Introduction to Data Analytics

Kafka

András Varga
IBM Consulting

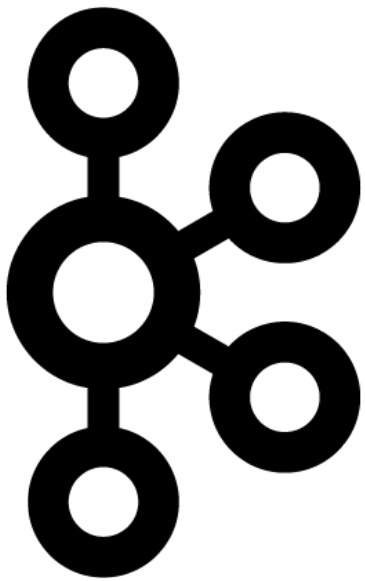
Bratislava, 2022

Contents

Introduction

Definition

Apache Kafka is a distributed streaming platform.



Capabilities:

- Publish and subscribe to streams of records (similar to message queues)
- Store streams of records in a fault-tolerant and durable way
- Process streams of events in real-time as they occur

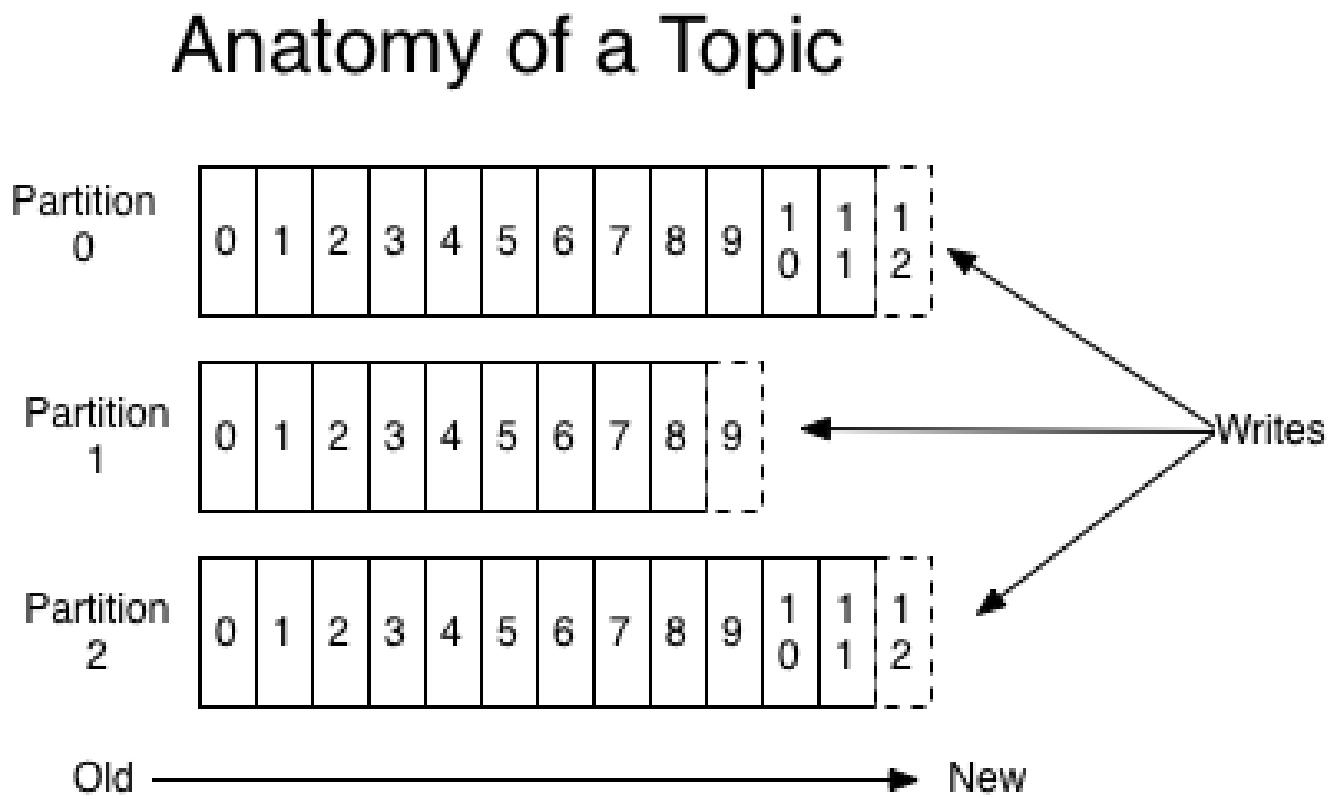
Kafka Basics

- Deployed over a cluster
- Kafka servers are called *Brokers*
- Kafka is streaming records in categories called *topics*
- *Producers* are creating new records
- Clients reading records are called *Consumers*
- Both are communicating using TCP/IP
- Each *message* (*record* or *event*) has a header (with a timestamp), a key and a value

