Stream Processing

Enterprise Architectures for Big Data

Duality of Streams and States

Relationship between the current application state and an event stream

Streams
record history

1. e4 e5
2. Nf3 Nc6
3. Bc4 Bc5
4. d3 Nf6
5. Nbd2

Tables
represent state

“The sequence of moves”

“The state of the board”

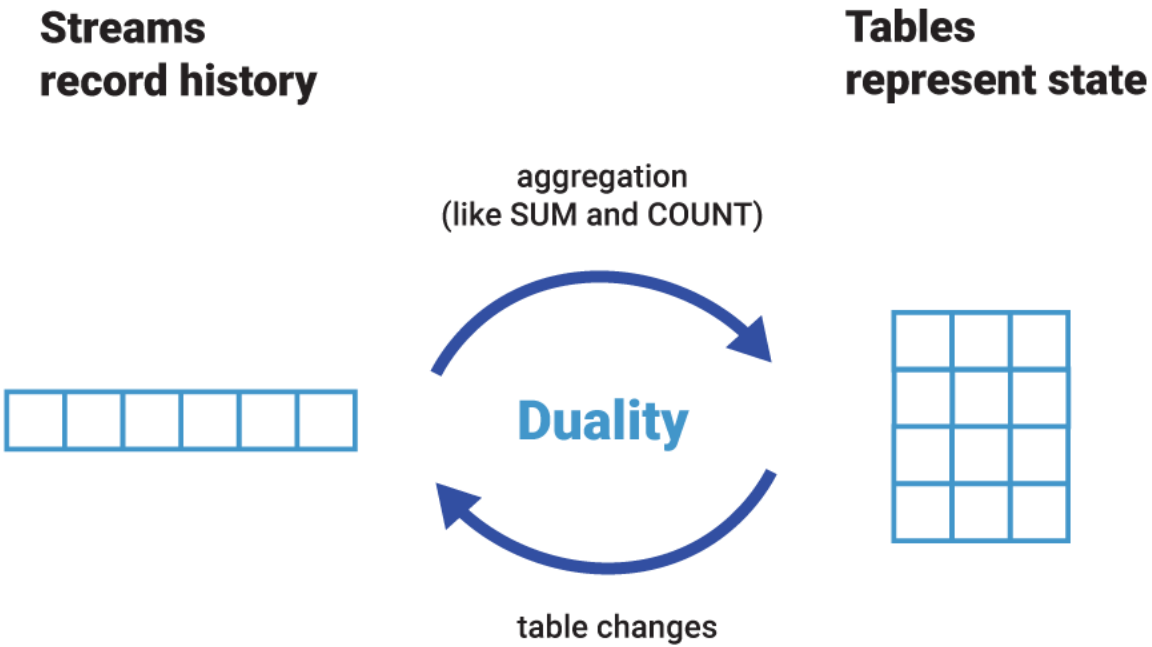
Relationship between the current application state and an event stream

$$state(now) = \int_{t=0}^{now} stream(t)\ \mathrm{d}t$$

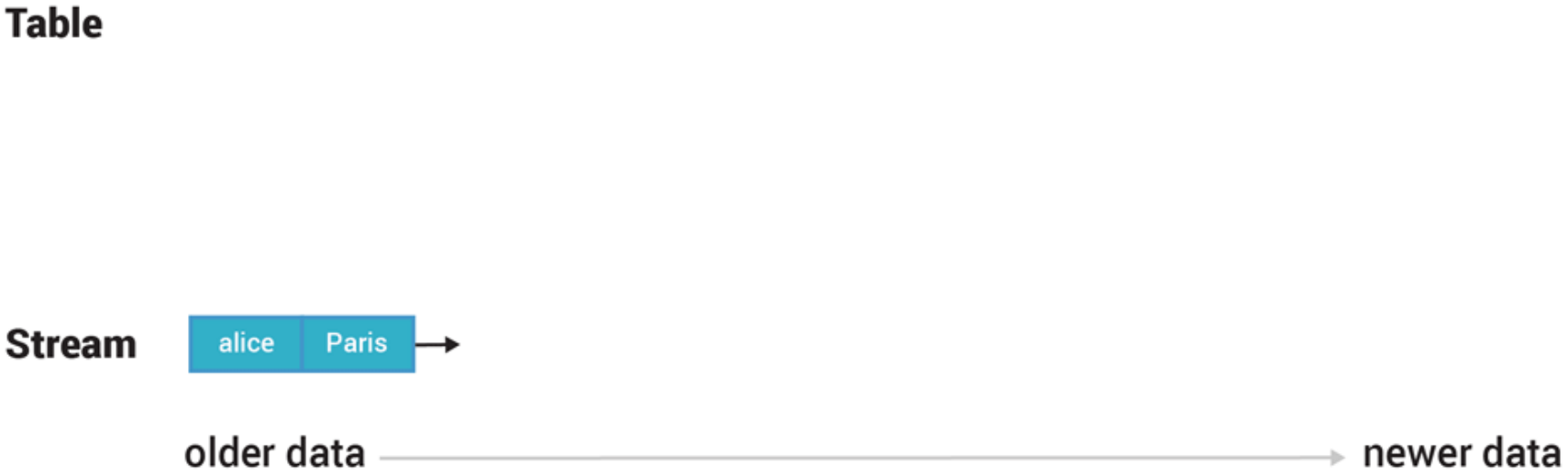
$$stream(t) = \frac{\mathrm{d}\ state(t)}{\mathrm{d}t}$$

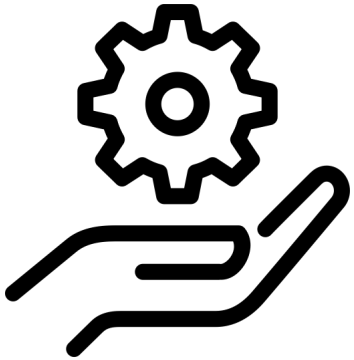
Source: Kleppmann (2017), p. 460

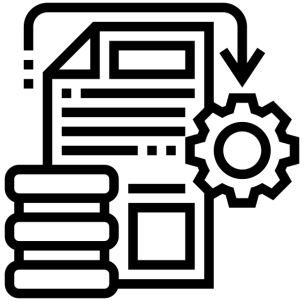
Relationship between the current application state and an event stream



Relationship between the current application state and an event stream





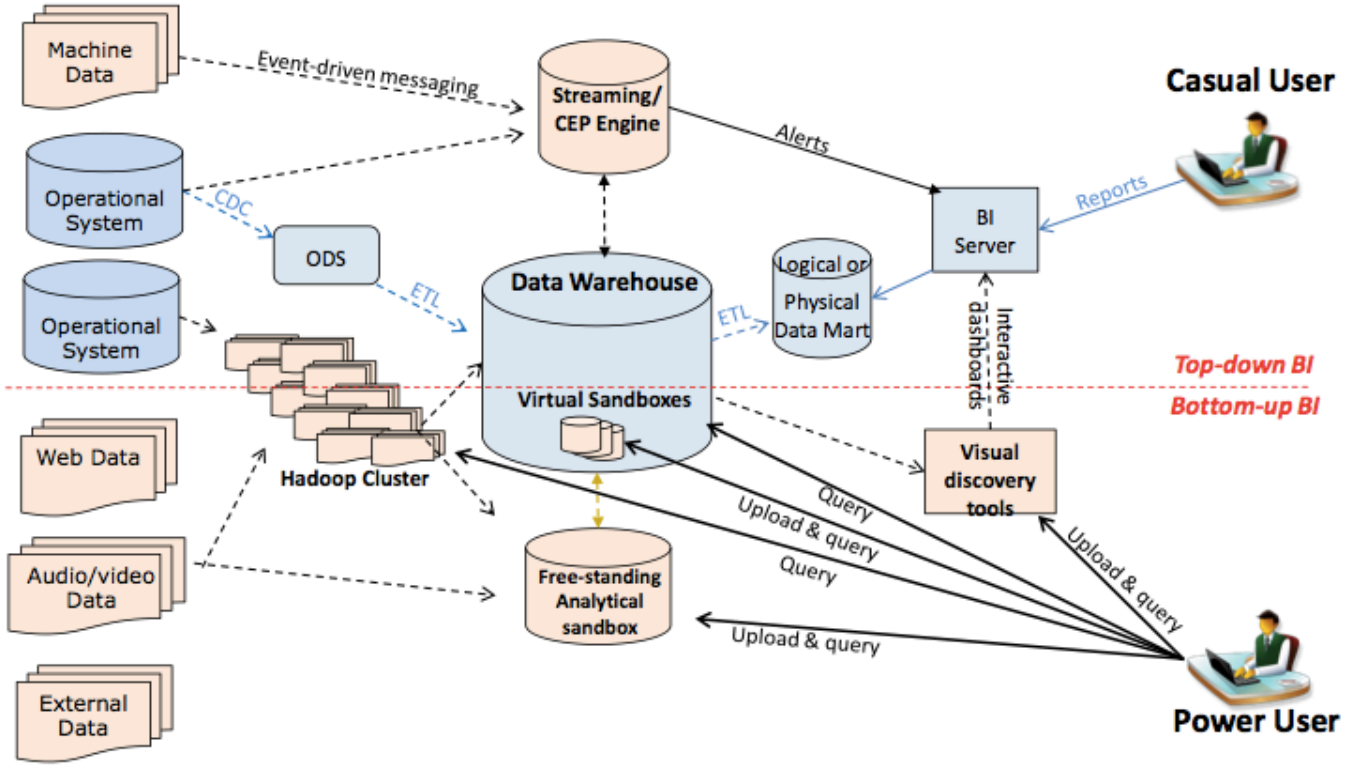


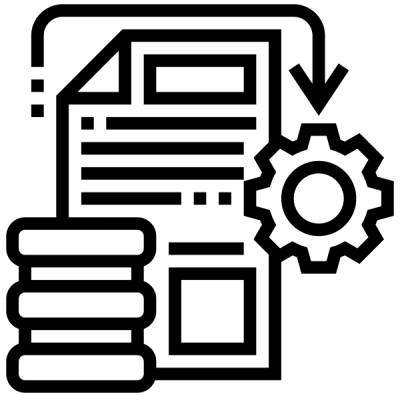
Service, Batch and Stream

Three Types of Systems

1. **Services** (online systems)
   - A service waits for a request or instruction from a client to arrive.
   - When one is received, the service tries to handle it as quickly as possible and sends a response back.
   - **Response time** is usually the primary measure of performance of a service, and **availability** is often very important.
2. **Batch processing systems** (offline systems)
   - A batch processing system takes a **large amount of input data**, runs a job to process it, and produces some output data.
   - Jobs often take a while (from a few minutes to several days), so **there normally isn't a user waiting** for the job to finish.
   - Instead, batch jobs are often scheduled to **run periodically** (for example, once a day).
   - The primary performance measure of a batch job is usually **throughput**.
3. **Stream processing systems** (near-real-time systems)
   - Stream processing is somewhere between online and offline/batch processing (so it is sometimes called near-real-time or nearline processing).
   - Like a batch processing system, a stream processor consumes inputs and produces outputs (rather than responding to requests).
   - However, a **stream job operates on events shortly after they happen**, whereas a batch job operates on a fixed set of input data.
   - This difference allows stream processing systems to have **lower latency** than the equivalent batch systems.

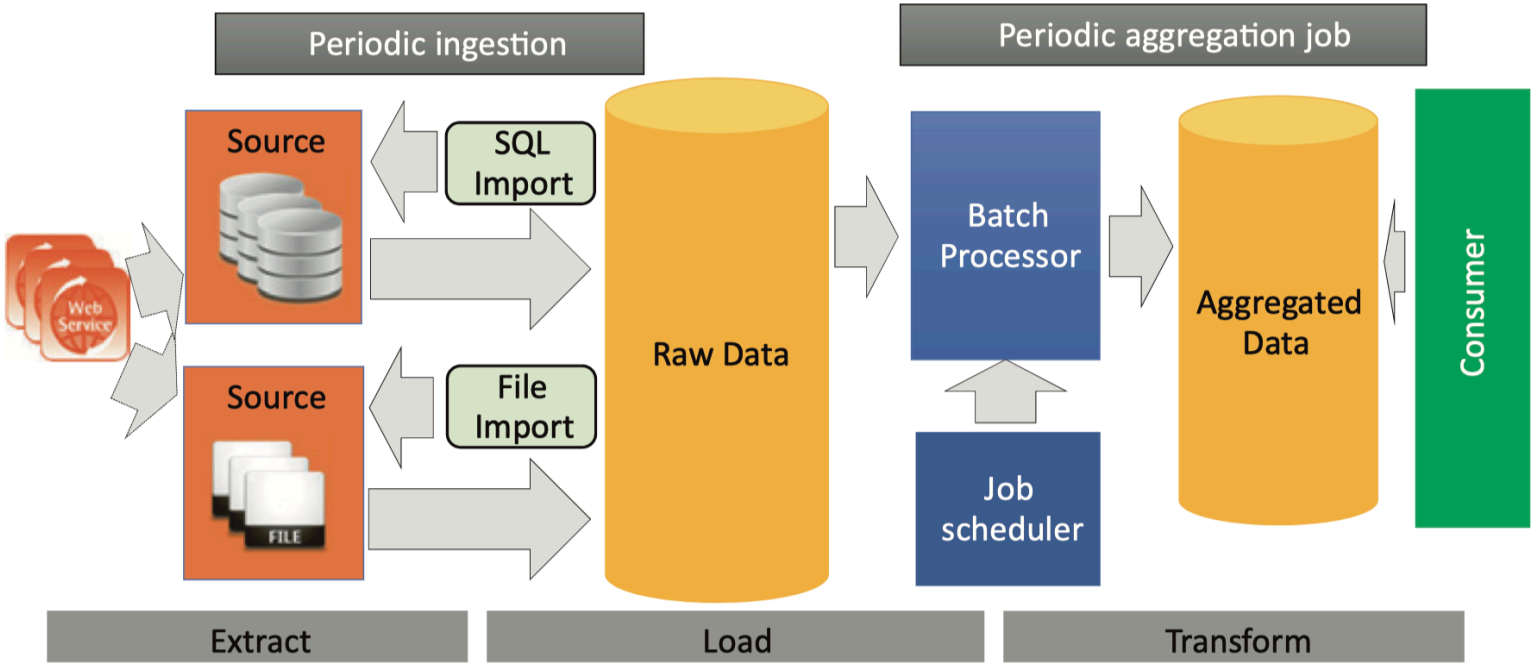
Big Data, Streaming, and Data Warehouse





Batch Processing

Batch Architecture

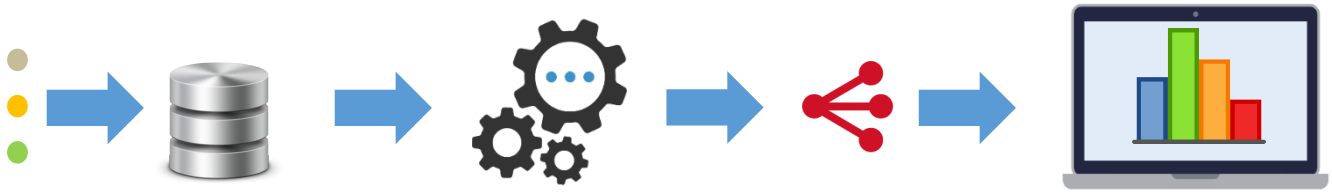


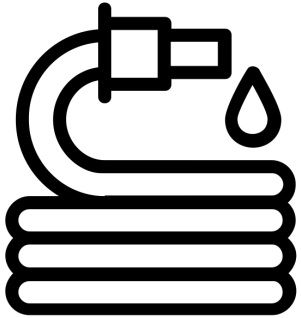
Batch Processing

- „Volume“
- Cost-effective
- Efficient
- Easy to reason about: operating on complete data

But:

- **High latency**: jobs periodically (e.g. during night times)

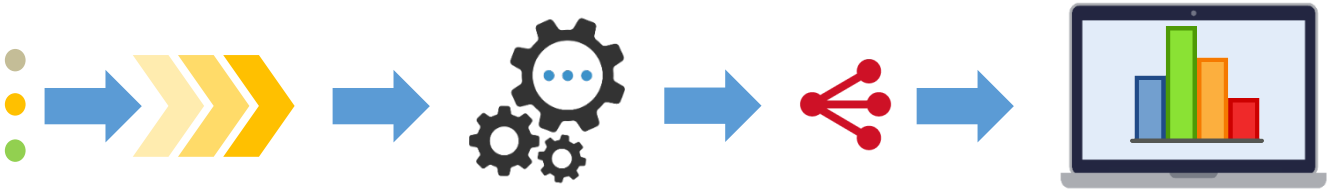




Stream Processing

Stream Processing

- „Velocity“
- Low end-to-end latency
- Challenges:
  - Long-running jobs: no downtime allowed
  - Asynchronism: data may arrive delayed or out-of-order
  - Incomplete input: algorithms operate on partial data
  - More: fault-tolerance, state management, guarantees, …



Source: Gessert (2016)

Big Data And Stream Analytics

- Data-in-motion analytics and real-time data analytics
- One of the V in Big Data = Velocity
- Analytic process of extracting actionable information from continuously flowing/streaming data

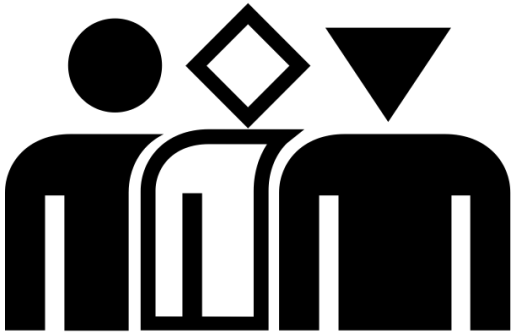
- Why Stream Analytics?
  - It may not be feasible to store the data
  - It may loose its value if not processed immediately

Data Streams

- Data that is “unbound”: Never finished
- “Stream” refers to data that is incrementally made available over time.
  - Like in the stdin and stdout of Unix
  - Like Generators in Python

Batch	Stream
File	Event
Filename identifies a set of related records	Events are usually grouped together into a topic or stream
Pull Data	Push Data

