

Week 13: Streaming Data

Keeyong Han

Table Of Contents

1. Internships
2. Recap of Week 12
3. What is Streaming Data?
4. Characteristics of Streaming Data Processing
5. Challenges of Streaming Data Processing
6. Break
7. Quiz #3 Review
8. Kafka Overview
9. Kafka Practice
10. Demo & Homework
11. Group Project Presentation
12. Next Week

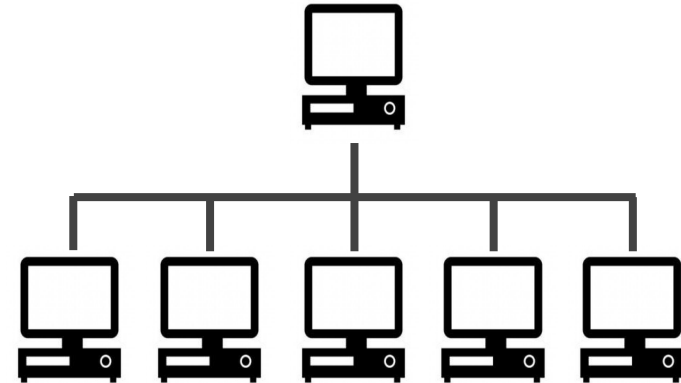
Finding Internship

- Look for referrals
- Search by specific skills in job sites

Recap of Week 12

Characteristics of Big Data Processing

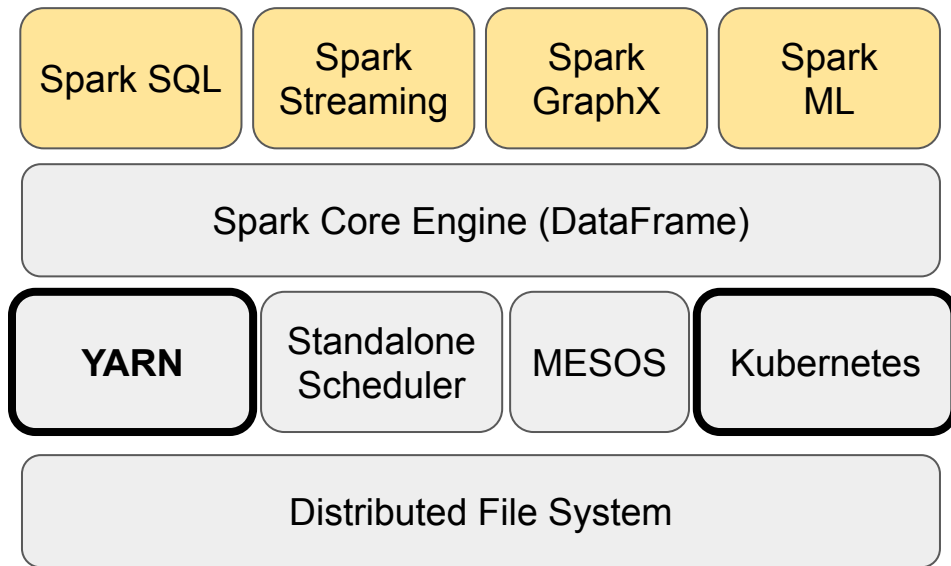
Challenges	Solutions
Need a way to store large volumes of data without loss	Distributed File System: HDFS, S3, Cloud Storage (Data Lake)
Sequential processing takes too long: Computing Power + Parallel Processing	Distributed Computing System MapReduce -> Hive/Presto -> Spark
Big data is often unstructured: SQL alone is not sufficient	DataFrame + SQL



Distributed System
Of storage and computing

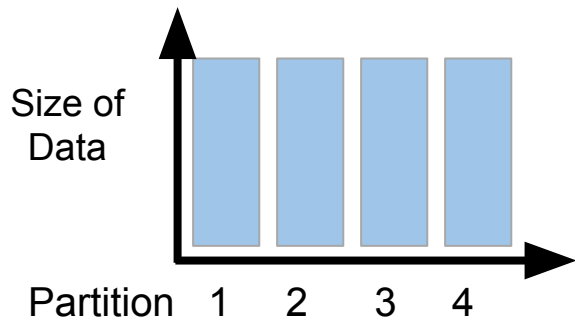
Spark System Overview

- An Apache open-source project in 2012 from UC Berkeley's AMPLab
 - Databricks
- Second-generation big data technology following Hadoop
- Provides diverse functionalities for big data processing!
 - DataFrame, SQL, ML, Streaming, Graph, ...

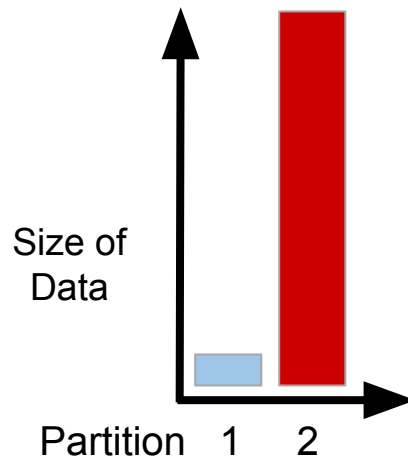


Key Concepts to Remember

- **Partitions:** data is split into multiple partitions (128MB or 256MB)
 - Each partition will be processed by a worker (CPU)
 - This is how parallelism is implemented
- When **shuffling** happens, new partitions will be created
 - GROUP BY or JOIN typically make shuffling happen
 - Some partitions can be too large -> **Data Skewness**

