**Apache Kafka** is a publish-subscribe based durable messaging system developed by Linkedin, let you send messages between processes, applications, and servers.

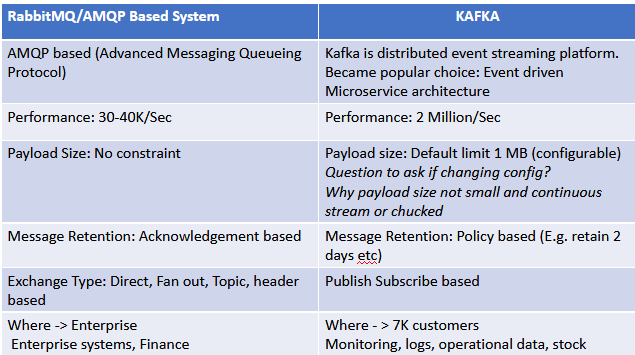
**Why use Kafka?**

* **Multiple producers and consumers at any given time** without interfering with each other. This is in contrast to many queuing system where once a message is consumed by one client
* **Disk-Based retention**:
  + Consumers do not always need to work in real time. Messages are commited to disk and stay for some periods of time.
  + There is no danger of losing data.
* **Fast**: Kafka is a good solution for applications that require a high througput, low latency messaging solution. Kafka can write up to 2 million requests per second
* **Scalable**:
  + Expansions can be performed while the cluster is online, with no impact on the availability of the system as a whole.
* **High Performance**: Excellent performance under high load.

**Why is Kafka fast?**

* **Zero Copy**: Basically Kafka calls the OS kernel directly rather than at the application layer to move data fast.
* **Batch data in chunks**: Kafka is all about batching the data into chunks. This minimises cross machine latency with all the buffering/copying that accompanies this.
* **Avoids Random Disk Access**: Kafka is designed to access the disk in sequential manner. This enables it to get similar speeds from a physical disk compared with memory.
* **Can scale Horizontally**: The ability to have thousands of partitions for a single topic spread among thousands of machines means Kafka can handle huge loads.

**Compared to other message queue systems:**



Here are some reasons why you might prefer RabbitMQ over Kafka in enterprise and finance systems:

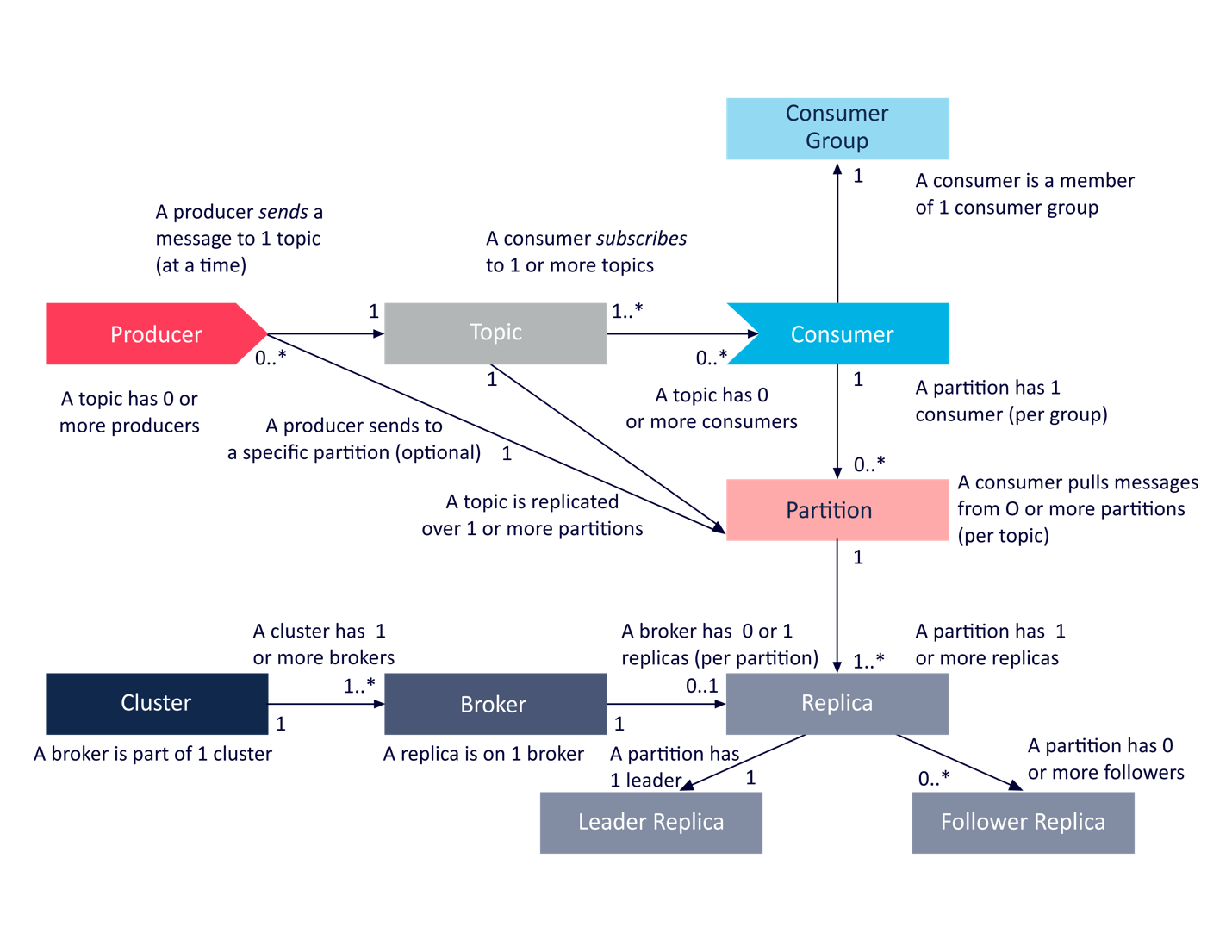
1. **Simplicity and Ease of Use**: RabbitMQ is known for its simplicity and ease of use. It provides a straightforward message queueing system that is well-suited for scenarios where you need to publish and consume messages with reliability but don't require the complex event streaming capabilities offered by Kafka.
2. **Traditional Message Queues**: RabbitMQ is a traditional message broker that is highly suitable for point-to-point communication, request-reply patterns, and message routing. In many finance systems, such traditional messaging patterns are prevalent.
3. **Advanced Routing**: RabbitMQ offers advanced message routing capabilities through exchanges and queues, making it well-suited for complex routing and filtering of messages, which can be important in finance systems.
4. **Message Acknowledgment**: RabbitMQ provides fine-grained control over message acknowledgment, making it possible to implement strict message processing guarantees.
5. **Ease of Integration**: RabbitMQ is easier to integrate with a wider range of programming languages and frameworks, making it a good choice for systems with diverse technology stacks.
6. **Well-Established in Finance**: RabbitMQ is widely used in the finance industry and has been proven to be effective for a variety of use cases in this domain.

On the other hand, you might prefer Kafka in certain scenarios, including:

1. **Log-Based Data Integration**: Kafka is optimized for log-based data integration and stream processing. It can handle high-throughput, real-time data streaming, making it suitable for financial trading platforms, real-time analytics, and event sourcing.
2. **Event Sourcing**: Kafka's log-based architecture is well-suited for implementing event sourcing patterns, which can be valuable in financial systems for tracking and auditing events over time.
3. **Scalability and Durability**: Kafka is known for its scalability and durability. It can handle large volumes of data, making it a good choice for systems with high throughput requirements.
4. **Stream Processing**: Kafka has built-in support for stream processing, allowing you to process data as it flows through the system, which is important for real-time analytics and event-driven architectures.
5. **Multi-Consumer Groups**: Kafka's support for multi-consumer groups makes it suitable for scenarios where multiple consumers need to read the same data independently.

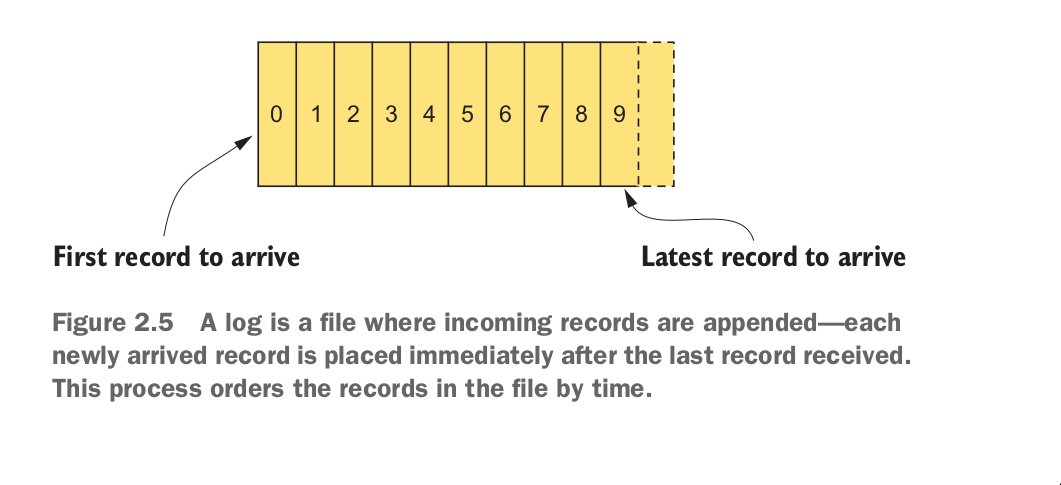
**Kafka architecture**

Kafka is a message broker. A broker is an intermediary that brings together two parties that don't necessarily know each other for a mutually beneficicial exchange or deal.



**Log:**

Is a file that Kafka appends incoming records to. A log is an append-only, totally ordered sequence of records ordered by time



Configuration setting log.dir, specifies where Kafka stores log data on disk.

