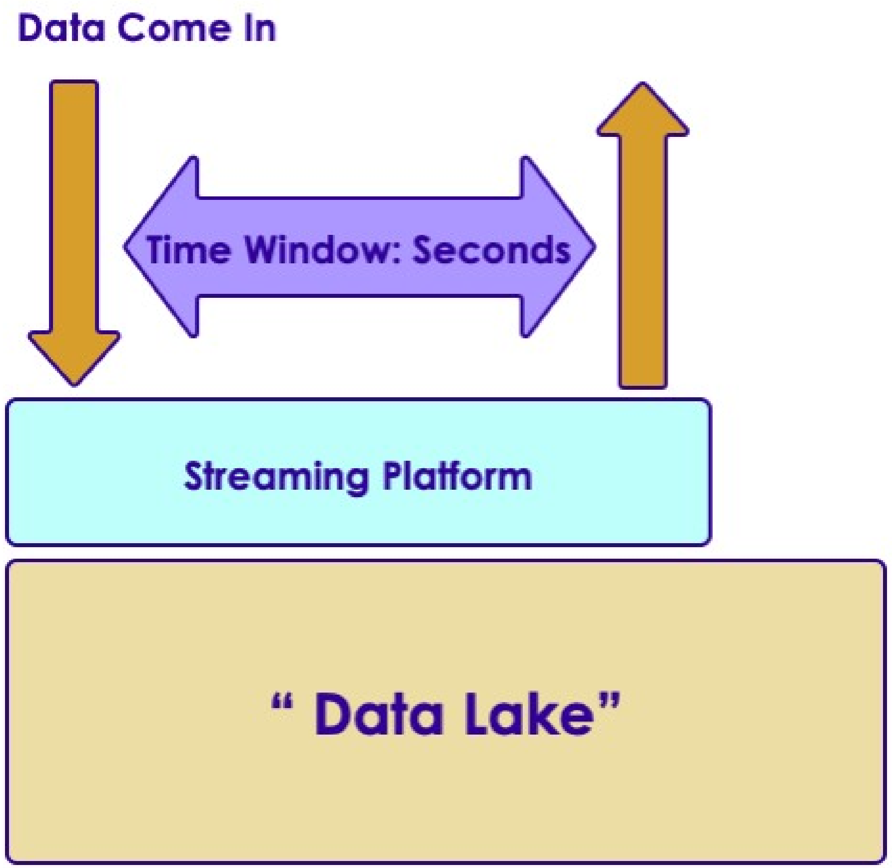
Microservices Architecture

Kafka

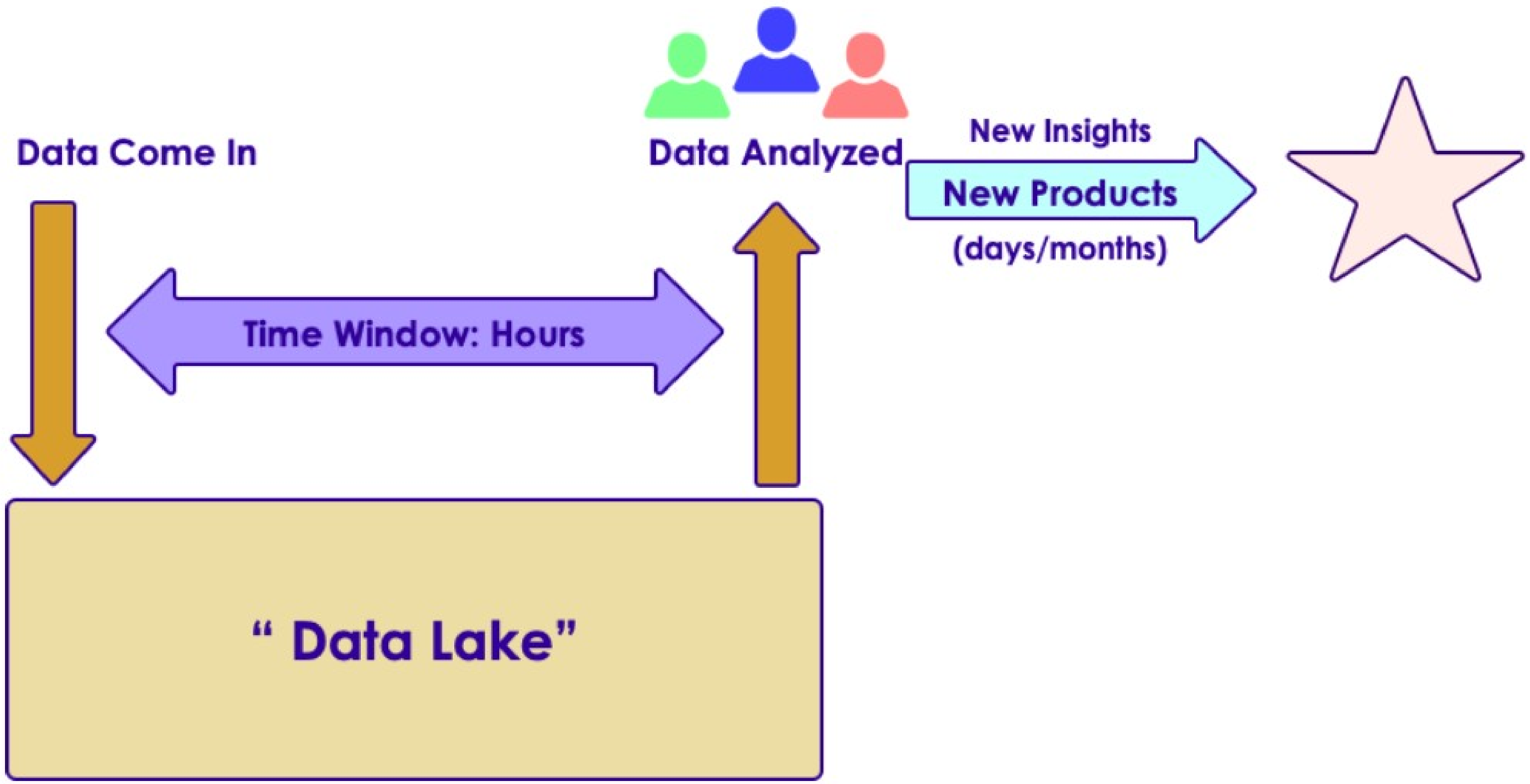
Moving Towards Fast Data: Version 2

- Decision time: (near) real time
  - Seconds (or milliseconds)
- Use Cases
  - Alerts (medical/security)
  - Fraud detection
- Streaming is becoming more prevalent
  - Connected Devices
  - Internet of Things
- Beyond Batch
  - We need faster processing and analytics



Big Data Evolution: Version 1

- Decision times: batch (hours / days)
  - Use cases: Modeling, ETL, Reporting



Streaming Use Cases

- Netflix
  - Recommendations 450 billion events/day
- Weather Company
  - Analyze weather sensor data
  - Billions of events/day
  - Multi-Petabyte (PB) traffic daily

Real Time / Near Real Time

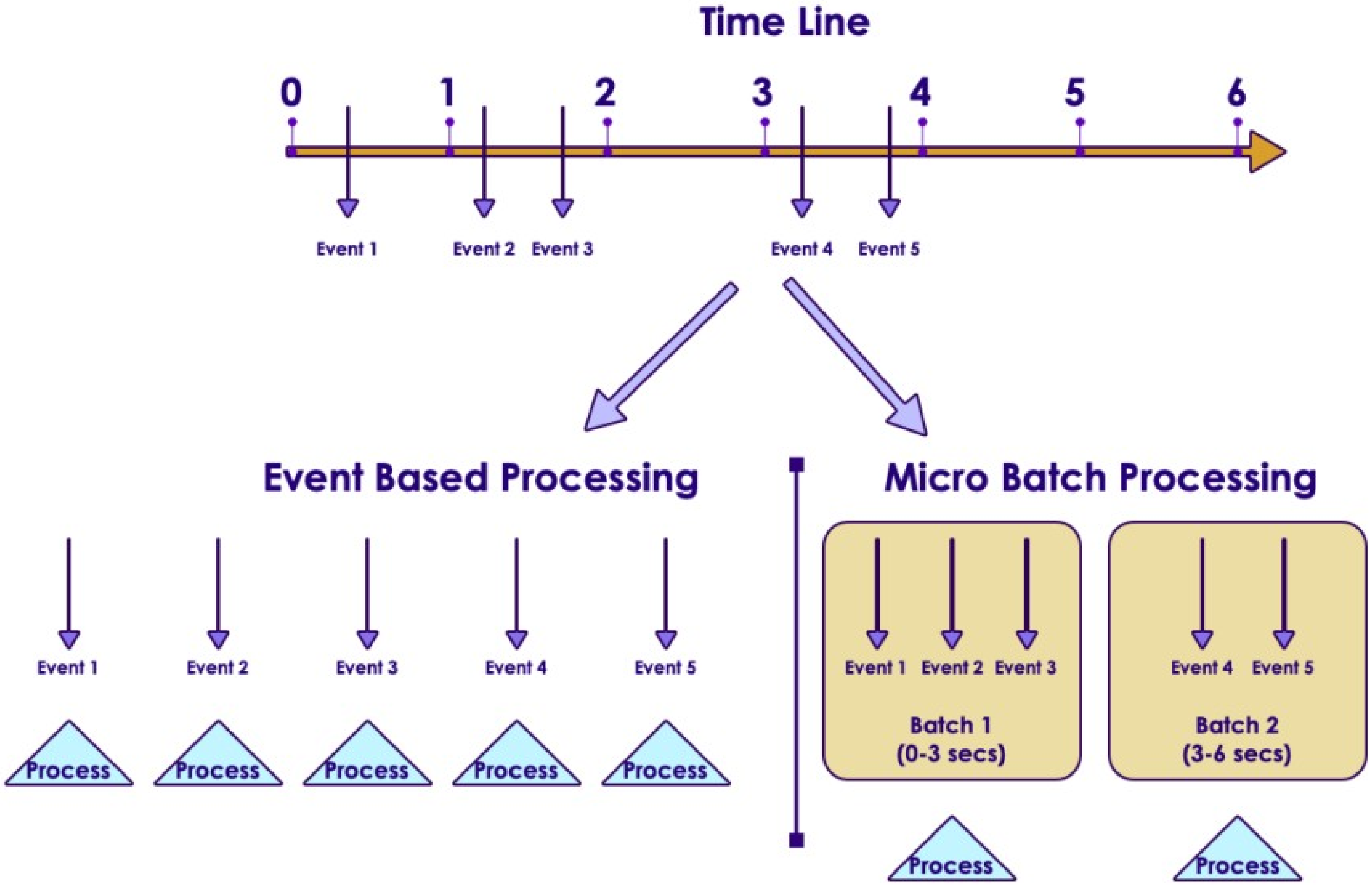
- The 'real' real time is in milliseconds order
  - DB query returns in 2 ms
- ‘Near real time' is seconds
  - We can process an event within 3 seconds of its generation time

Name	Time	Example
Hard real time	Single order ms,sub milli seconds 1 ms,0.5 ms	Space shuttle control systems
Credit card transaction processing	50 ms, 300 ms	Db queries
Sending Emails	2 secs +	Stream processing latency
	1 min +	Mini batch queries

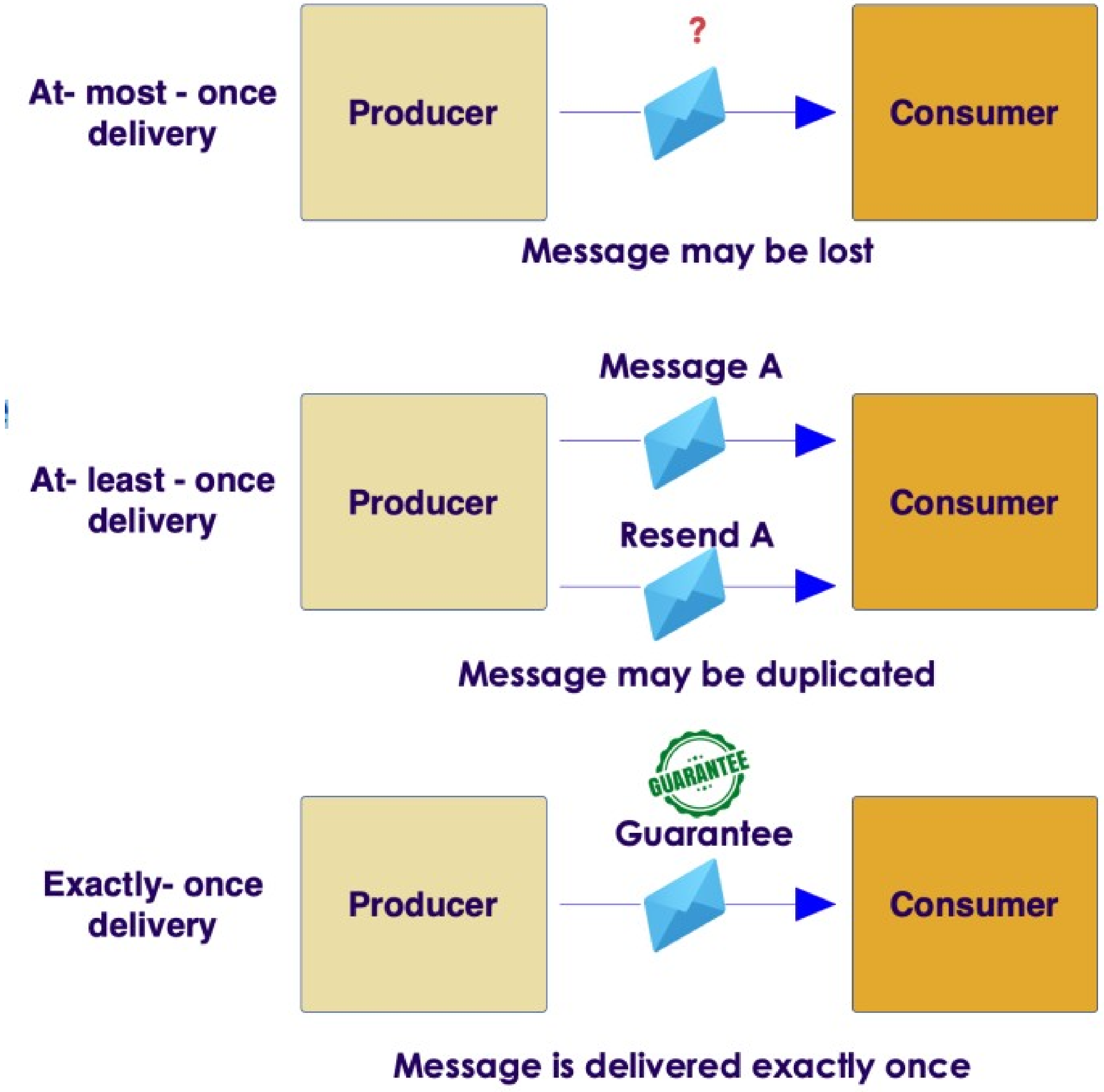
Streaming Concepts

- Processing model
  - Event based or micro batch based
- Processing guarantees
  - At least once
  - At most once
  - Exactly once
- State management
  - Event time vs. Arrival time
- Window Operations
- Back-pressure adjustment