Source: Kleppmann (2017), p. 479

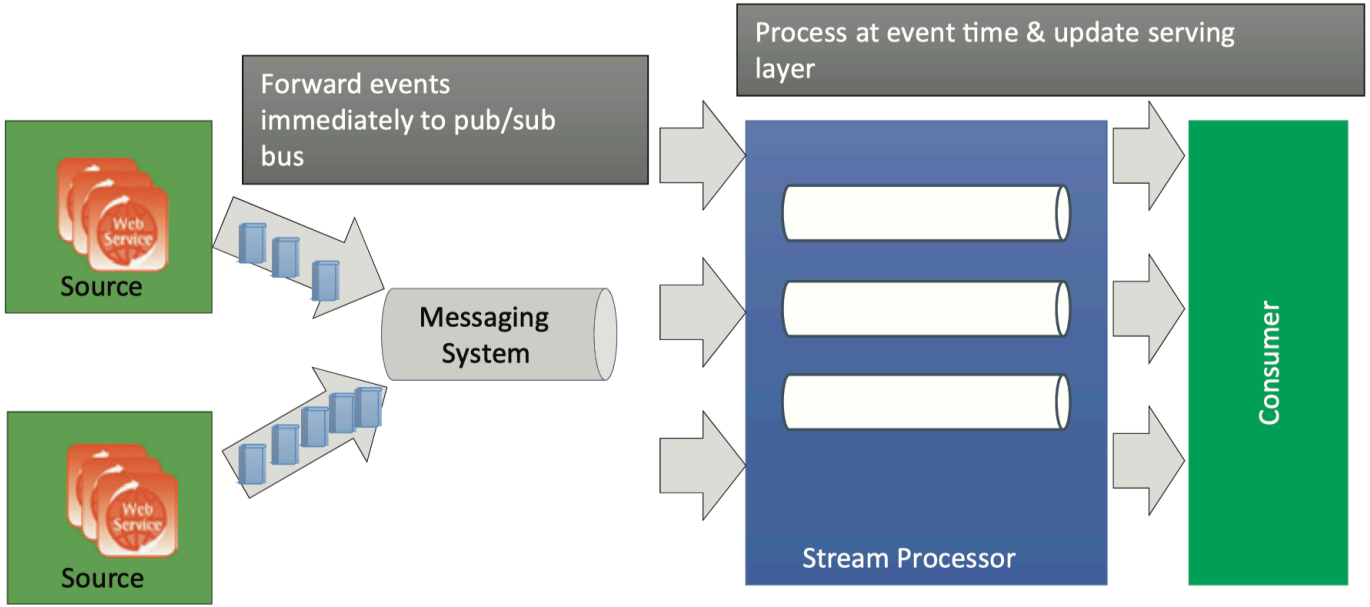


Different Types of Stream Processing

Two Types of Stream Processing

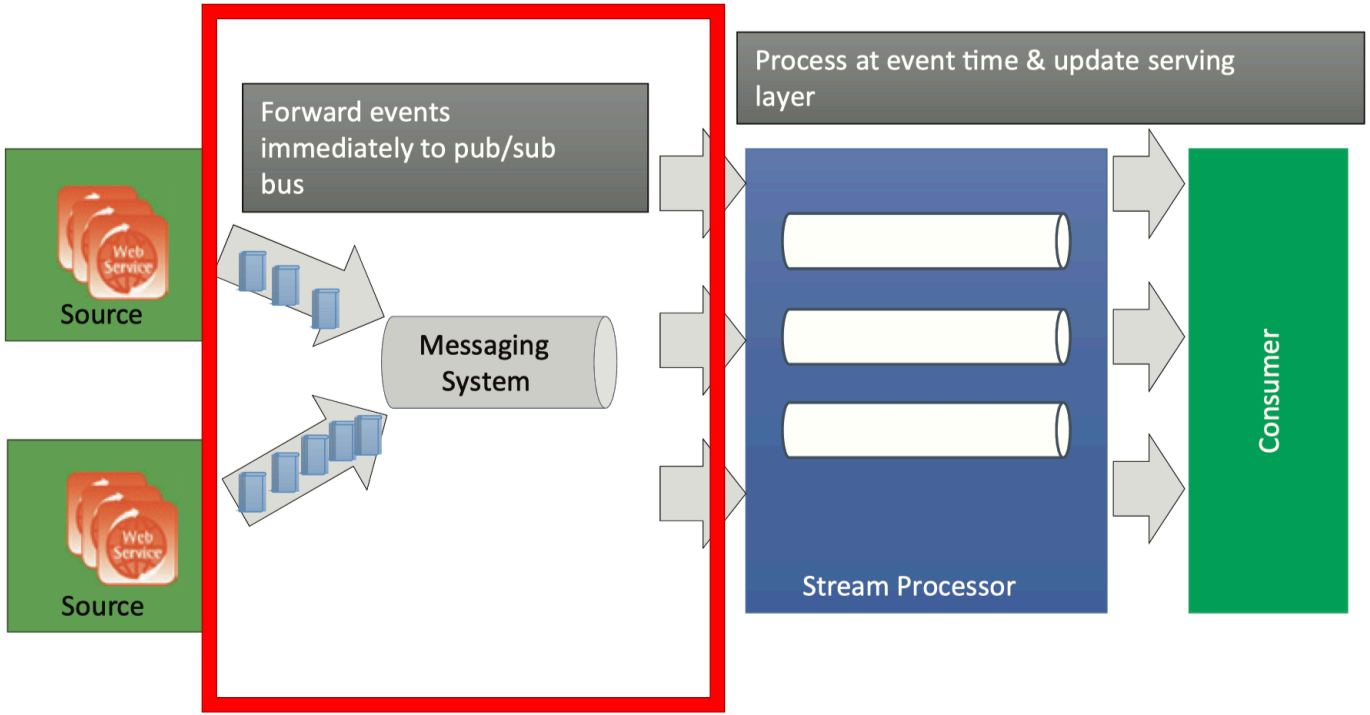
- Transmitting Event Streams
- Processing Streams

Stream Analytics Architecture



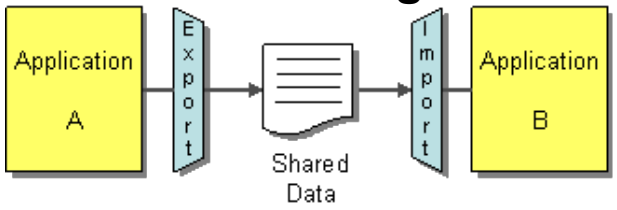
Transmitting Event Streams

Stream Analytics Architecture

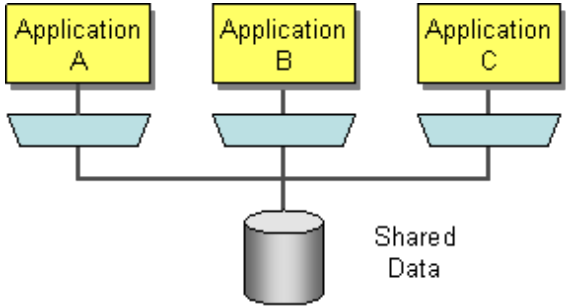


Integration Styles

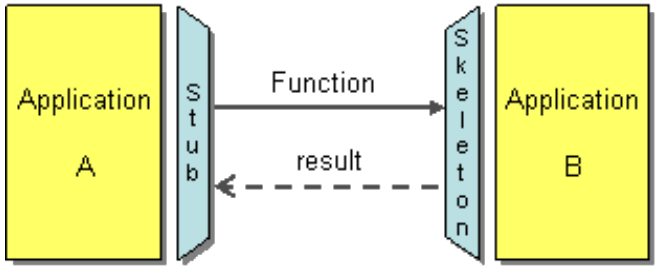
File Transfer Integration



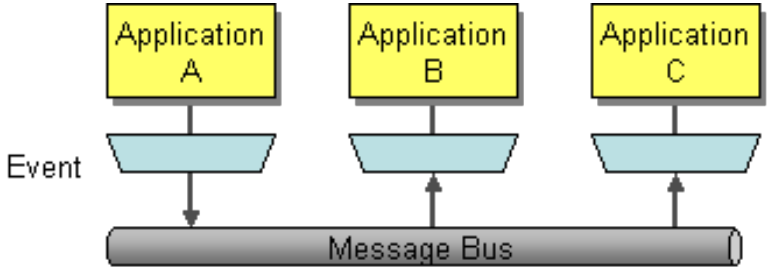
Shared Database



Remote Procedure Invocation



Message Queue



Different Decisions for Messaging Systems

- **Directing Messaging** from event producers and consumers or via a **message queue** (aka broker) ?
- What happens if the **producers send messages faster than the consumers can process them**?
  - Drop messages
  - Buffer messages in a queue
  - Backpressure
- What happen if the **queue does not fit anymore in RAM**? Save to disk?
- What happens if nodes **crash** or temporarily go offline—are any **messages lost**?
- Should the **messages be kept after delivery** or deleted?
- Is the **order** of the messages preserved or not?
- Is the messaging system **distributed**?
- Are we using a **publish/subscribe** mechanism?

Source: Kleppmann (2017), p. 389

Transmitting Event Streams (Message Queue / Broker)

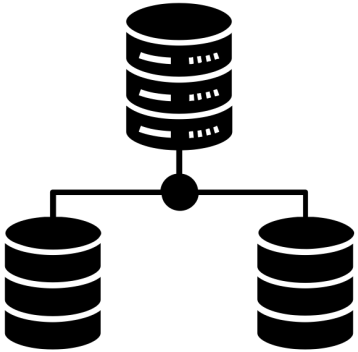
1. **Advanced Message Queuing Protocol (AMQP) based style**
   - Assigns individual messages to consumers, and consumers **acknowledge** individual messages when they have been successfully processed
   - **Messages are deleted from the queue** once they have been acknowledged
   - The exact **order of message processing is not important**
   - No need to go back and read old messages again after they have been processed
2. **Log-based message queue / broker**
   - Assigns all messages in a partition to the same consumer node, and always delivers messages in the **same order**
   - Parallelism is achieved through **partitioning**, and consumers track their progress by checkpointing the offset of the last message they have processed.
   - There is no ordering guarantee across different partitions.
   - The **queue retains messages on disk**, so it is possible to jump back and reread old messages if necessary. Similarities to the replication logs found in databases.

Source: Kleppmann (2017), p. 389

Difference to Databases

Message Queues / Broker	Database
AMQP-style brokers automatically delete a message when it has been successfully delivered to its consumers. Log-based Message Brokers do not delete the messages automatically	Keep data until it is explicitly deleted
AMQP-style brokers assume that their working set is fairly small—i.e., the queues are short. Log-based Message Brokers can archive or delete old messages.	No assumption of database size
Support some way of subscribing to a subset of topics matching some pattern	Support secondary indexes. Various ways of searching for data.
Do not support arbitrary queries.	Complex queries based on a point-in-time snapshot of the data.
Notify clients when data changes (i.e., when new messages become available).	If another client adds something to the database, the first client does not find out (unless it repeats the query, or polls for changes)

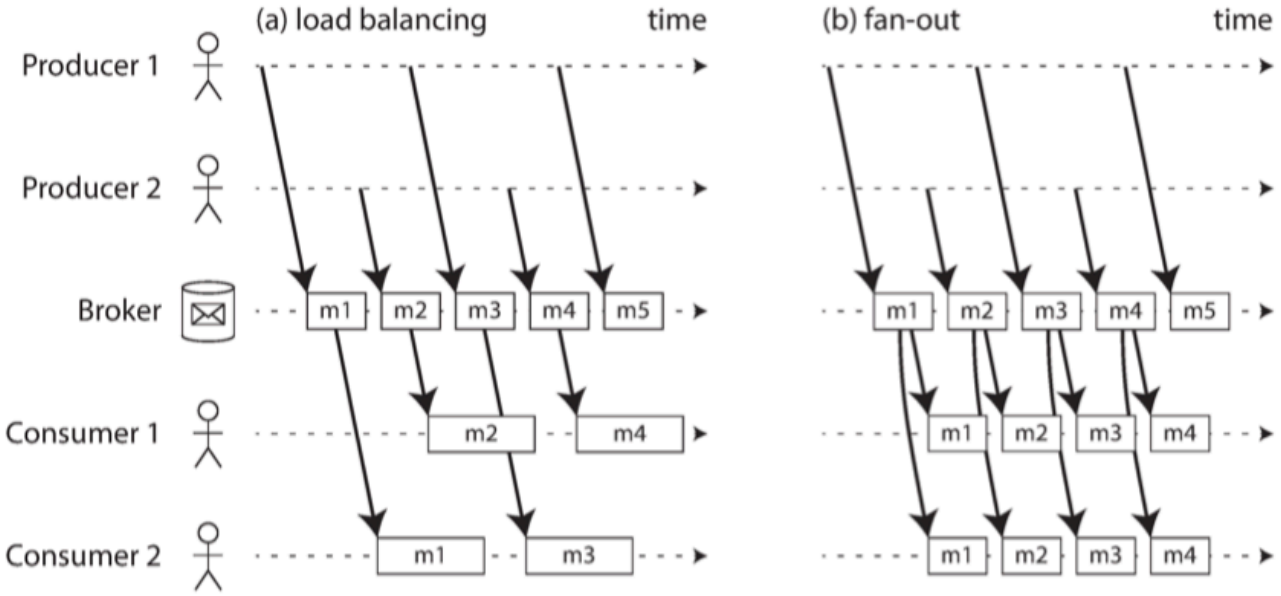
Source: Kleppmann (2017), p. 444



Distributed Systems Concerns

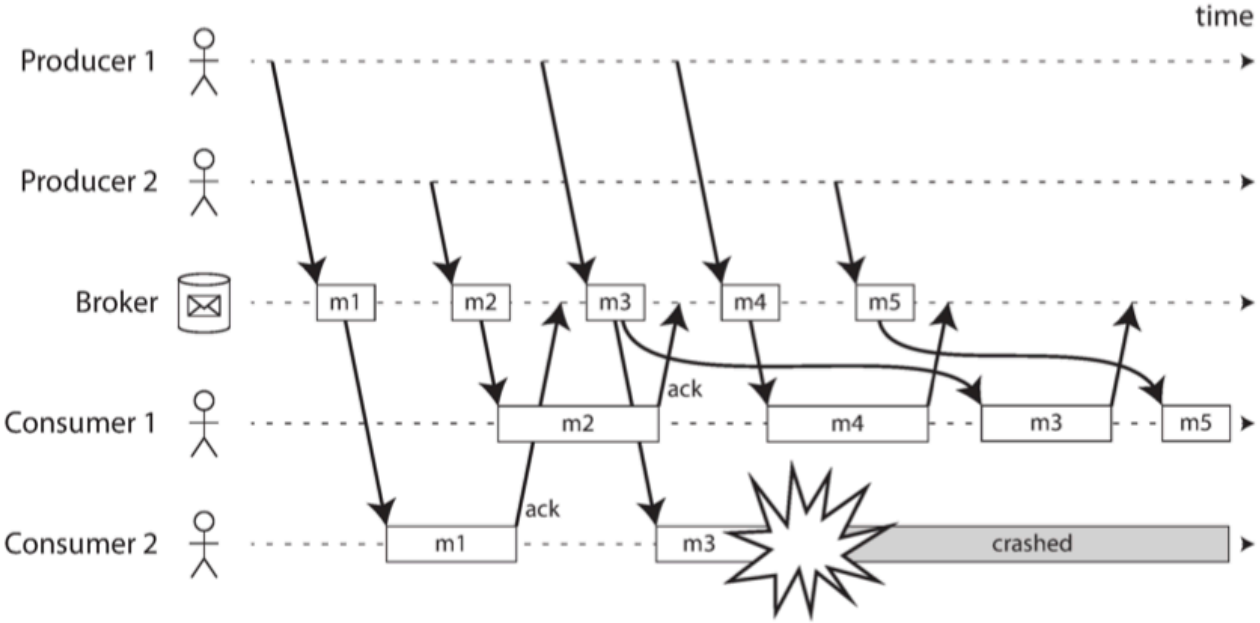
Multiple Consumers

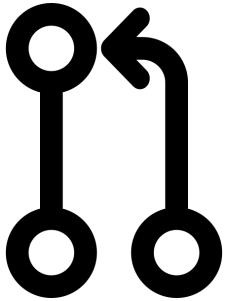
- Load balancing (send to one consumer)
- vs. Fan-out (send to all consumers)



Effect of Load Balancing and Redelivery on message order

- Consumer 2 crashes while processing m3, so it is redelivered to consumer 1 at a later time

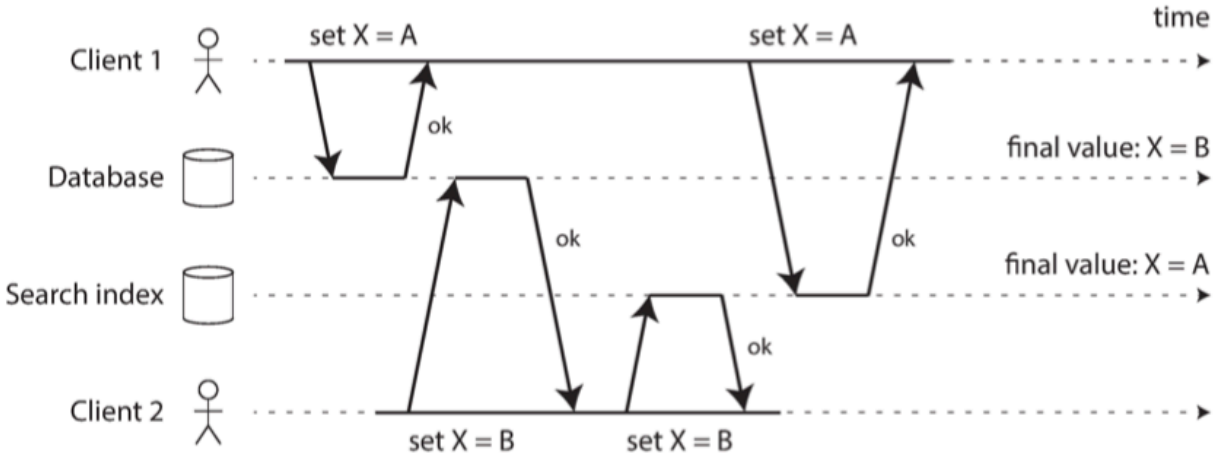




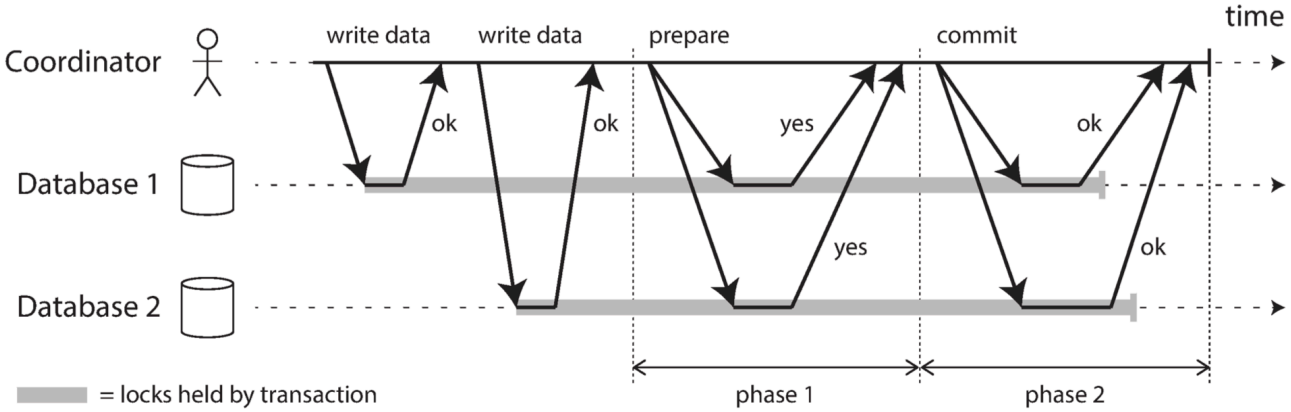
Two-Phase Commit

Problems of dual writes for keeping systems in sync

- In the database,
  - X is first set to A and then to B,
- while at the search index
  - the writes arrive in the opposite order.