**Topics**:

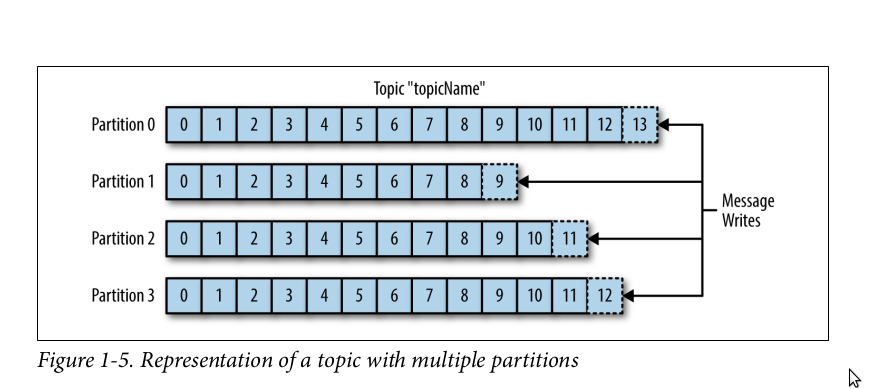
Topics are logs that are seperated by topic name. Thinks topics as labeled logs. The closest analogies for a topic are a database table or a folder in a filesystem

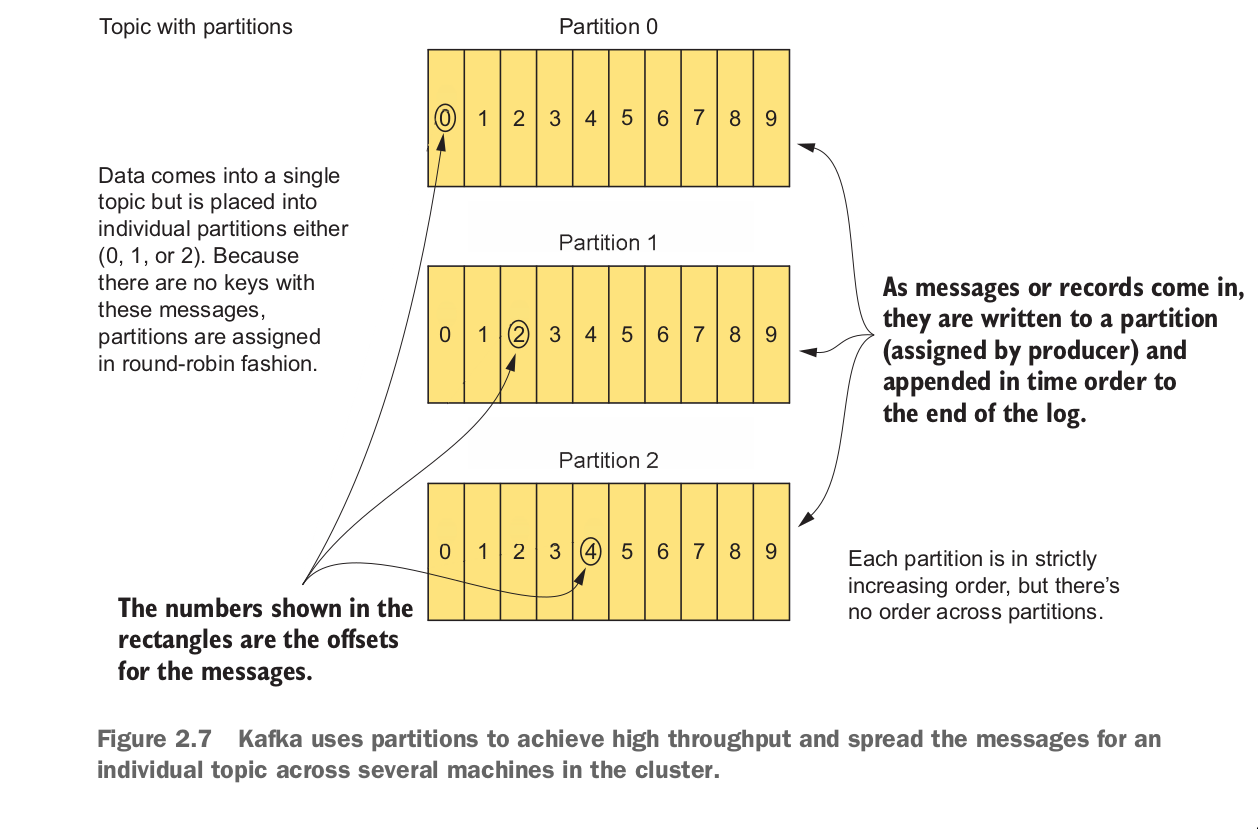
To help manage the load of messages coming into a topic. Kafka use partitions

Topics are broken down into a number of partitions.

**Partitions:**

Partitions are the way that Kafka provides redundancy and scalability. Each partition can be hosted on a different server, which means that a single topic can be scaled horizontally across multiple servers.



* Help increasing throughput
* Allows topic messages to be spread across several machines so that the capacity of a given topic isn't limited to the availble disk space one one server
* 

Difference between Partition and Log?

* Log: physical part of a topic, where a topic is stored on the disk.
* Partition: logical unit used to break down a topic into splits for redundancy and scalability. You can see Log stored on disk. But with Partition, you can't. Partition is handled logically.

Partitions group data by key

When a message is sent to kafka, you can specify a key option for that message.

If the key(key will be explained in the next section) isn't null. Kafka uses the following formula to calculate which partition the message will be sent to:

**Producer -> message(key, value) -> Kafka**

**// Kafka choose a partition by using the formula**

**target\_partition = HashCode.(key) % number of partitions**

Records with the same key will always be sent to the same partition and in order.

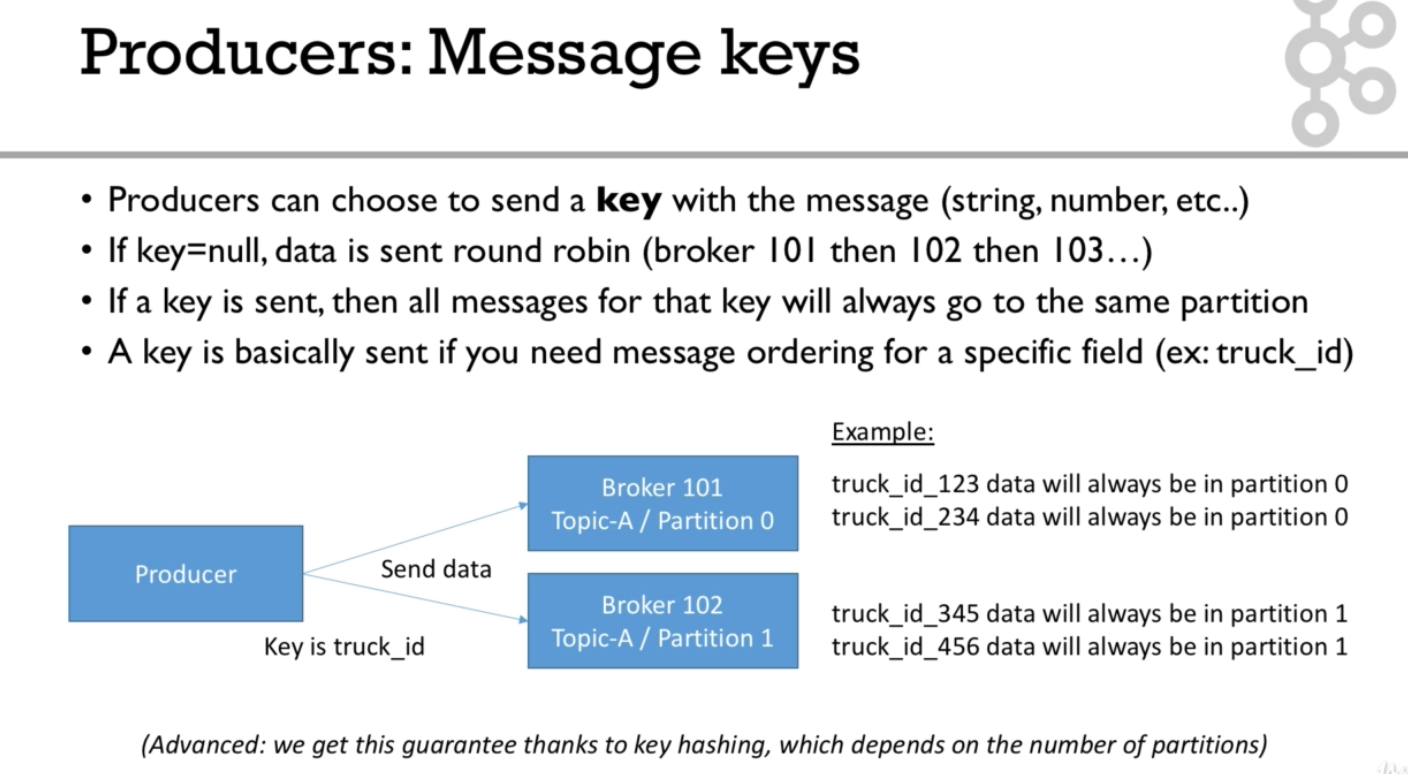
**Important characteristics of Kafka**

* Kafka stores and retrieves message from topic. Doesn't keep any state of producers or consumers
* Messages are written into Kafka in batches. A batch is just a collection of messages, all of which are being produced to the same topic and partition.

**Producers and Consumers:**

**Producer**

Producers create new messages. In other publish/subscribe systems, these may be called publishers or writers. A message will be produced to a specific topic.

