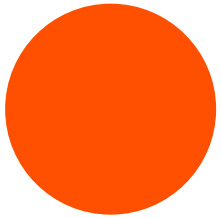
CRITEO

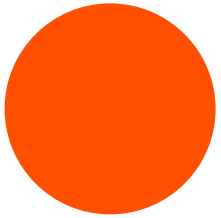
Kafka Internals

Ilyas Toumlilt
i.toumlilt@criteo.com

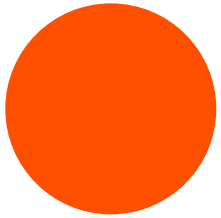
Session 4 – 29/05/2024



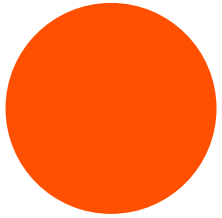
Session 3 Recap



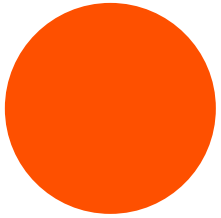
Hands on!
Getting started to Kafka



Kafka Internals



☕☕☕☕☕



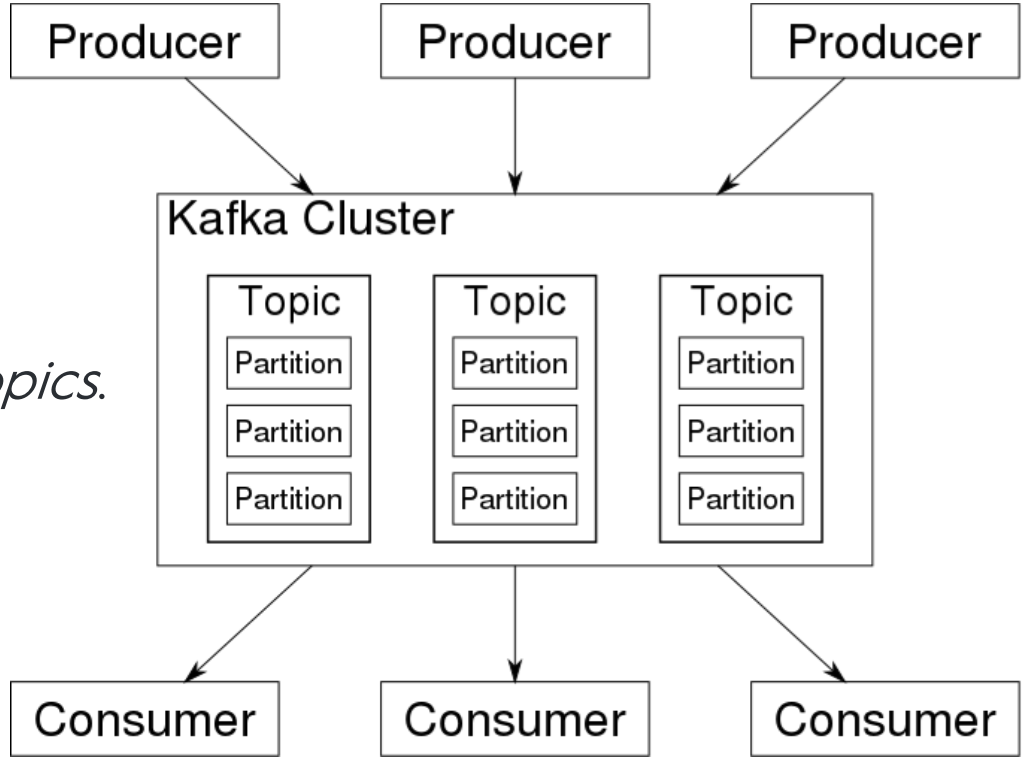
Kafka Lab

Session 3 Recap

CRITEO

Anatomy of Kafka

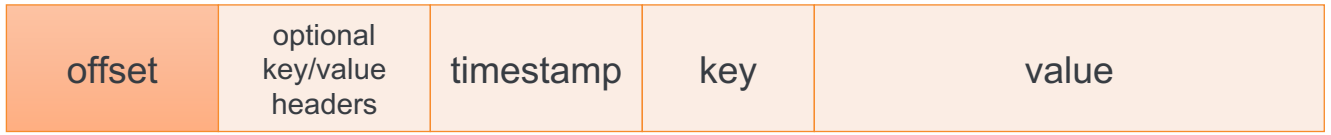
- Message is the unit of data.
- Producer creates a new message
- Consumer (i.e., Subscribers/Readers)
- Messages in Kafka are categorized into *topics*.
- Topics are broken down into a *partitions*.
- Messages are persistant,
- and written in an append-only fashion.



CRITEO

Message

- The unit of data within Kafka is a *message.*
- A message is simply an array of bytes.
- A message can have an optional *key* (also an array of bytes).
- Messages contain a timestamp and optional headers.
- Messages are stored in order within a topic-partition.
- Once in Kafka topic, a message has an offset.
- For efficiency, messages are written into Kafka in batches(collection of messages).



CRITEO

Topics and Partitions

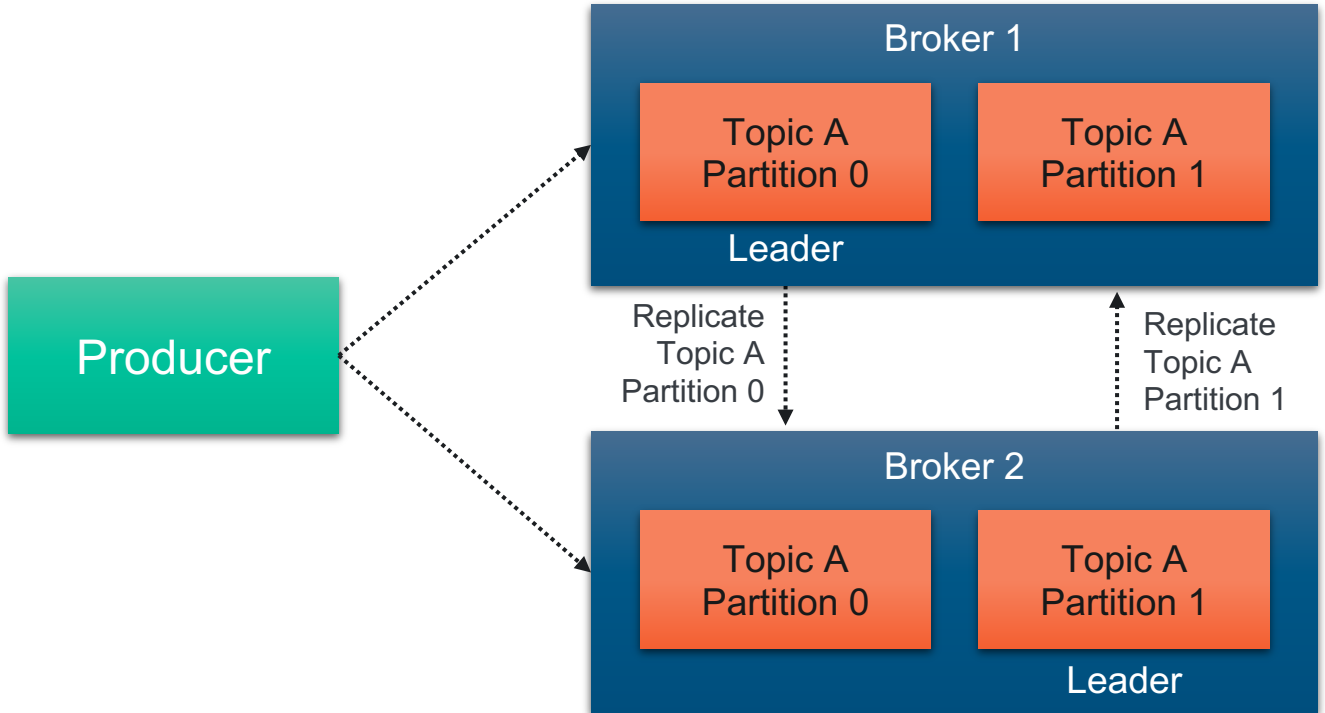
- Partitions are replicated across the cluster
- Each partition has one leader and zero or more server follower servers

```
bin/kafka-topics.sh --describe --bootstrap-server=localhost:9092 --topic xxx_yyy_zzz
```

