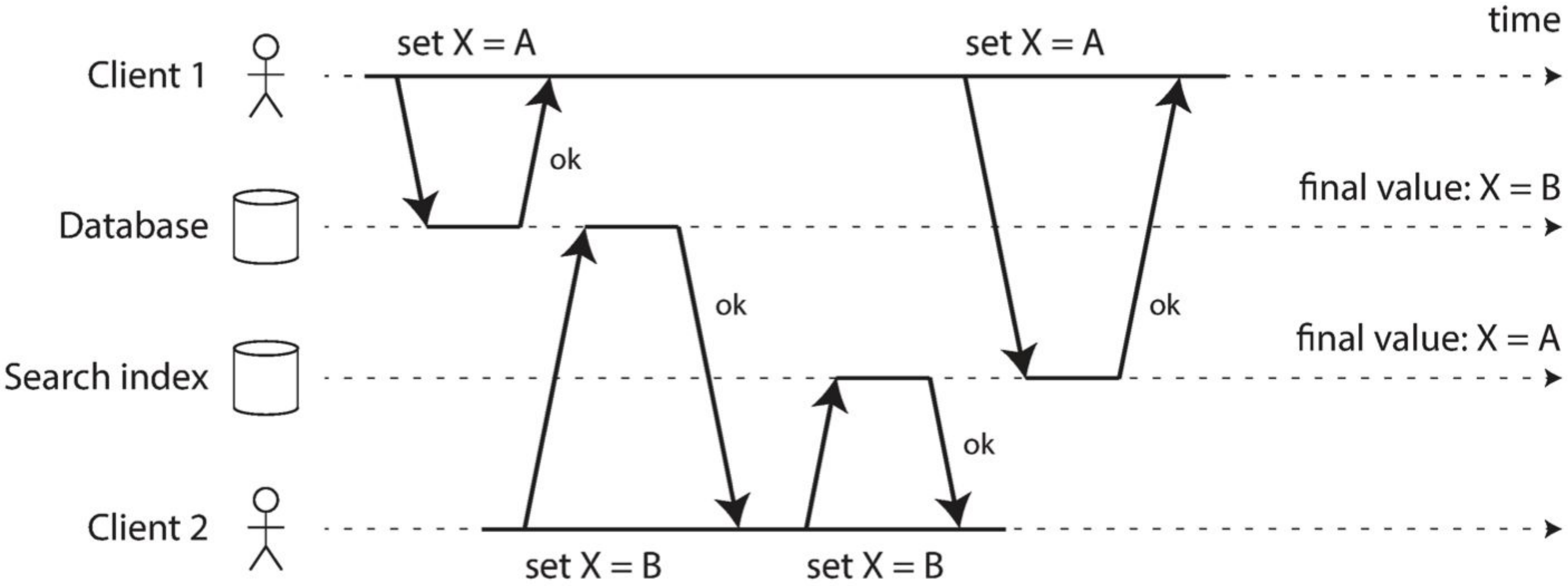
- OLTP systems and data warehouse
- database and cache or search index

Dual writes by clients can have **race conditions:**
concurrent writes silently **overwriting** each other.

Ensuring both writes commit or abort is the **atomic commit issue**:
If only one succeeds the two systems become **inconsistent**.