Event Based Vs. Batch

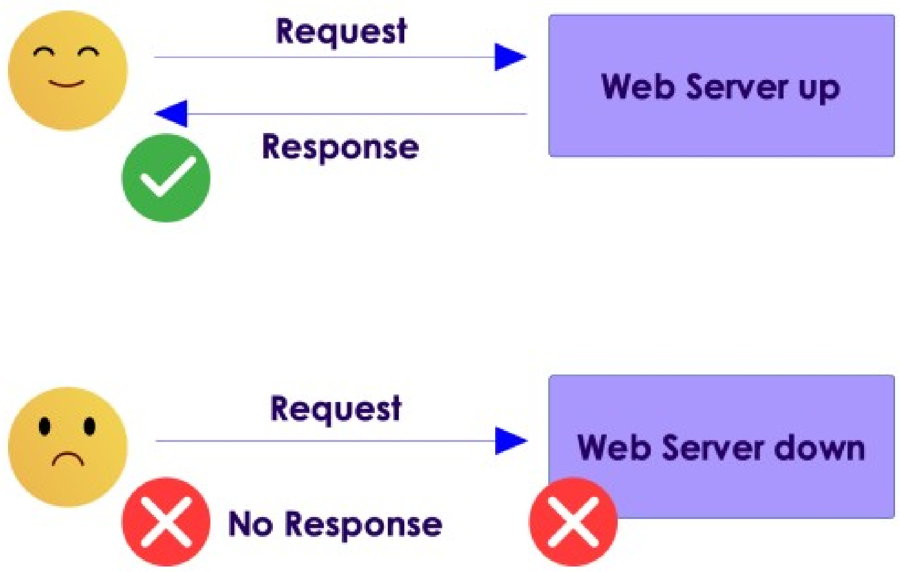


Processing Guarantees



At Most Once

- Event is sent only once
  - No duplicate processing
  - Events can be dropped due to crashes or heavy load
  - E.g. Web requests (if the web server is busy, requests are dropped)



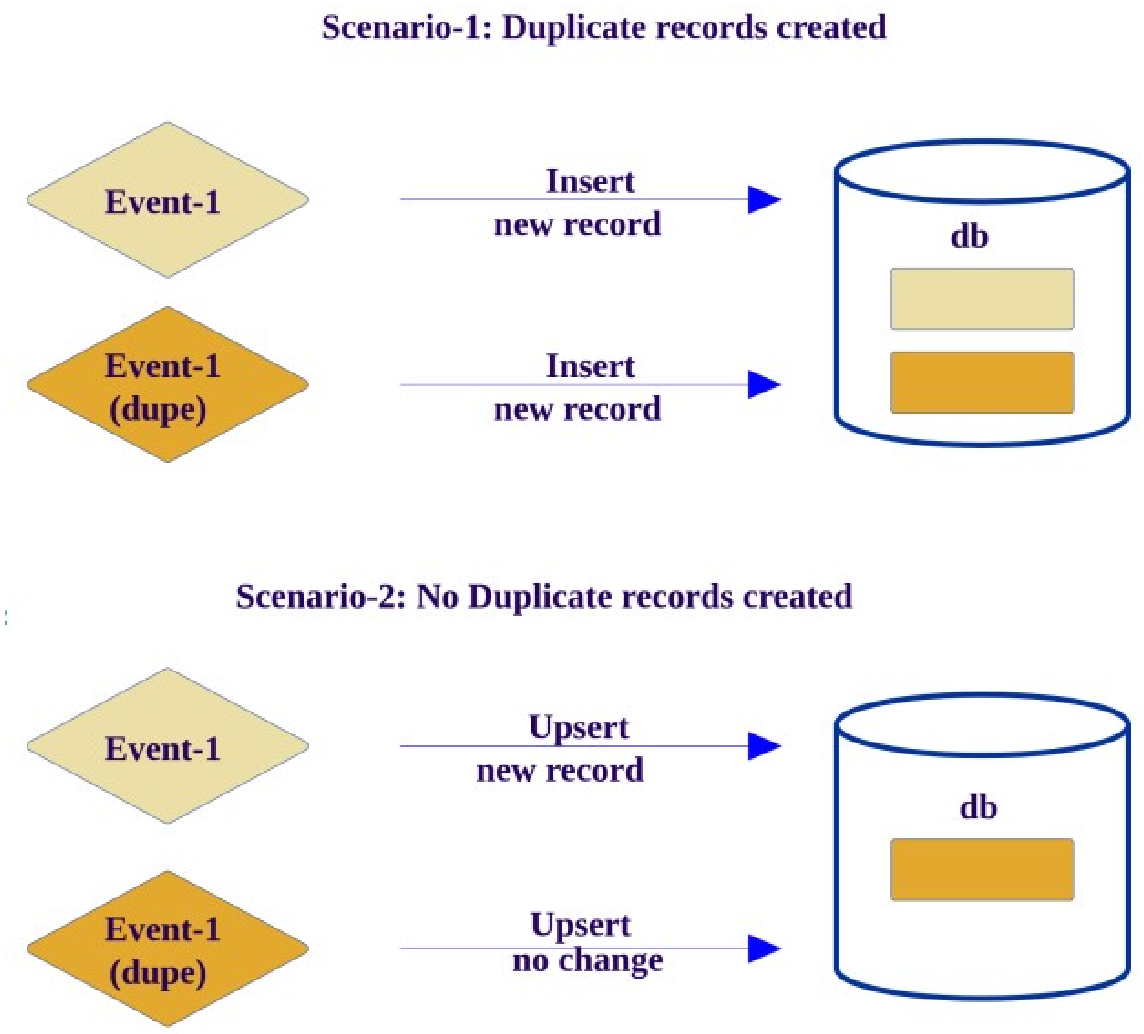
At Least Once

- All events are guaranteed to be processed (no dropped events)
  - However, events can be processed more than once
  - In case of failure recovery, events can be re-sent and processed again.
- Most common implementation
  - Frameworks: All (Storm, Spark, NiFi, Samza, Flink)

Handling Duplicate Events

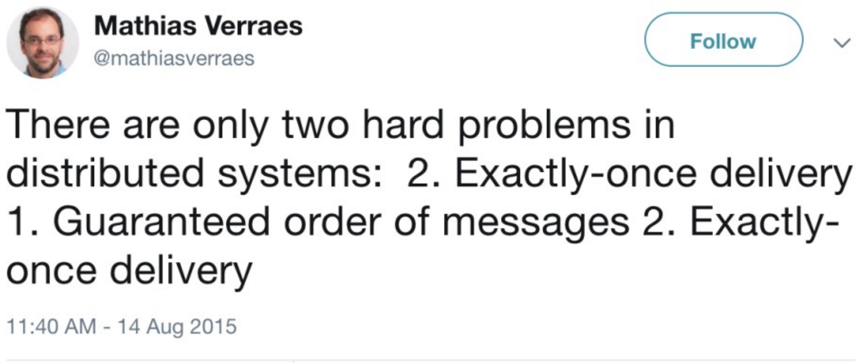
- A resilient streaming system, has to be ready to handle duplicate events
- We have 2 scenarios:
  - First one, we are inserting a new record for each event received. This will result in duplicate records in the database
  - Second one, we are checking to see if the event is processed already, only if not, then a new record is inserted
- Second approach is more resilient, can deal with duplicate events
  - This is called idempotent processing (no side effects for duplicate events)

Handling Duplicate Events



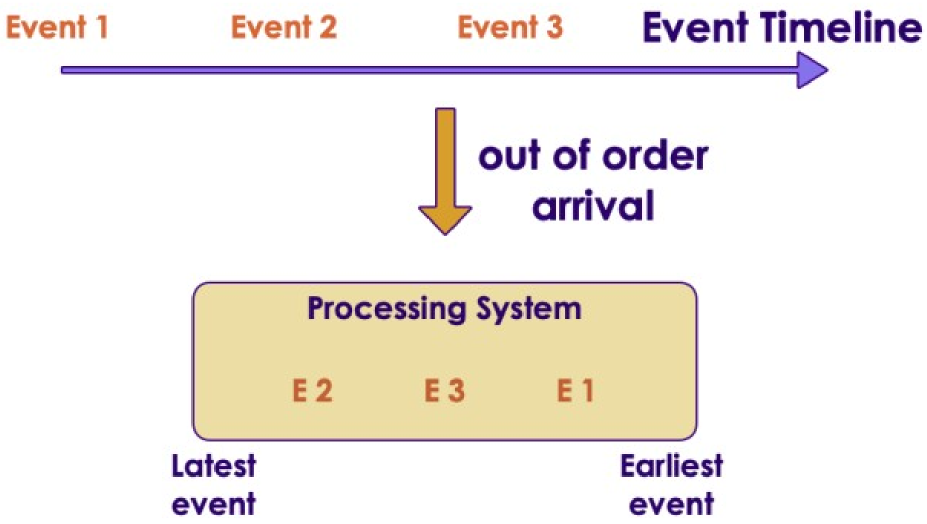
Exactly Once

- Events are guaranteed to be processed once and only once
  - No dropped events
  - No duplicate processing
  - Frameworks: Storm (with Trident), Flink, Spark, Samza
- Sample applications
  - Credit card processing



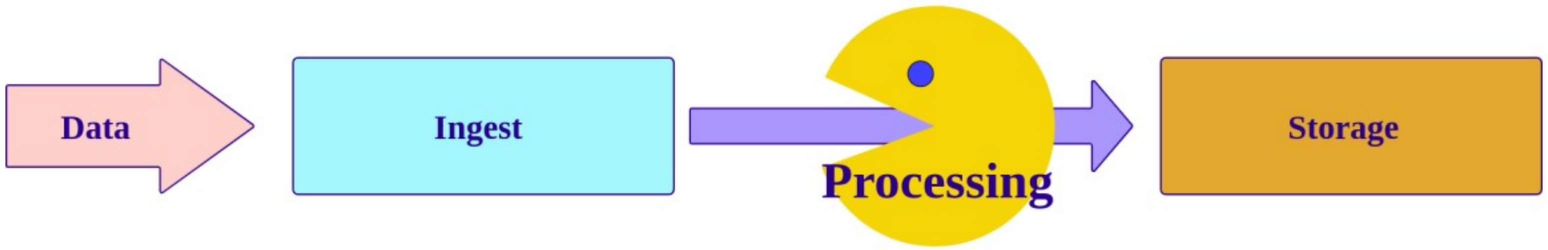
Event Time and Arrival Time

- Event Time: When the event occurred / generated
- Arrival Time: When event arrives for processing
- Event Time < Arrival Time
  - Some times events may arrive 'out of order' (due to network lag, outage ..etc)



3 Tier Streaming Architecture

- Here is a simplified streaming architecture
- We see 3 distinct stages
  - Ingest stage captures data
  - Processing handles the data
  - And the processed data is stored in Storage layer



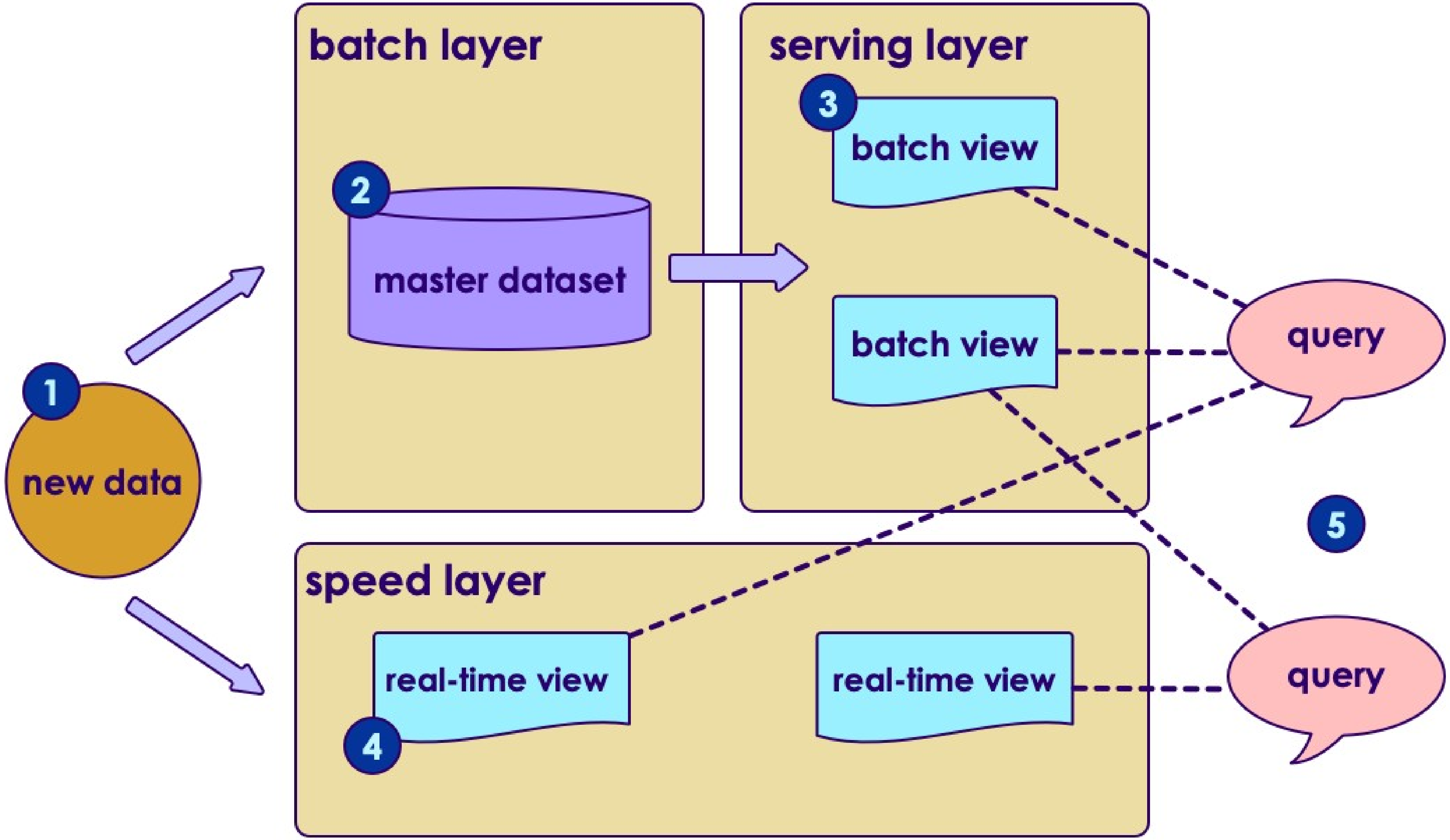
Ingest / Capture

- This layer:
  - Captures incoming data
  - Acts as a 'buffer' - smoothes out bursts So even if our processing offline, we won't loose data
- Choices
  - Kafka
  - Queues (MQ, JMS ..etc)
  - Cloud based queues like Amazon Kinesis

