Stream Processing

Lecture 04

2022/2023

Table of Contents

Table of Contents

- Stream processing ecosystem
  - **Apache Flume**
  - Apache Kafka
  - AWS Stream Processing Ecosystem

Apache Flume: Motivation

- In many application scenarios, data comes from multiple sources and needs to be conveniently prepared before processing…
  - e.g. logging information needs to be ingested into HDFS, before map reduce jobs can process them…
- Most processing systems rely on external tools that perform the necessary adaptation before data is processed

Flume: What is it?

- Flume is a system for collecting, aggregating, and moving large amounts of data.
  - it has a simple and flexible architecture based on streaming data flows;
  - it is robust and fault tolerant, with tunable reliability mechanisms and recovery mechanisms.

Flume: Architecture (1)

- Core Concepts
  - **event** - unit of data flow having a byte payload and an optional set of string attributes.

Flume: Architecture (2)

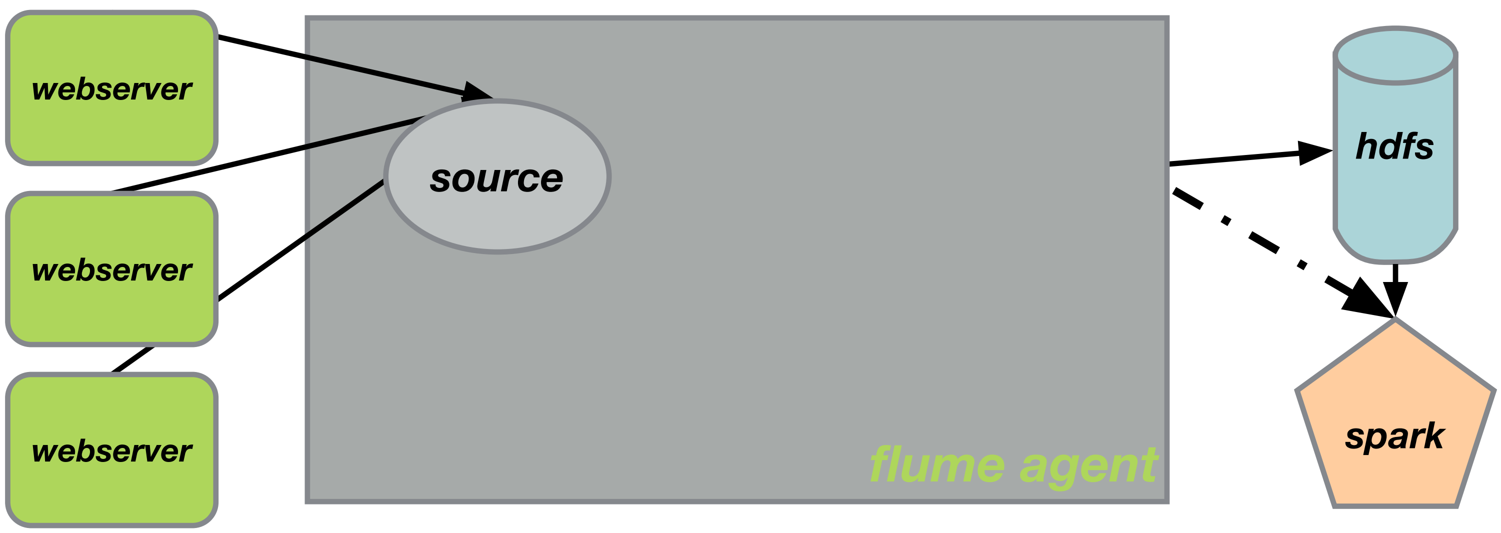
- **flume agent** - a (JVM) process that hosts the components through which events flow from an external source to the target destination.

Flume: Architecture (3)

– **sources** - acquire events produced by external sources like a set of web servers.

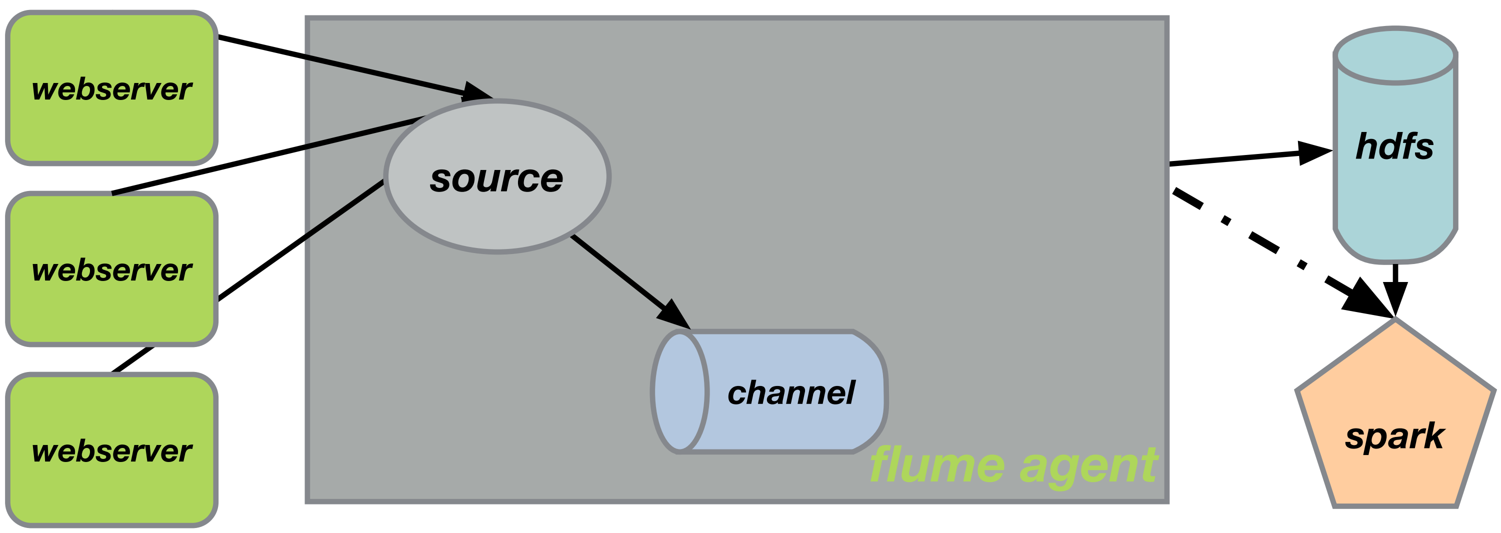
Flume: Architecture (4)

- **sources** - acquire events produced by external sources like a set of web servers.
  - an agent can aggregate the input of multiple sources



Flume: Architecture (5)

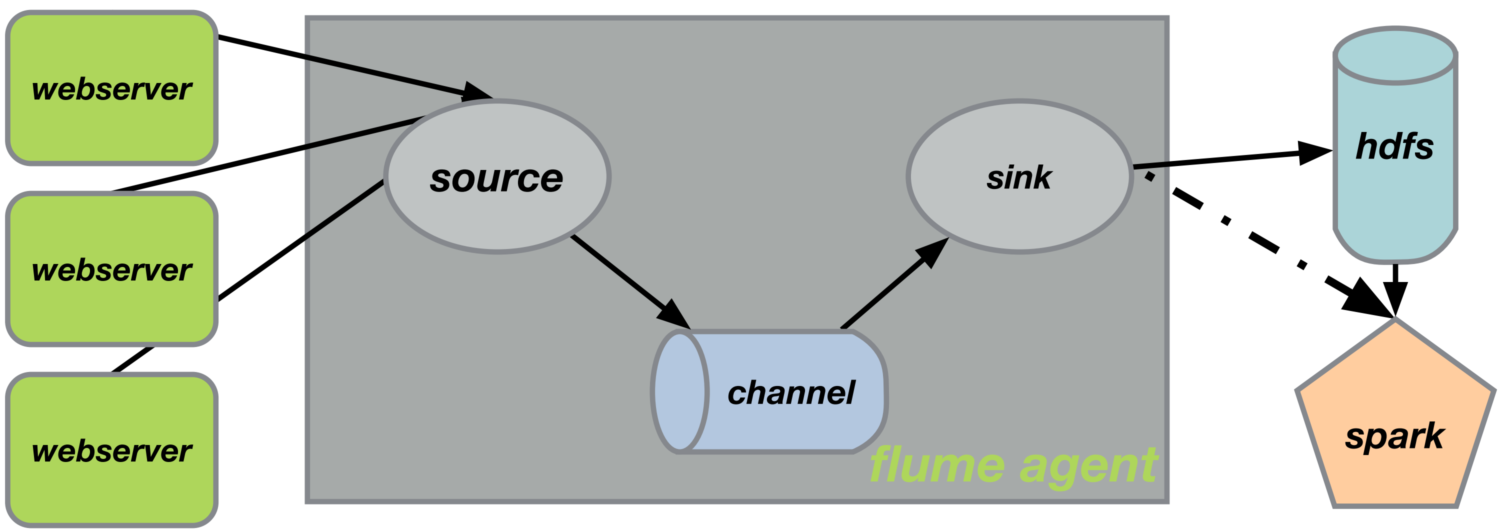
- **channels** - move events around
  - the type determines the delivery guarantees
  - reliable channels may not respect order



Flume: Architecture (6)