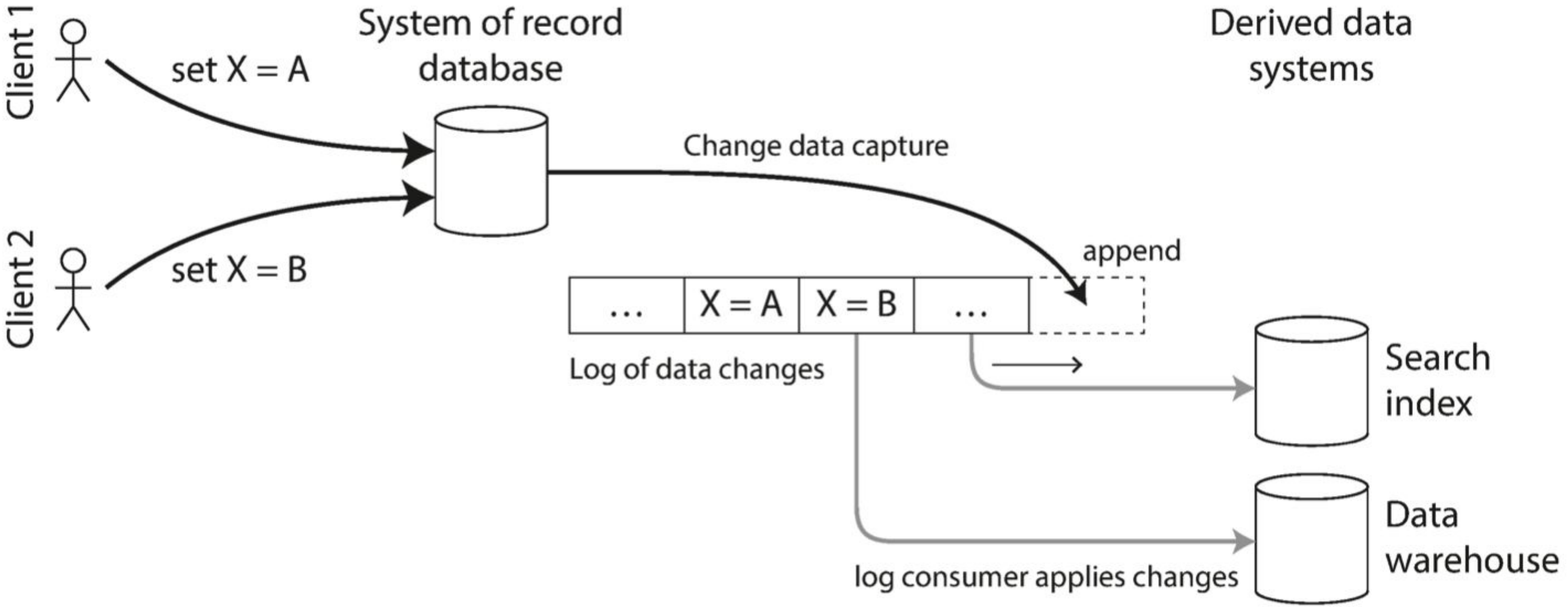
Keeping Systems in Sync

In DB, X is set to A & then to B. At search index writes arrive in the opposite order.



Change Data Capture (CDC)

CDC **exposes all data changes** externally so they can be replicated by other sys.



Change Data Capture (CDC)

CDC **exposes all data changes** externally so they can be replicated by other sys.

- CDC provides changes immediately **as a stream**
- pata changes can be **continually applied by another system**
- A log-based message broker can **preserve the order** of messages
- One database becomes the single **leader**, and the others act as **followers**.
- Having **snapshots to start from** is a lot faster for initial synchronization than replaying all change (requires a known position or **offset in the change log**).
- **Log compaction** is beneficial for keeping down the size of logs.
- Implementations:
  - using database **triggers** (significant performance overheads)
  - **parsing** the **replication log**: Bottled Water, Debezium, Mongoriver, GoldenGate.
  - newer **DBs support CDC** functionality directly

Event Sourcing

- Event sourcing records logical, immutable actions at a higher level.
- The event store can only be appended - updates/deletes are forbidden.
- Does not record entire records:
  - full history is needed to get the complete current state of a record
  - log compaction may not be possible
- Events start as commands

  after validation is completed, they become immutable events

Immutable Events

Advantages:

