WA3285 Introduction to Kafka
for PGR

The following terms are trademarks of other companies:

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

IBM, WebSphere, DB2 and Tivoli are trademarks of the International Business Machines Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Table of Contents

Chapter 1 - Introduction to Kafka

Objectives

Key objectives of this chapter

- Messaging Architectures

- What is Kafka?

- Need for Kafka

- Where is Kafka useful?

- Architecture

- Core concepts in Kafka

- Overview of Kraft

- Cluster, Kafka Brokers, Producer, Consumer, Topic

1.1 Messaging Architectures – What is Messaging?

- Application-to-application communication

- Supports asynchronous operations.

- Message:
  - ◇ A message is a self-contained package of business data and network routing headers.

❑Broker:
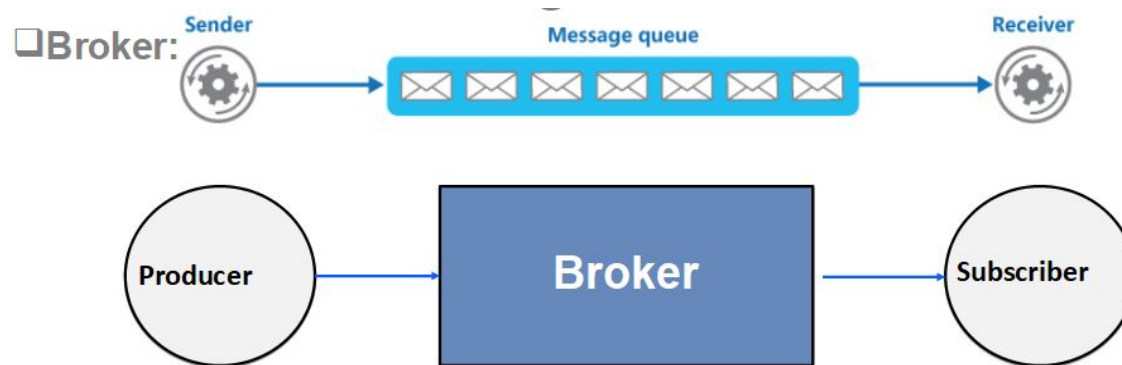
1.2 Messaging Architectures – Steps to Messaging

- Messaging connects multiple applications in an exchange of data.

- Messaging uses an encapsulated asynchronous approach to exchange data through a network.

- A traditional messaging system has two models of abstraction:

  ◊ Queue – a message channel where a single message is received exactly by one consumer in a point-to-point message-queue pattern. If there are no consumers available, the message is retained until a consumer processes the message.

  ◊ Topic - a message feed that implements the publish-subscribe pattern and broadcasts messages to consumers that subscribe to that topic.

- A single message is transmitted in five steps:

  ◊ Create

  ◊ Send

  ◊ Deliver

◇ Receive

◇ Process

1.3 Messaging Architectures – Messaging Models

- 1. Point to Point

- 2. Publish and Subscribe

Point-to-point (1→1)

| Sender | ┈┈┈▶ | Queue | ┈┈┈▶ | Potential receiver |

Publish-and-subscribe (1→Many)

| Publisher | ┈┈┈▶ | Topic | ┈┈┈▶ | Subscriber |
| | | | ┈┈┈▶ | Subscriber |

1.4 What is Kafka?

- In modern applications, real-time information is continuously generated by:

    ◇ Applications (publishers/producers)

    ◇ Routed to other applications (subscribers/consumers)

- Apache Kafka is an open source, distributed publish-subscribe messaging system.

    ◇ Written in the Scala language with multi-language support and runs on the Java Virtual Machine (JVM).

- Kafka allows integration of information of producers and consumers to avoid any kind of rewriting of an application at either end.

- Kafka overcomes the challenges of real-time data usage for consumption of data volumes that may grow in order of magnitude, larger than the real data.

- Kafka also supports parallel data loading in Hadoop systems.

1.5 Kafka Overview

- When used in the right way and for the right use case, Kafka has unique attributes that make it a highly attractive option for data integration.

Data Sources
(Producers) → Data Integration → Data Consumers
(Subscribers)
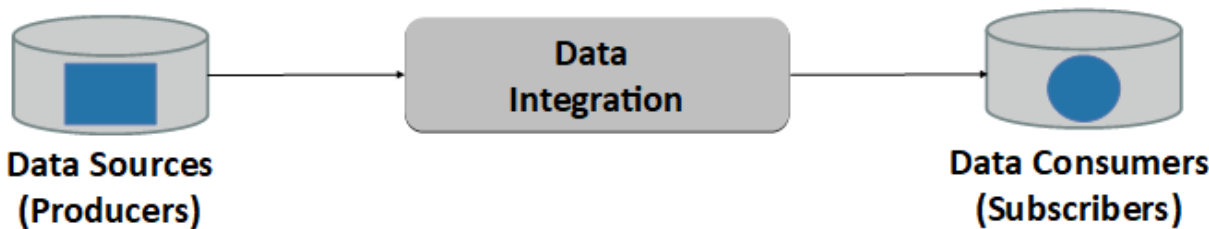
- Data Integration is the combination of technical and business processes used to combine data from disparate sources into meaningful and valuable information.

  ◇ A complete data integration solution encompasses discovery, cleansing, monitoring, transforming and delivery of data from a variety of sources

- Messaging is a key data integration strategy employed in many distributed environments such as the cloud.

  ◇ Messaging supports asynchronous operations, enabling you to decouple a process that consumes a service from the process that implements the service.

1.6 Kafka Overview (Contd.)
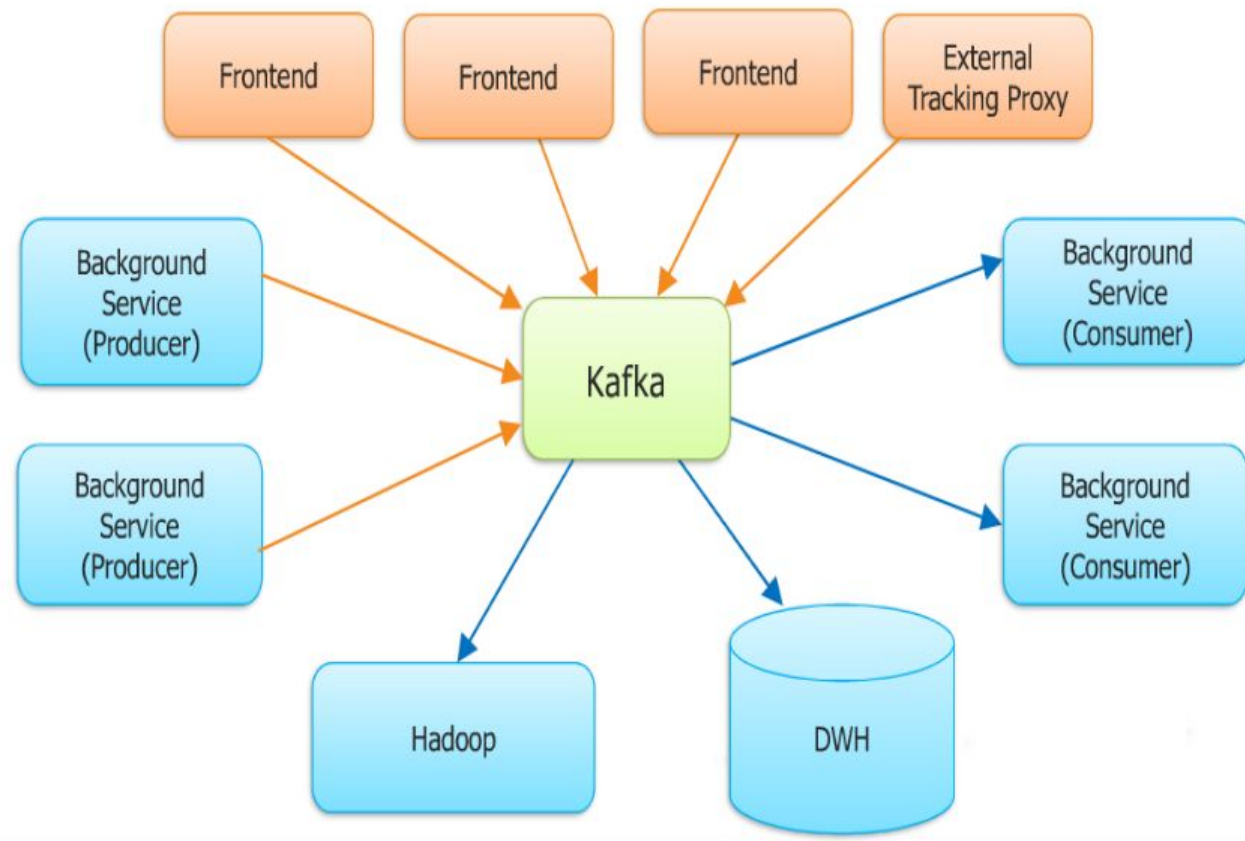
1.7 Need for Kafka

- High Throughput
  - ◊ Provides support for hundreds of thousands of messages with modest hardware

- Scalability
  - ◊ Highly scalable distributed systems with no downtime

- Replication
  - ◊ Messages can be replicated across a cluster, which provides support for multiple subscribers and also in case of failure balances the consumers

- Durability
  - ◊ Provides support for persistence of messages to disk which can be further used for batch consumption

- Stream Processing
  - ◊ Kafka can be used along with real-time streaming applications like Spark, Flink, and Storm

- Data Loss
  - ◊ Kafka with proper configurations can ensure zero data loss - idempotency

1.8 When to Use Kafka?

- Kafka is highly useful when you need the following:
  - A highly distributed messaging system
  - A messaging system which can scale out exponentially
  - high throughput on publishing and subscribing
  - varied consumers having varied capabilities by which to subscribe these published messages in the topics
  - A fault tolerance operation
  - Durability in message delivery
  - All of the above without tolerating performance degrade

1.9 Kafka Architecture
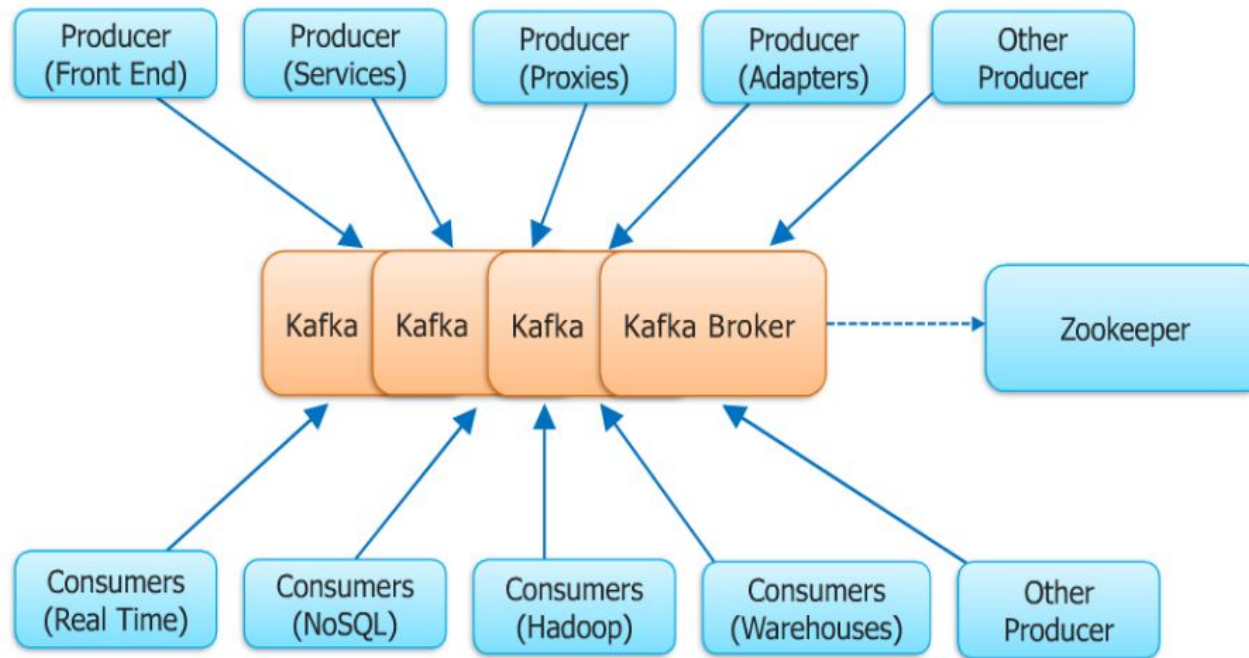
1.10 Core concepts in Kafka

- **Topic**

  ◇ A category or feed to which messages are published

- **Producer**

  ◇ Publishes messages to Kafka Topic

- **Consumer**

  ◇ Subscribes and consumes messages from Kafka Topic

- **Broker**

  ◇ Handles hundreds of megabytes of reads and writes

1.11 Kafka Topic

- User defined category where the messages are published

- For each topic, a partition log is maintained

- Each partition basically contains an ordered, immutable sequences of messages where each message assigned a sequential ID number called *offset*