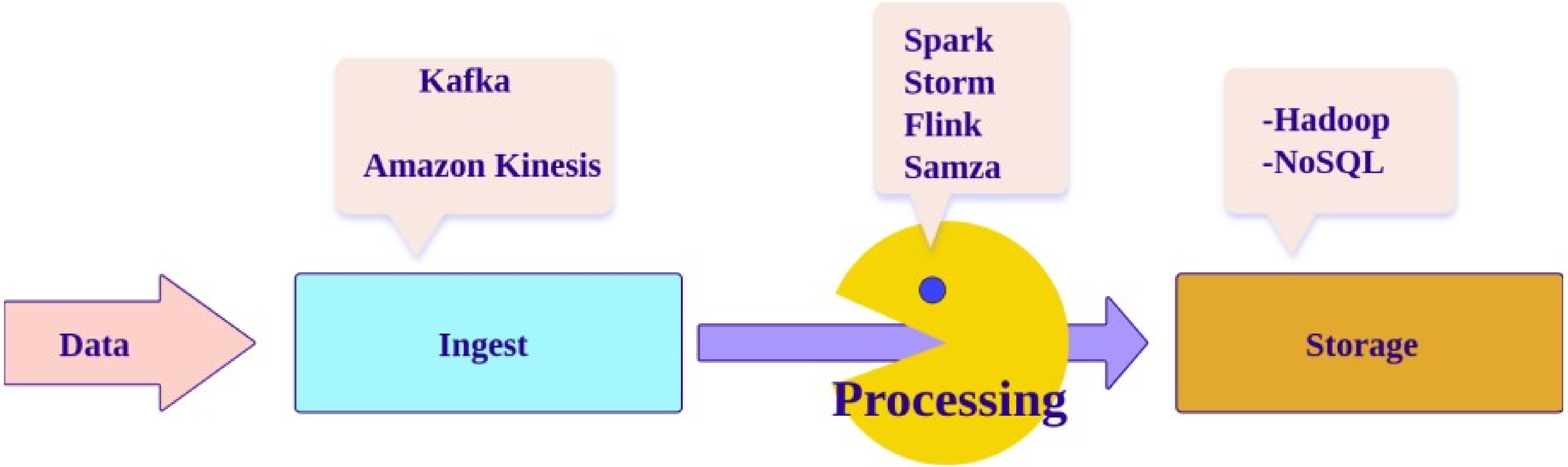
Storage

- After processing, they are stored for later retrieval
- Two choices:
  - Real time store
  - 'Forever' store
- Real Time Store
- Need to absorb data in real time
  - Usually a NoSQL storage (HBase, Cassandra ...etc)
  - May contain subset of data (last 1 year ..etc)
- 'Forever store'
  - Needs to store massive amounts of data
  - Support analytics (usually batch)
  - Hadoop / HDFS

Lambda Architecture



Streaming Stack - Summary

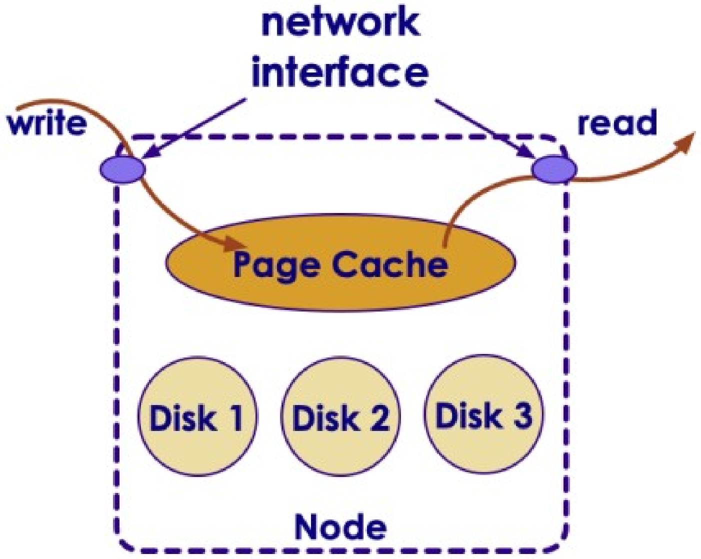


Apache Kafka

- Kafka is a Publisher / Subscriber (Pub-Sub) messaging system
  - Distributed
  - Scales seamlessly
- High throughput
  - Capable of handling billions of messages per day
- Replicated
  - Safeguards data in case of machine failures
- Created @ LinkedIn in 2010
  - Now Apache Project (Open Source)

Why Is Kafka Very Fast?

- Write: Disk writes are buffered in page cache
- Read: The data from page cache can be transferred to network interface very efficiently
- 99% of the time data is read from page cache, no disk access at all

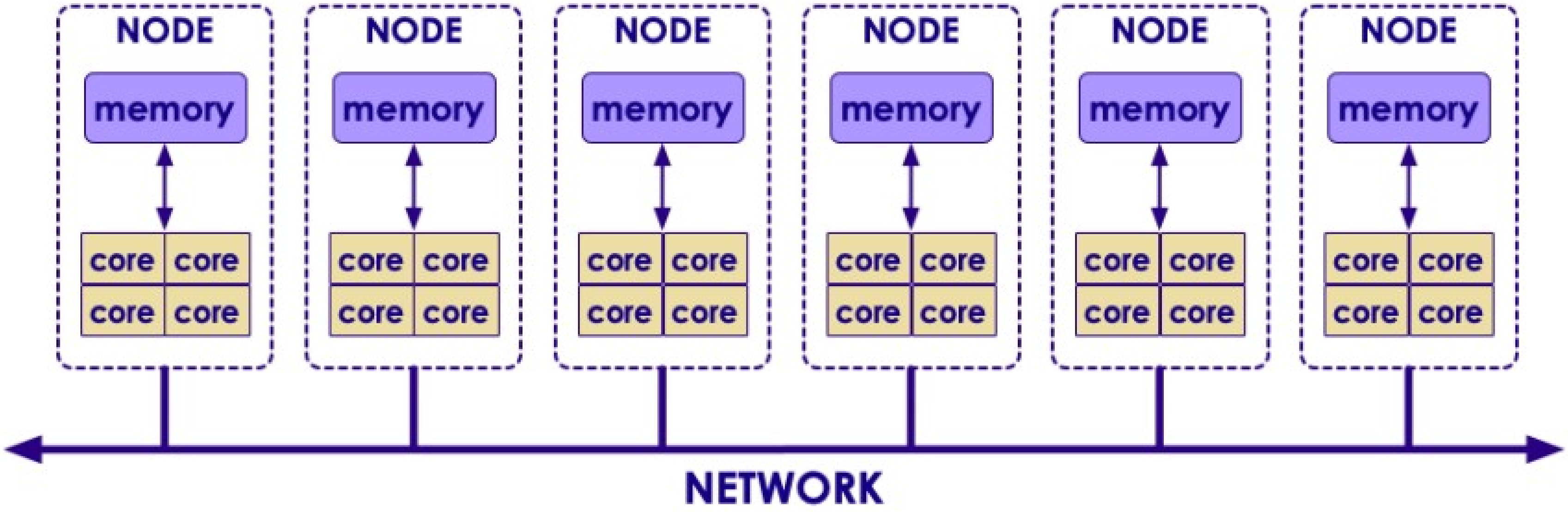


Kafka Features

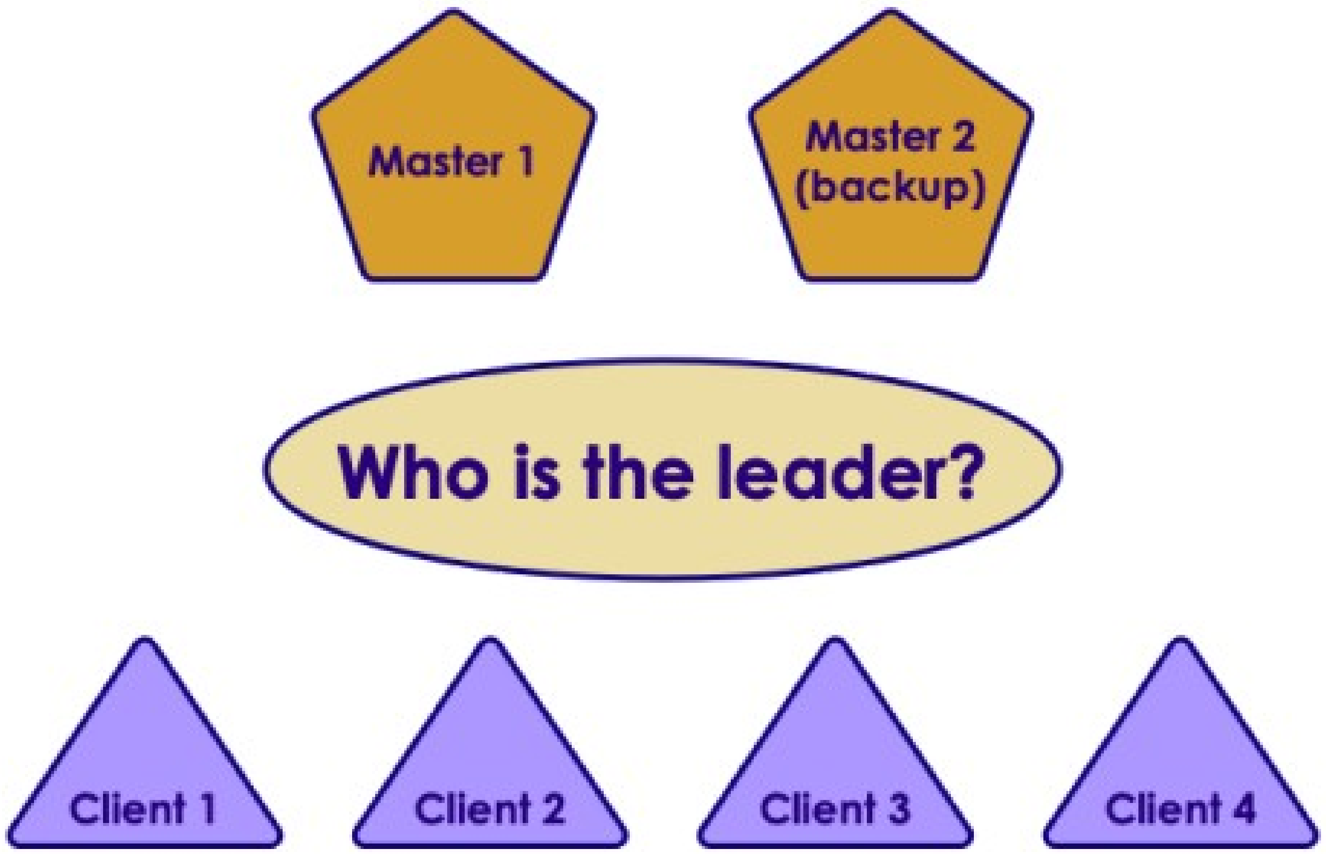
Feature	Kafka	Other Queue Systems
Deleting messages	Clients can not delete. ,Kafka auto-expires messages	Clients can delete
Message processing order	Can read in or out-of order	Usually read in order
Message processing guarantee	Kafka guarantee no duplicate processing of a message	Usually no
Concurrent read / write	Supported.,High throughput	Low throughput due to locking & blocking
Message priorities	None	Yes
Message ACKs,(Client notify producer that a message is processed)	No	May be

The Distributed Problem

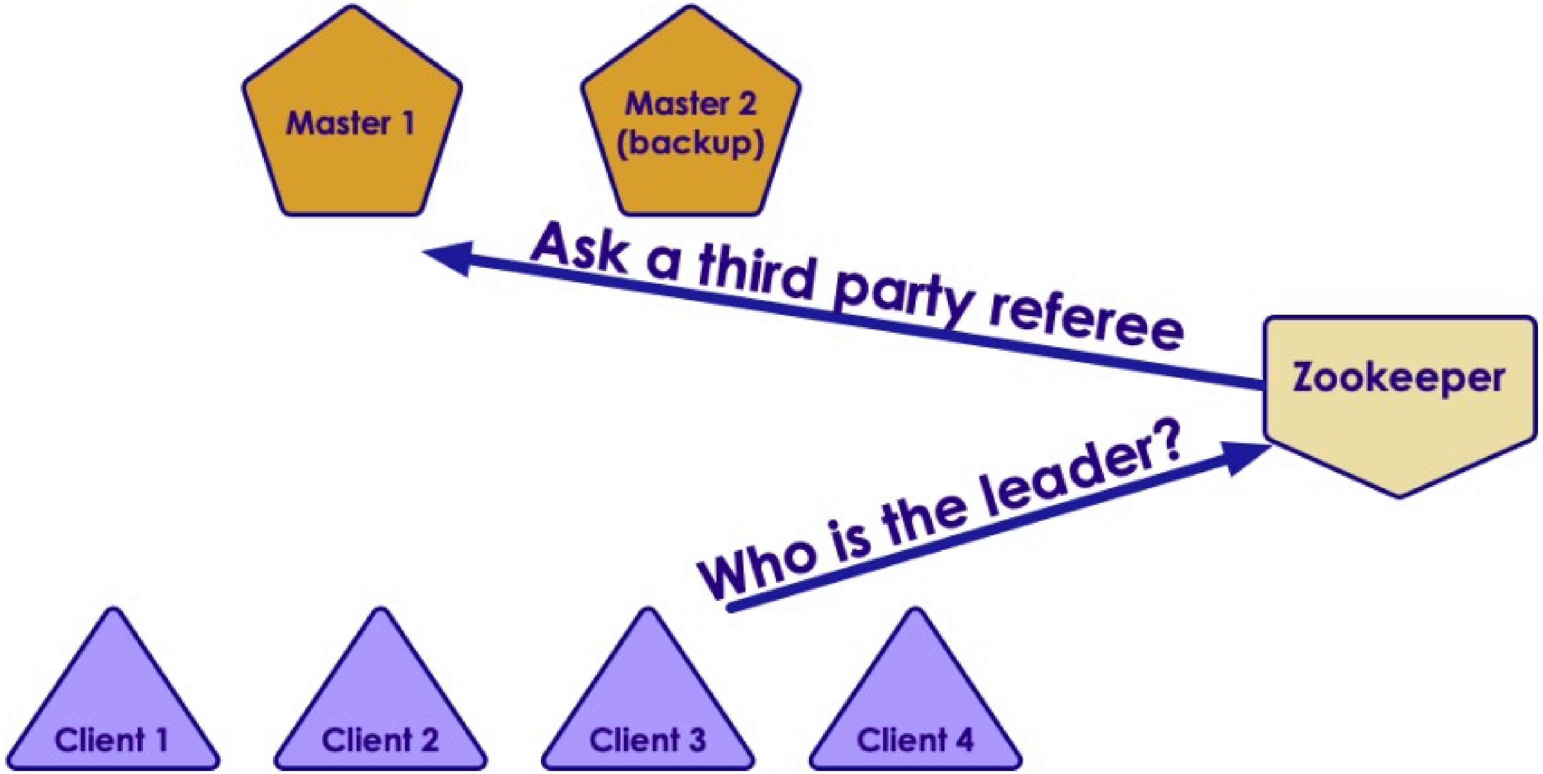
- Distributed systems with
  - Multiple nodes
  - Each with multiple cores
  - How do we co-ordinate them all?