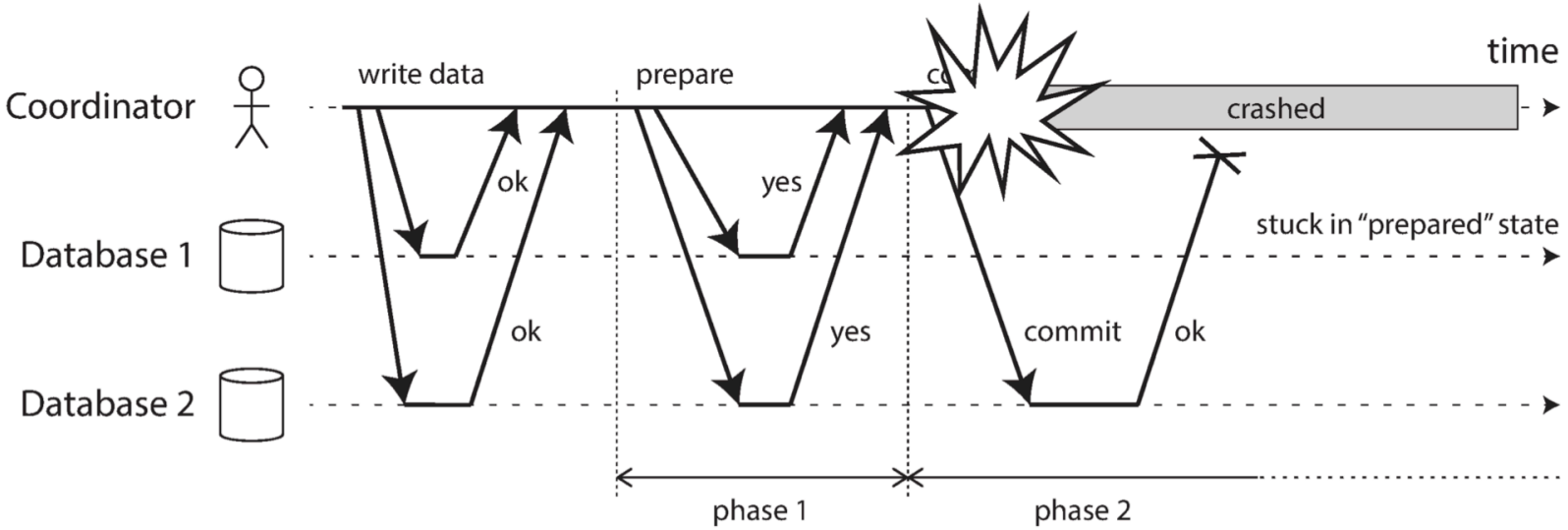
Two-Phase-Commit (2PC)



2PC Transaction:

1. **Write Phase**: Coordinator writes / read from participating databases
2. **Prepare Phase**: Coordinator send “prepare” request
3. **Commit Phase**:
   - If all participating database reply “yes:
     - Coordinator sends “commit” to all participating databases
   - If any participating database reply “no”:
     - Coordinator send “abort” to all participating databases

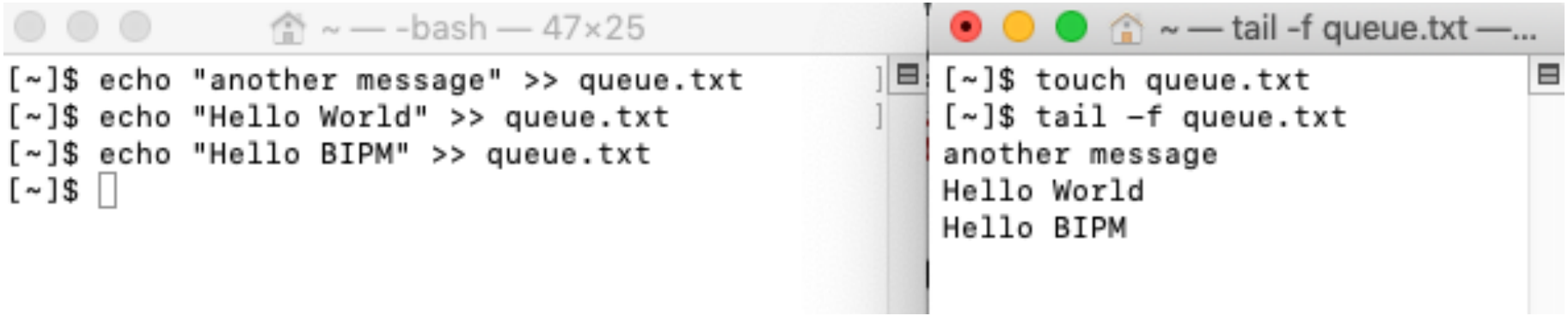
Two-Phase-Commit (2PC)



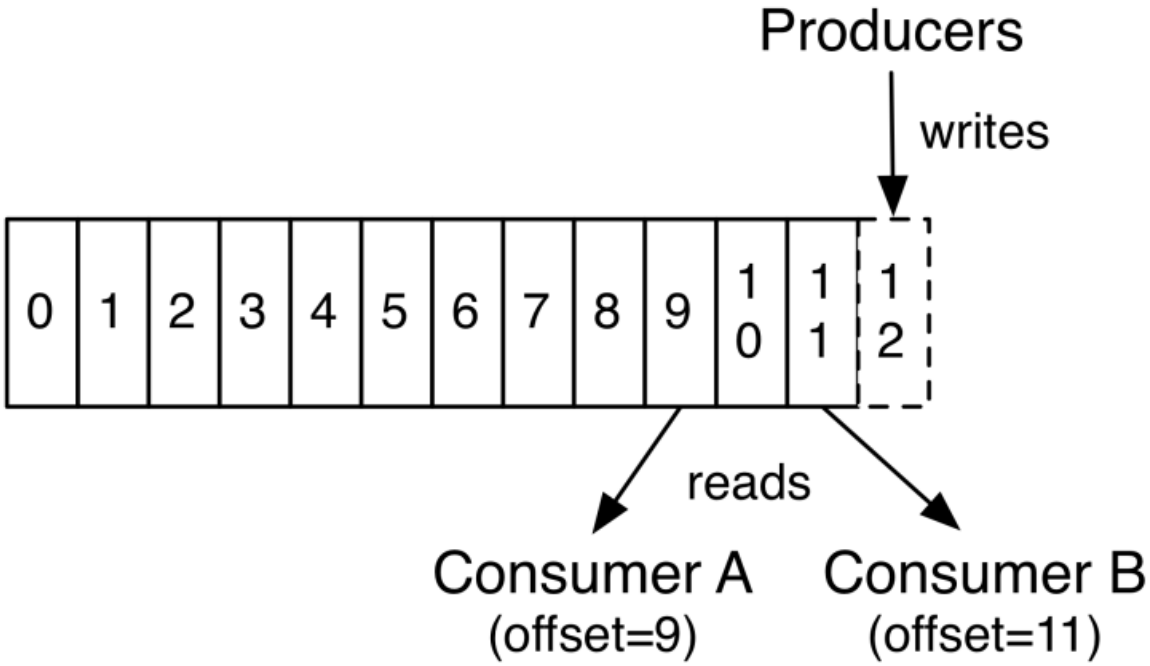
- The coordinator crashes after participants vote "yes."
- Database 1 does not know whether to commit or abort.

Queues

Simple append only log-based queue

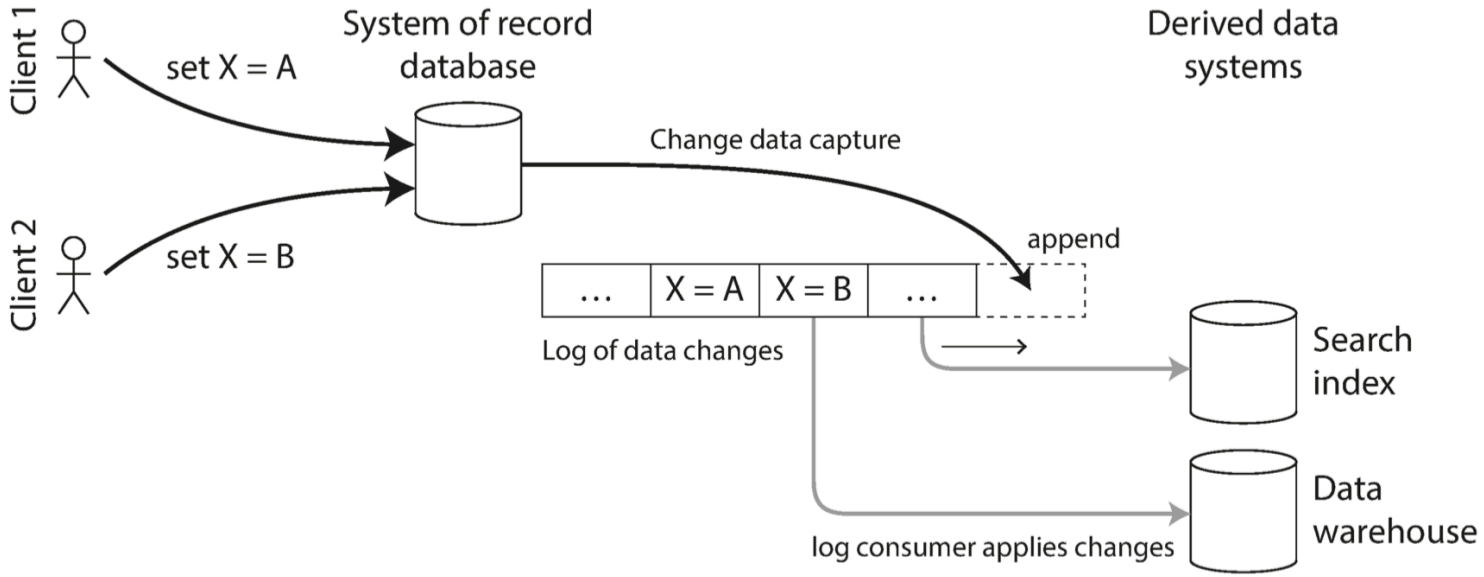


Log-structured Message Queue



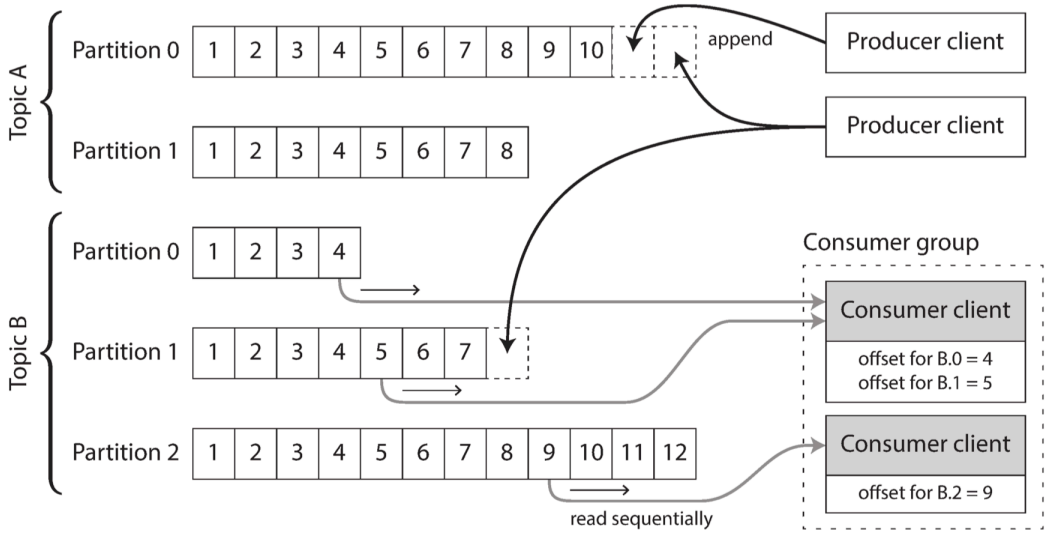
Use of Data Queues for Service Integration

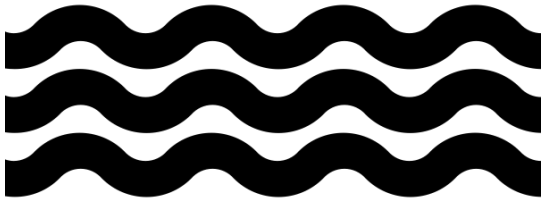
- E.g. Change Data Capturing



Partitioned logs

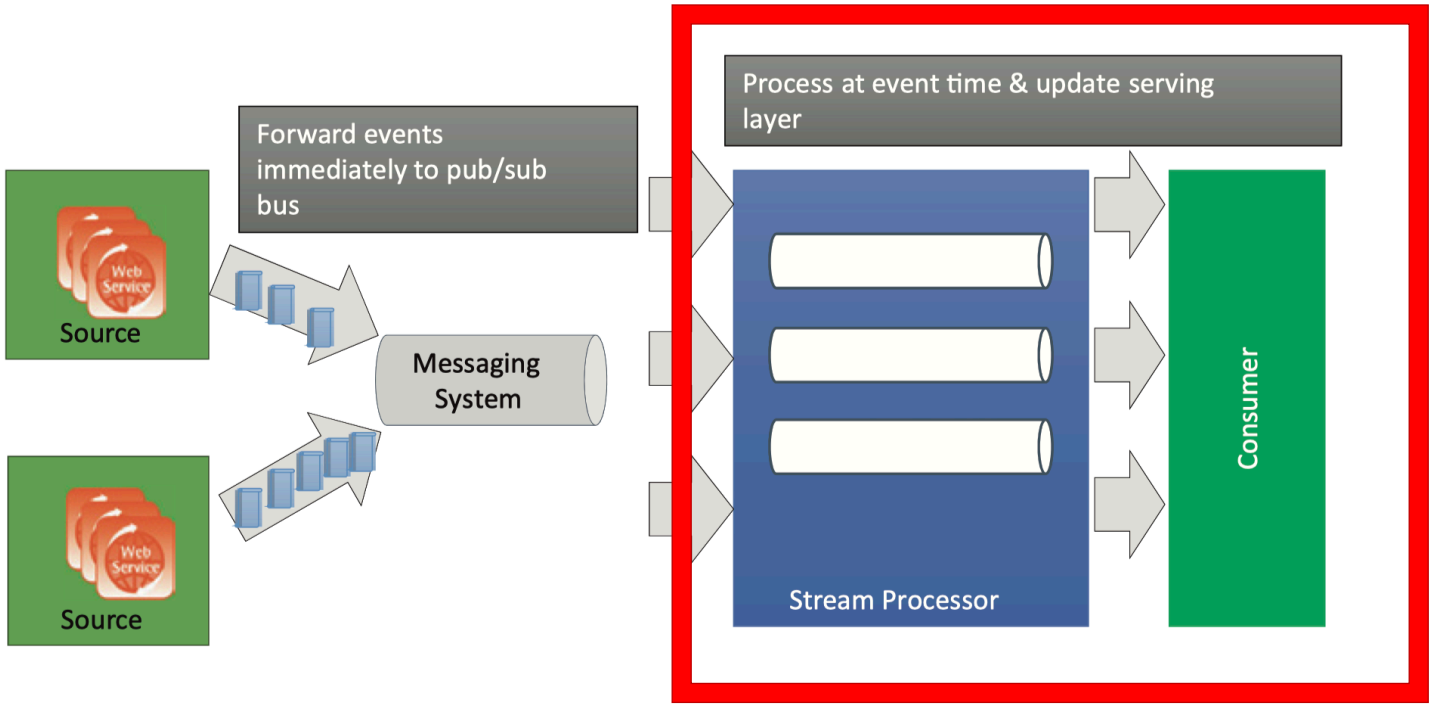
- Producers send **messages**
- by **appending** them to a **topic-partition** file,
- and consumers **read** these files **sequentially**.