- Writes to a partition are generally sequential thereby reducing the number of hard disk seeks

- Reading messages from partition can either be from the beginning and also can rewind or skip to any point in a partition by supplying an offset value

Anatomy of a Topic

Partition 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12

Partition 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Partition 2 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12

Writes

Old ──────────────→ New
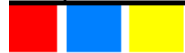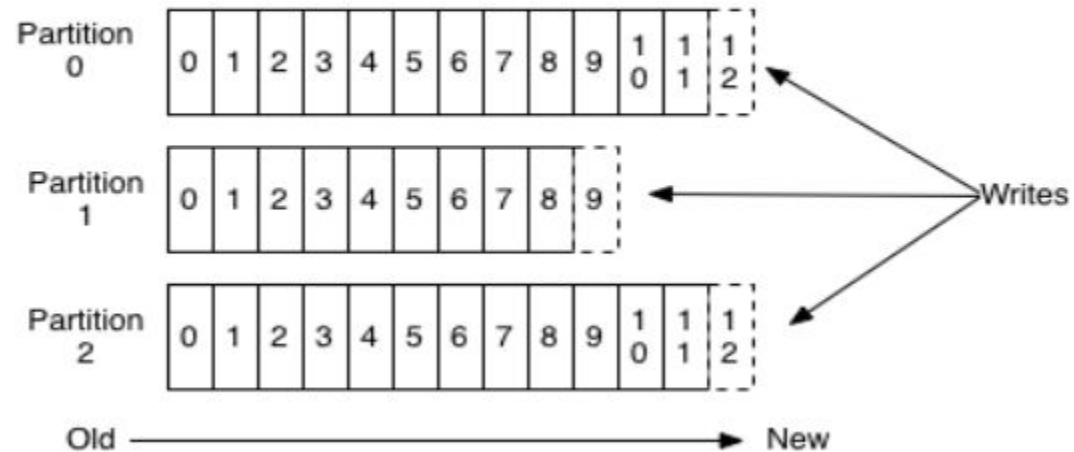
1.12 Kafka Partitions

- In Kafka, a topic can be assigned to a group of messages of a similar type that will be consumed by similar consumers.

- Partitions (a unit of parallelism in Kafka), parallelize the consumption of messages in a Kafka topic with the total number in a cluster not less than the number of consumers in a consumer group.
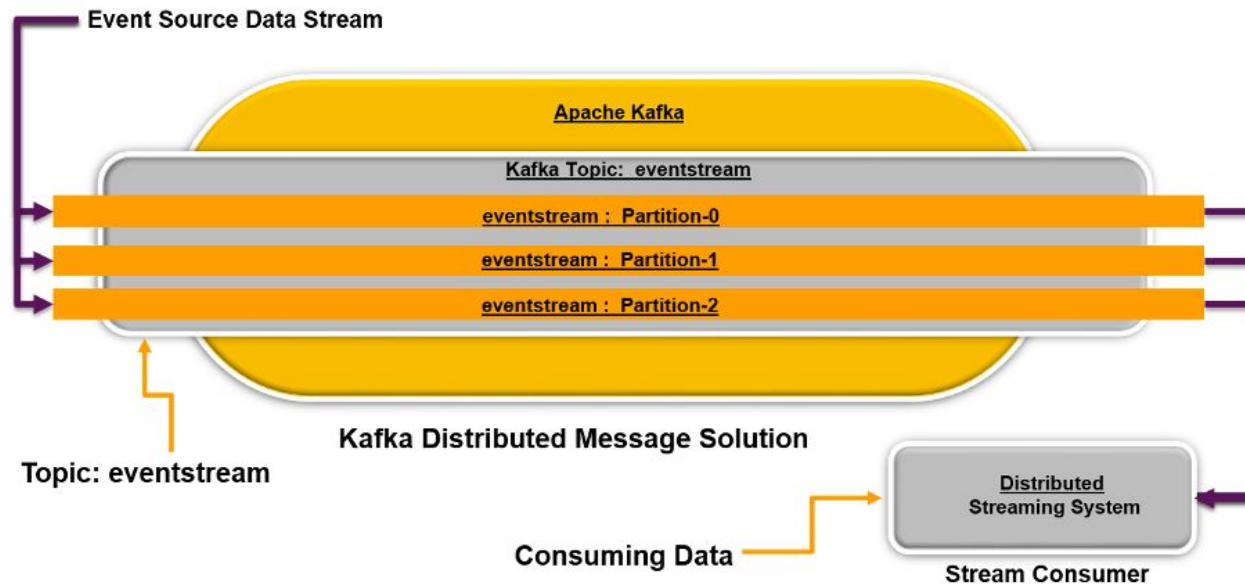
1.13 Kafka Producer

- Application publishes messages to the topic in Kafka Cluster

    ◇ Can be of any kind like an API Front End, Streaming etc.

- While writing messages, it is also possible to attach a key to the message

    ◇ By attaching key the producers basically provide a guarantee that all messages with the same key will arrive in the same partition

- Supports both async and sync modes

- Publishes as many messages as fast as the broker in a cluster can handle
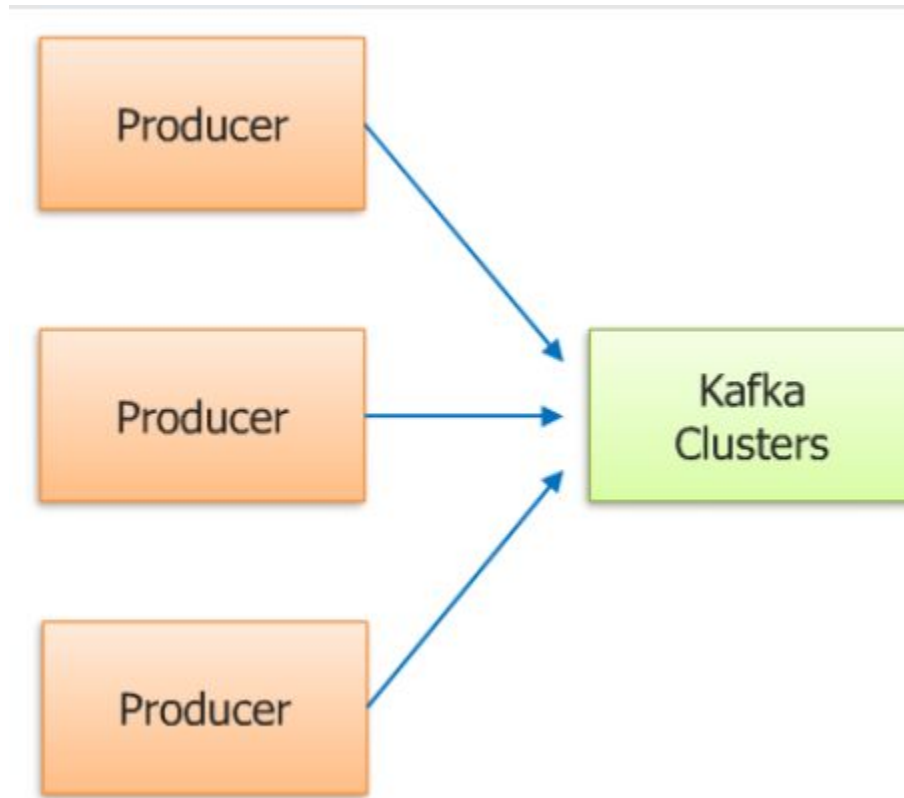
1.14 Kafka Consumer

- Applications subscribe and consume messages from brokers in Kafka Cluster

    ◇ Can be of any kind like real-time consumers, NoSQL consumers etc.

- During consumption of messages from a topic a consumer group can be configured with multiple consumers.

    ◇ Each consumer of consumer group reads messages from a unique subset of partitions in each topic they subscribe to

    ◇ Messages with the same key arrive at the same consumer

- Supports both Queuing and Publish-Subscribe

- Consumers have to maintain the number of messages consumed
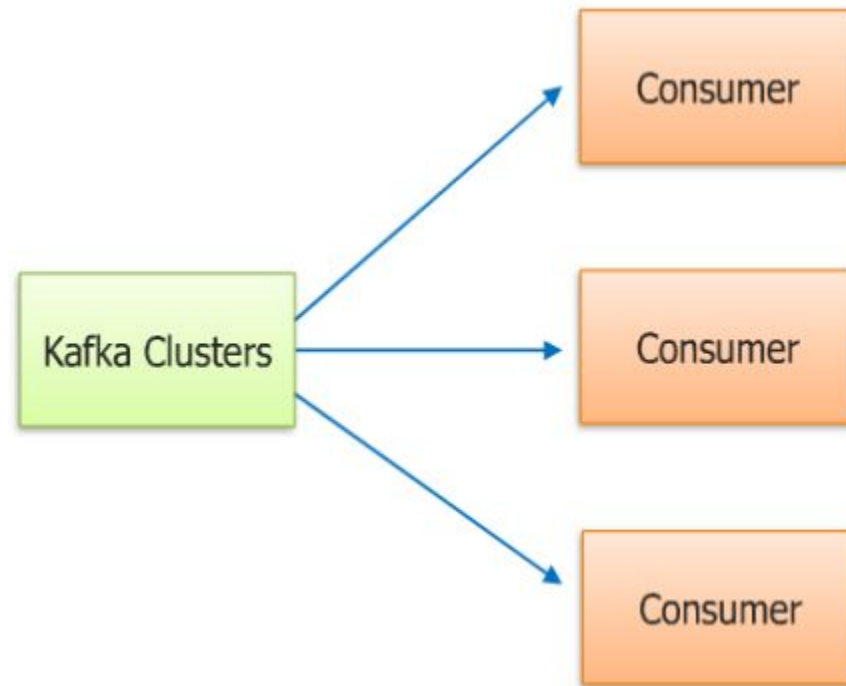
1.15 Kafka Broker

- Kafka cluster basically is comprised of one or more servers

  ◇ Each of the servers in the cluster is called a **broker**

- Handles hundreds of megabytes of writes from producers and reads from consumers

- Retains all the published messages irrespective of whether it is consumed or not

- If retention is configured for n days, published message is available for consumption for configured n days and thereafter it is discarded
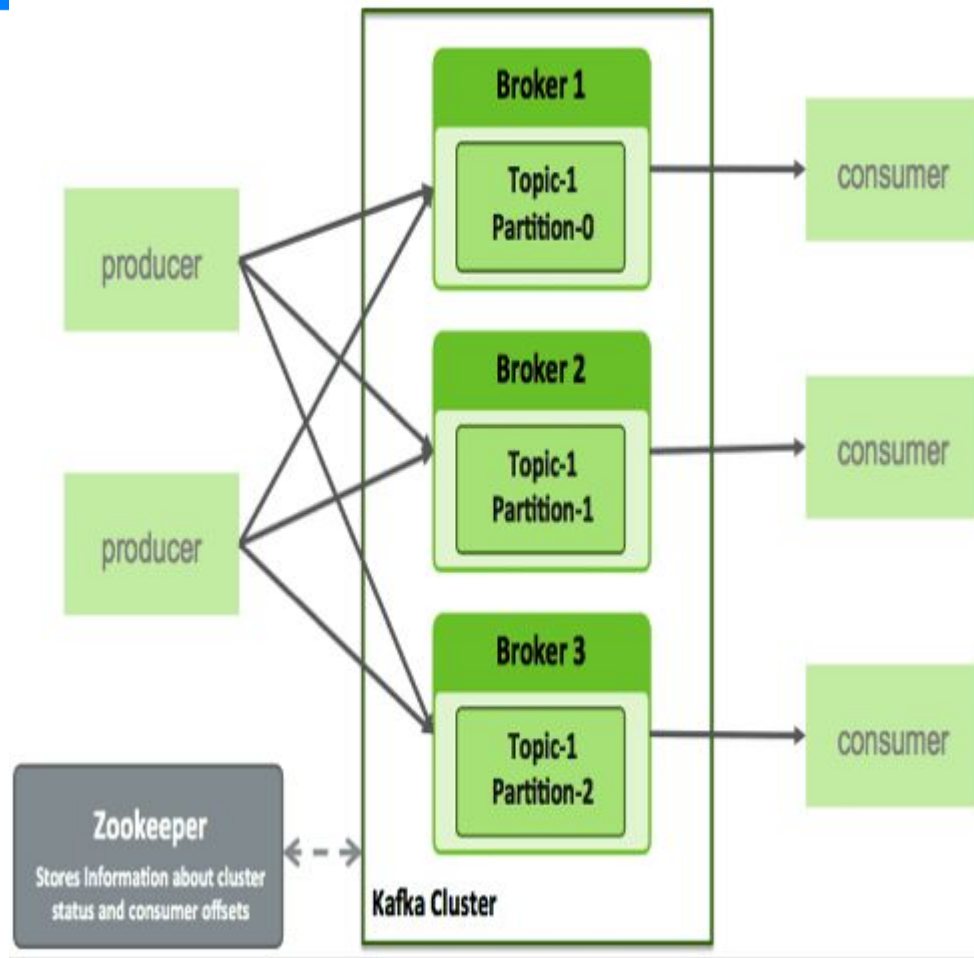
1.16 Kafka Cluster

- A Kafka Cluster is generally fast, highly scalable messaging system

- A publish-subscribe messaging system

- Can be used effectively in place of ActiveMQ, RabbitMQ, Java Messaging System (JMS), and Advanced Messaging Queuing Protocol (AMQP)

- Can be integrated with the Hadoop Ecosystem

- Expanding of the cluster can be done with ease

- Effective for applications which involve large-scale message processing

1.17 Why Kafka Cluster?

- Kafka is preferred in place of more traditional brokers like JMS and AMQP

  - With Kafka, we can easily handle hundreds of thousands of messages in a second, which makes Kafka a high throughput messaging system

  - The cluster can be expanded with no downtime, making Kafka highly scalable

  - Messages are replicated, which provides reliability and durability

  - Fault-tolerant

1.18 Sample Multi-Broker Cluster

SINGLE NODE MULTI BROKER CLUSTER

Consumer Group 1
- This group contains two consumers and each consumer will be listening to one partition in the topic
- Acts as a queuing system

KAFKA CLUSTER

NODE 1

Producer
- Source system from where messages will be published

BROKER 1

Input Topic

Partition 0

Partition 1

PRODUCER (Source)

Consumer Group 1

Consumer 1

Consumer 2

BROKER 2

Input Topic

Partition 0

Partition 1

Consumer Group 2

Consumer 1

Consumer Group 2
- This group contains only one consumer and that consumer will be listening to both the partition in the topic
- Acts as a publish-subscribe messaging system

1.19

1.20 Kraft

◇ Apache Kafka® Raft (KRaft) is the consensus protocol that was introduced to remove Kafka's dependency on ZooKeeper for metadata management.