Topic : xxx_yyy_zzz	Partition: 0	Leader: 19	Replicas: 19,20,17	ISR: 19
Topic : xxx_yyy_zzz	Partition: 1	Leader: 20	Replicas: 20,17,18	ISR: 18, 17, 20
Topic : xxx_yyy_zzz	Partition: 2	Leader: 17	Replicas: 17,18,16	ISR: 18, 16, 17
Topic : xxx_yyy_zzz	Partition: 3	Leader: 18	Replicas: 18, 16, 19	ISR: 18, 16, 19
Topic : xxx_yyy_zzz	Partition: 4	Leader: 16	Replicas: 16,19, 20	ISR: 16,19, 20

Kafka Cluster

- Kafka brokers are designed to operate as part of a cluster.
- A partition is owned by a single broker in the cluster (*leader*).
- A partition can be replicated to other brokers in the cluster (*replicas*)



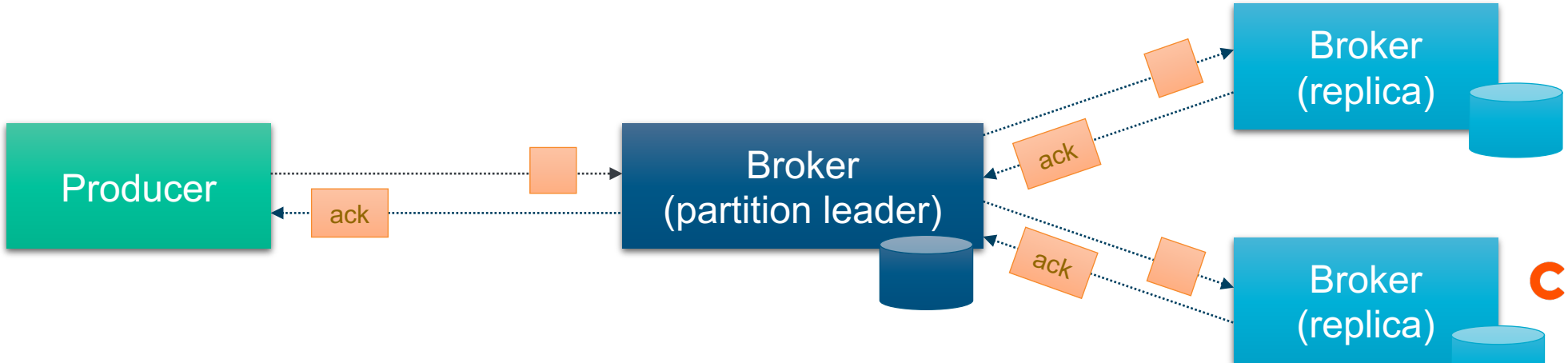
CRITEO

Kafka Producer

Producers decide to which partitions to send messages to.

Producers wait for the acknowledgment from the broker:

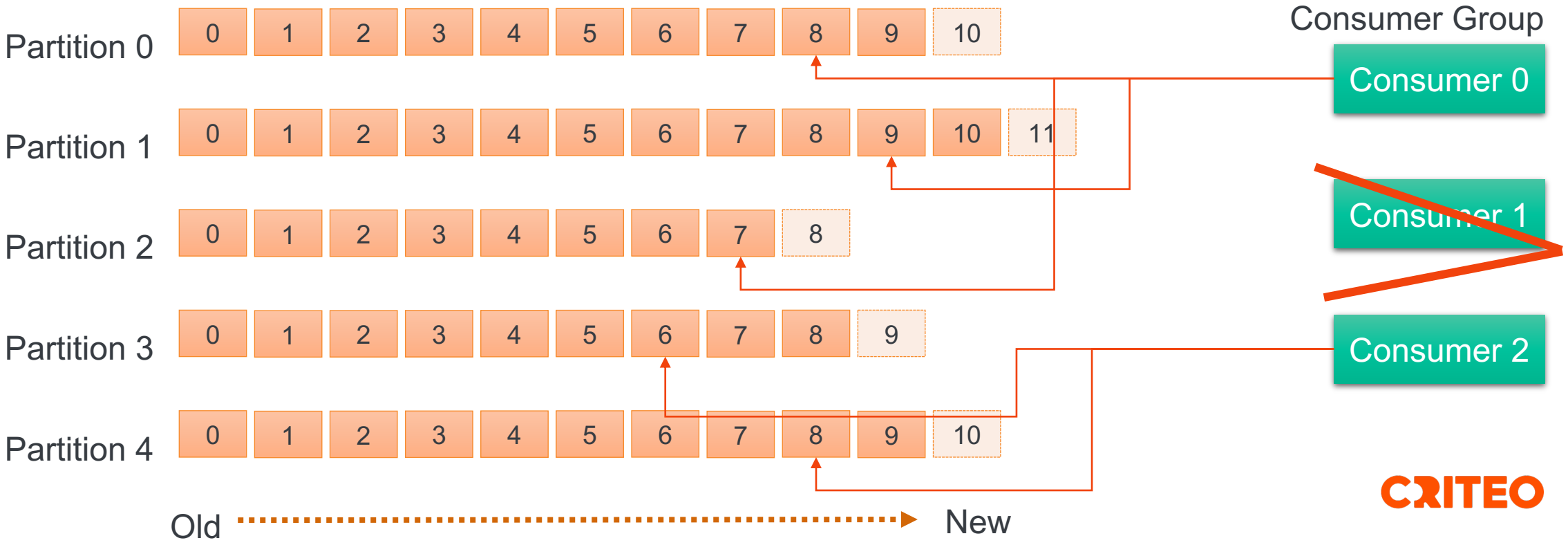
- ack=0
  - Doesn’t wait for confirmation from the partition leader broker.
- ack=1
  - Only waits for the acknowledgment of the leader broker.
- ack=all
  - Waits for the acknowledgments of the leader broker and all replicas.



CRITEO

Consumer Rebalance

- If a consumer is removed from the consumer group, the partitions from that consumer will be assigned to the remaining consumers.
- If a new consumer is added to the consumer group, it will receive partitions from other consumers, keeping a good balance.



