How Kafka Works

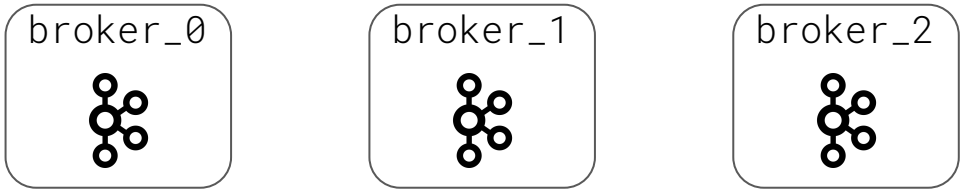
How Kafka Works

Kafka Runs Anywhere

- Kafka is written on the **JVM** in **Java** and **Scala**
- The JVM allows Kafka to target many OS architectures
- Kafka has a diverse ecosystem of tools and libraries
- Kafka's flexibility has made it pervasive in data work
- Kafka's few requirements make it easy to build on

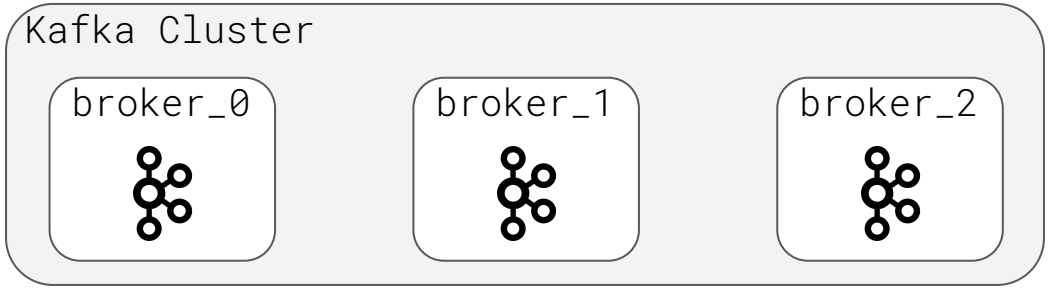
Kafka Clustering

Brokers are the building blocks of Kafka



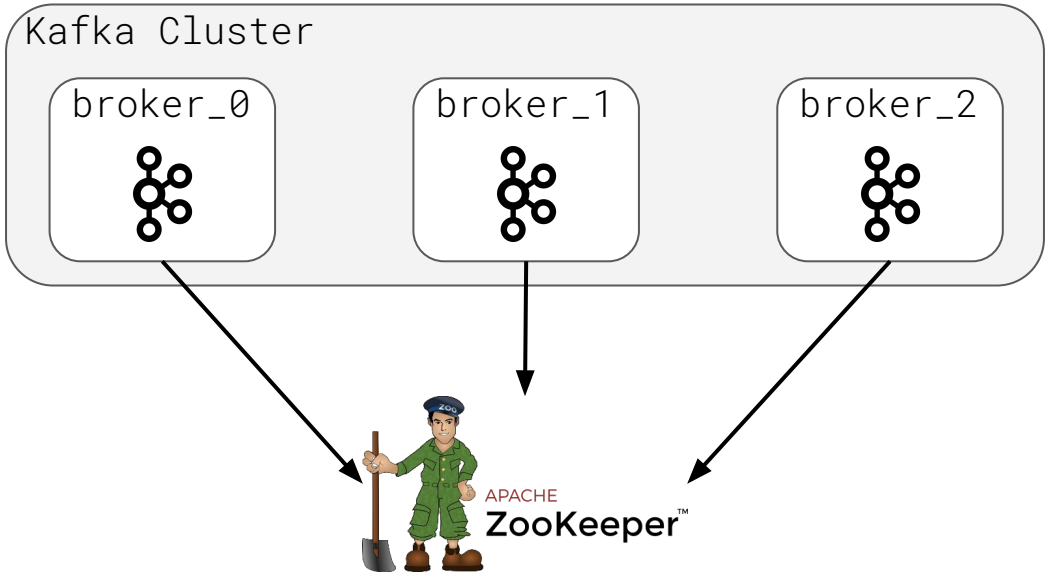
Kafka Clustering

Brokers are organized into **clusters**. Clusters contain 1 or more Brokers.



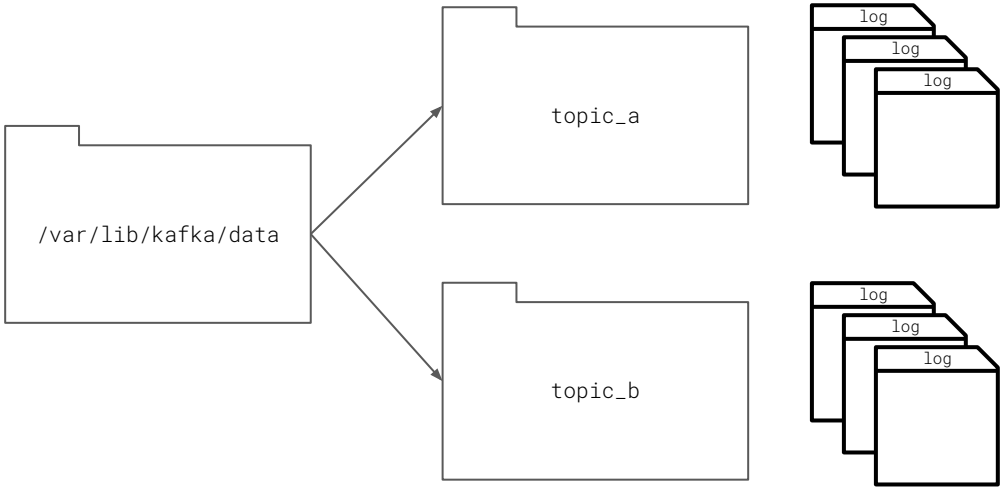
Kafka Clustering

Zookeeper is used to track cluster state, membership and leadership



How Kafka Stores Data

Kafka **stores** all of its **data** in a **directory on the broker disk**

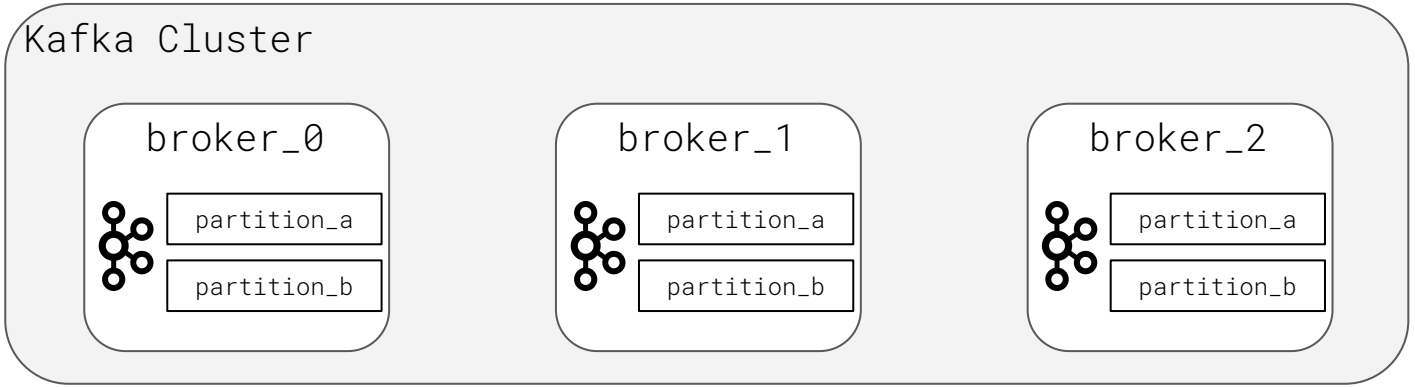


Data Partitions

- All Kafka topics consist of **one or more partitions**
- A partition contains a **strictly-ordered subset** of all of the data in the Topic
- Partitions enable Kafka to achieve **high throughput**