Kafka Topics I

- The "feed" into which records are published
- It is partitioned into *buckets*
- Events are consumed by possibly several consumers (multi-subscribers)
- Kafka maintains a partitioned log for each topic
- It is a structured commit log
- Each partition is an ordered, immutable sequence of records
- *Offset* - An unique ID of a record within a partition

Kafka Topics II



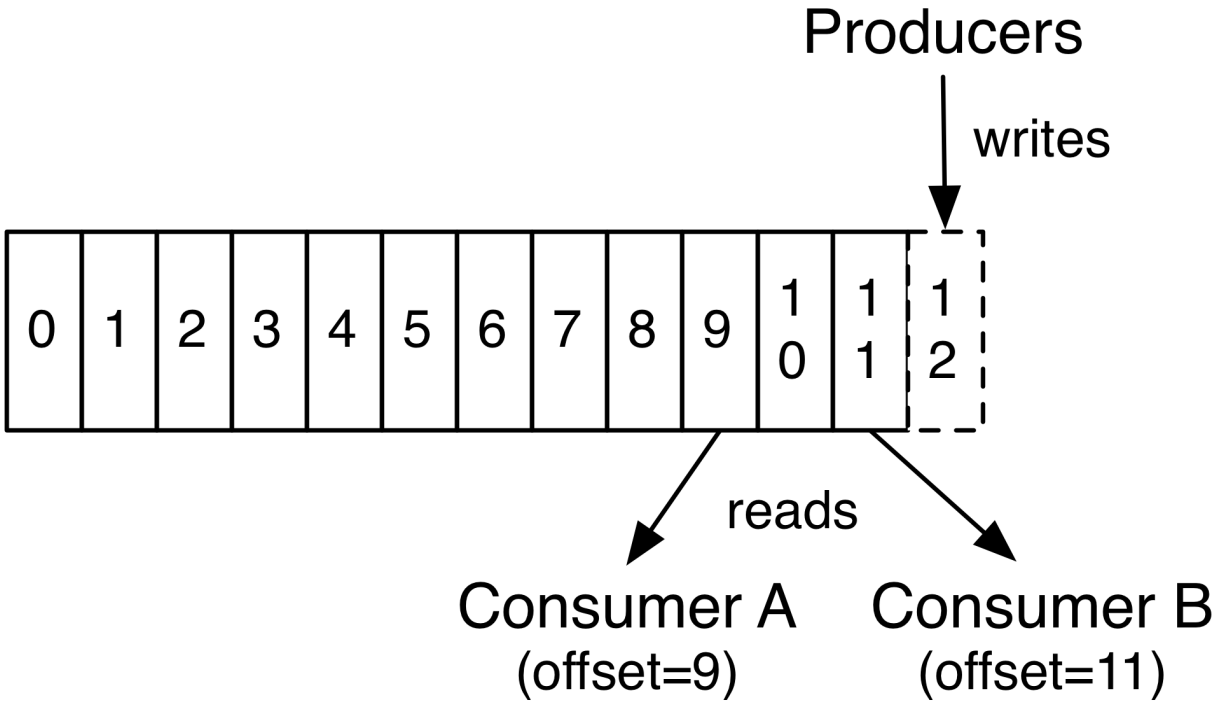
Kafka Topics III

- Kafka stores records for a configurable retention period
- Records are stored regardless if they were consumed or not
- Kafka stores the position of each consumer on the log
- This position (offset) is controlled by the consumer - can make "jumps"
- Nothing else is stored about the consumers, so consumers are cheap