Zookeeper

- Apache Zookeper is a distributed coordination service for distributed applications.
- Zookeeper provides some guarantees:
  - Sequential Consistency
  - Atomicity
  - Single System Image
  - Reliability
  - Timeliness
- Kafka uses Apache Zookeeper to keep the cluster state:
  - List of currently members of a cluster;
  - List of topics, partitions leaders and replicas;
  - Current controller; etc.
- **Deprecated since Kafka 3.0.0, not required anymore**

CRITEO

Cap Theorem in context of Kafka

CAP Theorem(published in PODC ‘02) states that any distributed system can provide at most two out of the three guarantees: Consistency, Availability and Partition tolerance.

Partition tolerance**(*necessity)**

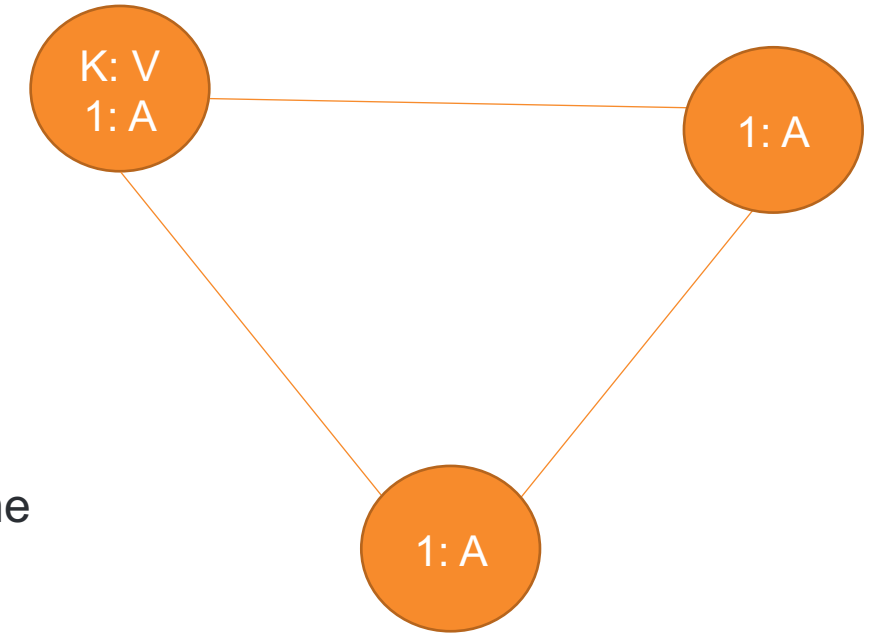
- The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

Consistency

- Every read receives the most recent write or an error

Availability

- Every request receives a (non-error) response, without the guarantee that it contains the most recent write



CRITEO

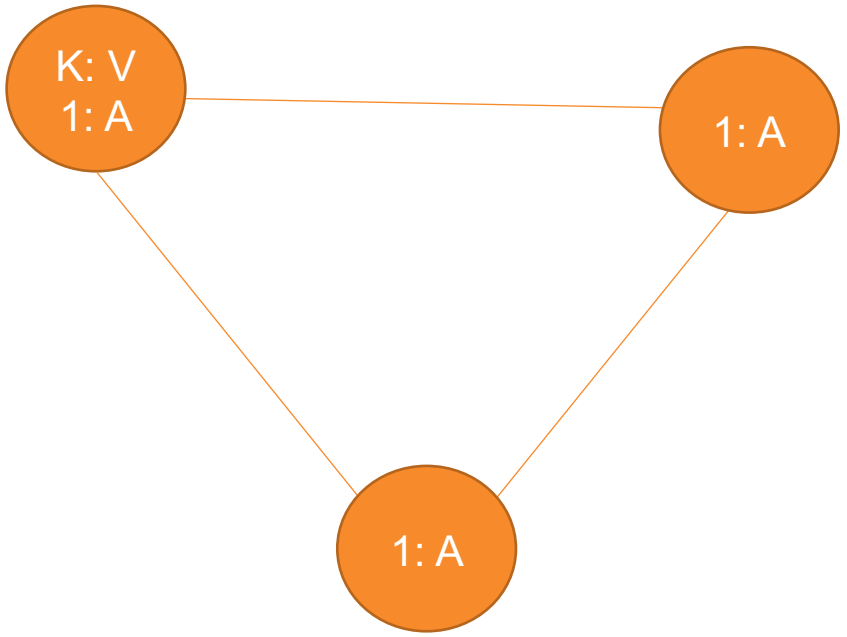
Cap Theorem on Kafka Brokers

For Consistency:

- Producers acks=all
- Replicator factor >= 3
- Disable unclean leader election
- Min insync replicas=replicatorFactor -1

For Availability:

- Enable unclean leader election
- Min insync replicas=1



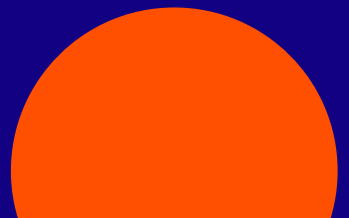
CRITEO

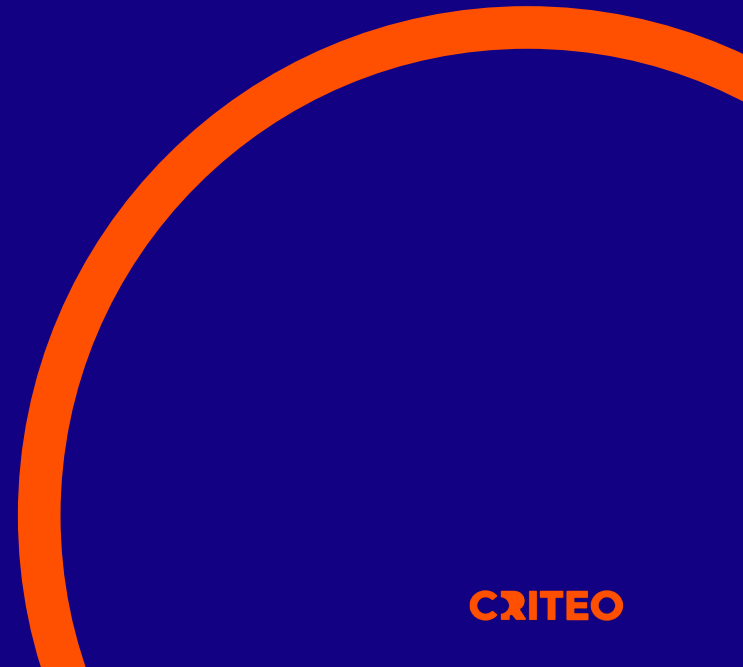
Practice time: Kafka Getting Started

Do you want a demo?