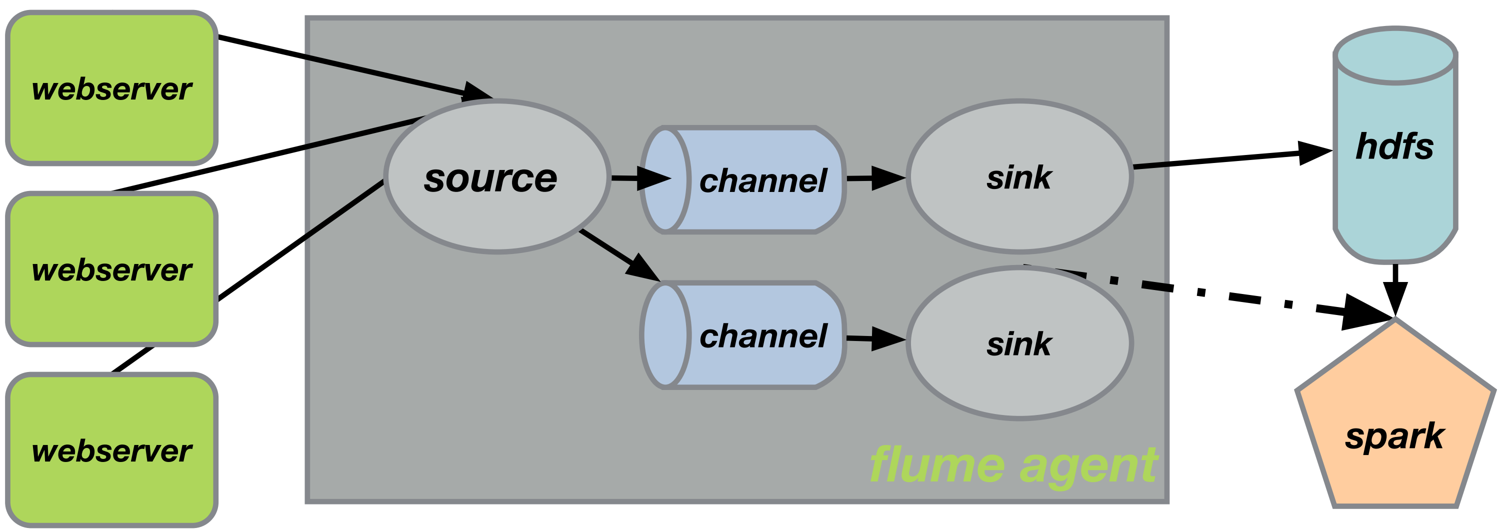
– **sinks** - output events to external consumers

  - an agent can feed multiple external consumers, using multiple sinks



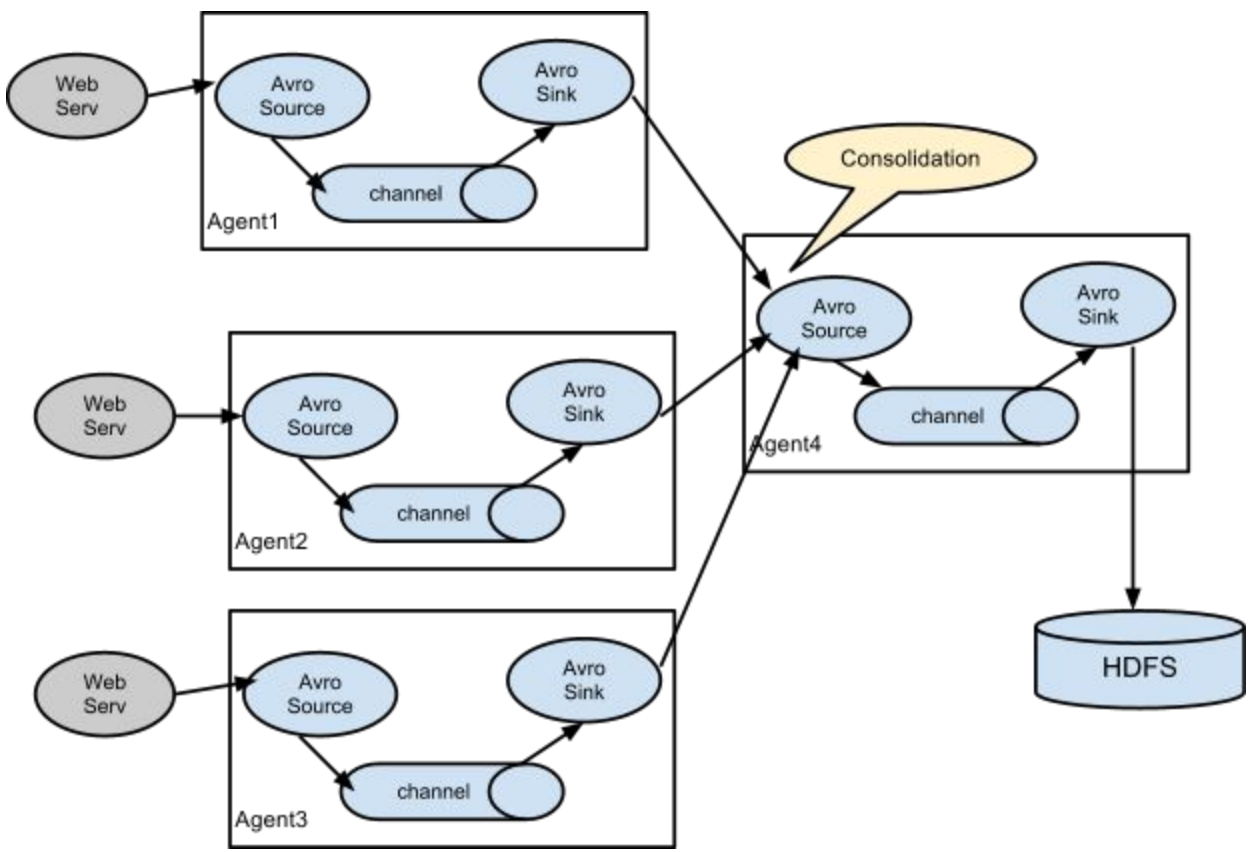
Flume: Architecture (7)

- **agents** - can have more complex topologies, with multiple sources, channels and skinks
  - agents can also be chained together



Flume: Architecture (8)

– **agents** - can have more complex topologies, with multiple sources, channels and skinks
  - agents can also be chained together

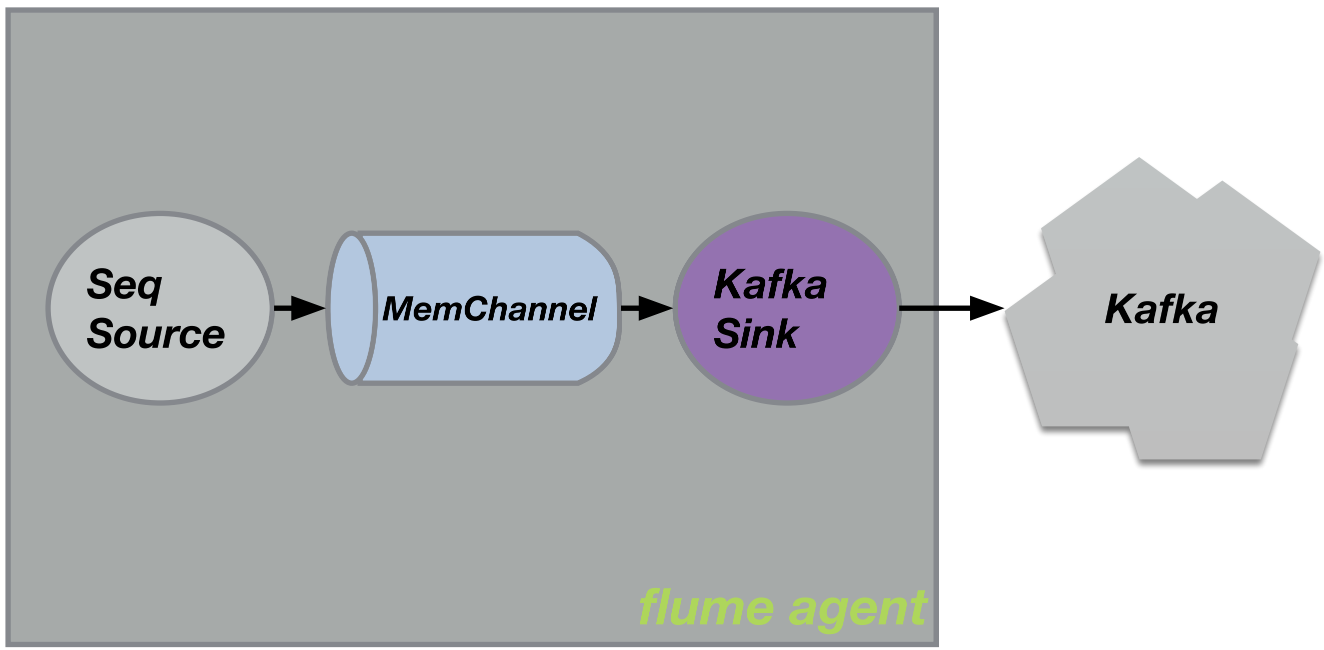


Flume: Programming

- A Flume agent is programmed via a configuration file
  - the file describes its topology in terms of sources, sinks, channels and how they connect together
  - for each component (sink, source, channel) the file also describes its parameters, in particular type
  - There's a library of sources, sinks and channels that can be used

Example: Sink to Kafka

- Using Flume to ingest a stream from a test source into Kafka



 # Name the components on this agent
a1.sources = r1
a1.sinks = k1
a1.channels = c1
 # Describe/configure the source
a1.sources.r1.type = seq

 # Describe/configure the sink
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.topic = mytopic
a1.sinks.k1.kafka.bootstrap.servers = localhost:9092
a1.sinks.k1.kafka.flumeBatchSize = 20
a1.sinks.k1.kafka.producer.acks = 1
a1.sinks.k1.kafka.producer.linger.ms = 1
a1.sinks.k1.kafka.producer.compression.type = snappy
 # Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100
 # Bind the source and sink to the channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1

 # Name the components on this agent
a1.sources = r1
a1.sinks = k1
a1.channels = c1

Define the name of the components. e.g. there will be a source r1.