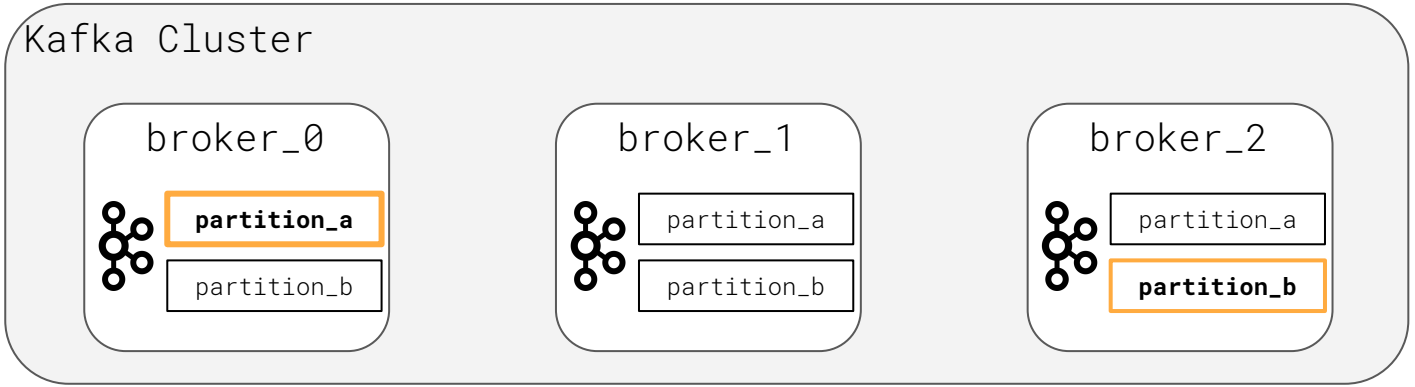
Data Partitions

Partitions are typically **distributed evenly** amongst brokers



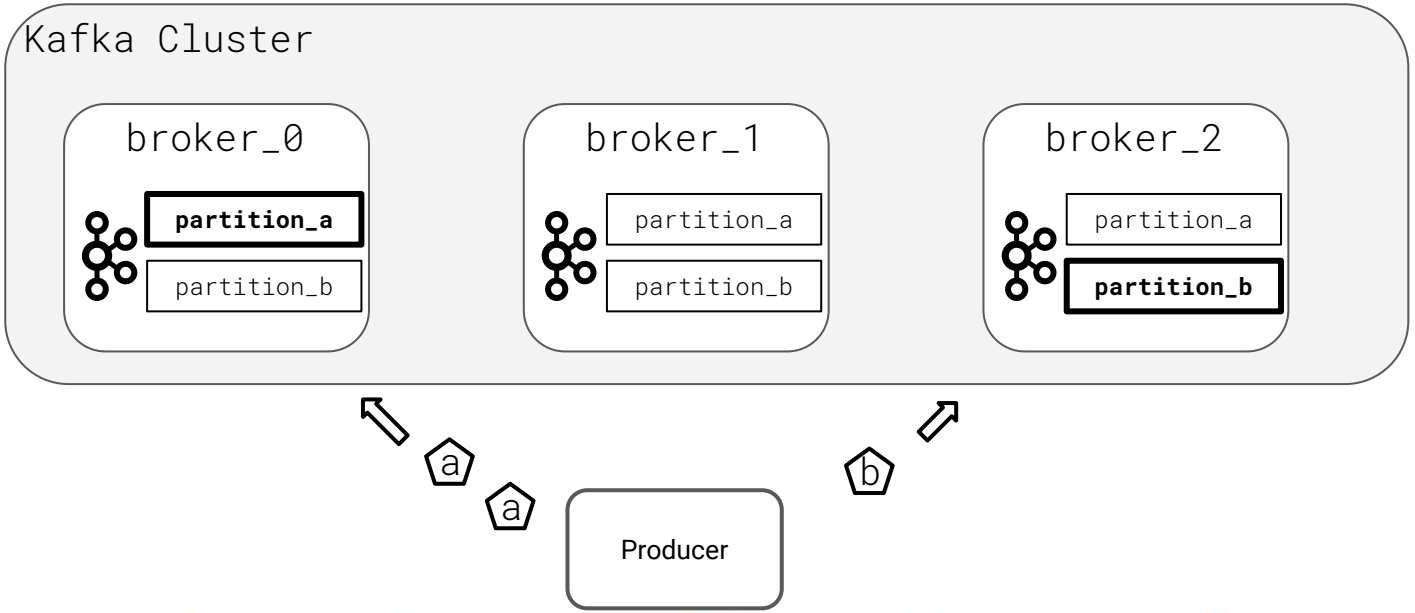
Data Partitions

Every partition has a single **leader broker**, elected with Zookeeper



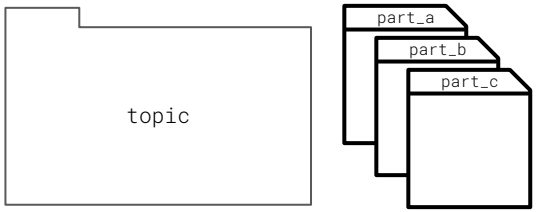
Data Partitions

Producers evenly **hash data** to partitions



Data Partitions

Key Takeaways



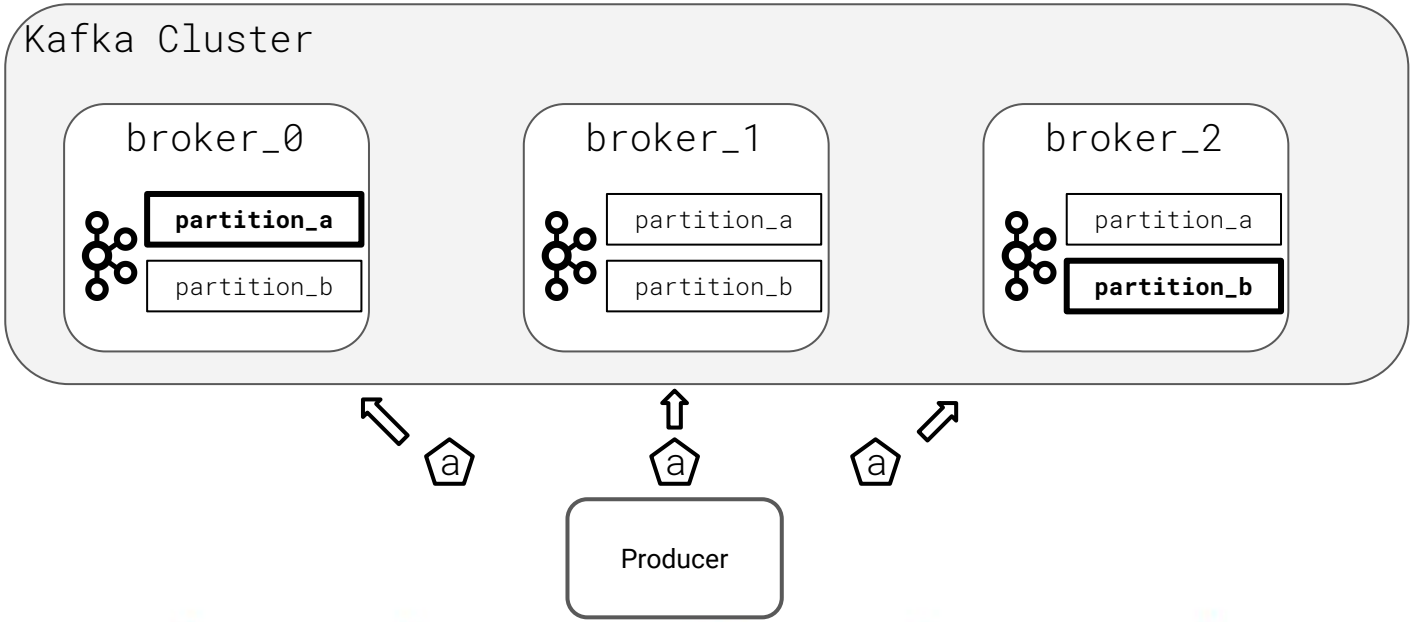
- Partitions are Kafka's unit of **parallelism**
- **Consumers** may **parallelize** data from each partition
- **Producers** may **parallelize** data from each partition
- **Reduces bottlenecks** by involving **multiple brokers**

