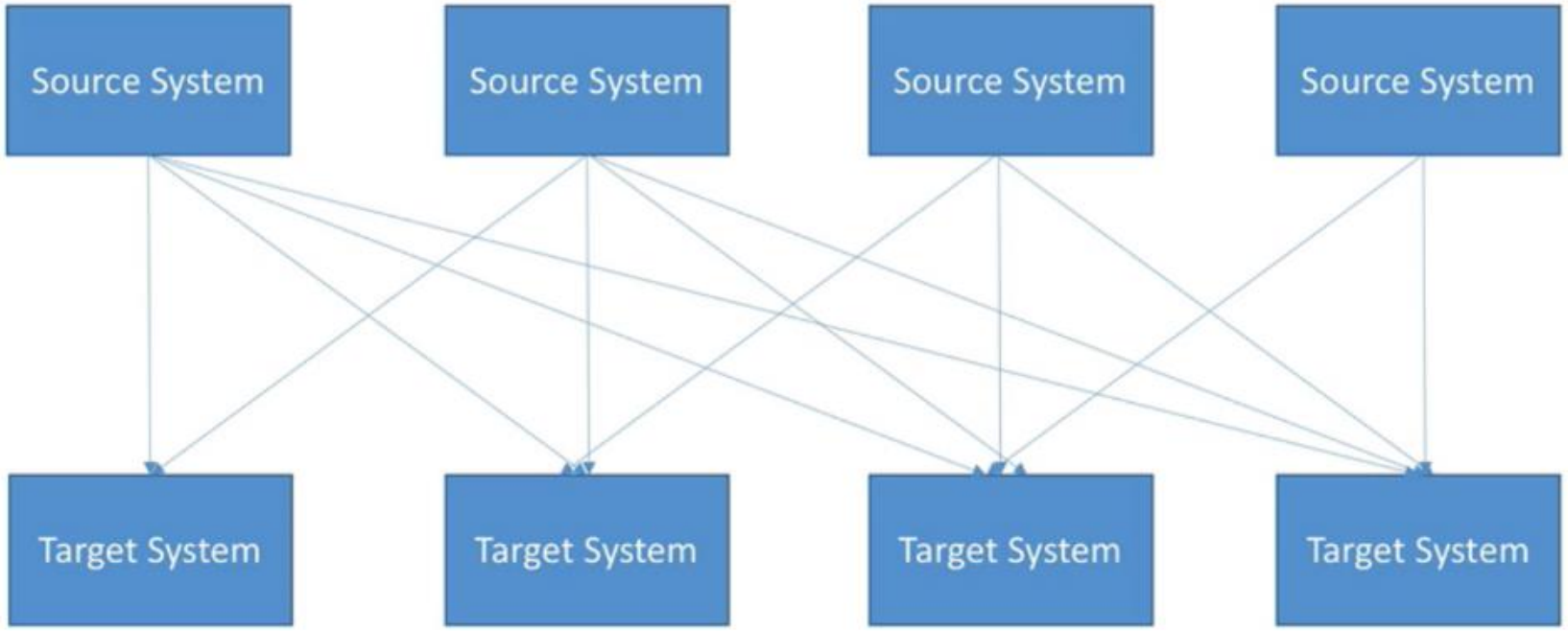
Apache Kafka

Naveen Kumar K S

Adith.Naveen@gmail.com

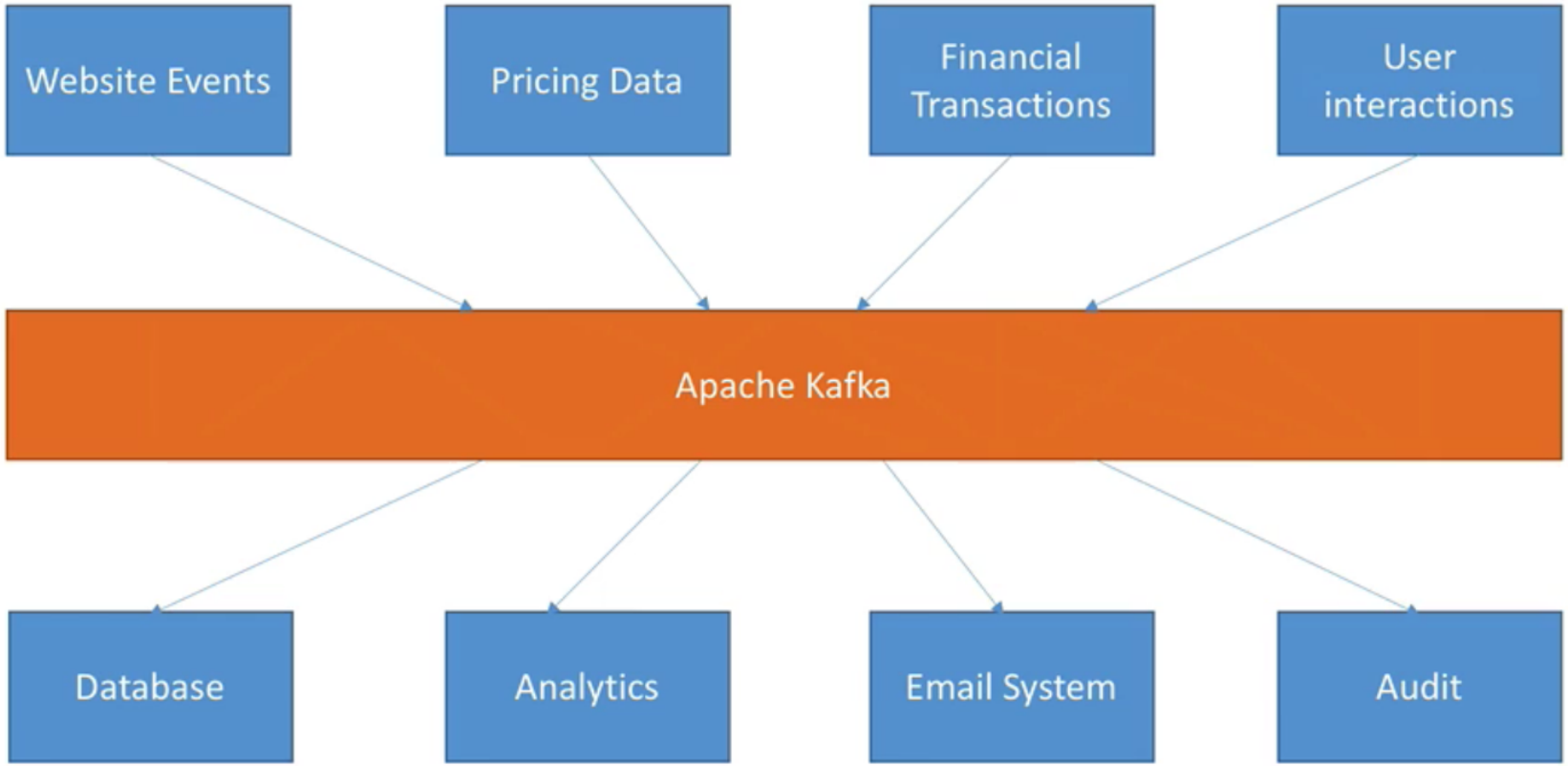
http://naveenks.com

Why Apache Kafka



Very complicated

Why Apache Kafka: Decoupling of data streams



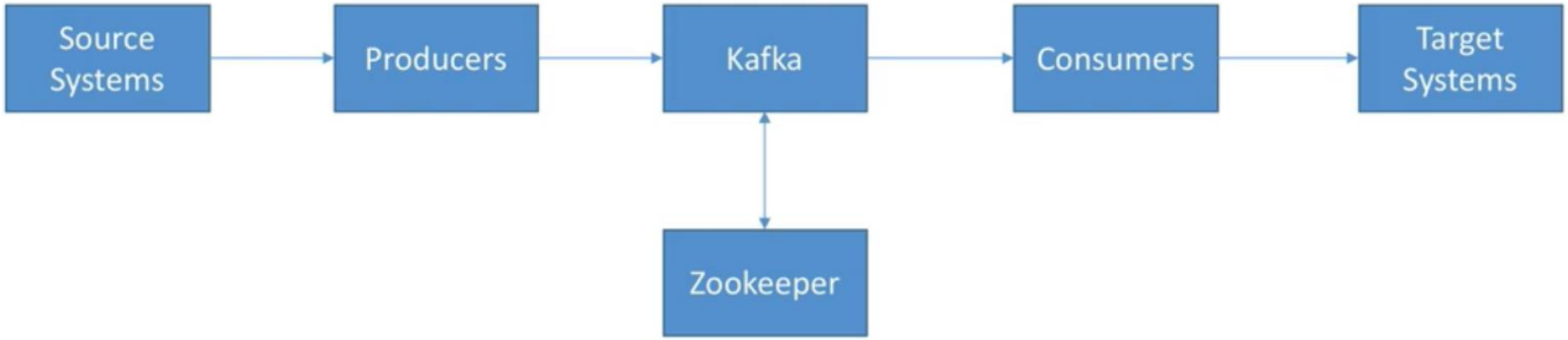
Why Apache Kafka

- Distributed, resilient architecture, fault tolerant
- Horizontal scalability
- High performance (latency of less than 10ms) – real time
- Used by the 2000+ firms:
  - LinkedIn
  - Netflix
  - AirBnB
  - Yahoo
  - Walmart

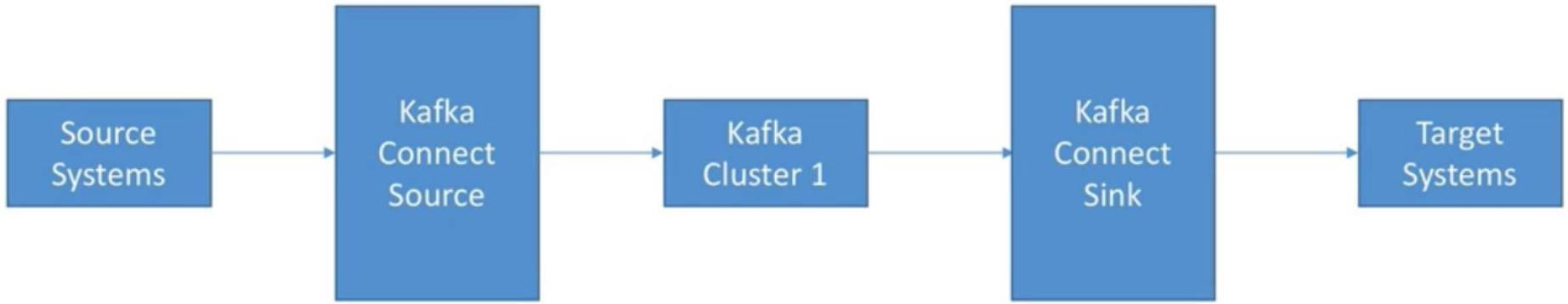
Apache Kafka: Use cases

- Messaging System
- Activity Tracking
- Gather metrics from many different locations
- Application Logs gathering
- Stream processing (with the Kafka Streams API or Spark for example)
- De-coupling of system dependencies
- Integration with Spark, Flink, Storm, Hadoop, and many other Big Data technologies