Kafka Topics IV

- The partitions of the log are distributed among the brokers
- Each partition is replicated across several brokers
- Each partition has a "leader" server - handles reads and writes
- All other servers are handling the same partitions are followers - simply replicating the leader
- If the leader fails one of the followers become a new leader
- The *metadata* of each topic is stored in ZooKeeper or KRaft

Kafka Topic Example



Producers

- Publish records into the topics of their choice
- Producer chooses which partition the message is being sent to (default is random)
- Communicates directly with the (topic) partition leader

Consumers

- Consumers are subscribed to topics they consume
- Consumers can be assigned to groups - *consumer group*
- Each new record in the topic is delivered to one consumer from each subscribed consumer group
- Kafka provides a total order of records, but only within a partition

Guarantees

- Messages sent by a producer to a particular topic partition will be appended in the order they are sent
- A consumer instance sees records in the order they are stored in the log
- For a topic with replication factor N, we will tolerate up to $N - 1$ server failures

Messaging System

- Usual approaches:
  - Queue
  - Publish-subscribe
- Kafka combines both approaches
- Traditionally queues can not maintain order with multiple consumers
- Using consumer groups it provides "multicasting" instead of traditional broadcasting of publish-subscriber design
- Is Kafka a message system?
  - Not in the purest sense
  - But can act as one

Additional Context

Storage System

- Fault tolerant distributed storage system
- Producer received acknowledgment once the data is fully replicated and guaranteed to persist
- Is Kafka a storage system?
  - It has to be
  - To store the messages during the retention period

Stream Processing

- Producer - Consumer approach is valid
- Kafka provides Streams API in addition

Kafka Use Cases

- Kafka can be used for several use cases
- Most of them directly follow from the nature of Kafka
- Other use cases are less trivial

Use Case Examples

- Messaging
- Website Activity Tracking
  - Fun fact: this was one of the original use cases
- Log Aggregation
- Stream Processing
- External Commit Log

Motivation

- To create a design capable handling of data from a large company
- Needs to deal with huge backlogs ...
- ... and high volume (big data)
- As a result some parts of the design are more similar to databases than streaming/messaging platforms

Filesystem

- Kafka relies on filesystem
- Design must find a way around "disk are slow" perception
- OS usually caches huge parts of disk into memory
- It also runs in JVM, which handles data "lazily" (garbage collector)
- Actually Kafka tries to leverage these facts
- Writing into file means writing into cache in most cases
- And this cache can survive service restarts

I/O Operations

- Too small I/O can cause problems
- To resolve this the protocol is set up to enable message grouping
- This enables the protocol to lower the messaging overhead
- This way the server appends a bigger linear chunk of messages into the log
- And consumers can also read linear chunks from the logs, lowering the amount of seeks

Compression

- Network bandwidth can cause problems as well
- Especially when dealing width huge data in a widespread network
- Users can compress their data
  - Compressing messages individually leads to bad ratios
  - It is more efficient to compress multiple messages together
- Kafka supports a batch format enabling compression of multiple messages at once
- These compressed batches are written into the logs compressed and decompression is done by the consumers
- Supports: GZIP, Snappy, LZ4 and ZStandard

The Producer

- Sends data directly to the broker - leader of the partition of the topic
- Producer can load balance - by selecting a partition in the topic, such as random, or according to a hash
- Can store all keys in one partition to achieve locality
- Tries to accumulate requests in memory and sends them as a batch in a single request
- This can be limited by amount of data or duration

The Consumer

- Sends "fetch" request to the broker - partition leader
- The consumer can set the offset of the log
- A chunk of the log is sent to the consumer starting from that partition
- So consumer can re-consume the same data several times
- This approach means that the data is *pulled* from the broker by the consumer
- The current offset of the consumer is stored by the broker

Message Availability

- Once a message is committed into a log it is available
- This message is not lost until at least one broker that replicates the partition is "alive"
- Publisher obtains a response once the message is committed: (publisher can choose)
  - Once message is committed in the leader
  - Once the message is fully replicated
  - Or work asynchronously

Failure During Publishing

- Upon failure publisher can not determine if the message is committed or not
  - The return message might be lost
  - The commit was not processed, etc.
- Publisher must re-send the message to be sure
- Which could lead to duplicates in the log
- Kafka also provides an idempotent delivery option:
  - Each producer is assigned an ID
  - Each message is assigned a sequence number (by the publisher)
  - Kafka identifies duplicates using the sequence number