CRITEO

Kafka Internals





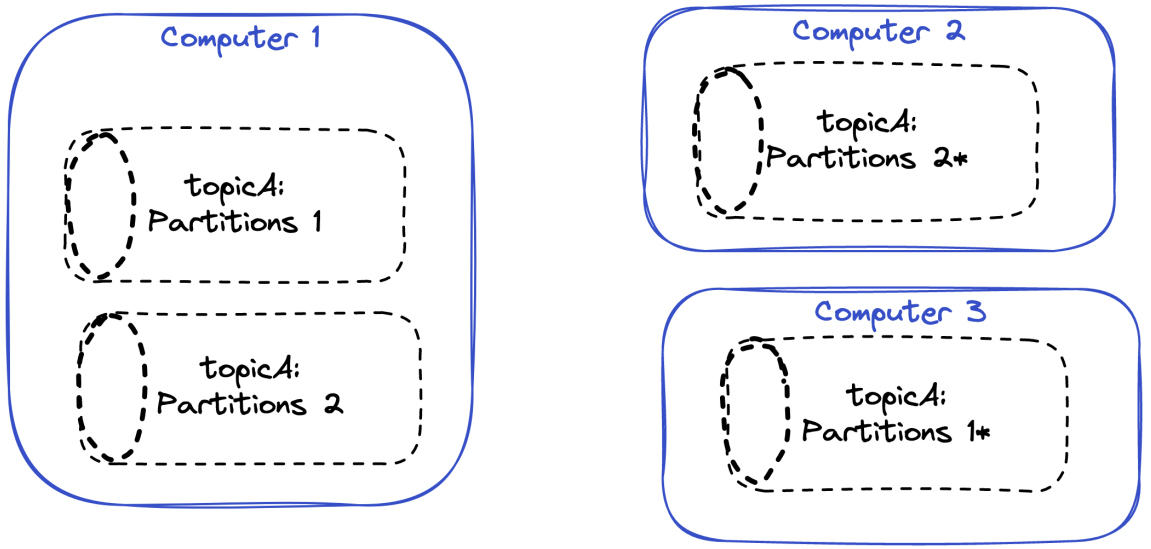
CRITEO

How can Kafka scale ? (server-side)

Vertical scaling ? - *Get a bigger computer*

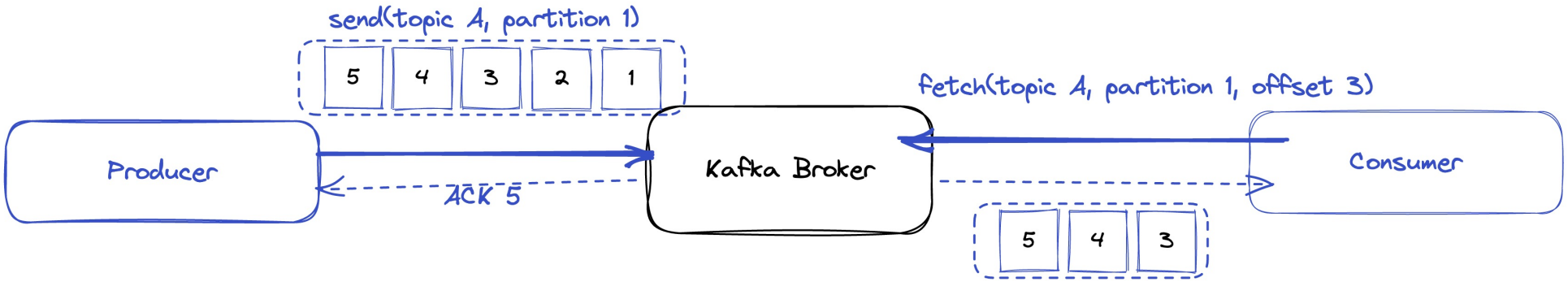
Horizontal scaling ?

- Sharding(called partitions)
- Replication Factor



*asterisks sign shows it is the leader
RF(Replication Factor): 2

How can Kafka scale ? (clients-side)



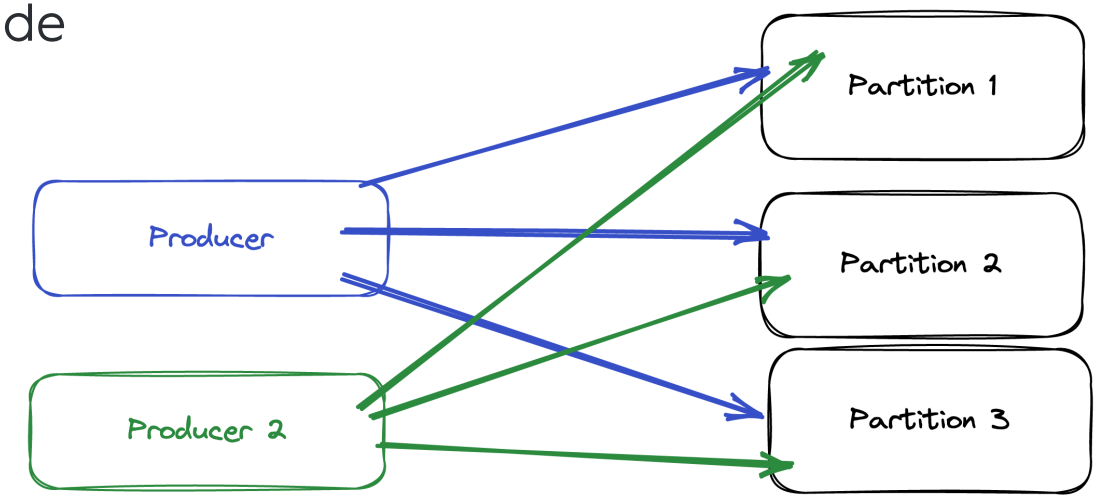
- Multiple producers
- Multiple consumer with different consumer group doesn't share workload(partitions)
- Multiple consumer with the same consumer group
  - Each consumer in same group can assign at least one partitions

    N(consumer_in_same_group) < N(partition_topic)

How can Kafka scale ? (clients-side)

- Many producers yes but how do they decide which partition(in same topic) to write ?

Problem: how do we balance each partition ?



- By default round-robin fashion.
- Key partitioning:

  partition = hash(key) % number_of_partition

  Key could be something like country=France

  - What happens if partition increased ?
- Not good enough: customise your own partition(by extending the partition class).

CRITEO

Kafka producer-side configs

- Producer ACK: The number of acknowledgments the producer requires the leader to have received before considering a request complete. This affects durability. (Options: acks=0, acks=1, acks=-1(all))

- Producer Idempotency(enable.idempotence = true):

  When set to 'true', the producer will ensure that exactly one copy of each message is written in the stream.

  If 'false', producer retries due to broker failures, etc., may write duplicates of the retried message in the stream

How does Kafka provide fault-tolerance ?

```
he.ciritoglu@C02FV0JEQ6LR ~ % kafka-topics.sh --describe --bootstrap-server $kafka_server:9092 --topic test_csharp_driver
Topic: test_csharp_driver       PartitionCount: 10      ReplicationFactor: 3    Configs: min.insync.replicas=2,segment.bytes=1073741824,max.message.bytes=10000000
type=LogAppendTime,unclean.leader.election.enable=false,retention.bytes=386547056640
        Topic: test_csharp_driver       Partition: 0    Leader: 34      Replicas: 34,37,33      Isr: 37,33,34
        Topic: test_csharp_driver       Partition: 1    Leader: 41      Replicas: 41,42,34      Isr: 41,42,34
        Topic: test_csharp_driver       Partition: 2    Leader: 33      Replicas: 33,38,34      Isr: 38,33,34
        Topic: test_csharp_driver       Partition: 3    Leader: 39      Replicas: 39,42,41      Isr: 41,42,39
        Topic: test_csharp_driver       Partition: 4    Leader: 33      Replicas: 33,34,37      Isr: 33,37,34
        Topic: test_csharp_driver       Partition: 5    Leader: 39      Replicas: 39,34,41      Isr: 41,39,34
        Topic: test_csharp_driver       Partition: 6    Leader: 42      Replicas: 42,34,33      Isr: 42,33,34
        Topic: test_csharp_driver       Partition: 7    Leader: 41      Replicas: 41,34,38      Isr: 38,41,34
```

- Hint: always use kafka-topics.sh --describe to see topic details

How does Kafka provide fault-tolerance ?