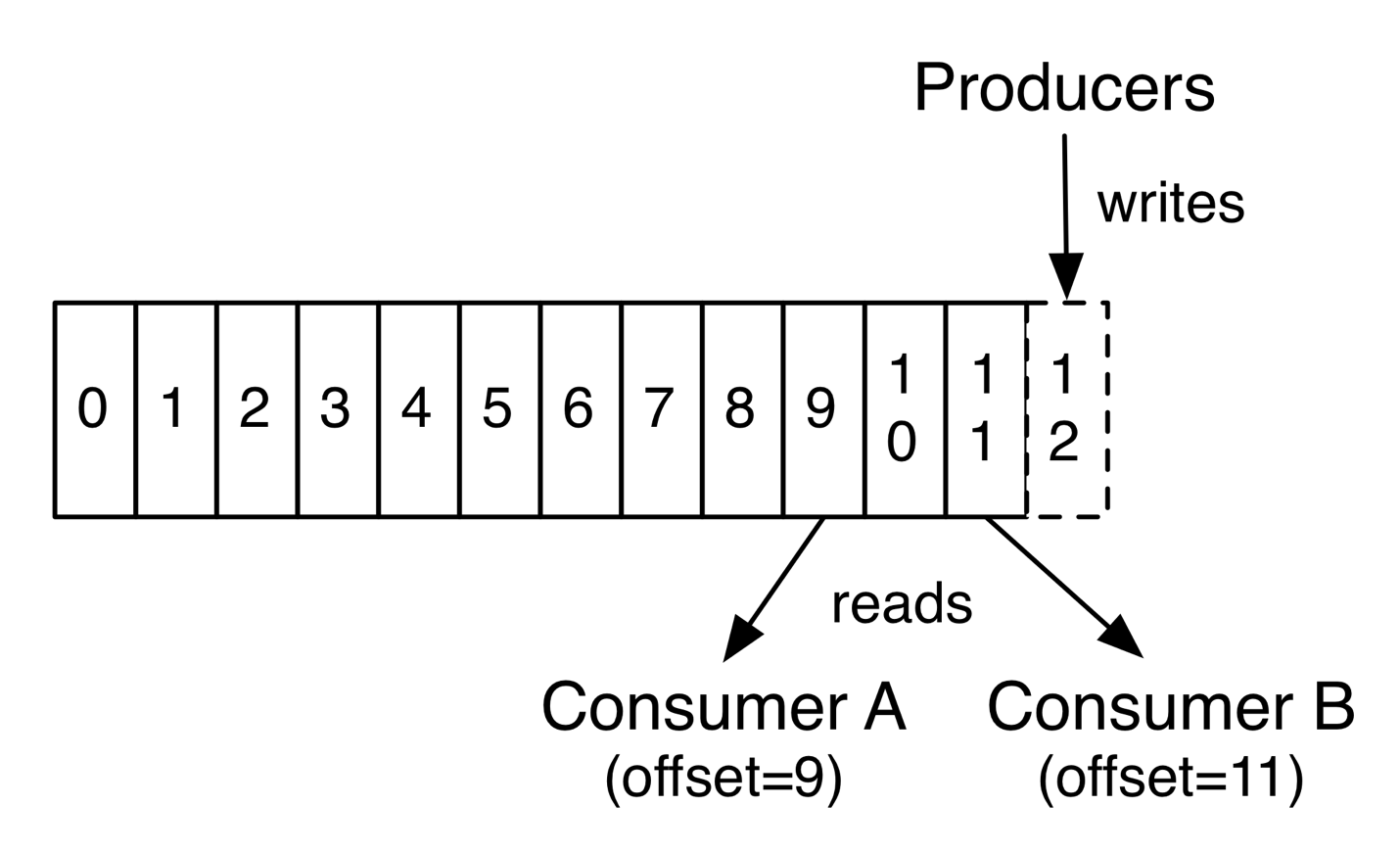
* The producer does not care what partition a specific message is written to and will balance messages over all partitions of a topic evenly
* In some cases, the producer will direct messages to specific partitions using message key. Messages with a specified message key will be ensured to come in the right order in a partition.
* 

**Consumer**

Consumers read messages. In other publish/subscribe systems, these may be called subscribers or readers.

* The consumer subscribes to one or more topics and reads the messages in the order in which they were produced.
* The consumer keeps track of which message it has already consumed by keeping track of the offset of messages.

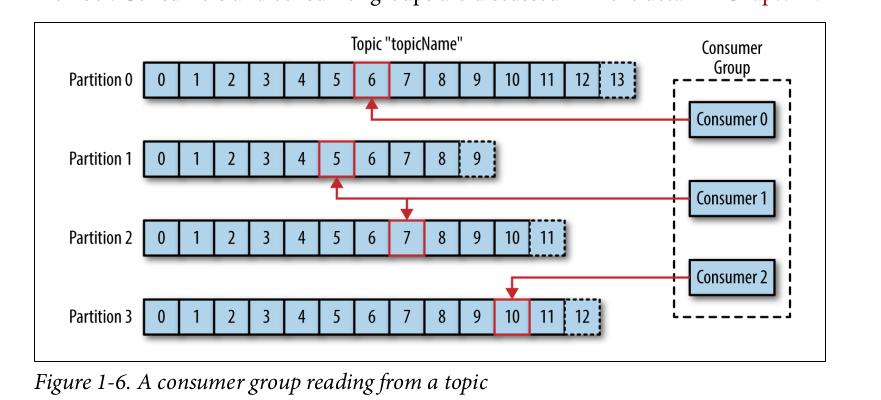
The offset is a simple integer number that is used by Kafka to maintain the current position of a consumer.



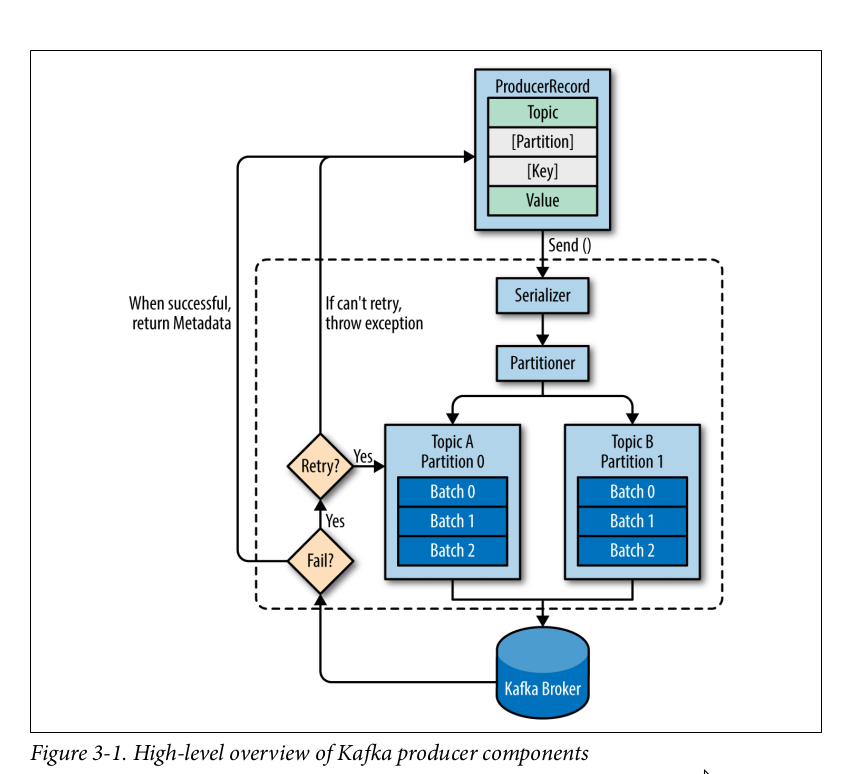
Consumers work as part of a consumer group, which is one or more consumers that work together to consume a topic. Group assures that each partition is only consumed by one member. If a single consumer fails, the remaning members of group will rebalance the partitions being consumed to take over the missing member.

**Consumer group**

Consumers groups used to read and process data in parallel.



**Flow of sending a message:**

* Create a ProducerRecord, which must include the topic we want to send the record to and a value. Optionally, we can also specify a key and/or a partition.
* Then Serialized the key and value objects to ByteArrays so they can be sent over the network
* Data is sent to a partitioner. The partition check if ProducerRecord has a specifed partition option. If yes, it doesn't do anything an reply the partition we specify. If not, the partitioner will choose a partition for us.
* Once a partition is selected, the producer then add the record to a batch of records that will also be sent to the same topic and partition.
* When broker receives the messages, it sends back a response.
  + If the messages were successfully writtent to Kafka, return a RecordMetatData object contains <topic, partition, offset>
  + If failed, the broker will return an error. The producer may retry sending the message a few more times before giving up and returning an error.
* 

**Broker and Clusters:**

**Broker:**

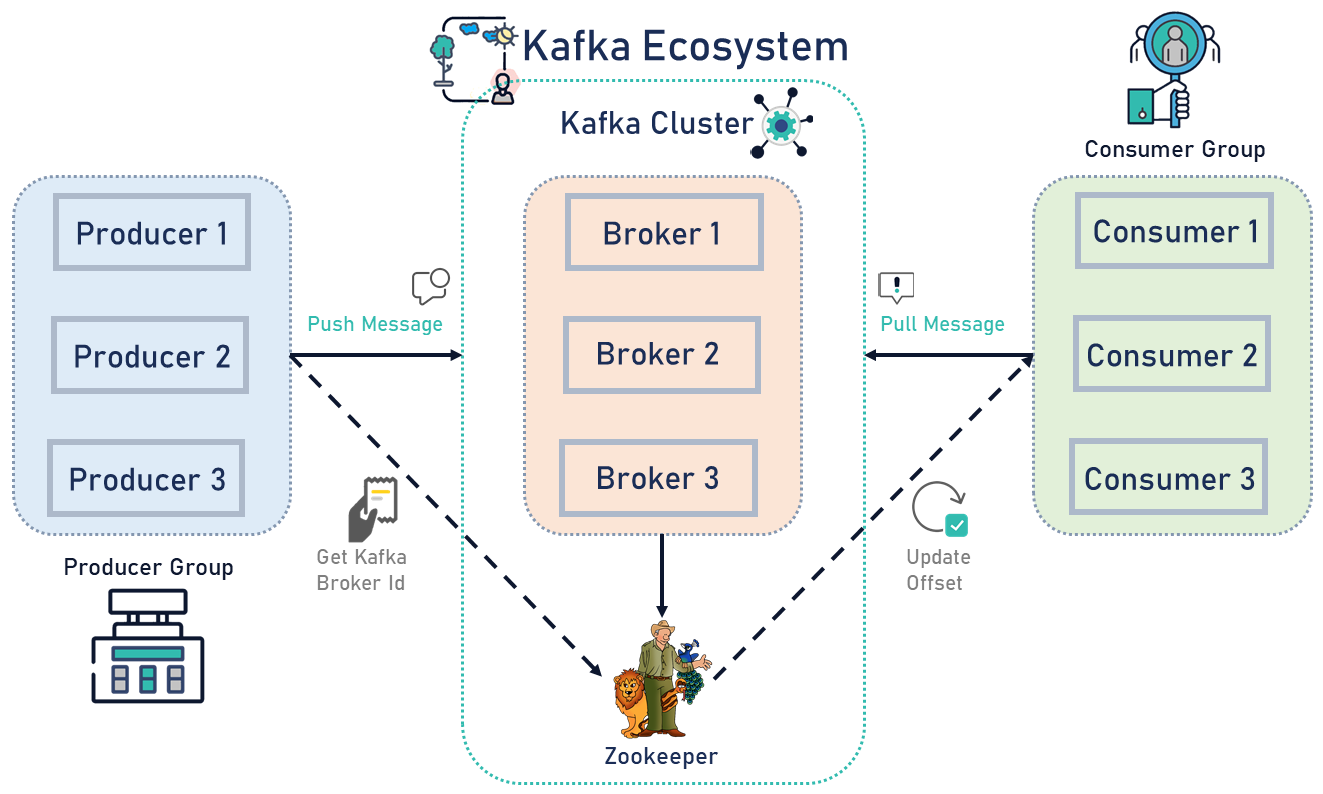
A single Kafka server is called a broker. The broker receives messages from producers, assigns offsets to them and commits the messages to storage on disk.