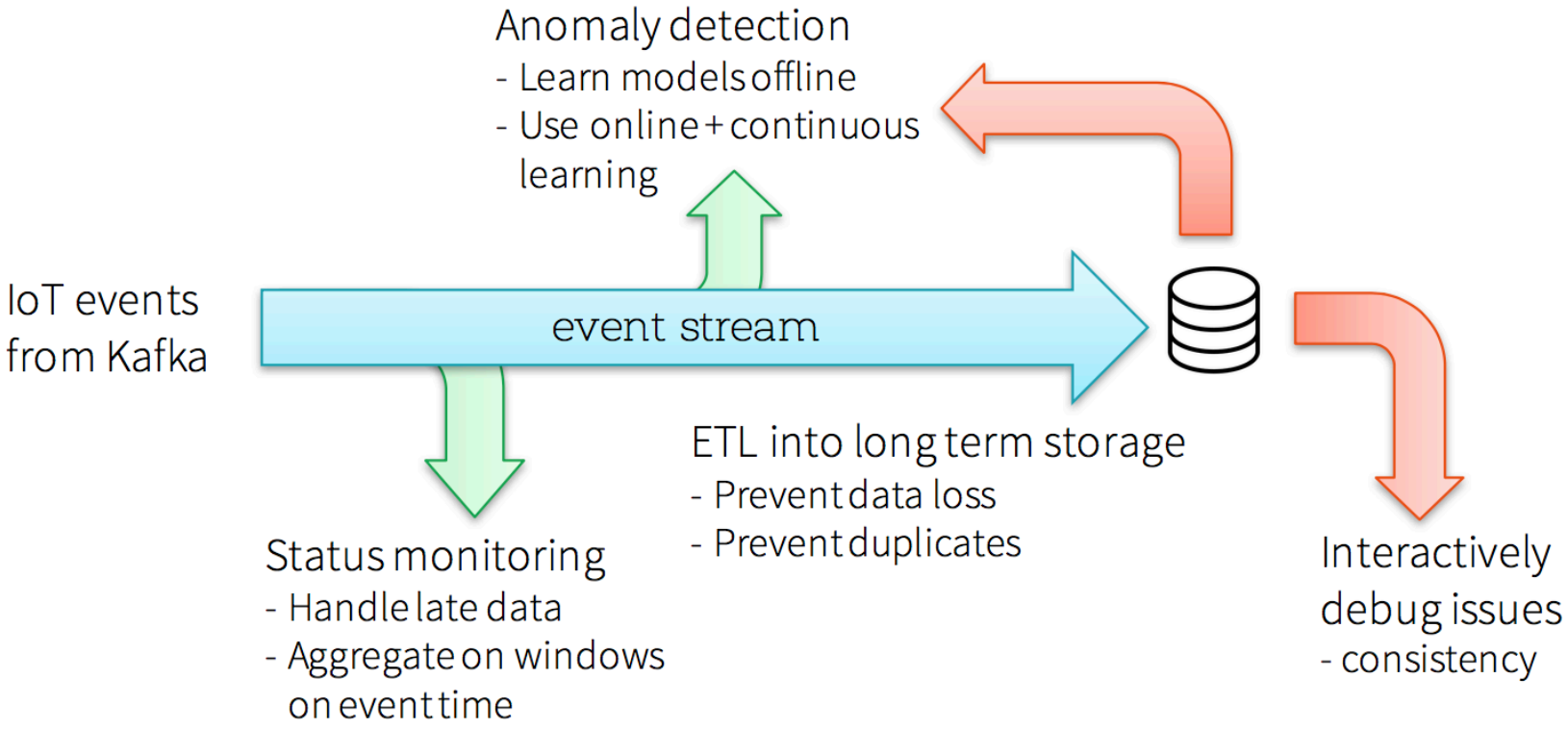


Processing Streams

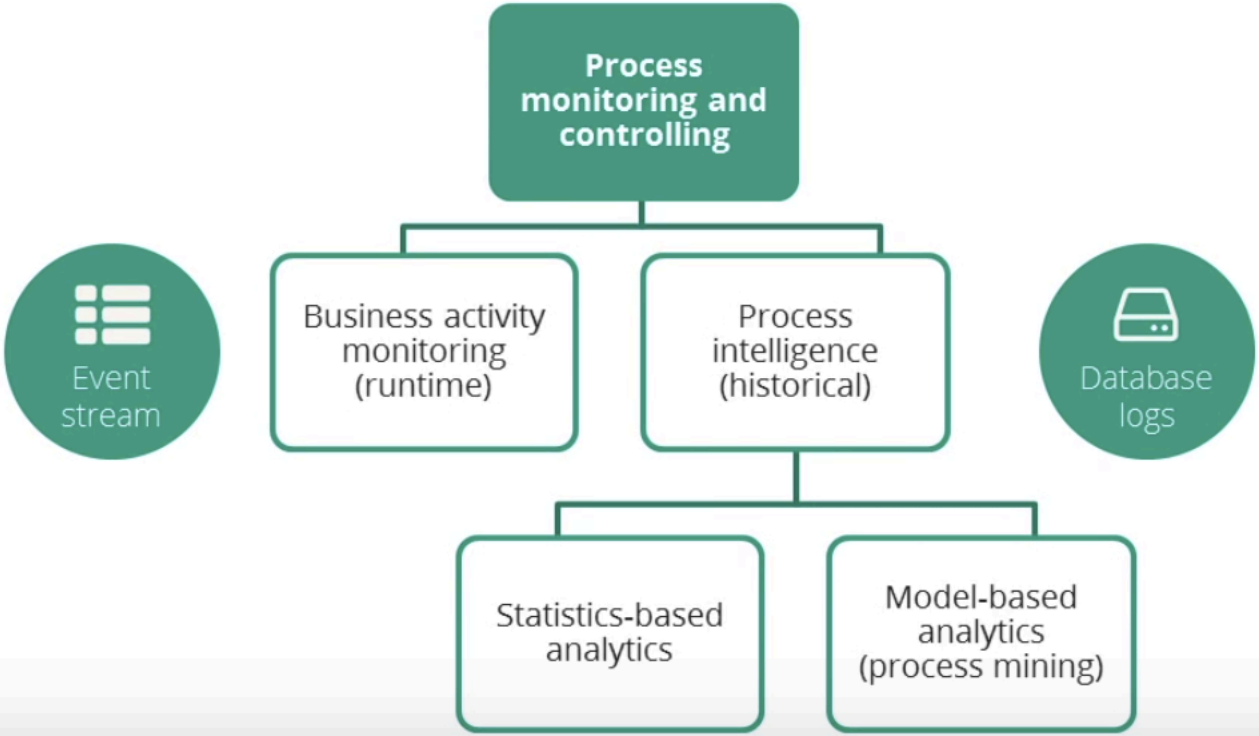
Stream Analytics Architecture



Use Case: IoT Device Monitoring

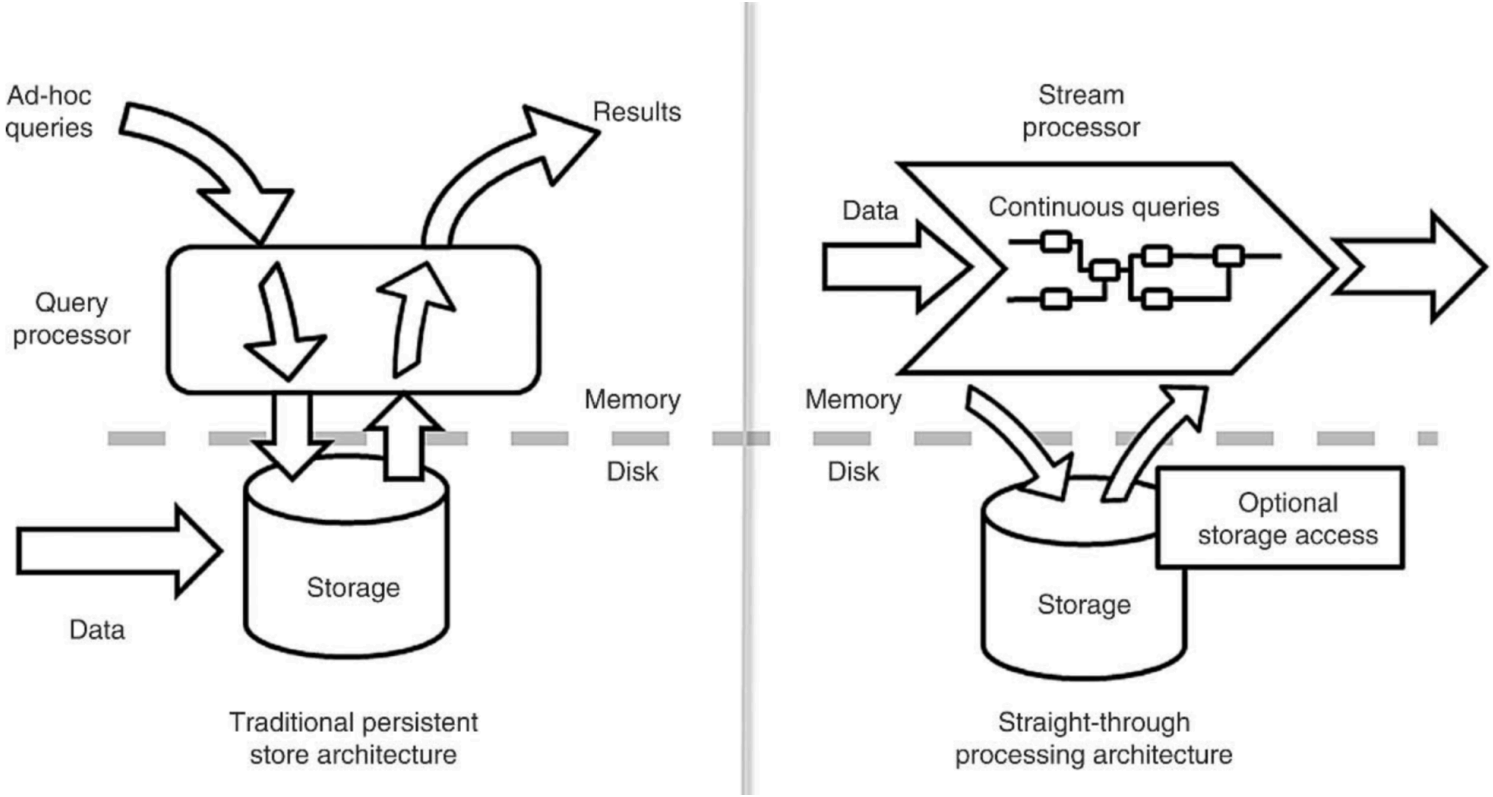


Use Case:
Process Monitoring and Controlling



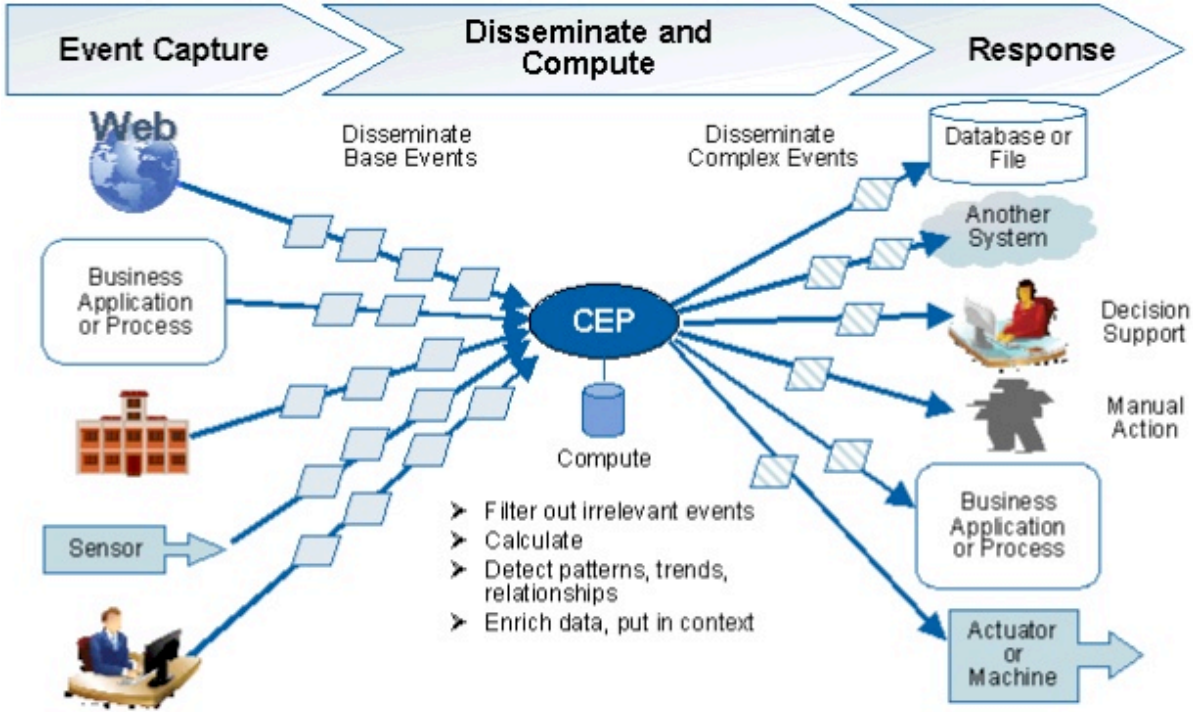
Source: La Rosa (2015)

Stream Processing



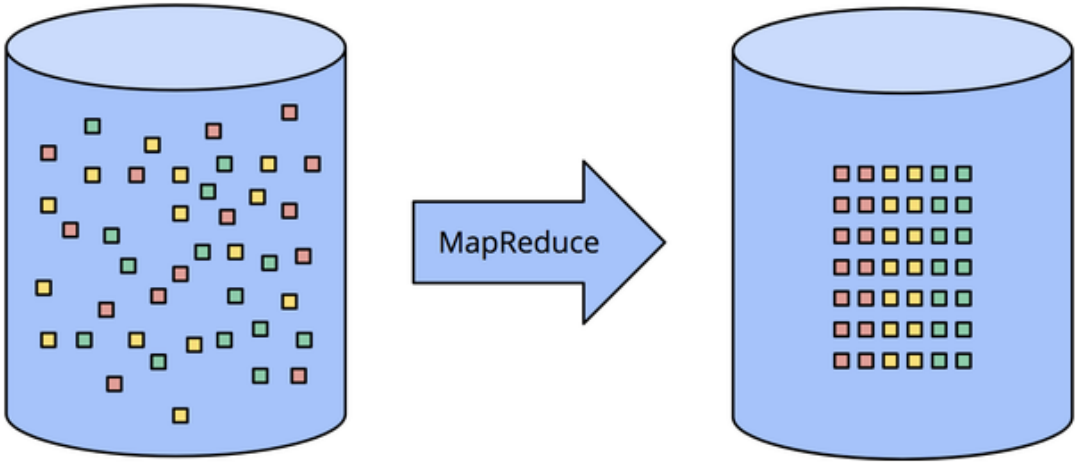
Complex Event Processing (CEP)

- “continuous query”



Source: SAP

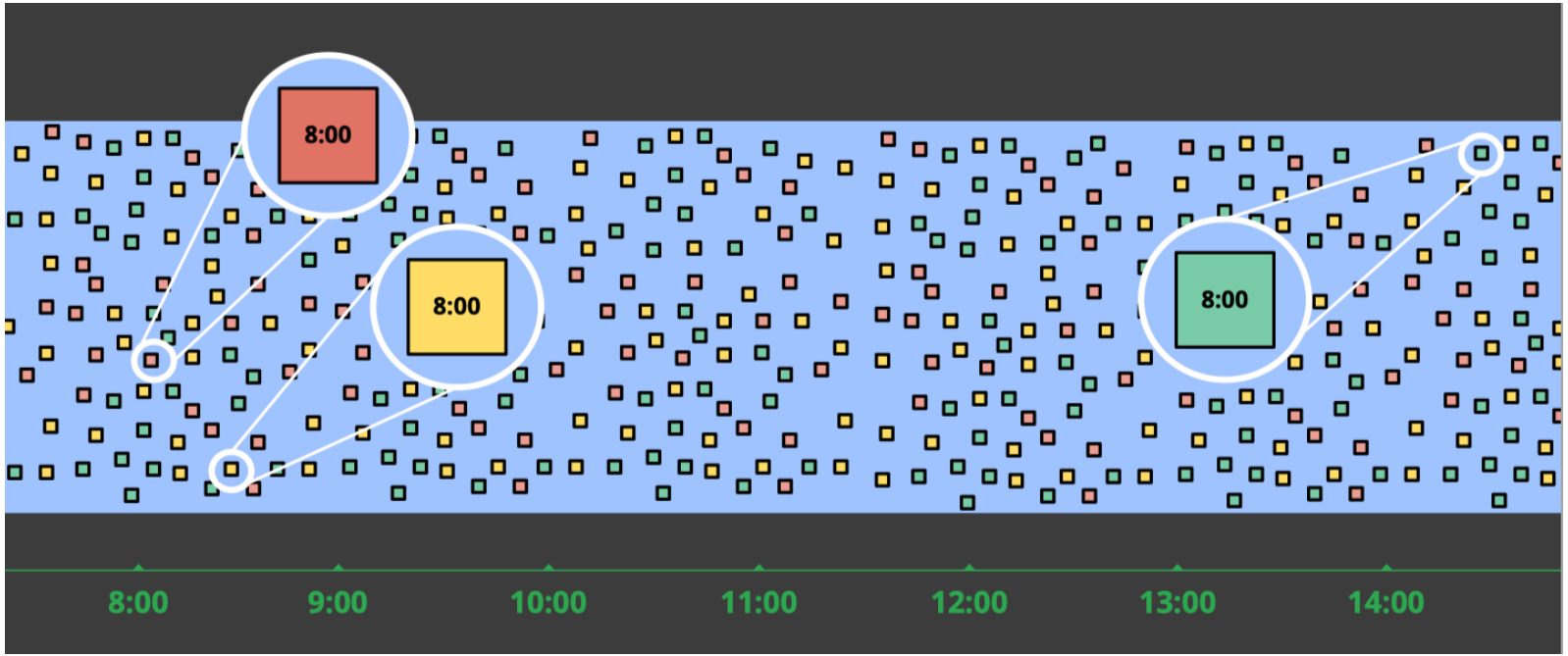
Bounded data processing



- classic batch engine
- finite pool of unstructured data on the left is run through a data processing engine, resulting in corresponding structured data on the right.

Source: Akidau, Chernyak & Lax (2018)

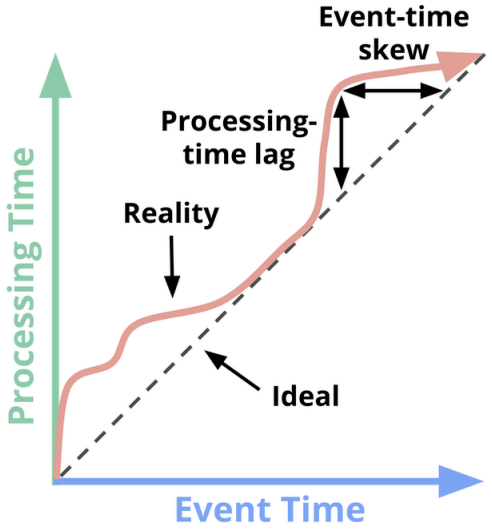
Valid Time (Event-Time) is not Transaction Time (Processing Time)



Source: https://www.infoq.com/presentations/stream-apache-beam

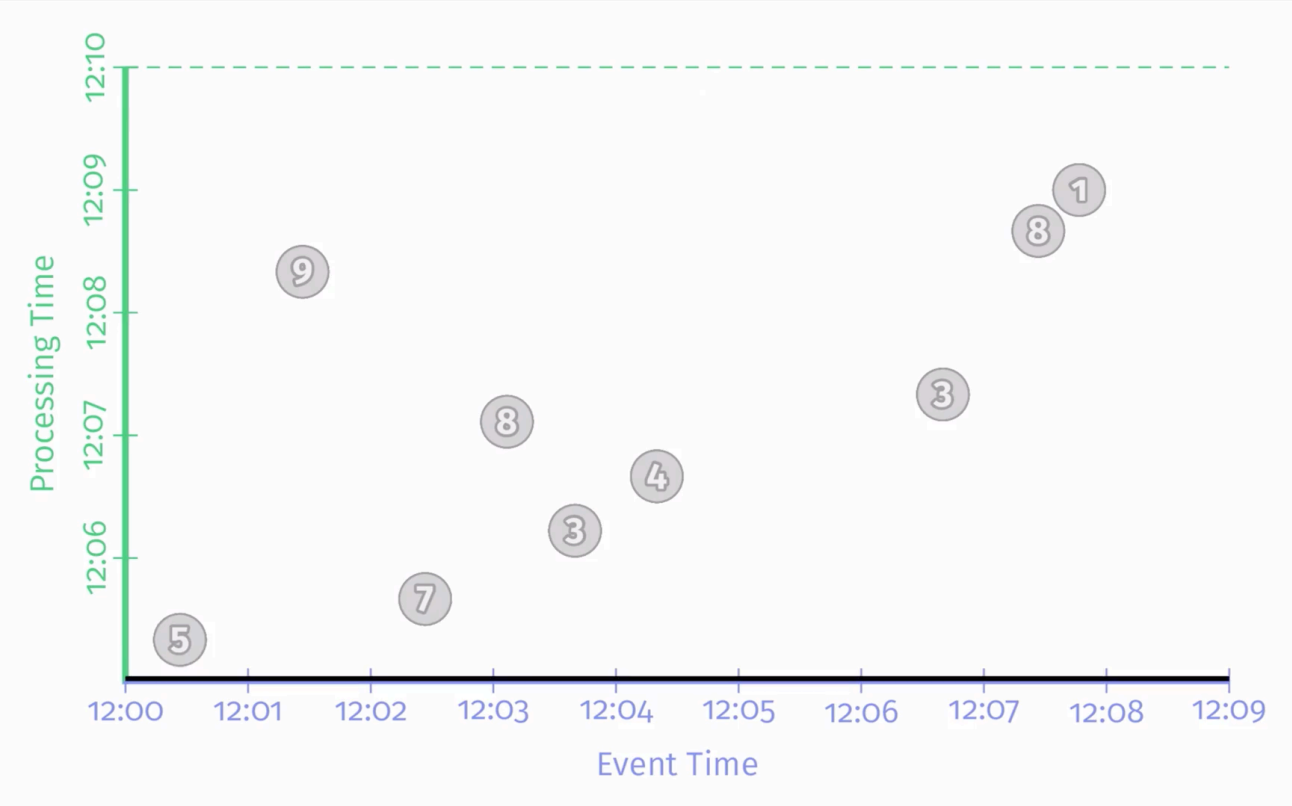
Time-domain mapping

- Event time: The time at which events actually occurred.
- Processing time: The time at which events are observed in the system.