◇ This simplifies the architecture and deployment dependencies.

1.21 Schema Registry

- Schema Registry is required for the following reasons:

  ◇ Ensure data quality and evolvability

  ◇ Define data standards

  ◇ Evolve without breaking applications

- Provides centralized schema management and the RESTful interface for storing and retrieving schemas for:

  ◇ Avro

  ◇ JSON

  ◇ Protobuf

- The Schema Registry provides a mapping between topics in Kafka and the schema they use.

- Helps with data governance and architects a loosely coupled and dynamically evolving systems

1.22 Schema Registry (contd.)

- It also enforces compatibility rules before messages are added.

- The Schema Registry will check every message sent to Kafka for compatibility

  ◇ Ensuring that incompatible messages will fail on publication.

- The Confluent Schema Registry provides both runtime validation of schema compatibility, as well as a caching feature for schemas so they don't need to be included in the message payload.

Documentation site:
https://docs.confluent.io/platform/current/schema-registry/index.html#schemaregistry-intro

Schema Registry tutorial:
https://docs.confluent.io/platform/current/schema-registry/schema_registry_tutorial.html#schema-registry-tutorial

1.23 Who Uses Kafka?

Linked in	**NETFLIX**	**Spotify**
Apache Kafka is used at LinkedIn for activity stream data and operational metrics. This powers various products like LinkedIn Newsfeed, LinkedIn Today in addition to our offline analytics systems like Hadoop.	Real-time monitoring and event-processing pipeline.	Kafka is used at Spotify as part of their log delivery system.

As part of their Storm stream processing infrastructure, e.g. this and this.

FOURSQUARE

Kafka powers online to online messaging, and online to offline messaging at Foursquare. We integrate with monitoring, production systems, and our offline infrastructure, including hadoop.

airbnb

Used in our event pipeline, exception tracking & more to come.

1.24 Summary

- Kafka is a unique distributed publish-subscribe messaging system written in the Scala language with multi-language support and runs on the Java Virtual Machine (JVM).

- Kafka relies on another service named Zookeeper – a distributed coordination system – to function.

- Kafka has high-throughput and is built to scale-out in a distributed model on multiple servers.

- Kafka persists messages on disk and can be used for batched consumption as well as real-time applications.

1.25 Review Questions

⋄ What two messaging architectures does Kafka utilize?

⋄ What are the four main concepts of Kafka?

⋄ What factors are important considerations when using Kafka partitions?

⋄ What is idempotency? How is it achieved with Kafka?

Chapter 2 - The Inner Workings of Apache Kafka

Objectives

In this chapter, participants will learn about:

- Apache Kafka's internals
    - Topics and partitions
    - Brokers
    - Consumer groups
    - Log compaction

2.1 A Kafka Cluster High-Level Interaction Diagram

Source: http://bit.ly/2AHuwFw
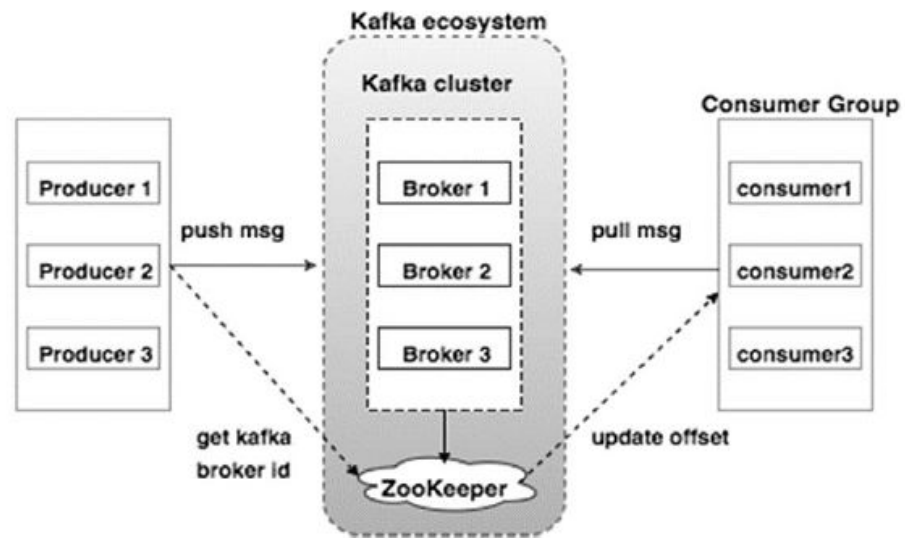
2.2 Topics & Partitions

- A Kafka topic is a mechanism for message classification and is often referred to as a distributed commit log; it can also be viewed as

  - ◇ Materialized event view into streamed events at rest

- Topics are made up of partitions, which are ordered "commit logs" (physical files) where events (messages/records) are stored

  - ◇ By default, on Linux, partitions are stored in the */tmp/kafka-logs/* directory. Partition files are named <topic>-<partion_id>, e.g. orders-0, orders-1, etc. Operational topic partitions are being served by the (partition) leader broker

- For fault tolerance, each partition can be replicated to one or more brokers in the cluster

  - ◇ The replication factor (the *replication.factor* configuration parameter) is configured at the topic's creation time; the recommended value in production is 3 replicas

2.3 The Terms Event/Message/Record

- The terms *event/message/record* are, for the most part, used interchangeably to refer to a unit of data moved or persisted within Kafka

  ◇ The term *record* is more often used to refer to a persisted message/event

- Internally, Kafka uses instances of the *ProducerRecord* class for published messages (https://tinyurl.com/2eb8r8vk)

2.4 Message Offset

- A message offset in Kafka is used as a message id

- It is a per-partition monotonically increasing integer [0,1,2,3, ...] that is unique within a partition

- An offset along with the partition id and broker node id uniquely identifies the location of the message on a Kafka cluster

2.5 Message Retention Settings

- During topic creation, you can configure message retention (applicable to all topic's partitions) using these two options:

  - ◇ **Retention time**

    - Controlled by the *retention.ms* parameter (-1 for unlimited storage time), or

  - ◇ **Retention size**

    - Controlled by the *retention.bytes* parameter (-1 for uncapped size)

  - ◇ **Note:** Those topic-level settings can be combined together such that whichever one comes first takes effect

2.6 Deleting Messages

- Messages identified as exceeding the established retention threshold, are either

    ◇ Deleted (the default setting), or

    ◇ Compacted (more on compaction a bit later ...)

    ◇ This strategy is controlled by the *cleanup.policy* parameter configured at the topic creation time

- When messages from a partition file are deleted (e.g. due to resetting the logs after the retention period), the new record's offset is set to the last recorded offset used + 1

2.7 The Flush Policies

- Flushing is forcing a disk write operation (the hard *fsync* operation on Linux) on messages that could potentially be cached by the IO subsystem and lazily flushed by the system

- Not flushing messages in time may result in lost messages

- The important trade-offs to be aware of:

  - *Durability*: Unflushed data may be lost if you are not using replication

  - *Latency*: Very large flush intervals may lead to latency spikes

  - *Throughput*: The flush comes at a cost – small flush intervals will flood your disk controller

- You have an option to flush over an interval or after a number of messages (or both at the same time)

- The flush policy is controlled in the *server.properties:*

  - It can be overridden on a per-topic basis

  - *log.flush.interval.messages=10000*

  - *log.flush.interval.ms=1000*

2.8 Writing to Partitions

- Messages are written to a partition in an append-only fashion

- Producer sends the data directly to the (partition) lead broker, which takes care of most of the housekeeping tasks
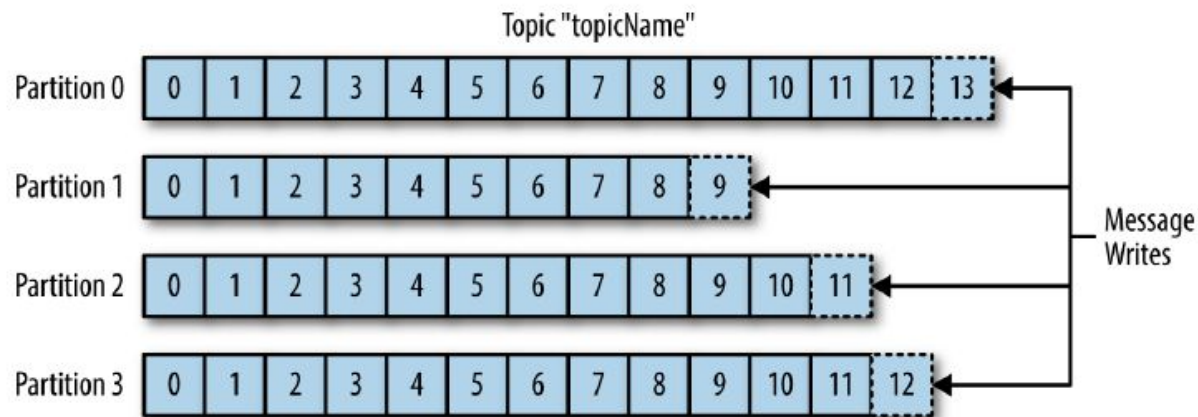
Source: *Kafka: The Definitive Guide*

2.9 Batches

- To improve overall Kafka throughput, brokers write messages to partitions in batches

    ◇ Batches are typically compressed for more efficient data transfer

    ◇ Messages are batched by topic-partition

- Batching can be configured as either a fixed number of messages or a batching window (e.g. 128K or 50 ms)

- **Note**: Using batches results in message buffering which introduces latency between a message production and the time it becomes a persistent record in Kafka (a factor contributing to a consumer lag situation)

2.10 Batch Compression

- For further efficiency, Kafka uses batch compression

- The batch of messages is written to the log in compressed form

- Consumers decompress the batch on read

- Kafka supports GZIP, Snappy, and LZ4 compression protocols

2.11 Partitions as a Unit of Parallelism

- A partition acts as a unit of write/read parallelism
  - ◇ More partitions allow for greater parallelism for message production and consumption at the expense of more files being stored across broker nodes
    - Adds complexity in partition replicated environments that may lead to saturating network bandwidth in the cluster and cases of under-replicated partitions → data loss!