- Each message stored replicated factor(RF) times.
- Trusting replicas in case of data corruption / server crash.
- Kafka disk writes are asynchronous.

```
he.ciritoglu@C02FV0JEQ6LR ~ % kafka-topics.sh --describe --bootstrap-server $kafka_server:9092 --topic test_csharp_driver
Topic: test_csharp_driver       PartitionCount: 10      ReplicationFactor: 3    Configs: min.insync.replicas=2,segment.bytes=1073741824,max.message.bytes=10000000,
type=LogAppendTime,unclean.leader.election.enable=false,retention.bytes=386547056640
        Topic: test_csharp_driver       Partition: 0    Leader: 34      Replicas: 34,37,33      Isr: 37,33,34
        Topic: test_csharp_driver       Partition: 1    Leader: 41      Replicas: 41,42,34      Isr: 41,42,34
        Topic: test_csharp_driver       Partition: 2    Leader: 33      Replicas: 33,38,34      Isr: 38,33,34
        Topic: test_csharp_driver       Partition: 3    Leader: 39      Replicas: 39,42,41      Isr: 41,42,39
        Topic: test_csharp_driver       Partition: 4    Leader: 33      Replicas: 33,34,37      Isr: 33,37,34
        Topic: test_csharp_driver       Partition: 5    Leader: 39      Replicas: 39,34,41      Isr: 41,39,34
        Topic: test_csharp_driver       Partition: 6    Leader: 42      Replicas: 42,34,33      Isr: 42,33,34
        Topic: test_csharp_driver       Partition: 7    Leader: 41      Replicas: 41,34,38      Isr: 38,41,34
        Topic: test_csharp_driver       Partition: 8    Leader: 34      Replicas: 37,33,34      Isr: 33,37,34
        Topic: test_csharp_driver       Partition: 9    Leader: 2       Replicas: 2,41,34       Isr: 41,2,34
```

How does Kafka provide fault-tolerance ?

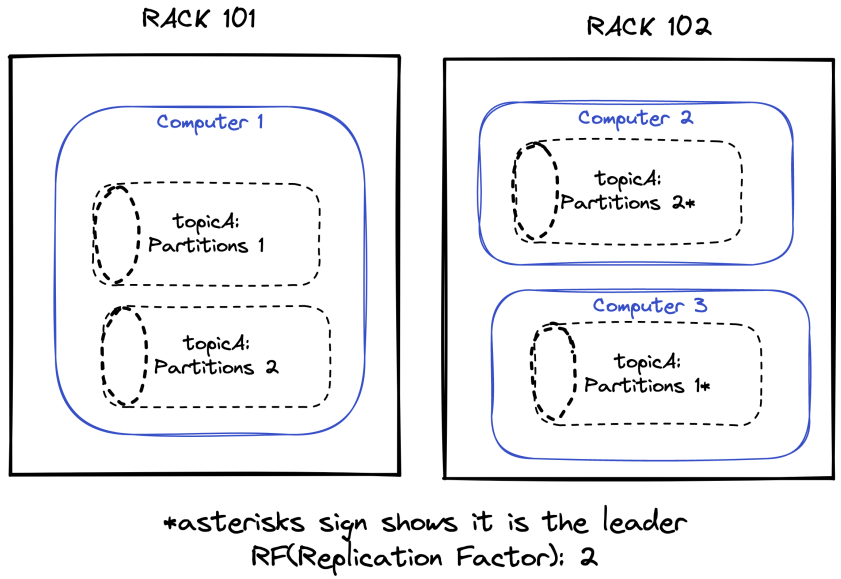
- Messages are always sent to (and consumed from) partition leader.
- Leaders replicate messages to brokers replicas synchronously.
- ISR: simply all the replicas of a partition that are "in-sync" with the leader.
  - min.insync.replicas: the number of replicas that have to be in sync for the broker to accept writes for the partition

```
he.ciritoglu@C02FV0JEQ6LR ~ % kafka-topics.sh --describe --bootstrap-server $kafka_server:9092 --topic test_csharp_driver
Topic: test_csharp_driver       PartitionCount: 10      ReplicationFactor: 3    Configs: min.insync.replicas=2,segment.bytes=1073741824,max.message.bytes=10000000,message.timestamp.
type=LogAppendTime,unclean.leader.election.enable=false,retention.bytes=386547056640
        Topic: test_csharp_driver       Partition: 0    Leader: 34      Replicas: 34,37,33      Isr: 37,33,34
        Topic: test_csharp_driver       Partition: 1    Leader: 41      Replicas: 41,42,34      Isr: 41,42,34
        Topic: test_csharp_driver       Partition: 2    Leader: 33      Replicas: 33,38,34      Isr: 38,33,34
        Topic: test_csharp_driver       Partition: 3    Leader: 39      Replicas: 39,42,41      Isr: 41,42,39
        Topic: test_csharp_driver       Partition: 4    Leader: 33      Replicas: 33,34,37      Isr: 33,37,34
        Topic: test_csharp_driver       Partition: 5    Leader: 39      Replicas: 39,34,41      Isr: 41,39,34
        Topic: test_csharp_driver       Partition: 6    Leader: 42      Replicas: 42,34,33      Isr: 42,33,34
        Topic: test_csharp_driver       Partition: 7    Leader: 41      Replicas: 41,34,38      Isr: 38,41,34
        Topic: test_csharp_driver       Partition: 8    Leader: 34      Replicas: 37,33,34      Isr: 33,37,34
```

How does Kafka provide fault-tolerance ?

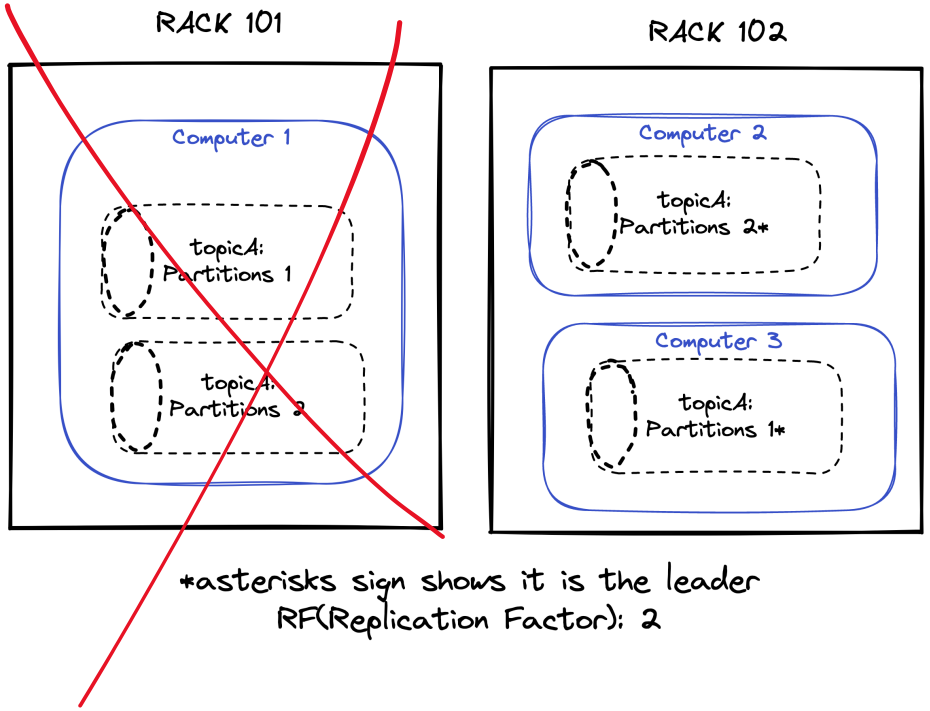
- A broker losses connectivity zookeper, it is consider out of sync with the cluster. Such case the new leader needs to be selected.