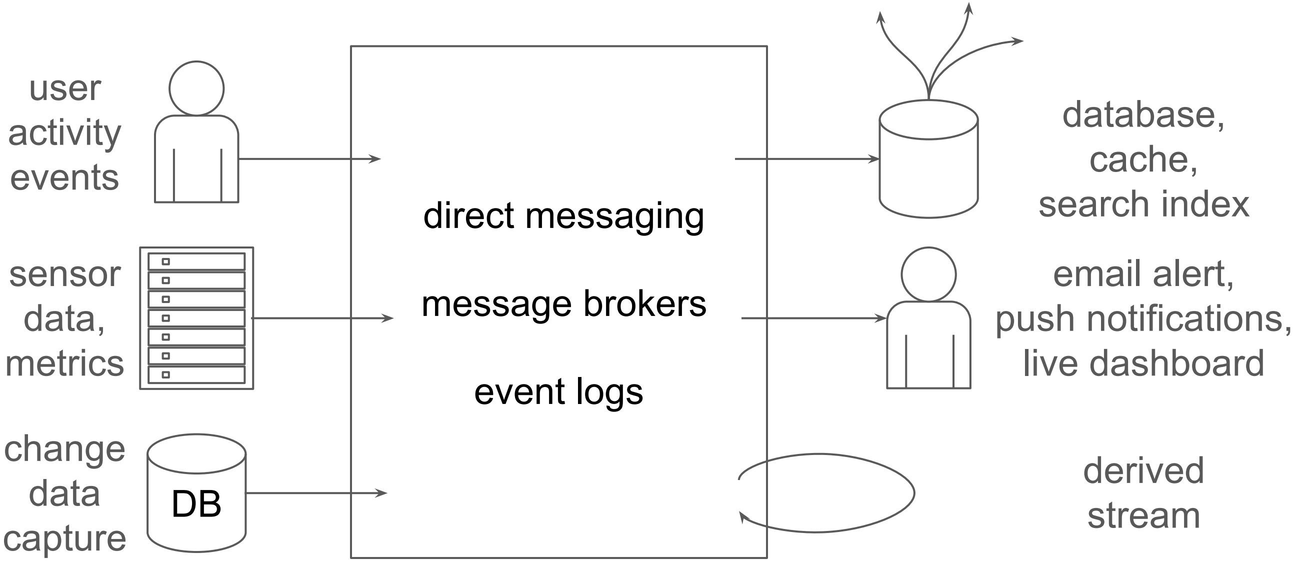
- used in **accounting** (append-only ledger)
- help **debugging** and provide a **richer history for analytics**
- **deriving several views** from the same event log (explicit translating process)
- **separating** how data is **written and read** can offer a lot of flexibility
- good self-contained event design may **eliminate** the need for **multi-object transactions**

Disadvantages:

- event log consumers are updated asynchronously, so a **read** after a write **may be stale**.
- workloads with lots of updates and deletes may be hard (fragmentation, compaction, gc)
- certain circumstances require deletion (privacy rules around someone closing their account)

Processing Streams

Processing Streams



Stream vs Batch Processing

Streams are similar to MapReduce and dataflow engines in terms of:

- patterns for **partitioning** and **parallelization**
- basic **mapping operations**: transforming and filtering records

The crucial difference: a **stream never ends**:

- **sorting does not make sense** with an unbounded dataset
  sort-merge joins cannot be used
- **fault-tolerance mechanisms** must also change:
  stream job running for years can't be restarting from beginning after crash

Uses of Stream Processing 1

Monitoring: alert if certain things happen:

- Manufacturing systems (monitor status of machines in factory, malfunctions)
- Fraud detection systems (credit card usage patterns)
- Trading systems (examine price changes, execute trades by rules)

Uses of Stream Processing 2

Stream analytics:

- **Aggregation** and computing statistical **metrics** over a large number of events:
  - Measuring the **rate** of some type of event (avg. queries/s over last 5 min)
  - Calculating the **rolling average** of a value over a period (p99th response time)
  - **Compare** current statistics **to previous time intervals** (trends & unusual metrics)
- **Averaging** over a few minutes smoothes out irrelevant fluctuations.
- **Window**: the time interval over which you aggregate.
- Use **probabilistic algorithms** (requiring less memory than exact algorithms).
- **Frameworks** with analytics in mind: Apache Storm, Spark Streaming, Flink, Concord, Samza, and Kafka Streams.
- **Hosted services**: Google Cloud Dataflow and Azure Stream Analytics.