2.12 Message Ordering

- Kafka provides strict record ordering within a single partition

- The onus of ordering across topic's partitions is on developers who would need to use record timestamps for application-controlled ordering in their apps

2.13 Kafka Default Partitioner

- Kafka default partitioner, which is part of the Kafka client library, uses the following logic for determining the target partition of a message created by the producer:

  ◇ If the producer provides a partition number in the message record, that partition is used

  ◇ If a producer provides a key (a message meta-parameter), the partitioner will calculate the partition id based on a hash value of the key

  ◇ When no partition number or key is present, Kafka default partitioner will pick a partition in a round-robin fashion

- Developers can create custom partitioners to better accommodate their problem domain requirements

2.14 The Load Balancing Aspect

- **Key**-less messages (the **key** meta-parameter is null) are written to a partition randomly (Kafka provides round-robin load-balancing) unless a key is provided as message metadata

  - ◇ That means that partitioning can also help with load balancing over brokers

- A Kafka message can have an optional **key** meta-parameter that makes message distribution more deterministic: messages with the same key are routed by the broker to the same partition

  - ◇ This can be of great value in apps that, for example, use the event sourcing pattern

2.15 Kafka Message Production Schematics
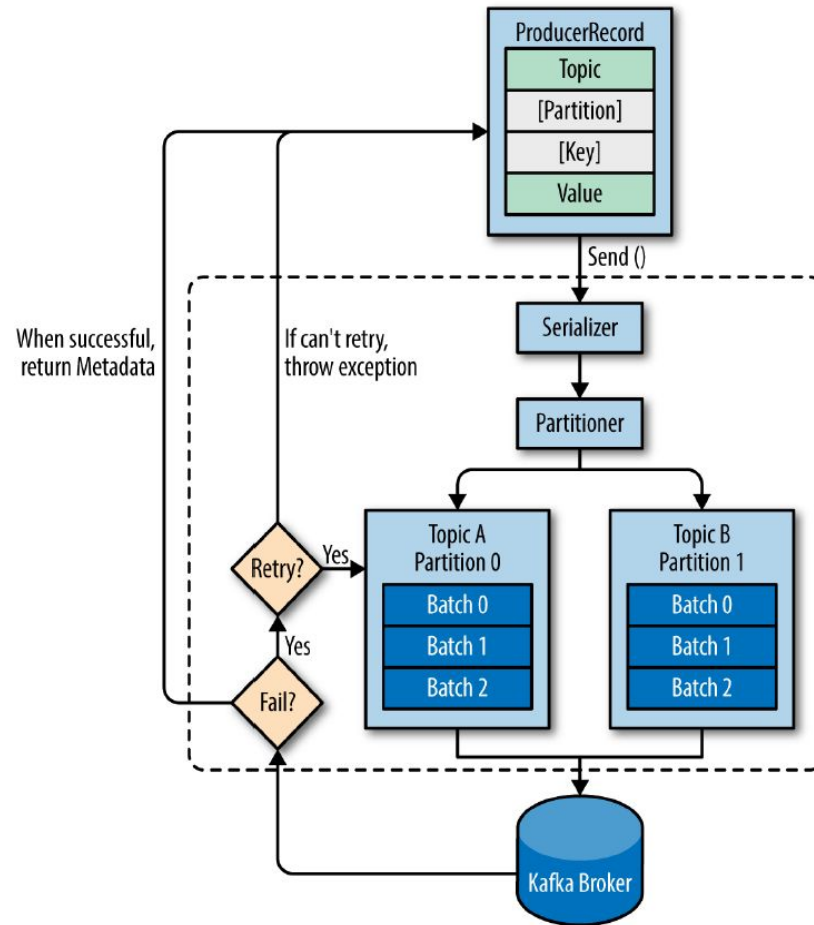
Source: *Kafka: The Definitive Guide*

2.16 Reading from a Topic

- A consumer (message reader from a topic) subscribes to one or more topics and performs sequential or random reads

- To read a record, a consumer needs three pieces of information:

  ◇ Topic name,

  ◇ Partition id within the topic, and

  ◇ Offset of the record within the partition (the "offset pointer")

  ◇ For efficiency, a consumer can request a block of records

- The "fetch" request is processed by the broker leading the partition

- A consumer can go back and re-read a previously fetched record

  ◇ The system retains the record ids (offsets) a consumer has read

2.17 Consumer Lag

- A metric that defines how far the consumer is from reading the latest message committed to the partition

- The ideal lag value is 0 with many factors contributing to larger values

2.18 Consumer Group

- In essence, a bunch of consumers with a common group id

- Grouping of consumers helps parallelize reads across different consumers in the same group

- Kafka guarantees a message is only read by a single consumer in a group

  - Each topic partition is assigned to a single consumer in the group

  - The following constraint applies:

    - $\sum$ (consumers in a group) $<=$ $\sum$ (partitions in a topic)

- Kafka checks for liveliness of consumers in a group

- In case of a consumer failure, Kafka rebalances the partitions between the remaining consumers

- Consumer group metadata is maintained in ZK

2.19 Consumer Group Diagram

4 partition : 2 consumers in a consumer group	4 partition : 4 consumers in a consumer group	4 partition : 2 consumer groups
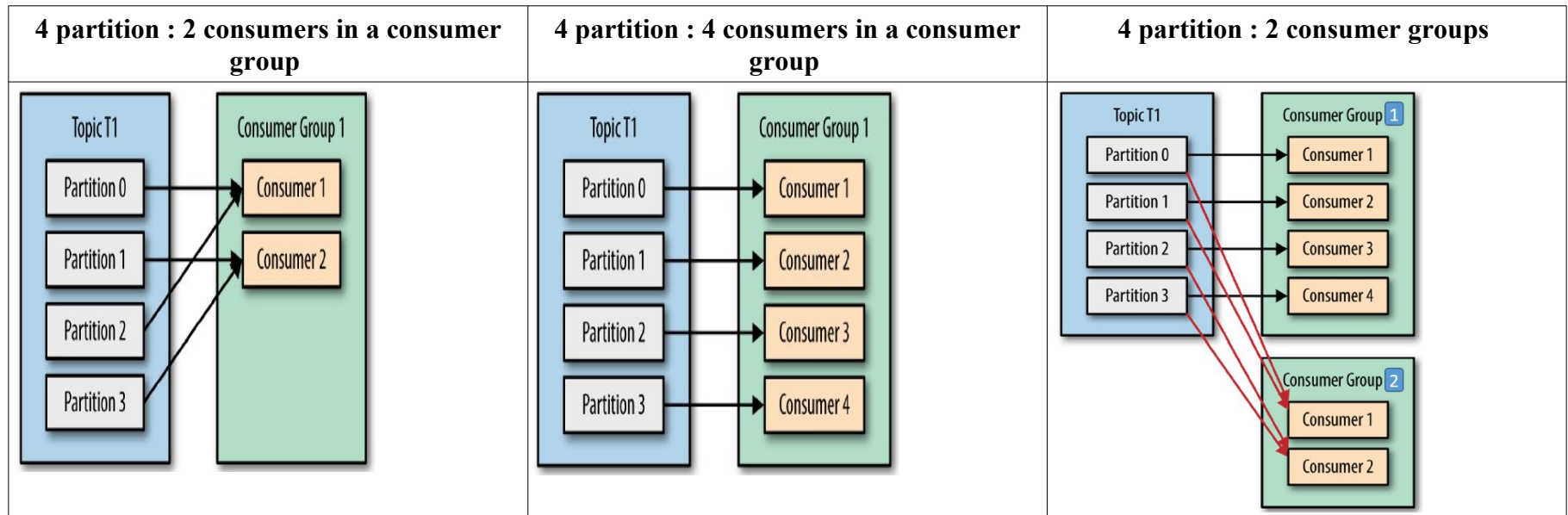
Adapted from *Kafka: The Definitive Guide*

2.20 The Broker

- Acks the message receipt from the producer

- Commits the message to disk

  ◇ Consumers can now read the message

- Replicates data

  ◇ One copy is committed to the "main" partition, the other replicas are sent to machines on the cluster

  ◇ Under-replicated data may lead to data loss

- In a cluster, one broker service is dedicated to act as the controller, performing administrative tasks like managing the states of partitions and replicas

- One broker process that handles all client (producers and consumers) reads and writes for a partition is called the "(partition) leader"

  ◇ more on this concept in a moment ...

2.21 The Leader and Followers Pattern

- The broker process managing the main partition replica acts as the "leader"; zero or more servers act as "followers"

  ◇ Followers exist when the replica is set > 1

- The leader handles all client (producers and consumers) reads and writes

  ◇ The followers passively replicate the leader's contents

- In case of the leader's failure, one of the followers will be automatically elected the "leader" (done by ZK)

- Each broker hosting partitions can act as a leader for some of its partitions and a follower for others

  ◇ This arrangement ensures load balancing within the cluster

- If there is a broker conflict on owning the main partition (due to a system failure), the partition is marked as leaderless and it is taken offline

2.22 Partition Replication Diagram

producer

topic1 - part1   topic1-part1   topic1- part1

topic1-part2   topic1-part2   topic1-part2

topic1- part3   topic1- part3   topic1- part3

topic1- part4   topic1- part4   topic1- part4

broker 1   broker 2   broker 3   broker 4

LEADER   FOLLOWER
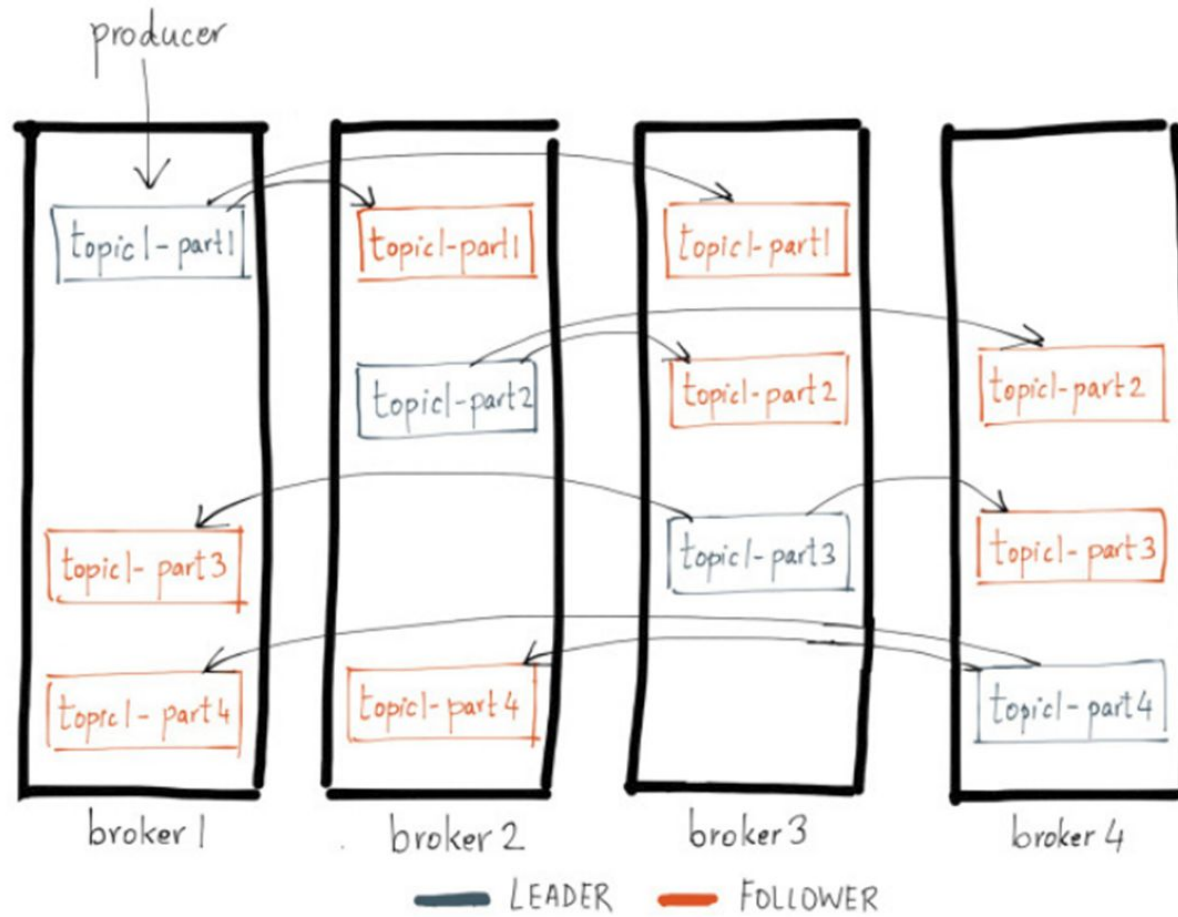
- Topic configured with *replication.factor = 3*

Source: Apache Kafka Confluent Documentation

2.23 Controlled Shutdown

- A shutdown may be needed to do a rolling upgrade

  ◊ If replicas are very much behind, a controlled shutdown may take substantial time to shut down the cluster

2.24 Controlling Message Durability with Minimum In-Sync Replicas

- This is done from the two sides: the producer's and the broker's

- A producer can set the *acks* request attribute to "all" (or "-1"), the server can further adjust this request through the value of the *min.insync.replicas* broker configuration parameter, which specifies the minimum number of replicas that must acknowledge a write for the write to be considered successful

- This parameter should be chosen based on the availability vs consistency trade-off analysis

    ◇ Consult business stakeholders if in doubt

2.25 Log Compaction

- Log compaction is an optional feature that allows you to remove old records where more recent records with the same keys exist

  ◇ Similar to the SQL's update operation

  ◇ If you request a compacted record, you will get the latest record with the same key

- It gives you a per-record retention control, leading to smaller log sizes

- Message ordering is preserved

2.26 Log Compaction

Offset	0	1	2	3	4	5	6	7	8	9	10	
Key	K1	K2	K1	K1	K3	K2	K4	K5	K5	K2	K6	Log Before Compaction
Value	V1	V2	V3	V4	V5	V6	V7	V8	V9	V 10	V 11	