Consumer Failures

The consumer can do the following:

- Read message, save new position, process message
  - Upon failure this can lead to message loss
  - "at-most-once" semantics
- Read message, process message, save new position
  - If consumer crashes between processing and storing the new position some messages might be read again
  - "at-least-once" semantics
- Kafka Streams provide "exactly-one" delivery semantics as well

Kafka Cluster

- Kafka replicates each topic's partition across a number of servers
- The amount of servers is *configurable* for each topic separately
- Each partition has an associated leader
- Each operation goes into the leader
- Followers keep an up-to-date replica of the log of the leader
- Followers act as consumers to replicate the messages from the leader

Status of Kafka Nodes

Kafka node is "in sync", when:

- Maintains a connection to ZooKeeper (Kafka uses ZK to track the cluster)
- Replicates the writes on the leader
- Is not "too-far" behind

A leader keeps a list of each "in sync" follower

A message is committed, when all "in sync" replicas have applied it

Replicated Logs

- Basic distributed structure
- Main goal: Coming into consensus on the order of a series of values
- Naive resolution: Leader determines the order
- Problems arise with failures
- Followers must replicate and leader waits till their acknowledgment
- Leader can wait till a majority of cluster reacts $\Longrightarrow$ quorum
- This is not how Kafka works

Leaders in Kafka

- Kafka is not using a majority vote quorum
- Set of in sync replicas is maintained (ISR)
- Only members of ISR are eligible to become a leader
- Kafka uses ZooKeeper for leader election
- Followers must catch up before getting back to ISR
- When all ISR replicas become unavailable no guarantees hold
- Two options to resolve this issue:
  - Select a node not within the ISR as leader
  - Wait till the first node from ISR to become online - to become a leader (Default option)

Influencing Availability

Following *topic* related options are available:

- Disable unclean leader election (with empty ISR)
- Specify minimal ISR size

Quotas

Kafka accepts the following quotas:

- Network bandwidth quotas
- Request rate quotas define CPU utilization thresholds as a percentage of network

Message Format

- Message parts:
  - variable-length header
  - variable length opaque key byte array
  - variable length opaque value byte array
- Messages are stored in record batches

Record Batch

- A record batch contains a header and a set of records/messages
- Contains the
  - Partition Leader epoch
  - Compression settings
  - Several timestamps
  - ...
- For more details see the official documentation

Record

On disk:

- length
- timestamps
- keyLength
- key
- ValueLength
- value
- headers
- ...

Installation

- Download the source
- Uncompress it
- Configure Zookeeper or KRaft connection
- Ready to go

Broker Configuration

- broker.id
- log.dirs
- zookeeper.connect
- compression.type
- log.retention.hours/minutes/...
- SSL configuration

Topic-Level Configs

Can be configured globally or per-topic, if no per-topic configuration is set a general global configuration is used as default

- compression.type
- retention.ms
- max.message.bytes

Adding New Machines

- Adding new machines is installing the machines, adding them broker ID and starting the servers
- New servers will not have any partitions
- Partitions must be re-arranged

Basic Java APIs

- Producer
- Consumer
- Streams
- Connect
- AdminClient
- ProducerRecord
- ConsumerRecords

Producer

- KafkaProducer Java class
- Main functions:
  - send() - asynchronous
  - send() - waiting for callback
- Sends classes ProducerRecord<K,V>

ProducerRecord<K,V>

- Topic
- Partition
- Key
- Value

Consumer API

- KafkaConsumer Java class
- Main functions:
  - subscribe() - to a list of topics
  - assign() - assign partitions to the consumer
  - close()
  - position() - get offset of next record
  - seek() - sets offset
  - poll() - read the messages
- Obtains ConsumerRecords<K,V>

ConsumerRecords<K,V>

- count() - number of records
- Iterator over ConsumerRecord<K,V>
- ConsumerRecord<K,V>
  - key()
  - value()
  - topic()