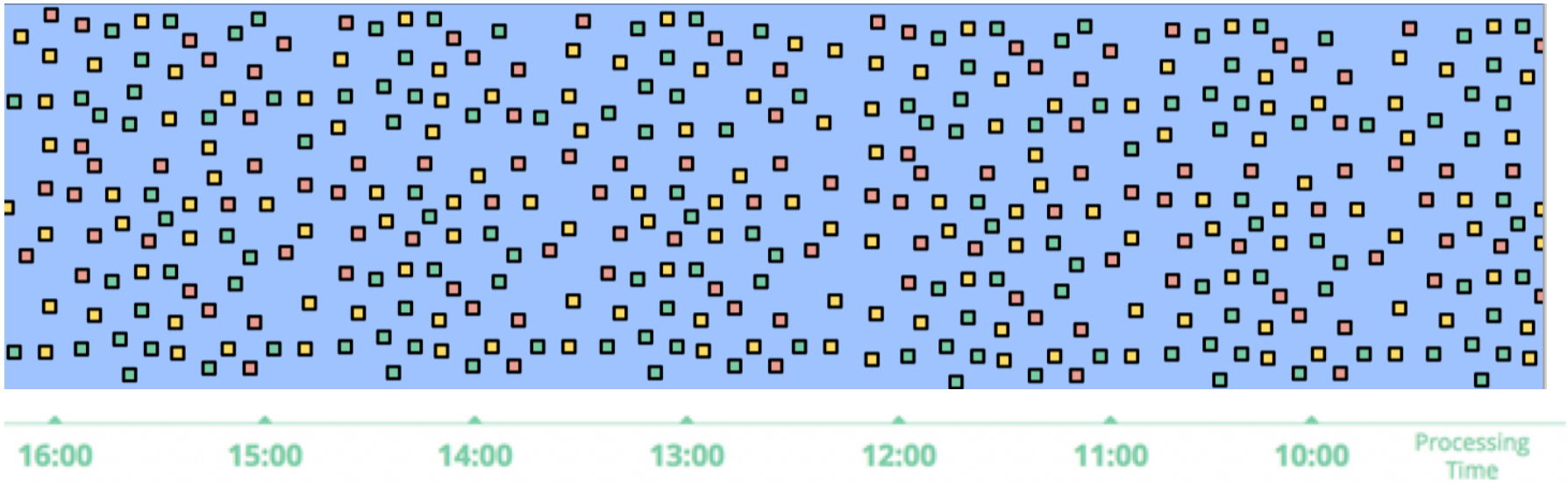
Source: Akidau, Chernyak & Lax (2018)

Bounded data processing

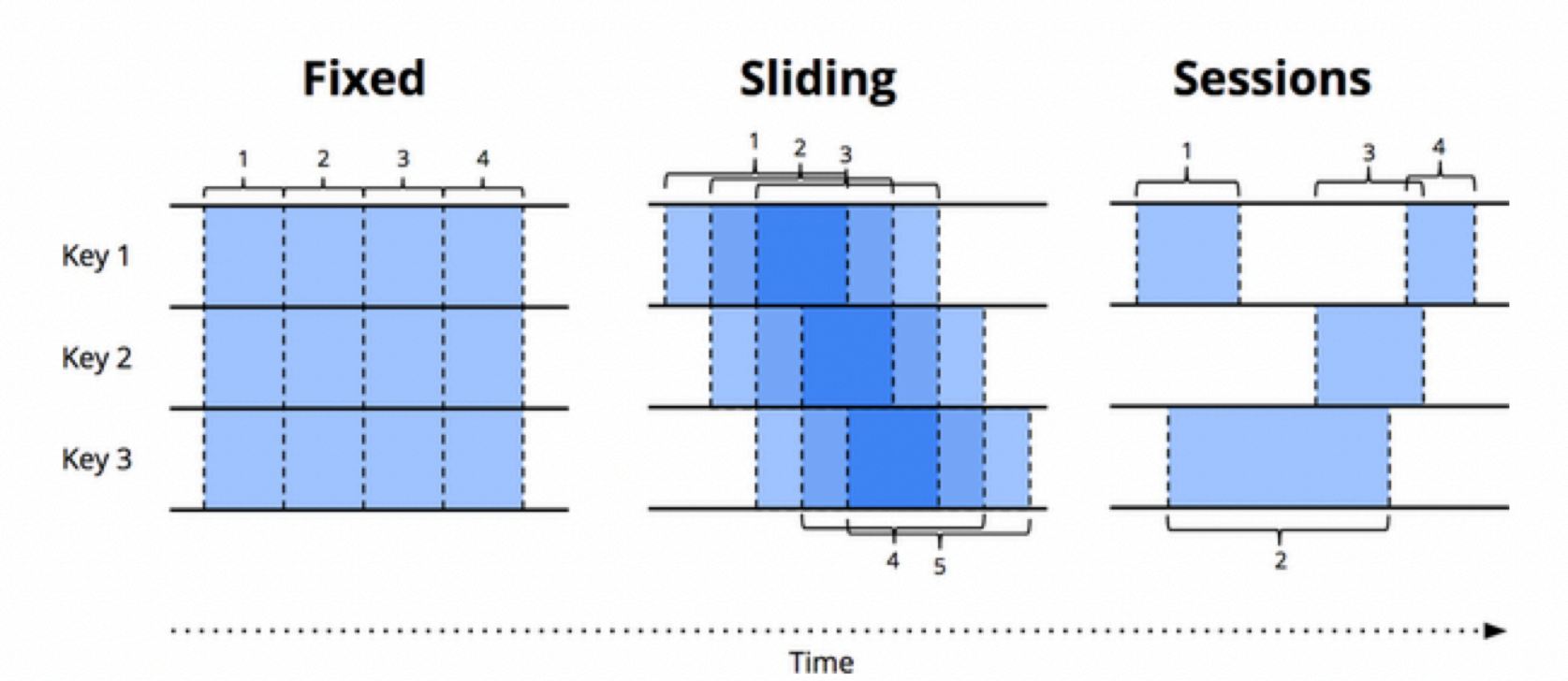


Source: Akidau, Chernyak & Lax (2018)

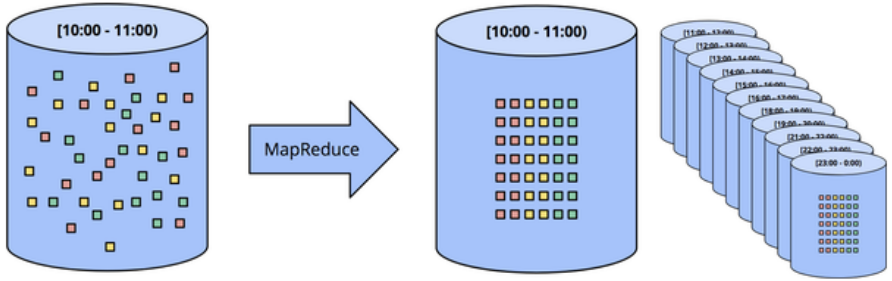
Unbounded Data



Windowing divides data into event-time-based finite chunks

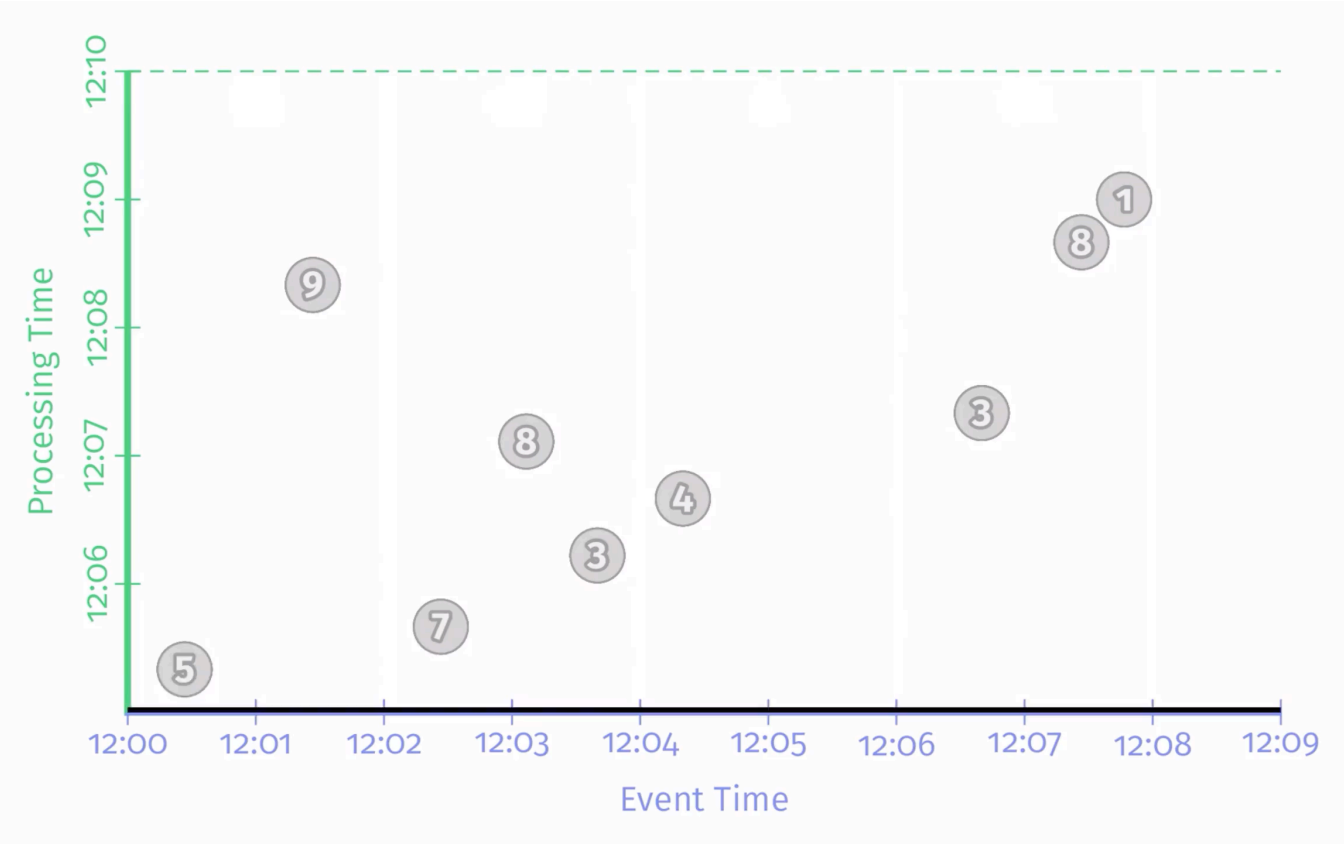


Batch Processing with Fixed Windows

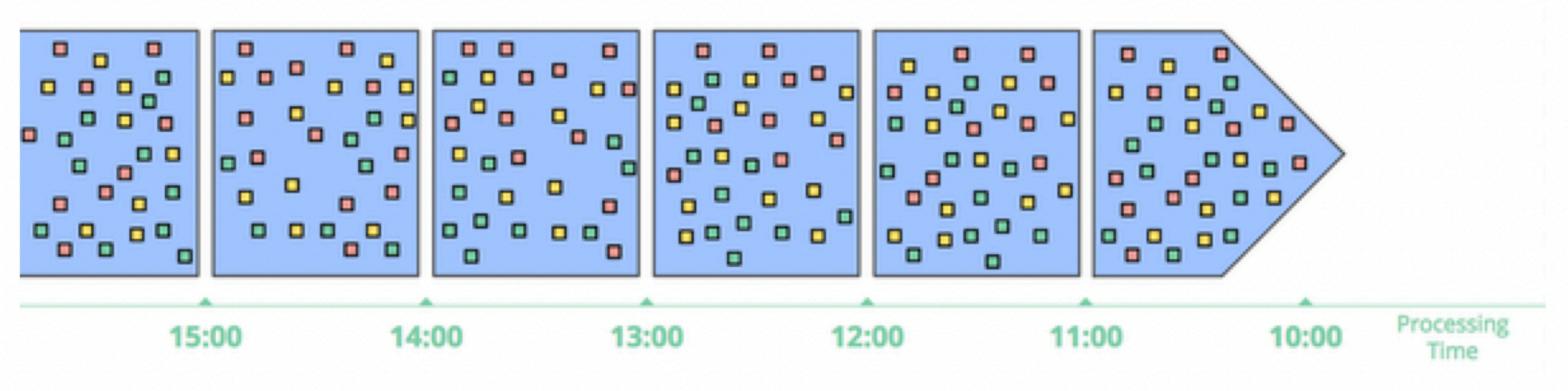


- Unbounded data processing via ad hoc fixed windows with a classic batch engine.
- An unbounded dataset is collected up front into
  - finite, fixed-size windows of bounded data that are
  - then processed via successive runs a of classic batch engine