Compaction

	3	4	6	8	9	10	
Keys	K1	K3	K4	K5	K2	K6	Log After Compaction
Values	V4	V5	V7	V9	V 10	V 11	
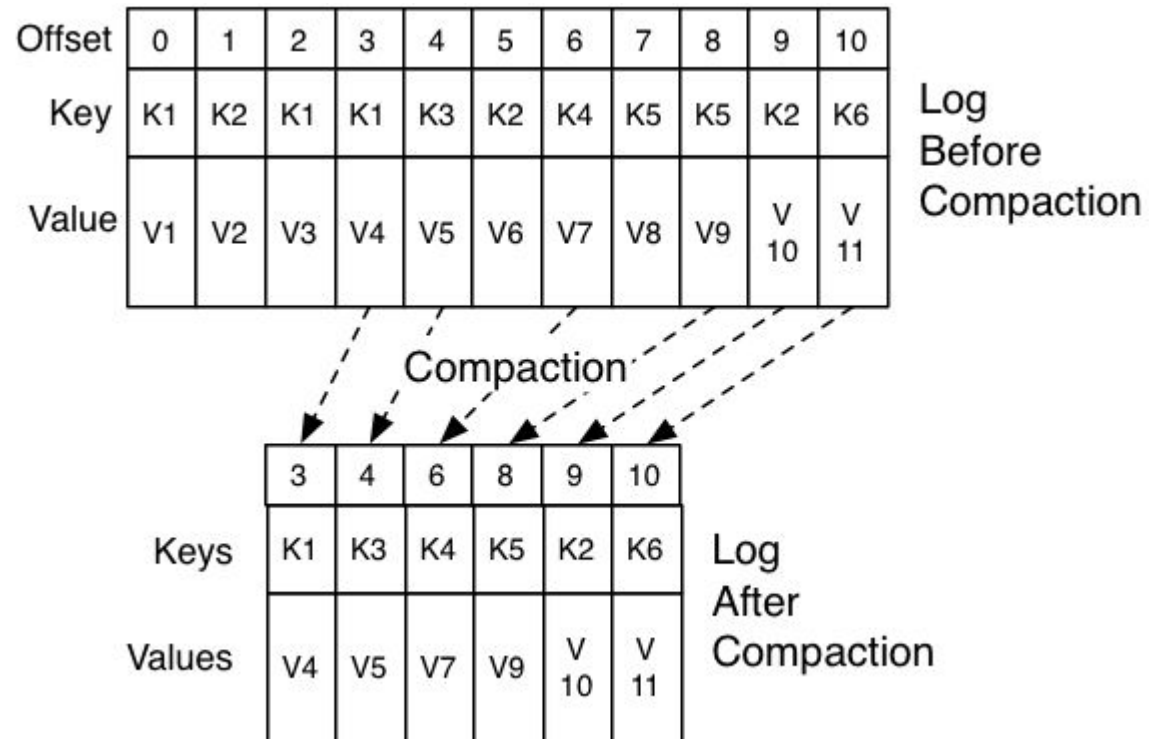
Source: Apache Kafka Documentation

2.27 Operational Problems

- Most frequently problems arise as a result of:

  - **Rebalancing**

    - Assignment of consumers in consumer groups to partitions (brokers) is dynamic -- new consumers can join the consumer group, or die. Any such group change events results in rebalancing: re-assignment of the active consumers to partitions, which may take a while to accomplish during which time consumers are unassigned from the original partitions and left in off-line mode (no processing happens at this time!)

  - **Consumer lag**

    - The consumer side problem:

      - Slow message processing (possibly, unoptimized GC policy)

      - Network issues

      - Rebalancing!!

2.28 Summary

- In this chapter, we reviewed a number of important internal details of Kafka's functionality

Chapter 3 - Using Apache Kafka

Objectives

Key objectives of this chapter

- Setting up Confluent Cloud on Azure

- Basics of Confluent Cloud Console

- Installing the CLI

- Using Kafka Command Line Client Tools

- Kafka Cluster Planning

- Manage topics in Cloud Console

- Adding Kafka dependency

- Creating a .NET Core Kafka Client

- Monitor cluster activity

3.1 What is Confluent?

- Confluent (https://www.confluent.io/) was formed by the original Kafka engineers with a focus on commercialized Kafka offerings with proprietary extensions and subscriptions

- Kafka (with Confluent proprietary extensions and technologies) is offered as

  - ◇ **Confluent Platform**, an on-premises enterprise platform [https://docs.confluent.io/platform/current/platform.html], or

  - ◇ **Confluent Cloud**, a fully-managed cloud-native service [https://www.confluent.io/confluent-cloud/]

    - Confluent Platform is deployed to other cloud environments like GCP, Azure, or AWS -- Confluent does not maintain a cloud of its own for hosting purposes and acts as a front to the hosting clouds

3.2 Confluent Cloud

- Fully-managed Kafka-based cloud service that comes with

  - Cloud Portal UI

    - In fact, just a front to the target hosting clouds ...

  - Out-of-the-box fully-managed connectors to the most popular data sources/sinks

  - ksqlDB

  - A governance suite for streaming data, and more ...

- Built on top of Confluent Platform

- Users deploy their Confluent clusters in any of the supported cloud environments:

  - AWS, Azure, or GCP

3.3 Confluent Cloud Resource Hierarchy

- Confluent Cloud has the following hierarchy of resources:

  ◇ **Environment > Cluster > Topic**

- You can have

  ◇ multiple environments,

  ◇ multiple clusters within a single environment, and

  ◇ multiple topics within a cluster

3.4 Setting up Confluent Cloud using Confluent.io

- Use this option if you don't have any existing Azure subscription and want to host Confluent Cloud on Azure.

- Sign-up for a Confluent.io account: https://www.confluent.io/get-started/

- The Confluent.io account itself is free. You only pay for the Kafka cluster set up.

3.5 Select the Confluent Cloud Cluster Type

- You begin by selecting a Confluent Cloud cluster type

  ◇ Initially, you are provided with the default environment named *default* from where you perform all the Confluent Cloud setting-up activities

  ◇ You can create additional environments to better align with cloud operational needs

- Cluster type choices: **Basic**, **Standard**, and **Dedicated**:

- You can review the Confluent Cloud type comparison chart at https://tinyurl.com/2p8nj78m

3.6 Choose the Cloud Provider

- Along with the target cloud platform that will host your Confluent clusters, you have a choice of choosing the deployment region and availability options (*Single* or *Multiple zones* within a region) in that cloud

- You can select one of the following cloud providers:

  ◇ Microsoft Azure, AWS, Google Cloud

- After selecting the cloud provider, you can enter the credit card details if you plan to use it in the production environment or use the free trial.

- The free option provides you with $400 to spend during your first 60 days.

3.7
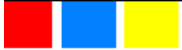
3.8 The Cluster View

- Once you created a cluster, it becomes part of the environment from where you created it (or the *default* one)

- Clusters are always shown in the **Running** status (there is no way to stop a cluster)

3.9 Exploring the Confluent Cloud Console

- The Confluent Cloud console lets you do the following: (Note: Not every feature will be covered in this course.)

  - ◇ Adding and deleting Apache Kafka clusters

  - ◇ Managing Accounts and Access

  - ◇ Creating and viewing API keys to access the cluster

  - ◇ Creating Confluent Cloud Schema Registry

  - ◇ Getting Schema Registry API keys to access the Schema Registry

  - ◇ Managing Topics

  - ◇ Cluster Linking to easily migrate data to other clusters

  - ◇ ksqlDB stream processing

  - ◇ Managing Billing and Quotas

  - ◇ Monitoring the cluster

3.10 Managing Topics in Confluent Cloud Console

- You can create, edit, and delete topics using the Confluent Cloud console.

- Ensure you navigate to the environment and then select a cluster where you want to create a topic.

- Click the **Topics** in the navigation menu and then click **Create topic**. If you already have an existing topic, you can click **Add a topic** to create another topic.
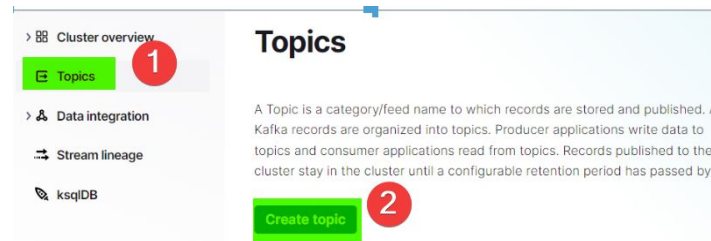
3.11 Topics

- You add a topic to your cluster on the *Topics* page:

- You can select any of these message retention strategies (or a combination thereof):
  - ◇ Retention time (from 1 hour to infinite duration)
  - ◇ Retention size (from 1 hour to infinite size)
- Maximum message size in bytes

3.12 Searching for Messages in a Topic

- Once you created a topic, you can start performing publishing and consuming operations against it

- The topic's UI offers users a widget to search for messages in a specific partition by offset or by timestamp

- You also have an option to manually produce messages to the topic using UI

3.13 The Confluent CLI

- Confluent Cloud does not have an embedded CLI console

- The Confluent Cloud CLI tool can be installed locally on your computer for remote management of your Confluent Cloud clusters

- The Confluent Cloud CLI client -- the *confluent* tool -- can be downloaded from https://docs.confluent.io/confluent-cli/current/overview.html

  ◇ The tool provides an interactive shell with command auto-completion

- You can use the CLI tool to produce and consume messages, as well as perform a wide range of administrative tasks

3.14 The confluent CLI Command Examples

confluent login
`// You need to provide your Confluent Cloud credentials here`

confluent environment list

confluent environment use <*environment id***>**

confluent kafka cluster list

confluent kafka topic create <topic> --partitions <*int32, default 6***> --cluster** <*Cluster id*> **--config** <*A comma-separated list of configuration overrides ("key=value") for the topic being created*>

confluent api-key create --resource <*cluster id*>

confluent kafka topic consume <*topic name***> --offset <***Offset #***> --partition <**Partition **#> 0 --api-key <**Your API Key**> --api-secret <***Your Secret***>**

confluent kafka topic delete <*topic name***>**

3.15 Kafka Cluster Planning – Producer/Consumer Throughput

- When planning a Kafka cluster, the following two areas should be considered:
  - ◇ Sizing for throughput
  - ◇ Sizing for storage
- Know the expected throughput of your Producer(s) and Consumer(s)
- The system throughput is as fast as your weakest link
- Consider the message size of the produced messages
- Know how many consumers would be consuming from the Topics/Partitions
- At what rate would the consumer(s) be able to process each message given the message size..

3.16 Kafka Cluster Planning – Sizing for Topics and Partitions

- The number of partitions depend on the desired throughput and the degree of parallelism that your Producer / Consumer ecosystem can support.

- Generally speaking, increasing the # of partitions on a given topic , linearly increases your throughput

- The throughput bottleneck could end up being the rate at which your Producer can produce or the rate at which your consumers can consume.

- Simple formula to size for topics and partitions:

  ◊ Lets say the desired Throughput is "t".

  ◊ Max Producer throughput is "p"

  ◊ Max Consumer throughput is "c".

  ◊ Number of Partitions = max (t/p, t/c).

- A rule of thumb often used is to have at least as many partitions as the number of consumers in largest consumer group.

3.17 Kafka Cluster Planning – Sizing for Topics and Partitions (contd.)

- You can use the following formula to get the partition size based on the the single Kafka topic throughput:

```
number of partitions = desired throughput / partition speed
```

- You can get the throughput from the Confluent Cloud console by navigating to the topic that you are interested in.

- Conservatively, you can estimate that a single partition for a single Kafka topic runs at 10 MB/s.

- As an example, if your desired throughput is 5 TB per day. That figure comes out to about 58 MB/s. Using the estimate of 10 MB/s per partition, this example implementation would require 6 partitions.

3.18 Kafka and .NET

- You can develop any type of .NET Kafka client application, such as console, MVC, Web API, and Worker

- The Kafka .NET library is developed and maintained by Confluent through the *confluent-kafka-dotnet* GitHub project [https://github.com/confluentinc/confluent-kafka-dotnet]

- Essentially, a C# client for Apache Kafka

- Compatible with

  - Kafka brokers >= v0.8,

  - Confluent Cloud, and

  - Confluent Platform

- It is a lightweight wrapper around *librdkafka*, a highly optimized C client

  - *librdkafka* is also at the core of the confluent-kafka-**python** and confluent-kafka-**go** projects

3.19 .NET Kafka Architectures

- You can have various architectures involving a consumer and a producer.

- In the following architecture, an MVC application directly writes a message to a Kafka topic.

- In the following architecture, an MVC application forwards requests to a Web API that writes a message to a Kafka topic. A background worker subscribes to the Kafka topic, consumes messages, and adds records to a database.