Leader Election

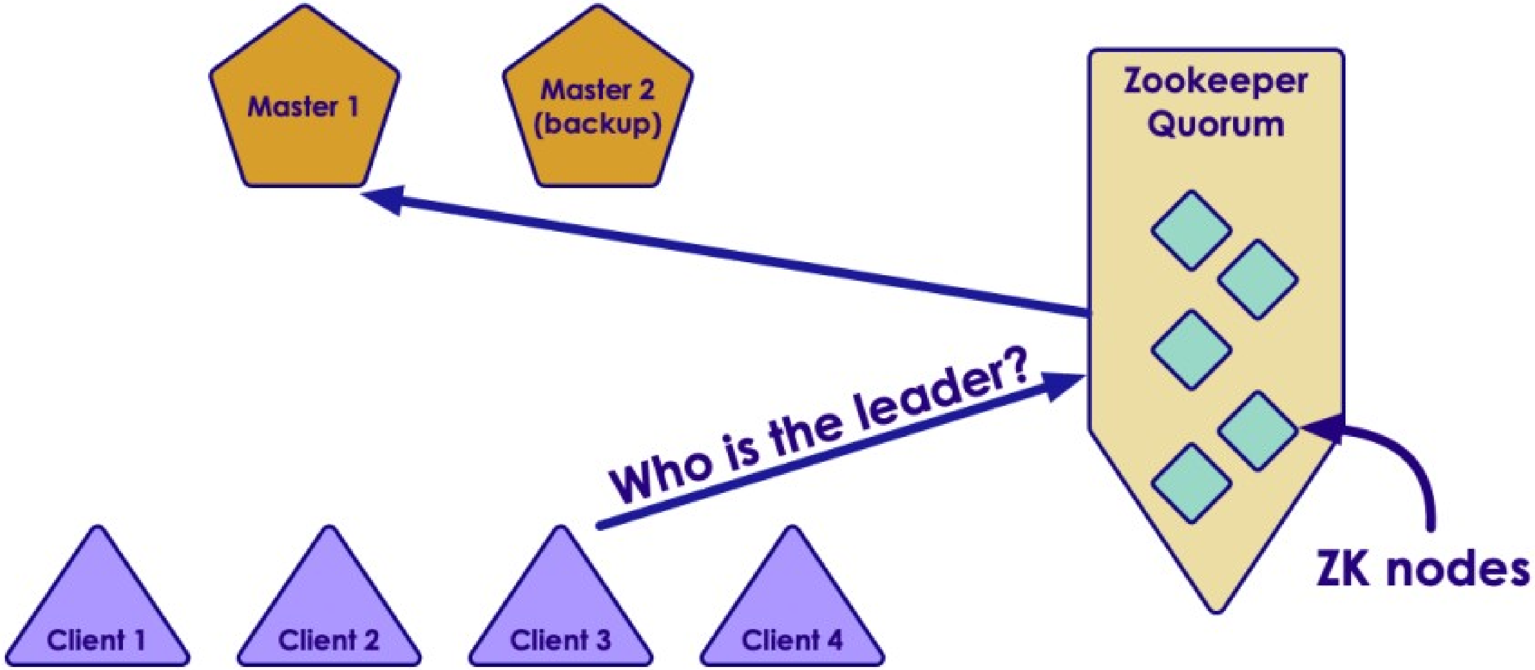


Leader Election With Zookeeper



Zookeeper Cluster / Quorum

- What if ZK goes down?
  - Run ZK as a cluster - quorum
  - No single point of failure

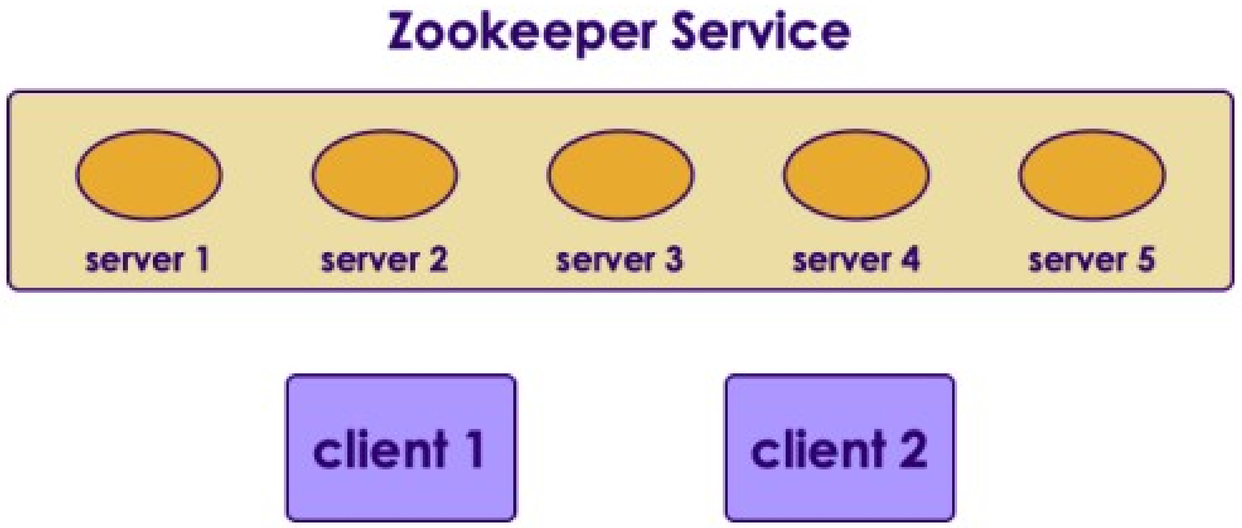


Zookeeper

- Distributed service that provides
  - Configuration
  - Synchronization
  - Name registry
  - Consensus
  - Leader election
- Open source
  - Apache open source project
  - Battle tested with very large distributed projects
  - Hadoop, HBase, Kafka

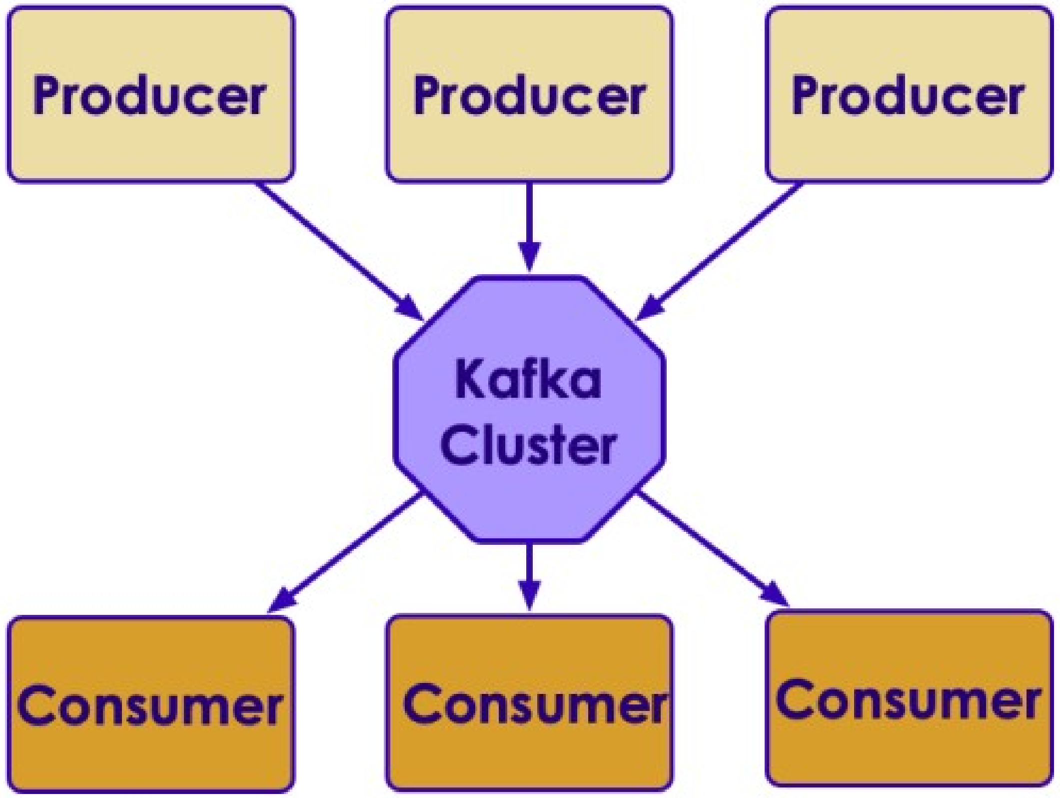
Zookeeper

- Runs as a quorum (multiple nodes)
  - No single point of failure
- Odd number of nodes (3, 5, 7 ...etc)
  - Odd number to break tie when voting
  - Minimum 3 nodes
- Small number of nodes can support thousands of clients



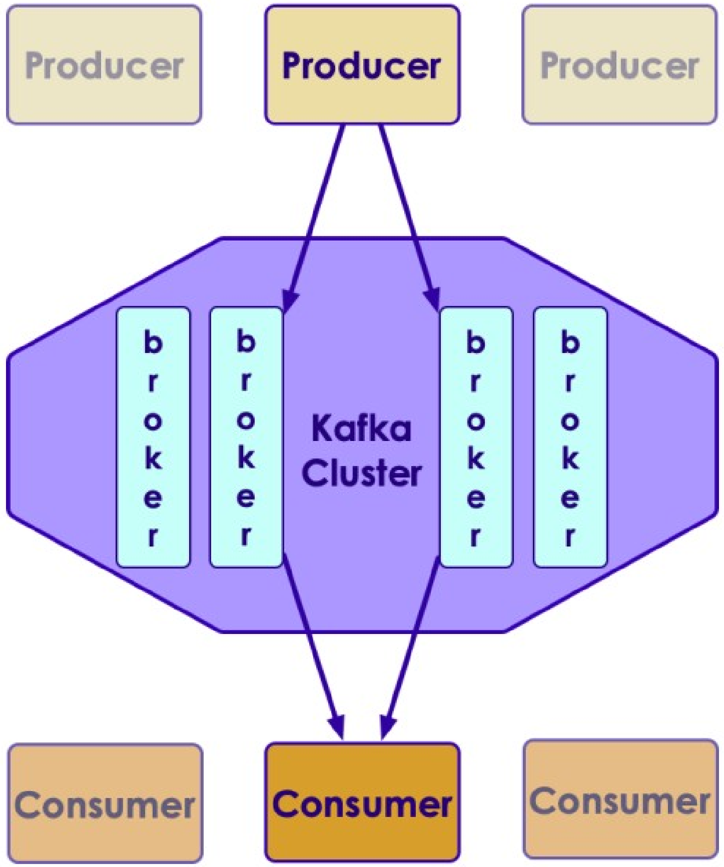
Kafka Architecture

- Kafka is designed as a Pub-Sub messaging system
  - Producers publish messages
  - Consumers consume messages



Kafka Architecture

- Kafka is designed to run on many nodes as a cluster
  - Kafka machines are called 'brokers'
  - Kafka automatically backs up data on at least another machine (broker)



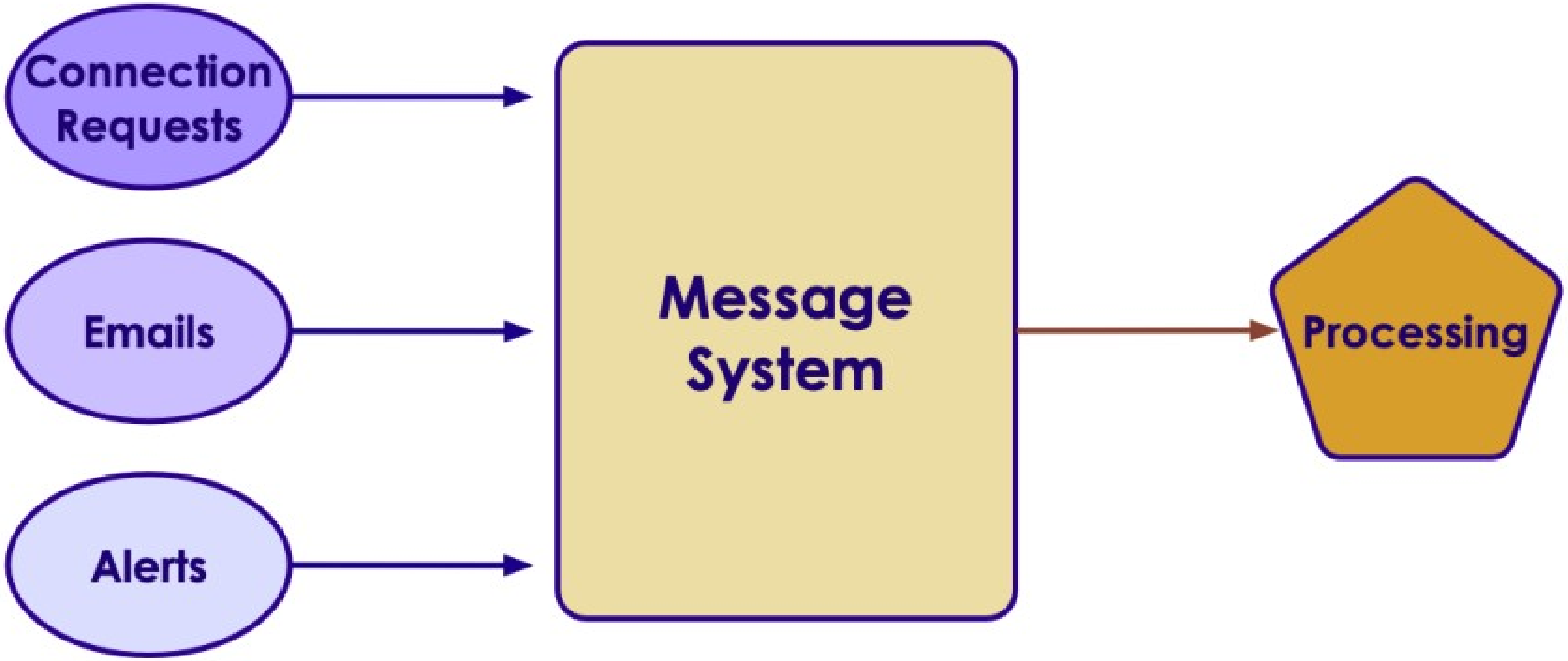
Kafka Terminology

- Roles
  - Producers: write data to Kafka
  - Consumers: read data from Kafka
  - Brokers: Kafka nodes
  - Zookeeper: Keep track of brokers
- Data
  - Message: 'basic unit' of data in Kafka
  - Topics: Messages are organized as topics
  - Partitions: Topics are split into partitions
  - Commit Log: How data is organized
  - Offset: message's position within a partition