 # Describe/configure the source
a1.sources.r1.type = seq

 # Describe/configure the sink
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.topic = mytopic
a1.sinks.k1.kafka.bootstrap.servers = localhost:9092
a1.sinks.k1.kafka.flumeBatchSize = 20
a1.sinks.k1.kafka.producer.acks = 1
a1.sinks.k1.kafka.producer.linger.ms = 1
a1.sinks.k1.kafka.producer.compression.type = snappy
 # Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100
 # Bind the source and sink to the channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1

 # Name the components on this agent

```
a1.sources = r1
a1.sinks = k1
a1.channels = c1
```

 # Describe/configure the source

```
a1.sources.r1.type = seq
```

Simple source that generates sequential events (1,2,3,...)

 # Describe/configure the sink

```
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.topic = mytopic
a1.sinks.k1.kafka.bootstrap.servers = localhost:9092
a1.sinks.k1.kafka.flumeBatchSize = 20
a1.sinks.k1.kafka.producer.acks = 1
a1.sinks.k1.kafka.producer.linger.ms = 1
a1.sinks.k1.kafka.producer.compression.type = snappy
```

 # Use a channel which buffers events in memory

```
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100
```

 # Bind the source and sink to the channel

```
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

 # Name the components on this agent
a1.sources = r1
a1.sinks = k1
a1.channels = c1
 # Describe/configure the source
a1.sources.r1.type = seq

 # Describe/configure the sink
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.topic = mytopic
a1.sinks.k1.kafka.bootstrap.servers = localhost:9092
a1.sinks.k1.kafka.flumeBatchSize = 20
a1.sinks.k1.kafka.producer.acks = 1
a1.sinks.k1.kafka.producer.linger.ms = 1
a1.sinks.k1.kafka.producer.compression.type = snappy

Sink to send event – Kafka in this case.

 # Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100
 # Bind the source and sink to the channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1

 # Name the components on this agent
a1.sources = r1
a1.sinks = k1
a1.channels = c1
 # Describe/configure the source
a1.sources.r1.type = seq

 # Describe/configure the sink
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.topic = mytopic
a1.sinks.k1.kafka.bootstrap.servers = localhost:9092
a1.sinks.k1.kafka.flumeBatchSize = 20
a1.sinks.k1.kafka.producer.acks = 1
a1.sinks.k1.kafka.producer.linger.ms = 1
a1.sinks.k1.kafka.producer.compression.type = snappy
 # Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100
 # Bind the source and sink to the channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1

Simple internal memory-based event-queue.

```
# Name the components on this agent
a1.sources = r1
a1.sinks = k1
a1.channels = c1
# Describe/configure the source
a1.sources.r1.type = seq


# Describe/configure the sink
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.topic = mytopic
a1.sinks.k1.kafka.bootstrap.servers = localhost:9092
a1.sinks.k1.kafka.flumeBatchSize = 20
a1.sinks.k1.kafka.producer.acks = 1
a1.sinks.k1.kafka.producer.linger.ms = 1
a1.sinks.k1.kafka.producer.compression.type = snappy
# Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100
# Bind the source and sink to the channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

Connects the sources and sinks to the channel.

pause

Table of Contents

- Stream processing ecosystem
  - Apache Flume
  - **Apache Kafka**
  - AWS Stream Processing Ecosystem

Publish/subscribe

A form of indirect communication:

- senders (publishers) do not address messages to specific receivers (subscribers).

- messages are relayed to subscribers (if any) that have shown interest in particular classes of messages (topics) or messages with particular contents (cotent-routing)

Publish/subscribe concepts

- Data producers are decoupled from data consumers
  - Publishers don’t know who the consumers are and vice versa
  - Publishers and subscribers may exist at different times
- A queue can provide durable storage of messages for some length of time
- A message can be consumed from the queue [0..n] times
  - No requirement that a message is delivered exactly once or at least once
- 1:n relationship between publishers and subscribers
  - “Fan-out” effect
  - A single message need not be delivered to all subscribers

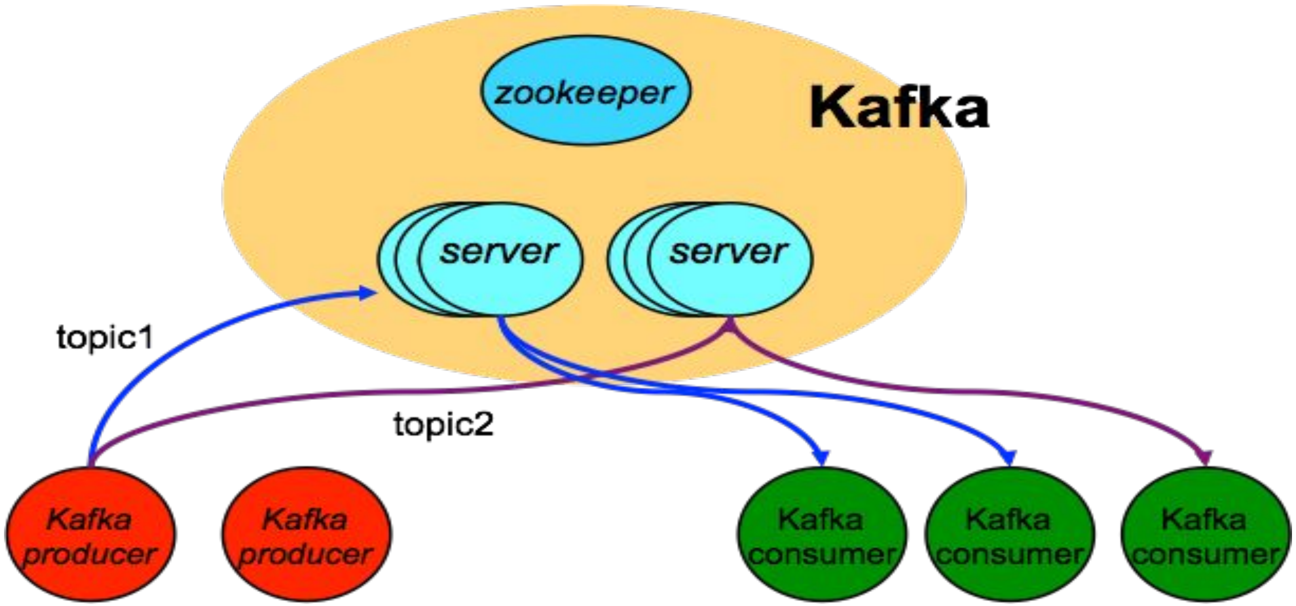
Kafka: What is it?

- Apache Kafka is a topic based publish-subscribe messaging system
  - In the context of distributed processing, it is often used to ingest data streams into a stream processing system

Kafka: What is it?

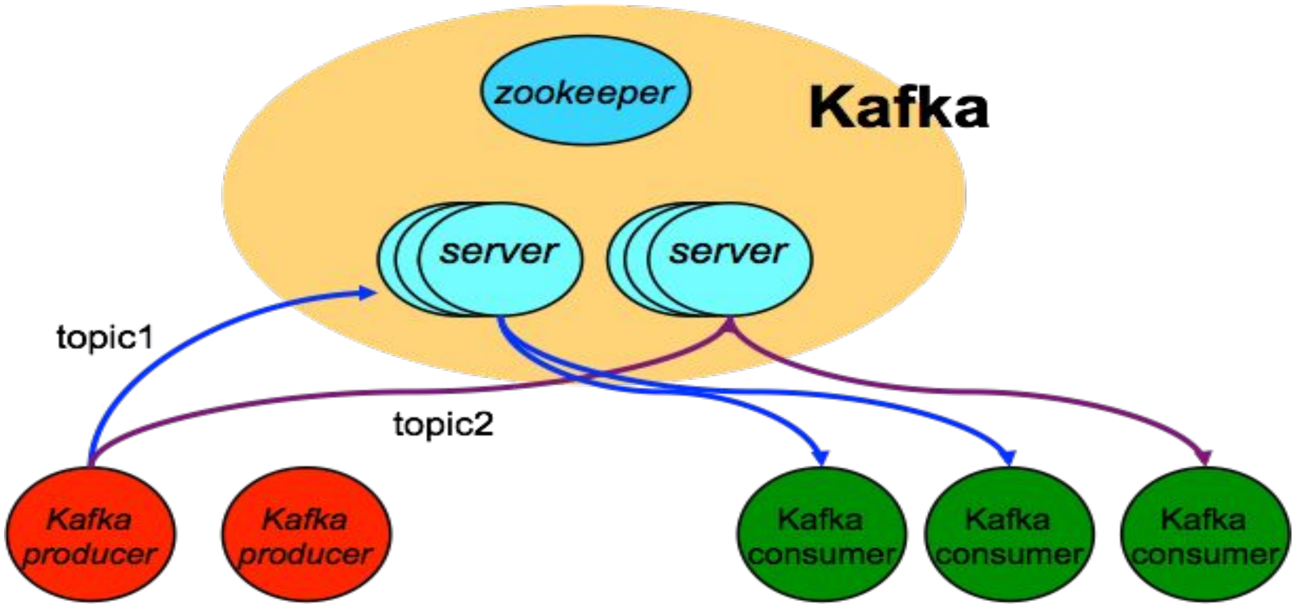
- Apache Kafka is a topic based publish-subscribe messaging system
  - Mediates and decouples interactions between event producers and the consumers
    - Producers send events to Brokers (Kafka Servers)
    - Consumers receive events via the Brokers
    - Don’t need to **know** each other directly or **execute** at the same time

Kafka: Architecture



- **Producer API** to produce a streams or records
- **Consumer API** to consume a stream of records