3.20 Packages

- Kafka packages for .NET are distributed via *NuGet*

- **Confluent.Kafka** - The core client library

- If you plan to use Confluent Schema Registry, you also require the following packages:

  - ◇ **Confluent.SchemaRegistry.Serdes.Avro** - Provides a serializer and deserializer for working with Avro serialized data with Confluent Schema Registry integration

  - ◇ **Confluent.SchemaRegistry.Serdes.Protobuf** - Provides a serializer and deserializer for working with Protobuf serialized data with Confluent Schema Registry integration

  - ◇ **Confluent.SchemaRegistry.Serdes.Json** - Provides a serializer and deserializer for working with Json serialized data with Confluent Schema Registry integration

  - ◇ **Confluent.SchemaRegistry** - Confluent Schema Registry client (a dependency of the Confluent.SchemaRegistry.Serdes packages)

3.21 Installing the Packages

- You can install the packages in various ways, including by using Visual Studio and the dotnet command.

- If you are using Visual Studio, right-click **Dependencies** and select **Manage NuGet Packages...**.

- In the **Browse** tab search for **Confluent.Kafka**

- If you are using the dotnet command-line option, use the following command:

```
dotnet add package Confluent.Kafka --version <version>
```
- **Note**: At the time of the writing of this course, 1.9.0 is the latest version.

3.22 Navigating .NET Client Documentation

- .NET Client API documentation site can be accessed by using the following URL:
  https://tinyurl.com/mruphss7

- Use the search widget on the *confluent-kafka-dotnet* documentation page to narrow down your searches:

- The Consumer and Producer client documentation can be found via their related interfaces: *IConsumer* and *IProducer*

3.23 Important Classes and Interfaces

- **ProducerConfig** - represents a Kafka producer configuration.

- **ConsumerConfig** - represents a Kafka consumer configuration.

- **ClientConfig** - a wrapper class that can wrap ConsumerConfig or ProducerConfig object.

- **IProducer** - a high-level producer interface.

- **IConsumer** - a high-level consumer interface.

- **ProducerBuilder** - used to construct a IProducer object from the ProducerConfig configuration.

- **ConsumerBuilder** - used to construct a IConsumer object from ConsumerConfig configuration.

- **AdminClientBuilder -** used to constuct an admin build used to perform admin operations, such as create/delete topics.

3.24 appsettings.json Kafka Configuration

- Although, you can use ClientConfig/ConsumerConfig/ProducerConfig objects directly in C#, but the best practice is to set it in the appsettings.json file.

- Here's a sample appsettings.json fragment.

```
"MyKafkaProducerConfig": {
  "BootstrapServers": "...",
  "SaslMechanism": "...",
  "SaslUsername": "...",
  "SaslPassword": "...",
  "SecurityProtocol": "...",
  "SslCaLocation": "..."
  }
}
```

- Refer to the following URL for parameter details:

https://www.ibm.com/docs/en/cloud-paks/cp-biz-automation/20.0.x?topic=parameters-apache-kafka

3.25 Loading the Configuration from appsettings.json

- There are several ways to load the configuration from appsettings.json file. Here is one of the common approaches that can be configured in the Startup.cs file.

```
public void ConfigureServices(IServiceCollection services)
{
services.AddControllers();

var producerConfig = new ProducerConfig();
    Configuration.Bind("MyKafkaProducerConfig",producerConfig);

services.AddSingleton<ProducerConfig>(producerConfig);
}
```

- Then you can use dependency injection in a controller's constructor to make the configuration available to it.

```
private readonly ProducerConfig _config;

public MyControllerConstructor(ProducerConfig config)
{
   _config = config;
}
```

3.26 Produce and ProduceAsync Methods

- The .NET client offers two core methods for producing messages: *Produce* and *ProduceAsynch;* the skeleton use of both is shown below

```
var p = new ProducerBuilder<Null, string>(config).Build();
. . .
var dr = await p.ProduceAsync
("test-topic", new Message<Null, string> { Value="test" });

WriteLine($"Sent '{dr.Value}' to '{dr.TopicPartitionOffset}'");

// -----------------------------------------------
p.Produce
("test-topic", new Message<Null, string> { Value = "test" },(dr) =>
    {
        if (dr.Error.Code != ErrorCode.NoError)
        {
            WriteLine($"Failed to deliver message: {dr.Error.Reason}");
        }
        else
        {
            WriteLine($"Sent '{dr.Value}' to '{dr.TopicPartitionOffset}'");
        }
    });
```

3.27 Produce vs ProduceAsync

- So which method should I use? According to .NET client documentation:

- *You should use the **ProduceAsync** method if you would like to wait for the result of your produce requests before proceeding. You might typically want to do this in highly concurrent scenarios, for example in the context of handling web requests. Behind the scenes, the client will manage optimizing communication with the Kafka brokers for you, batching requests as appropriate ... but there will be a delay on each **await** call.*

- *In stream processing applications, where you would like to process many messages in rapid succession, you would typically use the **Produce** method instead*

3.28 Error Handling

- The Produce and ProduceAsync methods allows you to pass it a handler that lets you know if the message has been successfully written to a Kafka topic or not.

- Here's the code snippet that illustrates the handler parameter.

```
...
producer.Produce(topicName, message, deliveryHandler);
…
void deliveryHandler(DeliveryReport<string, string> deliveryReport)
        {

            if (deliveryReport.Error.Code == ErrorCode.NoError)
            {
                Debug.WriteLine($"\n* Message delivered to: ({deliveryReport.TopicPartitionOffset}) with these
details:");
                Debug.WriteLine($"-- Key: {deliveryReport.Key},\n-- Timestamp:
{deliveryReport.Timestamp.UnixTimestampMs}");

            }
            else
            {
                Debug.WriteLine($"Failed to deliver message with error: {deliveryReport.Error.Reason}");
            }
        }
```

3.29 Consuming Messages

- To consume messages, use ConsumerBuild to construct IConsumer object and call the Subscribe method.

```
IConsumer<string, string> consumer = new ConsumerBuilder<string, string>(this._config).Build();
consumer.Subscribe(topicName);

        var res = this._consumer.Consume();
        return res.Message.Value;
    }
```

- Typically, a consumer is implemented as a worker or BackgroundService that subscribes to a Kafka topic and continuously receives messages.

```
    public class UserRegistrationWorker : BackgroundService
    {
        ...
        protected override async Task ExecuteAsync(CancellationToken stoppingToken)
        {
            while (!stoppingToken.IsCancellationRequested)
            {
                string message = consumer.Consume.Message.Value;

                //Deserilaize the message and use it
                // deserialization logic

                Console.WriteLine($"---Info: User Registered: {message}---");


            }
        }
```

3.30 Creating and Deleting Topics

■ AdminClientBuild and AdminClient can be used to perform various operations, including:

◇ CreateTopicsAsync

◇ DeleteTopicsAsync

3.31 Using AdminClient

```
using (IAdminClient adminClient = new AdminClientBuilder(new
AdminClientConfig(clientConfig)).Build())
{
    try
    {
      // create or delete topic(s) here
    }
    catch (Exception e)
    {
    }
}
```

3.32 Deleting a Topic

```
var topicsToDelete = new List<string> { "topic-to-create" };

try
{
    await adminClient.DeleteTopicsAsync(topicsToDelete);
}
catch (Exception e)
{
    ViewBag.Message = e.Message;
}
```

3.33 Creating a Topic

```
var topicToCreate = new List<TopicSpecification> {
    new TopicSpecification { Name = "topic-to-create" }
};

try
{
    await adminClient.CreateTopicsAsync(topicToCreate);
    ViewBag.Message = "Topic created!";
}
catch (Exception e)
{
    ViewBag.Message = e.Message;
}
```

3.34 Copying Data from Between Environments

- To copy data from between environments, e.g. production to to test, you can use various options:

  ◇ Cluster Linking

  ◇ Confluent Replicator

  ◇ Kafka's Mirror Maker

- Alternatively, you can generate mock data for your topics with predefined schema definitions, including complex records and multiple fields by using Datagen Connector.

3.35 Mocking Datasets using Datagen Connector

- To mock datasets, you can use Datagen Connector.

- The Datagen Connector can be configured from Data inegration | Connectors page.

- The Datagen Connector lets you select an existing template.

3.36 Monitoring Confluent Cloud

- You can use the built-in monitoring options available in Confluent Cloud.

- Select the environment. It will list all clusters and various metrics, such as Product/Consumption throughput, Storage, Topics client, and number of clients connected to Kafka.

3.37 Monitoring Confluent Cloud (contd.)

- To see more metrics, click the cluster you want to monitor and click **Cluster overview | Dashboard.**

- Click **Explore metrics.**

- You can specify the following three parameters:
  - ◇ Metric, Resource, and Time Frame

- The Time Frame is averaged over the Last hour, Last 6 hours, Last 24 hours, or Last 7 days
- There are several metrics listed in the drop-down. Here are some of the Kafka oriented metrics.
  - ◇ Received bytes, Sent bytes, Received records, Sent records, Active connection count, Request count, Partition count, and Successful authentication count.

3.38 Monitoring Confluent Cloud using cURL

- On the Metrics page, configure Metric, Resource, and Time Frame and then click Copy cURL template.

- The copied command can be executed from the command prompt using Confluent CLI.

- For example,Sent bytes metric for the cluster over Last hour cURL command looks like this:

```
curl 'https://api.telemetry.confluent.cloud/v2/metrics/cloud/query' \
-H "Content-Type: application/json" \
-d '{"aggregations":[{"metric":"io.confluent.kafka.server/sent_bytes"}],"filter":{"op":"OR","filters":
[{"field":"resource.kafka.id","op":"EQ","value":"lkc-o36739"}]},"granularity":"PT1M","intervals":["2022-08-
02T14:56:00-04:00/2022-08-02T15:56:00-04:00"],"limit":1000}' \
-H "Authorization: Basic base64(<API_KEY>:<API_SECRET>)"
```

3.39 Motoring Confluent Cloud using third-party Tools

■ You can also use third-party tools to monitor Confluent Cloud.

- ◇ Datadog

- ◇ Dynatrace

- ◇ Grafana Cloud

- ◇ Prometheus

- ◇ Splunk

3.40 Summary

- In this chapter, you learned about:
  - Confluent Cloud
  - Confluent Cloud resource hierarchy
  - Confluent CLI
  - .NET Kafka Client
  - Monitoring

3.41 Review Questions

⬦ What kinds of operations are available with Confluent Cloud?

⬦ What different types of applications does .Net allow development of?

⬦ What should you consider when planning a Kafka Cluster?

Chapter 4 - Building Data Pipelines

Objectives
Key objectives of this chapter
■ Considerations when building data pipelines
■ Schema for Evolving Data
■ Confluent Schema Registry

4.1 Building Data Pipelines
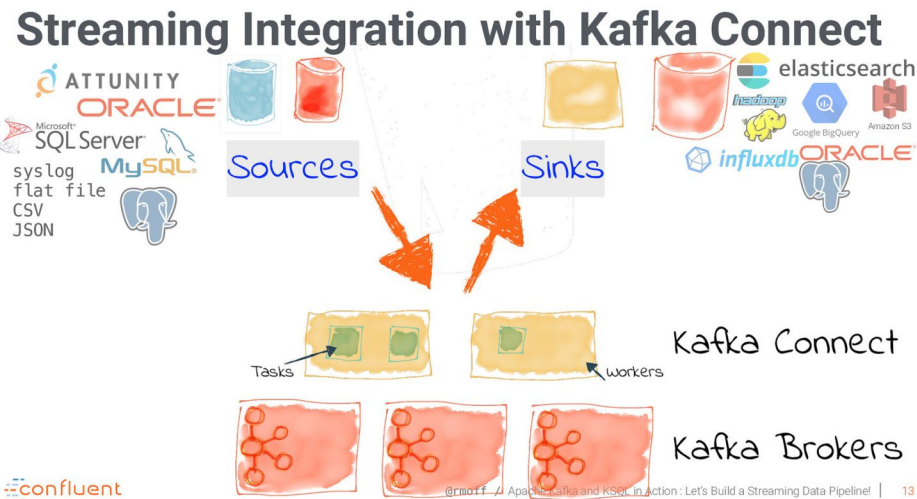
- Kafka-based data pipelines serve as a very large, reliable buffer between various stages in the pipeline, decoupling producers and consumers of data within the pipeline.

- The decoupling, combined with reliability, security, and efficiency, makes Kafka a very good solution to build most data pipelines.

- Here are some use cases that can involve data pipelines:

  ◇ A data pipeline where Apache Kafka is one of the two endpoints.

    - For example, getting data from Kafka to S3/storage account or getting data from

MongoDB/Cosmos DB into Kafka.

◇ A data pipeline between heterogeneous systems but using Kafka in between.

- For example, getting data from Twitter to Elasticsearch by sending the data first from Twitter to Kafka and then from Kafka to Elasticsearch.

4.2 What to Consider When Building Data Pipelines

- Timeliness

- Reliability

- High and varying throughput

- Data formats

- Transformations

- Security

- Failure handling

- Coupling and agility

4.3 Timeliness

- Data integration systems should support different timeliness requirements for different pipelines

- Kafka can be used to support anything from near-real-time pipelines to hourly batches.

- Producers can write to Kafka and consumers can also read the latest events as they arrive.

- Consumers can work in batches to connect to Kafka and read the events that accumulated since the last read.