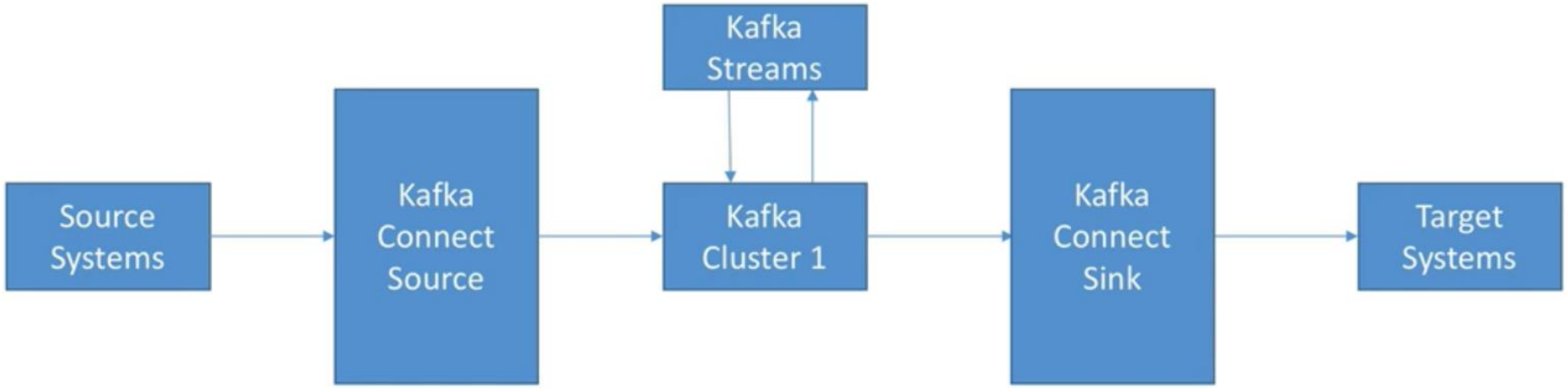
Kafka Ecosystem
Kafka Core



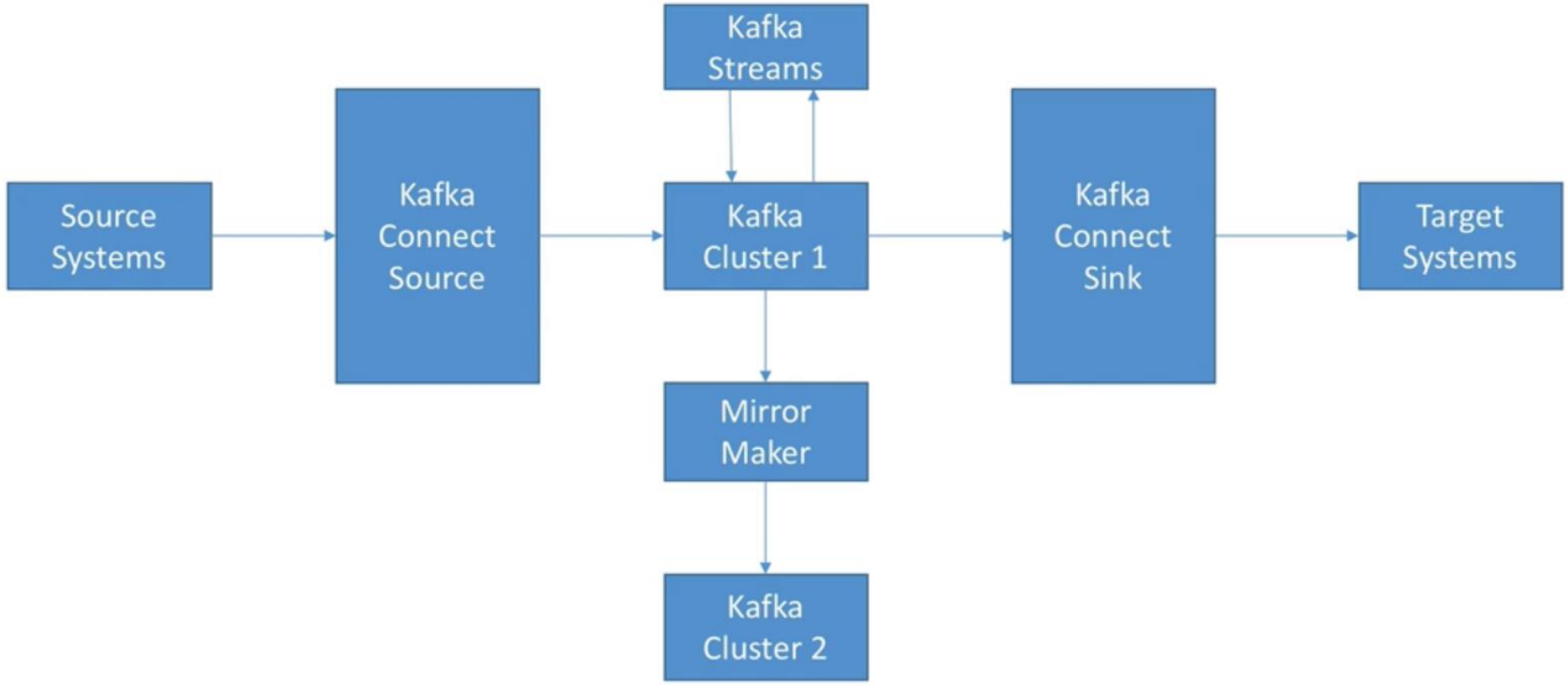
Kafka Ecosystem
Kafka Extended API



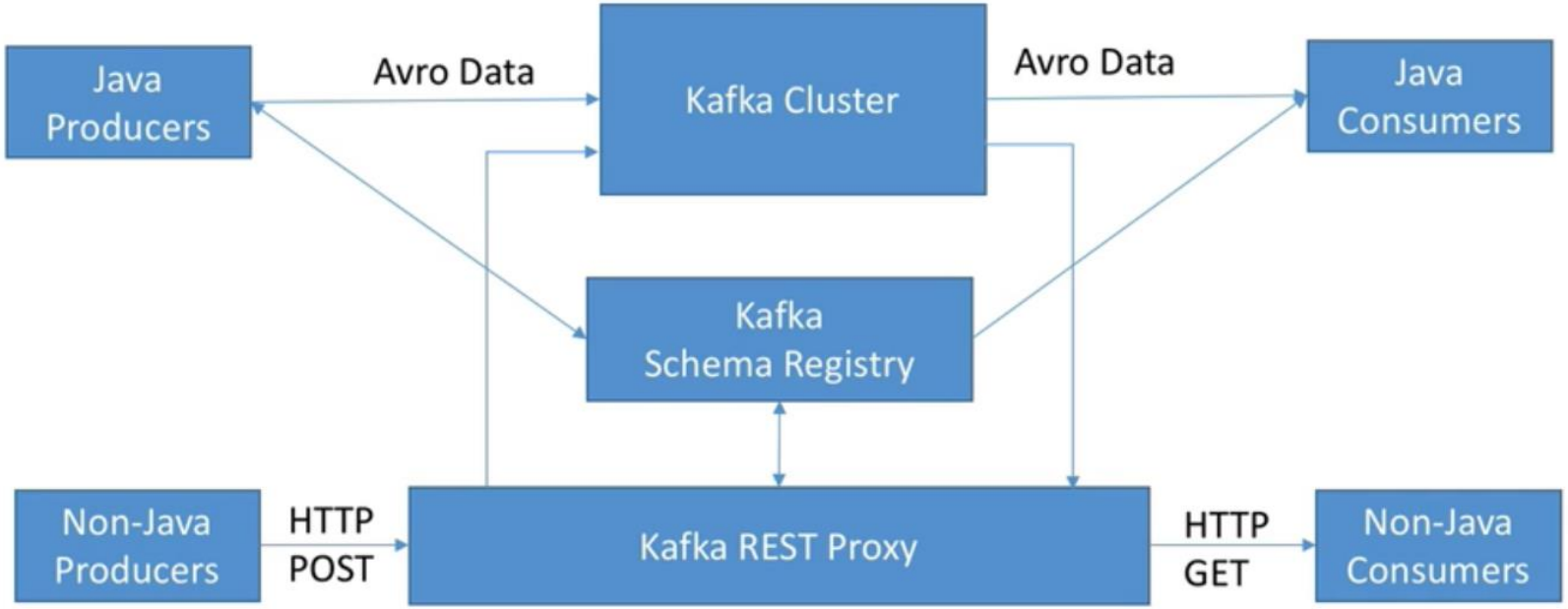
Kafka Ecosystem
Kafka Extended API



Kafka Ecosystem
Kafka Extended API



Kafka Ecosystem: Confluent Components Schema Registry and REST Proxy



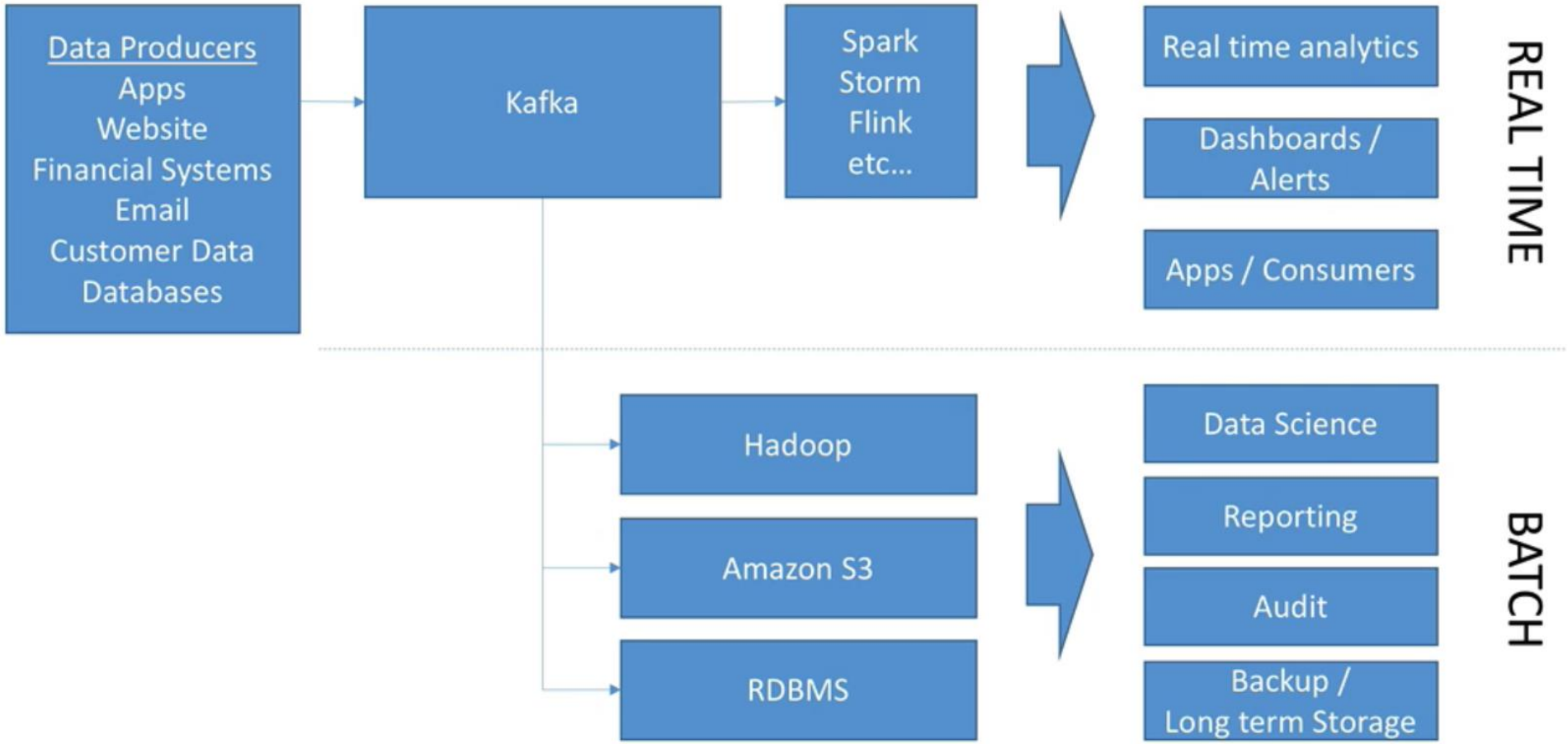
Kafka Ecosystem: Administration and Monitoring Tools