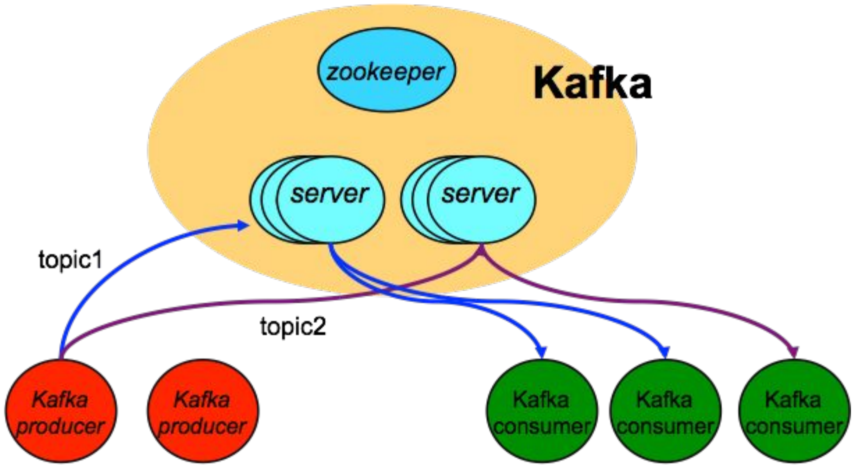
Kafka: Architecture



- **Broker server**: Kafka server that runs in a Kafka Cluster. Brokers form a cluster. Cluster consists on many Kafka Brokers on many servers.
- **ZooKeeper**: Coordinates the brokers/cluster topology: configuration information and leadership election for Broker Topic Partition Leaders (optional in recent versions)

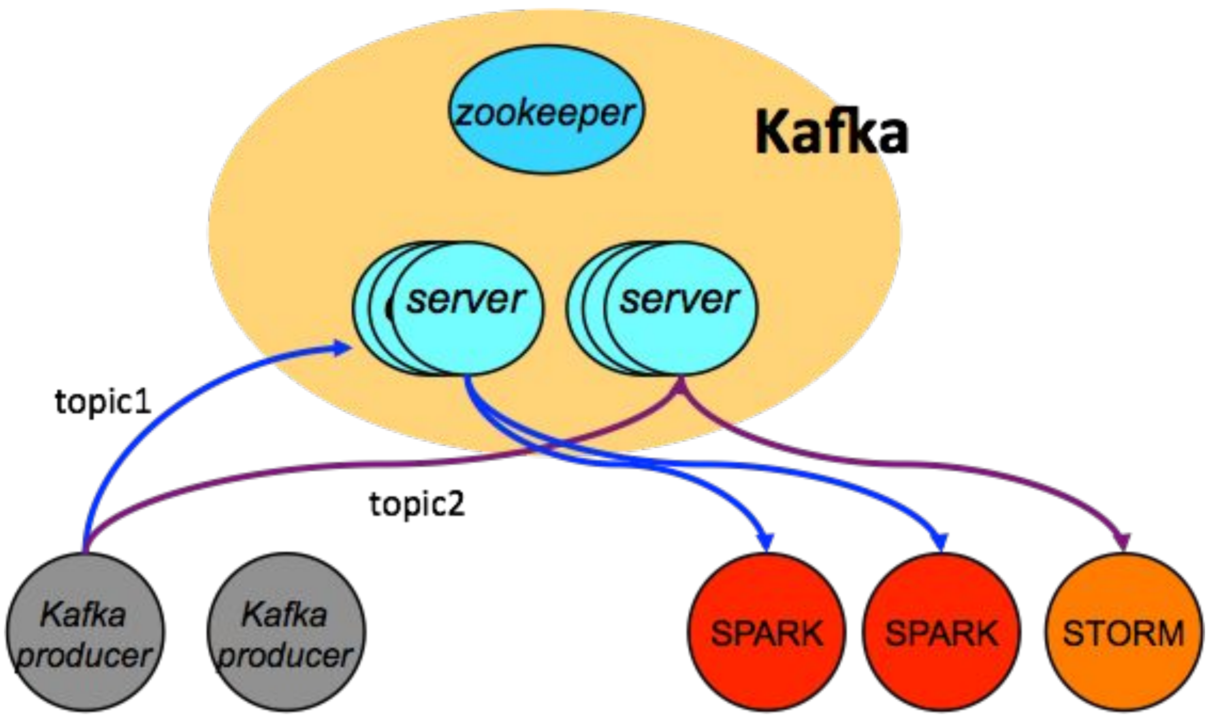
Kafka: Key Facts

- Kafka is implemented as distributed commit log
  - offers event persistency, backed by the filesystem
  - fault-tolerance and high-availability due to **replication**
  - high throughput via **partitioning**



Kafka: Usage Scenario

- Kafka can interface directly with many stream-processing engines, such as <u>Spark Streaming</u>, Storm and Flink



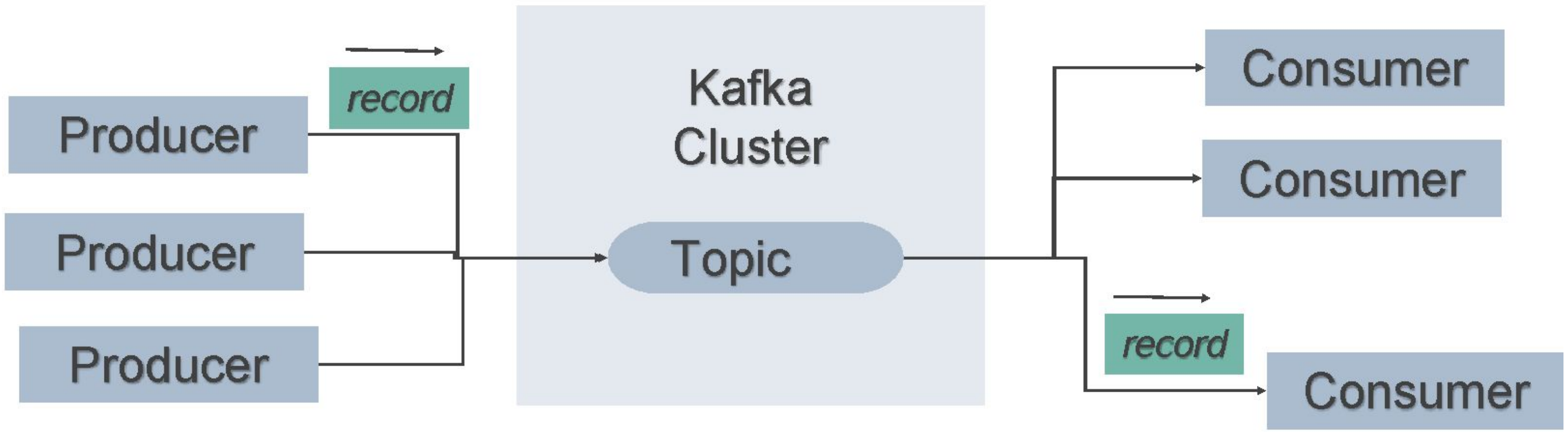
Kafka: records and topics

- Records are immutable and have a key (optional), value and timestamp
- A **topic** is a stream of records ("/orders", "/user-signups"), feed name
  - Topics stored on disk
  - Topics broken up in partitions and segments (parts of Topic Log)

Kafka record retention

- Kafka cluster retains all published records
  - *Time based* – configurable retention period
  - *Size based* – configurable based on size
  - *Compaction* – keeps latest record given key
- E.g.: retention policy of three days or two weeks or a month
- An event is available for consumption until discarded by time, size or compaction

Kafka messaging

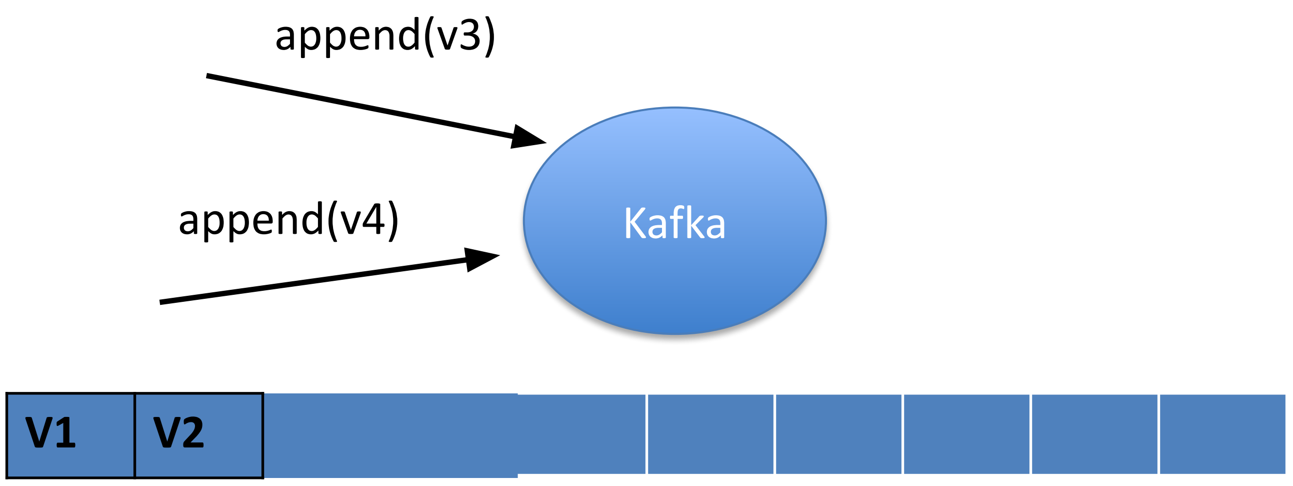


Message processing

- Producers write to and Consumers read from Topic(s)
- Producer(s) append Records at end of Topic log
- Consumers read from Kafka at their own cadence
  - Each Consumer (Consumer Group) tracks offset from where they left off reading
- Partitions can be distributed on different machines in a cluster
  - High performance with horizontal scalability; and failover with replication