Data Replication

Preventing Data Loss

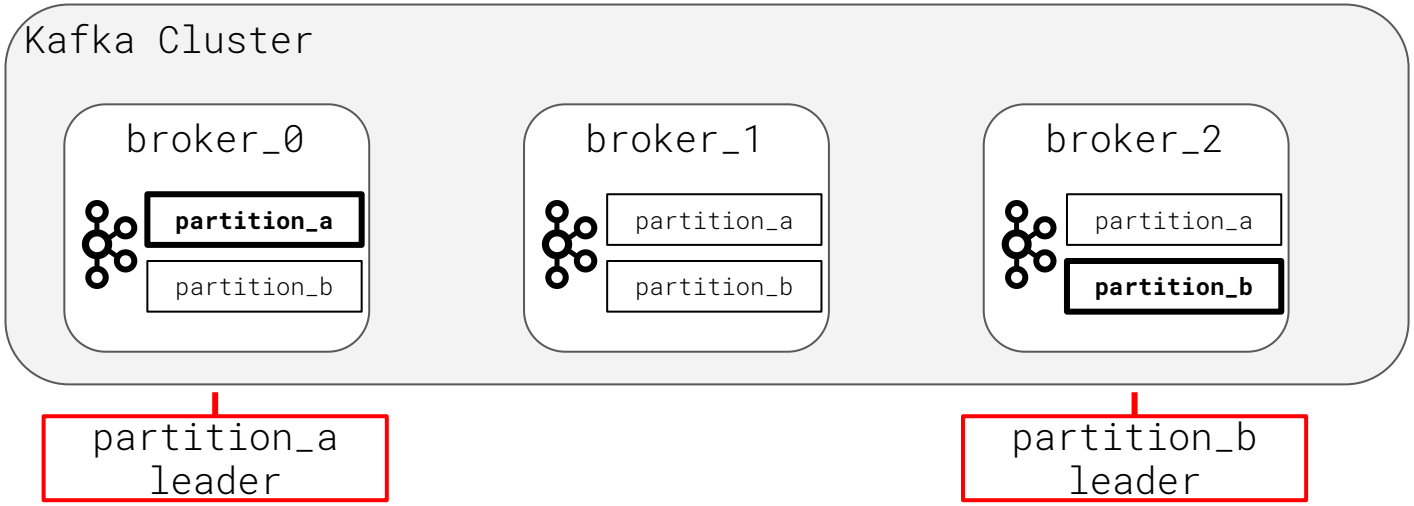
Data Replication

With replication, data is written to **multiple brokers**, not just the leader



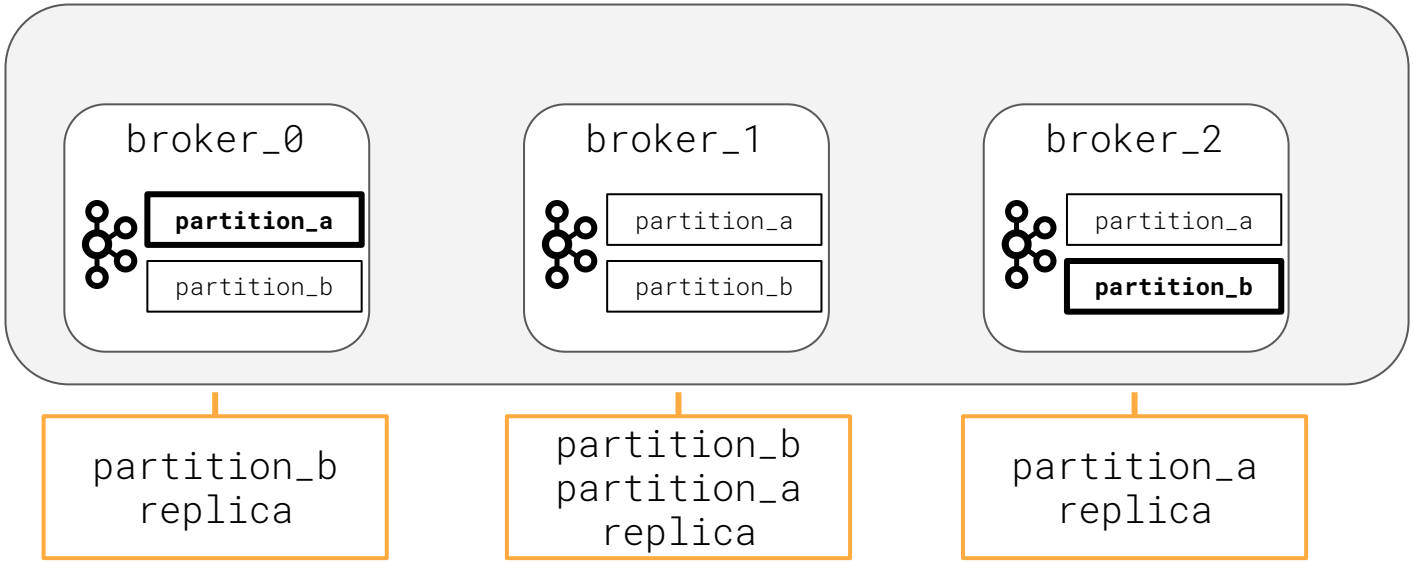
Data Replication

The **leader** of a partition sends data and coordinates client interactions



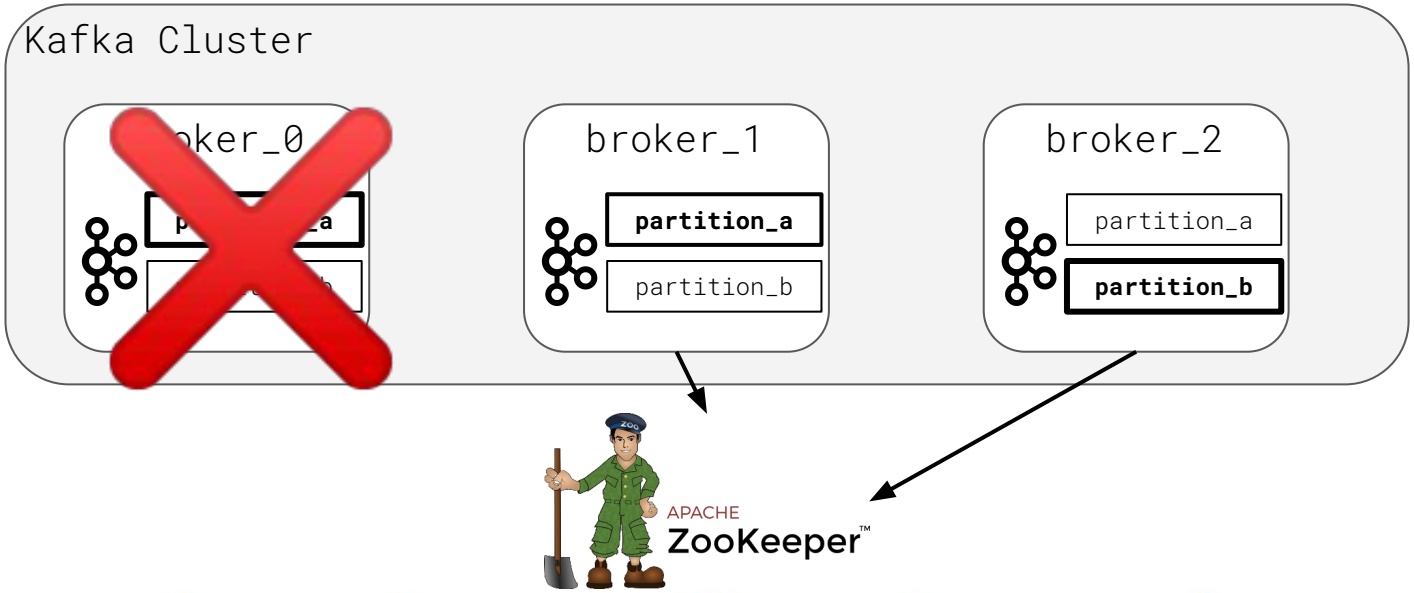
Data Replication

Replicas store a copy of the leader's data, in case the broker is lost



Data Replication

When a leader broker is lost, a new leader is **elected** with ZooKeeper



Data Replication

Tips and Takeaways

- **Default** number of replicas can be configured globally
- Can’t have more replicas than you have brokers
- Data replication incurs some overhead
- Always enable replication in a Production cluster

Demo: Explore how Kafka Works

Kafka Topics in Depth

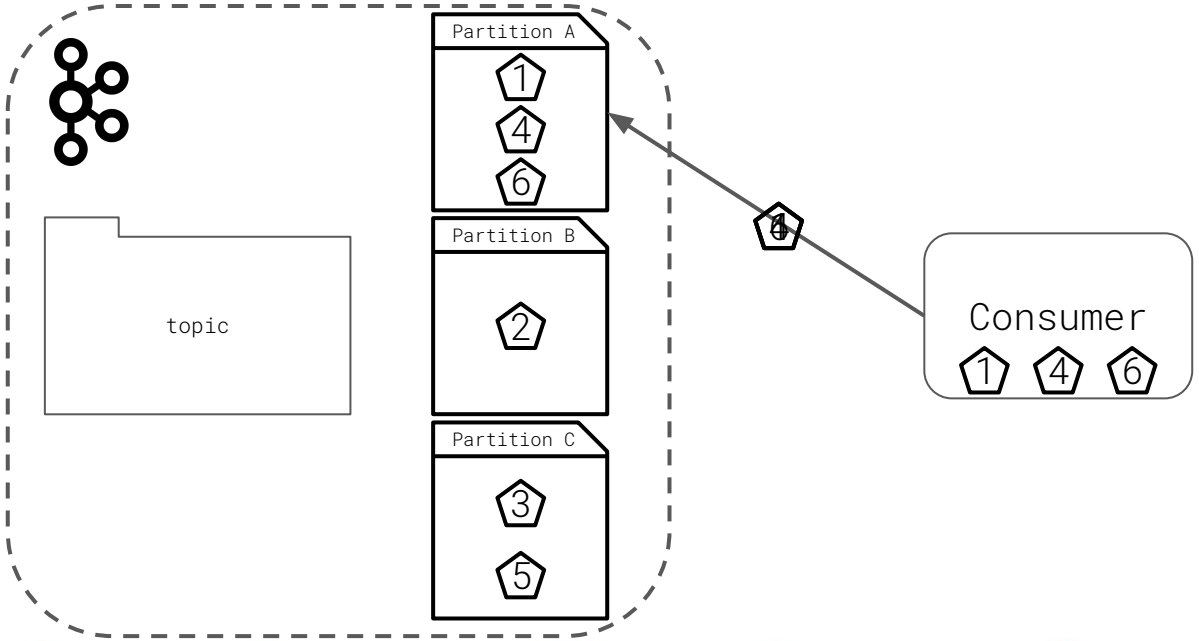
Replication

Topic Configuration

- Data replication can be set on a **per-topic basis**
- A broker must be an **“In Sync Replica” (ISR)** to become leader
- **Desired number of ISRs** can be set on topics
- **Number of ISRs must succeed** when data is sent
- If ISR is **too high** it will **slow down** processing

Partitioning

Topic **ordering guarantees** only apply within partitions



Partitioning

Topic Configuration

- No “right number” of partitions
- First determine desired throughput, e.g. 100MB/s
- Rough equation to determine number of partitions:

$$\#\ \text{Partitions} = \text{MAX}\left(\frac{\text{Throughput}}{\#\ \text{Producers}}, \frac{\text{Throughput}}{\#\ \text{Consumers}}\right)$$

- Partitions have performance consequences

Naming Conventions

Topic Configuration

- No official or idiomatic pattern defined
- Kafka requires names < 256 chars, [a-zA-Z0-9.-_]
- Name topics according to some consistent strategy
- Consistent naming leads to simpler consumption
- Recommended approach:

  < domain > . < model > . <event type>

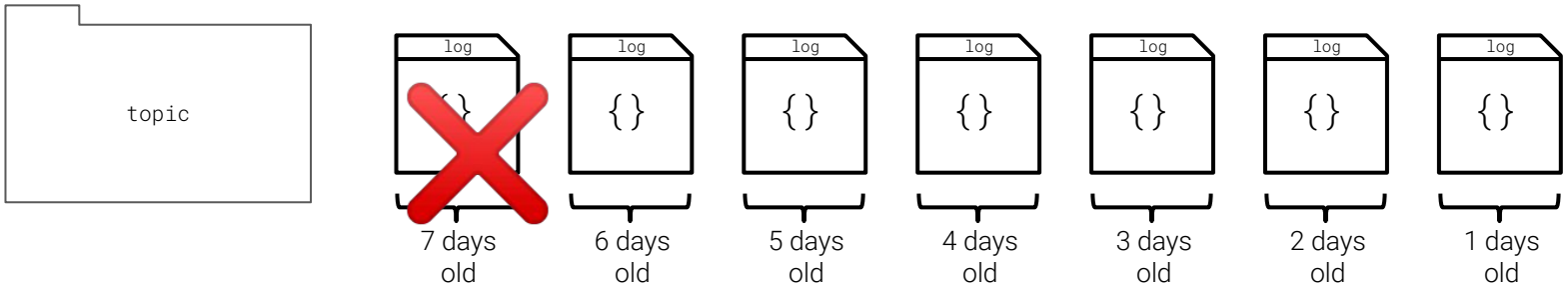
  ex: **com.udacity.lesson2.quiz.result**

Data Management

Data Retention is how long Kafka stores the data in a topic

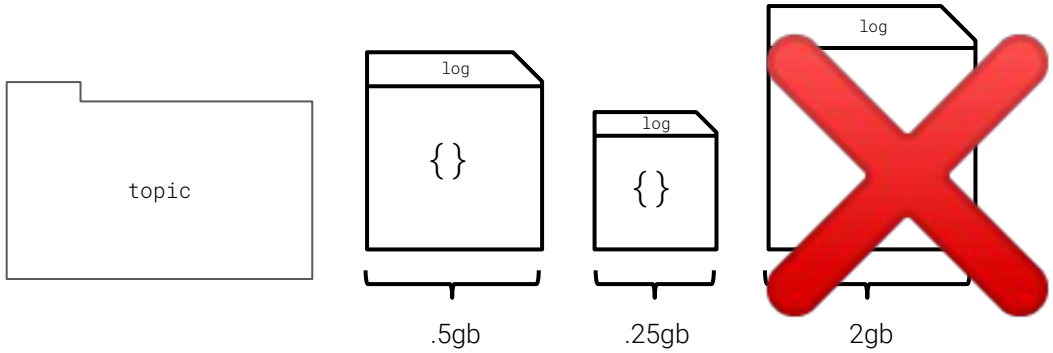
Data Management

Topics may be configured to **expire** data after a period of retention time



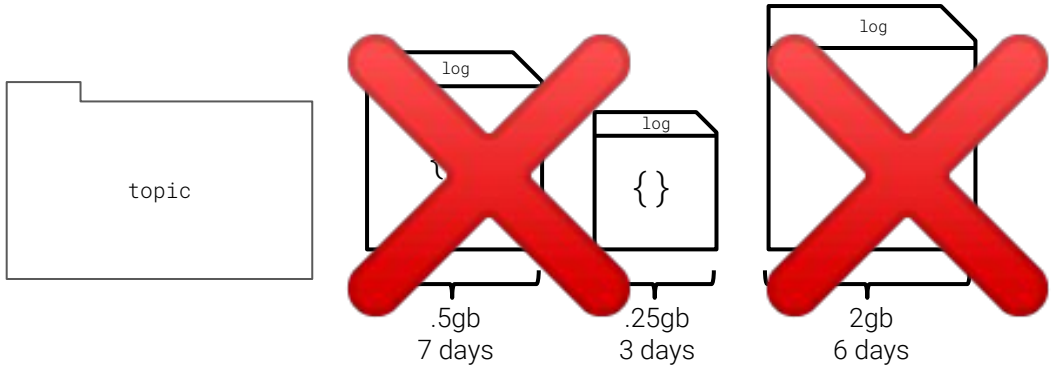
Data Management

Topics may **expire** data based on a **size threshold**



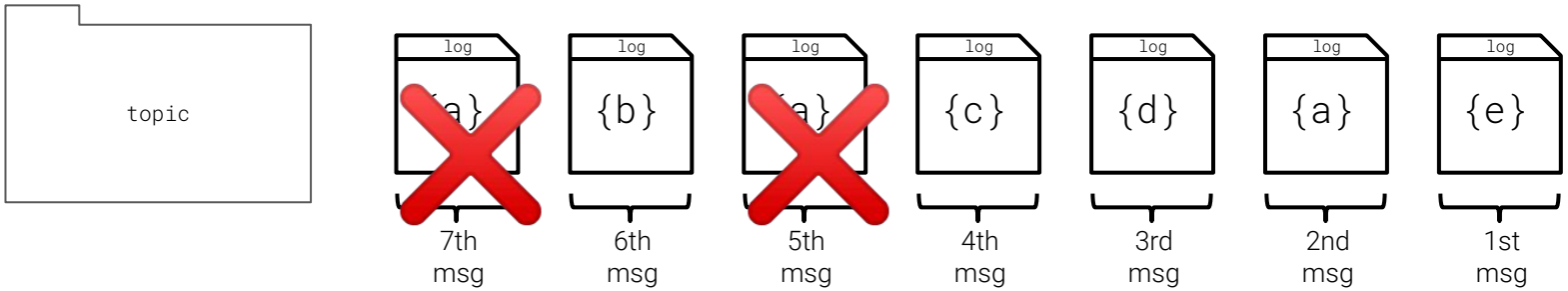
Data Management

Topics may **expire** data based on **both time** and **size**



Data Management

Log **compaction** has no size or time limit. If a key appears more than once, old versions of that data are **compacted** (deleted).



Best Practices

Topic Configuration

- Configure **compression** on a topic for speed
- `lz4`, `ztsd`, `snappy`, and `gzip` are supported
- **Never** put more than one event type in a topic
- Mixing event types introduces complexity
- Mixing event types causes outages

Topic Creation

Create Topics Manually!

Kafka *can* create topics automatically, but this is an anti-pattern and bad practice.

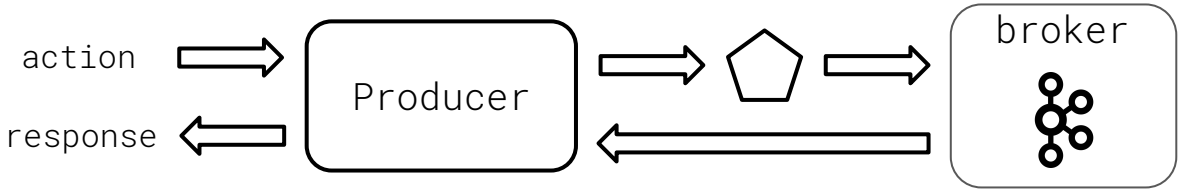
Write code or use a provisioning tool to **manually create your topics** as needed.

