Kafka

#Intro

- **What is Kafka?**
  - Kafka is a tool for handling large amounts of data in real-time.
  - It’s like a messaging system that helps send and receive data quickly and in an organized way.
  - It works by sending data in "topics," which are like channels or categories, so different parts of a system can read and write data without getting in each other’s way.
- **What is it used for?**
  - **Real-time data processing**: Kafka is used to send and receive real-time data between different applications. For example, tracking live user activity or sensor data.
  - **Data integration**: It helps different parts of a company’s systems to talk to each other without needing to be directly connected.
  - **Storing logs**: Kafka can be used to store logs of systems or servers in a way that’s easy to search through later.
- **Who developed Kafka?**
  - Kafka was developed by **LinkedIn**.
  - It was created by **Jay Kreps, Neha Narkhede, and Jun Rao**.
  - Later, Kafka was open-sourced and is now maintained by **Apache Software Foundation**.

Consider Below Scenario :-

HOW ZOMATO GIVES LIVE LOCATION OF THE DRIVER REACHING TO THE CUSTOMER?????

(i) After Every second the location of driver is fetched and sent to the zomato server where the location information(Current time and location) is stored inside the DB. The customer then gets information.

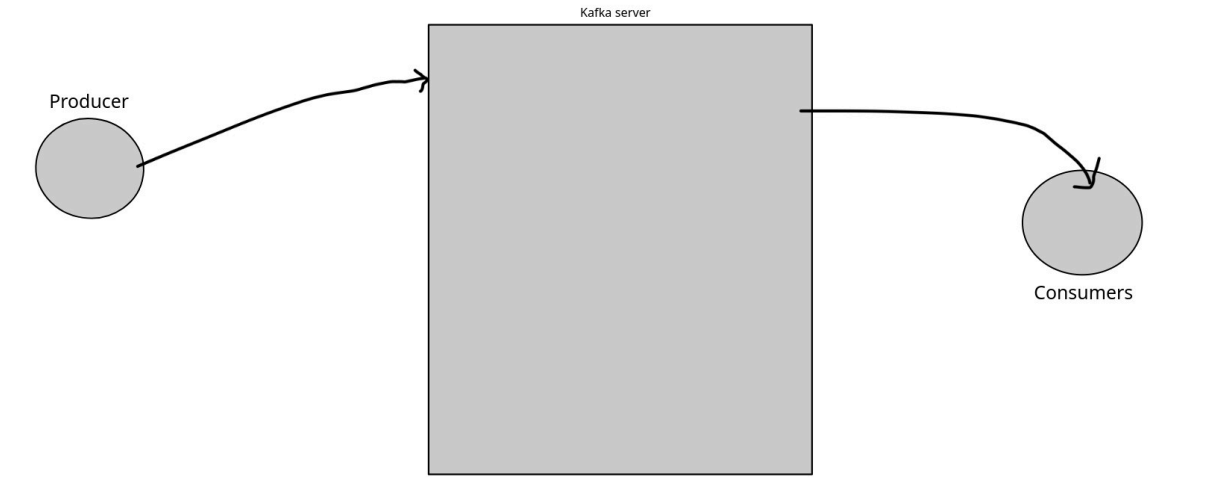
The problem is the inserts which are happening inside the DB every seconds is very large as there are many drivers. As , the operations per second is high and DB has low throughput hence after some time database goes down. Also insertion inside database takes some time , hence overall process becomes time consuming.

Solution

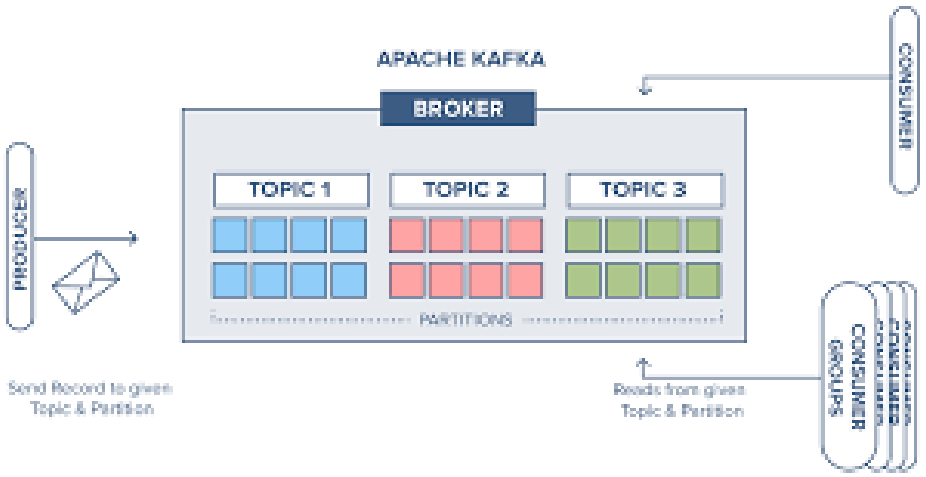
- The Above problem is solved using Kafka Because Kafka has high Throughput.
- But Kafka can’t store large data just like a database ALSO we can’t query data in kafka.
- So we use combination of Both the database and Kafka to Solve the above problem.