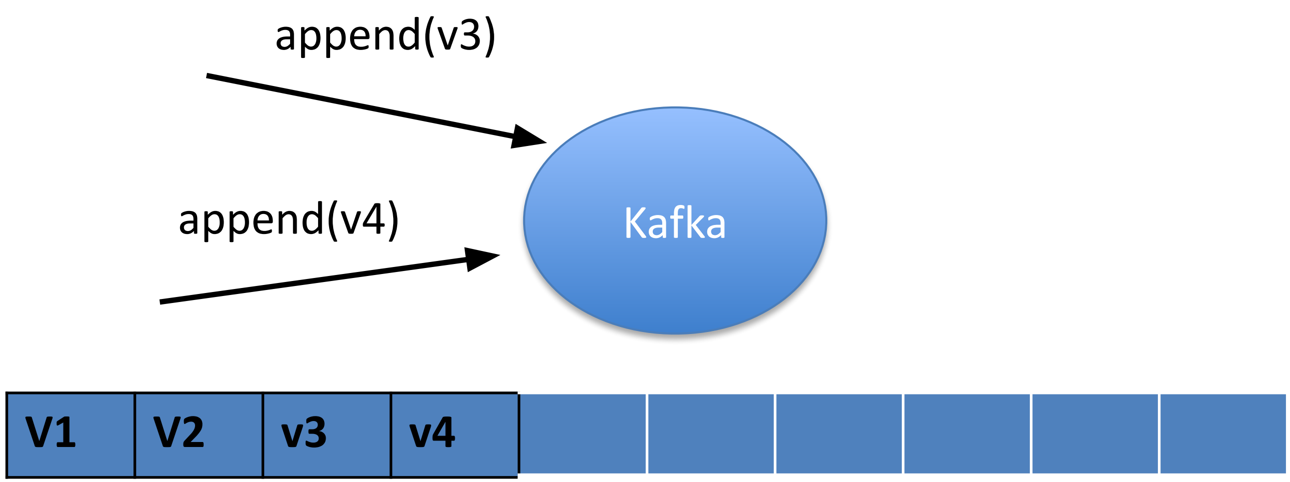
Message processing

- Producers write to and Consumers read from Topic(s)
- Producer(s) append Records at the end of Topic log



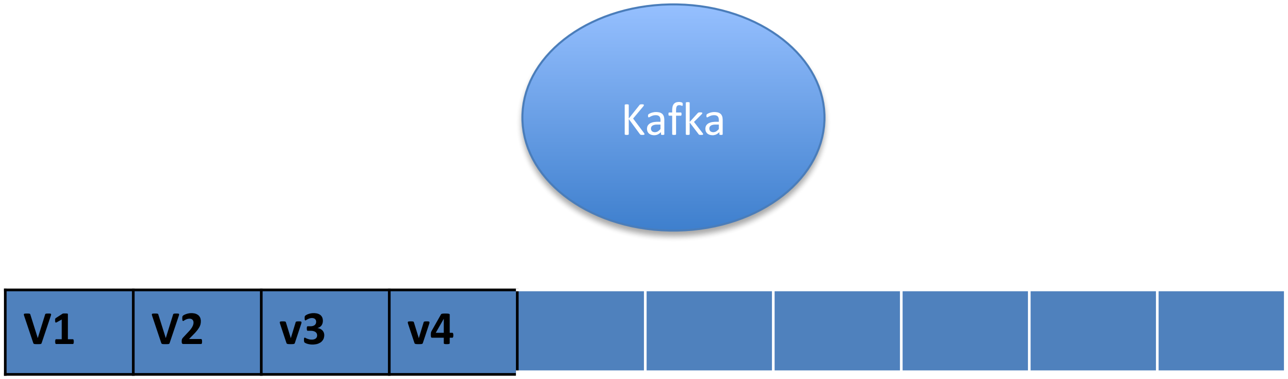
Message processing

- Producers write to and Consumers read from Topic(s)
- Producer(s) append Records at end of Topic log. Records are totally ordered (within a given partition).



Message processing

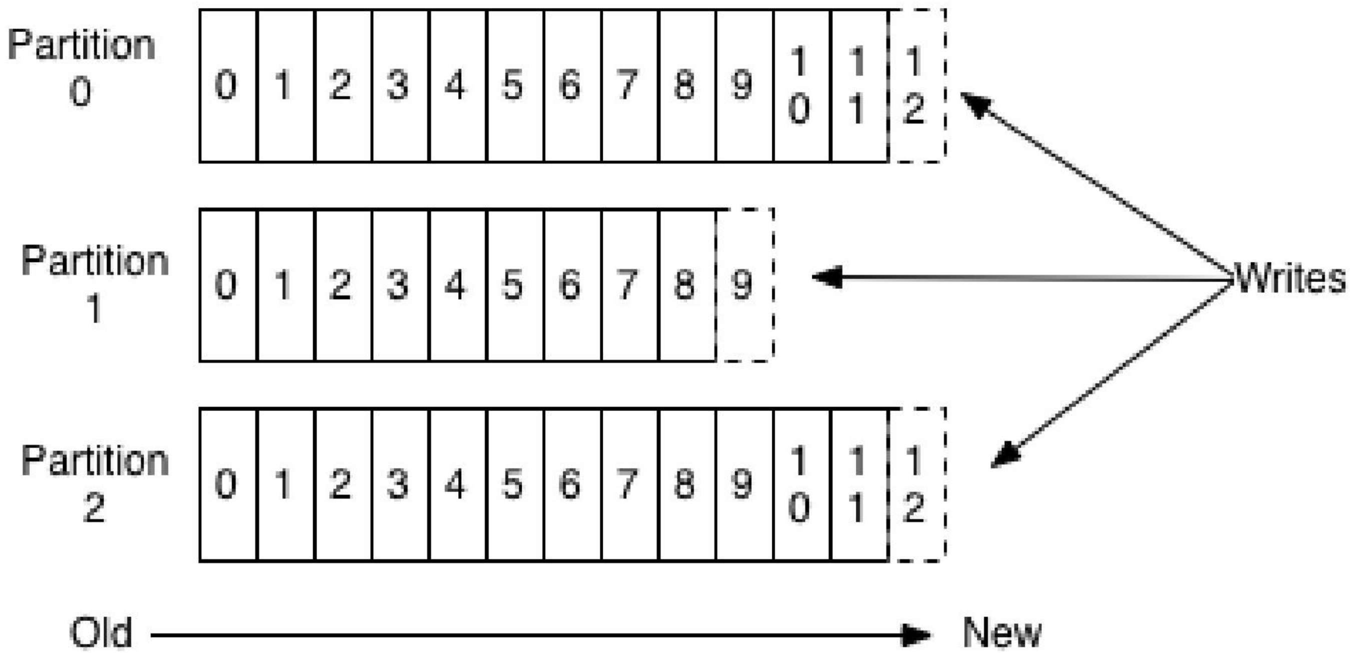
- Consumers read from Kafka at their own cadence
  - Each Consumer (Consumer Group) tracks the **offset** from where they left off reading
- Partitions can be distributed on different machines in a cluster
  - High performance with horizontal scalability and failover with replication



Topic partitions

- Topics are broken up into partitions
  - Key of record determines which partition will be used
  - Partitions can be replicated to multiple brokers
- Partitions are used to scale Kafka across many servers
- Partitions are used to facilitate parallel producers and consumers
  - Records are consumed in parallel up to the number of partitions

Topic partitions



Topic partitions: order

- Order is maintained only in a single partition
  - Partition is an ordered, immutable sequence of records that is continually appended to
- Records in partitions are assigned sequential id number called the **offset**
- The **offset** identifies each record within the partition

Kafka Producers and Partitions

- Producers send records to topics
- Producer picks which partition to send record to per topic
  - Can be done in a round-robin
  - Can be based on priority
  - Typically based on key of record
  - Kafka *default partitioner* for Java uses hash of keys to choose partitions, or a round-robin strategy if no key

Kafka Consumers

- Consumers are grouped into a Consumer Group
  - A consumer group has a unique id
  - Each consumer group maintains its own offset
  - There might be multiple consumer groups
- A Record is delivered to one Consumer in a Consumer Group
- Each consumer in consumer groups takes records and only one consumer in group gets the same record
  - Consumers in Consumer Group load balance record consumption

Kafka Consumers (cont.)

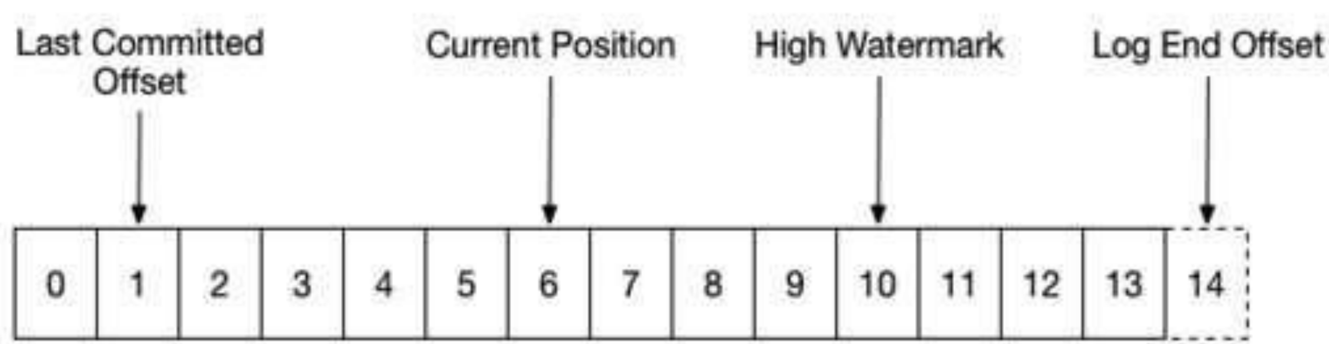
- Kafka divides partitions over consumers in a Consumer Group
  - Each Consumer is the exclusive consumer of a "fair share" of partitions
- Consumer management is handled by Kafka, with one server becoming the group coordinator
  - assigns partitions when new members arrive – there is, at most, one consumer per partition
  - or reassign partitions when group members leave or topic changes
- When Consumer group is created, offset set according to reset policy of topic