Demo: Create a Kafka Topic

Kafka Producers

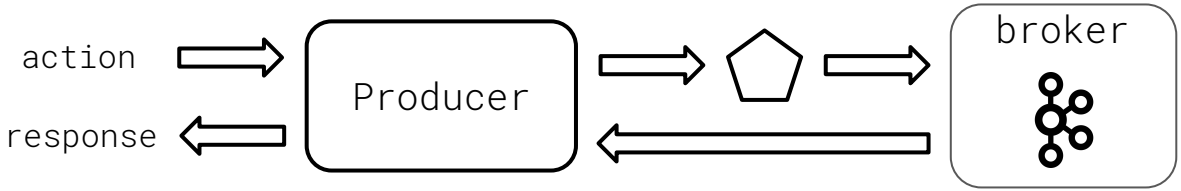
Synchronous Production

Synchronous producers block producer program execution until the broker has confirmed receipt



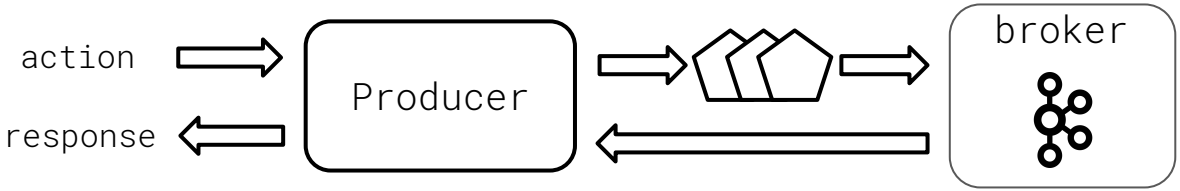
Synchronous Production

Use synchronous producers only as necessary!



Asynchronous Production

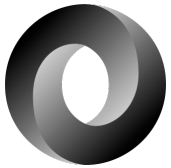
Asynchronous producers send the data and immediately continue



Message Serialization

Message Serialization

Data sent to Kafka should be **serialized** into a standard format



- Data sent to Kafka should be **serialized** into a format
- Kafka client libraries can assist in serialization
- Formats include binary, string, csv, JSON, Avro
- Never change serialization type without a new topic!

Demo: Synchronous vs Asynchronous Production

Additional Settings

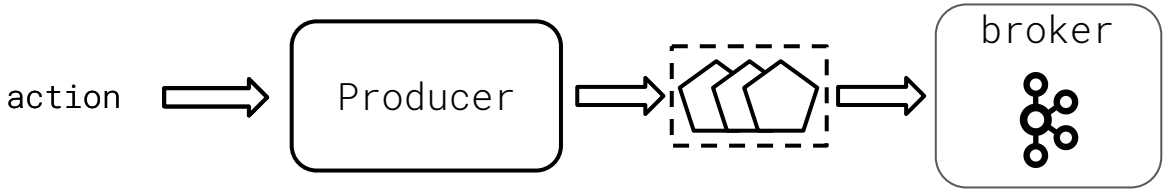
Producer Configuration

- Set `client.id` for a better debugging experience
- Configure retries to ensure data is delivered
- Set `enable.idempotence` to `true` for in-order retry
- Configure compression on individual topics
- Topic-level compression always takes precedence
- `acks` determine number of required ISR confirmations

Demo: Message Compression

Controlling Message Batches

Kafka producer libraries **batch** messages together for efficiency



Controlling Message Batches

- Messages may be batched on one or more of: **time**, **count**, or **size**
- Kafka clients allow configuration of these batch settings
- Batch settings can be critical for producer performance

Demo: Producer Configuration

Kafka Consumers Introduction

Kafka Consumers

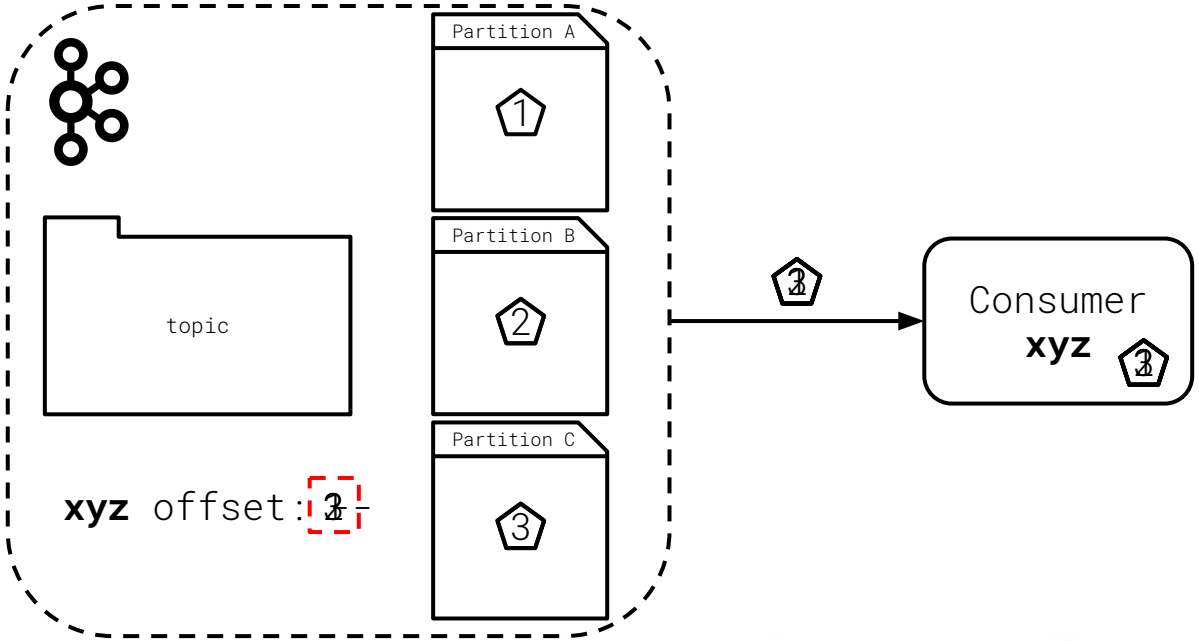
Kafka consumers **subscribe** to one or more topics