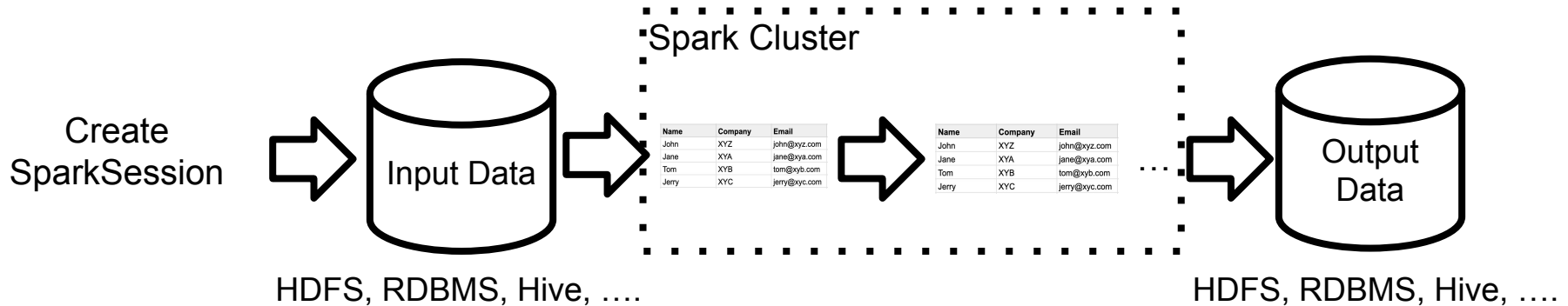


Shuffling



Overall Spark Program Flow

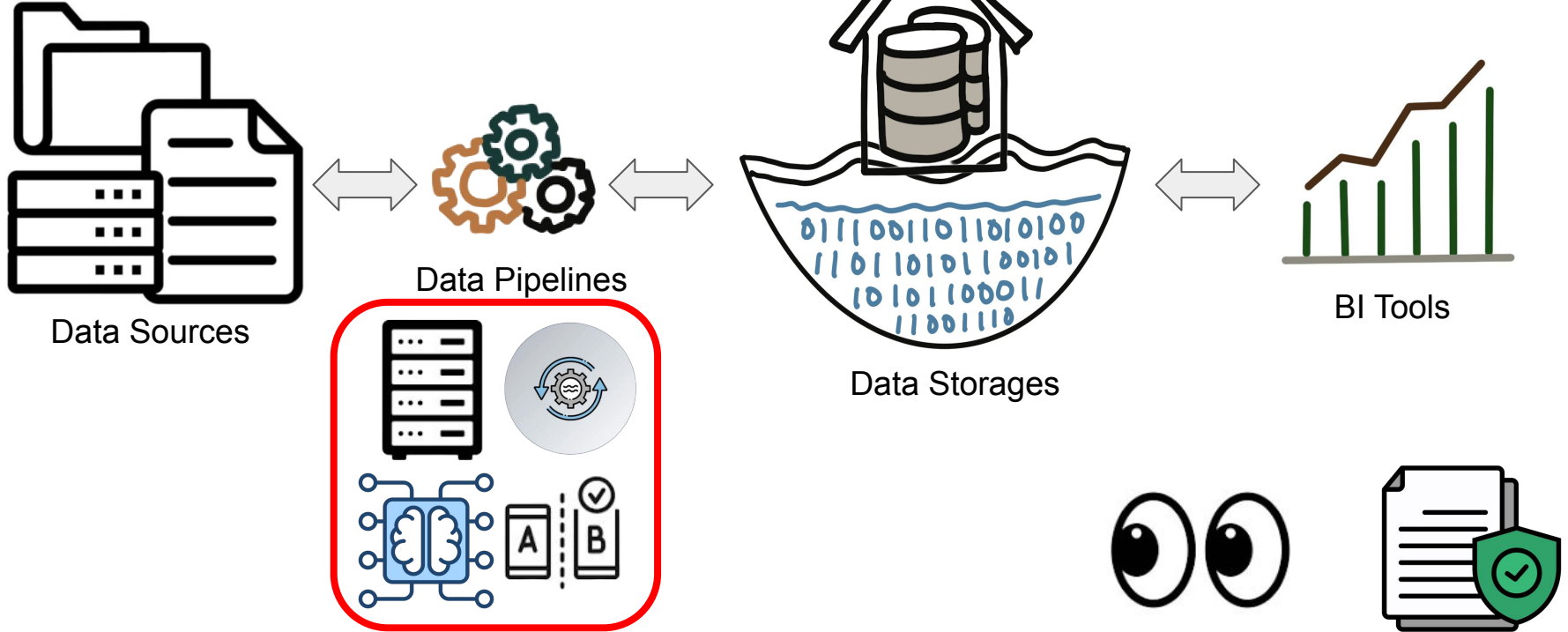
1. Creating a SparkSession
2. Loading the input data as a DataFrame
3. Performing data manipulation tasks (very similar to Pandas)
4. Continuously creating new DataFrames until the desired result is achieved
5. Saving the final result
6. Example code: [Pandas to Spark](#)



Quiz #3 Review

What is Streaming Data?

Data System Diagram



Data System: Advanced Data Processing Layer

- Next Steps after foundation (data warehouse + data pipelines)
 - Small Data -> Big Data
 - Batch Processing -> Streaming Processing
 - ML models used in services
 - A/B testing becomes standardized
- The data team processes, models, and derives insights from data
 - ML Pipeline (training data collection, model building, model testing, model deployment, model monitoring)
- Key Tools:
 - Spark, NoSQL
 - Kafka/Kinesis
 - SageMaker, MLflow

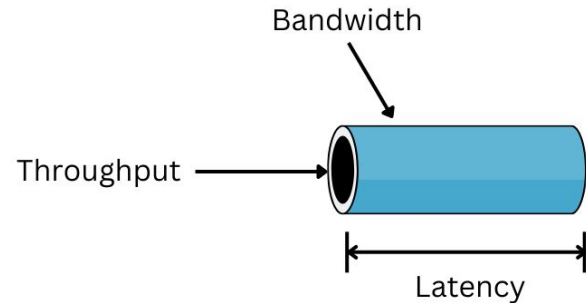


Evolution of Data Processing

- Initially, Processing begins with batch
 - At this stage, the volume of data that can be processed is critical.
- The demand for real-time processing gradually increases over time
 - Real-time Processing vs. Semi-Real-time Processing
 - The need for shared data consumption grows: multiple data consumers emerge
 - Latency over throughput