Consumer fault tolerance

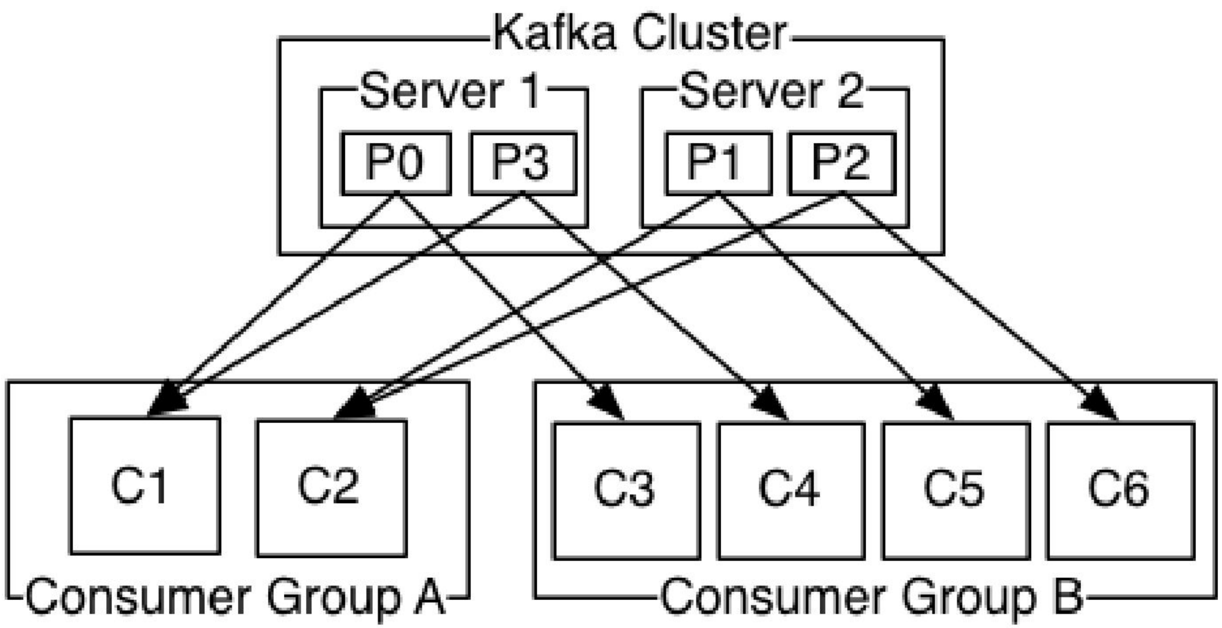
- Consumers notify broker when it successfully processed a record
  - Broker advances offset
- If Consumer fails before sending commit offset to Kafka broker
  - different Consumer can continue from the last committed offset
  - some Kafka records could be reprocessed
    - At least once behavior
    - Message processing should be idempotent

Log offsets

- "Log end offset" is the offset of the last record written to log partition and where Producers write to next
- "High watermark" is the offset of the last record successfully replicated to all partitions followers
- Consumer only reads up to “high watermark”. Consumer cannot read un-replicated data



Example

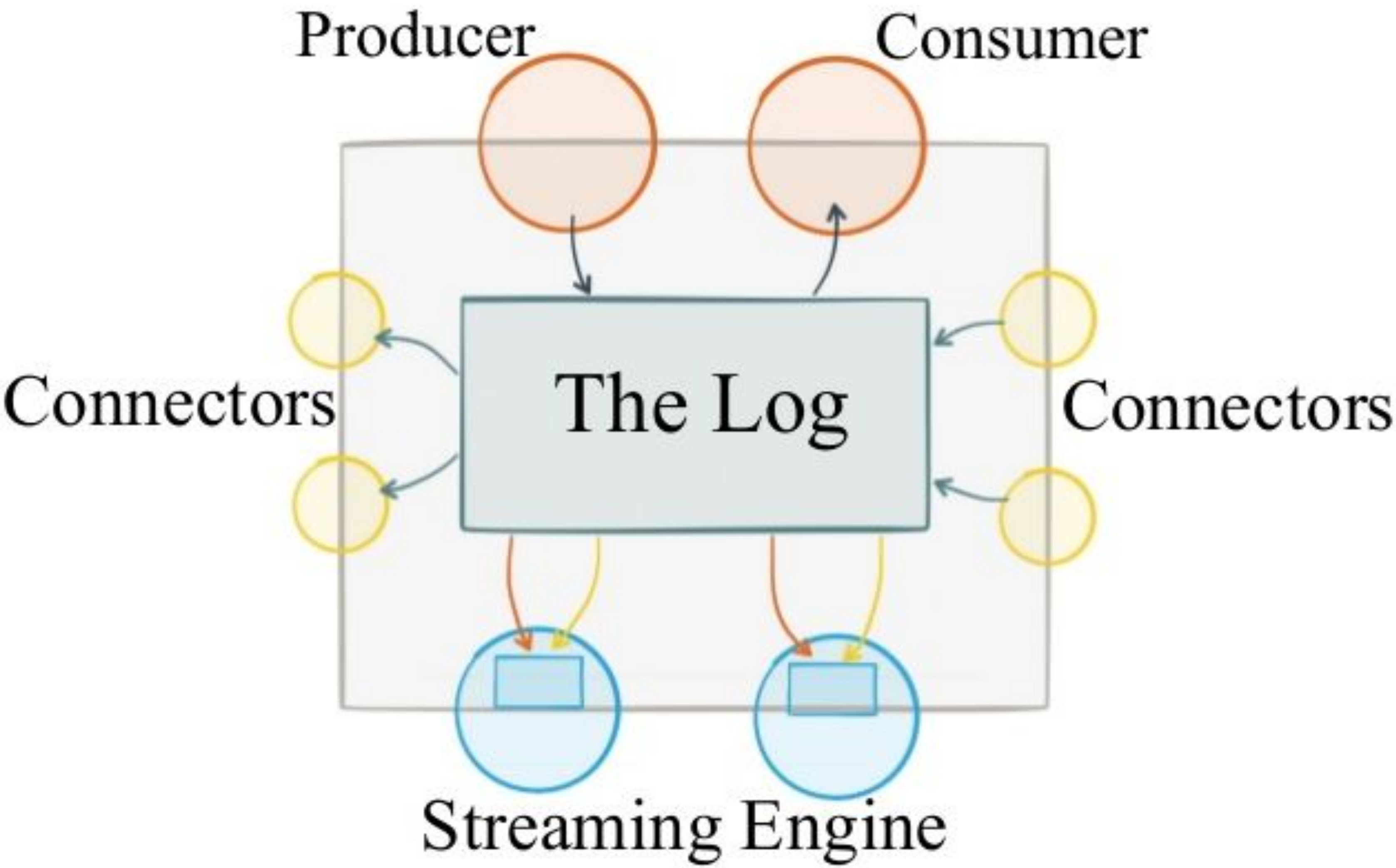


Kafka Cluster
Server 1
Server 2
P0
P3
P1
P2
C1
C2
C3
C4
C5
C6
Consumer Group A
Consumer Group B

Partition replication

- Each partition has one leader server and zero or more follower servers
- The leader handles all reads and writes of Records for partition
- Writes to partition are replicated to followers using a primary backup protocol
- A follower that is in-sync is called an ISR (in-sync replica)
  - If a partition leader fails, one ISR is chosen as new leader

Kafka ecosystem



APIs


Flexibility
Simplicity
Consumer
Producer
subscribe()
poll()
send()
flush()
Kafka Streams
mapValues()
filter()
punctuate()
KSQL
Select…from…
Join…where…
Group by..

KSQL

- KSQL is a streaming SQL engine for Kafka
- KSQL uses Kafka Streams to run the user queries

KSQL data model

- KSQL provide a relational data model with a schema
- Message values in a topic should conform to the schema associated with the topic
- The schema has typed columns
  - Primitive data types supported include BOOLEAN, INTEGER, BIGINT, DOUBLE and VARCHAR along with the complex types of ARRAY, MAP and STRUCT

KSQL data model (cont.)

- KSQL can map a topic to a stream or table
- Topic as stream
  - Consider the messages as independent and unbounded sequence of structured values, we interpret the topic as a stream
  - Messages have no relation with each other and will be processed independently.
- Topic as table
  - Consider the messages as an evolving set of structured values where a new message either updates the previous structured values in the set with the same key, or adds a new structured values when there is no structured values with the same key
  - A table is a state-full entity since we need to keep track of the latest values for each key

Query language

- Create a **stream** from a topic

```
CREATE STREAM pageviews (viewtime BIGINT,
 userid VARCHAR, pageid VARCHAR) WITH
 (KAFKA_TOPIC='pageviews_topic',
 VALUE_FORMAT='JSON');
```

Query language (cont.)

- Create a **table** from a topic

```
CREATE TABLE users (
    registertime BIGINT,
    gender VARCHAR,
    regionid VARCHAR,
    userid VARCHAR,
    address STRUCT<street VARCHAR, zip INTEGER>
) WITH (
      KAFKA_TOPIC='user_topic',
      VALUE_FORMAT='JSON',
      KEY='userid'
  );
```

Query language (cont.)