Brokers are desigined to operate as part of a cluster.

**Cluster membership management with Zookeeper**

Kafka uses Apache Zookeeper to maintain the list of brokers that are currently members of a cluster. ZooKeeper is a consistent file system for configuration information.

It acts as a centralized service and helps to keep track of the Kafka cluster nodes status, Kafka topics, and partitions.

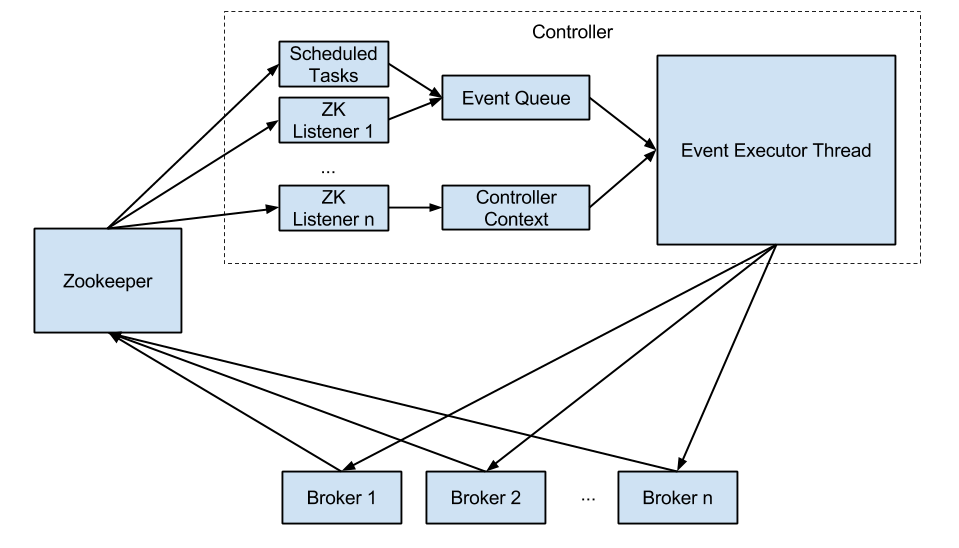


[Kafka’s Shift from ZooKeeper to Kraft | Baeldung](https://www.baeldung.com/kafka-shift-from-zookeeper-to-kraft)

**Cluster controller:**

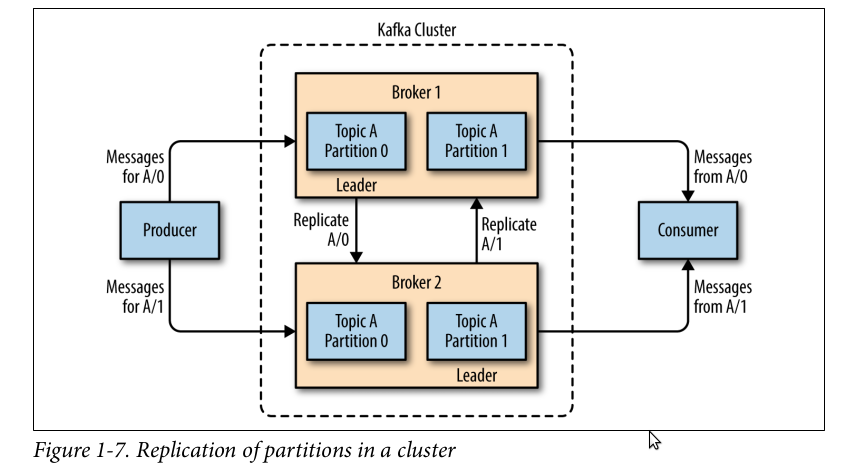
In a cluster, one broker will also function as the cluster controller

A cluster controller is one of the kafka brokers that in addition to the usual broker functionality:

* administrative operations: assigning partitions to brokers and monitoring for broker failures
* electing partition leaders(explained in the next section)
* Cluster only have one controller at a time
* The first broker that starts in the cluster becomes the controller.
* 

**Replica:**

Replication is at the heart of Kafka's architecture. It guarantees availability and durability when individual nodes inevitably fail.



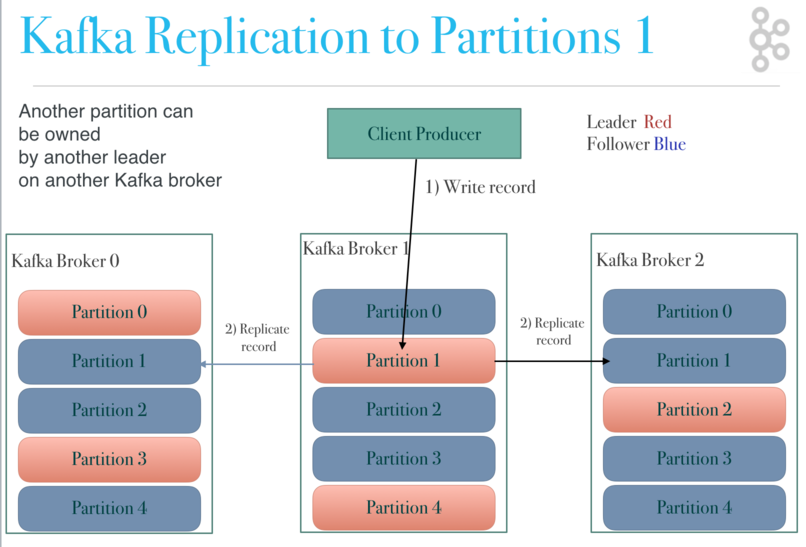
Each broker holds a number of partitions and each of these partitions can be either a leader or a replica for a topic

There are two types of replica:

#### **Leader replica**

* Each partition has a single replica designated as the leader.
* All produce and consume requests go through the leader, in order to guarantee consistency.
* Each partition also have a prefered leader, the replica that was the leader when the topic is originally created.

#### **Follower replica**

* All replicas for a partition that are not leaders are called followers
* Followers don't serve client requests
* Only replicate messages from the leader and stay up-to-date with the most recent message the leader has
* When a leader crashes, one of follower replica will be promoted to become the leader
* A Follower replica that catch up with the most recent messages of the leader are callled In-Sync replica
* Only in-sync replicas are eligible to be elected as partition leader in case the existing leader fail
* 

When a controller notices that a broker left the cluster. All the partitions that had a leader on that broker will need a new leader, so the controller will choose a new leader for all of these partitions

## Configurations

### Hardware selection

#### Disk Throughput

* Faster disk writes will equal lower produce latency
* SSDs have drastically lower seek and access times and will provide the best performance

#### Disk capacity

* If the broker is expected to receive 1 TB of traffic each day, with 7 days of retention, then the broker will need a minimum of 7 TB of usable storage for log segment

#### Memory

Having more memory available to the system for page cache will improve the performance of consumer clients'

### Partitions count, replication factor

The two most important parameters when creating a topic: Patition and replication factor They impact performance and durability of the system overall

#### Patitions

* Each partition can handle a throughput of a few MB/s
* More patitions implies:
  + Better parallelism, better throughput
  + Ability to run more consumers in a group to scale
  + Ability to leverage more brokers if you have a large cluster
  + BUT more elections to perform for Zookeeper
  + BUT more files opened on Kafka

Guidelines:

* Patitions per topic = MILLION DOLAR QUESTION
  + Small cluster(<6 brokers>): #partitions per topic = 2 x number of brokers
  + Big cluster(>12 brokers): 1 x # of brokers

#### Replication: should be at least 2, usually 3, maximum 4

The higher the replication factor(N):

* Better resilience of your system(N-1 brokers can fail)
* But more replication (higher latency if acks=all)