Uses of Stream Processing 3

Search on streams:

- **Search** for **individual events** based on complex criteria
- Formulate **search query** in advance
  continually match stream of items against the query.
- Examples:
  - **Media monitoring** services subscribe to feeds of news articles and search for news mentioning companies, products, or topics of interest (full-text search).
  - Users of **real estate websites** can ask to be notified when a new property matching their search criteria appears on the market.
- Can be implemented using the percolator feature of Elasticsearch.

Uses of Stream Processing 4

Complex event processing:

- CEP allows you to specify **rules** to **search for patterns** of events in a stream (like regular expression search for patterns of characters in a string)
- Declarative **queries** (event pattern descriptions in SQL) are stored **long-term**.
- **Events** from the input streams continuously **flow past queries**.
- CEP internally maintains a **state machine** that performs the required matching.
- When a **match** is found, the engine **emits a complex event**.
- Implementations: Esper, IBM InfoSphere Streams, Apama, TIBCO StreamBase, and SQLstream.

Uses of Stream Processing 5

Maintaining materialized views:

- Derive an **alternative view onto a dataset** (so you can query it efficiently) and **update** that view **whenever** the underlying **data changes**
- Examples:
  - Stream of changes to a database can be used to **keep derived data systems** (cache, search index, data warehouse) **up to date** with a source database.
  - In **event sourcing**, application state is maintained by applying a log of events - application state is a kind of a materialized view.
- Requires **all events** over an arbitrary time period (not just a window).
- Samza and Kafka Streams support this kind of usage.

Reasoning About Time

Batch processing:

- **event timestamps** are used if time is relevant
- the time of processing is irrelevant
- results are therefore **deterministic**

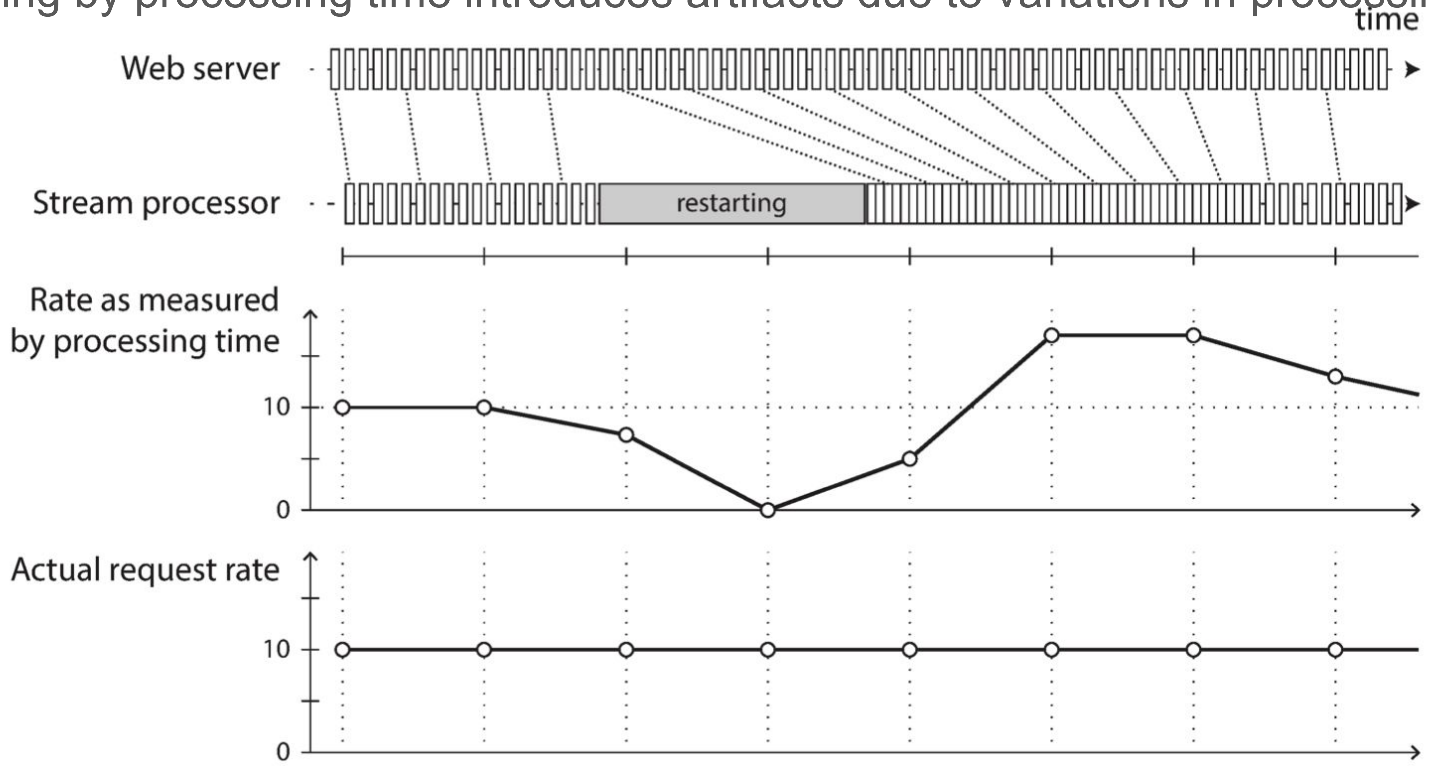
Stream processing frameworks use **processing time** to determine windowing:

- **Event time != processing time** (unbounded message delays).
- reasonable if delay between event creation and processing is negligibly short
- **breaks down if** there is any significant **processing lag** (queuing, restarts)

It's better to take into account the **original event time** to count rates.
But you can never be sure **when you have received all the events**.

Event Time vs Processing Time

Windowing by processing time introduces artifacts due to variations in processing rate.



Reasoning About Time

Process all messages in a window between 1:00 and 1:01.

- How do you know when you have **received all the events**?
- Declare window complete after there are no new events for a while (**timeout**).
- But **delayed events** may still come later (e.g. due to network interruption).
  - **drop stragglers** that arrive late + **monitor** the number of dropped events
  - **voiding prior output** and **publish a correction** (updated for win. with stranglers)
  - **producer send messages** "There will be no more msgs with timestamp < t"

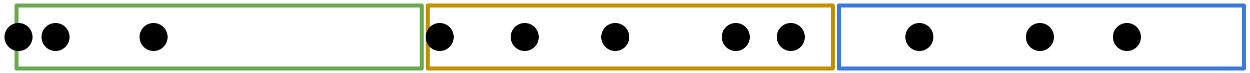
Whose clock to use? Track three times:

- To = when event **occurred** per producer’s clock
- Ts = when message was **sent** per producer’s clock
- Tr = when message was **received** per broker’s clock
- **T ~ To + (Ts - Tr)** if network delay is short & producers clock was not adjusted

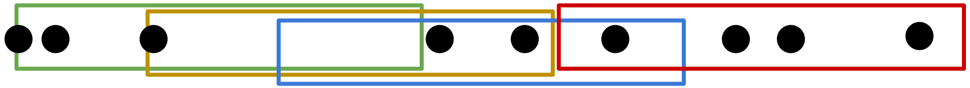
Stream Processing Windows

Several types of windows are in common use:

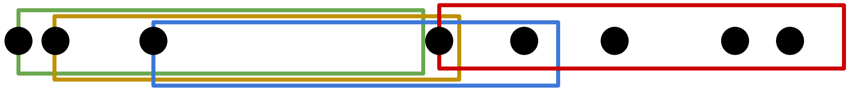
- **Tumbling** window - adjacent time slots



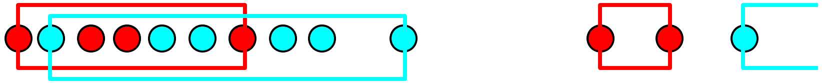
- **Hopping** window - overlapping time slots



- **Sliding** window - boundaries are not fixed - they are sliding



- **Session** window - e.g. per user events until a period of inactivity



Stream Joins

Stream-stream join (window join):