- Topics UI (Landoop): View the topics content
- Schema UI (Landoop): Explore the Schema registry
- Connect UI (Landoop): Create and monitor Connect tasks
- Kafka Manager (Yahoo): Overall Kafka Cluster Management
- Burrow (LinkedIn): Kafka Consumer Lag Checking
- Exhibitor (Netflix): Zookeeper Configuration, Monitoring, Backup
- Kafka Monitor (LinkedIn): Cluster health monitoring
- Kafka Tools (LinkedIn): Broker and topics administration tasks simplified
- Kafkat (Airbnb): More broker and topics administration tasks simplified
- JMX Dump: Dump JMX metrics from Brokers
- Control Centre / Auto Data Balancer / Replicator (Confluent): Paid tools

Apache Kafka Series

1. **Kafka For Beginners:** get a strong base for Kafka, basic operations, write your first producers and consumers
2. Kafka Connect API: Understand how to import / export data to / from Kafka
3. Kafka Streams API: Learn how to process and transform data within Kafka
4. Kafka Cluster Setup & Administration: Get a deep understanding of how Kafka & Zookeeper works, how to Setup Kafka and various administration tasks in AWS.
5. Confluent Components: REST Proxy and Schema Registry
6. Kafka Security: Setup Kafka security in a Cluster and Integrate your applications with Kafka Security

Kafka in the Enterprise Architecture



This slide is intentionally left blank

Topics and partitions

- Topics: a particular stream of data
  - Similar to a table in a database (without all the constraints)
  - You can have as many topics as you want
  - A topic is identified by its name
- Topics are split in partitions
  - Each partition is ordered
  - Each message within a partition gets an incremental id, called offset

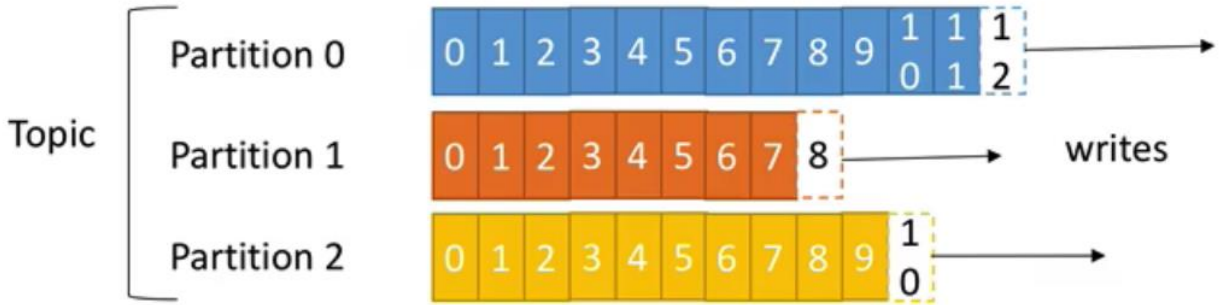
Partition 0

Topics and partitions

- Topics: a particular stream of data
  - Similar to a table in a database (without all the constraints)
  - You can have as many topics as you want
  - A topic is identified by its name
- Topics are split in partitions
  - Each partition is ordered
  - Each message within a partition gets an incremental id, called offset

Topics and partitions

- Topics: a particular stream of data
  - Similar to a table in a database (without all the constraints)
  - You can have as many topics as you want
  - A topic is identified by its name
- Topics are split in partitions
  - Each partition is ordered
  - Each message within a partition gets an incremental id, called offset



Topics and partitions

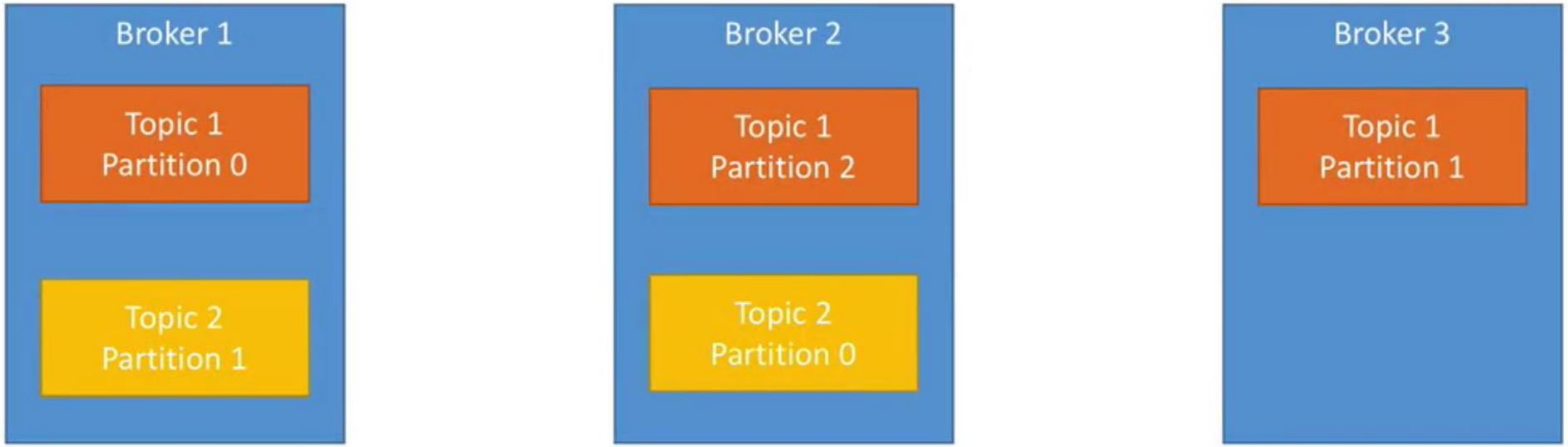
- Offset only have a meaning for a specific partition.
  - E.g. offset 3 in partition doesn't represent the same data as offset 3 in partition 1
- Order is guaranteed only within a partition (not across partitions)
- Data is kept only for a limited time (default is two weeks)
- Once the data is written to a partition, <u>it can't be changed</u> (immutability)
- Data is assigned randomly to a partition unless a key is provided (more on this later)
- You can have as many partitions per topic as you want

Brokers

- A Kafka cluster is composed of multiple brokers (servers)
- Each broker is identified with its ID (integer)
- Each broker contains certain topic partitions
- After connecting to any broker (called a bootstrap broker), you will be connected to the entire cluster
- A good number to get started is 3 brokers, but some big clusters have over 100 brokers

Brokers and topics

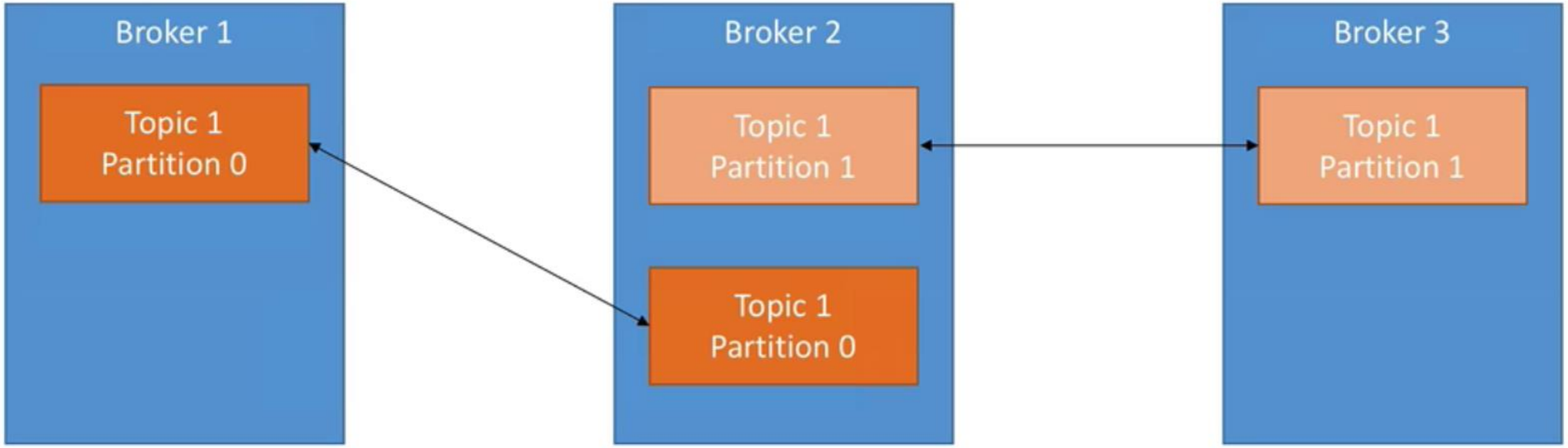
- Example of 2 topics (3 partitions and 2 partitions)