### Configure topic

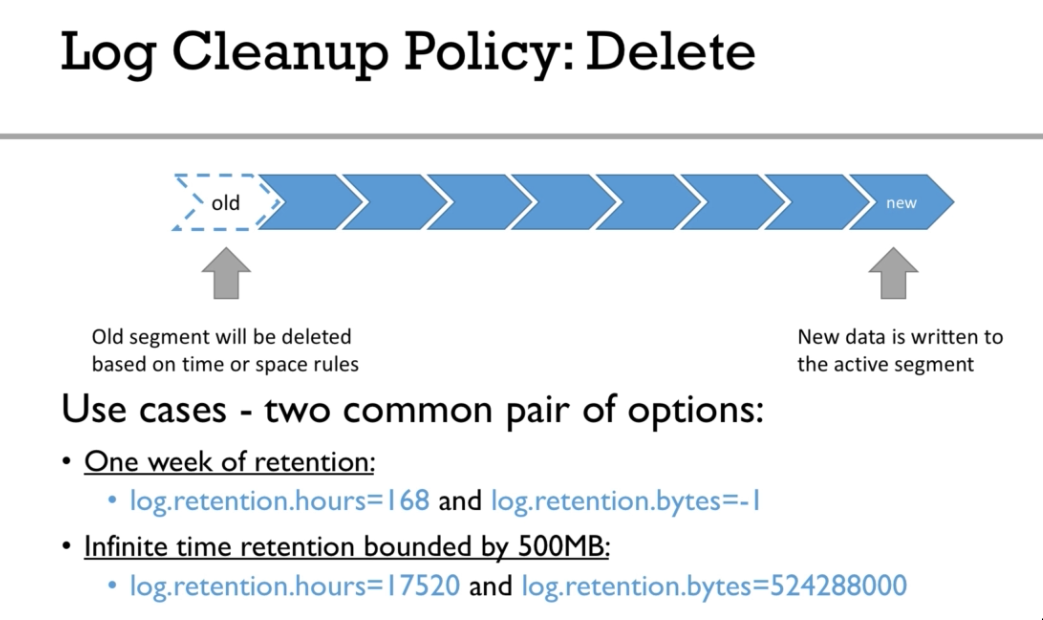
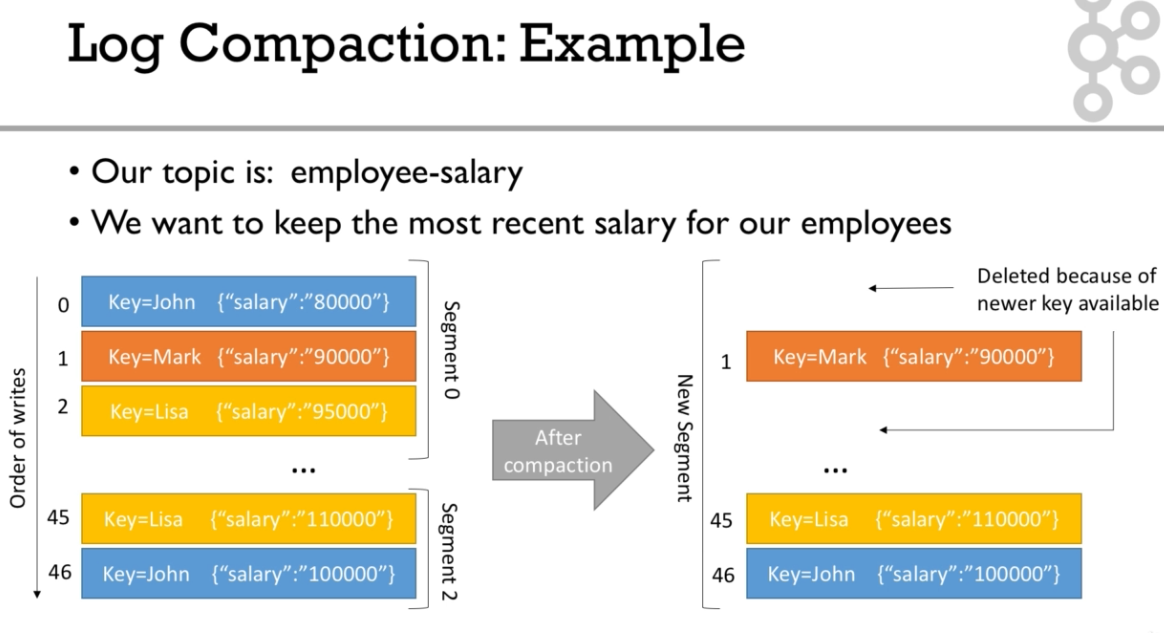
Why should I care about topic config?

* Brokers have defaults for all the topic configuration paramters
* These parameters impact performance and topic behavior

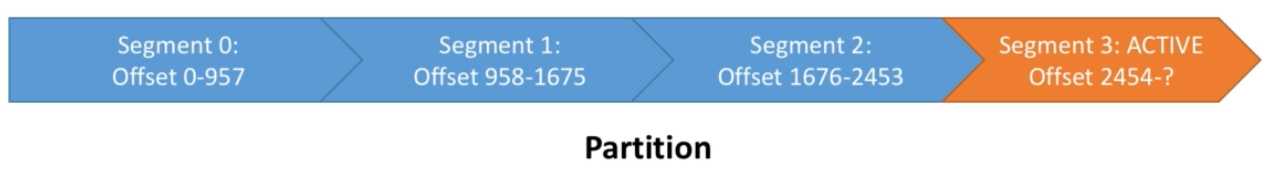
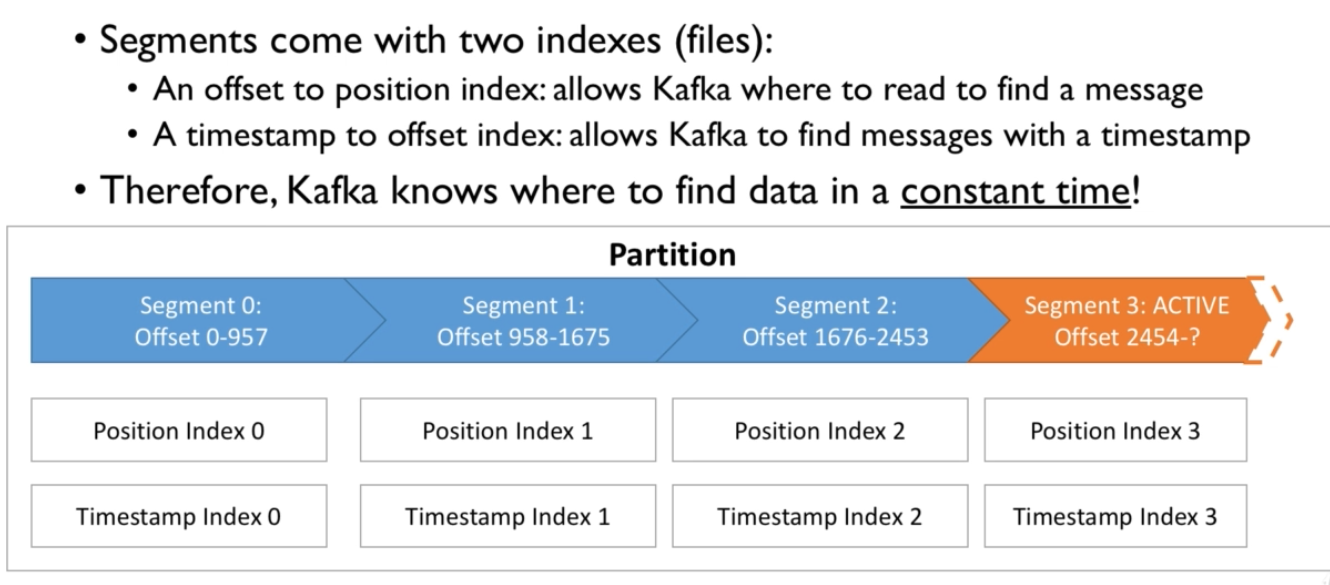
#### **Retention and clean up policies(Compaction)**

Retention is the durable storage of messages for some period of time. For example, a tracking topic might be retained for several days, whereas application metrics might be retained for only a few hours.

Retention policy:

* Delete: delete events older than retention time
* Compact: Only stores the most recent value for each key in the topic. Only works on topics for which applications produce events that contain both a key and a value
* log.cleanup.policy=delete:
  + Delete based on age of data(default is a week)
  + Deleted based on max size of log(default is -1 == inifinite)
* log.cleanup.policy=compact:
  + Delete based on keys of your message
  + Will delete old duplicate keys after the active segment is commited
* log.retention.hours:
  + number of hours to keep data fro
  + Higher number means more disk space
  + Lower number means that less data is retained(if your consumers are down for too long, they can miss data)
* log.retention.bytes:
  + Max side in bytes for each partition
* 
* 
* Delete records can still be seen by consumers for a period of delete.retention.ms

#### **Paritions and Segments:**

* Partitions are made of ...segments(files)
* Active segment means the segment are still being written to
* log.segment.bytes: the max size of a single segment in bytes
* log.segment.ms: the time Kafka will wait before comming the segment if not full
* 
* 

### Configure producer

#### **Kafka broker discovery:**

* Every Kafka broker is also called a "boostrap server"
* You only need to connect to one broker and you will be connected to the entire cluster
* Each broker knows about all brokers, topics and patitions

#### **Options for producer configuration:**

* **bootstrap.servers**: List of host:port of brokers that the producer will use to establish initial connection to the kafka cluster. It is recommended to include at least two, incase one goes down, the producer will still be able to connect to the cluster
* **acks**: Controls how many partition replicas must receive the record before the producer can consider write successful.
  + acks = 0: the producer will not wait for a reply from the broker before assuming the message was sent successfully. The message may be lost but it can send messaes as fast as the network will support, so this setting can be used to achieve very high throughput
  + acks=1: With a setting of 1, the producer will consider the write successful when the leader receives the record. The leader replica will know to immediately respond the moment it receives the record and not wait any longer.
  + acks=all: the producer will consider the write successful when all of the in-sync replicas receive the record. This is achieved by the leader broker being smart as to when it responds to the request — it’ll send back a response once all the in-sync replicas receive the record themselves.
  + Acks=all must be used in conjunction with min.insync.replicas
  + In-sync replicas: An in-sync replica (ISR) is a replica that has the latest data for a given partition. A leader is always an in-sync replica. A follower is an in-sync replica only if it has fully caught up to the partition it’s following.
  + Minimum In-Sync Replica: min.insync.replicas is a config on the broker that denotes the minimum number of in-sync replicas required to exist for a broker to allow acks=all requests. That means if you use replication.factor=3, min.insync=2, acks=all, you can only tolerate 1 broker going down, otherwise the producer will receive an exception on send.
  + max.in.flight.request.per.connection: setting while controls how many produce requests can be made in parallel. Set it to 1 if you need to ensure ordering(may impact throughput)

**acks setting is a good way to configure your prefered trade-off between durability guarantees and performance**

* **buffer memory**: this sets the amount of memory the producer will use to buffer messages waiting to be sent to brokers.
* **compression.type**: By default, messages are sent uncompressed. We can ues gzip, lz4. Enabling compression reduce network utilization and storage
* **retries**: How many times the producer will retry sending the message
* **batch.size**: The producer will batch them together. When the batch is full, all the messages in the batch will be sent.
* **client.id**: Use by the brokers to identify messages sent from the client

#### **Message compression for high-throughpput producer:**

* Compression is more effective the bigger the batch of message being sent to kafka is !
* compression.type can be none, gzip, lz4, snappy
* Much smaller producer request size
* Faster to transfer data over the network => less latency
* Better throughput
* Better disk utilization in Kafka(stored messages on disk are smaller)
* BUT Producers must commit some CPU cycles to compression and decompression
* Should use snappy or lz4, gzip is slow
* Always use compression in production and especially if you have high throughput
* Consinder tweaking linger.ms and batch.size to have bigger batches, and therefore more compression and higher throughput

#### **Producer batching:**

* By default, Kafka tries to send records as soon as possible:
  + It will have up to 5 requests in flight meanning 5 messages sent at the same time
  + If more messages have to be sent while others are in flight, Kafka is smart and will start batching them while they wait to send them all at once
* **linger.ms**: Number of milliseconds a producer is willing to wait before sending a batch out.
  + linger.ms=5 we increase the chances of messages being sent together in a batch.
  + At the expense of introducing a small delay, we can increase throughput, compression and efficiency for our producer
* **batch.size**: Maximum number of bytes that will be included in a batch. The default is 16KB
  + Increase batch size to 32KB or 64KB can help increasing throughput
  + A batch is allocated per partition, make sure don't set it to a number that's too high

If the producer produces faster than the broker can take, the records will be buffered in memory

* buffer.memory=33554432(32MB)
* If the buffer is full(all 32 MB), .send() method wil start to block
* max.block.ms=60000, the time the .send() will block until throwing an exception.