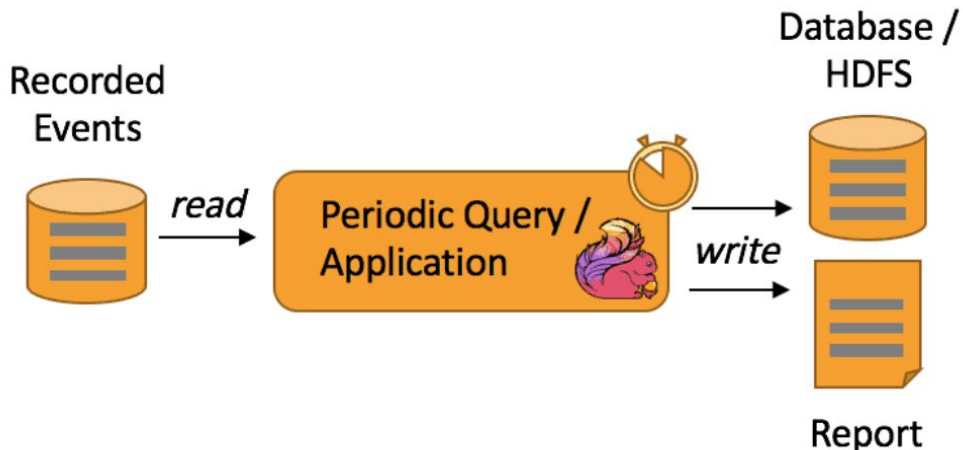
Throughput vs. Latency

- **Throughput**
 - The amount of data that can be processed within a given time unit (e.g., MB per second)
 - Higher throughput means a larger volume of data can be processed. This is especially important in batch systems (e.g., data warehouses).
- **Latency**
 - The time it takes to process data.
 - Lower latency means faster response times, which is crucial in real-time systems (e.g., production databases and NoSQL).
- **Bandwidth = Throughput x Latency**



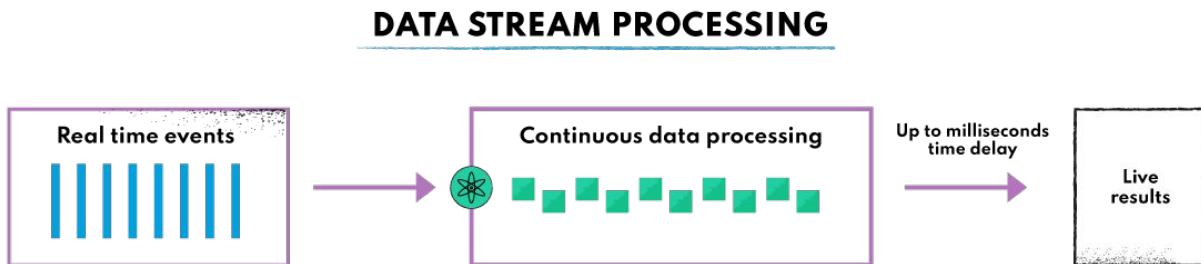
Batch Processing

- Periodically move or process data from one location to another
 - Throughput is critical
 - Processing intervals are typically in the range of minutes, hours, or days
- Data is collected and processed in batches
 - Pandas, Spark DataFrames, SparkSQL, ...
 - Airflow is commonly used for scheduling tasks



Streaming Data Processing

- Continuous Data Processing (vs. Batch Data Processing)
 - This type of data is called event or message at the individual record level
 - Some kind of a queue is needed to store events/messages
 - Producer and Consumer are needed
- Low Latency is very critical
 - Realtime vs. Semi-realtime (micro batch)



Characteristics of Streaming Data Processing

Streaming Data Processing (1)

- Next level of advancement after batch processing
 - The complexity of tasks like system management increases
- Continuous data processing at sub-second intervals
 - This data is typically called "events," which are characterized by being immutable
 - The ongoing flow of events is referred to as an "event stream" or "data stream"



Streaming Data Processing Structure

- A **Producer (Publisher)** generates data
- The generated data is stored in a system like a message queue
 - a. Systems such as Kafka, Kinesis, and Pub/Sub are available
 - b. Each data stream (called a topic in Kafka) can have its own data retention period
- **Consumers (Subscribers)** reads and processes data from the queue
 - a. Each consumer maintains a separate pointer
 - b. Multiple consumers can read data concurrently



Streaming Data Technologies

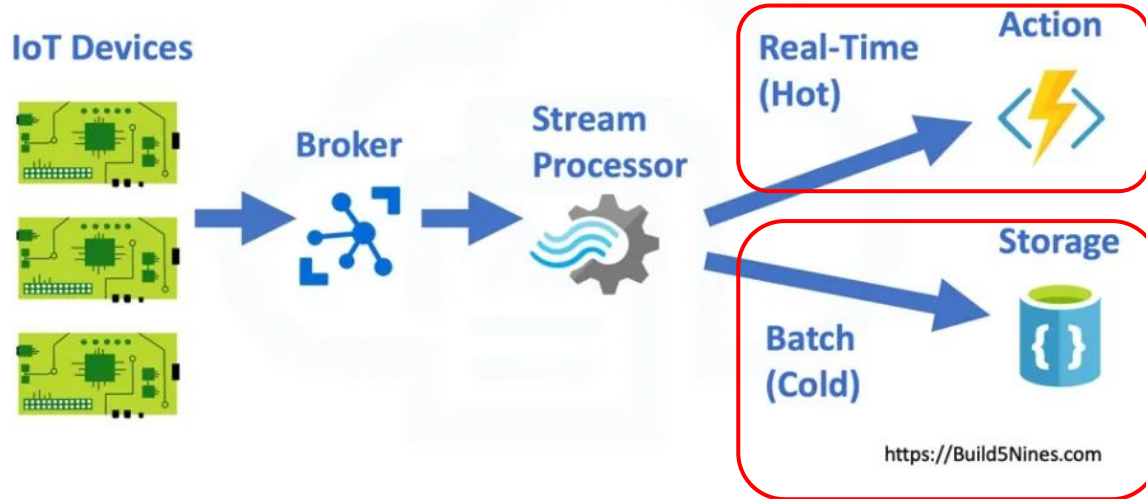
- Needs for different types of services arise
 - Message queues for storing event data: Kafka, Kinesis, Pub/Sub, etc.
 - Processing systems for handling event data: Spark Streaming, Samza, Flink, etc.
 - Analytics/Dashboards for this type of data analysis: Druid