- Data is distributed and Broker 3 doesn't have any Topic 2 data

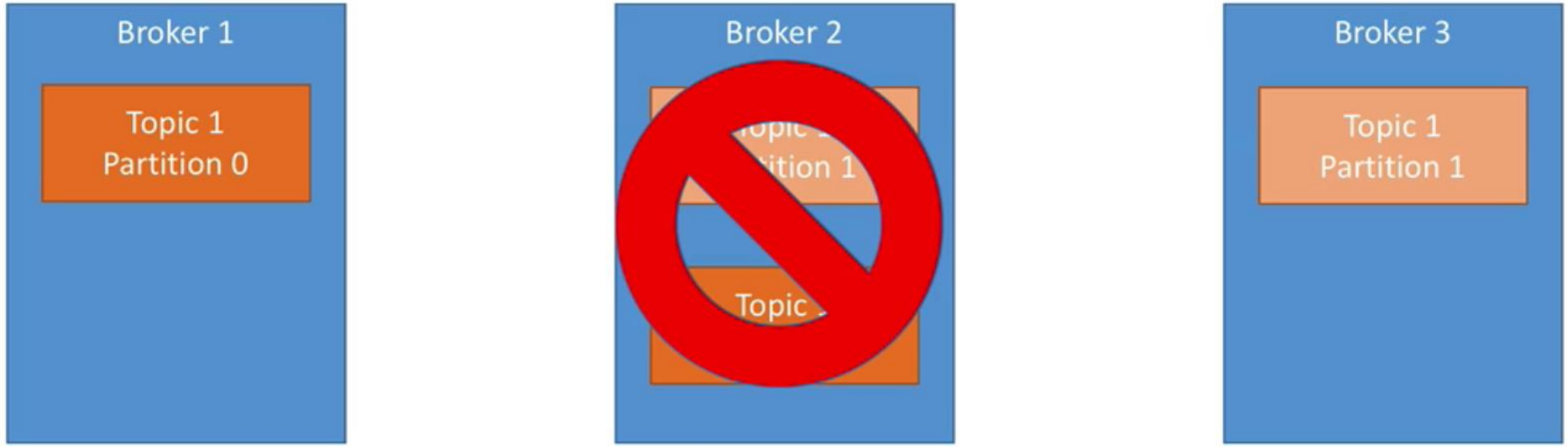
Topic replication factor

- Topics should have a replication factor > 1 (usually between 2 and 3)
- This way if a broker is down, another broker can serve the data
- Example: Topic with 2 partitions and replication factor of 2



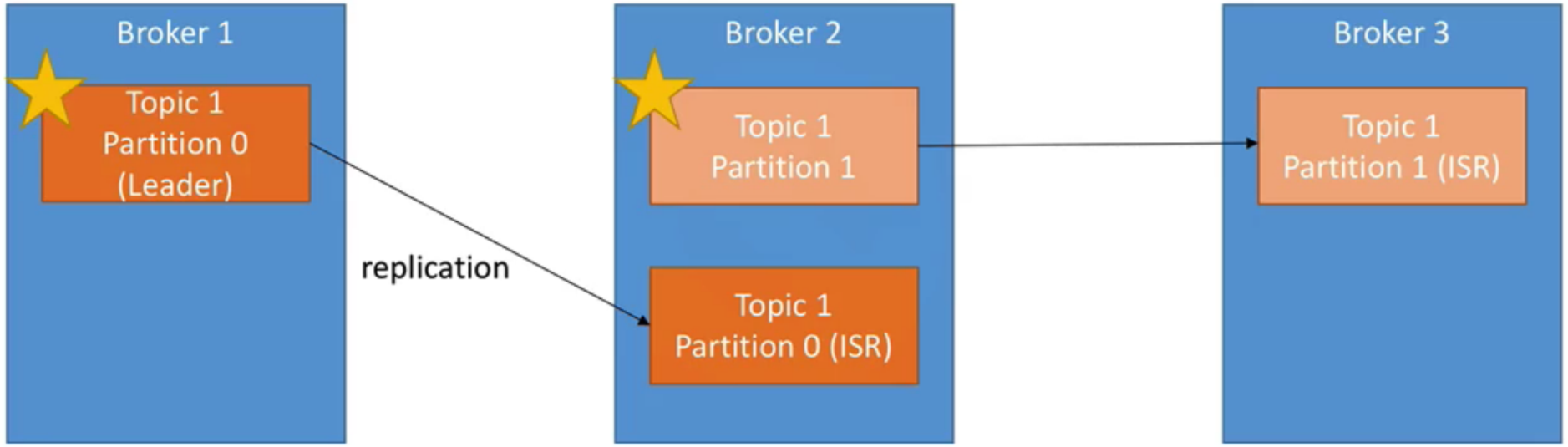
Topic replication factor

- Example: we lost Broker 2
- Result: Broker 1 and 3 can still serve the data



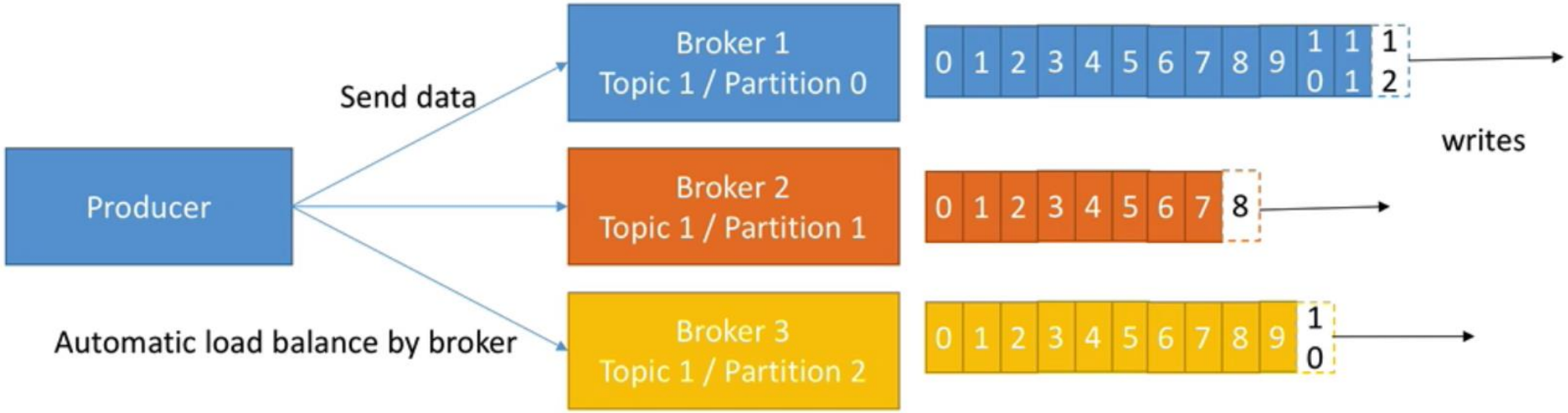
Concept of Leader for a partition

- **At any time only 1 broker can be a leader for a given partition**
- **Only that leader can receive and serve data for a partition**
- The other brokers will synchronize the data
- There each partition has: one leader, and multiple ISR (in-sync replica)



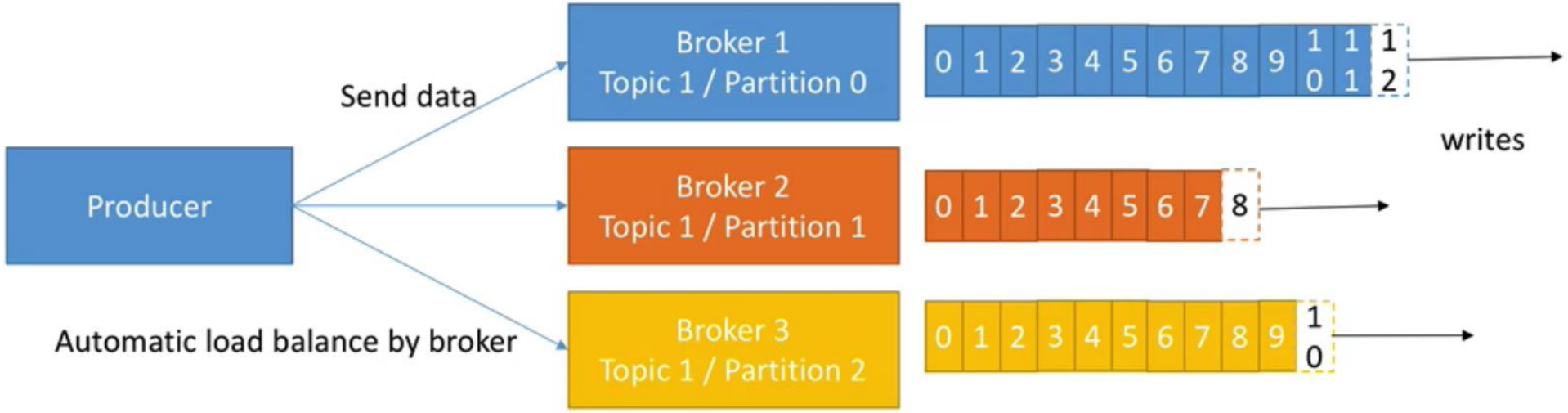
Producers

- Producers write data to topics.
- **They only have to specify the topic name and one broker to connect to, and Kafka will automatically take care of routing the data to the right brokers**



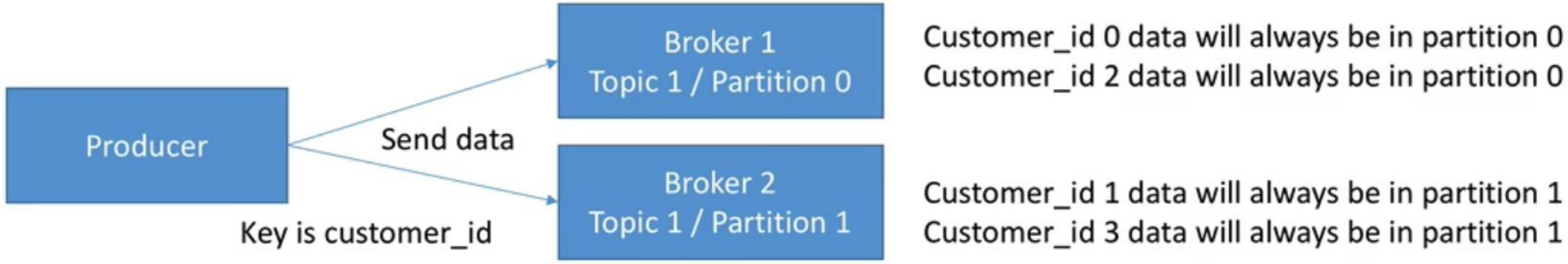
Producers

- Producers can choose to receive acknowledgment of data writes:
  - Acks=0: Producer won't wait for acknowledgment (possible data loss)
  - Acks=1: Producer will wait for leader acknowledgment (limited data loss)
  - Acks=all: Leader + replicas acknowledgment (no data loss)



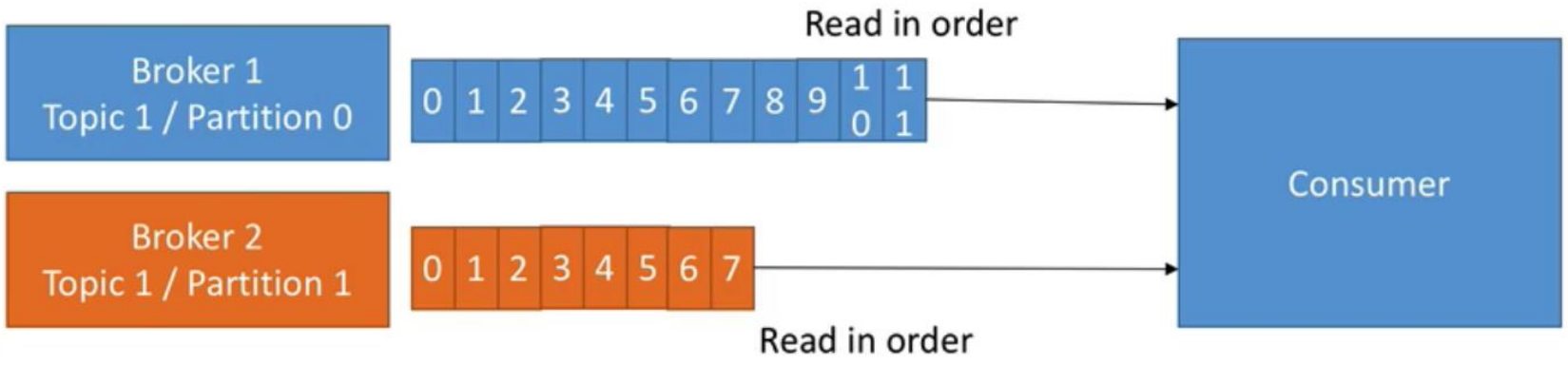
Producers: Message keys

- Producers can choose to send a key with the message
- If a key is sent, then the producer has the guarantee that all messages for that key will always go to the same partition
- This enables to guarantee ordering for a specific key