#### **Idempotent producer:**

The producer can introduce duplicate messages in Kafka due to network errors

### Configure consumer:

One of the kafka brokers get elected as a group coordinator. When a consumer want to join a group, it send a request to the group coordinator

The first consumer participating in a group is called a group leader.

During the rebalance activity, none of the consumers are allowed to read any messages.

rebalance: When a new consumer joins a consumer group the set of consumers attempt to "rebalance" the load to assign partitions to each consumer.

#### **Controlling consumer liveness?**

* **session.timeout.ms**: (default 10 seconds)
  + If no heartbeat is sent during that period, the consumer is considered dead
  + Set even lower to faster consumer rebalances
* **heartbeat.interval.ms**: (default 3 seconds)

#### **Consumer offset**

* Kafka stores the offset at which a consumer group has been reading
* The offsets are commited in a Kafka topic named **\_\_consumer\_\_offsets**
* If a consumer dies, it will be able to read back from where it left off thanks to the commited consumer offset

#### **Consumer offset reset behaviours:**

* Consumer can be down so it need to know where to start to read the log
* auto.offset.reset=latest: will read the end of the log
* auto.offset.reset=earliest: will read from the start of the log
* auto.offset.reset=none: Wil throw exception if no offset is found
* offset.retention.minutes: Consumer offsets can be lost if a consumer hasn't read new data in 7 days

#### **Delivery semantics for consumers:**

* At most once: offsets are commited as soons as the message is received
  + If the processing goes wrong, the message will be lost(it won't be read again)
  + Commits first then read
* At least once(usually preferred):
  + Offsets are commited after the message is processed
  + If the processing goes wrong, the message will be read again
  + This can result in duplicate processing of messages. Make sure your processing is idempotent
  + Read then commits
* Exactly once: only from kafka to kafka

#### **Offset management:**

In the event of rebalancing, when a consumer is a assigned the same partition, it should ask a question where to start. What is already processed by the previous owner? That's where Commited offset comes into play

* Current offset: Delivered records
* Commited offset: Processed records

How to commit?

* AutoCommit:
  + enable.auto.commit
  + auto.commit.interval.ms Can't avoid processing a record multiple times. If rebalancing happens before producer hasn't automatically commited
* Manual Commit:
  + Commit sync: block
  + Commit async: Commit async will not retry

#### **Consumer offset commits strategies:**

2 strategies

* (easy) enable.auto.commit = true & synchronous processing of batches If you don't use synchronous processing, you will be in "at-most-once" behavior because offsets will be commited before your data is processed. Quite risky for beginners
* (medium) enable.auto.commit = false & manual commit of offsets

## **Kafka Streams:**

* Kafka Streams applications do not run inside the Kafka brokers (servers) or the Kafka cluster – they are client-side applications. Kafka Streams is a library that is embedded and run within the user client application
* The unit of parallelism in Kafka Streams is **task**. But, the number of tasks is dependent on threads and input topic partitions
* One or more partitions get exclusively assigned to a task, and one or more tasks get exclusively assigned to a thread
* The maximum limit of tasks across all the application instances is the maximum number of input topic partitions
* Similarly the maximum limit of threads across all application instances is the maximum number of tasks possible
* Assigning more threads of application instances will keep the extra threads and instances idle, but they can take over tasks if one or more instances die
* Some stream processing applications don’t require state – they are **stateless** – which means the processing of a message is independent from the processing of other messages. Examples are when you only need to transform one message at a time, or filter out messages based on some condition
* In practice, however, most applications require state – they are **stateful** – in order to work correctly, and this state must be managed in a fault-tolerant manner. Your application is stateful whenever, for example, it needs to join, aggregate, or window its input data
* Kafka Streams uses **RocksDB** (other DBs are also pluggable) to store local states
* Local state in an application instance as well as remote states in other instances of the application can be queried from within the application. However, the data will be **read-only**
* For fault-tolerance of the state-store, an internal compacted **changelog topic** is used
* The changelog topic is also partitioned so that each task can exclusively be assigned a fair share of partitions
* The state store sends changes to the changelog topic in a batch, either when a default batch size has been reached or when the commit interval is reached
* If a machine, which runs tasks, fails and the tasks are restarted on a different machine, this internal changelog topic is replayed on the state store of the restarted task to restore its state
* To minimize this restoration time, users can configure their applications to have standby replicas of local states (i.e. fully replicated copies of the state) in other application instances
* When a task migration happens, Kafka Streams attempts to assign a task to an application instance where such a standby replica already exists
* Kafka Streams provides two APIs
  + **Kafka Streams DSL**
    - A high level API
    - Provides the most common data transformation operations such as map, filter, join, and aggregations out of the box
    - Built on top of the Streams Processor API
    - Provides built-in abstractions for streams and tables in the form of **KStream**, **KTable**, and **GlobalKTable**. A topic can be processed in the form of **KStream**, **KTable**, and **GlobalKTable**
  + **Processor API**
    - A low level API
    - Provides more flexibility than the DSL but at the expense of requiring more manual work
* **KTable** & **GlobalKTable** contain the latest value of each key (similar to a database table) and each record represents an UPDATE
* **KStream** provides all values for a given key and exh record represents an INSERT
* Unlike KTables, **GlobalKTables** load all partitions of the input topic within each application instance. Thus
  + GlobalKTables does not need to be co-partitioned with the input data
  + Input data can be joined with the GlobalKTable on any field (not necessarily the key)
  + GlobalKTable has more storage requirement that KTable as it loads all partitions in each app instance
  + GlobalKTables are good for broadcasting lookup data to all instances (a.k.a. star joins)
  + You must provide a name for the table (more precisely, for the internal state store that backs the table). This is required for supporting interactive queries against the table
* **KTables**
  + the local KTable instance of every application instance will be populated with data from only a subset of the partitions of the input topic. Collectively, across all application instances, all input topic partitions are read and processed
  + You must provide a name for the table (more precisely, for the internal state store that backs the table). This is required for supporting interactive queries against the table
* The computational logic of a Kafka Streams application is defined as a processor topology, which is a graph of stream processors (nodes) and streams (edges)
* The steps of writing a stream processing application:
  + Specify one or more input streams that are read from Kafka topics
  + Compose transformations on these streams
  + Write the resulting output streams back to Kafka topics, or expose the processing results of your application directly to other applications through interactive queries (e.g., via a REST API)
* The KStream and KTable interfaces support a variety of transformation operations. Each of these operations can be translated into one or more connected processors into the underlying processor topology
* Some KStream transformations may generate
  + one or more KStream objects, for example: - filter and map on a KStream will generate another KStream - branch on KStream can generate multiple KStreams
  + a KTable object, for example an aggregation of a KStream also yields a KTable
* All KTable transformation operations can only generate another KTable. However, the Kafka Streams DSL does provide a special function that converts a KTable representation into a KStream
* All of these KStream and KTable transformation methods can be chained together to compose a complex processor topology
* **Interactive Queries** - Allows an external application query the state of Kafka stream application instances. For e.g. a KTable local store in a given application instance does not contain the complete data - it contains the data of a subset of all partitions. If an external application queries the info of a given key and the query is routed through a load balancer, it can land in any of the app instances. Now, **interactive queries** will allow the application instance serve that request even if the data is present in the data store of some other app instance
* **Interactive queries** work in the following way:
  + The app developer needs to provide an endpoint in the Kafka stream application instance to query the locat data store
  + The app developer needs to configure the host:port pair of the endpoint in the property application.server. Each instance will have its own endpoint details in thi property
  + Kafka Streams keep track of this information by piggybacking on the consumer group membership protocol. Thus every app instance will discover the endpoint host:port details of every other app instance
  + Kafka streams provide APIs to query this endpoint metadata which allows an app instance determine the details of the app instance that holds the value of a given key
  + The app instance serving the request of the external app can now use the host:port details obtained from the metadata to query the appropriate app instance for the data and return the same to the external app
* There are two main differences between non-windowed and windowed aggregation with regard to key-design
  + For window aggregation the key is <K,W>, i.e., for each window a new key is used
  + Instead of using a single instance, Streams uses multiple instances (called “segments”) for different time periods
* After the window retention time has passed old segments can be dropped. Thus, RocksDB memory requirement does not grow infinitely
* Stateless transformations – Branch, Filter, Inverse Filter, FlatMap, ForEach, GroupByKey, GroupBy, Map, Peek, Print, SelectKey, Table To Stream
* Join operands - KStream-to-Kstream, KTable-to-KTable, Kstream-to-Ktable, KStream-to-GlobalKTable
* KStream-KStream join is always windowed join
* Outer join is supported only for KStream-to-KStream & KTable-to-KTable
* No join is supported for KTable-to-GlobalKTable
* Windowed joins are not supported for KStream-to-KTable, KTable-to-KTable, Kstream-to-GlobalKTable
* Input data must be co-partitioned when joining. This ensures that input records with the same key, from both sides of the join, are delivered to the same stream task during processing. It is the responsibility of the user to ensure data co-partitioning when joining
* The requirements for data co-partitioning are:
  + The input topics of the join (left side and right side) must have the same number of partitions.
  + All applications that write to the input topics must have the same partitioning strategy so that records with the same key are delivered to same partition number
* GlobalKTables do not require co-partitioning
* Kafka streams cannot verify the co-partitioning requirement for partition strategy
* Kafka streams throws TopologyBuilderException if the number of partitions on both sides of the join are not same
* Stream joining windows
  + **Tumbling time window** - Fixed-size, non-overlapping, gap-less windows
  + **Hopping time window** - Hopping time windows are windows based on time intervals. They model fixed-sized, (possibly) overlapping windows. A hopping window is defined by two properties: the window’s size and its advance interval (aka “hop”). The advance interval specifies by how much a window moves forward relative to the previous one
  + **Sliding time window** - A sliding window models a fixed-size window that slides continuously over the time axis; here, two data records are said to be included in the same window if (in the case of symmetric windows) the difference of their timestamps is within the window size. Thus, sliding windows are not aligned to the epoch, but to the data record timestamps
  + **Session window**
* It is generally preferable to use mapValues() and flatMapValues() as they ensure the key has not been modified and thus, the repartitioning step can be omitted
* Kafka Streams inserts a repartitioning step if a key-based operation like aggregation or join is preceded by a key changing operation like selectKey(), map(), or flatMap()
* Joining with a KStream will always yield a KStream
* Any streams and tables may be (continuously) written back to a Kafka topic with the method to()
* You may also leverage the Processor API from the DSL
* **StreamsConfig.APPLICATION\_ID\_CONFIG (application.id)** - It is a mandatory parameter and is used to derive
  + Consumer group id
  + Internal topic name prefix
  + Client id prefix
* **ConsumerConfig.AUTO\_OFFSET\_RESET\_CONFIG (auto.offset.reset)** -
  + It's applicable when the consumer group doesn't have any offset associated in Kafka
  + The possible values are - **earliest** (read the messages from the begining), **latest** (read the new messages)
* num.stream.threads - The number of threads per streams app instance