Subscribing to a topic that does not exist will create it with default settings

Topics are **polled** for data by the Kafka client

Consumer Offsets

The **offset** is the last message a consumer successfully processed



Consumer Offsets

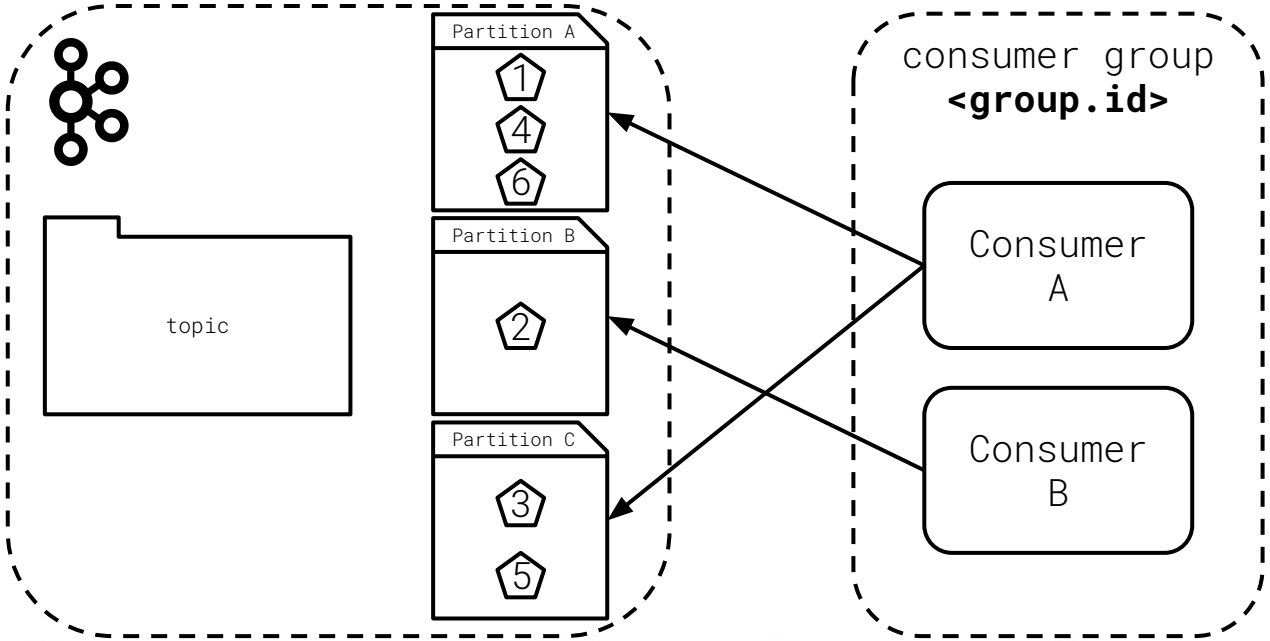
Consumer Configuration

- Offset tells Kafka consumer **where to start on restart**
- Offsets are committed to Kafka, typically automatically
- Client libraries asynchronously make commits
- Commits may be synchronous if required
- Commits may be manually managed if required
- Change commit to **earliest** to start from first message

Demo: Consumer Offsets

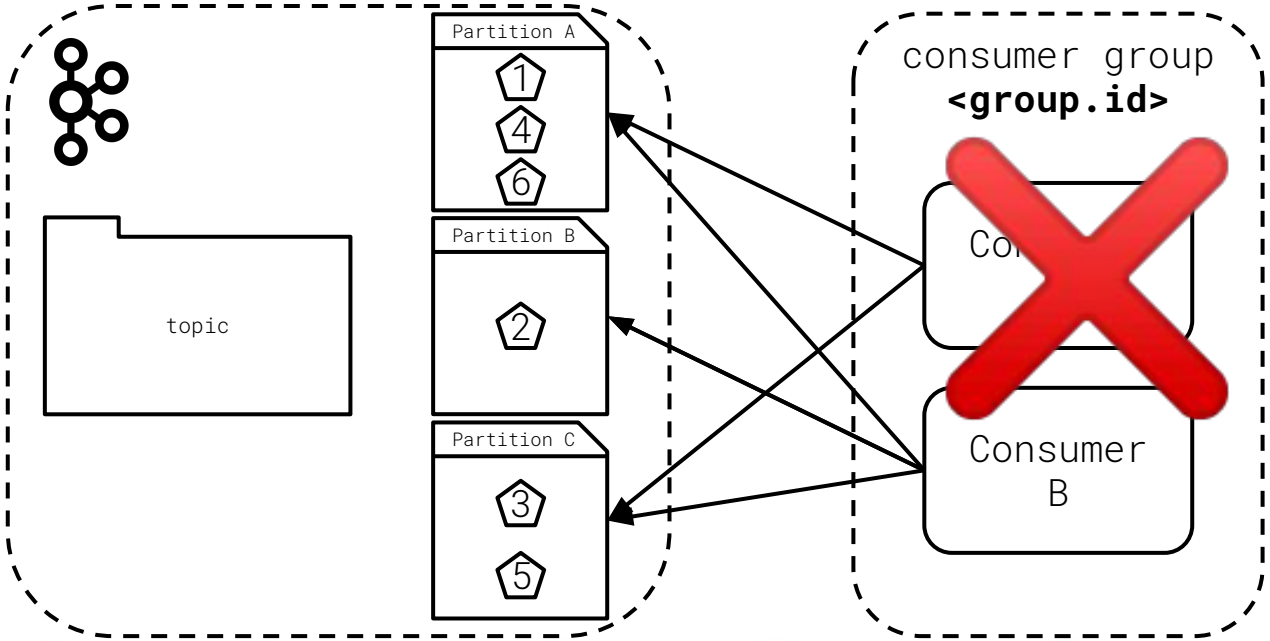
Consumer Groups

Consumer groups are clients consuming from the same topic



Consumer Groups

Consumer groups **rebalance** when a consumer leaves or joins the group



Topic Subscriptions

Consumer Configuration

- Subscriptions may specify a full topic name:

```
consumer.subscribe(
     “com.udacity.lesson2.quiz.results”
)
```

- Subscriptions may specify a regex, starting with ^:

```
consumer.subscribe(
     “^com.udacity.lesson2.*”
)
```

- The regex above would match all lesson 2 topics

Deserializers

Data being consumed must be **deserialized** in the same format it was serialized in. If the producer serialized in JSON, your consumer must also deserialize in JSON format. **Failure** to correctly deserialize **may cause crashes** or **inconsistent data** in processing.

Retrieving Data from Kafka

The Consumer **Poll Loop** fetches data from Kafka

```
...
while True:
    message = consumer.poll(timeout=1.0)
    if message is None:
        continue # No data was retrieved
    elif message.error() is not None:
        continue # Log error in production
    else:
        print(message.key(), message.value())
```

Retrieving Data from Kafka

The **consume** method can fetch multiple messages at once

```
while True:
    messages = consumer.consume(5, timeout=1.0)
    for message in messages:
        if message is None:
            continue # No data was retrieved
        elif message.error() is not None:
            continue # Log error in production
        else:
            print(message.key(), message.value())
```

Demo: Retrieving Data from Kafka

Consumer Performance Considerations

Consumer Performance

Key Considerations

- **Consumer Lag** measures how far behind consumer is
- `Lag = Latest Topic Offset - Consumer Topic Offset`
- Messages Per Second indicates throughput
- Kafka Java Metrics Exporter provides real-time metrics

Producer Performance

Key Considerations

- Measure producer latency to understand performance
- `latency = time broker received - time produced`
- High latency may indicate `acks` setting is too high
- High latency may indicate too many partitions
- High latency may indicate too many replicas
- **Producer response rate** tracks overall delivery rate

Broker Performance

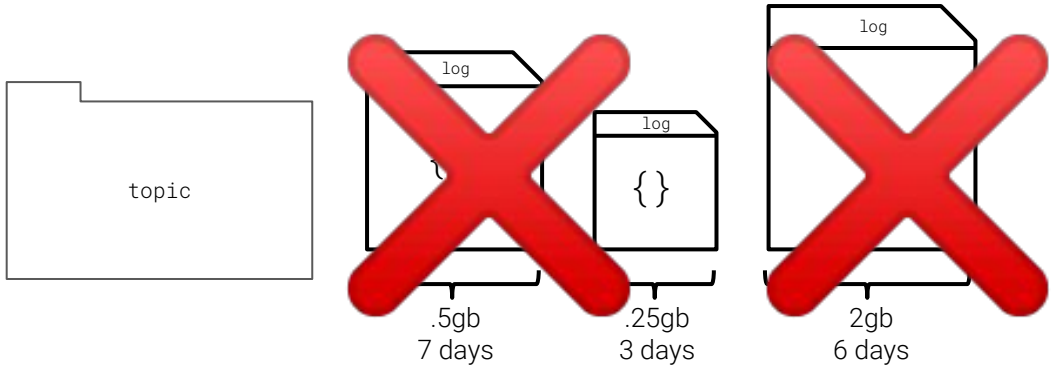
Key Considerations

- Track disk usage, high disk usage can cause outages
- High network usage may slow consume/produce
- Election frequency is critical, should be infrequent
- Elections stop all consume/produce, **very** disruptive
- Frequent elections indicate broker instability