A Kafka Use Case: 'My Connect'

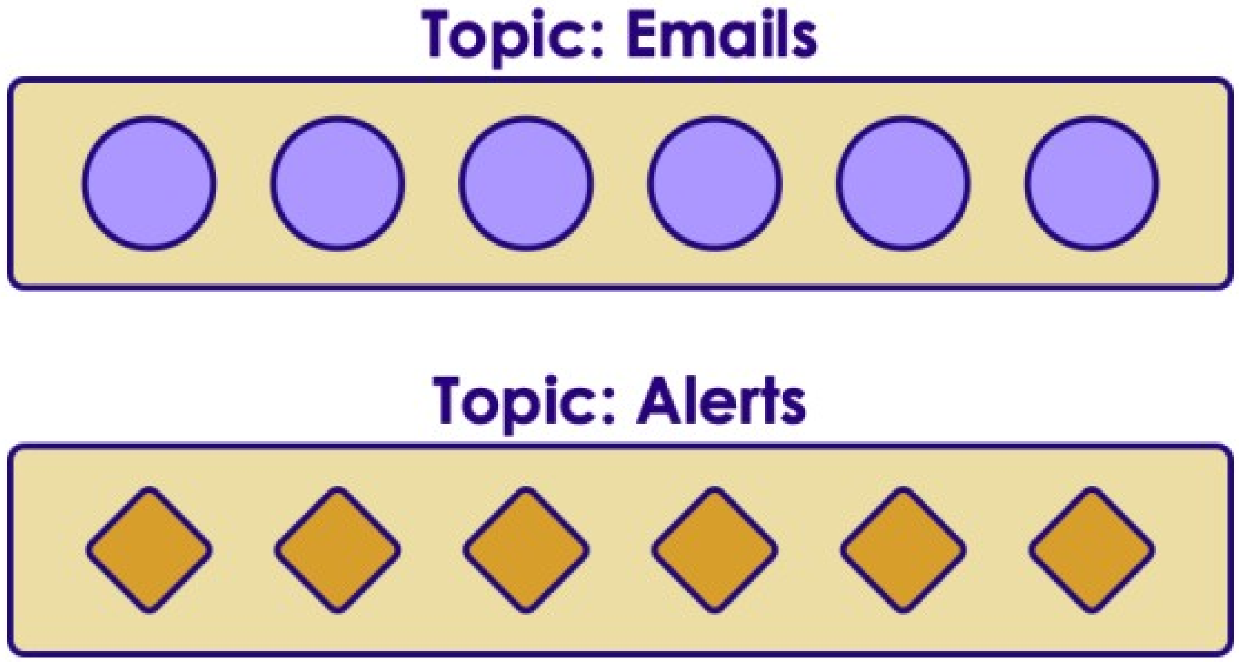
- Features
  - Users can connect with each other
  - Users can send messages to each other
  - Analyze user's usage pattern to customize home page
  - System metrics and diagnostics
- Design
  - We will use a message queue instead of database
  - We are going to send messages for each event
  - Each user email is sent as a message
  - System metrics are sent as events

A Kafka Use Case: 'My Connect'



Kafka Concepts

- In Kafka a basic unit of data is a 'message'
  - Message can be email / connection request / alert event
- Messages are stored in 'topics'
  - Topics are like 'queues'
  - Sample topics could be: emails / alerts

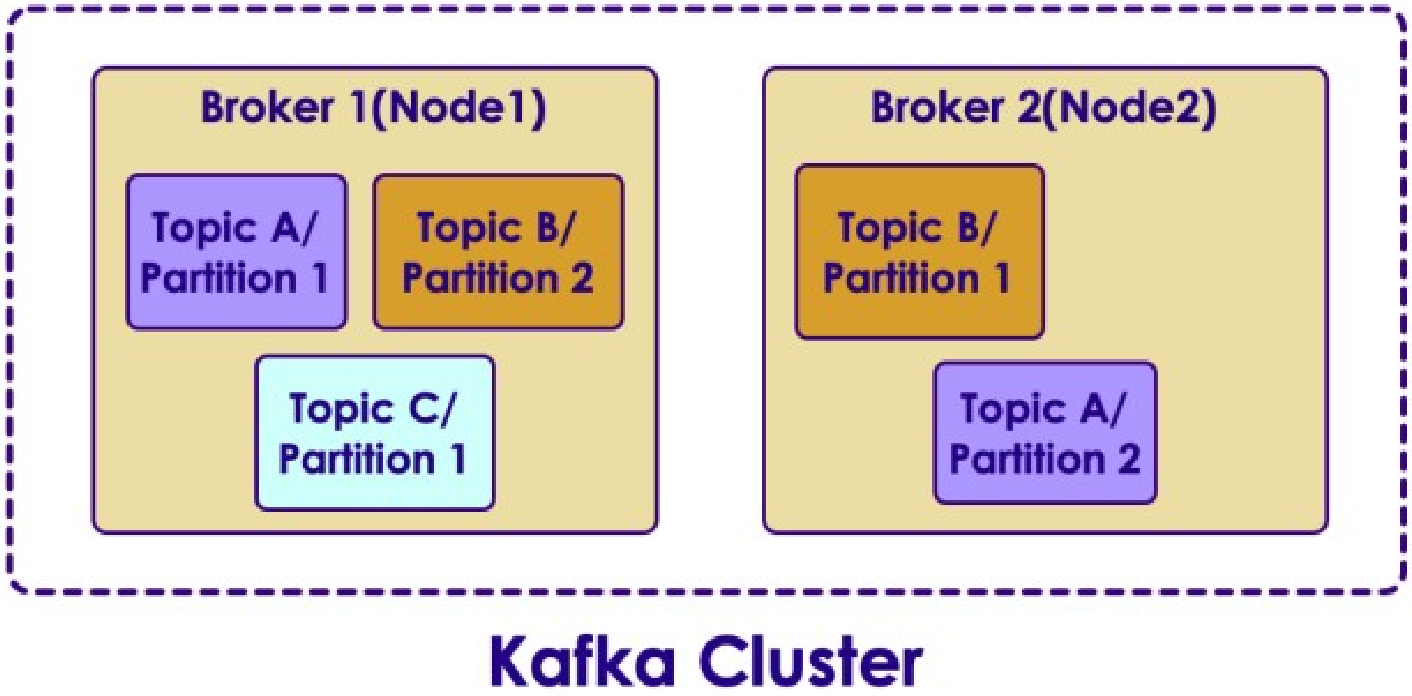


Topics

- Analogous to a 'queue' in a queuing system
  - Logical / virtual entity
  - We can set expiration-times & replication settings per topic
  - Topics are broken into smaller units called partitions

Partitions

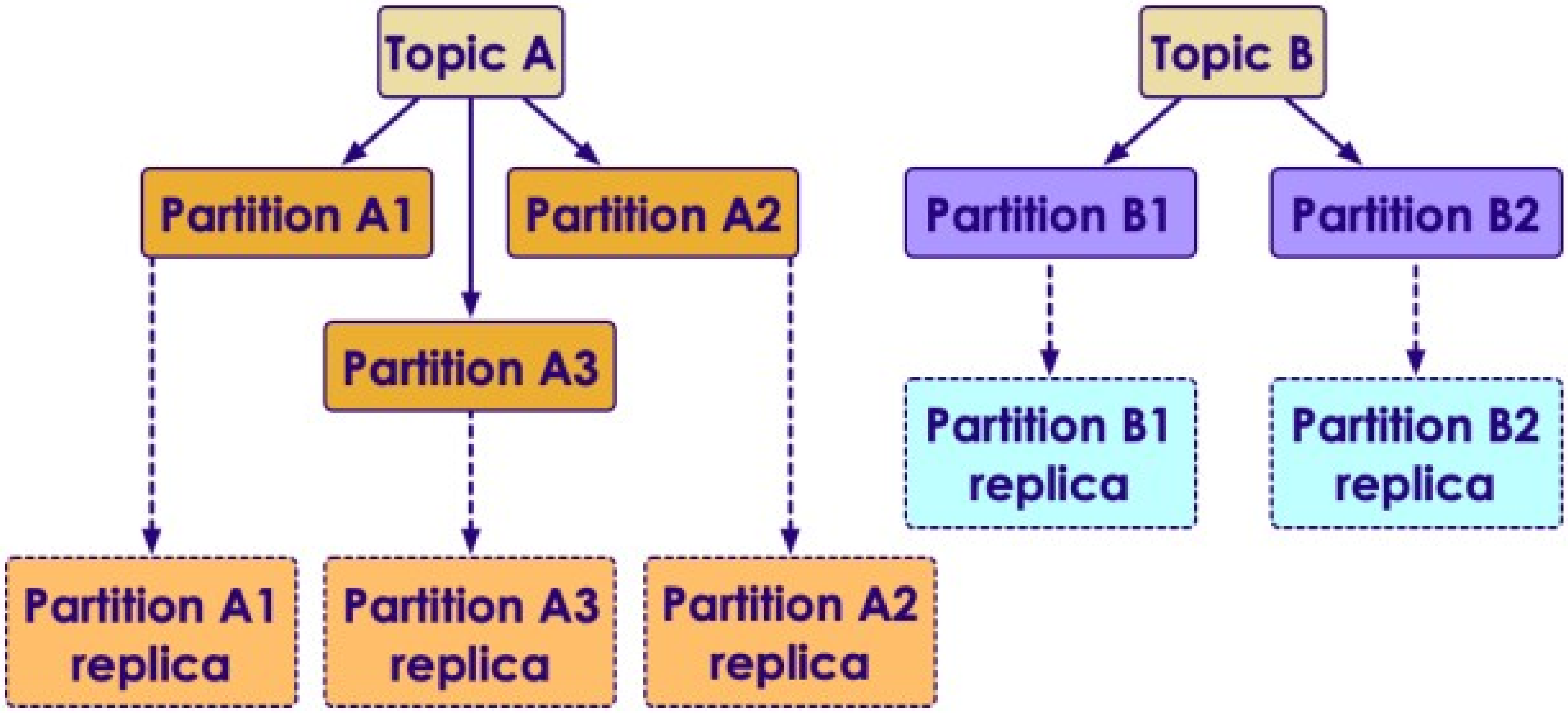
- Partition is a physical entity
  - This is where data lives
  - One partition resides on ONE machine (1 to 1)
  - One machine will host many partitions (N <-> M)
  - Possibly from many topics



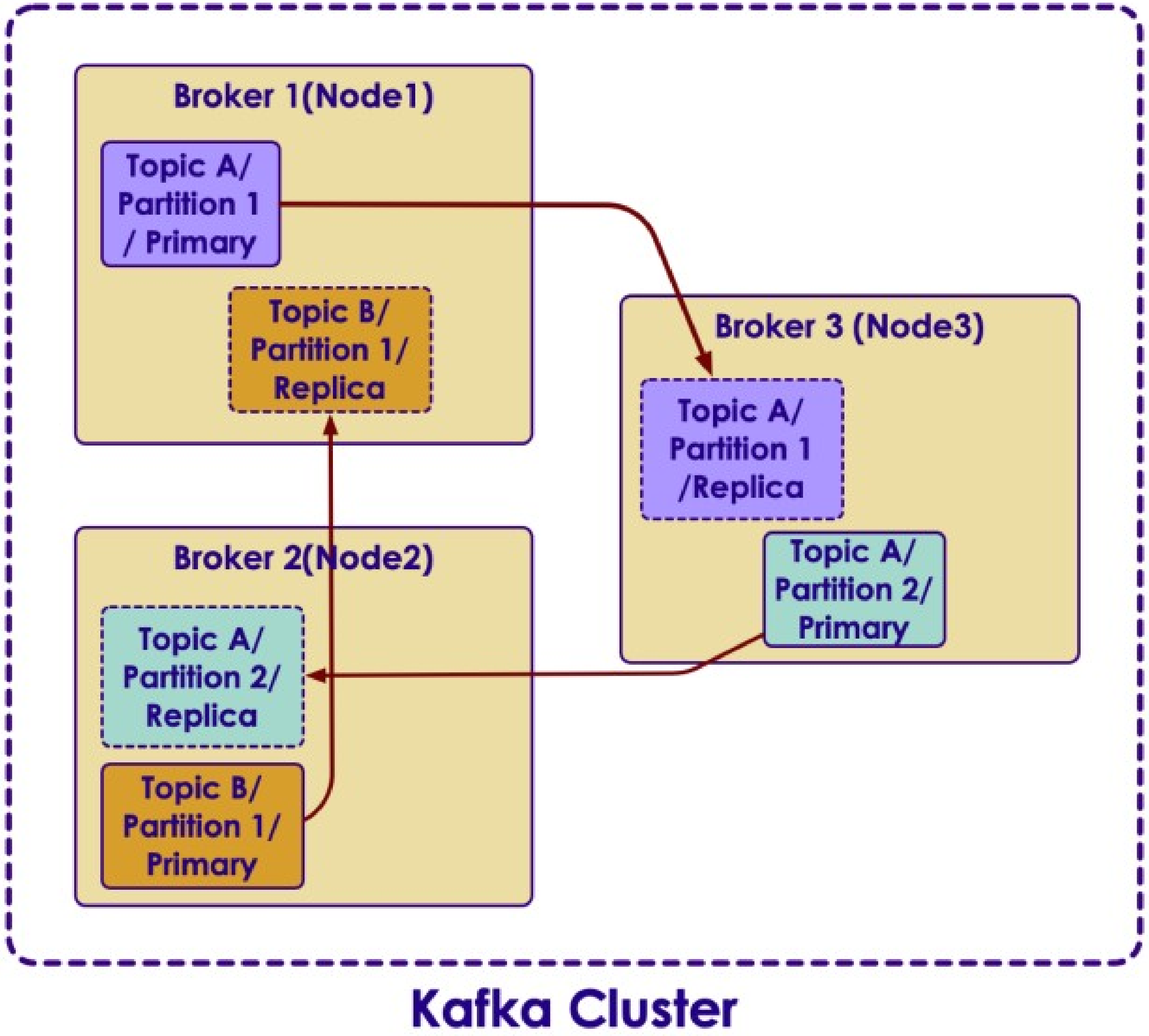
Partitions / Replicas

- One partition is stored in one machine (broker)
  - Partitions are replicated to prevent data loss, in case a machine crashes
  - Default setup is 2 copies (one primary, one replica)
  - One broker is the 'owner' for a partition
  - Replicas are purely there to prevent data loss
  - Replicas are never written to, nor read from so increasing number of replicas does not increase throughput