## **Kafka Connect:**

* Kafka Connect is a tool for scalably and reliably streaming data between Apache Kafka and other systems
* Connectors can be configured with transformations to make lightweight message-at-a-time modifications
* To copy data between Kafka and another system, users create a Connector for the system they want to pull data from or push data to
* Connectors come in two flavors: SourceConnectors import data from another system (e.g. JDBCSourceConnector would import a relational database into Kafka) and SinkConnectors export data (e.g. HDFSSinkConnector would export the contents of a Kafka topic to an HDFS file)
* Connectors do not perform any data copying themselves: their configuration describes the data to be copied, and the Connector is responsible for breaking that job into a set of Tasks that can be distributed to workers
* Tasks also come in two corresponding flavors: SourceTask and SinkTask
* With an assignment in hand, each Task must copy its subset of the data to or from Kafka

## **Schema Registry:**

* Schema Registry defines a scope in which schemas can evolve, and that scope is the subject. The name of the subject depends on the configured subject name strategy, which by default is set to derive subject name from topic name
* KafkaAvroSerializer and KafkaAvroDeserializer default to using -Key and -value as the corresponding subject name while registering or retrieving the schema
* The default behavior can be modified using the following properties:
  + key.subject.name.strategy - Determines how to construct the subject name under which the key schema is registered with the Schema Registry
  + value.subject.name.strategy - Determines how to construct the subject name under which the value schema is registered with Schema Registry
* Integration with Schema Registry means that Kafka messages do not need to be written with the entire Avro schema. Instead, Kafka messages are written with the schema id. The producers writing the messages and the consumers reading the messages must be using the same Schema Registry to get the same mapping between a schema and schema id
* A producer sends the new schema for Payments to Schema Registry. Schema Registry registers this schema Payments to the subject transactions-value, and returns the schema id of 1 to the producer. The producer caches this mapping between the schema and schema id for subsequent message writes, so it only contacts Schema Registry on the first schema write
* When a consumer reads this data, it sees the Avro schema id of 1 and sends a schema request to Schema Registry. Schema Registry retrieves the schema associated to schema id 1, and returns the schema to the consumer. The consumer caches this mapping between the schema and schema id for subsequent message reads, so it only contacts Schema Registry the on first schema id read
* Best practice is to register schemas outside of the client application to control when schemas are registered with Schema Registry and how they evolve.
* Disable automatic schema registration by setting the configuration parameter auto.register.schemas=false, as shown in the example below
* props.put(AbstractKafkaAvroSerDeConfig.AUTO\_REGISTER\_SCHEMAS, false);
* Kafka is used as Schema Registry storage backend

## **KSQL**

* The DDL statements (Imperative verbs that define metadata on the KSQL server by adding, changing, or deleting streams and tables) include:
  + CREATE STREAM
  + CREATE TABLE
  + DROP STREAM
  + DROP TABLE
  + CREATE STREAM AS SELECT (CSAS)
  + CREATE TABLE AS SELECT (CTAS)
* The DML statements (Declarative verbs that read and modify data in KSQL streams and tables. Data Manipulation Language statements modify data only and don’t change metadata) include:
  + SELECT
  + INSERT INTO
  + CREATE STREAM AS SELECT (CSAS)
  + CREATE TABLE AS SELECT (CTAS)
* The CSAS and CTAS statements occupy both categories, because they perform both a metadata change, like adding a stream, and they manipulate data, by creating a derivative of existing records
* In headless mode, KSQL stores metadata in the config topic
* In interactive mode, KSQL stores metatada in and builds metadata from the KSQL command topic. To secure the metadata, you must secure the command topic
* SHOW STREAMS and EXPLAIN statements run against the KSQL server that the KSQL client is connected to. They don’t communicate directly with Kafka
* CREATE STREAM WITH and CREATE TABLE WITH write metadata to the KSQL command topic
* Non-persistent queries based on SELECT that are stateless only read from Kafka topics, for example SELECT … FROM foo WHERE ….
* Non-persistent queries that are stateful read and write to Kafka, for example, COUNT and JOIN. The data in Kafka is deleted automatically when you terminate the query with CTRL-C

**Kafka CLI**

**Start Zookeeper with default configuration**

zookeeper-server-start.bat config\zookeeper.properties

**Start Kafka broker with default configuration**

kafka-server-start.bat config\server.properties

**Create a new topic**

kafka-topics.bat ^

--create ^

--bootstrap-server localhost:9092 ^

--replication-factor 1 ^

--partitions 2 ^

--topic word-count-input

**Delete a topic**

* If the flag delete.topic.enable is not enabled, the following command will be ignored
* The process is asynchronous and hence the topic won't be deleted immediately

kafka-topics.bat ^

--delete ^

--bootstrap-server localhost:9092 ^

--topic word-count-input

**Adding partitions**

* Key to partition mapping will change if the number of partitions is changed
* It is not advisable to add partitions to topics that have keys
* Deletion of partition is not possible

kafka-topics.bat ^

--alter ^

--bootstrap-server localhost:9092 ^

--topic word-count-input ^

--partitions 16 ^

**List topics**

kafka-topics.bat ^

--list ^

--bootstrap-server localhost:9092

**Describe a topic**

The output includes - partition count, topic configuration overrides, linting of each partition and its replica assignments

kafka-topics.bat ^

--describe ^

--bootstrap-server localhost:9092 ^

--topics-with-overrides ^ REM Display topics having configurtion overrides

--under-replicated-partitions ^ REM Display partitions with one or more out-of-sync replicas

--unavailable-partitions ^ REM Display partitions without a leader

--topic mytopic

**List consumer groups**

kafka-consumer-groups.bat ^

--bootstrap-server localhost:9092 ^

--list

**Describe consumer group**

Among other things following fields are displayed

* CURRENT-OFFSET - The next offset to be consumed by the consumer group
* LOG-END-OFFSET - The current high-water mark offset from the broker for the topic partition. The offset of the next message to be produced to this partition
* LAG - The difference between the consumer Current-Offset and the broker Log-End-Offset for the topic partition

kafka-consumer-groups.bat ^

--bootstrap-server localhost:9092 ^

--describe ^

--group testgroup ^

--members ^ REM Lists active members

--verbose REM Gives partition assignments

**Delete consumer group**

kafka-consumer-groups.bat ^

--bootstrap-server localhost:9092 ^

--delete ^

--group my-group ^

--group my-other-group

**Reset Offset**

kafka-consumer-groups.bat ^

--bootstrap-server localhost:9092 ^

--reset-offsets ^

--group consumergroup1 ^

--topic topic1 ^

--to-latest

**Adding or decommissioning brokers**

* The following command only generates the plan. However, it is not aware of partitions size, and neither can provide a plan to reduce the number of partitions to migrate from brokers to brokers. Therefore, edit the plan on your own
* Once the plan is ready, execute it with the option --execute
* Finally verify the status using --verify
* Throttling is important to ensure producers and consumers in the cluster are not impacted
* The throttle can be changed even when the rebalance is ongoing by rerunning the same command with the new throttle value
* --verify option removes the throttle

kafka-reassign-partitions.bat ^

--zookeeper localhost:2181 ^

--topics-to-move-json-file topics-to-move.json ^

--broker-list "5,6" ^

--generate

kafka-reassign-partitions.bat ^

--zookeeper localhost:2181 ^

--reassignment-json-file expand-cluster-reassignment.json ^ REM JSON file content is the output of --generate command

--throttle 50000000 ^ REM 50MB/sec

--execute

**Update dynamic config**

* The following command updated sets the parameter log.cleaner.thread for broker id 0

kafka-configs.bat ^

--bootstrap-server localhost:9092 ^

--entity-type brokers ^

--entity-name 0 ^

--alter ^

--add-config log.cleaner.threads=2

**Describe dynamic config**

kafka-configs.bat ^

--bootstrap-server localhost:9092 ^

--entity-type brokers ^

--entity-name 0 ^

--describe

**Delete dynamic config**

kafka-configs.bat ^

--bootstrap-server localhost:9092 ^

--entity-type brokers ^

--entity-name 0 ^

--alter ^

--delete-config log.cleaner.threads

**Update cluster-wide default dynamic config**

kafka-configs.bat ^

--bootstrap-server localhost:9092 ^

--entity-type brokers ^

--entity-default ^

--alter ^

--add-config log.cleaner.threads=2

**Console producer**

kafka-console-producer.bat ^

--broker-list localhost:9092 ^

--topic word-count-input

**Console consumer**

kafka-console-consumer.bat ^

--bootstrap-server localhost:9092 ^

--topic word-count-output ^

--from-beginning ^

--formatter kafka.tools.DefaultMessageFormatter ^

--property print.key=true ^

--property print.value=true ^

--property key.deserializer=org.apache.kafka.common.serialization.StringDeserializer ^