Removing Records and Data Privacy

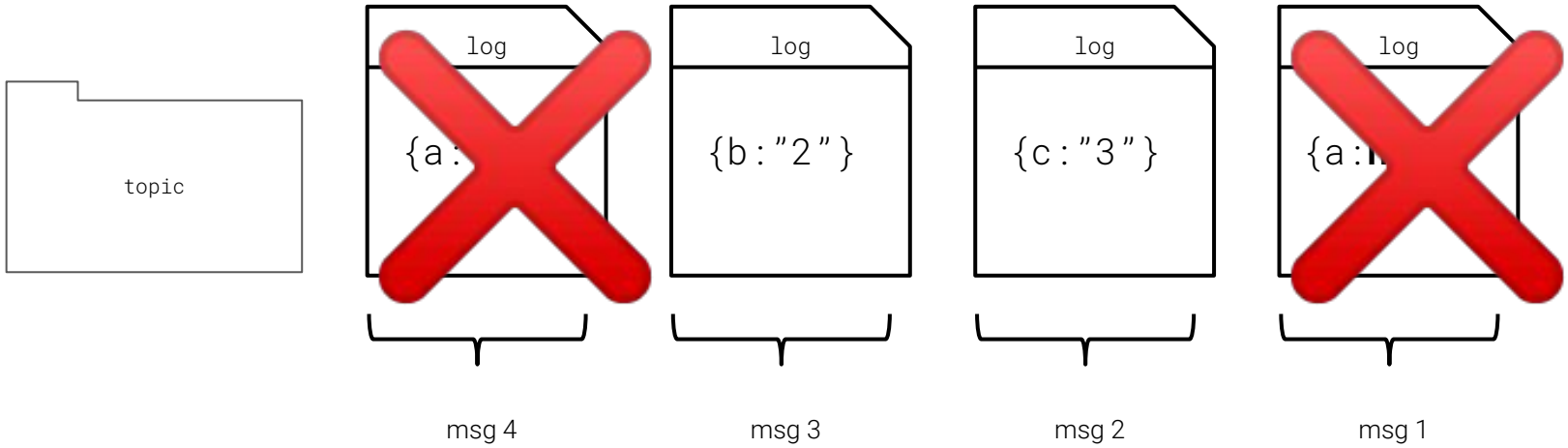
Message Expiration

Simplest method to delete records is to simply use expiration



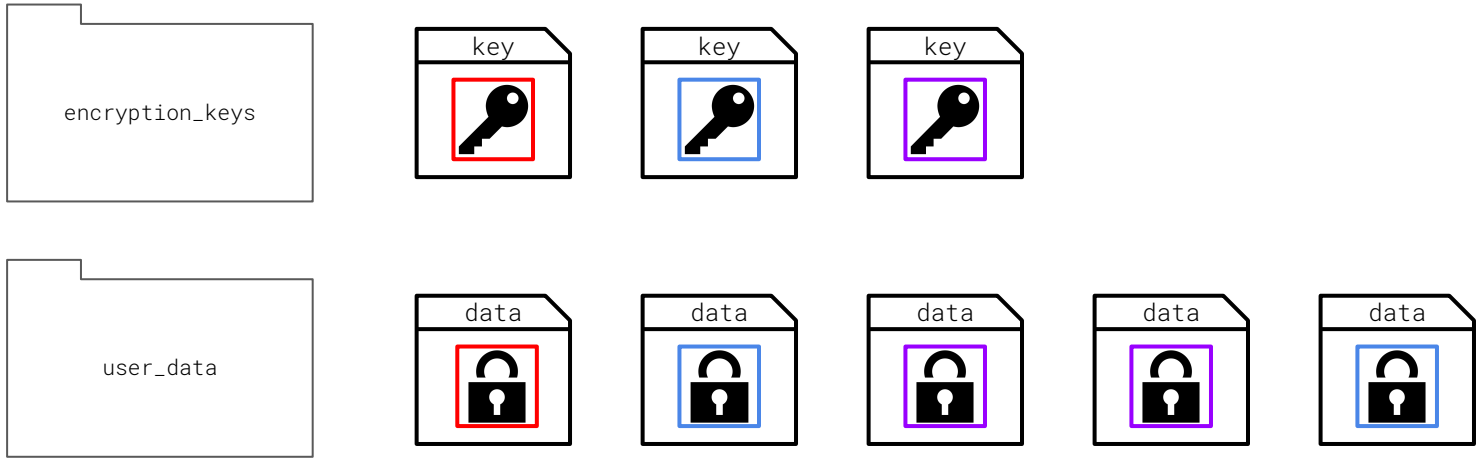
Message Compaction

`null` messages in a compacted topic delete the data for that key



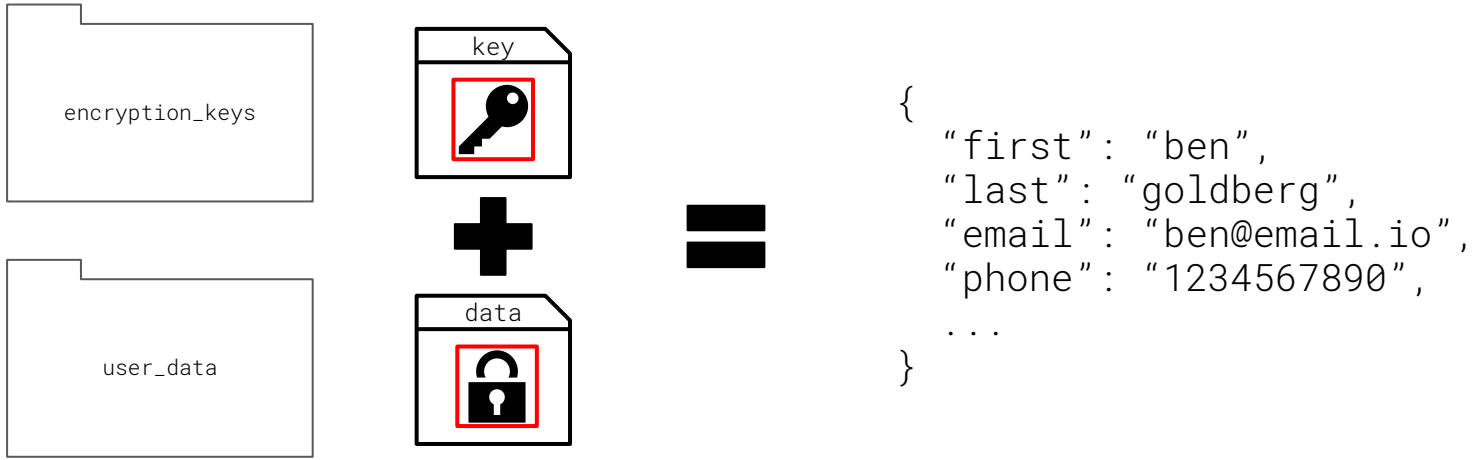
Encrypted User Keys

User data is **encrypted** with a special key



Encrypted User Keys

The key is used to decrypt and access the data



Encrypted User Keys

Deleting the encryption key makes the remaining data unusable