Partitions / Replicas

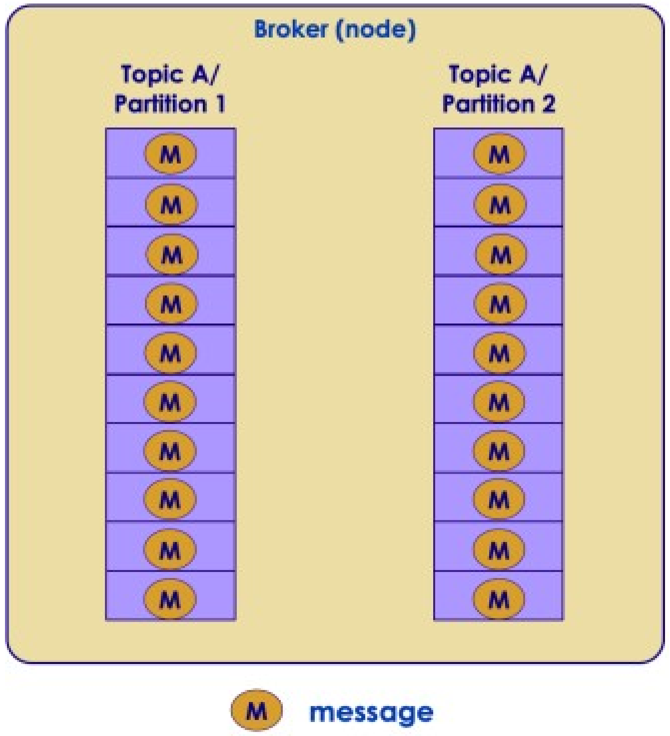


Topics + Partitions + Replicas



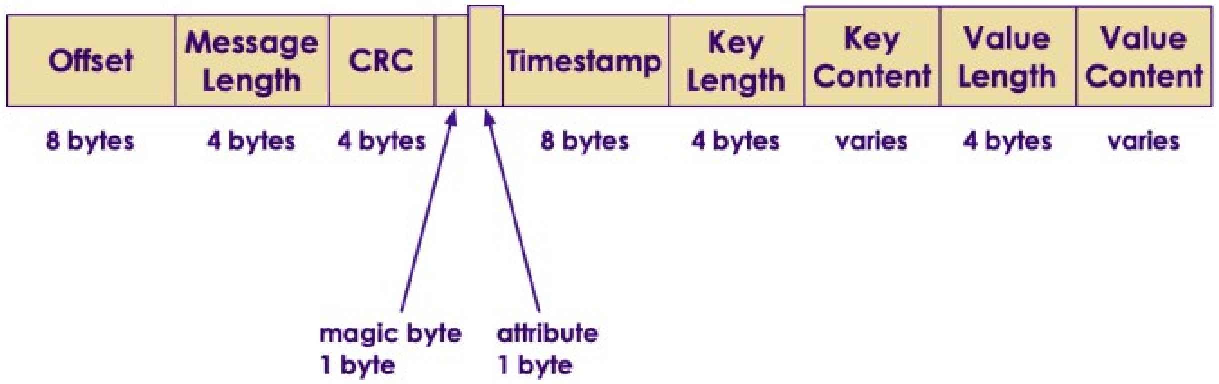
Commit Log

- Commit Log is simple file on disk that stores message bytes
  - Messages are always appended (to the end) of commit log
  - Commit log can not be modified in the middle (immutable)
  - Can read messages in order
  - Provides high concurrency & high throughput with no locking
  - Each Partition has it's own commit log



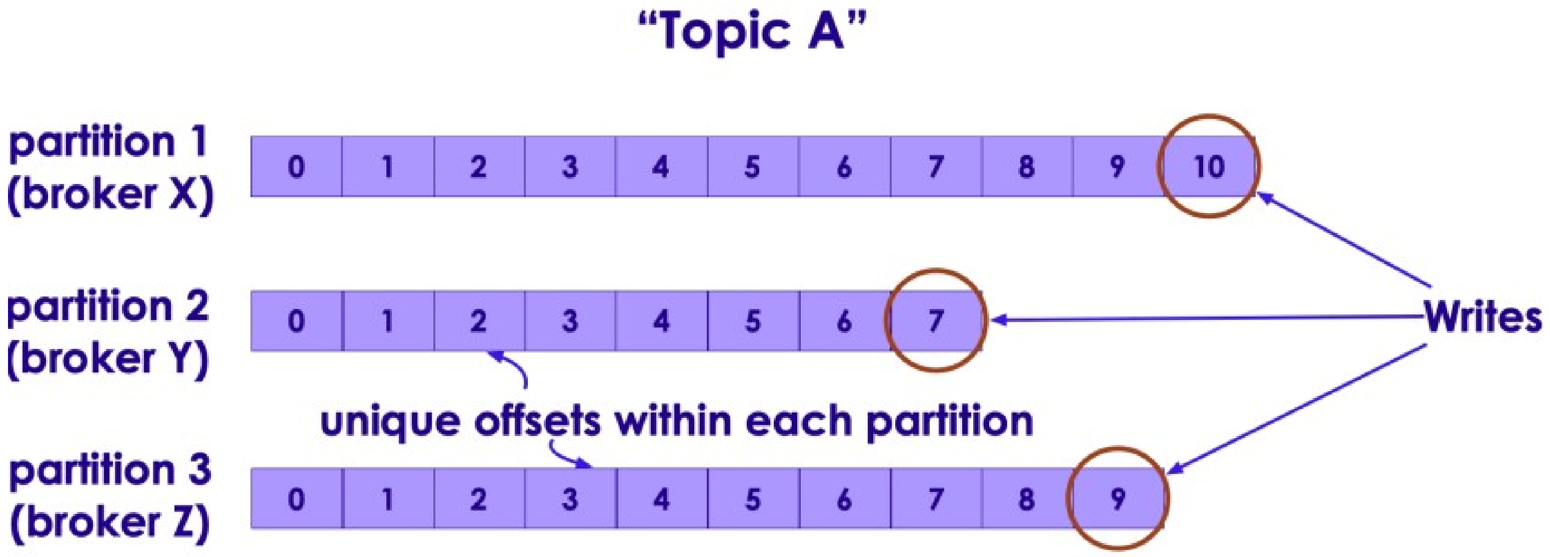
Kafka Message

- In Kafka basic 'data unit' is a message
  - Kafka treats messages as 'bunch of bytes'
  - Doesn't really care what the message payload is
- Optionally messages can have metadata, like keys
  - Keys are bytes too
  - Keys are used to determine which partition to write to
  - Think 'hashing', Same key always go to same
- Messages can have optional schema



Partitions / Messages

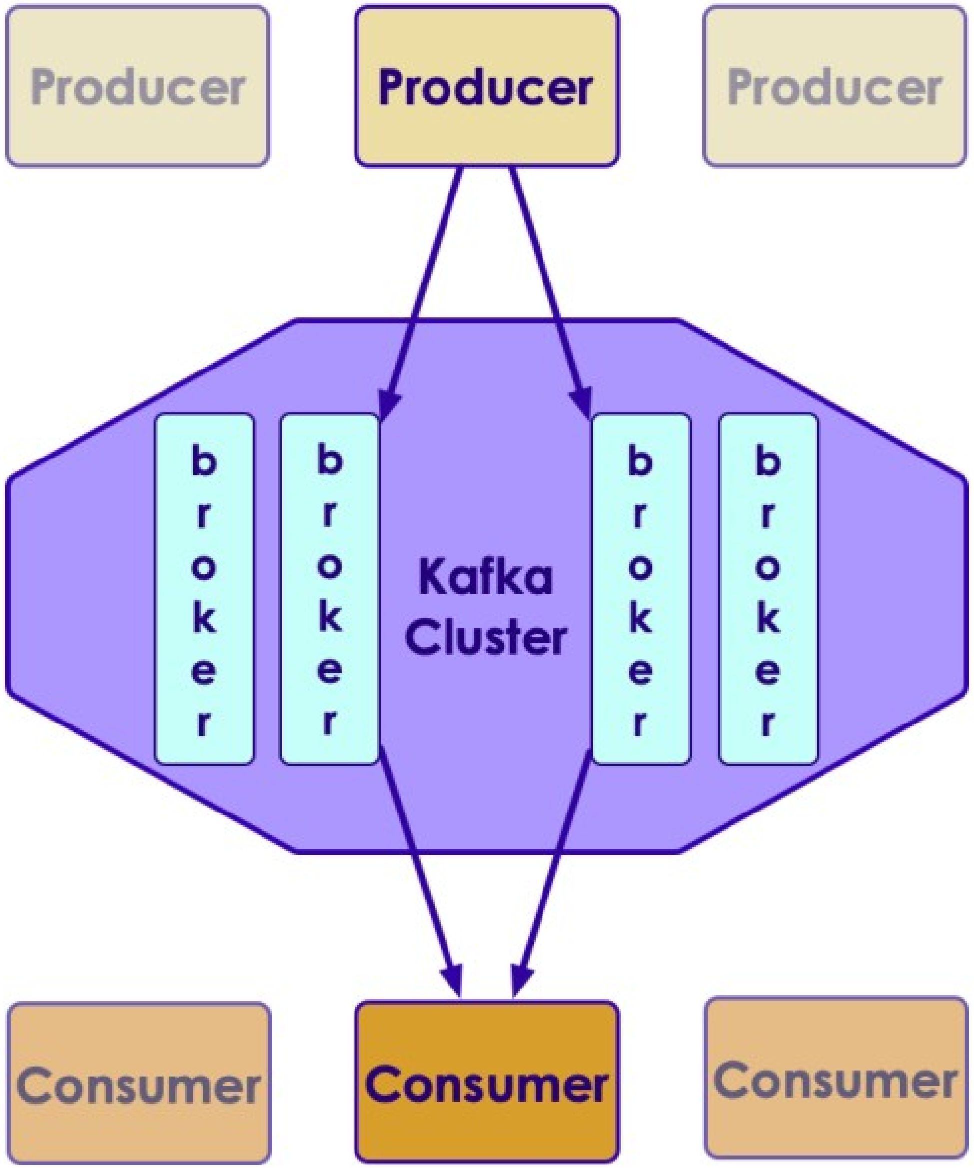
- Messages are written in order on each partition
  - Partitions are ordered and immutable
  - No order maintained across partitions
  - Producers write at the end of partition (append)
  - Sequential writes -> higher throughput



Brokers

- A Kafka broker is a Java process that runs on a node (machine / host)
  - Runs as a daemon (background process)
  - One broker daemon per node
- Brokers are designed to run as cluster
  - Usually bare metal preferred for performance as opposed to virtualized machines
- A single broker can handle thousands of partitions and millions of messages

Brokers

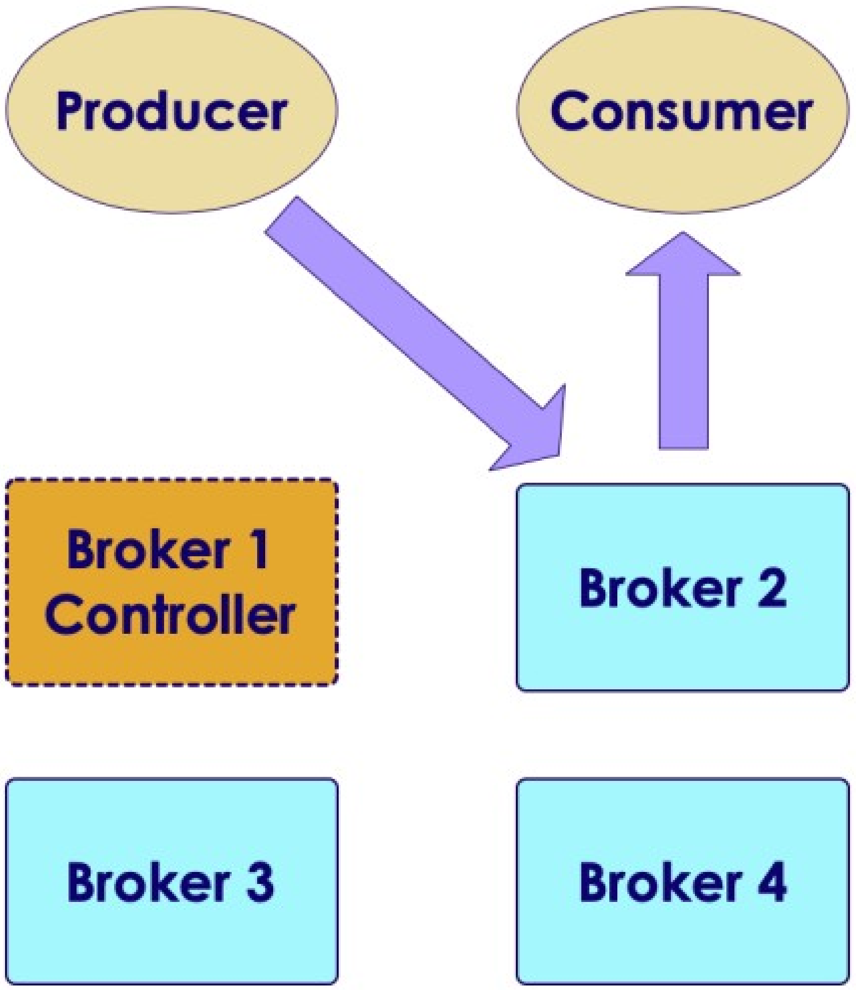


Producer
Producer
Producer
broker
broker
Kafka Cluster
broker
broker
Consumer
Consumer
Consumer