Consumers

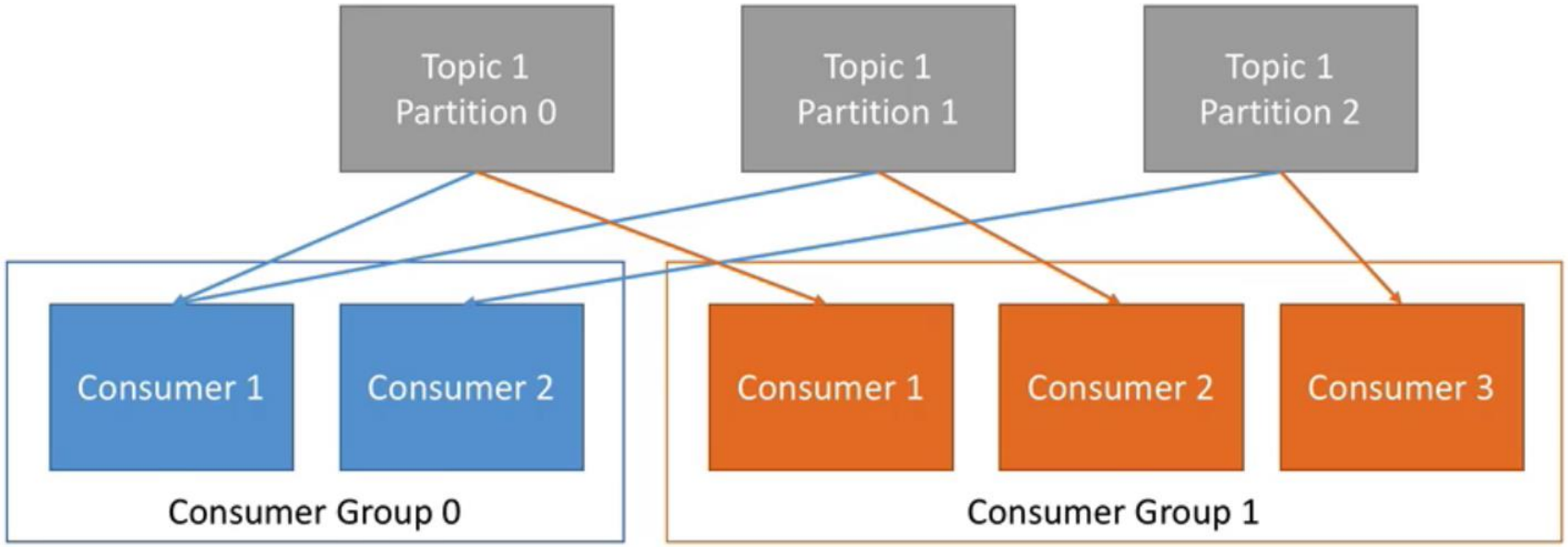
- Consumers read data from a topic
- **They only have to specify the topic name and one broker to connect to, and Kafka will automatically take care of pulling the data from the right brokers**
- Data is read in order **for each partitions**



This slide is intentionally left blank

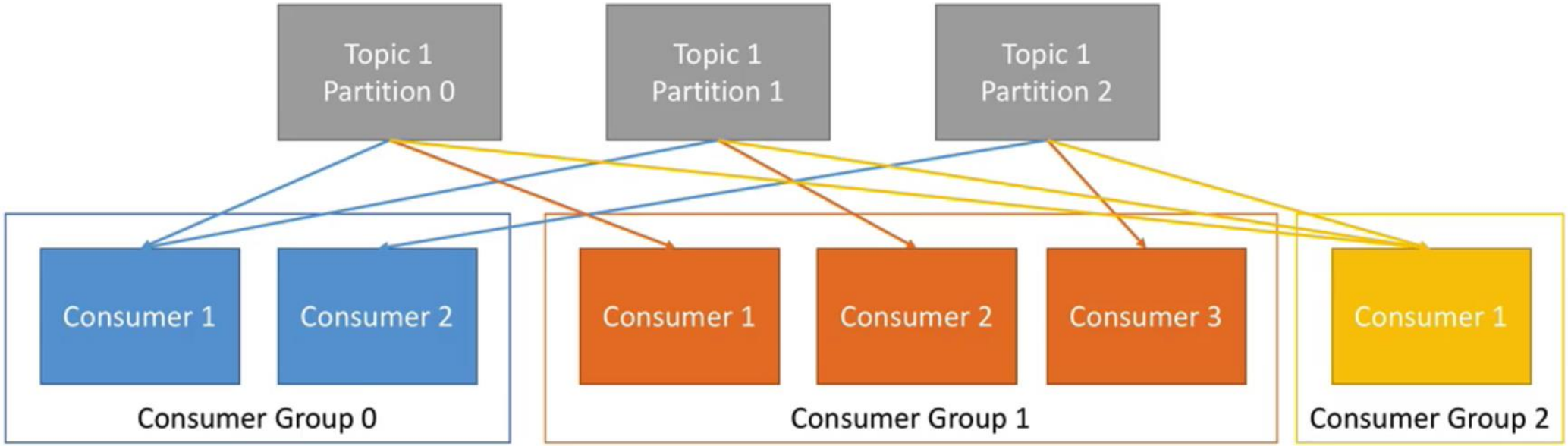
Consumer Groups

- Consumers read data in consumer groups
- Each consumer within a group reads from exclusive partitions
- You cannot have more consumers than partitions (otherwise some will be inactive)



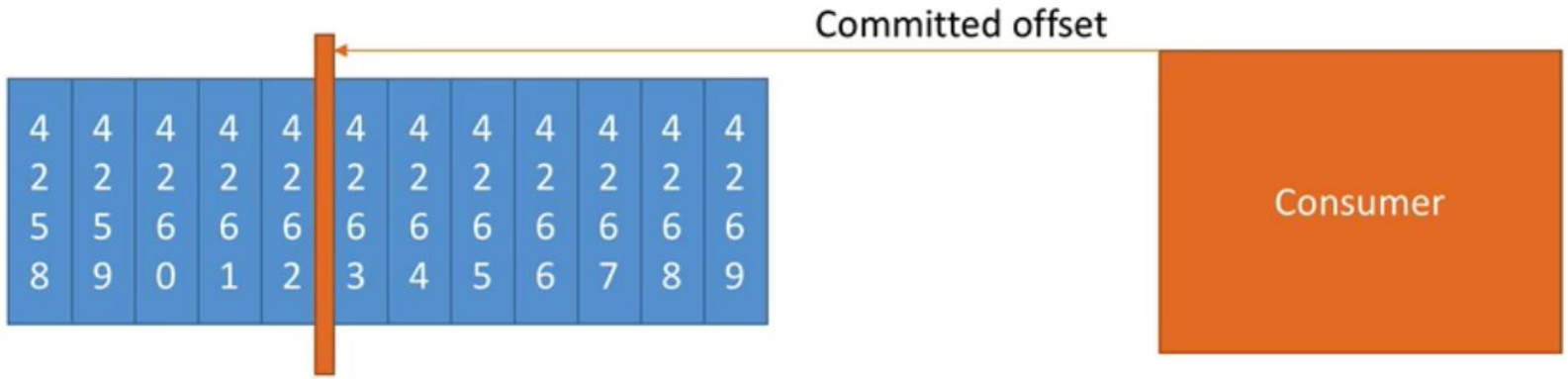
Consumer Groups

- Consumers read data in consumer groups
- Each consumer within a group reads from exclusive partitions
- You cannot have more consumers than partitions (otherwise some will be inactive)



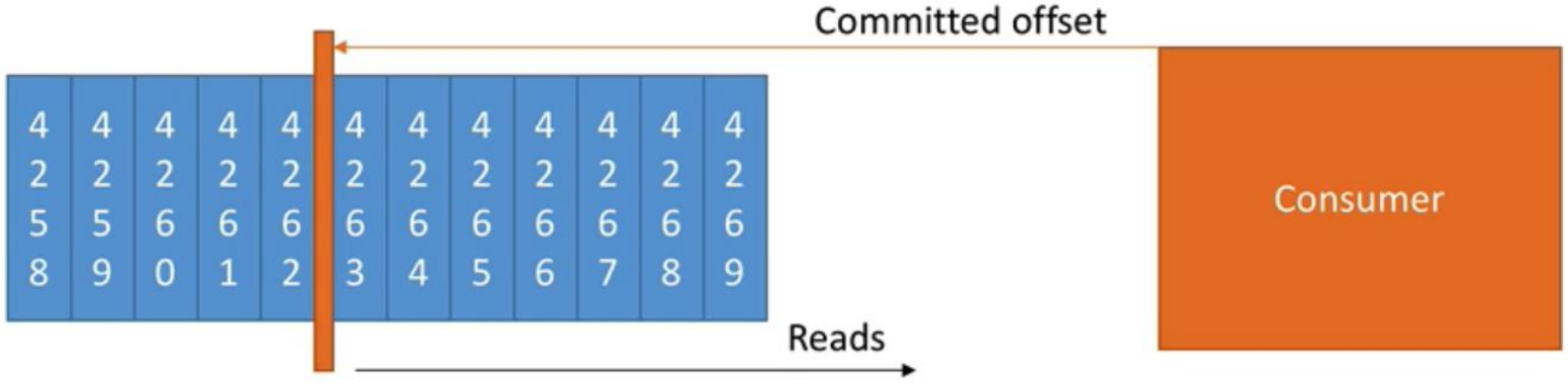
Consumer Offsets

- Kafka stores the offsets at which a consumer group has been reading
- The offsets commit live in a Kafka topic named "__consumer_offsets"
- When a consumer has processed data received some Kafka, it should be committing the offsets
- If a consumer process dies, it will be able to read back from where it left off thanks to consumer offsets!