Security

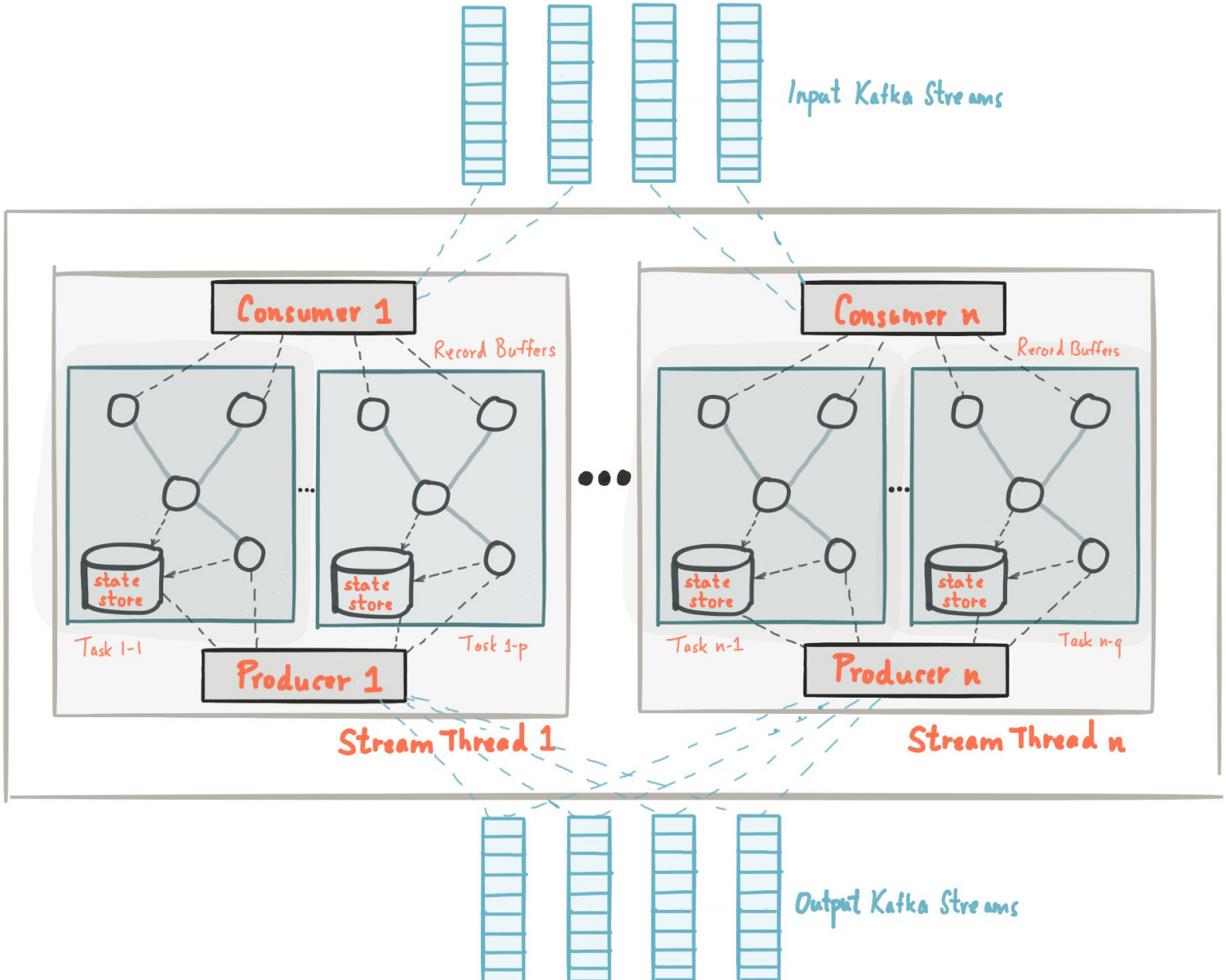
Kafka is capable of:

- SSL
- ACL
- ZooKeeper Authentication

Kafka Streaming

- Client library for building applications and microservices
- No external dependencies
- Exactly-once processing semantics
- One-record-at-a-time processing to achieve millisecond processing latency
- The streaming library runs directly on the application side, not Kafka side

Kafka Streaming Architecture



Kafka Streaming Architecture

- Kafka Streaming is utilizing the existing Consumers and Producers
- Streaming partitions data utilizing topic partitions
- Data records map to messages
- Keys determine the partitions, it can not be changed
- It helps scaling, the load from additional applications is distributed amongst the partitions using keys

References I

 "Apache™ Kafka Official Web Page."

 "Big Data University."