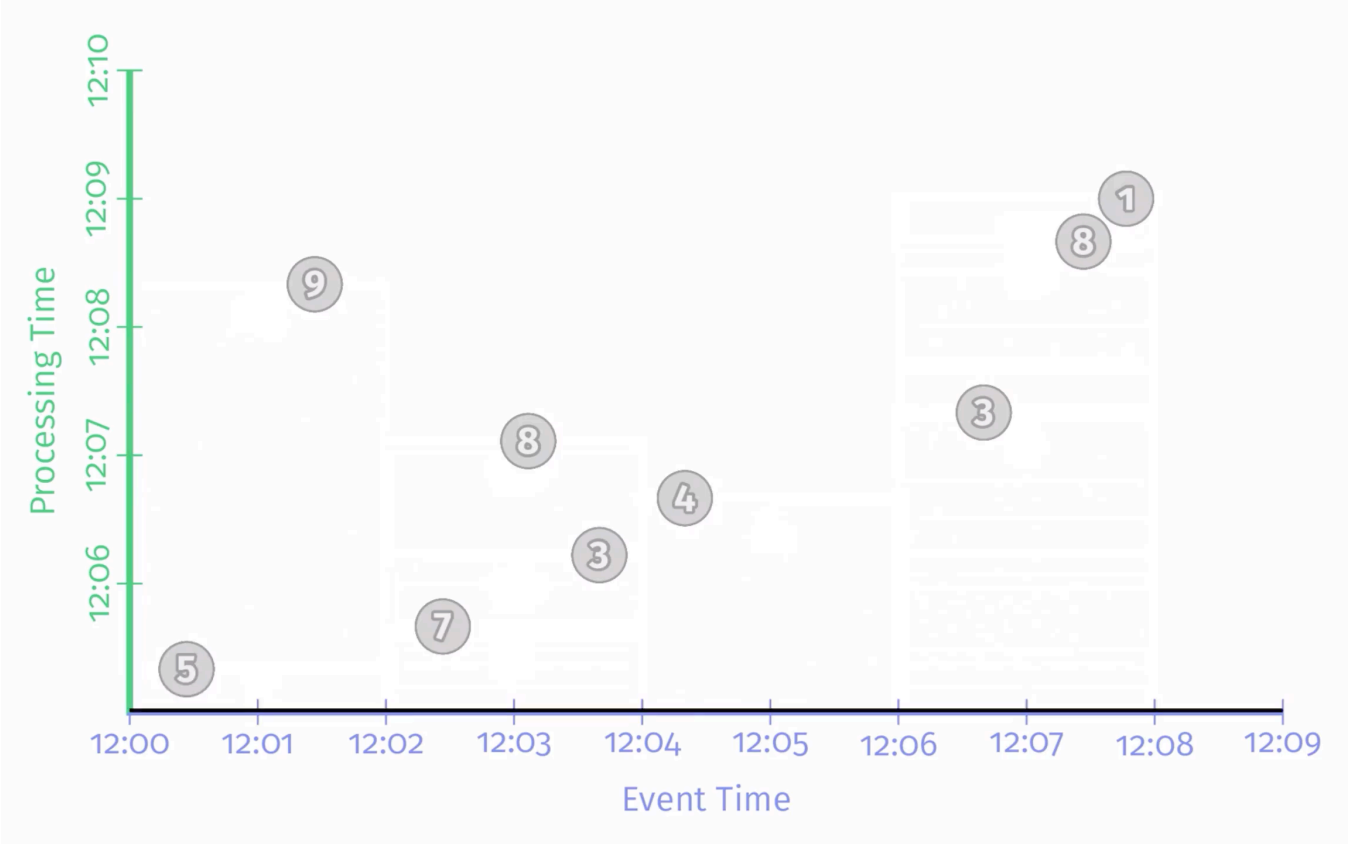
Batch Processing with Fixed Windows



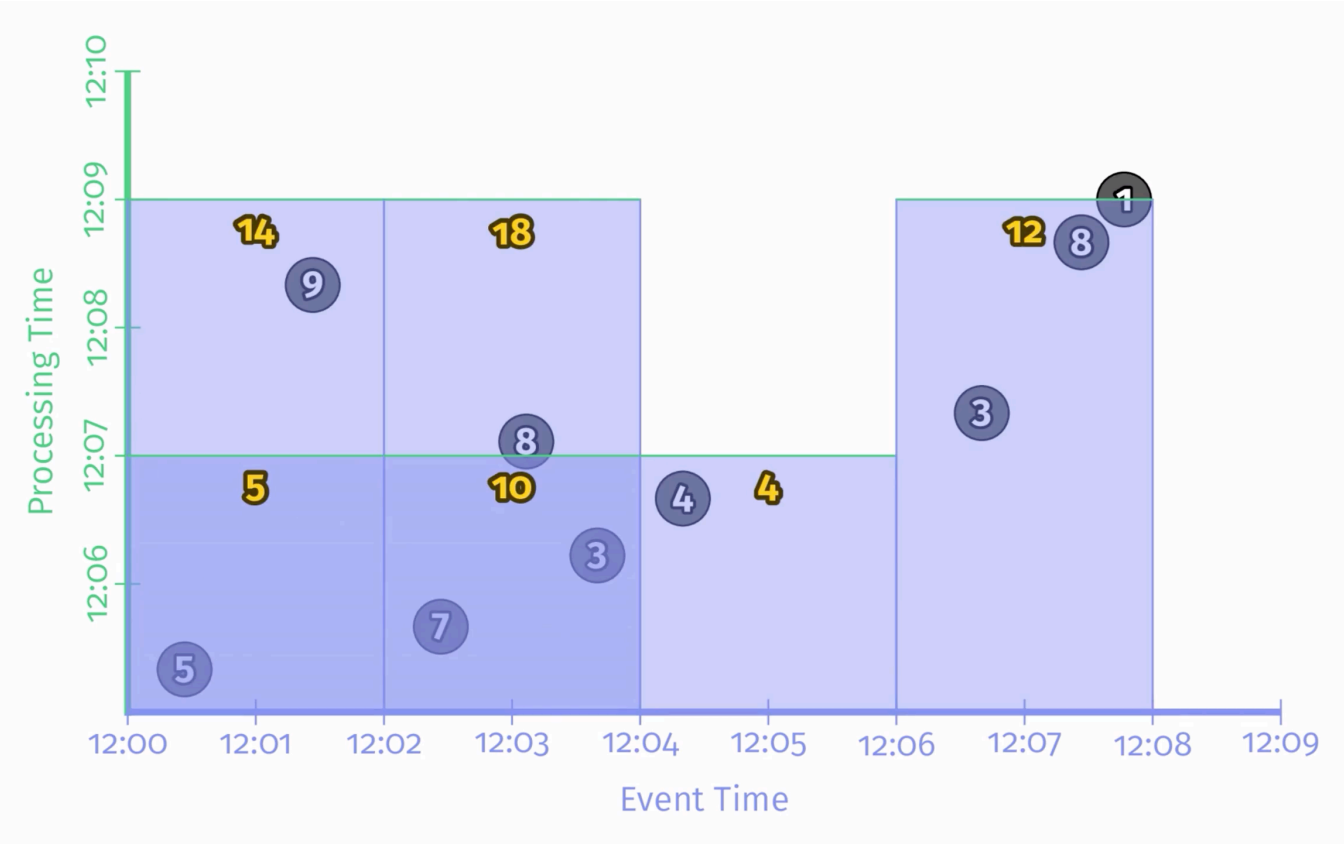
Aggregating via Fixed Time Windows



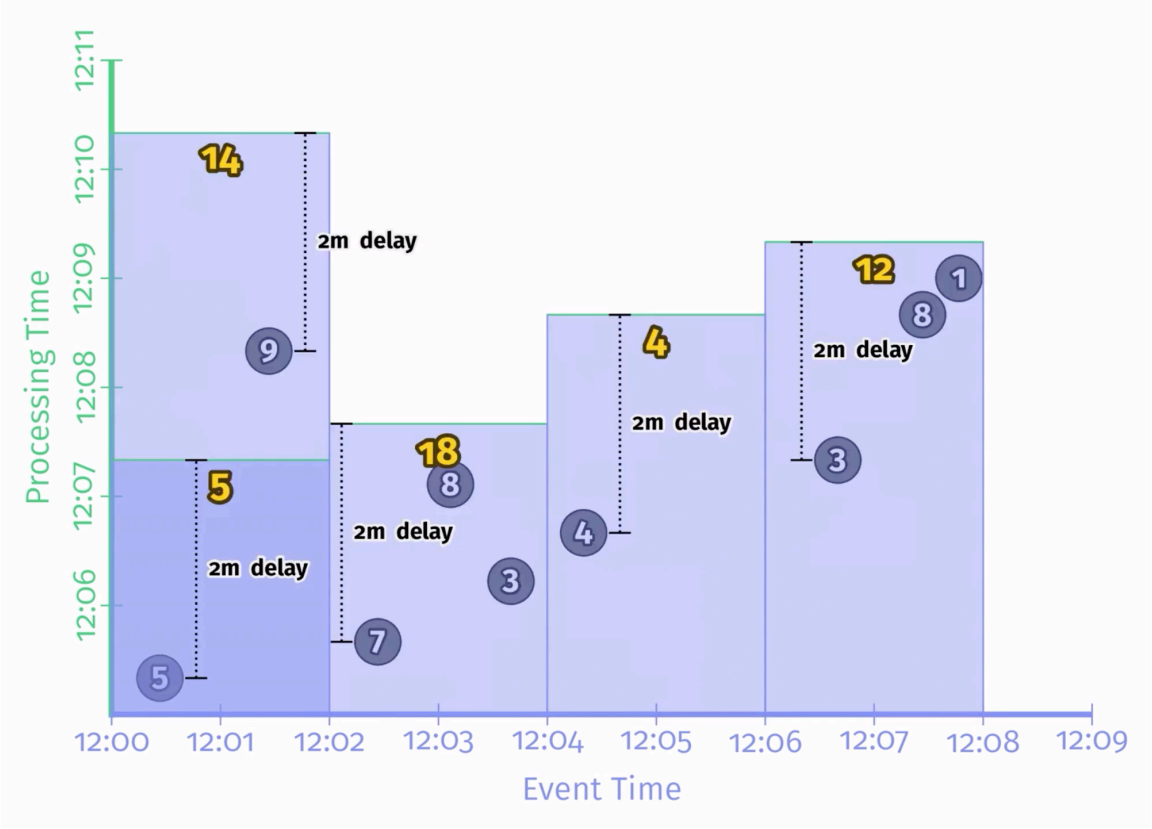
Per-record triggering on a streaming engine



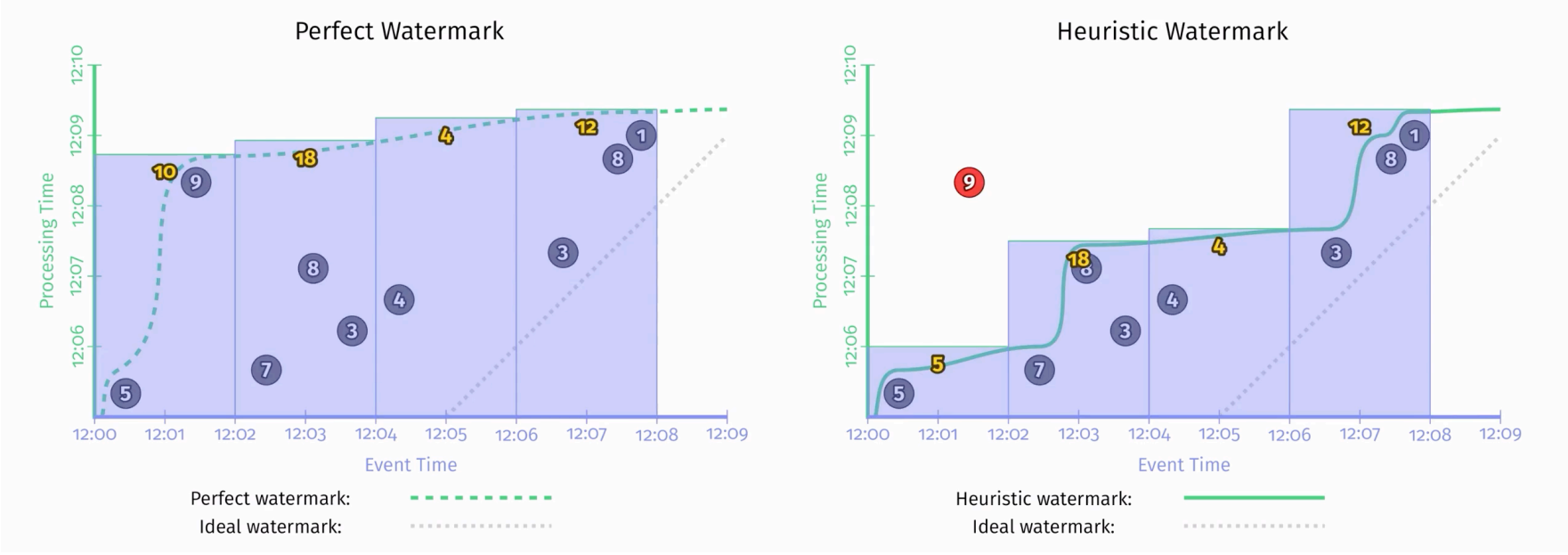
Two-minute aligned delay triggers (microbatching) on a streaming engine



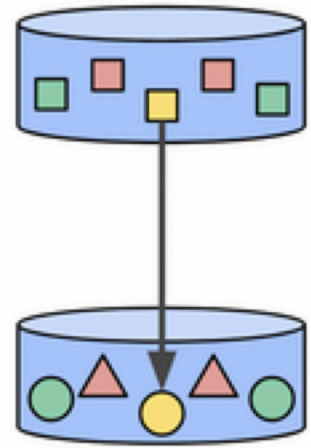
Two-minute unaligned delay triggers on a streaming engine



Tradeoff between Waiting for Late Events and Discarding Late Events

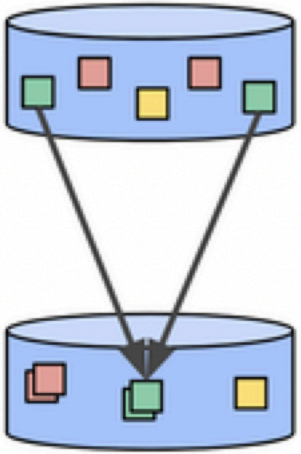


Types of Computation



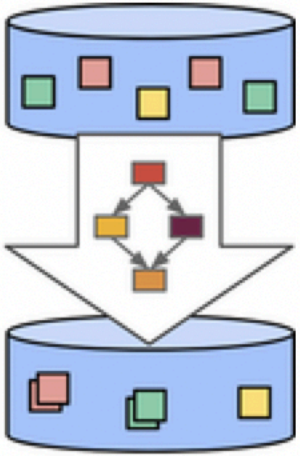
Element-wise

Similar to Map



Grouping

Similar to Reduce

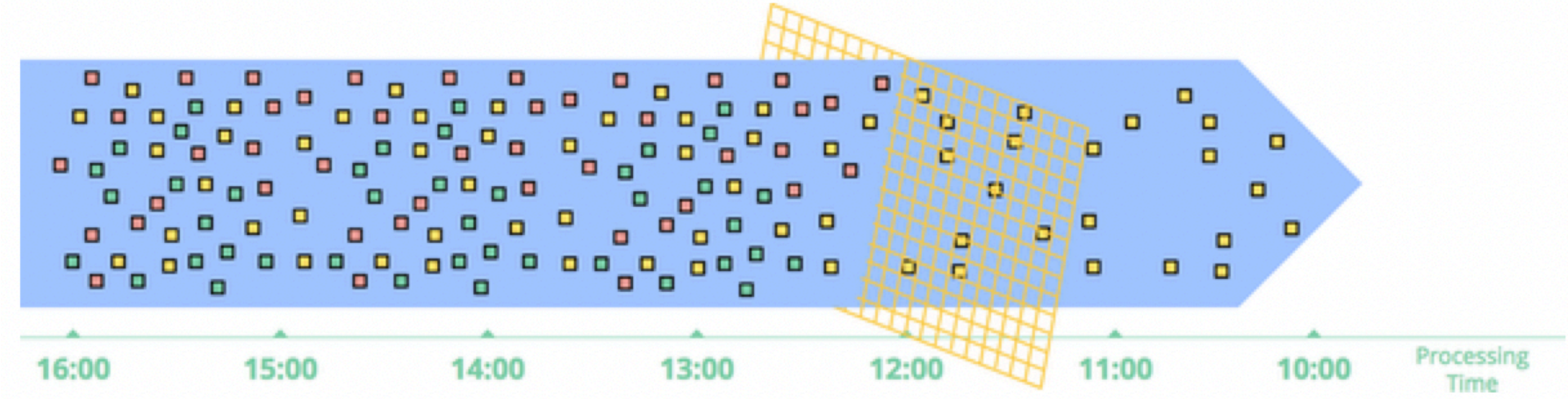


Composite

Combines Streams
Similar to Shuffle/Sort

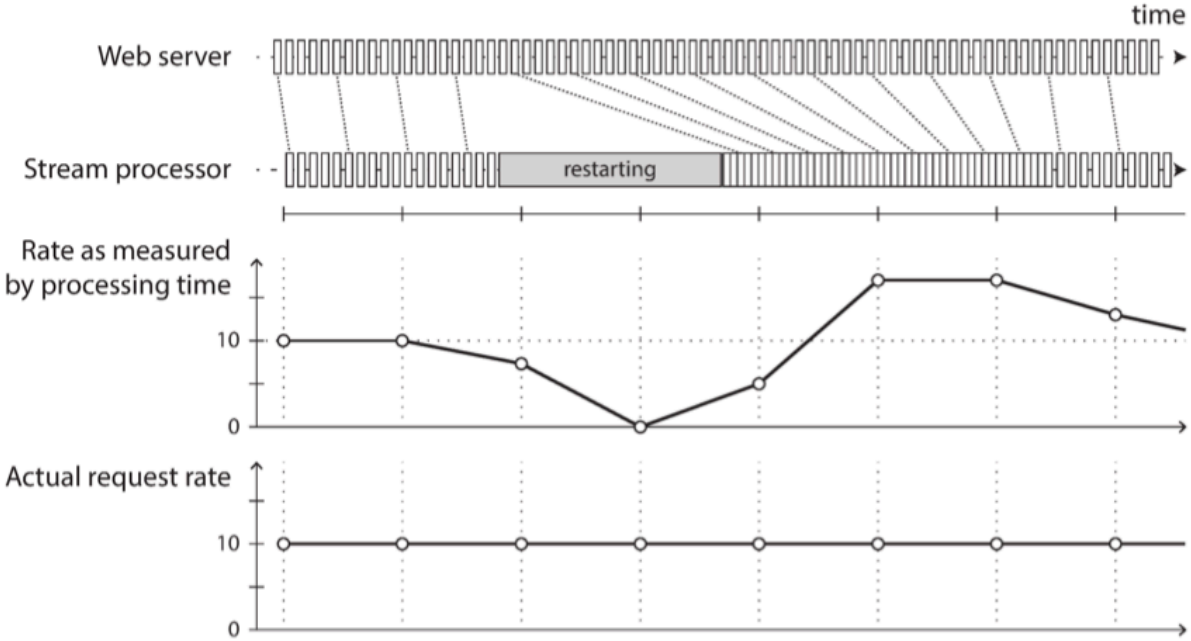
Source: Akidau, Chernyak & Lax (2018)

Element-wise Transformation



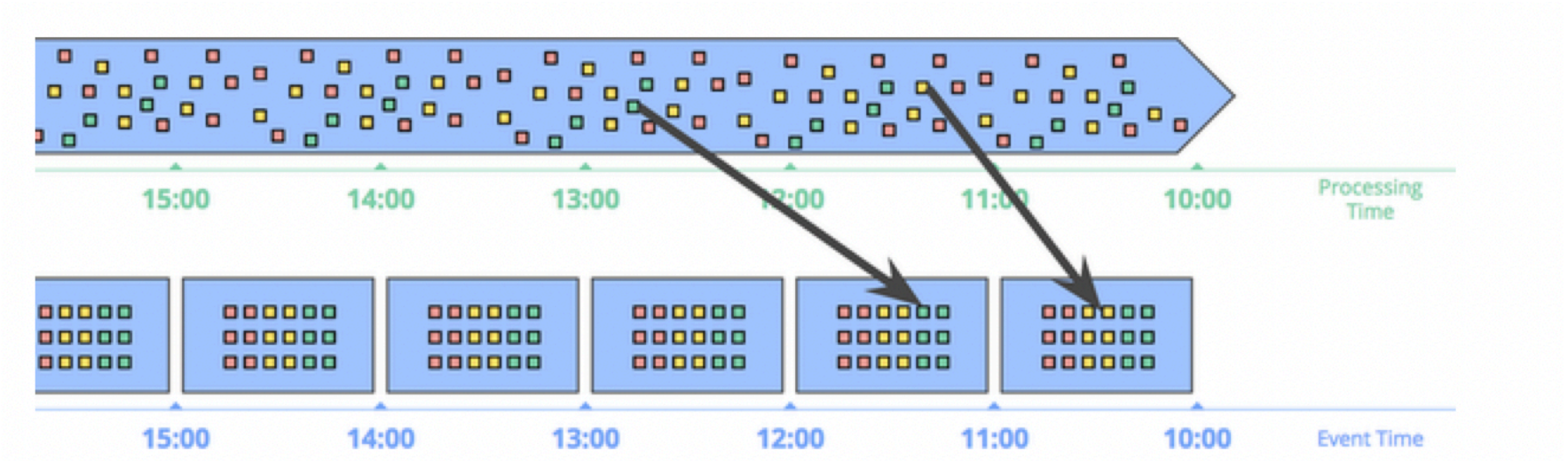
Source: Akidau, Chernyak & Lax (2018)

Aggregating via Processing-Time Windows

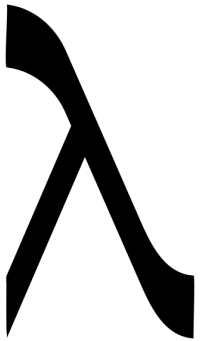


- Windowing by processing time introduces artifacts due to variations in processing rate.