#Architecture of Kafka

1. So , Producer will Produce the data and Send it to Kafka Server and then from Kafka server the data is consumed by different Consumers.



2. To further Improve the Efficiency of Kafka , Kafka is divided into Different Topics , Topics help manage data flow by putting it into specific categories, like folders. This makes it easier to send and receive the right data to the right places. For Example By using different topics for "Orders," "Payments," and "Shipping," the system keeps the data organized and makes it easier for different parts of the business to handle each type of data separately.



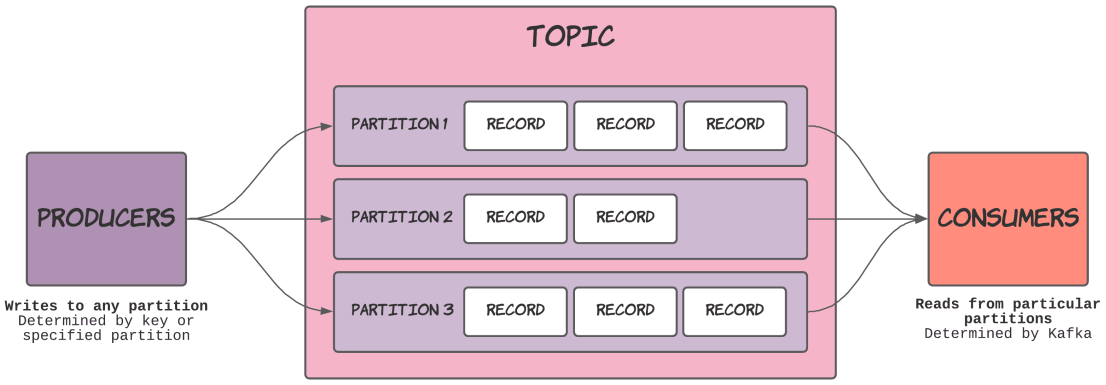
3. The topics are further Divided into Partitions , **partitions** are divisions of a **topic** that allow Kafka to scale and distribute data across multiple brokers (servers). Each partition is an ordered, immutable sequence of messages, and each message within a partition has a unique ID called an **offset**.

Example:

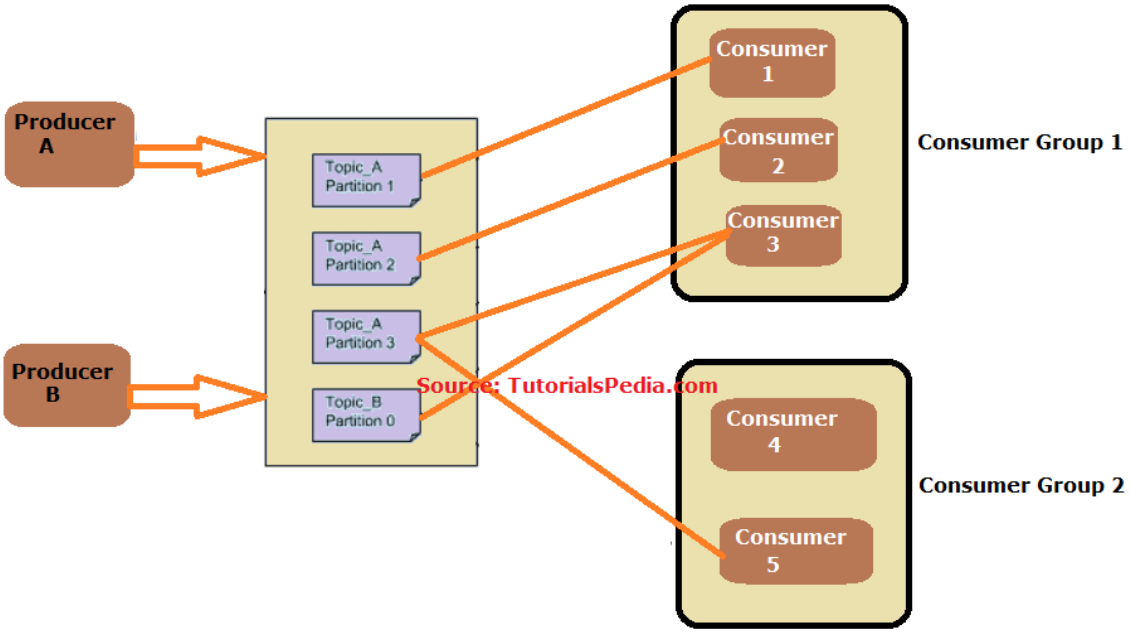
Imagine you have a Kafka topic called `orders` where each order is a message. If the `orders` topic is divided into 3 partitions, you could have:

- **Partition 1**: Contains orders from 1 to 1000.
- **Partition 2**: Contains orders from 1001 to 2000.
- **Partition 3**: Contains orders from 2001 to 3000.

If 3 consumers are reading from the topic, each can independently read from one of the partitions at the same time, allowing faster processing of messages.



4. In Kafka, **consumer groups** are a way to organize multiple consumers that work together to consume messages from a topic. Each consumer in a group reads from a subset of the partitions of the topic, ensuring that the work is divided among them.



Key Points:

- A **consumer group** is a group of consumers that share the same group ID.
- Each **partition** of a topic is assigned to only **one consumer** within a group at a time (but one partition can be read by different consumers if they're in different groups).
- This allows multiple consumers to work in parallel without overlapping on the same partition.

Partition Assignment in a Consumer Group:

- Kafka will **assign each partition to only one consumer** in a consumer group. This ensures that each consumer processes a different part of the data, and no consumer processes the same partition at the same time.
- If you have more consumers than partitions, some consumers will be **idle**.
- If you have more partitions than consumers, some consumers will have to process more than one partition.

Example:

Imagine a topic called **"orders"** with 4 partitions:

- **Partition 1, Partition 2, Partition 3, Partition 4**

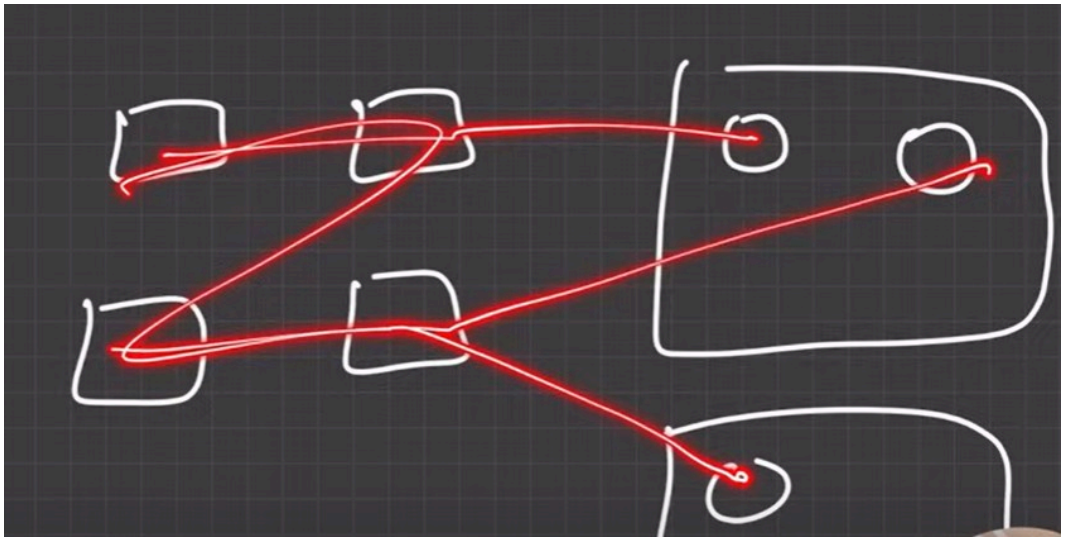
Consumer Group 1:

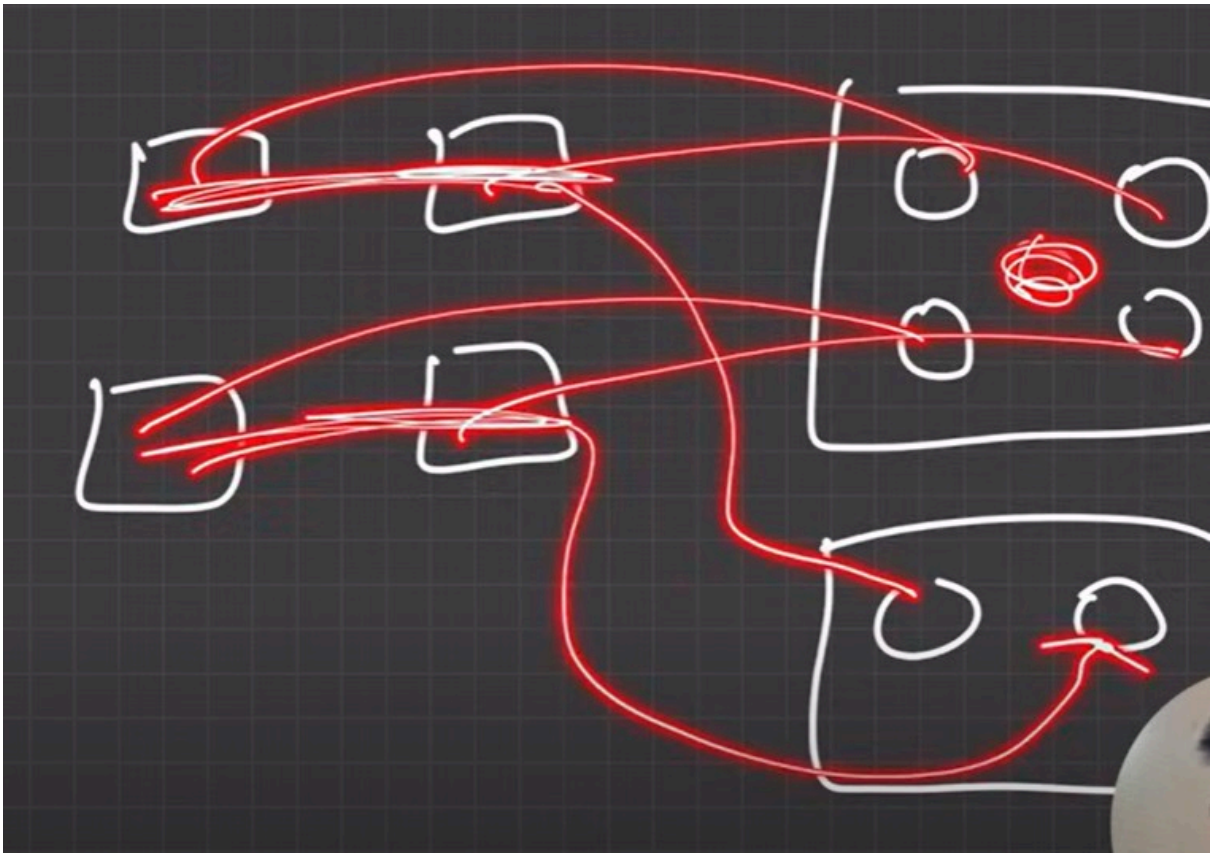
- Consumer 1 reads from Partition 1.
- Consumer 2 reads from Partition 2.
- Consumer 3 reads from Partition 3.
- Consumer 4 reads from Partition 4.

Consumer Group 2 (a different group):

- Consumer 1 reads from Partition 1.
- Consumer 2 reads from Partition 2.
- Consumer 3 reads from Partition 3.
- Consumer 4 reads from Partition 4.

- 1 CONSUMER CAN TAKE MULTIPLE PARTITIONS BUT 1 PARTITION CAN BE CONSUMED BY ONLY ONE CONSUMER





In Kafka, **queue** and **publish/subscribe (pub/sub)** are two messaging patterns that determine how messages are consumed. Here's their importance in simple terms:

1. Queue (Point-to-Point) Model:

In this model, each message sent to a Kafka **topic** is consumed by **only one consumer** in a **consumer group**. Once a message is consumed, it is not available for other consumers in the same group.

- **Importance**: This is useful when you want to **distribute work** (e.g., task processing) to multiple consumers, where each consumer processes only a part of the work.

Example:

If you have a topic called "orders" and three consumers in a group:

- Each consumer gets a unique message from the topic.
- Once a consumer processes the message, it’s gone for the rest.

2. Publish/Subscribe (Pub/Sub) Model:

In this model, **multiple consumers** can receive the same message from the topic. Each consumer or consumer group receives a copy of the message.

- **Importance**: This is ideal when you want to **broadcast** messages to multiple consumers, like sending the same notification to many users.