Lambda Architecture

- Operating separate batch and real-time layers

Lambda Architecture for IoT & Big Data



Benefits of Streaming Data Handling

- Immediate insight discovery
- Improved operational efficiency
- Quick response to events like incidents or user signup
- More efficient, and personalized user experiences
- Utilization of IoT and sensor data
- Fraud detection and security
- Real-time collaboration and communication

Downsides of Streaming Data Handling

- Overall system complexity increases
 - Batch systems operate periodically and are usually not directly exposed to end users.
 - In real-time processing, however, it's more likely to be used in user-facing scenarios, making it essential to handle system failures promptly.
 - Batch recommendations vs. real-time recommendations
 - Begins to enter the realm of DevOps
- This results in increased operational costs.
 - Batch processing generally has a low risk of data loss even if something goes wrong
 - But real-time processing has a higher risk, so data backup must always be prioritized.

Streaming Data Processing: Realtime vs. Semi-Realtime

- Realtime

- Short latency
- Continuous data stream
- Event-driven architecture: tasks or computations are triggered by incoming data events
- Dynamic and responsive: performs real-time analytics, monitoring, and decision-making in response to changes in the data stream

- Semi-realtime

- Reasonable Latency
- Processing similar to batch (**micro-batch**): more like very short periodic updates
- Balance between timeliness and efficiency: may sacrifice some immediacy to optimize processing capacity and resource utilization

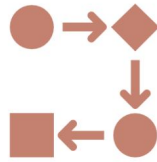
Challenges of Streaming Data Processing

Real-Time Data Processing Stages

- Determine Event Data Model
- Transmit/Store Event Data
- Process Event Data
- Monitor and Resolve Event Data Management Issues



Determine Event Data Model



Transmit/Store Event Data



Process Event Data

Determine Event Data Model

```
{  
  "TransactionID": "100001",  
  "CreatedTime": 1550149860000,  
  "SourceAccountID": "131100",  
  "TargetAccountID": "151837",  
  "Amount": 3000  
}
```

Payment API



A minimum of a Primary Key and Timestamp is required!

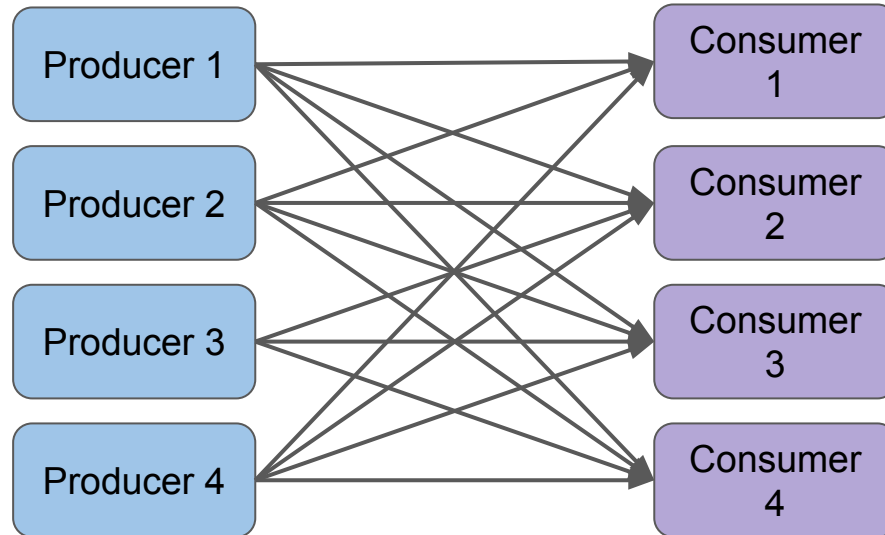
- User information may also be needed.
- Detailed information about the event itself is necessary.

Event Data Storage Options

- Point to Point
 - Many to Many connections are required
- Messaging Queue
 - Work with an intermediary data store to keep the producer and consumer decoupled

Event Data Storage Options: Point to Point

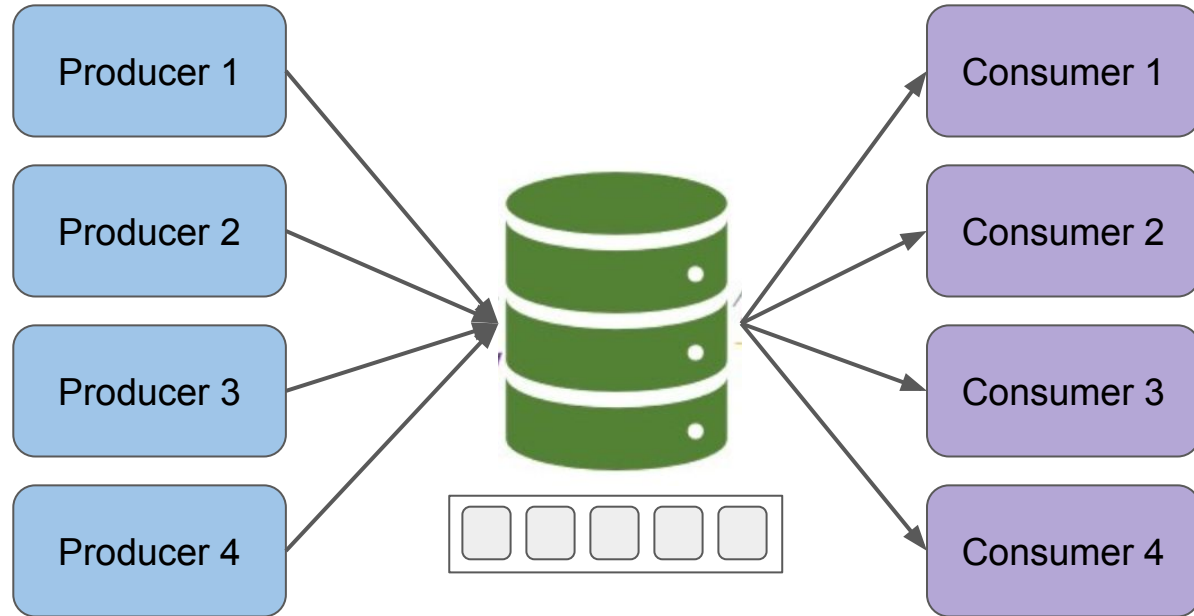
- Throughput is important, but usable in systems where latency is critical.
- Many API layers operate in this way.
- When there are multiple consumers, data may need to be sent redundantly