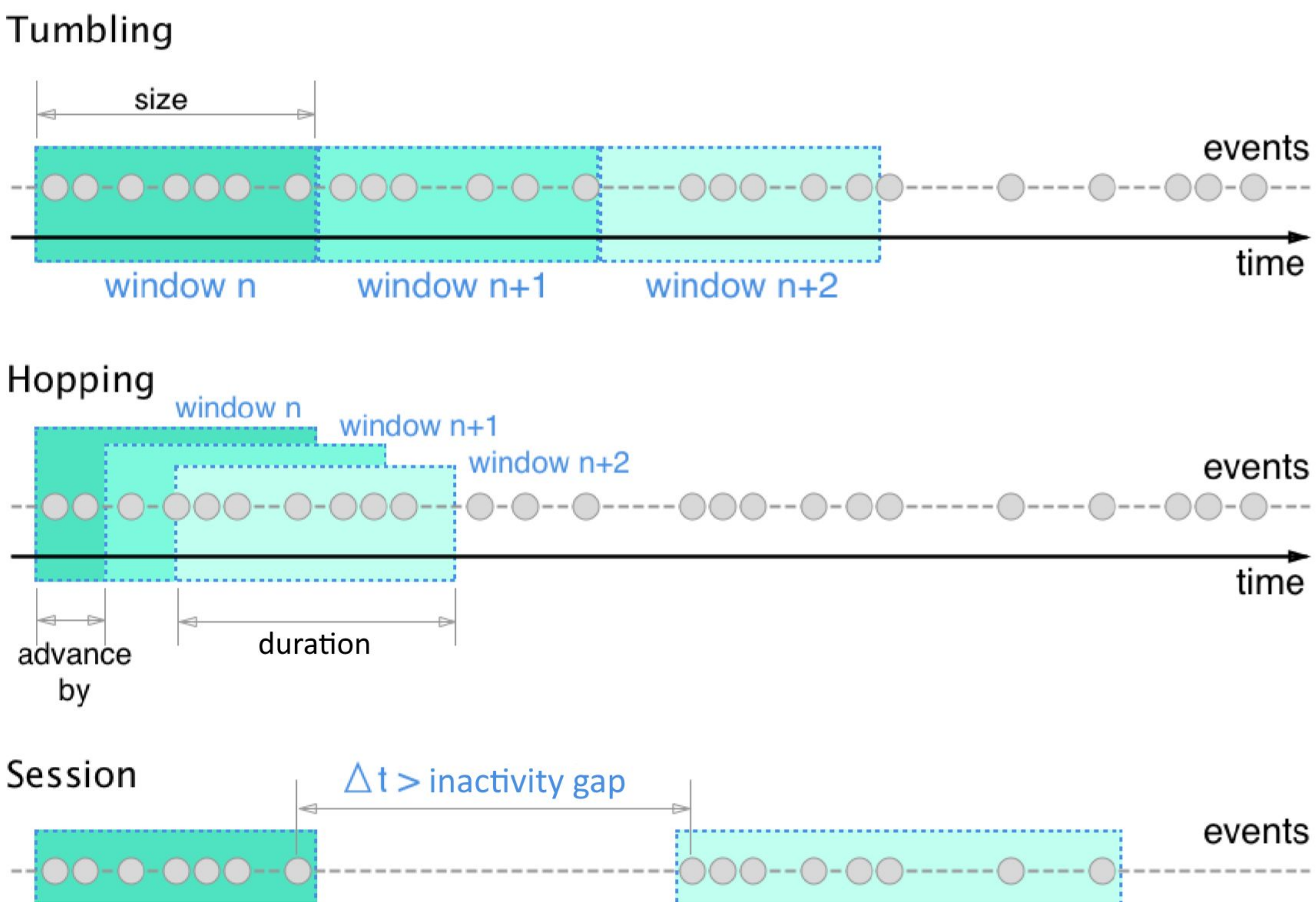
- Continuous queries expressed as the creation of new streams or tables

```
CREATE STREAM enrichedpageviews AS
 SELECT * FROM pageviews LEFT JOIN
 users ON pageviews.userid = users.userid
 WHERE regionid = 'region 10';
```

KSQL support for windows

- Records can be grouped in windows. Currently, KSQL supports three types of windows:
  - **Tumbling window** which are time-based, fixed-sized, non-overlapping and gap-less windows
  - **Hopping window** which are time-based, fixed-sized and overlapping windows
  - **Session window** which are session-based, dynamically-sized, non-overlapping and data-driven windows

KSQL Windows



https://docs.ksqldb.io/en/latest/img/ksql-window-aggregation.png

Query language (cont.)

- Continuous queries expressed as the creation of new streams or tables

```
CREATE TABLE userviewcount AS
 SELECT userid, count(*)
 FROM pageviews
 WINDOW TUMBLING (SIZE 1 HOUR)
 GROUP BY userid;
```

Query processing

- A query is processed and transformed into code that uses the Kafka Stream API
- This process includes query plan optimization

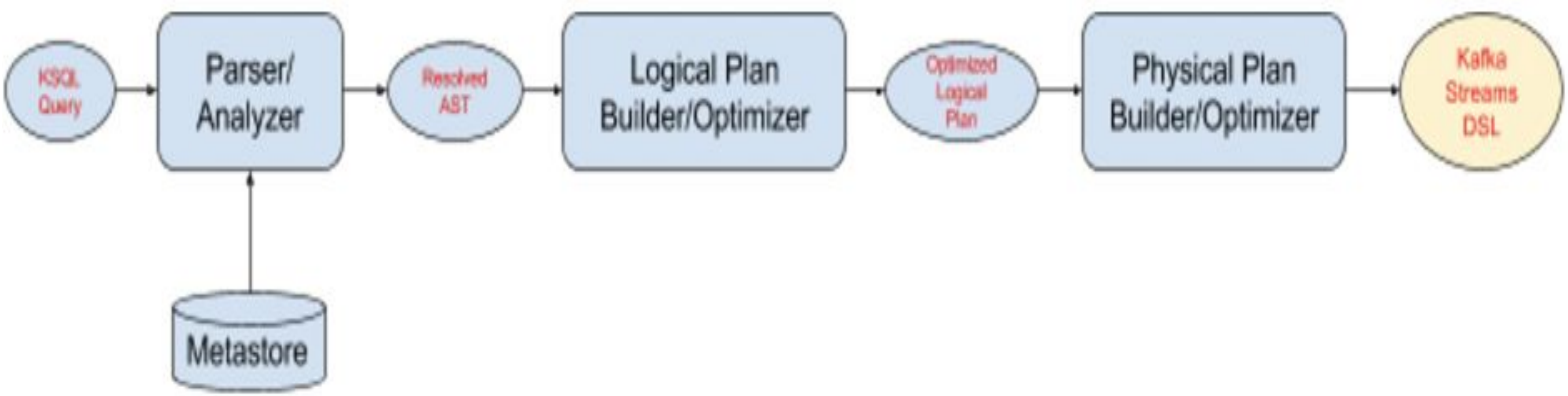


Table of Contents

- Stream processing ecosystem
  - Apache Flume
  - Apache Kafka
  - **AWS Stream Processing Ecosystem**

Streaming Data Scenarios

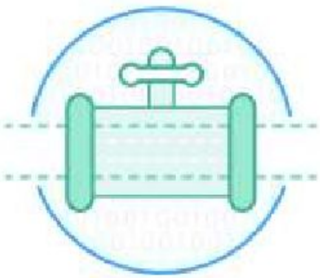
Scenarios/ Verticals	Accelerated Ingest-Transform-Load	Continuous Metrics Generation	Machine Learning and Actionable Insights
Digital Ad Tech/Marketing	Publisher, bidder data aggregation	Advertising metrics like coverage, yield, and conversion	User engagement with ads, optimized bid/buy engines
IoT	Sensor, device telemetry data ingestion	Operational metrics and dashboards	Device operational intelligence and alerts
Gaming	Online data aggregation, e.g., top 10 players	Massively multiplayer online game (MMOG) live dashboard	Leader board generation, player-skill match
Consumer Online	Clickstream analytics	Metrics like impressions and page views	Recommendation engines, proactive care
Operation Security	DevOps tools, Ingesting VPCFlowLogs	Subscribe to CloudwatchLogs and analyze logs in Real-Time	Anomaly Detection

Amazon Kinesis goals

Amazon (AWS) cloud-based **product** for processing big data in ***real-time***:

- Easy to provision, deploy, and manage
- Elastically scalable
- Real-time latencies
- Pay as you go, no up-front costs
- Right services for your specific use cases