Aggregating via Event-Time Windows

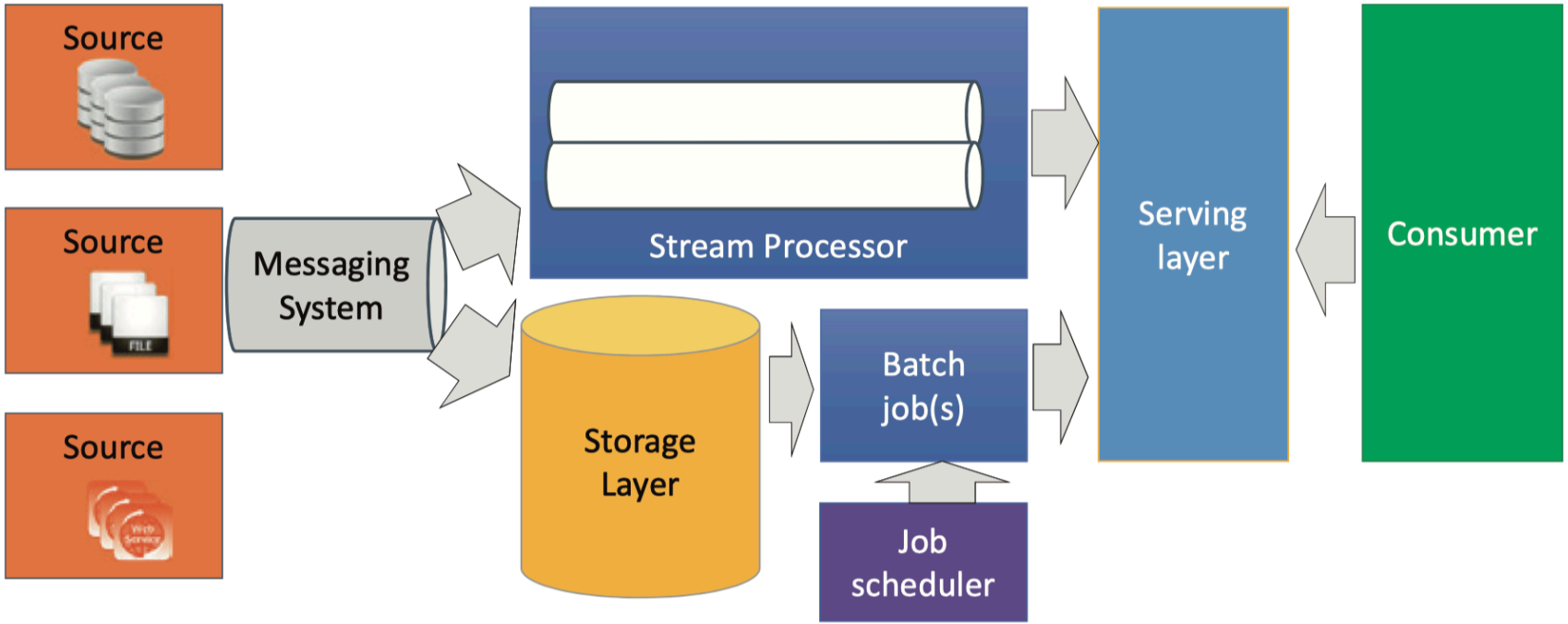


Source: Akidau, Chernyak & Lax (2018)



Lambda Architecture

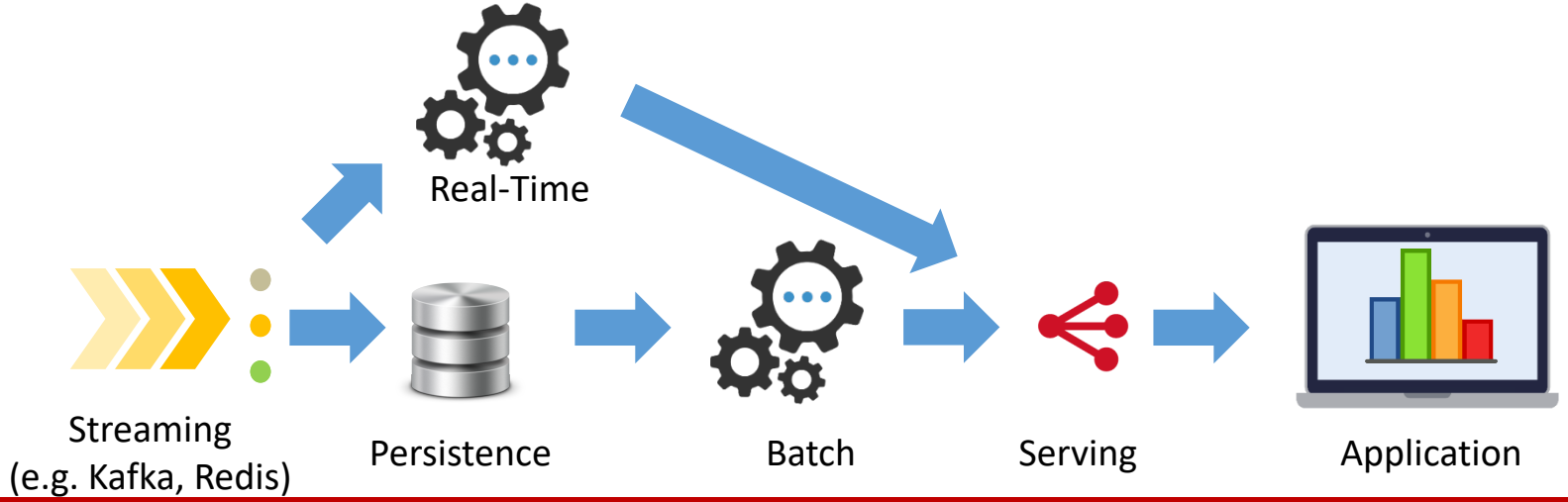
Lambda Architecture



Source: Freiknecht & Papp (2018, p.478), Marz (2011)

Lambda Architecture

- $Batch(D_{old}) + Stream(D_{\Delta now}) \approx Batch(D_{all})$
- Fast output (real-time)
- Data retention + reprocessing (batch)
  - „eventually accurate“ merged views of real-time and batch layer
  - Typical setups: Hadoop + Storm (→ Summingbird), Spark, Flink



Source: Gessert (2016), Marz (2011)

Lambda Architecture

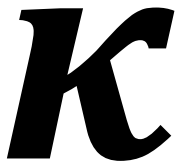
Advantages

- **Retention** of unchanged input
- **Fault-tolerance**
- **Code can evolve**: new code will produce new output on the entire data set. You don‘t have to work with old (broken) output

Disadvantage

- **High Complexity**
  - Business logic in 2 separate systems
  - Twice the code
  - Synchronizing code
  - Possibly two deployments
    - not necessarily the case for systems that provide batch and stream processing (Spark, Flink, Apex)
- No specialized tools
- No good abstractions
  - debugging, performance tuning etc. still require intimate knowledge of the underlying implementations

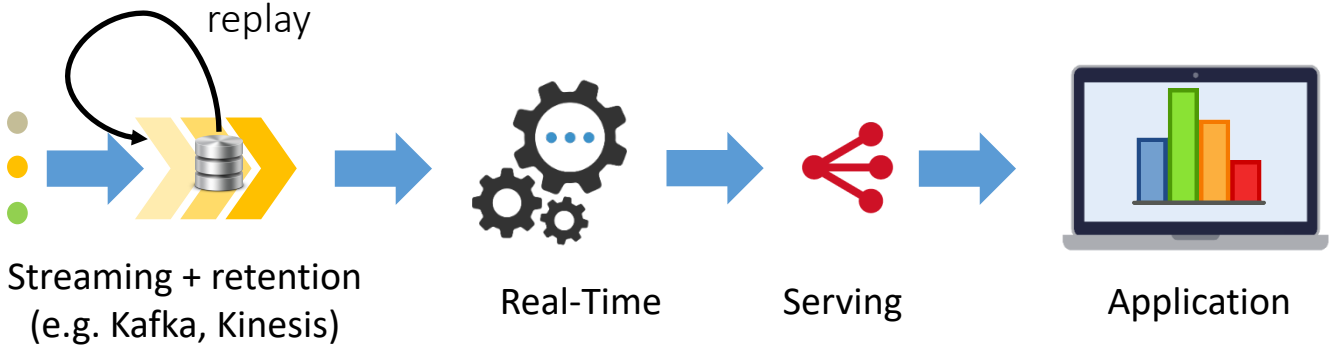
Source: Gessert (2017)

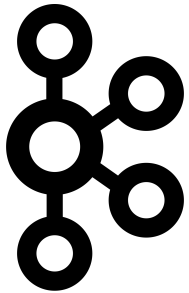


Kappa Architecture

Kappa Architecture (Event Sourcing)

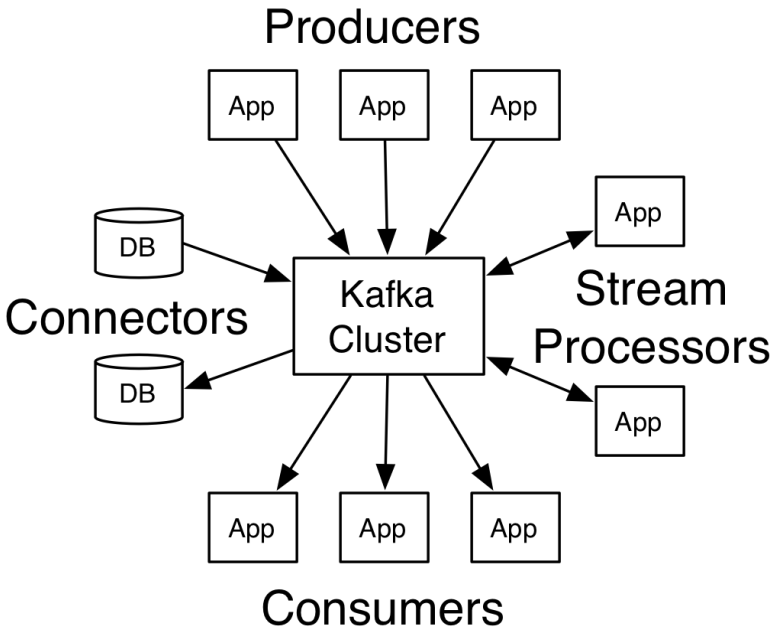
- $Stream(D_{all}) = Batch(D_{all})$
- Simpler than Lambda Architecture
- Data retention for relevant portion of history
- Reasons to forgo Kappa:
  - Legacy batch system that is not easily migrated
  - Special tools only available for a particular batch processor
  - Purely incremental algorithms





Apache Kafka

Apache Kafka



- Created by LinkedIn
- Kafka is run as a cluster on one or more servers.
- The Kafka cluster **stores** streams of records in categories called **topics** as **commit logs**
- Each record consists of a **key**, a **value**, and a **timestamp**

Source: https://kafka.apache.org/intro

Kafka Topic

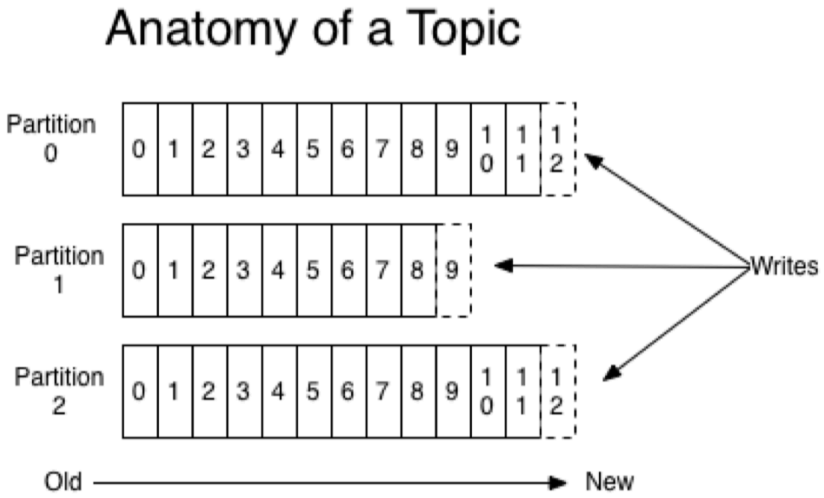
- A **topic**
  - is a **category** (feed name)
  - to which **events** (records)
  - are **published**.
- **Topics** in Kafka are always **multi-subscriber**
  - a topic can have zero, one, or many consumers that subscribe to the data written to it.

Kafka Core APIs

- The **Producer API** allows an application to **publish** a stream records to one or more Kafka **topics**.
- The **Consumer API** allows an application to **subscribe** to one or more **topics** and process the stream of records produced to them.
- The **Streams API** allows an application to act as a **stream processor**, **consuming** an input stream from one or more topics and **producing** an output stream to one or more output topics, effectively transforming the input streams to output streams.
- The **Connector API** allows **building and running** reusable **producers** or **consumers** that **connect** Kafka topics to existing applications or data systems. For example, a connector to a relational database might capture every change to a table.

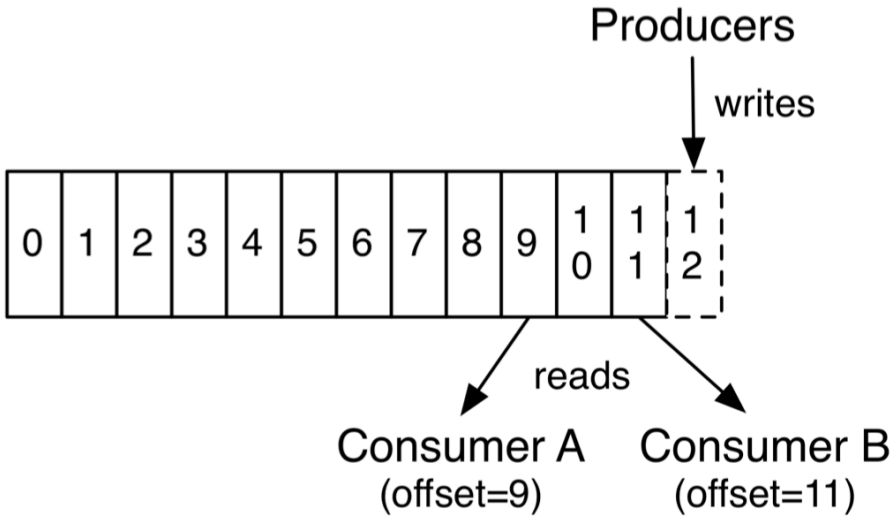
Kafka Partition

- For each **topic**,
  - the Kafka cluster maintains
  - a **partitioned log**.



Kafka Logs

- Each **partition** is an
  - **ordered**, **immutable** sequence of records
  - that is **continually appended** to—a **structured commit log**.



Distribution

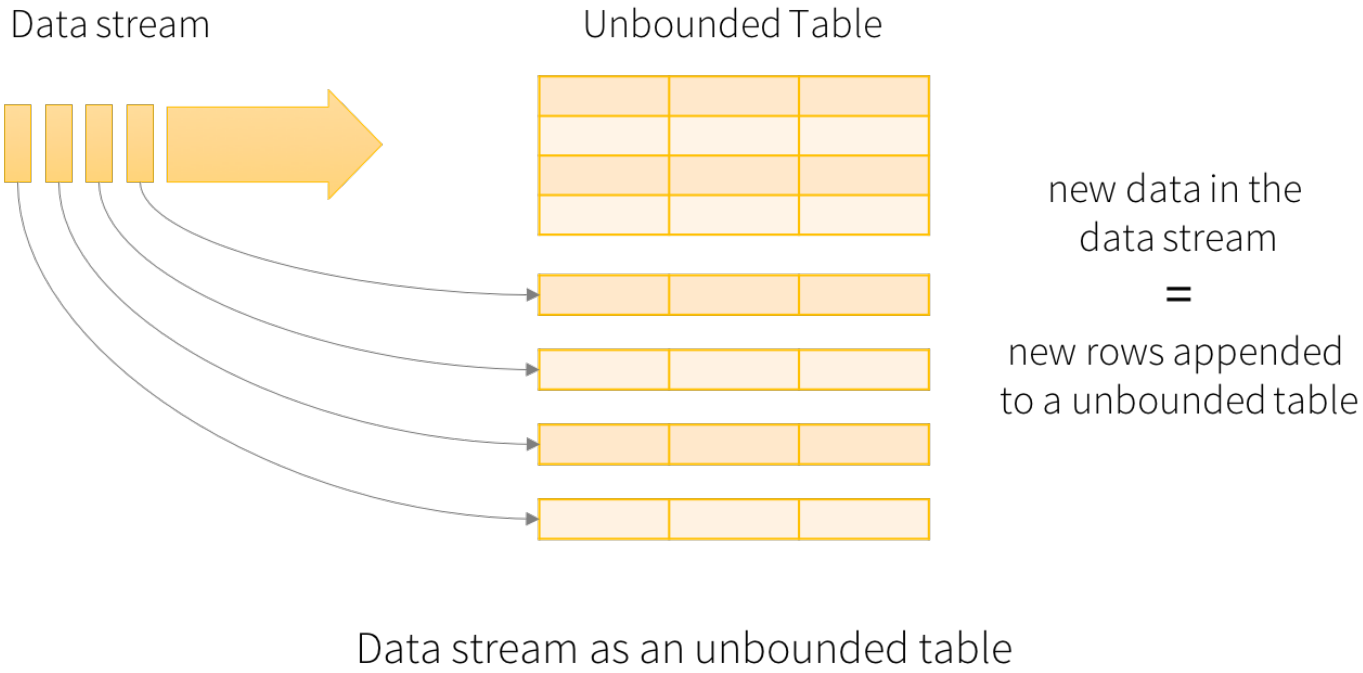
- Each **partition**
  - has one server which acts as the "**leader**" (master)
  - and zero or more servers which act as "**followers**" (slave).
- The **leader handles all read and write requests** for the partition
  - while the followers passively replicate the leader.
  - Follower is only for fault tolerance,
  - not read distribution (for consistency reasons)

Kafka as a Storage System

- Data written to Kafka
  - is **written to disk**
  - and **replicated** for fault tolerance.
- Kafka allows producers
  - to wait on **acknowledgement** so that a write isn't considered complete until it is fully replicated and guaranteed to persist even if the server written to fails.