- Rack-awareness: provides fault tolerance in that if a rack goes down, the remaining racks can continue to serve traffic.



What happens if a broker is down

- Underreplicated metrics will be increased
- Min ISR >= N(Alive replicas)
  - no more new data will be inserted.
- Min ISR < N(Alive replicas)
  - leader change: no impact



Min ISR: will be determined by "*min*.insync.replicas"

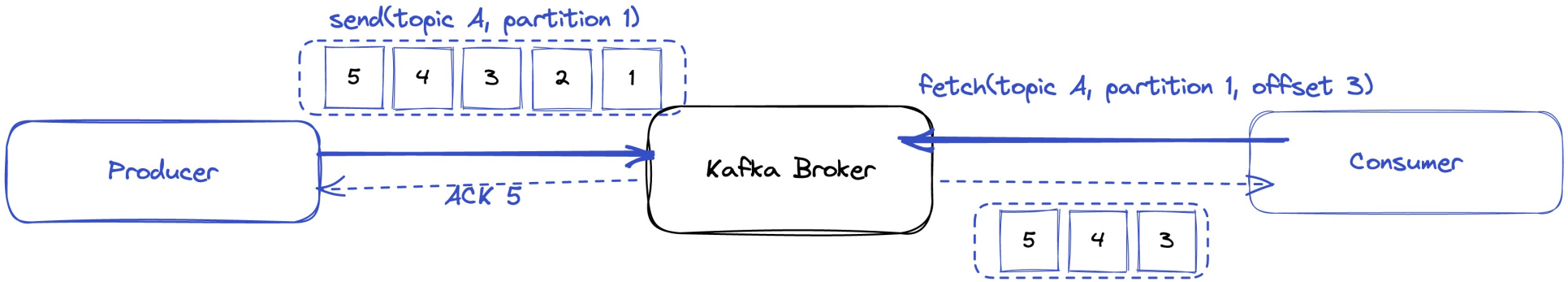
Kafka leader election

```
he.ciritoglu@C02FV0JEQ6LR ~ % kafka-topics.sh --describe --bootstrap-server $kafka_server:9092 --topic test_csharp_driver
Topic: test_csharp_driver       PartitionCount: 10      ReplicationFactor: 3    Configs: min.insync.replicas=2,segment.bytes=1073741824,max.message.bytes=10000000,message.timestamp.
type=LogAppendTime,unclean.leader.election.enable=false,retention.bytes=386547056640
        Topic: test_csharp_driver       Partition: 0    Leader: 34      Replicas: 34,37,33      Isr: 37,33,34
        Topic: test_csharp_driver       Partition: 1    Leader: 41      Replicas: 41,42,34      Isr: 41,42,34
        Topic: test_csharp_driver       Partition: 2    Leader: 33      Replicas: 33,38,34      Isr: 38,33,34
        Topic: test_csharp_driver       Partition: 3    Leader: 39      Replicas: 39,42,41      Isr: 41,42,39
        Topic: test_csharp_driver       Partition: 4    Leader: 33      Replicas: 33,34,37      Isr: 33,37,34
        Topic: test_csharp_driver       Partition: 5    Leader: 39      Replicas: 39,34,41      Isr: 41,39,34
        Topic: test_csharp_driver       Partition: 6    Leader: 42      Replicas: 42,34,33      Isr: 42,33,34
        Topic: test_csharp_driver       Partition: 7    Leader: 41      Replicas: 41,34,38      Isr: 38,41,34
```

- Election of the new the leader(in case of the node is down)
  - *if* : there is not enough ISR > min.insync.replicas(e.g 2) **wait**
  - *else* : then one of the ISR(in-sync replicas) will be the new leader.
- The leader will be selected through the replica list.

How Kafka is super fast : Batching

- Producers send messages together in batches.
- Brokers acknowledge the last message within the batch.
- Consumers request messages after an offset.
- The broker will send the same batch of messages sent by the producer.



How Kafka is super fast : Batching

- Batch.size:
  - batch.size measures batch size in total bytes instead of the number of messages.
  - It controls how many bytes of data to collect before sending messages to the Kafka broker.
- Linger.ms:
  - Instead of sending immediately, you can introduce artificial delay as linger.ms
  - Idea: to reduce the number of requests sent by introducing a small delay, we can increase the throughput,

How Kafka is super fast : Compression

HIGHER THROUGPUT

- **When**: the bottleneck is not CPU nor disk but the **network bandwidth.**
- Efficient compression requires compressing multiple messages together rather than compressing each message individually.
- Kafka supports compression of a batch of messages: gzip,snappy,lz4,zstd
- The producer will compress a batch of messages.
- This batch of messages will be written in compressed form and will remain compressed in the log.
- Only the consumers will decompress the batch of messages.