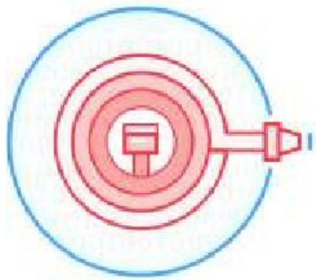
Amazon Kinesis Solutions



Amazon Kinesis Streams

For Technical Developers

Build your own custom applications that process or analyze streaming data



Amazon Kinesis Firehose

For all developers, data scientists

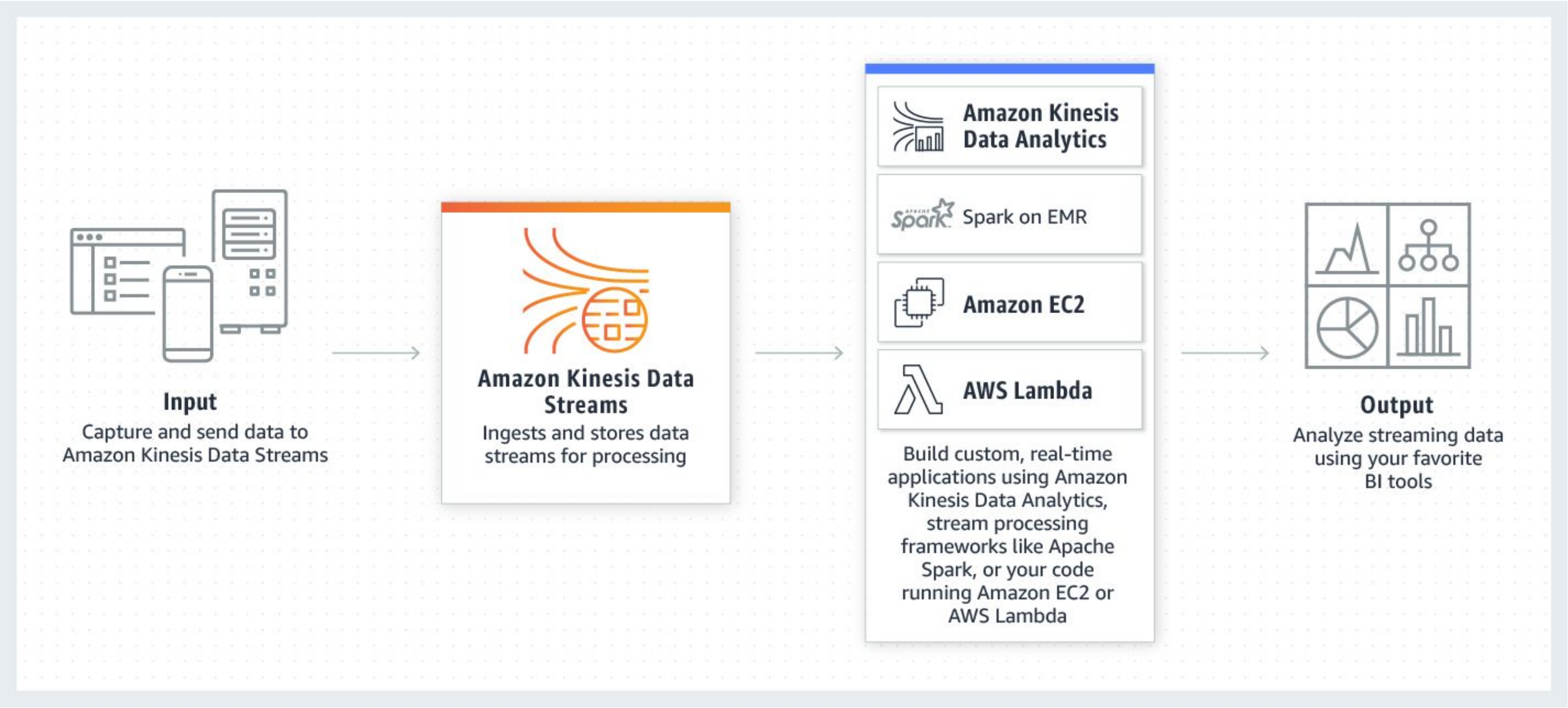
Easily load massive volumes of streaming data into S3,Amazon Redshift and Amazon Elasticsearch

Amazon Kinesis Analytics

For all developers, data scientists

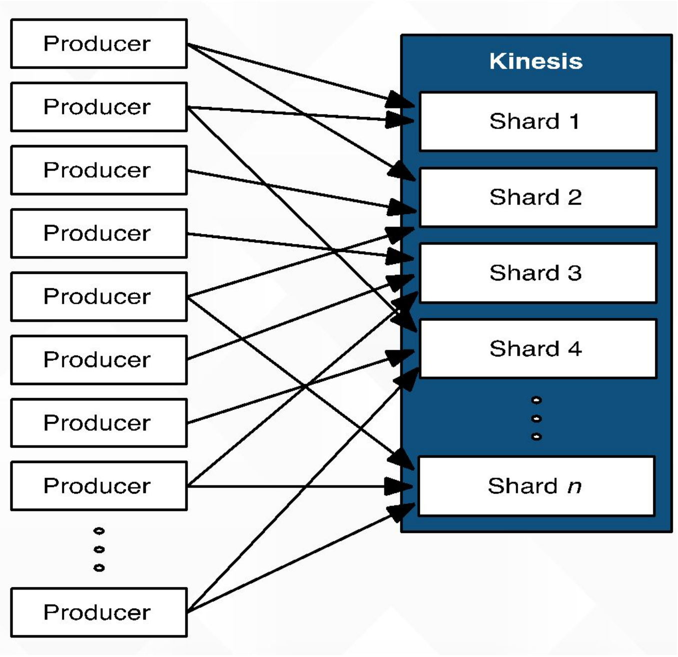
Easily analyze data streams using standard SQL queries

Amazon Kinesis Streams



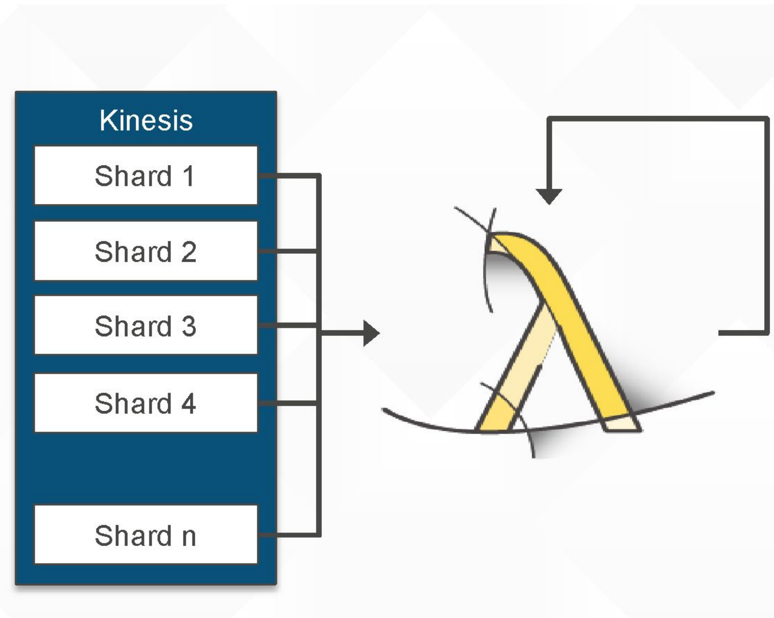
Amazon Kinesis Streams

- Amazon Kinesis Data Streams (KDS) is a massively scalable and durable real-time data streaming service.
- Streams are made of Shards
  - A Partition Key is used to distribute the PUTs across Shards
  - A unique Sequence # is returned to the Producer for each Event



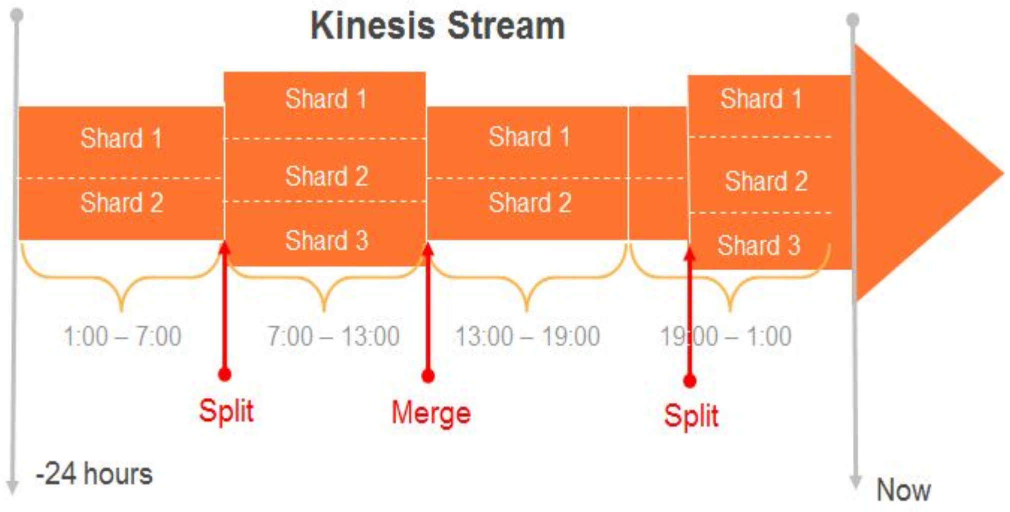
Process with Lambda

- Stateless JavaScript & Java functions run against an Event Stream
- Functions automatically invoked against a Shard
- Access to underlying filesystem for read/write
- Call other Lambda Functions



Amazon Kinesis Streams

- Elastic operation
  - Scale Kinesis streams by splitting or merging Shards



Amazon Kinesis Firehose

- Amazon Kinesis Data Firehose is used to reliably **ingest** streaming data into data stores and analytics tools.
- It can capture, transform, and load streaming data into Amazon S3, Amazon Redshift, Amazon Elasticsearch Service, and Splunk, enabling near real-time analytics with existing business intelligence tools and dashboards.

Kinesis Streams vs. Kinesis Firehose

- Amazon Kinesis Streams is for use cases that require **custom processing**, per incoming record, with sub-1 second processing latency, and a choice of stream processing frameworks.

- Amazon Kinesis Firehose is for use cases that require zero administration, ability to use **existing analytics tools** based on Amazon S3, Amazon Redshift and Amazon Elasticsearch, and a data latency of 60 seconds or higher.

Amazon Kinesis Analytics

- Apply SQL on streams: Easily connect to a Kinesis Stream or Firehose Delivery Stream and apply SQL skills.
- Build real-time applications: Perform continual processing on streaming big data with sub-second processing latencies.
- Easy Scalability : Elastically scales to match data throughput.

Bibliography

- https://flume.apache.org/releases/content/1.9.0/FlumeUserGuide.html#
  - Architecture
- https://kafka.apache.org/documentation/#design
- https://docs.confluent.io/current/ksql/docs/index.html
  - (These references are too detailed for preparing for tests)

Acknowledgments

- Some images from:
- Kai Wähner, KSQL – An Open Source Streaming Engine for Apache Kafka
- Sites of the systems presented.