Backpressure

- In streaming systems, data is typically generated at a steady rate
 - However, sometimes data generation can surge dramatically
 - Producer side
- Data needs to be processed in a timely manner at downstream stages
 - If the system cannot keep up with the incoming data rate, data accumulates, leading to delays, increased memory usage, and potential system failures—this is known as a **backpressure** issue
 - Consumer side
- One way to reduce backpressure is to introduce a queue in between.
 - While this approach can alleviate backpressure issues, it does not completely solve them.

Event Data Storage Options: Messaging Queue



A queue is assigned per Producer

Process Event Data

- Data processing models are determined by the data storage model
- In Point-to-Point:
 - The burden on the Consumer side is heavy, and data must be processed immediately (Backpressure).
 - There is a high possibility of data loss.
 - Typically characterized by low throughput and low latency
- In Messaging Queue
 - Data is usually aggregated and processed in very short intervals, known as micro-batches.
 - Spark Streaming is a prime example.
 - It is easy to support multiple Consumers, which is an advantage.
 - Generally easier to manage compared to Point-to-Point.

Kafka Overview

Origin of Kafka

- Developed by LinkedIn in 2008 for internal real-time data processing
 - Written in Scala and Java
 - In 2014, the developers of Kafka left LinkedIn to found a company called Confluent
 - In 2021, Confluent went public on the NASDAQ
- Open-sourced in early 2011 (Apache)
 - <https://kafka.apache.org/>
- Currently, over 80% of Fortune 100 companies use Kafka



What is Kafka?

- Distributed streaming platform designed for real-time data processing
 - A distributed commit log that allows data replay
 - A publish-subscribe messaging system providing scalability and fault tolerance
 - Producer-Consumer model
- Optimized for **high throughput** and **low latency** real-time data processing
- Follows a distributed architecture, enabling scalability through scale-out
 - Scalability is achieved by adding servers (Scale Out)
 - Messages are stored for a specified retention period
- Rich Ecosystem
 - Kafka has a rich ecosystem consisting of connectors and integration tools, making it easy to connect with other data systems and frameworks.
 - Examples include Kafka Connect and Kafka Schema Registry.

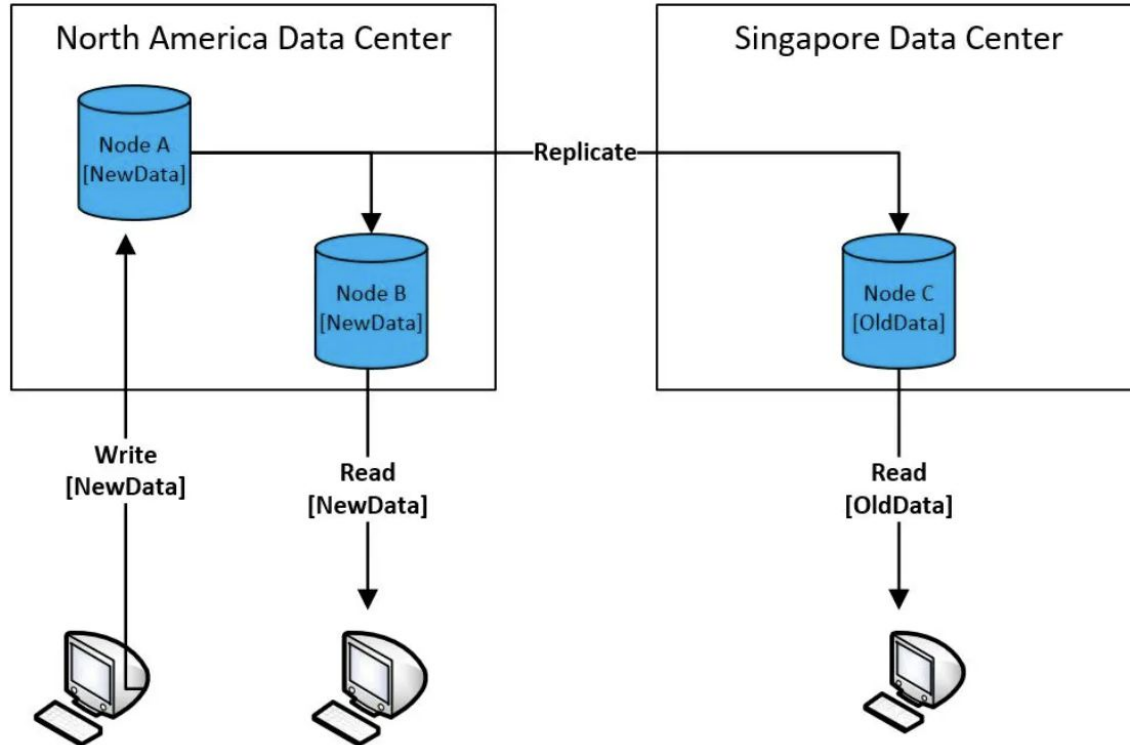
Comparison with Traditional Messaging Systems & Databases

- Kafka stores messages for the entire **retention period**
 - Ensures durability and fault tolerance even when consumers are offline
 - The default retention period is one week
- Kafka decouples message production and consumption,
 - Allows producers and consumers to operate independently at their **own** pace.
 - This design improves system stability.
- Kafka provides high-throughput and low-latency with a scale-out architecture
 - It guarantees message order within a single partition
 - While across multiple partitions, it is **eventually consistent** (but you can choose Strong consistency)
- It began to be used as an internal data bus within companies
 - Enabled by its high data throughput and support for multiple consumers.