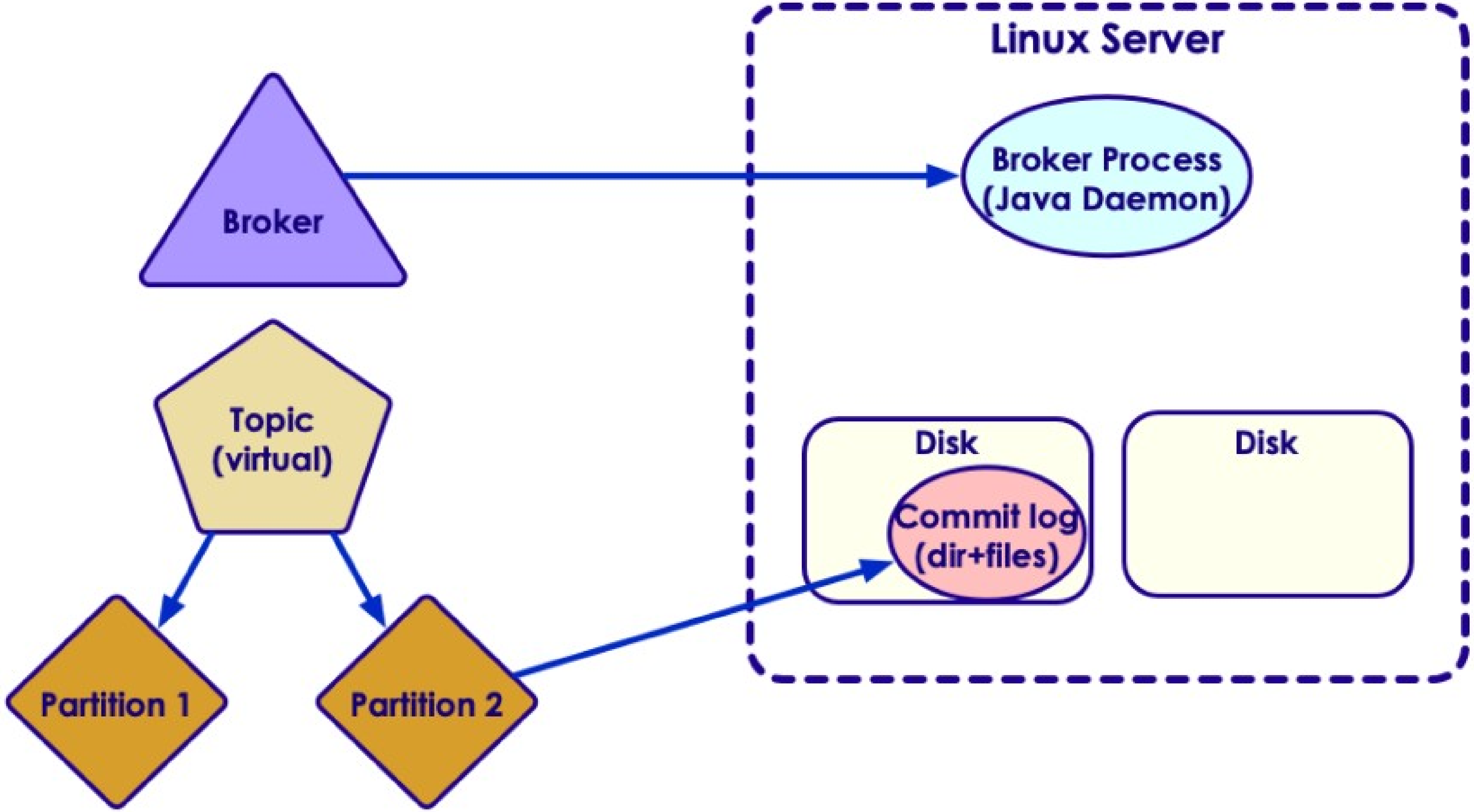
Broker Services

- Cluster
  - One broker is designated as controller / administrator of cluster
  - Selected automatically from all brokers
  - Monitors other brokers and handles failures
  - Assigns partition ownership
- Services to Producer
  - Accepts messages from Producers
  - Assigns a unique offsets (incrementing) to messages
  - Commits the messages to commitlog
- Services to Consumer
  - Serve message requests
  - Assign partitions to consumers in consumer groups

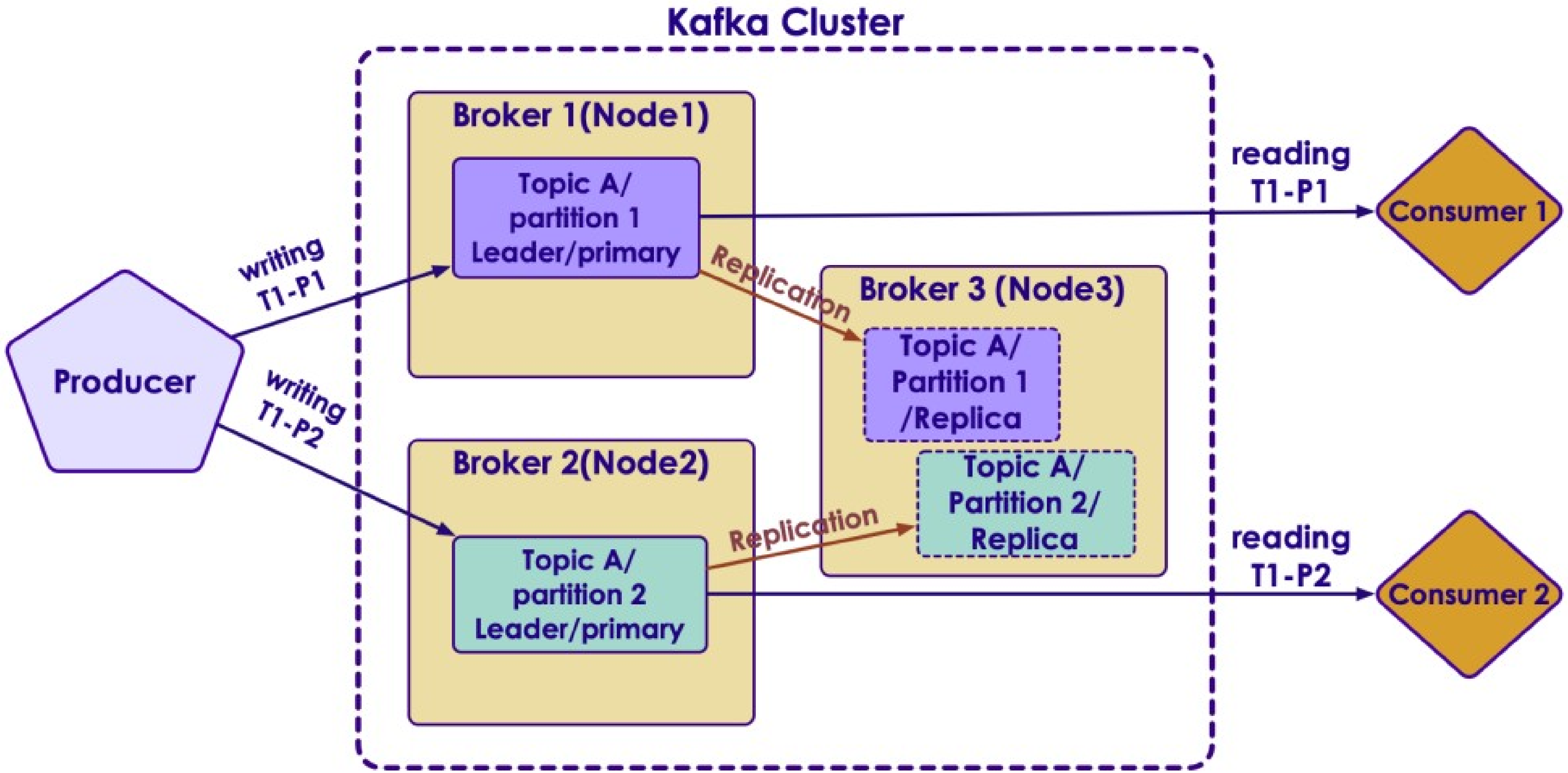
Broker Services



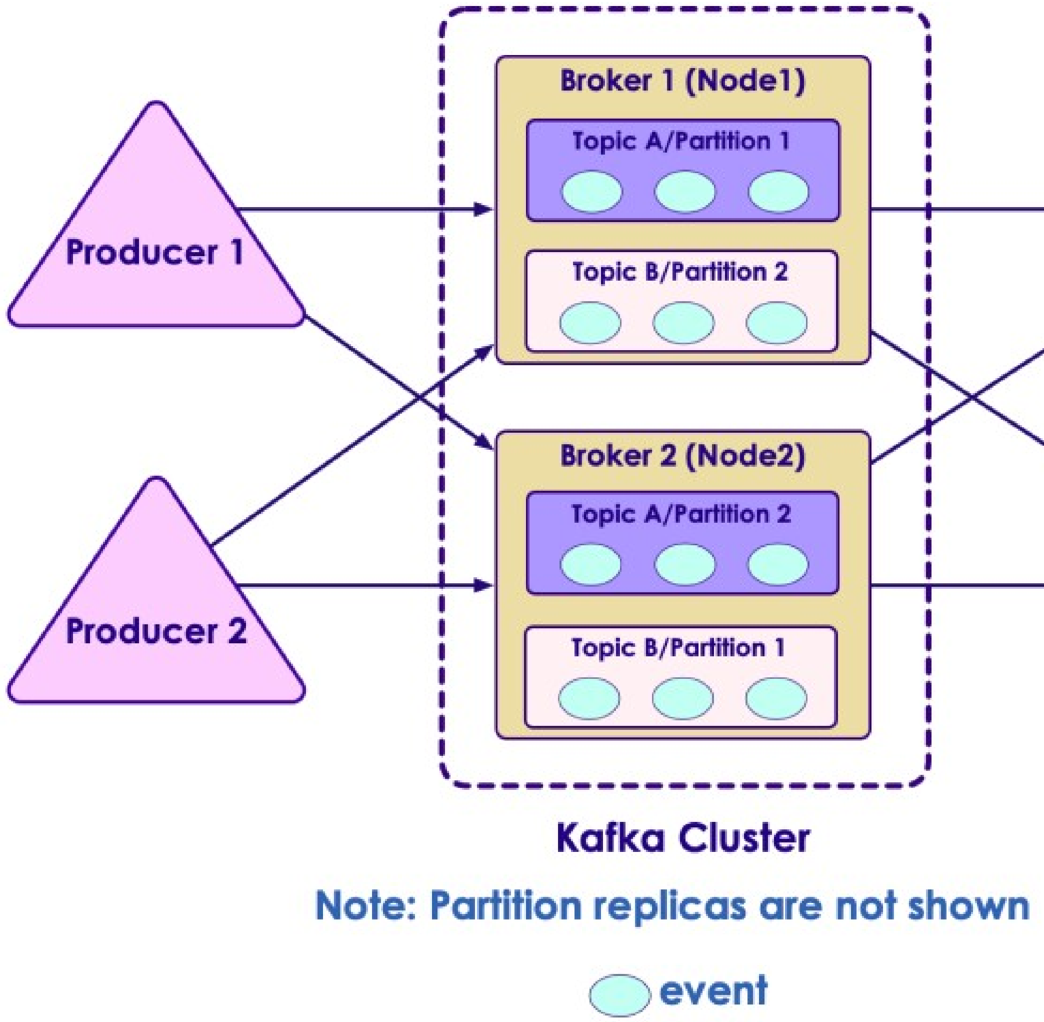
Kafka: Physical and Logical



Brokers / Leaders / Partitions / Replicas



Producers / Consumers / Topics / Partitions



Kafka Command Utilities in BIN

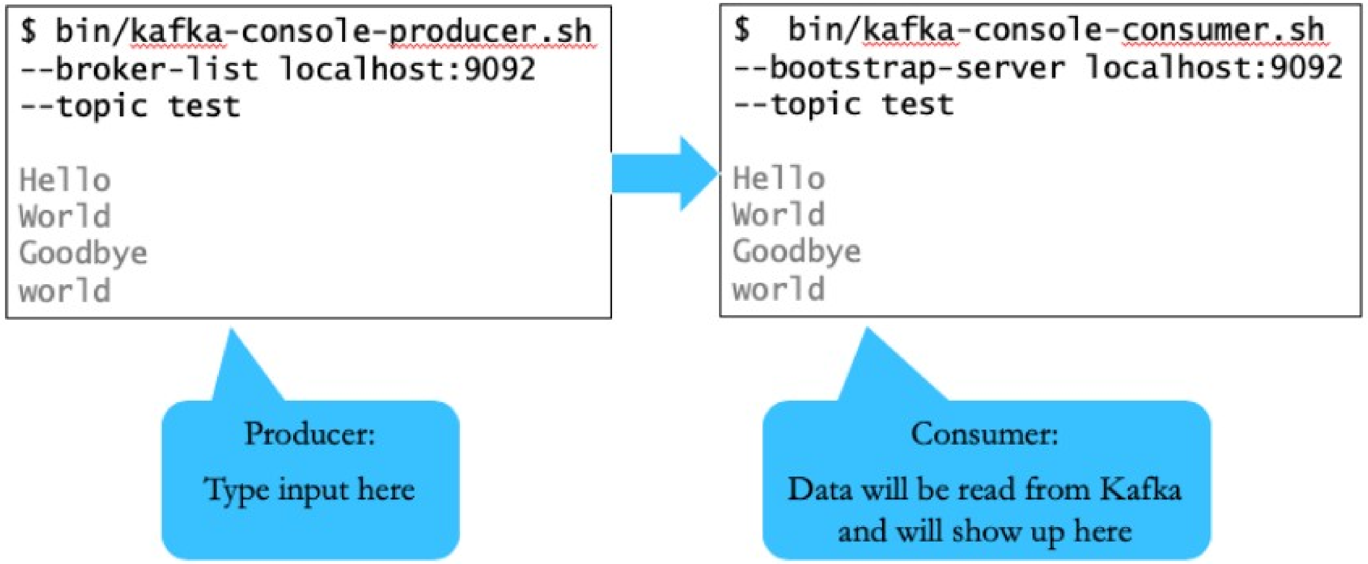
- Starting Kafka brokers
  - bin/kafka-server-start
  - bin/kafka-server-stop
- Managing topics
  - bin/kafka-topics: Lists / create / delete topics
- Sending Messages
  - bin/kafka-console-producer.sh
- Consuming messages
  - bin/kafka-console-consumer.sh

Creating Topics

```
$   bin/kafka-topics.sh  --bootstrap-server localhost:9092 --list
# ... empty ...

## create a topic with one replica and two partitions
$  bin/kafka-topics.sh  --bootstrap-server localhost:9092  --create
--topic test --replication-factor 1  --partitions 2


$  bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic test

# Topic:test    PartitionCount:2    ReplicationFactor:1    Configs:
# Topic: test    Partition: 0    Leader: 0    Replicas: 0    Isr: 0
# Topic: test    Partition: 1    Leader: 0    Replicas: 0    Isr: 0
```

Using Producer / Consumer Utils

- bin/ kafka-console-producer:
  - utility for producing messages
- bin/kafka-console-consumer:
  - utility for reading messages



Kafka Clients

- Java is the 'first class' citizen in Kafka
  - Officially maintained
- Python on par with Java
  - Maintained by Confluent.io
- Other language libraries are independently developed
  - May not have 100% coverage
  - May not be compatible with latest versions of Kafka

Kafka Java API

- Rich library that provides high level abstractions
  - No need to worry about networking / data format ..etc
- Write message / Read message
- Supports native data types
  - String
  - Bytes
  - Primitives (int, long ...etc.)