- e.g. click-through rate for each URL in the search results (search & click streams)
- Select a window for join (can’t wait forever)
- **state is maintained** (events in window) to join events from two different streams

Stream-table join (stream enrichment):

- e.g. join user activity & user profiles table to get favourite URL per age group
- keep a **local** copy of the table using **change data capture** (window from beginning)

Table-table join (materialized view maintenance)

- e.g. Tweeter timeline cache from tweets and (un)follows streams/tables
- streams updating a materialized view of a join between two tables
- changes to either stream need to inform potentially multiple rows from the other table in the join output

Time-Dependence of Joins

Joins require stream processor to:

- maintain state based on one join input (search and click events, user profiles)
- and query that state on messages from the other join input.

Order of events that maintain the state is important

- event **order within partition** is preserved
- **no ordering** guarantee **across** different **partitions**
- In which **order** are events processed if they happen **around similar time** on different streams?
- **Join** are **nondeterministic** if event ordering across streams is undetermined.
- **Slowly changing dimension:** using unique identifier for each version of joined record (tax rate changes) makes join deterministic, but log compaction is not possible

Fault Tolerance

Batch processing:

- If MapReduce task/**job fails**, it can simply be **restarted** on another machine.
- **Input** files are **immutable** and the **output** is written to a **separate file**.
- **Exactly-once-semantics**: even if records are processed multiple times, output is as if they had been processed once (effectively-once).

Stream processing:

- waiting until a task is finished before making its output visible is not an option because streams are infinite.

Fault Tolerance

Microbatching:

- Break stream into **small blocks**, treat each block like a miniature batch process.
- Batch size is typically around **one second** (still fast but not too much overhead).
- Implicitly **provides tumbling window** equal to the batch size;
  explicitly **carry state over** microbatches for longer windows.
- This technique is used in Spark Streaming.

Checkpoints:

- Periodically **generate checkpoints** of state and write them to **durable storage**.
- If a stream operator crashes, it can **restart from** its **most recent checkpoint** and **discard** any **output** generated between the last checkpoint and the crash.
- Checkpoints **triggered by barriers** in the message stream (arbitrary window size).
- Used in Apache Flink.

Fault Tolerance

- Microbatching and checkpointing provide **exactly-once semantics**.
- After output leaves the stream processor (writing to DB, send msg to broker or email to user), the framework **can not discard output** of a failed batch.
- **Restarting** failed task causes the **external side effect** to happen **twice**.
- Things either need to happen atomically or none of them must happen: distributed transactions, two-phase commit.
- The goal: **discard** partial **output of failed tasks** so they can be safely retired.
- Messaging systems provide support for efficient **atomic commit internally** between streams.
- Used in Google Cloud Dataflow, VoltDB, Apache Kafka.