Eventual Consistency? (1)

- If we write a record to a distributed system with 100 servers, can we read that record immediately?
 - Will the record we just wrote be returned?
 - Typically, a single data block is stored across multiple servers (**Replication Factor**).
 - So, when data is written or modified, it takes time to propagate across the system.
 - Since reads usually target one of the multiple data copies, the data may or may not be available, depending on the propagation time.
- Strong Consistency vs. Eventual Consistency
 - Generally, if the system waits until replication is complete before returning, this is Strong Consistency.
 - If it returns immediately without waiting for replication, this is Eventual Consistency.

Eventual Consistency? (2)

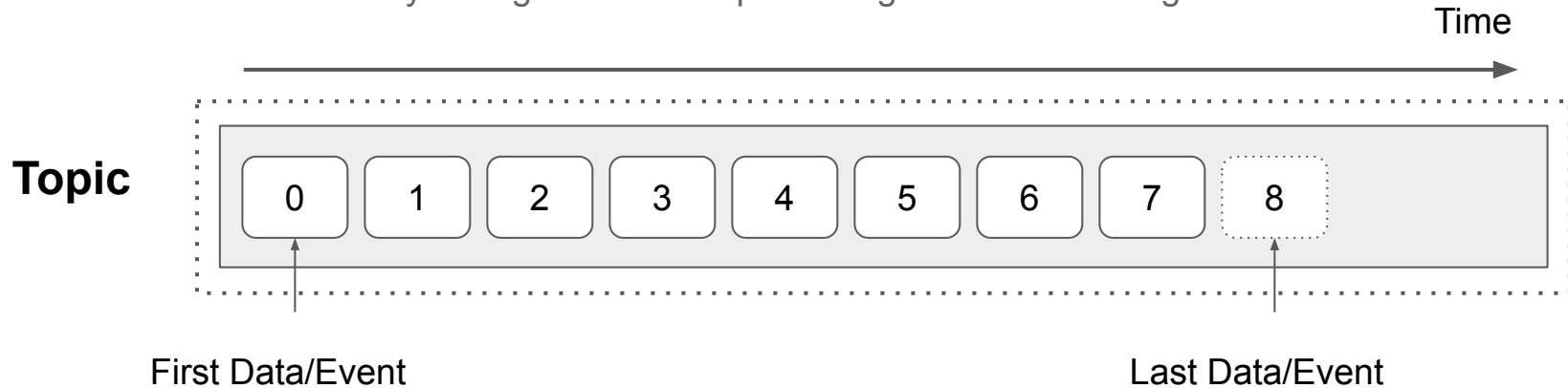


Break

Kafka Architecture

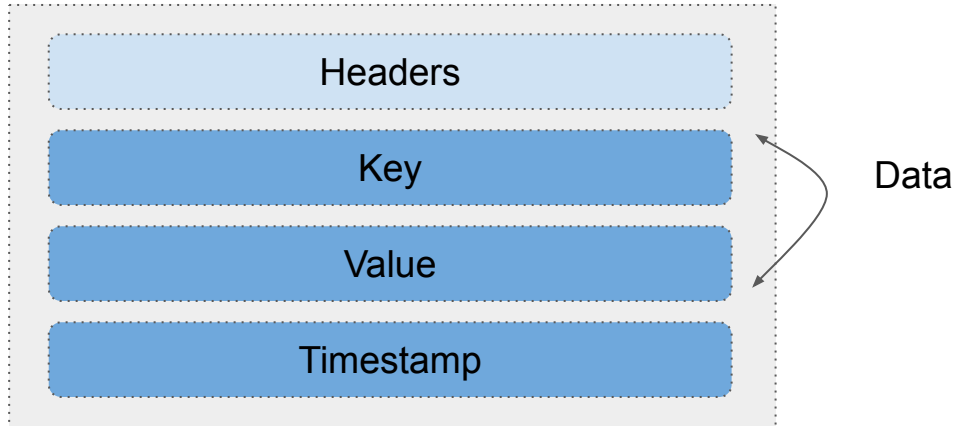
Data Event Stream

- It is called “**Topic**” controlled by producer (more on this later)
- The Producer creates a Topic, and the Consumer reads data from the Topic.
- Multiple Consumers can read from the same Topic
 - Each consumer maintains its own pointer
 - Data will stay during its retention period regardless of reading it or not