Java Producer Code (Abbreviated)

```java
// ** 1 **
import java.util.Properties;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.apache.kafka.common.serialization.IntegerSerializer;
...

// ** 2 **
Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ProducerConfig.CLIENT_ID_CONFIG, "SimpleProducer");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

KafkaProducer< Integer, String > producer = new KafkaProducer<>(props);

// ** 3 **
String topic = "test";
Integer key = new Integer(1);
String value = "Hello world";
ProducerRecord < Integer, String > record = new ProducerRecord<> (topic, key, value);
producer.send(record);
producer.close();
```

Producer Code Walkthrough

```java
// ** 2 **  Recommended approach: use constants

import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.apache.kafka.common.serialization.IntegerSerializer

Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ProducerConfig.CLIENT_ID_CONFIG, "SimpleProducer");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

KafkaProducer < Integer, String > producer = new KafkaProducer<>(props);
```

```java
// ** 2 ** another approach
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("client.id", "SimpleProducer");
props.put("key.serializer",  "org.apache.kafka.common.serialization.IntegerSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

KafkaProducer < Integer, String > producer = new KafkaProducer<>(props);
```

Producer Code Walkthrough

- Each record represents a message
  - Here we have a <key,value> message
  - send() doesn't wait for confirmation
- We send in batches
  - For increased throughput
  - Minimize network round trips

```
// ** 3 **
String topic = "test";
Integer key = new Integer(1);
String value = "Hello world";
ProducerRecord< Integer, String > record = new ProducerRecord<> (topic, key, value);
producer.send(record);
producer.close();
```

Producer Properties

```
Properties props = new Properties();
props.put("boostrap.servers", "localhost:9092");
props.put("client.id", "SimpleProducer");
props.put("acks", "all");
props.put("retries", 0);
props.put("batch.size", 16384);  // 16k
props.put("linger.ms", 1);
props.put("buffer.memory", 33554432); // 32 M
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

KafkaProducer < Integer, String > producer = new KafkaProducer<>(props);

for(int i = 0; i < 100; i++) {
  producer.send(new ProducerRecord < String, String >(
      "my-topic", Integer.toString(i), Integer.toString(i)));
}
producer.close();
```

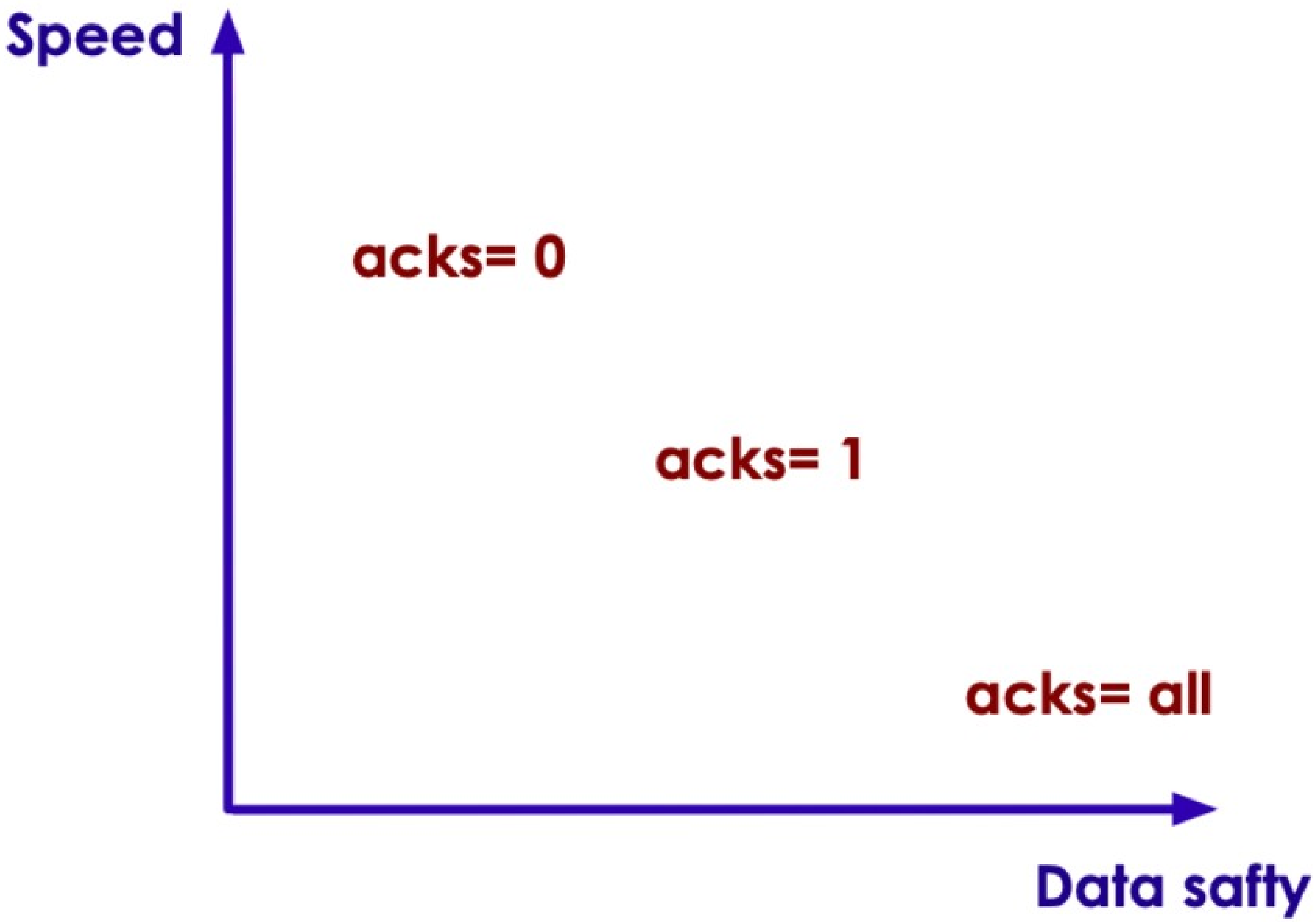
Producer Acknowledgments

ACK	Description	Speed	Data safety
acks=0	- Producer doesn't wait for any acks from broker, - Producer won't know of any errors	High	Low No guarantee that broker received the message
acks=1, **(default)**	- Broker will write the message to local log, - Does not wait for replicas to complete	Medium	Medium Message is at least persisted on lead broker
acks=all	- Message is persisted on lead broker and in replicas, - Lead broker will wait for in-sync replicas to acknowledge the write	Low	High Message is persisted in multiple brokers

Producer Properties

```
Properties props = new Properties();
props.put("boostrap.servers", "localhost:9092");
props.put("client.id", "SimpleProducer");
props.put("acks", "all");
props.put("retries", 0);
props.put("batch.size", 16384);  // 16k
props.put("linger.ms", 1);
props.put("buffer.memory", 33554432); // 32 M
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

KafkaProducer < Integer, String > producer = new KafkaProducer<>(props);

for(int i = 0; i < 100; i++) {
  producer.send(new ProducerRecord < String, String >(
      "my-topic", Integer.toString(i), Integer.toString(i)));
}
producer.close();
```

Producer Acknowledgments



Consumer Code (Abbreviated)

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.kafka.common.serialization.IntegerDeSerializer

...

Properties props = new Properties(); // ** 1 **
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ConsumerConfig.GROUP_ID_CONFIG, "group1");
props.put(ConsumerConfig.CLIENT_ID_CONFIG, "Simple Consumer");
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, IntegerDeSerializer.class.getName());
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());


KafkaConsumer < Integer, String > consumer = new KafkaConsumer<>(props);

consumer.subscribe(Arrays.asList("topic1")); // ** 2 **

try {
    while (true) {
       ConsumerRecords < Integer, String > records = consumer.poll(Duration.ofMillis(1000)); // ** 3 **
       System.out.println("Got " + records.count() + " messages");
       for (ConsumerRecord < Integer, String > record : records) {
          System.out.println("Received message : " + record);
       }
    }
}
finally {
    consumer.close(Duration.OfSeconds(60));
}
```

Consumer Code Walkthrough

- bootstrap,servers: "broker1:9092,broker2:9092"
  - Connect to multiple brokers to avoid single point of failure
  - group.id: consumers belong in a Consumer Group
  - We are using standard serializers
- Consumers can subscribe to one or more subjects // ** 2 **

```
Properties props = new Properties(); // ** 1 **
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ConsumerConfig.GROUP_ID_CONFIG, "group1");
props.put(ConsumerConfig.CLIENT_ID_CONFIG, "Simple Consumer");
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, IntegerDeSerializer.class.getName());
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeSerializer.class.getName());


KafkaConsumer < Integer, String > consumer = new KafkaConsumer<>(props);

consumer.subscribe(Arrays.asList("topic1")); // ** 2 **
```

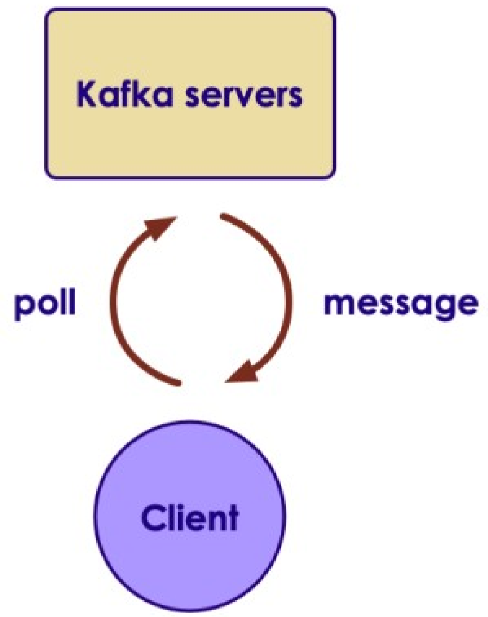
Consumer Code Walkthrough

- Consumers must subscribe to topics before starting polling
  - Consumer.subscribe ("test.*") // wildcard subscribe
  - Poll: This call will return in 1000 ms, with or without records
  - Must keep polling, otherwise consumer is deemed dead and the partition is handed off to another consumer

```
try {
   while (true) {
      ConsumerRecords < Integer, String > records = consumer.poll(Duration.ofMillis(1000)); // ** 3 **
      System.out.println("Got " + records.count() + " messages");
      for (ConsumerRecord < Integer, String > record : records) {
        System.out.println("Received message : " + record);
      }
   }
}
finally {
   consumer.close();
}
```

Consumer Poll Loop

- Polling is usually done in an infinite loop.
  - First time poll is called
  - Finds the GroupCoordinator
  - Joining Consumer Group
  - Receiving partition assignment
- Work done in poll loop
  - Usually involves some processing
  - Saving data to a store
  - Don't do high latency work between polls; otherwise the consumer could be deemed dead.
- Do heavy lifting in a separate thread



ConsumerRecord

- org.apache.kafka.clients.consumer.ConsumerRecord <K,V>
  - K key(): key for record (type K), can be null
  - V value(): record value (type V - String / Integer ..etc)
- String topic(): Topic where this record came from
- int partition(): partition number
- long offset(): long offset in

```
ConsumerRecords < Integer, String > records = consumer.poll(Duration.ofMillis(1000));
  for (ConsumerRecord < Integer, String > record : records) {
     System.out.printf("topic = %s, partition = %d, offset = %d,
            key= %s, value = %s\n",
            record.topic(), record.partition(), record.offset(),
            record.key(), record.value());
  }
```

Configuring Consumers

- max.partition.fetch.bytes (default : 1048576 (1M))
  - Max message size to fetch. Also see message.max.bytes broker config
- session.timeout.ms (default : 30000 (30 secs))
  - If no heartbeats are not received by this window, consumer will be deemed dead and a partition rebalance will be triggered

```
Properties props = new Properties(); // ** 1 **

...
props.put("session.timeout.ms", 30000); // 30 secs
props.put("max.partition.fetch.bytes", 5 * 1024 * 1024); // 5 M

KafkaConsumer < Integer, String > consumer = new KafkaConsumer<>(props);
```

Clean Shutdown Of Consumers

- Consumers poll in a tight, infinite loop
- Call ' consumer.wakeup () ' from another thread
- This will cause the poll loop to exit with ‘ ‘WakeupException '

```
try {
  while (true) {
   ConsumerRecords < Integer, String > records = consumer.poll(100);
    // handle events
 }
}
catch (WakeupException ex) {
   // no special handling needed, just exit the poll loop
}
finally {
   // close will commit the offsets
   consumer.close();
}
```

Signaling Consumer To Shutdown

- Can be done from another thread or shutdown hook
- ' consumer.wakeup () ' is safe to call from another thread

```
Runtime.getRuntime().addShutdownHook(new Thread() {
    public void run() {
        System.out.println("Starting exit...");
        consumer.wakeup(); // signal poll loop to exit
        try {
            mainThread.join(); // wait for threads to shutdown
        } catch (InterruptedException e) {
            e.printStackTrace();
    }
  }
}
```