- Kafka acts as a buffer that decouples the time-sensitivity requirements between producers and consumers.
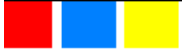
4.4 Reliability

- Data integration systems should avoid single points of failure and allow for fast, reliable, and automatic recovery from all sorts of failure events.

- Data pipelines are often the way data arrives in business critical systems

- Another important consideration for reliability is delivery guarantees

- Kafka offers reliable and guaranteed delivery.

4.5 High and Varying Throughput

- The data pipelines should be able to scale to very high throughput

- They should be able to adapt if throughput intermittently increases and reduces

- You no longer need to couple consumer throughput to the producer throughput since Kafka acts as a buffer between producers and consumers.

- If producer throughput exceeds that of the consumer, data will accumulate in Kafka until the consumer can catch up.

- Kafka is a high-throughput distributed system capable of processing hundreds of megabytes per second on even modest clusters

4.6 High and Varying Throughput (Contd.)

- Kafka's ability to scale by adding consumers or producers independently allows us to scale either side of the pipeline dynamically and independently to match the changing requirements.

- Kafka also focuses on parallelizing the work and not just scaling it out.

- Parallelizing allows data sources and sinks to split the work between multiple threads of execution and use the available CPU resources even when running on a single machine.

- Kafka also supports several types of compression that allows users and admins to control the use of network and storage resources as the throughput requirements increase.

4.7 Evolving Schema

- Schemas are the APIs used by event-driven services, so a publisher and subscriber need to agree on exactly how a message is formatted.

- This creates a logical coupling between sender and receiver based on the schema they both share.

- In the same way that request-driven services make use of service discovery technology to discover APIs, event-driven technologies need some mechanism to discover what topics are available and what data (i.e., schema) they provide.

- There are a few options available for schema management: Protobuf (Protocol Buffers) and JSON Schema are both popular, but most projects in the Kafka space use Avro.

- For central schema management and verification, Confluent has an open source Schema Registry that provides a central repository for Avro schemas.

4.8 Data Formats

- Data integration platforms should allow and reconcile heterogeneous data formats and data types.

- The data types supported vary among different databases and other storage systems.

  ◇ e.g. loading XMLs and relational data into Kafka and then convert data to JSON and CSV when writing it

- Kafka and the Connect APIs are agnostic when it comes to data formats.

- Producers and consumers can use any serializer to represent data in any format based on your business requirements.

4.9 Data Formats (Contd.)

- Kafka Connect has in-memory objects that include data types and schemas.

- It allows for pluggable converters to allow storing data in any format.

- Many sources and sinks have a schema that can be read from the source and use it to validate compatibility or update the schema in the sink database.

  - ◇ e.g. if someone added a column in Oracle, a pipeline will make sure the column gets added to SQL Server too as we are loading new data into it.

- Sink connectors are responsible for the format in which the data is written to the external system.

- Some connectors make this format pluggable.

  - ◇ For example, the HDFS connector allows a choice between Avro and Parquet formats.

4.10 Protobuf (Protocol Buffers) Overview

- Protocol Buffers are a way of encoding structured data in an efficient yet extensible format.

- Google developed Protocol Buffers for use in their internal services.

- It is a binary encoding format that allows you to specify a schema for your data using a specification language, like so:

```
message Person {
  required int32 id = 1;
  required string name = 2;
  optional string email = 3;
}
```

- The snippet defines the schema for a Person data type that has three fields: id, name, and email. In addition to naming a field, you can provide a type that will determine how the data is encoded and sent over the wire - above we see an int32 type and a string type. Keywords for validation and structure are also provided (required and optional above), and fields are numbered, which aids in backward compatibility

- The Protocol Buffers specification is implemented in various languages: Java, C, Go, etc.

- This means that one spec can be used to transfer data between systems regardless of their implementation language.

4.11 Avro Overview

- Apache Avro is an open source, row-based, compressed, and language-neutral data serialization system.

- It was developed by Doug Cutting, the father of Hadoop.

- A language-independent schema is associated with its read and write operations.

- Avro serializes the data which has a built-in schema.

- Avro serializes the data into a compact binary format, which can be deserialized by any application.

- Avro uses JSON format to declare the data structures.

- Schema is stored along with the Avro data in a file for any further processing.

4.12 Avro Schema Example

```
{
  "type": "record",
  "namespace": "com.mycorp.mynamespace",
  "name": "Person",
  "doc": "Person schema.",
  "fields": [
    {
      "name": "UserId",
      "type": "int",
      "doc": "This is the user id."
    },
    {
      "name": "Name",
      "type": "string",
      "doc": "First name."
    },
    {
      "name": "Email",
      "type": "string",
      "doc": "Email address."
    }
```

```
        ]
}
```

4.13 JSON Schema Example

```json
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "http://example.com/myURI.schema.json",
  "title": "Person",
  "description": "Person schema.",
  "type": "object",
  "additionalProperties": false,
  "properties": {
    "UserId": {
      "type": "integer",
      "description": "User id."
    },
    "UserName": {
      "type": "string",
      "description": "User name."
    },
    "Email": {
      "type": "string",
      "description": "Email address."
    }
  }
}
```

}

4.14 Managing Data Evolution Using Schemas

- Schemas provide a contract that defines what a message should look like.

- Most schema technologies provide a mechanism for validating whether a message in a new schema is backward-compatible with previous versions (or vice versa).

  - E.g. if you added a "return code" field to the schema for an order; this would be a backward-compatible change.

  - Programs running with the old schema would still be able to read messages, but they wouldn't see the "return code" field (termed forward compatibility).

  - Programs with the new schema would be able to read the whole message, with the "return code" field included (termed backward compatibility).

- Unfortunately, you can't move or remove fields from a schema in a compatible way

- This ability to evolve a schema with additive changes that don't break old programs is how most shared messaging models are managed over time.

4.15 Confluent Schema Registry

- The Confluent Schema Registry can be used to police this approach.

- The Schema Registry provides a mapping between topics in Kafka and the schema they use.

- It also enforces compatibility rules before messages are added.

- The Schema Registry will check every message sent to Kafka for the schema compatibility, ensuring that incompatible messages will fail on publication.

- The Confluent Schema Registry provides both runtime validation of schema compatibility, as well as a caching feature for schemas so they don't need to be included in the message payload.

Confluent
Schema Registry

Validate
Schema

Orders
Service

producer.send()

Cache
Schema
Locally

KAFKA
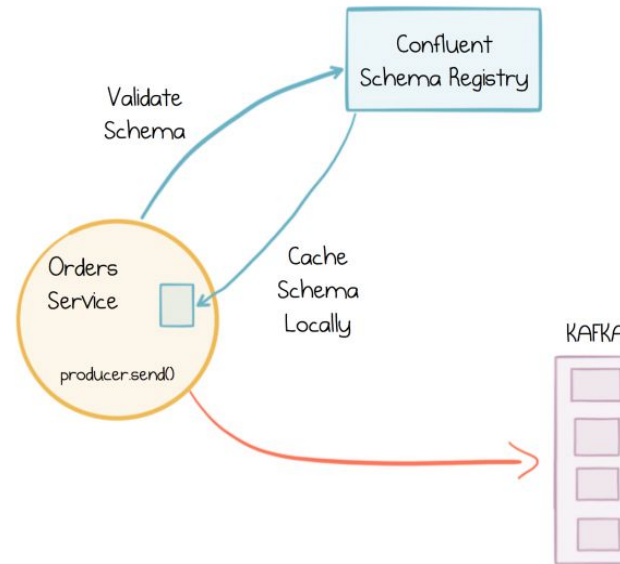
4.16 Confluent Schema Registry in a Nutshell

- Ensure data quality and evolvability

- Define data standards

- Evolve without breaking applications

4.17 Schema Management on Confluent Cloud

- Confluent Cloud Schema Registry can be used to manage schemas in Cloud Cloud.

- A Schema Registry is enable per Confluent Cloud environment.

- Registry is accessible over port 443.

- The Schema Registry provides a schema editor where you can edit schemas and associate them with Kafka topics.

- Schema Registry is enabled in Confluent Cloud by administrators.

4.18 Create a Schema

- You can create schemas for topics in various ways:

  ◊ Cloud Console

  ◊ Confluent CLI

  ◊ Schema Registry API (POST *subjects*(string: subject)/versions)

4.19 Create a Schema using Confluent CLI

- Create a file containing the following sample JSON and name it user-registration.json

```
{
    "type" : "record",
    "namespace" : "LearnSchemaRegistry",
    "name" : "UserRegistration",
    "fields" : [
        { "name" : "FirstName" , "type" : "string" },
        { "name" : "LastName" , "type" : "string" },
        { "name" : "Email" , "type" : "string" }
    ]
}
```

- Create a schema that uses user-registration.json.

```
confluent schema-registry schema create --subject user-registration-
value --schema user-registration.json --type AVRO
```

4.20 Create a Schema from the Web UI

- Select an environment and cluster.

- Select a topic then click the Schema tab for the topic.

- Select a schema format type.

- For example:

4.21 Schema Change and Backward Compatibility

- Evolving schemas are a fundamental attribute of the way data ages.

- Late-bound/schema-on-read approaches allow many incompatible data schemas to exist in the same table, topic, or the like at the same time. This pushes the problem of translating the format—from old to new—into the application layer, hence the name "schema on read."

- In many cases recent data is more valuable than older data, so programs can move forward without migrating older data they don't really care about.

- Schema on read can also be simple to implement if the parsing code for the previous schemas already exists in the codebase (which is often the case in practice).

- Most of the time backward compatibility between schemas can be maintained through additive changes (i.e., new fields, but not moves or deletes).

- Periodically schemas will need upgrading in a non-backward-compatible way. The most common approach for this is Dual Schema Upgrade Window, where we create two topics, orders-v1 and orders-v2, for messages with the old and new schemas, respectively.
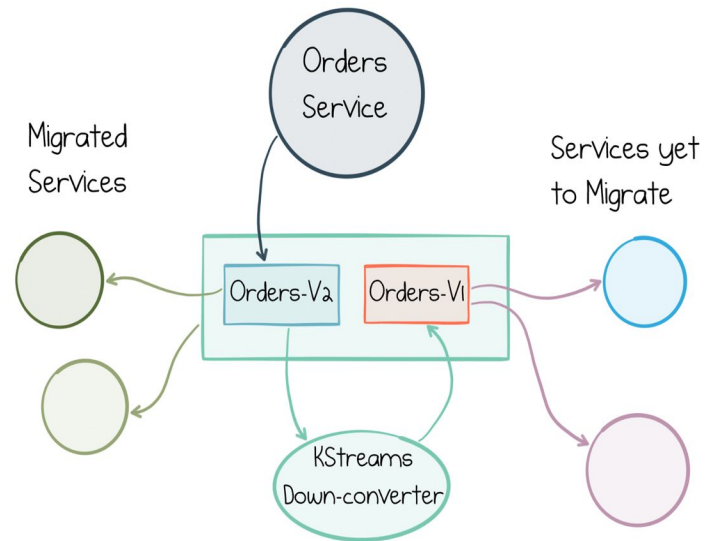
- Services continue in this dual-topic mode until fully migrated to the v2 topic, at which point the v1 topic can be archived or deleted as appropriate.

4.22 Collaborating over Schema Change

- Before rolling out schema change(s), some form of team-to-team collaboration should take place to work out whether the change is appropriate.

- The best approach is to use GitHub. This works well because:

  ◇ schemas are code and should be version-controlled for all the same reasons code is

  ◇ GitHub lets implementers propose a change and raise a pull request (PR), which they can code against while they build and test their system.

  ◇ Other interested parties can review, comment, and approve.

  ◇ Once consensus is reached, the PR can be merged and the new schema can be rolled out.

4.23 Handling Unreadable Messages

- Schemas aren't always enough to ensure downstream applications will work.

- There is nothing to prevent a semantic error—for example, an unexpected character, invalid country code, negative quantity, or even invalid bytes (say due to corruption)—from causing a process to stall.

- Such errors will typically hold up processing until the issue is fixed, which can be unacceptable in some environments.

- Traditional messaging systems often include a related concept called a dead letter queue,1 which is used to hold messages that can't be sent, for example, because they cannot be routed to a destination queue.

- Some implementers choose to create a type of dead letter queue of their own in a separate topic.

- If a consumer cannot read a message for whatever reason, it is placed on this error queue so processing can continue. Later the error queue can be reprocessed.

4.24 Deleting Data

- When you keep datasets in the log for longer periods of time, or even indefinitely, there are times you need to delete messages, correct errors or corrupted data, or redact sensitive sections.

- The simplest way to remove messages from Kafka is to simply let them expire.

- By default, Kafka will keep data for two weeks, and you can tune this to an arbitrarily large (or small) period of time.

- There is also an Admin API that lets you delete messages explicitly if they are older than some specified time or offset.

- When using Kafka for Event Sourcing, you typically don't need to delete. Instead, removal of a record is performed with a null value (or delete marker as appropriate). This ensures the fully versioned history is held intact, and most Connect sinks are built with delete markers in mind.