Idempotence

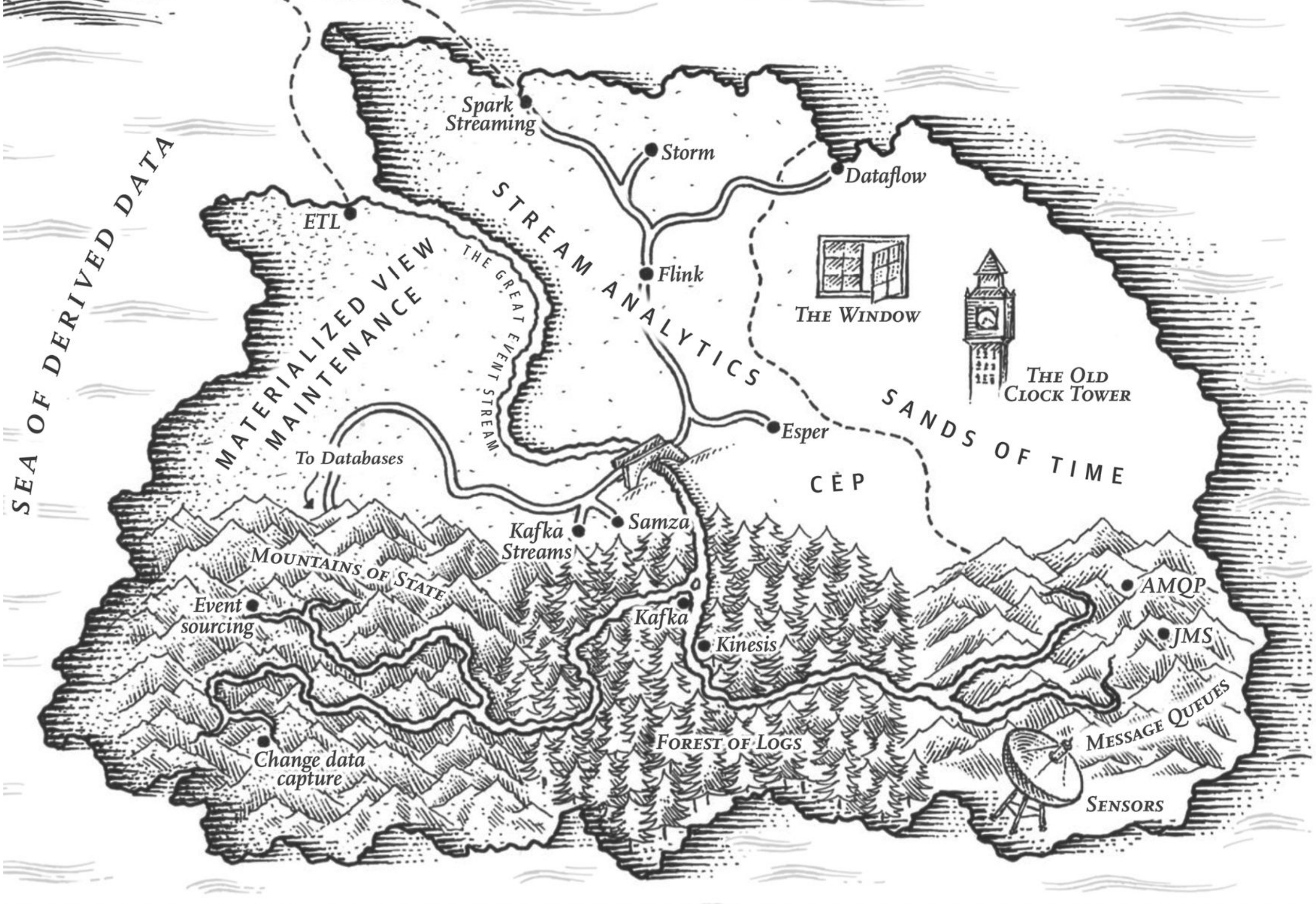
Idempotent operation is one that you can be performed multiple times, but it has the same effect as if it was performed only once.

- If operation is not naturally idempotent, it can often be made idempotent - with extra **metadata** you can **prevent repeating actions** more than once.
- Idempotence is an alternative to distributed transactions for achieving exactly-once semantics with only a **small overhead**.

Rebuilding State After a Failure

Stream process that requires **state must be able to recover** it after a failure.

- Keep the state **in a remote datastore** and replicate it - this is slow.
- **Keep state local** to the stream processor and **replicate it periodically**.
  - Periodically capture **snapshots** and write them **to durable storage** e.g. Flink uses HDFS to make state durable.
  - **Send state changes to** a **dedicated topic** with log compaction e.g. Samza and Kafka Streams use Kafka topics to replicate state.
  - Redundantly **process** each input **message on several nodes** e.g. VoltDB does this to replicates state over multiple workers.



Sea of Derived Data
Spark Streaming
Storm
Dataflow
ETL
Stream Analytics
Materialized View Maintenance
The Great Event Stream
Flink
The Window
The Old Clock Tower
Sands of Time
Esper
To Databases
CEP
Kafka Streams
Samza
Mountains of State
Event sourcing
Kafka
Kinesis
AMQP
JMS
Message Queues
Forest of Logs
Change data capture
Sensors