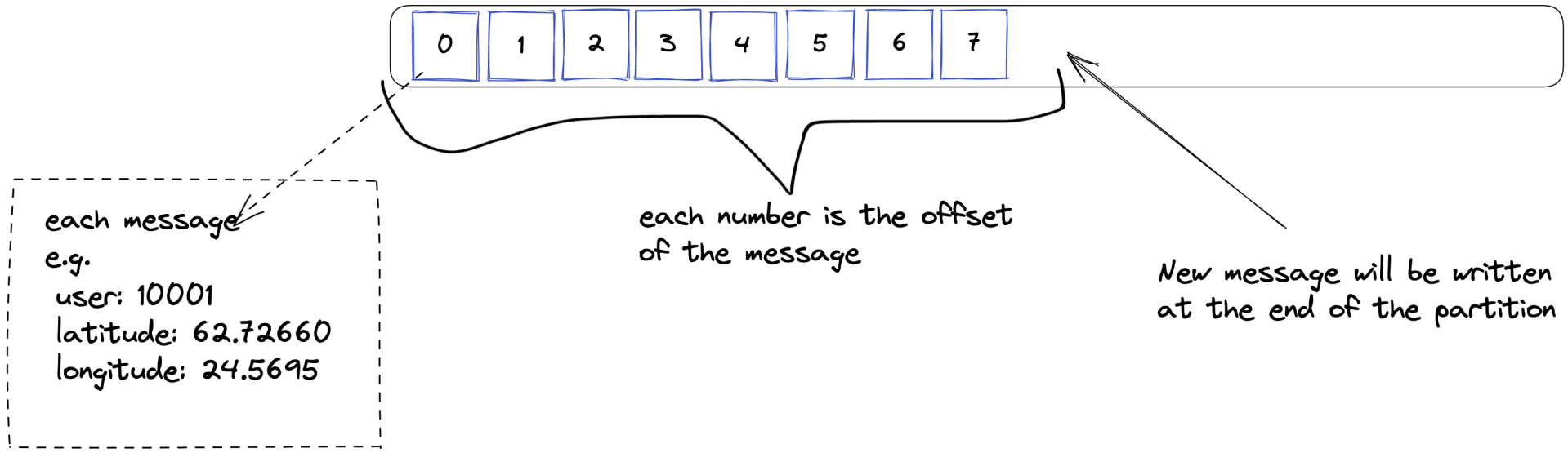
How Kafka is fast : OS Cache

- Kafka relies on native Linux Page cache (read-ahead and write-behind)
- JVM off-heap cache for free
- No serialisation/deserialisation cost on the broker
  - No Java object memory overhead
  - No OutOfMemory issue
  - No big GC pauses

How Kafka is fast : Append-only log

But I know disk access is slow ?

- Append/only + Immutability



- Provides sequential I/O(read/writes)
- Order guaranteed within the same partition

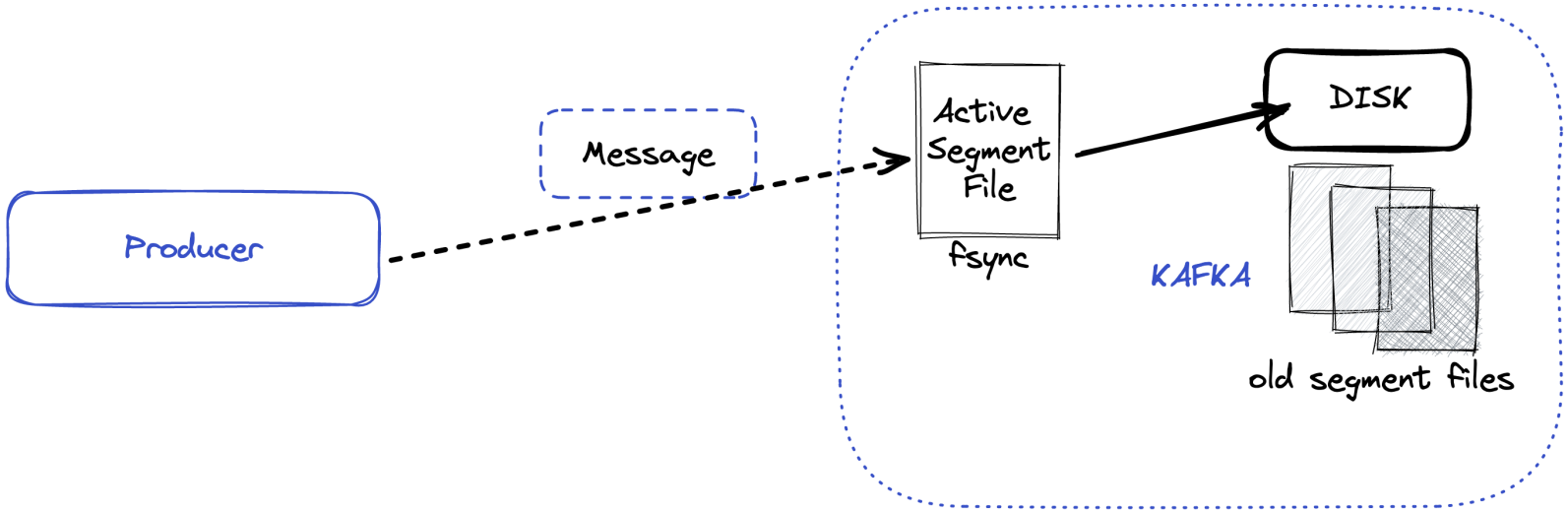
CRITEO

How does Kafka manage partitions ?

- Each Kafka partition is mapped to segment files.
- Segment file: log append structure.
- After a certain limit in size the segment is closed and a new one is opened.
- Records are immutable.
- Broker does very few random disk search.

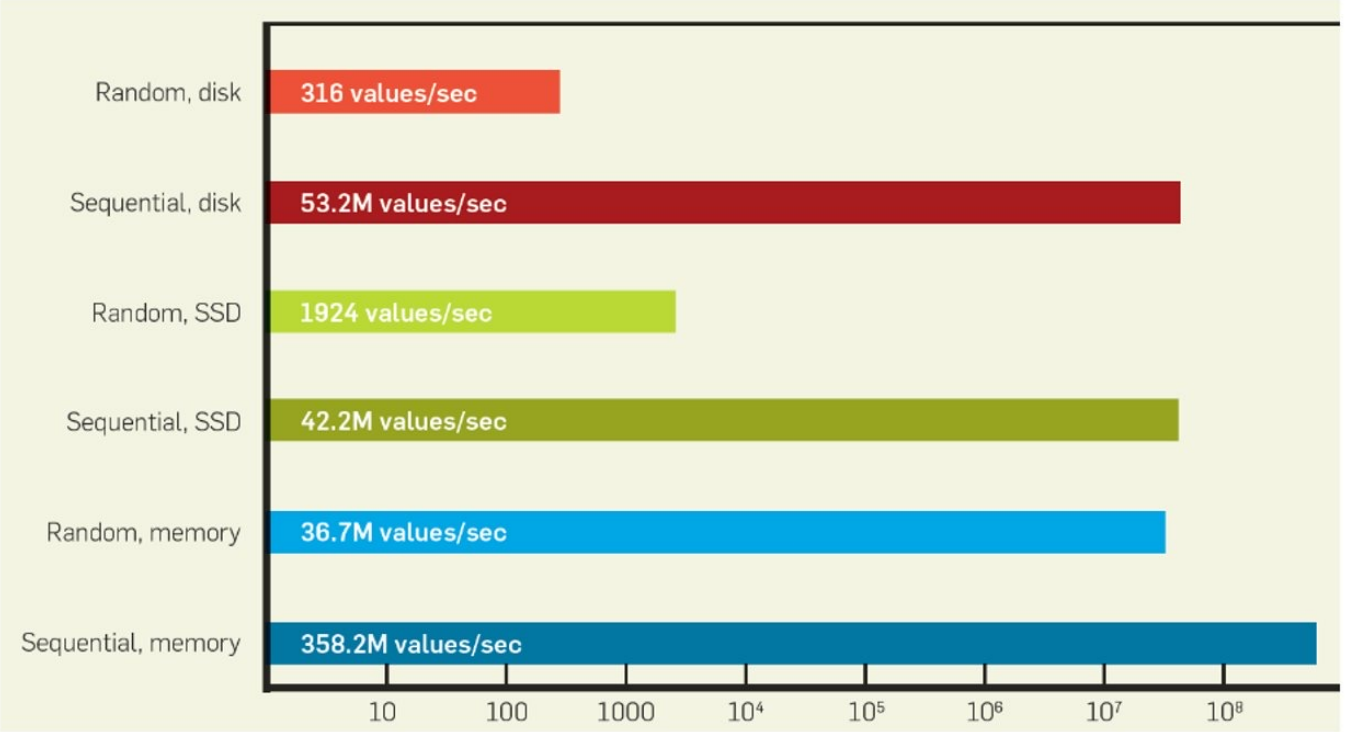
Kafka Log Segment

- Broker accumulates the messages in cache/buffer before flushing it to disk.
- log.segment.bytes determines the maximum size(in bytes) of a segment in the cluster.
- Within each segment, there are three files with the following extensions - .log, .index & .timeindex.