Example:

If you have a topic called "notifications" and two consumer groups:

- One group (Group A) might consume the message and send it to its subscribers.
- The other group (Group B) consumes the same message and sends it to a different set of subscribers.

Summary:

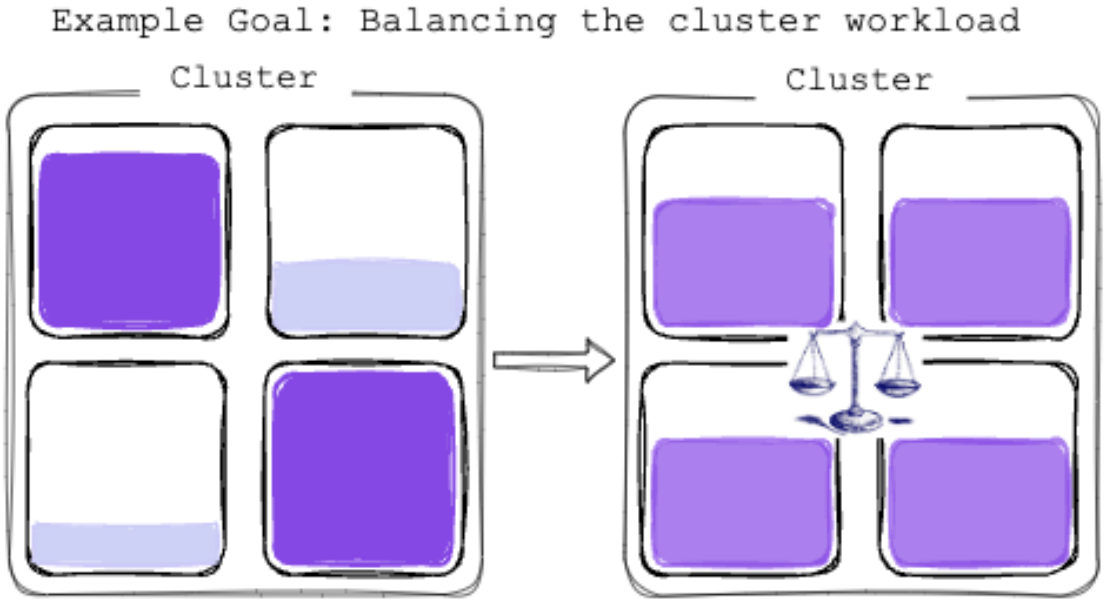
- **Queue**: One message is consumed by one consumer (for distributing tasks).
- **Pub/Sub**: One message can be consumed by many consumers (for broadcasting messages).

---

1. Self-Balancing in Kafka:

Self-balancing in Kafka refers to the automatic redistribution of partitions among consumers in a **consumer group** when the number of consumers changes (e.g., adding or removing consumers). This ensures that the workload is evenly spread across available consumers without manual intervention.

- **Why it's useful**: It helps **optimize performance** and **resource usage**. If a consumer is added or removed, Kafka will reassign partitions to ensure all consumers are actively working and no partitions are left unprocessed.



Example:

Imagine a topic with 4 partitions and 2 consumers:

- Initially, Consumer 1 gets Partition 1 and Partition 2.
- Consumer 2 gets Partition 3 and Partition 4.

Now, if a **third consumer** is added, Kafka will automatically rebalance the partitions so that each consumer gets a fair share of the work. For example:

- Consumer 1: Partition 1
- Consumer 2: Partition 2
- Consumer 3: Partition 3 and Partition 4

This self-balancing makes sure that the system adapts to changes in the number of consumers efficiently.

---

2. Use of Zookeeper in Kafka:

Zookeeper is a distributed coordination service that Kafka uses for managing and maintaining the state of its clusters. It helps in tasks like:

- **Leader election**: Zookeeper helps elect a leader for partitions. This ensures only one consumer processes a partition at a time.

- **Metadata storage**: It stores information about Kafka brokers, topics, and partitions.
- **Fault tolerance**: It helps Kafka handle failures by keeping track of which brokers are alive and which partitions need to be reassigned.

Example:

When Kafka starts, Zookeeper keeps track of the brokers, which ones are active, and which partitions are assigned to which brokers. If a broker fails, Zookeeper will notify Kafka and help it reassign the partitions to healthy brokers, ensuring minimal disruption.

In Simple Terms:

- **Self-Balancing**: Kafka automatically adjusts and redistributes the workload among consumers when there are changes in the consumer group.
- **Zookeeper**: A tool Kafka uses to manage its cluster's state, coordinate tasks like leader elections, and ensure fault tolerance.