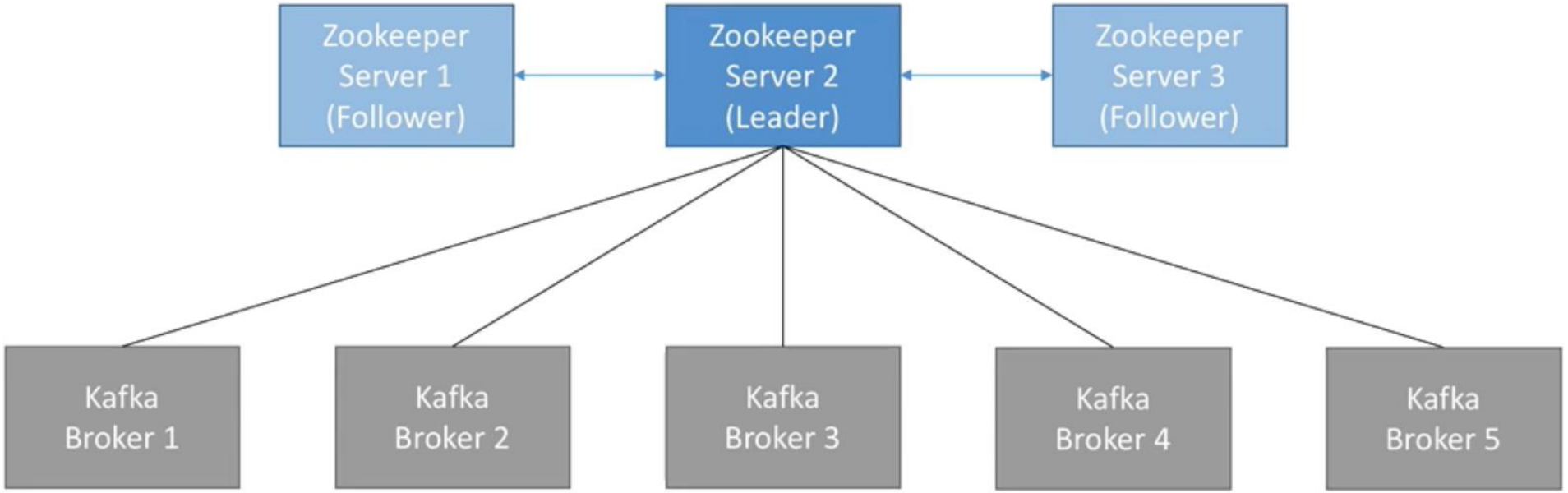


Consumer Offsets

- Kafka stores the offsets at which a consumer group has been reading
- The offsets commit live in a Kafka topic named "__consumer_offsets"
- When a consumer has processed data received some Kafka, it should be committing the offsets
- If a consumer process dies, it will be able to read back from where it left off thanks to consumer offsets!



Zookeeper


Zookeeper Server 1 (Follower)
Zookeeper Server 2 (Leader)
Zookeeper Server 3 (Follower)
Kafka Broker 1
Kafka Broker 2
Kafka Broker 3
Kafka Broker 4
Kafka Broker 5

Kafka Guarantees

- Messages are appended to a topic-partition in the order they are sent
- Consumers read messages in the order stored in a topic-partition
- With a replication factor of N, producers and consumers can tolerate up to N-1 brokers being down
- This is why a replication factor of 3 is a good idea:
  - Allows for one broker to be taken down for maintenance
  - Allows for another broker to be taken down unexpectedly
- As long as the number of partitions remains constant for a topic (no new partitions), the same key will always go to the same partition

Delivery semantics for consumers

- As learned before, consumers choose when to commit offsets.

- At most once: offsets are committed as soon as the message is received. If the processing goes wrong, the message will be lost (it won't be read again).
- At least once: offsets are committed 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** (i.e. processing again the messages won't impact your systems)
- Exactly once: Very difficult to achieve / needs strong engineering.

Delivery semantics for consumers

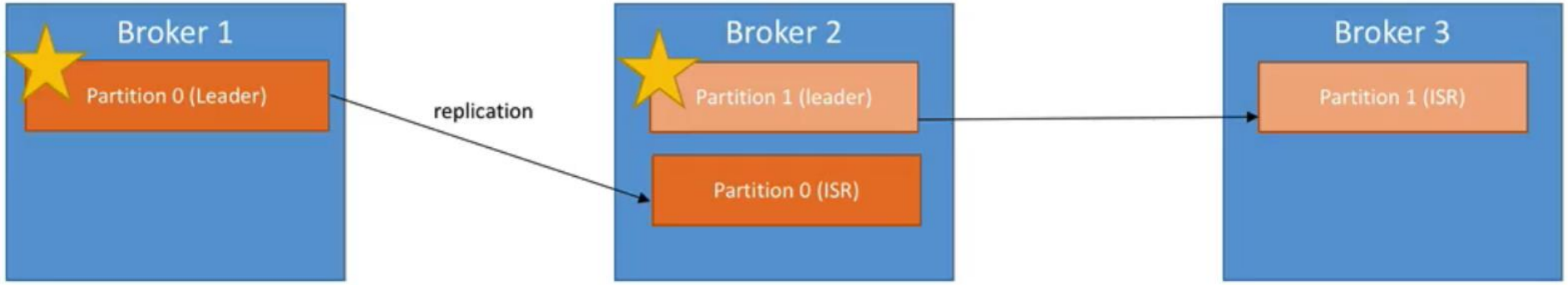
- As learned before, consumers choose when to commit offsets.

- At most once: offsets are committed as soon as the message is received. If the processing goes wrong, the message will be lost (it won't be read again).
- At least once: offsets are committed 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** (i.e. processing again the messages won't impact your systems)
- Exactly once: Very difficult to achieve / needs strong engineering.

Bottom line: most often you should use at least once processing and ensure your transformations / processing are idempotent

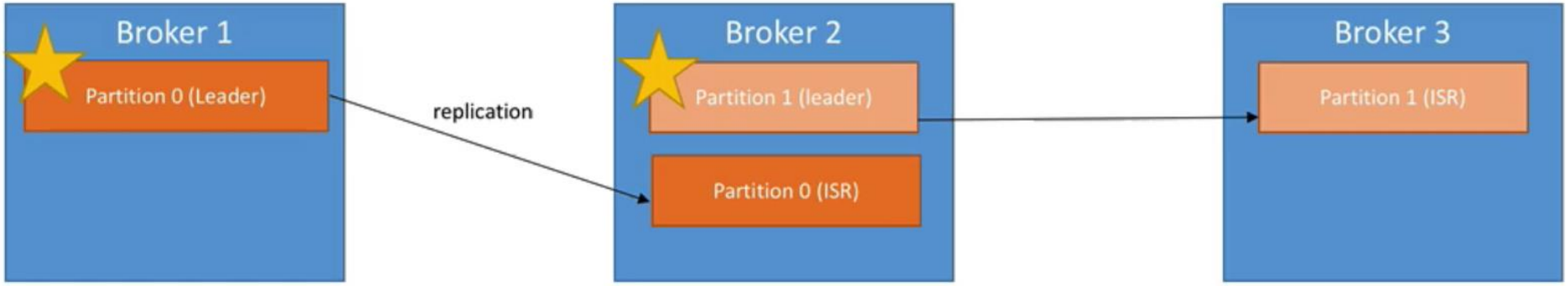
Partitions Count, Replication Factor

- The two most important parameters when creating a topic.
- They impact performance and durability of the system overall



Partitions Count, Replication Factor

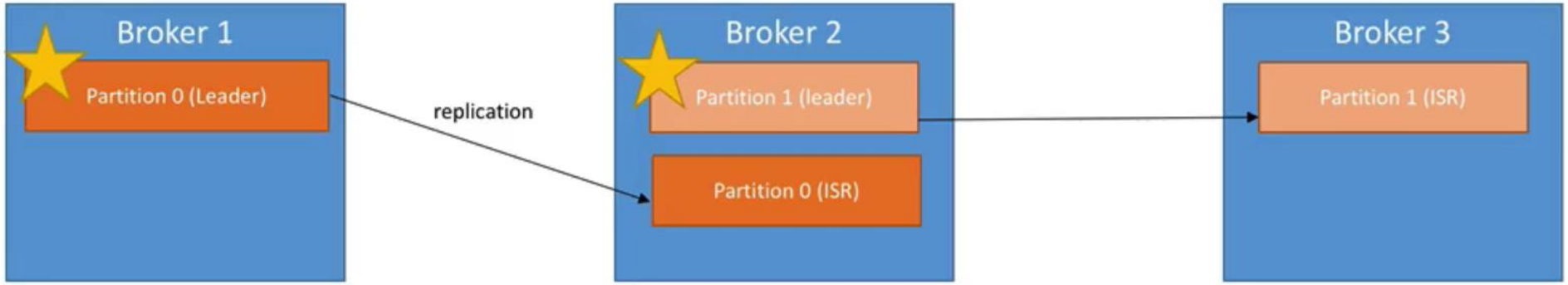
- The <u>two most important</u> parameters when creating a topic.
- They impact performance and durability of the system overall



- It is best to get the parameters right the first time!
  - If the Partitions Count increases during a topic lifecycle, you will break your keys ordering guarantees

Partitions Count, Replication Factor

- The two most important parameters when creating a topic.
- They impact performance and durability of the system overall



- It is best to get the parameters right the first time!
  - If the Partitions Count increases during a topic lifecycle, you will break your keys ordering guarantees
  - If the Replication Factor increases during a topic lifecycle, you put more pressure on your cluster, which can lead to unexpected performance decrease

Partitions Count

- Roughly, each partition can get a throughput of 10MB / sec
- More partitions implies:
  - Better parallelism, better throughput
  - BUT more files opened on your system
  - BUT if a broker fails (unclean shutdown), lots of concurrent leader elections
  - BUT added latency to replicate (in the order of milliseconds)

- <u>Guidelines:</u>
  - Keep the number of partition per brokers between 2000 and 4000
  - Keep the number of partitions in the cluster to less than 20,000

Partitions Count

- Roughly, each partition can get a throughput of 10MB / sec
- More partitions implies:
  - Better parallelism, better throughput
  - BUT more files opened on your system
  - BUT if a broker fails (unclean shutdown), lots of concurrent leader elections
  - BUT added latency to replicate (in the order of milliseconds)

- <u>Guidelines:</u>
  - Keep the number of partition per brokers between 2000 and 4000
  - Keep the number of partitions in the cluster to less than 20,000
  - **Partitions per topic = (1 to 2) x (# of brokers), max 10 partitions**

Partitions Count

- Roughly, each partition can get a throughput of 10MB / sec
- More partitions implies:
  - Better parallelism, better throughput
  - BUT more files opened on your system
  - BUT if a broker fails (unclean shutdown), lots of concurrent leader elections
  - BUT added latency to replicate (in the order of milliseconds)

- <u>Guidelines:</u>
  - Keep the number of partition per brokers between 2000 and 4000
  - Keep the number of partitions in the cluster to less than 20,000
  - **Partitions per topic = (1 to 2) x (# of brokers), max 10 partitions**
  - Example: in a 3 brokers setup, 3 or 6 partitions is a good number to start with

Replication Factor

- Should be at least 2, maximum of 3
- The higher the replication factor:
  - Better resilience of your system (N-1 brokers can fail)
  - BUT longer replication (higher latency is acks=all)
  - BUT more disk space on your system (50% more if RF is 3 instead of 2)

- <u>Guidelines:</u>
  - **Set it to 3** (you must have at least 3 brokers for that)
  - If replication performance is an issue, get a better broker instead of less RF

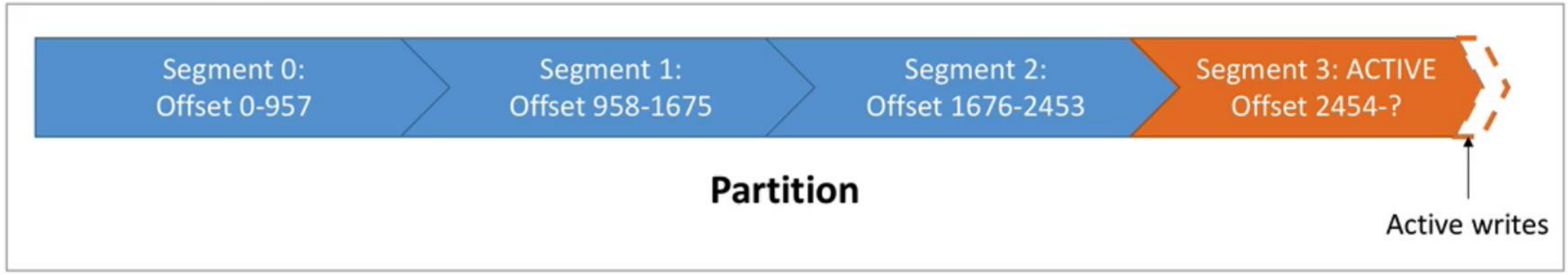
This slide is intentionally left blank

Partitions and Segments

- Topics are made of partitions (we already know that)
- Partitions are made of… segments (files)!