How Kafka is fast : Sequential I/O

How come mechanical disk support fast operations ?

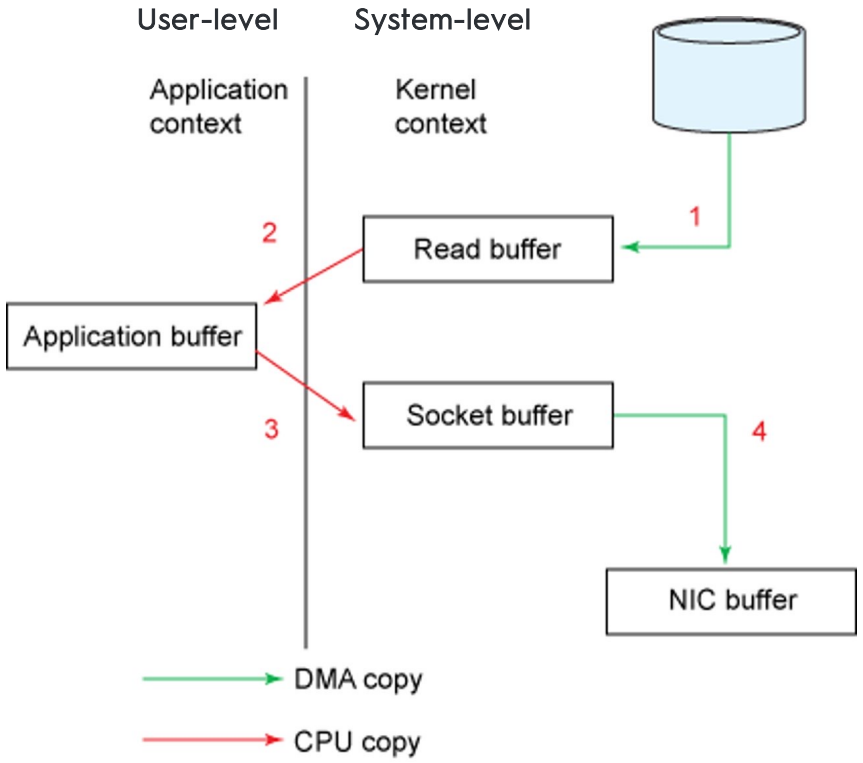


"Pathologies of Big Data" by Adam Jacobs in the ACM Communications, 2009

How Kafka is fast ?

Copying data from Disk

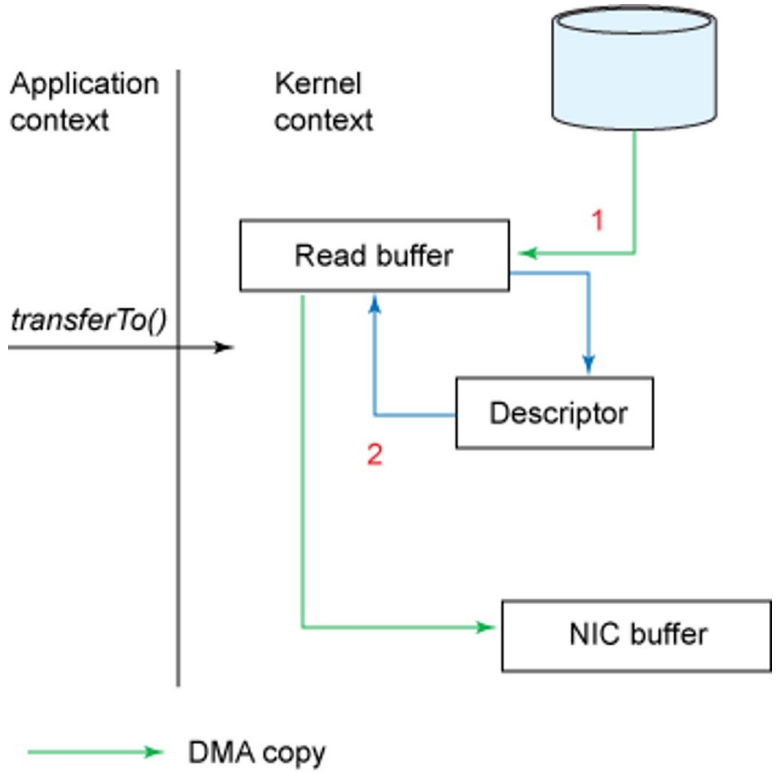
- In the traditional way, data is copied to 4 different buffers.
- Context switches from kernel and user space.



How Kafka is fast : Zero Copy Principal

Copying data from Disk

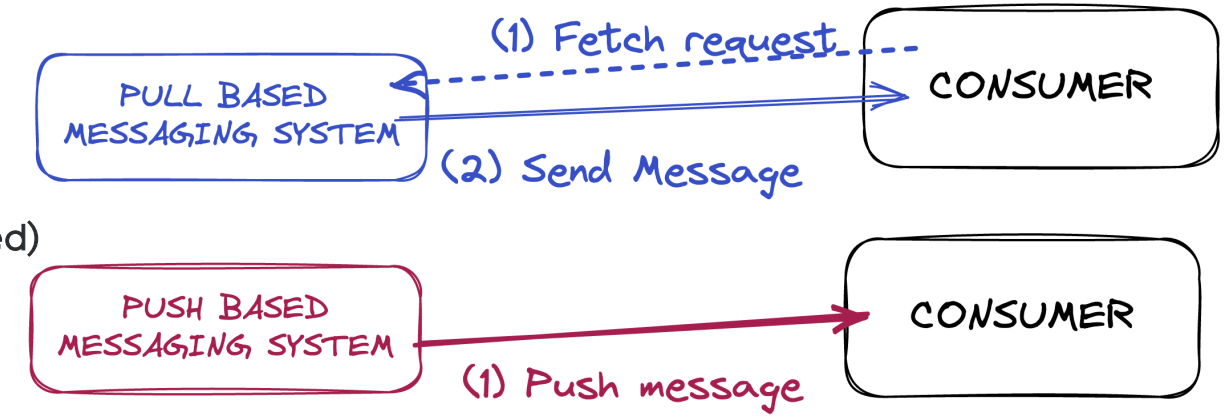
- With zero-copy only 2 buffers are used.
- No context switch is required.
- Java lang feature: **transferTo()**



Kafka's alternative

Understand your superpower

- Open source:
  - RabbitMQ (push-based)
  - Apache Pulsar (controlled push-based)



- Pure cloud:
  - Google Pub/Sub (can be configured as both pull and push)
  - AWS Kinesis (pull-based)

CRITEO

What about a break? ☕

CRITEO

Practice time