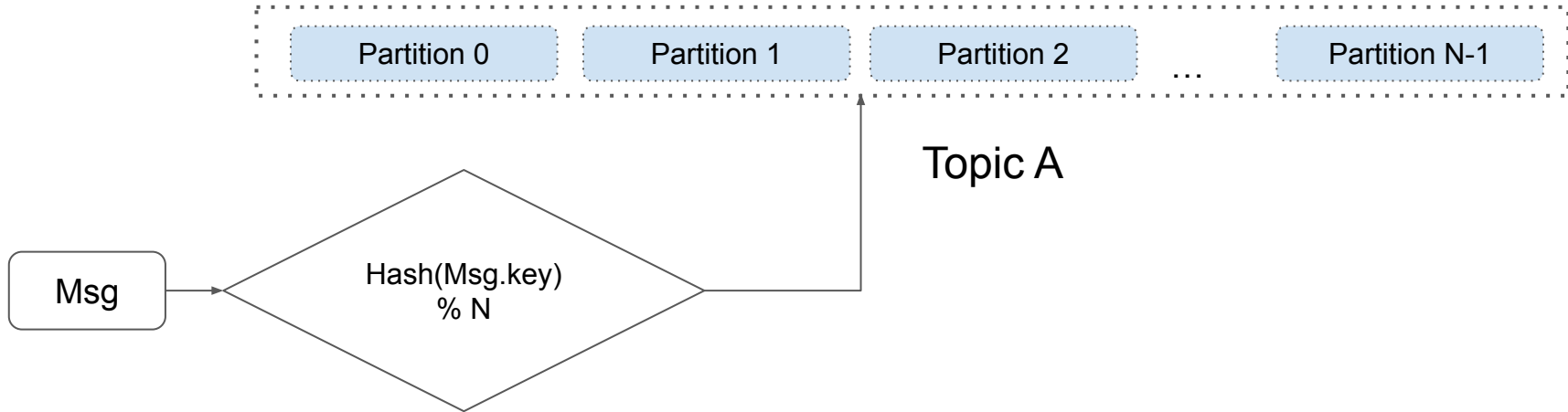
Message (Event) Structure: Key, Value, Timestamp

- 1MB at maximum
- Timestamp indicates when the data was added to Topic
- Key itself can have a complex structure
 - Key is used in partitioning
- Header is an optional lightweight meta data in key-value pairs



Topic & Partition

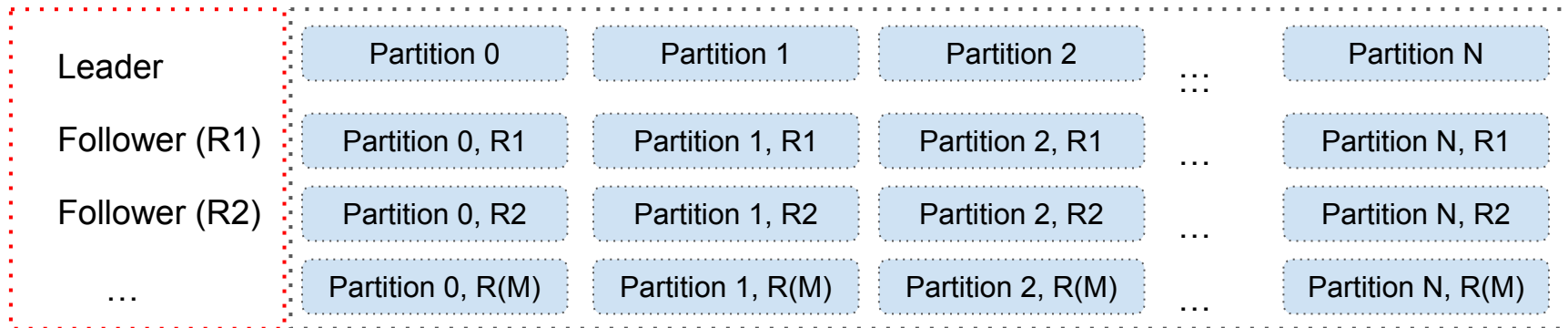
- A Topic is divided into multiple Partitions to enhance scalability/parallelism
- The method for determining which Partition a message belongs to depends on whether there is a key.
 - If a key exists, the partition is determined by taking the hash value of the key and calculating the remainder when divided by the number of partitions.
 - If there is no key, partitions are assigned in a round-robin fashion (not recommended).



Topic, Partition & Replica

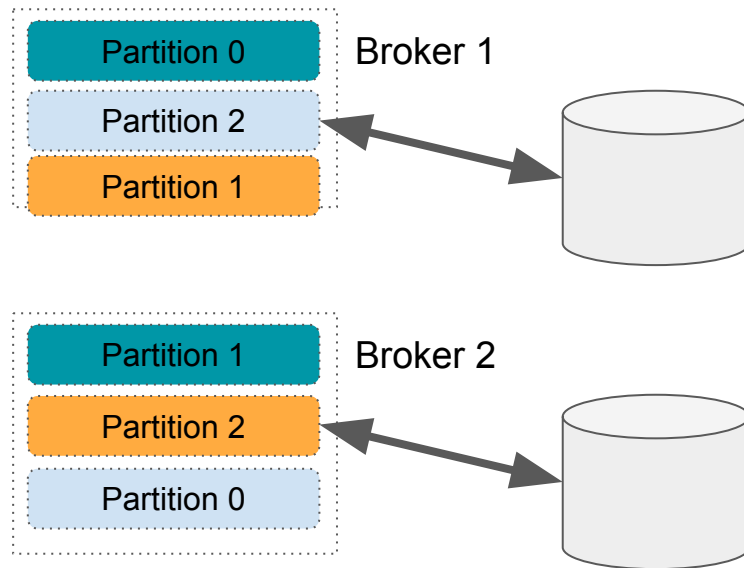
- Each Partition has a Replication Partition for fail-over.
- Each Partition has a Leader and Followers.
 - Writes are performed through the Leader, while reads can occur through either
 - The Consistency Level can be set per Partition (using in-sync replicas – "ack").

Topic A



Broker

- Kafka is composed of 1+ Broker
 - Very similar to HDFS
- Kafka Broker
 - Also known as Kafka Server or Kafka Node
 - Brokers store partitions
- Who manages Broker & Topic information
 - ZooKeeper



Producer & Consumer Basics

- Producer
 - Can be written in most programming languages:
 - Java, C/C++, Scala, Python, Go, .Net, REST API
 - A Command Line Producer utility also exists
- Consumer
 - Reads Messages based on the Topic (concept of Subscription exists)
 - Maintains position information of the last read Message using an Offset
 - A Command Line Consumer utility is available
 - Scaling is implemented through the concept of Consumer Groups
 - Solutions to address Backpressure issues
 - Consumers can also create new topics in Kafka
 - A common approach is for a single process to act as both Consumer and Producer