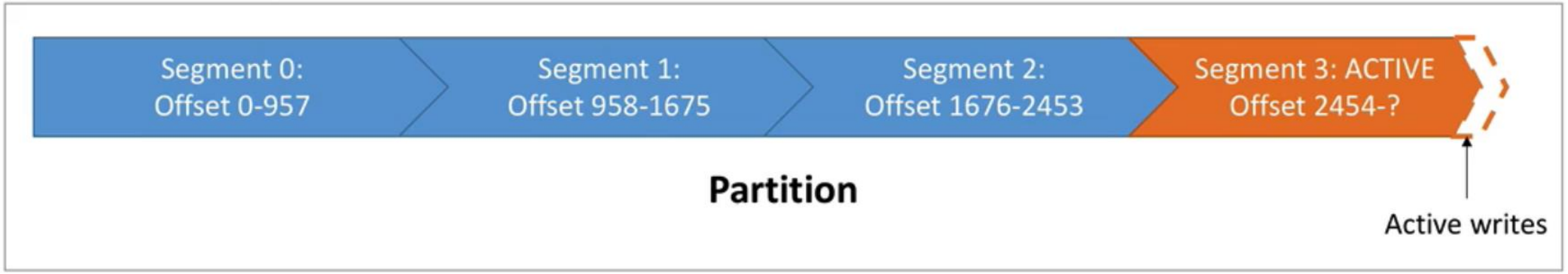
Partitions and Segments

- Topics are made of partitions (we already know that)
- Partitions are made of… segments (files)!



Partitions and Segments

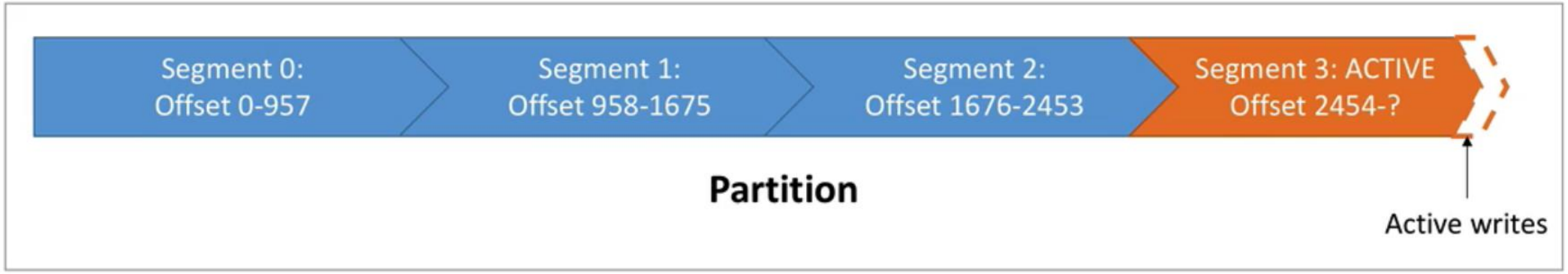
- Topics are made of partitions (we already know that)
- Partitions are made of… segments (files)!



- Only one segment is ACTIVE (the one data is being written to)

Partitions and Segments

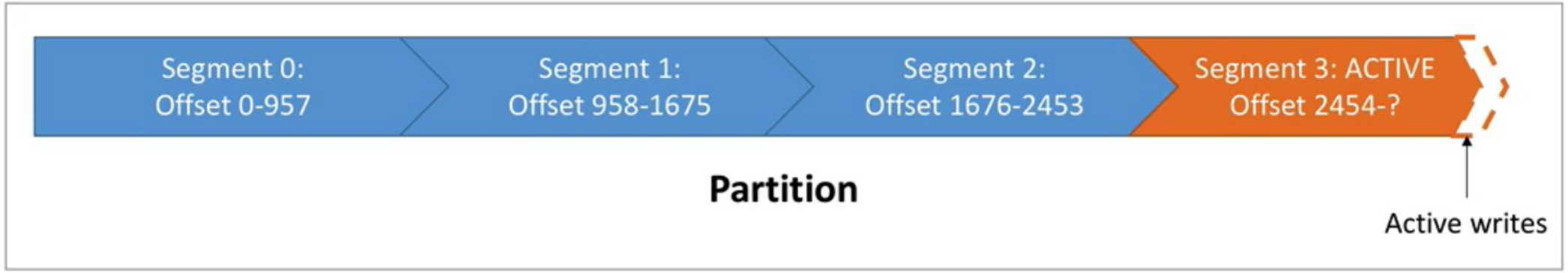
- Topics are made of partitions (we already know that)
- Partitions are made of… segments (files)!



- Only one segment is ACTIVE (the one data is being written to)
- Two segment settings:
  - log.segment.bytes: the max size of a single segment in bytes

Partitions and Segments

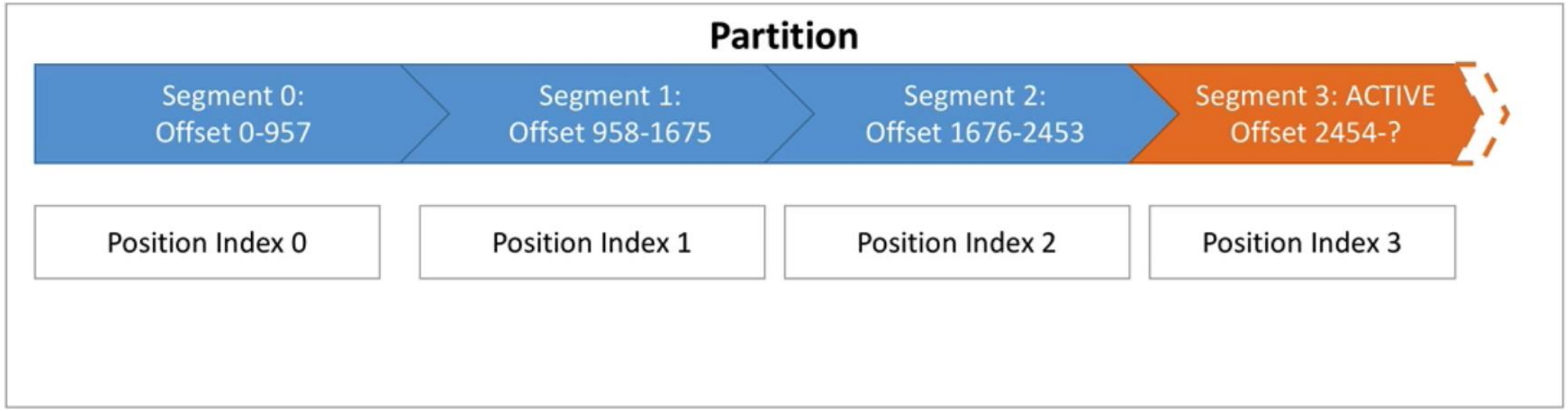
- Topics are made of partitions (we already know that)
- Partitions are made of… segments (files)!



- Only one segment is ACTIVE (the one data is being written to)
- Two segment settings:
  - log.segment.bytes: the max size of a single segment in bytes
  - log.segment.ms: the time Kafka will wait before committing the segment if not full

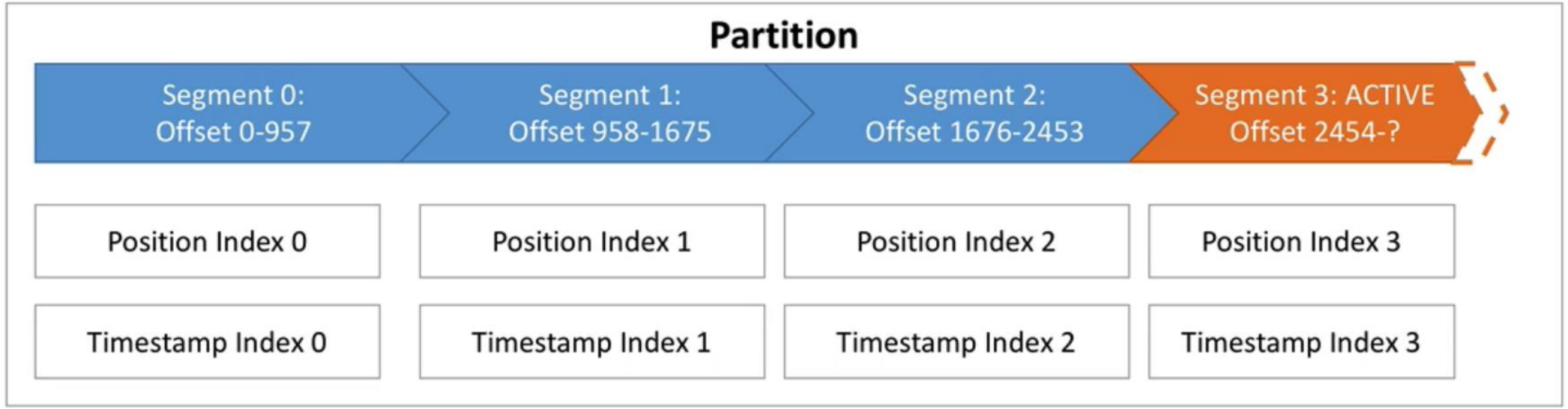
Segments and Indexes

- Segments come with two indexes (files):
  - An offset to position index: allows Kafka where to read to find a message
  - A timestamp to offset index: allows Kafka to find messages with a timestamp
- Therefore, Kafka knows where to find data in a <u>constant time</u>!



Segments and Indexes

- Segments come with two indexes (files):
  - An offset to position index: allows Kafka where to read to find a message
  - A timestamp to offset index: allows Kafka to find messages with a timestamp
- Therefore, Kafka knows where to find data in a constant time!



Segments: Why should I care?

- A smaller **log.segment.bytes** (size, default: 1 GB) means:

Segments: Why should I care?