--property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer

## **Miscellaneous**

### Production Deployment

* Java 8 with G1 Collector
* On AWS, for lower latency I/O optimized instances will be good
* Extents File System (XFS) perform well for Kafka Workload
* The mount points should have noatime option set to eliminate the overhead of updating access time
* export KAFKA\_JVM\_PERFORMANCE\_OPTS="-server -XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35 -XX:+DisableExplicitGC-Djava.awt.headless=true"
* vm.swappiness = 1 - A low value means the kernel will try to avoid swapping as much as possible making less memory available for page cache
* vm.dirty\_background\_ratio = 5 (default - 10) - The percentage of system memory which when dirty, system will start writing to disk
* vm.dirty\_ratio = 60 (default - 15) - The percentage of memory which when dirty, the process doing writes would block and write out dirty pages to the disks
* Tuning vm.dirty\_background\_ratio and vm.dirty\_ratio as above relies on:
  + Good disk I/O performance (SSD or RAID)
  + Replication is used in the cluster to guard against system failure
* net.core.wmem\_default = 131072 (128 KiB)
* net.core.rmem\_default = 131072 (128 KiB)
* net.core.wmem\_max = 131072 (128 KiB)
* net.core.rmem\_max = 2097152 (2 MiB)
* net.ipv4.tcp\_wmem = 4096 65536 2048000 (4 KiB 64 KiB 2 MiB)
* net.ipv4.tcp\_rmem = 4096 65536 2048000 (4 KiB 64 KiB 2 MiB)
* vm.max\_map\_count = 65535 - Maximum number of memory map areas a process may have (aka vm.max\_map\_count). By default, on a number of Linux systems, the value of vm.max\_map\_count is somewhere around 65535. Each log segment, allocated per partition, requires a pair of index/timeindex files, and each of these files consumes 1 map area. In other words, each log segment uses 2 map areas. Thus, each partition requires minimum 2 map areas, as long as it hosts a single log segment. That is to say, creating 50000 partitions on a broker will result allocation of 100000 map areas and likely cause broker crash with OutOfMemoryError (Map failed) on a system with default vm.max\_map\_count
* File descriptor limits: Kafka uses file descriptors for log segments and open connections. If a broker hosts many partitions, consider that the broker needs at least (number\_of\_partitions)\*(partition\_size/segment\_size) to track all log segments in addition to the number of connections the broker makes. We recommend at least 100,000 allowed file descriptors for the broker processes as a starting point
* Kafka uses heap space very carefully and does not require setting heap sizes more than 6 GB
* You can do a back-of-the-envelope estimate of memory needs by assuming you want to be able to buffer for 30 seconds and compute your memory need as write\_throughput \* 30
* A machine with 64 GB of RAM is a decent choice, but 32 GB machines are not uncommon. Less than 32 GB tends to be counterproductive
* Do not share the same drives used for Kafka data with application logs or other OS filesystem activity to ensure good latency
* You should use RAID 10 if the additional cost is acceptable. Otherwise, configure your Kafka server with multiple log directories, each directory mounted on a separate drive
* You should avoid network-attached storage (NAS). NAS is often slower, displays larger latencies with a wider deviation in average latency, and is a single point of failure
* Modern data-center networking (1 GbE, 10 GbE) is sufficient for the vast majority of clusters.
* You should avoid clusters that span multiple data centers, even if the data centers are colocated in close proximity; and avoid clusters that span large geographic distances
* num.partitions - The default number of log partitions for auto-created topics. You should increase this since it is better to over-partition a topic. Over-partitioning a topic leads to better data balancing and aids consumer parallelism. For keyed data, you should avoid changing the number of partitions in a topic
* Key Service Goals
  + Throughput
  + Latency
  + Durability
  + Availability
* Partitions are the unit of parallelism. On both the producer and the broker side, writes to different partitions can be done fully in parallel. Therefore, in general, more partitions lead to higher throughput. However, following tradeoffs need to be considered
  + More partitions need more open file handles (3 per segment - the segment itself, offset index, time index)
  + More partitions need more memory map areas (2 per segment - the offset index, time index)
  + Post release 1.1.0 with the use of Zookeeper async APIs and batching request to other brokers during failover, things have improved here
  + More partitions may increase latency as a limited number of threads in a given broker will do replication in the broker for all the partitions in the cluster. Unless the in-sync replicas are all updated (committed), the message will not be available to the consumers
  + More partitions means more memory requirements for producer as each producer thread will have some buffer for each partiton
* As a rule of thumb, we recommend each broker to have up to 4,000 partitions and each cluster to have up to 200,000 partitions

### Key Metrics

* Consumer Metrics
  + MaxLag - The number of messages the consumer lags behind the producer
  + MinFetchRate - If the MinFetchRate of the consumer drops to almost 0, the consumer is likely to have stopped. If the MinFetchRate is non-zero and relatively constant, but the consumer lag is increasing, it indicates that the consumer is slower than the producer. If so, the typical solution is to increase the degree of parallelism in the consumer. This may require increasing the number of partitions of a topic
* Broker Metrics
  + UnderReplicatedPartitions - Number of under-replicated partitions (| ISR | < | all replicas |). Alert if value is greater than 0
  + OfflinePartitionsCount - Number of partitions that don’t have an active leader and are hence not writable or readable. Alert if value is greater than 0
  + ActiveControllerCount - Number of active controllers in the cluster. Alert if the aggregated sum across all brokers in the cluster is anything other than 1 because there should be exactly one controller per cluster
  + MinFetchRate - Rate of fetching messages from the leader by the replicas
  + MaxLag - The number of messages the replica lags behind the leader

### Deploying on AWS

* EBS st1 for log.dirs
* d2.xlarge if you’re using instance storage, or r4.xlarge if you’re using EBS
* Kafka was designed to run within a single data center. As such, we discourage distributing brokers in a single cluster across multiple regions. However, we recommend “stretching” brokers in a single cluster across availability zones within the same region
* Kafka 0.10 supports rack awareness, which makes spreading replicas across availability zones much easier to configure

### ZooKeeper Basics

* A Zookeeper cluster is called an ensemble
* Due to the algorithm used, it is recommended that ensembles contain an odd number of servers (e.g., 3, 5, etc.)
* A three-node ensemble can run with one node missing
* A five-node ensemble can run with two nodes missing
* It's a good idea to run Zookeeper in a five-node ensemble so that the ensemble can run even when one node goes down while another node is taken down due to configuration change etc.
* ZooKeeper parameters
  + initLimit - the amount of time to allow the followers to connect with a leader
  + syncLimit - Limits how out-of-sync followers can be with the leader
  + tickTime - Both values are a number of tickTime units
* In the following example, the initLimit is 20 \* 2000 ms or 40 seconds
* tickTime=2000
* initLimit=20

syncLimit=5

### Avro Basics

* BACKWARD: consumer using schema X can process data produced with schema X or X-1
* BACKWARD\_TRANSITIVE: consumer using schema X can process data produced with schema X, X-1, or X-2
* FORWARD: data produced using schema X can be ready by consumers with schema X or X-1
* FORWARD\_TRANSITIVE: data produced using schema X can be ready by consumers with schema X, X-1, or X-2
* FULL: backward and forward compatible between schemas X and X-1
* FULL\_TRANSITIVE: backward and forward compatible between schemas X, X-1, and X-2
* Allowed operations for FULL & FULL\_TRANSITIVE
  + Add optional fields
  + Delete optional fields
* Allowed operations for BACKWARD & BACKWARD\_TRANSITIVE
  + Add optional fields
  + Delete fields
* Allowed operations for FORWARD & FORWARD\_TRANSITIVE
  + Add fields
  + Delete optional fields
* Order of upgrading clients
  + BACKWARD or BACKWARD\_TRANSITIVE: there is no assurance that consumers using older schemas can read data produced using the new schema. Therefore, upgrade all consumers before you start producing new events.
  + FORWARD or FORWARD\_TRANSITIVE: there is no assurance that consumers using the new schema can read data produced using older schemas. Therefore, first upgrade all producers to using the new schema and make sure the data already produced using the older schemas are not available to consumers, then upgrade the consumers.
  + FULL or FULL\_TRANSITIVE: there are assurances that consumers using older schemas can read data produced using the new schema and that consumers using the new schema can read data produced using older schemas. Therefore, you can upgrade the producers and consumers independently.
* Primitive data types –
  + null, boolean, int, long, float, double, bytes, string
* Complex data types –
  + record, enum, array, map, union, fixed
* Record attributes –
  + name, namespace, doc, aliases, type, fields
* Enum attributes
  + name, namespace, aliases, doc, symbols
* Enums once defined cannot be changed. Otherwise, the compatibility will break
* Arrays are a way to define an unbounded list of items:
  + {"type": "array", "items": "string"}
* Unions can allow a field value to take different types. One common use case is to define an optional value:
  + {"name": "middle\_name", "type": {"null", "string"}, "default": null}
* Maps are a way to define a list of keys and values, where the keys are strings:
  + {"type": "map", "values": "string"}

### Operations & Troubleshooting

* **Logging** -
  + server.log
  + controller.log - ERROR, FATAL & WARN messages need to be looked at
  + state-change.log
* **Rolling Restart**
  + Run the brokers with controlled.shutdown.enable=true
  + Ensure the server is healthy with no under replicated partition
  + Identify the Controller broker. It should be the last one to shutdown
  + Login to one broker server and run bin/kafka-server-stop
  + Restart the server using bin/kafka-server-start etc/kafka/server.properties and wait until it catches up before stopping the next broker

References:

* Udemy courses by <https://www.udemy.com/user/stephane-maarek/>
* <https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines>
* <https://github.com/gkoenig/kafka-benchmarking>
* others