Spark Structured Streaming

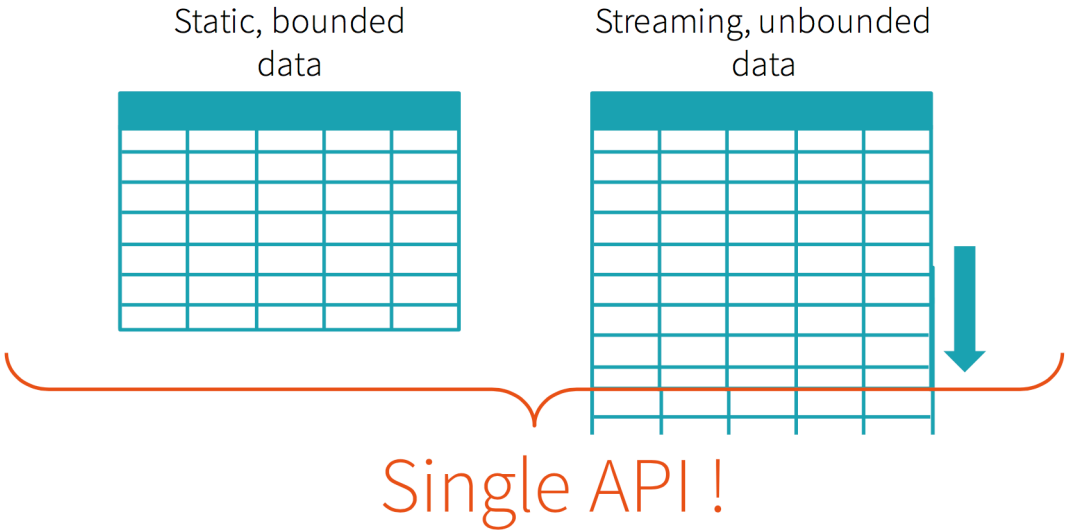
Spark Structured Streaming



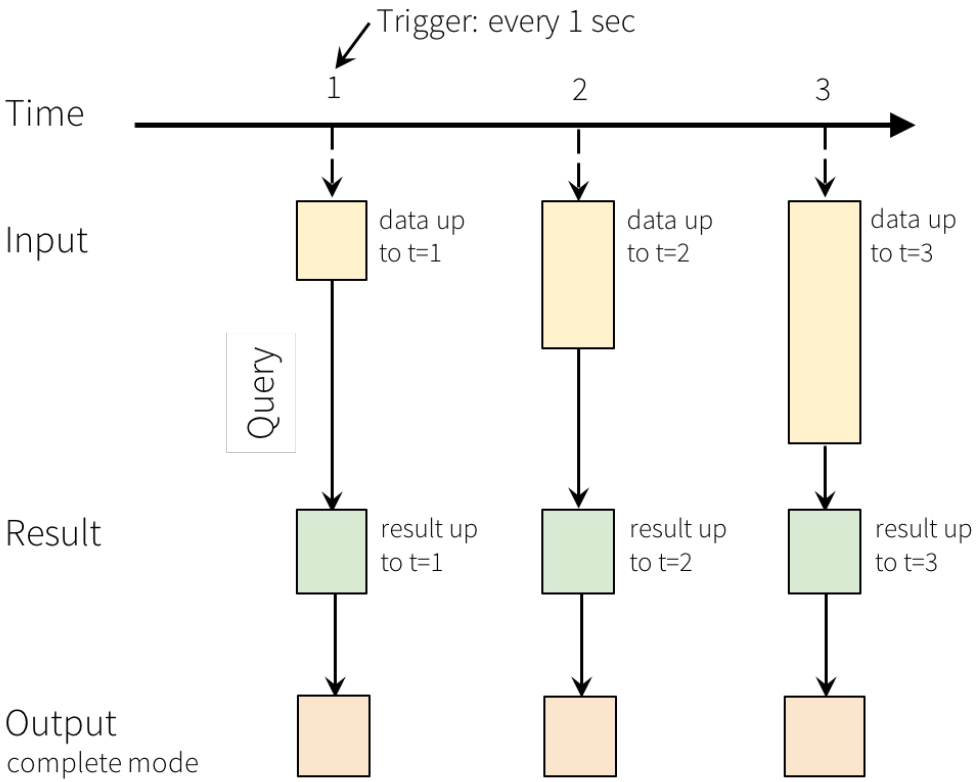
Data stream as an unbounded table

Spark Structured Streaming

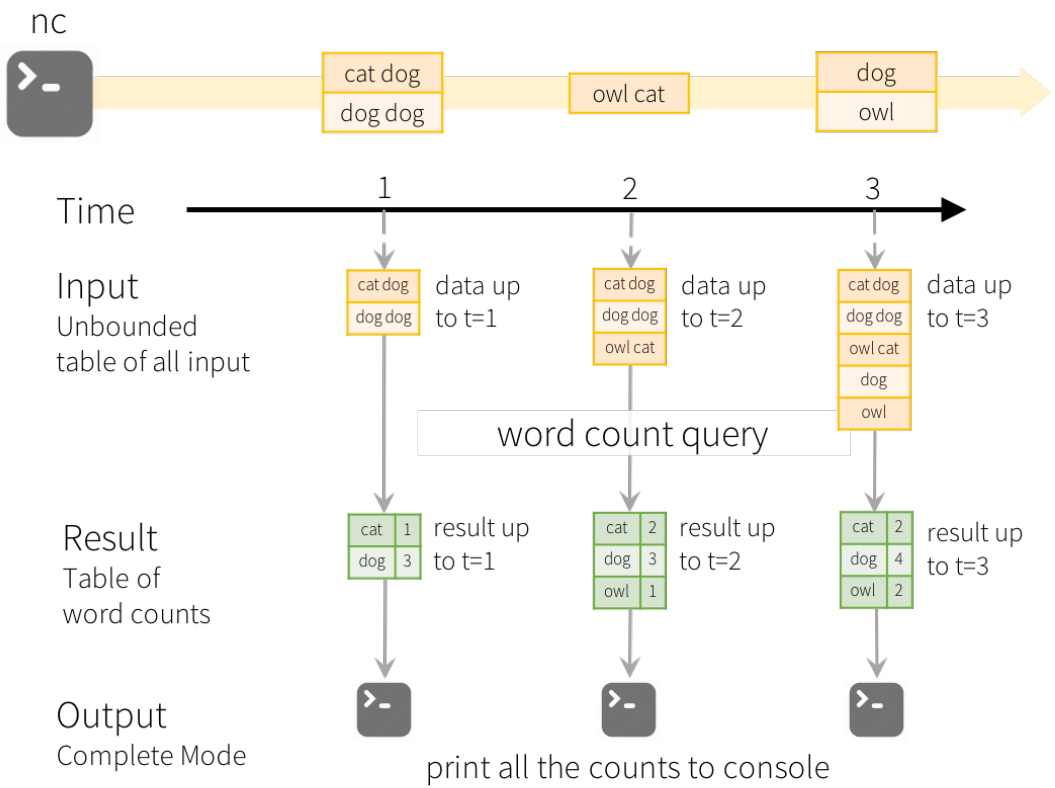
Spark Streaming is based on “Microbatching”



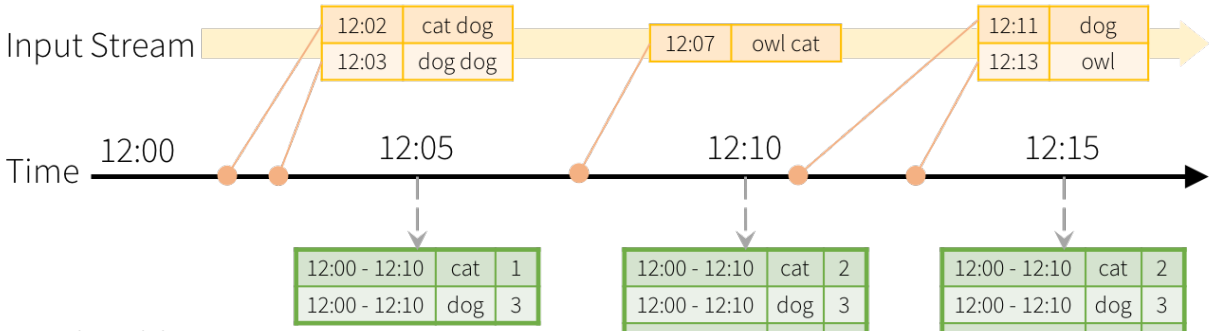
Programming Model for Structured Streaming



Programming Model for Structured Streaming



Window Operations



Result Tables
after 5 minute triggers

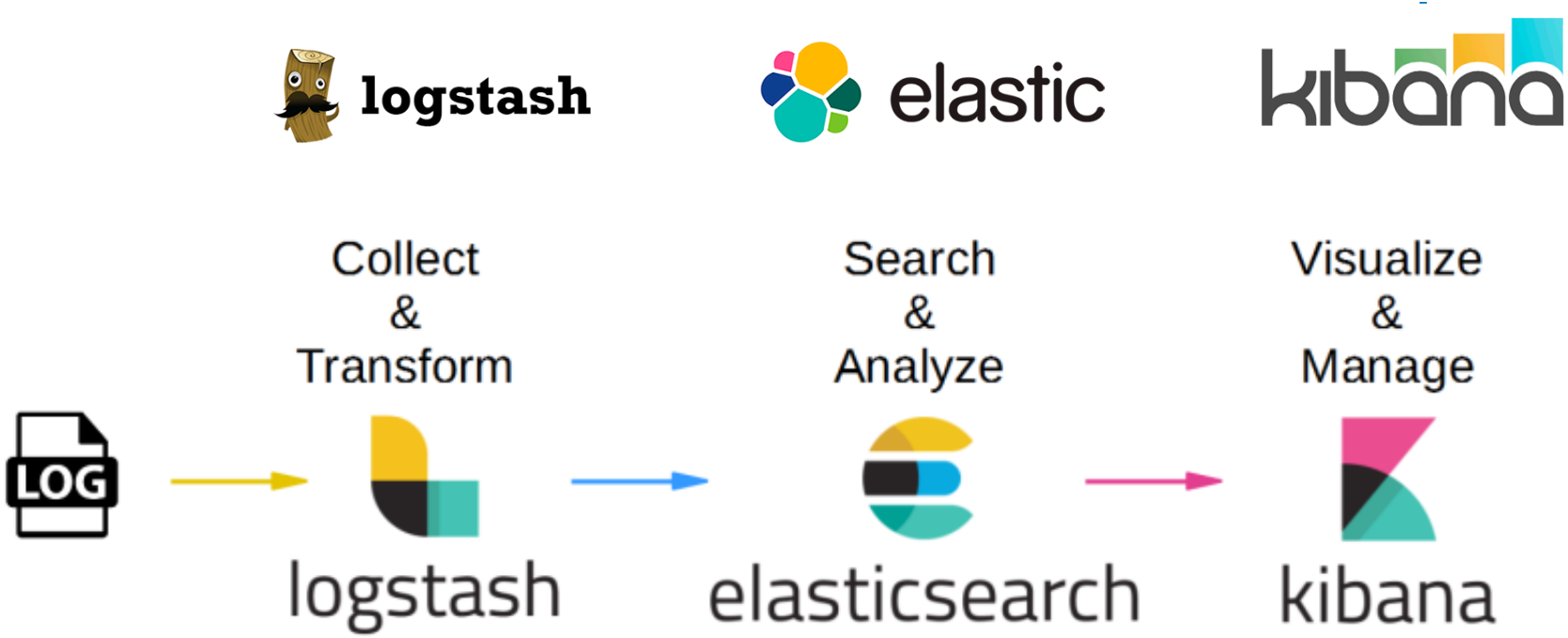
counts incremented for windows
12:00 - 12:10 and 12:05 - 12:15

counts incremented for windows
12:05 - 12:15 and 12:10 - 12:20

Windowed Grouped Aggregation
with 10 min windows, sliding every 5 mins

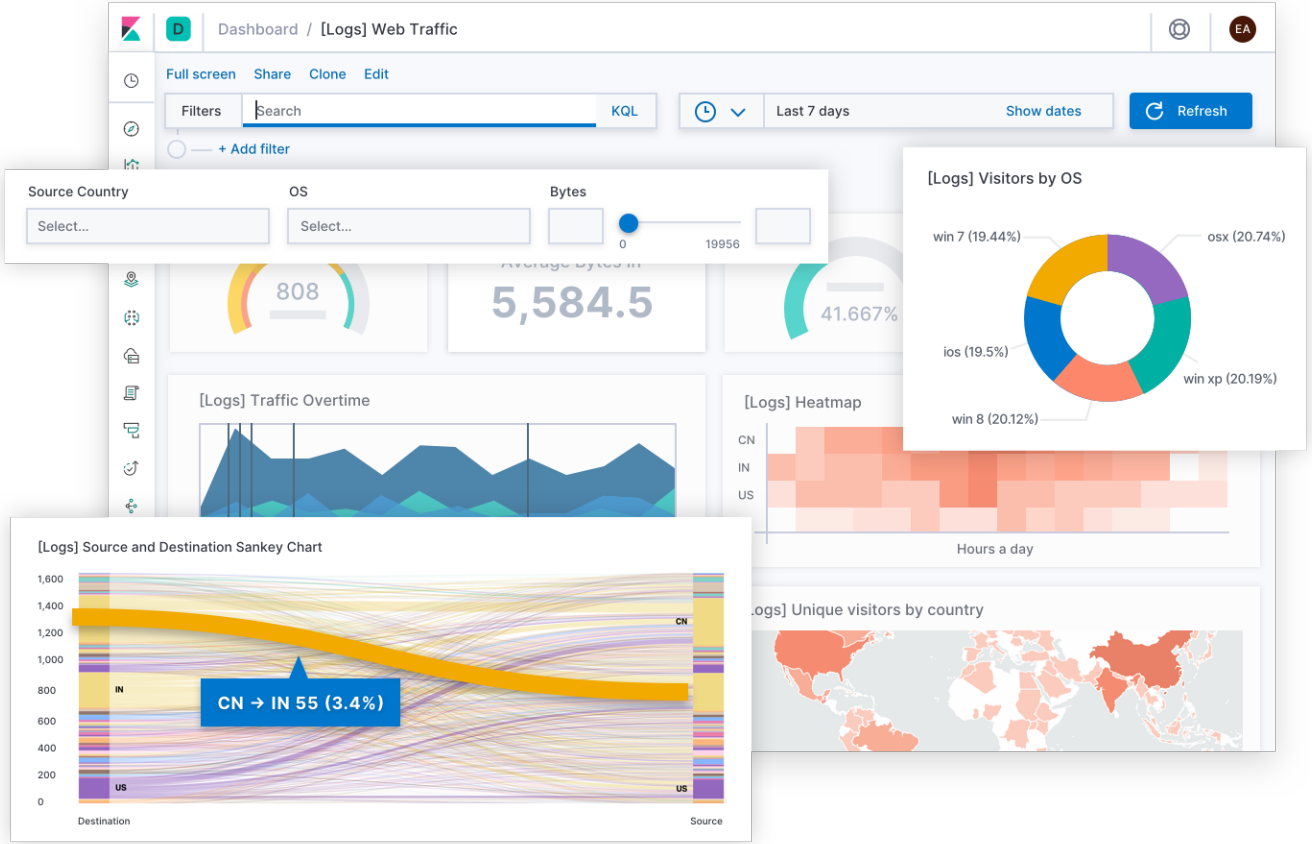
Other Streaming and Messaging Solutions

ELK Stack: Elasticsearch, Logstash and Kibana

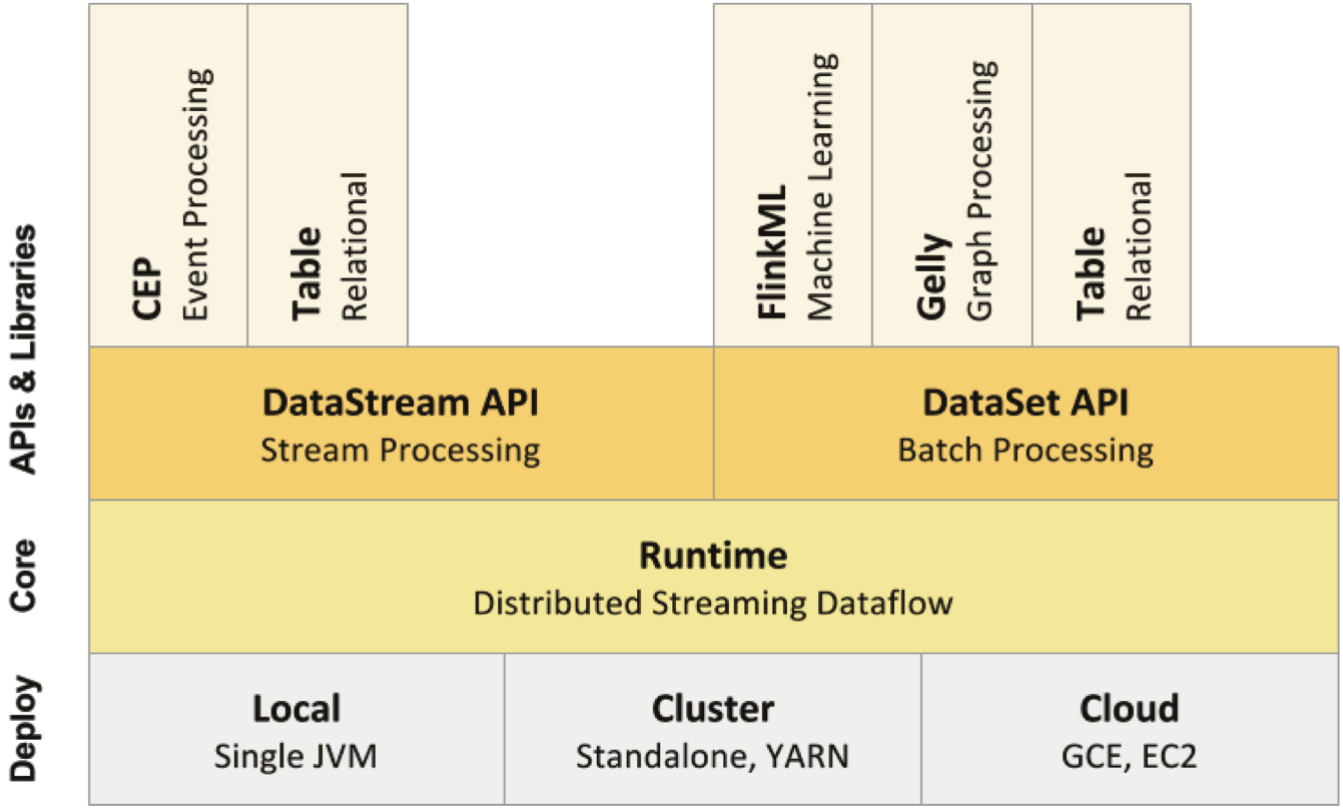


Source: Elastic

Kibana


Dashboard / [Logs] Web Traffic
Full screen
Share
Clone
Edit
Filters
Search
KQL
Last 7 days
Show dates
Refresh
+ Add filter
Source Country
Select...
OS
Select...
Bytes
0
19956
808
5,584.5
41.667%
[Logs] Visitors by OS
win 7 (19.44%)
osx (20.74%)
ios (19.5%)
win xp (20.19%)
win 8 (20.12%)
[Logs] Traffic Overtime
[Logs] Heatmap
CN
IN
US
Hours a day
[Logs] Source and Destination Sankey Chart
1,600
1,400
1,200
1,000
800
600
400
200
0
CN → IN 55 (3.4%)
Destination
Source
[Logs] Unique visitors by country

Apache Flink



Source: Freiknecht & Papp (2018)

Amazon AWS Messaging

Amazon MQ

WHAT IT IS

Managed message broker service for Apache ActiveMQ that makes it easy to set up and operate message brokers in the cloud and enable hybrid architecture

USE CASE

Migrate to a managed message broker to automate software administration and maintenance, without having to re-write existing applications

Watch Use Case Video >>

COOL CAPABILITIES

Compatible with industry standard APIs and protocols, incl. JMS, NMS, AMQP, STOMP, MQTT, and WebSocket

Learn More >>

Amazon SQS

WHAT IT IS

Simple, flexible, fully managed message queuing service for reliably and continuously exchanging any volume of messages from anywhere

USE CASE

Build decoupled, highly scalable microservices, distributed systems, and serverless applications in the cloud

Watch Use Case Video >>

COOL CAPABILITIES

Nearly infinite scalability and ability to increase message throughput without pre-provisioning capacity

Learn More >>

Amazon SNS

WHAT IT IS

Simple, flexible, fully managed publish/subscribe messaging and mobile push notification service for high throughput, highly reliable message delivery

USE CASE

Push messages to a variety of endpoints and clients in distributed systems, microservices, and serverless applications and enable event-driven architecture

Watch Use Case Video >>

COOL CAPABILITIES

Highly reliable delivery of any volume of messages to any number of recipients across multiple protocols

Learn More >>

Source: AWS

Amazon AWS Messaging

Amazon Pinpoint

WHAT IT IS

Customer engagement platform for managing targeted and transactional multi-channel engagement using email, mobile push, SMS, and Lambda

USE CASE

Deliver the right message to the right customer at the right time to improve engagement and conversion

Watch Use Case Video >>

COOL CAPABILITIES

Integrated analytics and messaging to drive insights and influence how customers interact with your apps

Learn More >>

Amazon Kinesis Streams

WHAT IT IS

Fully managed, highly scalable service for collecting and processing real-time data streams for analytics and machine learning

USE CASE

Build custom, real-time applications that process data streams using popular stream processing frameworks

Watch Use Case Video >>

COOL CAPABILITIES

Continuously capture and store terabytes of data per hour from hundreds of thousands of sources

Learn More >>

AWS IoT Message Broker

WHAT IT IS

Fully managed publish/subscribe broker service that enables the sending and receiving of messages to and from AWS IoT Core

USE CASE

Send messages to/from devices and AWS IoT apps in a secure fashion using MQTT, HTTP, and WebSockets

Watch Use Case Video >>

COOL CAPABILITIES

Allows you to send messages to, or receive messages from, millions of devices

Learn More >>

Source: AWS