- A smaller **log.segment.bytes** (size, default: 1GB) means:
  - More segments per partitions
  - Log Compaction happens more often
  - BUT Kafka has to keep more files opened (*Error: Too many open files*)

Segments: Why should I care?

- A smaller **log.segment.bytes** (size, default: 1GB) means:
  - More segments per partitions
  - Log Compaction happens more often
  - BUT Kafka has to keep more files opened (*Error: Too many open files*)
- Ask yourself: how fast will I have new segments based on throughput?

- A smaller **log.segment.ms** (time, default 1 week) means:
  - You set a max frequency for log compaction (more frequent triggers)
  - Maybe you want daily compaction instead of weekly?
- Ask yourself: how often do I need log compaction to happen?

Log Cleanup Policies

- Many Kafka clusters make data expire, according to a policy

Log Cleanup Policies

- Many Kafka clusters make data expire, according to a policy
- That concept is called log cleanup.

Policy1: log.cleanup.policy=delete (Kafka default for all user topics)

- Delete based on age of data (default is a week)

Log Cleanup Policies

- Many Kafka clusters make data expire, according to a policy
- That concept is called log cleanup.

Policy1: log.cleanup.policy=delete (Kafka default for all user topics)

- Delete based on age of data (default is a week)
- Delete based on max size of log (default is -1 == infinite)

Policy 2: log.cleanup.policy=compact (Kafka default for topic *__consumer_offsets*)

- Delete based on keys of your messages

Log Cleanup Policies

- Many Kafka clusters make data expire, according to a policy
- That concept is called log cleanup.

Policy1: log.cleanup.policy=delete (Kafka default for all user topics)

- Delete based on age of data (default is a week)
- Delete based on max size of log (default is -1 == infinite)

Policy 2: log.cleanup.policy=compact (Kafka default for topic *__consumer_offsets*)

- Delete based on keys of your messages
- Will delete old duplicate keys **after** the active segment is committed
- Infinite time and space retention

Log Cleanup: Why and When?

- Deleting data from Kafka allows you to:
  - Control the size of the data on the disk, delete obsolete data
  - Overall: Limit maintenance work on the Kafka Cluster

- How often does log cleanup happen?
  - Log cleanup happens on your partition segments!
  - Smaller / More segments means that log cleanup will happen more often!
  - Log cleanup shouldn't happen too often => takes CPU and RAM resources

Log Cleanup: Why and When?

- Deleting data from Kafka allows you to:
  - Control the size of the data on the disk, delete obsolete data
  - Overall: Limit maintenance work on the Kafka Cluster

- How often does log cleanup happen?
  - Log cleanup happens on your partition segments!
  - Smaller / More segments means that log cleanup will happen more often!
  - Log cleanup shouldn't happen too often => takes CPU and RAM resources
  - The cleaner checks for work every 15 seconds (log.cleaner.backoff.ms)

Log Cleanup Policy: Delete

- log.retention.hours:
  - number of hours to keep data for (default is 168 – one week)
  - Higher number means more disk space
  - Lower number means that less data is retained (your consumers may need to replay more data than less)

- log.retention.bytes:
  - Max size in Bytes for each partition (default is -1 – infinite)

Log Cleanup Policy: Delete

- log.retention.hours:
  - number of hours to keep data for (default is 168 – one week)
  - Higher number means more disk space
  - Lower number means that less data is retained (your consumers may need to replay more data than less)

- log.retention.bytes:
  - Max size in Bytes for each partition (default is -1 – infinite)
  - Useful to keep the size of a log under a threshold

Log Cleanup Policy: Delete

Old segment will be deleted based on time or space rules

New data is written to the active segment

Log Cleanup Policy: Delete

Old segment will be deleted based on time or space rules

New data is written to the active segment

Use cases - two common pair of options:

- <u>One week of retention:</u>
  - log.retention.hours=168 and log.retention.bytes=-1
- <u>Infinite time retention bounded by 500MB:</u>
  - log.retention.hours=17520 and log.retention.bytes=524288000

Log Cleanup Policy: Compact

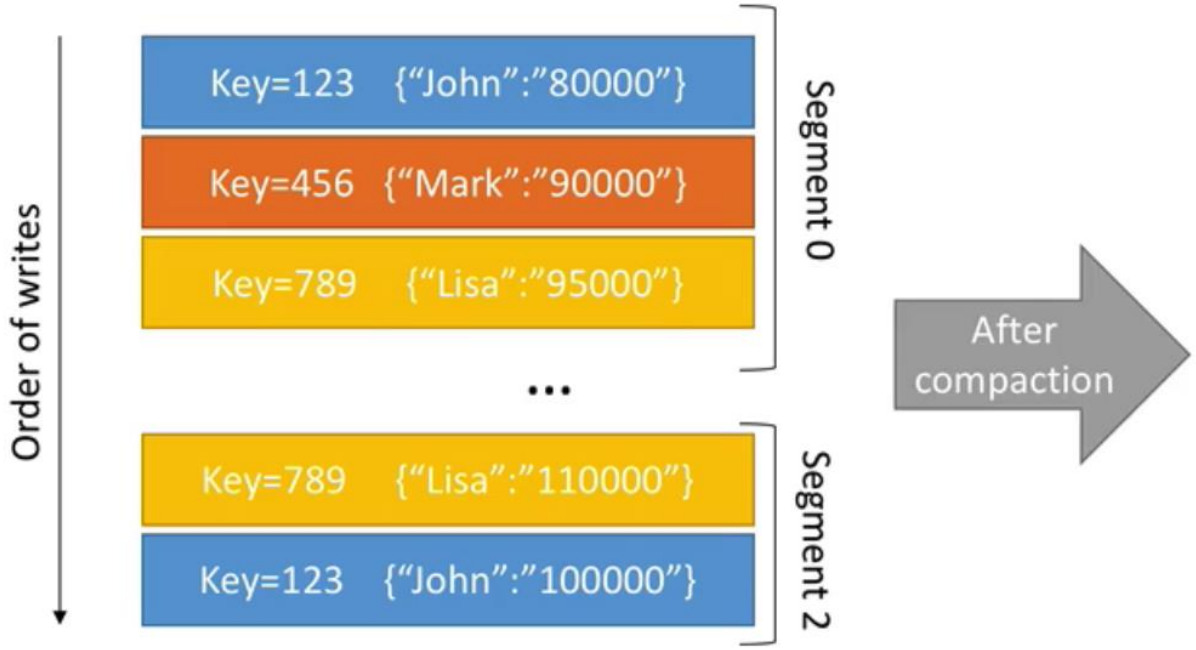
- Log compaction ensures that your log contains at least the last known value for a specific key within a partition
- Very useful if we just require a SNAPSHOT instead of full history (such as for a data table in a database)
- The idea is that we only keep the latest "update" for a key in our log

Log Compaction: Example

- Our topic is: employee-salary
- We want to keep the most recent salary for our employees

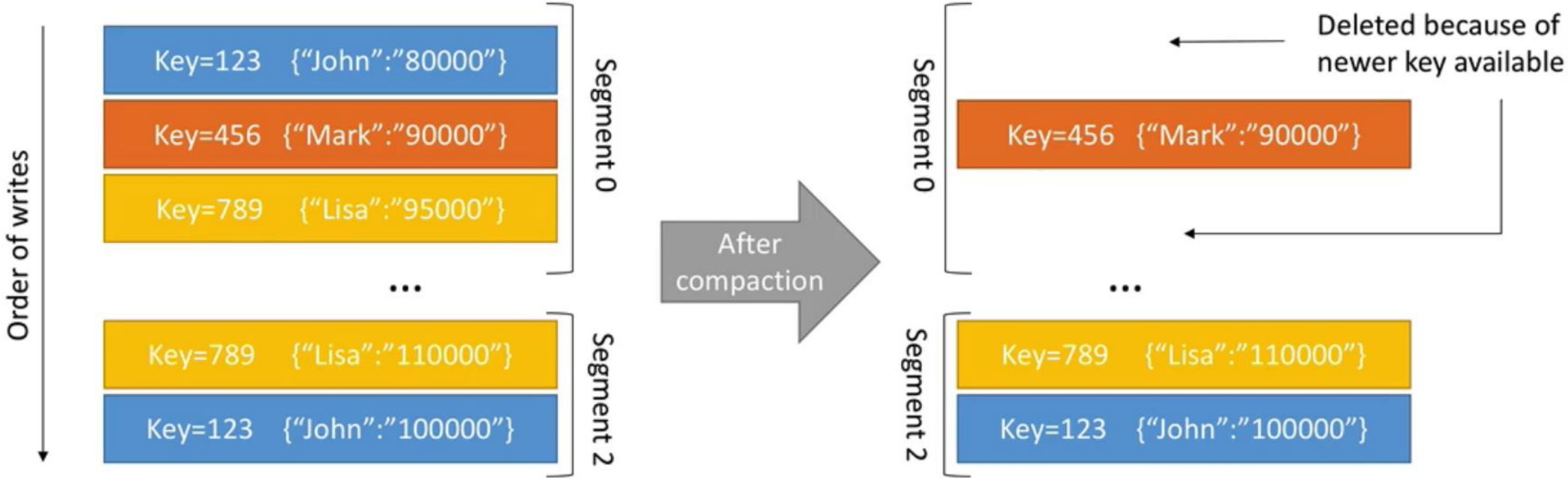
Log Compaction: Example

- Our topic is: employee-salary
- We want to keep the most recent salary for our employees



Log Compaction: Example

- Our topic is: employee-salary
- We want to keep the most recent salary for our employees



Log Compaction Guarantees

- **Any consumer that is reading from the head of a log will still see all the messages sent to the topic**
- Ordering of messages it kept, log compaction only removes some messages, but does not re-order them
- The offset of a message is immutable (it never changes). Offsets are just skipped if a message is missing
- Deleted records can still be seen by consumers for a period of delete.retention.ms (default is 24 hours).

Log Compaction Myth Busting

- It doesn't prevent you from pushing duplicate data to Kafka
  - De-duplication is done after a segment is committed
  - Your consumers will still read from head as soon as the data arrives

- It doesn't prevent you from reading duplicate data from Kafka
  - Same points as above

Log Compaction Myth Busting

- It doesn't prevent you from pushing duplicate data to Kafka
  - De-duplication is done after a segment is committed
  - Your consumers will still read from head as soon as the data arrives
- It doesn't prevent you from reading duplicate data from Kafka
  - Same points as above
- Log Compaction can fail from time to time
  - It is an optimization and it the compaction thread might crash
  - Make sure you assign enough memory to it and that it gets triggered

Log Compression

- Topics can be compressed using compression.type setting.
- Options are 'gzip', 'snappy', 'lz4', 'uncompressed', 'producer'
- If you need compression, ideally you keep default as 'producer'.
  - The producer will perform the compression on its side
  - The broker will take the data as is => Saves CPU resources on the broker
- If compression is enabled, make sure you're sending batches of data
- Data will be uncompressed by the consumer!
- Compression only makes sense if you're sending non-binary data (compress: JSON, text… | don't compress: Parquet, Protobuf , Avro)

What's next?

- Get familiar with the Kafka API
  - Write your own consumers
  - Write your own producers
  - Use the command line as much as possible