Kafka Installation








Kafka Installation: Docker Compose

- Use [this Github repo](#) and zk-single-kafka-single.yml file
 - full-stack.yml has more complete configuration but more resource intensive
- Run the following commands in your terminal

```
git clone https://github.com/conduktor/kafka-stack-docker-compose.git
```

```
cd kafka-stack-docker-compose
```

```
docker compose -f zk-single-kafka-single.yml up
```

Name	Image	Status	Port(s)
 kafka-stack-docker-compose		Running (2/2)	
 zoo1 f065d2185745 	confluentinc/cp-zookeeper:7.3.2	Running	2181:2181 
 kafka1 b4a7c9699704 	confluentinc/cp-kafka:7.3.2	Running	29092:29092  Show all ports (3)

Kafka Programming

Kafka Python Programming

- Python:
 - Confluent Kafka Python: Official Kafka Python Client Library developed by Confluent
 - **Kafka-Python**: Another Python Library. We will use this one since it is easy to install
- pip3 install kafka-python

```
keeyong kafka-stack-docker-compose % pip3 install kafka-python
DEPRECATION: Configuring installation scheme with distutils config files is deprecated and will no longer
work in the near future. If you are using a Homebrew or Linuxbrew Python, please see discussion at https://
/github.com/Homebrew/homebrew-core/issues/76621
Collecting kafka-python
  Downloading kafka_python-2.0.2-py2.py3-none-any.whl (246 kB)
    |████████████████████████████████████████| 246 kB 3.0 MB/s
Installing collected packages: kafka-python
DEPRECATION: Configuring installation scheme with distutils config files is deprecated and will no longer
work in the near future. If you are using a Homebrew or Linuxbrew Python, please see discussion at https://
/github.com/Homebrew/homebrew-core/issues/76621
Successfully installed kafka-python-2.0.2
WARNING: You are using pip version 21.3.1; however, version 23.1.2 is available.
You should consider upgrading via the '/usr/local/opt/python@3.9/bin/python3.9 -m pip install --upgrade pi
p' command.
```

Simple Producer

```
from time import sleep
from json import dumps
from kafka import KafkaProducer
```

```
producer = KafkaProducer(
    bootstrap_servers=['localhost:9092'],
    value_serializer=lambda x: dumps(x).encode('utf-8')
)
```

```
for j in range(999):
    print("Iteration", j)
    data = {'counter': j}
    producer.send('topic_test', value=data)
    sleep(0.5)
```

Specify all or one of the Brokers

- Create a `KafkaProducer` object that connects to a local Kafka instance.
- Define a method to convert the data to be sent into a JSON string and then serialize it by encoding it in UTF-8.
- Remember this lambda function syntax

- Send an event every 0.5 seconds containing the topic name "topic_test" and a repeating counter as data.
- Configure the data to have a key called "counter" with an integer value.

Format of lambda function: lambda arguments: expression
Ex) lambda x: dumps(x).encode('utf-8')

Key & headers aren't specified

Execute the Producer

- If Kafka isn't running, "NoBrokersAvailable" error message will appear

```
keeyong kafka-stack-docker-compose % python3 producer.py
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
```

Check the Topic creation

- Log in to the Kafka docker container and run commands

```
docker exec -it kafka1 sh
```

```
$ kafka-topics --list --bootstrap-server localhost:9092
```

```
$ kafka-topics --describe --topic topic_test --bootstrap-server localhost:9092
```

```
sh-4.4$ kafka-topics --list --bootstrap-server localhost:9092
topic_test
sh-4.4$ kafka-topics --describe --topic topic_test --bootstrap-server localhost:9092
Topic: topic_test      TopicId: PbvP1gYYSquTkqMtcGfF6w PartitionCount: 1      Replic
ationFactor: 1  Configs:
      Topic: topic_test      Partition: 0      Leader: 1      Replicas: 1      Isr: 1
```

Instantiate a Consumer object

```
consumer = KafkaConsumer(
```

```
    'topic_test',
```

```
    bootstrap_servers=['localhost:9092'],
```

```
    auto_offset_reset='earliest',
```

```
    enable_auto_commit=True,
```

```
    group_id='my-group-id',
```

```
    value_deserializer=lambda x: loads(x.decode('utf-8'))
```

```
)
```

'earliest' vs. 'latest'

In case of `enable_auto_commit=False`, you have to explicitly set the offset value with commit function

Perform the reverse operation of the `value_serializer` used in the Producer

Simple Consumer

```
from kafka import KafkaConsumer
from json import loads
from time import sleep
```

```
consumer = KafkaConsumer(
    'topic_test',
    bootstrap_servers=['localhost:9092'],
    auto_offset_reset='earliest',
    enable_auto_commit=True,
    group_id='my-group-id',
    value_deserializer=lambda x: loads(x.decode('utf-8'))
)
```

```
for event in consumer:
    event_data = event.value
    # Do whatever you want
    print(event_data)
    sleep(2)
```

- Create a `KafkaConsumer` object that connects to a local Kafka instance.
- Configure it to read data from the "topic_test" topic, starting from the earliest data, continuously updating the offset information, and joining a consumer group named "my-group-id".

- Configure it to read the counter value from the "topic_test" topic every 2 seconds.

Execute the Consumer

- If Kafka isn't running, "NoBrokersAvailable" error message will appear

```
keyong kafka % python3 consumer.py
{'counter': 0}
{'counter': 1}
{'counter': 2}
{'counter': 3}
{'counter': 4}
{'counter': 5}
{'counter': 6}
{'counter': 7}
{'counter': 8}
```


Kafka Python Producer/Consumer Demo

- Let's run the previous example
- 999 might be too big for the demo so we will change it to 50

Demo & ~~Homework~~

Demo: Streaming in Snowflake

- <https://docs.snowflake.com/en/user-guide/streams-examples>

Group Project Presentation

Proposal changes need to be submitted by EOW
Presentation order will be randomly determined
10 minutes per group

Next Week

Going over the end to end system

Final Exam (SQL)

Guest Speaker (5 to 5:45PM)