- But data isn't removed from compacted topics in the same way as in a relational database. Instead, Kafka uses a mechanism closer to those used by Cassandra and HBase, where records are marked for removal and then later deleted when the compaction process runs.

- Deleting a message from a compacted topic is as simple as writing a new message to the topic with the key you want to delete and a null value. When compaction runs, the message will be deleted forever.

4.25 Segregating Public and Private Topics

- When using Kafka for stream processing, in the same cluster through which different services communicate, we typically want to segregate private, internal topics from shared, business topics.

- Some teams prefer to do this by convention, but you can apply a stricter segregation using the authorization interface.

- You assign read/write permissions, for your internal topics, only to the services that own them.

- This can be implemented through simple runtime validation, or alternatively fully secured via TLS or SASL.

4.26 Transformations

- Data pipelines are typically implemented as:

  ◇ ETL (Extract-Transform-Load)

  ◇ ELT (Extract-Load-Transform)

4.27 Transformations - ELT

- **ETL** – the data pipeline is responsible for making modifications to the data as it passes through.

    ◇ It saves time and storage because there is no need to store the data, modify it, and store it again.

    ◇ The burden of computation and storage is shifted to the data pipeline itself, which may or may not be desirable.

    ◇ It's not recommended in scenarios where you wish to process the data farther down the pipe.

    ◇ If users require access to the missing fields, the pipeline needs to be rebuilt and historical data will require reprocessing (assuming it is available).

4.28 Transformations - ELT

- **ELT** – the data pipeline does the only minimal transformation (mostly involving data type conversion) and ensures the data that arrives at the target is as similar as possible to the source data.

  - ◇ The target system's CPU and storage resources are utilized for transformations.

  - ◇ These are also called data-lake architecture or high-fidelity pipelines.

  - ◇ The "raw data" collection and processing is done at the target system.

  - ◇ These systems are easier to troubleshoot since all data processing is limited to one system rather than split between the pipeline and additional applications.

4.29 Security

- The main security concerns in a data pipeline are:

  ◇ **Encryption** - encrypting the data going through the pipe especially where the cross datacenter boundaries are involved. Kafka's encryption feature ensures the sensitive data can't be piped into less secured systems by unauthorized users/applications.

  ◇ **Authorization** - who is allowed to make modifications to the pipelines?

  ◇ **Authentication** - can the data pipeline authenticate properly if it needs to read or write from access-controlled locations? Kafka supports authentication via SASL.

- Kafka provides an audit log to track access—unauthorized and authorized.
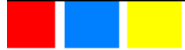
4.30 Failure Handling

- Failure handling should be planned in advance, such as:

  ◇ prevent faulty records from making it into the pipeline

  ◇ recover from records that cannot be parsed

  ◇ fix and reprocess bad records

- Kafka stores all events for long periods of time, therefore it is possible to go back in time and recover from errors when needed.

4.31 Agility and Coupling

- Decoupling the data sources and data targets is one of the main goals.

- There are multiple ways accidental coupling can happen:

    ◇ Ad-hoc pipelines

    ◇ Metadata Loss

    ◇ Extreme processing

4.32 Ad-hoc Pipelines

- Often organizations build a custom pipeline for each pair of applications they want to connect. For example:

  - ◇ Use Informatica to get data from MySQL and XMLs to Oracle

  - ◇ Use Flume to dump logs to HDFS

  - ◇ Use Logstash to dump logs to Elasticsearch

- This tightly couples the data pipeline to the specific endpoints that requires significant effort to deploy, maintain, and monitor.

- Data pipelines should only be built for the necessary systems.

4.33 Metadata Loss

- The software producing the data at the source and the software using it at the destination can get tightly coupled if the data pipeline doesn't preserve schema metadata and does not allow for schema evolution.

- Both software products need to include information on how to parse the data and interpret it if the schema information isn't available. For example:

  - ◇ If data flows from Oracle to MongoDB and a DBA added a new field in Oracle without preserving schema information and allowing schema evolution, either every app that reads data from MongoDB will break or all the developers will need to upgrade their applications at the same time.

- Each team can modify their applications without breaking the data pipeline if support for schema evolution is included in the pipeline

4.34 Extreme Processing

- Processing/transformation of data is always required by data pipelines

- Processing ties all the downstream systems to decisions made when building the pipelines. For example,

  - which fields to preserve

  - how to aggregate data

  - …

- This often leads to constant changes to the pipeline as requirements of downstream applications change, which isn't agile, efficient, or safe.

- The more agile way is to preserve as much of the raw data as possible and allow downstream apps to make their own decisions regarding data processing and aggregation.

4.35 Kafka Connect vs. Producer and Consumer

- Writing to Kafka or reading from Kafka can be done in of these two ways:

  - ◇ traditional producer and consumer clients

  - ◇ the Connect APIs and the connectors

- Kafka clients approach is preferred when you can modify the code of the application that you want to connect an application to and when you want to either pull data from Kafka or push data into Kafka.

- The Connect approach is preferred to connect Kafka to datastores that you did not write and whose code you cannot or will not modify.

- Connect is used to pull data from the external datastore into Kafka or push data from Kafka to an external store.

4.36 Kafka Connect vs. Producer and Consumer (Contd.)

- Connect can be used by non-devlopers for datastores where a connector already exists and the non-developers will only need to configure the connectors.

- Connect provides out-of-the-box features like configuration management, offset storage, parallelization, error handling, support for different data types, and standard management REST APIs.

- If a connector does not exist, you can choose between writing an app using the Kafka clients or the Connect API.

- When writing an app, there are many little details you will need to handle data types and configuration that make the task challenging.

- Kafka Connect handles most of the configuration details for you, allowing you to focus on transporting data to and from the external stores.

4.37 Summary

- Kafka can be used to implement data pipelines

- When designing the data pipelines, various factors should be considered.

- One of the most important Kafka features is its ability to deliver all messages under all failure conditions.

- Often schema change can be managed simply by evolving the schema with a format like Avro or Protobuf that supports backward compatibility.

- At other times evolution will not be possible and the system will have to undergo a non-backward-compatible change.

4.38 Review Questions

◇ What is a Kafka Data Pipeline? How can it be implemented?

◇ What is the significance of schemas used with Kafka?

◇ Transformations are a key concept with Kafka. What is ETL vs ELT?

Chapter 5 - Integrating Kafka with Other Systems

Objectives
Key objectives of this chapter ■ Integrating Kafka with Other Systems ■ Kafka Connect

5.1 Introduction to Kafka Integration

- Kafka supports integration with various technologies to implement different use cases, such as:

    ◇ Real-time processing using Storm

    ◇ Spark Streaming

    ◇ Batch processing using Hadoop

- LinkedIn uses it to power the LinkedIn news feed by capturing activity data and operational metrics.

- Twitter's stream-processing infrastructure is powered by Kafka.

- Foursquare uses Kafka to integrate monitoring and production systems with Hadoop-based offline infrastructures.

- Square uses Kafka as a bus to move system events, such as metrics, logs, and custom events, through Square's various data centers. On the consumer side, it outputs into Splunk, Graphite, or Esper-like real-time alerting.

Note:

There are several other companies who use Kafka. Netflix uses it for 300-600BN messages per day

It's also used by Airbnb, Mozilla, Goldman Sachs, Tumblr, Yahoo, PayPal, Coursera, Urban Airship, Hotels.com, and other major organizations
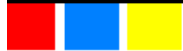
5.2 Kafka Connect
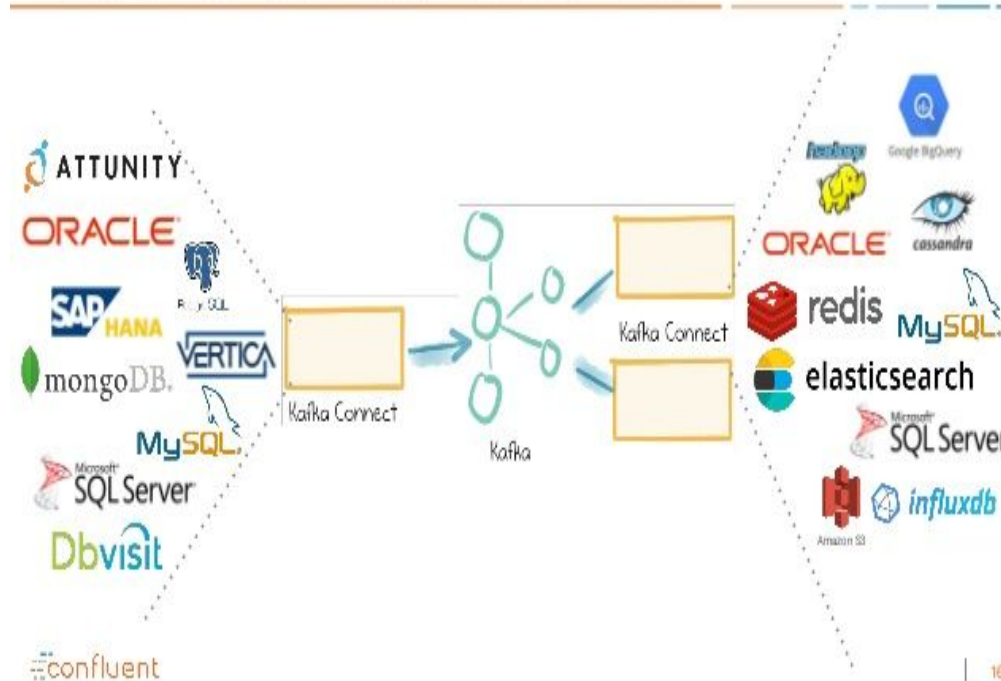
- Kafka Connect is a tool included with Kafka that can be used by non-developers to import data in to and export data from Kafka.

- It has support for pluggable connectors that connect to various external systems.

- Alternatively, you can create a custom application using the Connect API to perform the

import/export operations.

5.3 Kafka Connect (Contd.)

- Connectors move large amounts of data in parallel and use the available resources on the worker nodes.

- Source connector read data from the source system provide Connect data objects to the worker processes.

- Sink connector get connector data objects from the workers and writes them to the target data system.

- Kafka Connect uses converters to support storing data objects in Kafka in different formats, such as JSON, Avro, and Parquet.

- Data format can be selected independent of the connectors they use.

- Apache Kafka should be installed on all the machines, the brokers should be started on some servers, and Connect should be started on other servers

5.4 Running Kafka Connect Operating Modes

- **Standalone Mode –** run bin/connect-standalone.sh and pass it the properties file.

  ◇ all the connectors run on the one standalone worker.

  ◇ use this mode for development and troubleshooting purposes as well as in cases where connectors need to run on a specific machine

- **Distributed Mode -** run bin/connect-distributed.sh and pass it the properties file.

  ◇ It is meant for production use

- Connect worker is started by a script with a properties file:


```
bin/connect-distributed.sh config/connect-distributed.properties
```

5.5 Key Configurations for Connect workers:

- **bootstrap.servers -** A list of Kafka brokers that Connect will work with.

  - ◇ Connectors will pipe their data either to or from those brokers.

  - ◇ it's recommended to specify at least three.

- **group.id -** A Connect cluster contains all workers with the same group ID.

  - ◇ A connector started on the cluster will run on any worker.

- **key.converter and value.converter -** The two configurations set the converter for the key and value part of the message that will be stored in Kafka.

  - ◇ Connect can handle multiple data formats stored in Kafka.

  - ◇ The default is JSON format using the JSONConverter included in Apache Kafka.

  - ◇ Converter-specific configuration parameters are specifies for some converters. For example, JSON messages can include a schema or be schema-less.

- **rest.host.name and rest.port** - The REST API of Kafka Connect is typically used to configure and monitor Connectors. Both the host and port can be configured.

5.6 Kafka Connect API

- To verify the cluster and workers are up and running, use the following command to check the REST API:

```
$ curl http://localhost:8083/
```

```
The above command should return the current version you are running.
{"version":"0.10.1.0-SNAPSHOT","commit":"561f45d747cd2a8c"}
```

- To get the connector list, run the following command:

```
$ curl http://localhost:8083/connector-plugins
```

```
The connector plugins get displayed like this:
[{"class":"org.apache.kafka.connect.file.FileStreamSourceConnector"},
{"class":"org.apache.kafka.connect.file.FileStreamSinkConnector"}]
```

5.7 Kafka Connect Example – File Source

- Support to use a file as a source and another file as a destination is provided by the OOB connector plugin.

- JSON is the default data format used by Kafka.

- To start a file source, run the following command:

```
echo '{"name":"load-kafka-config", "config":{"connector.class":
"FileStreamSource","file":"sample-data.txt","topic":
"kafka-config-topic"}}' | curl -X POST -d @-
http://localhost:8083/connectors
--header "content-Type:application/json"
```

- Start Kafka Console consumer to check that we have loaded the sample-data.txt into a topic:

```
$ bin/kafka-console-consumer.sh --new-consumer --bootstrap-
server=localhost:9092 --topic kafka-config-topic –from-beginning
```

5.8 Kafka Connect Example – File Sink
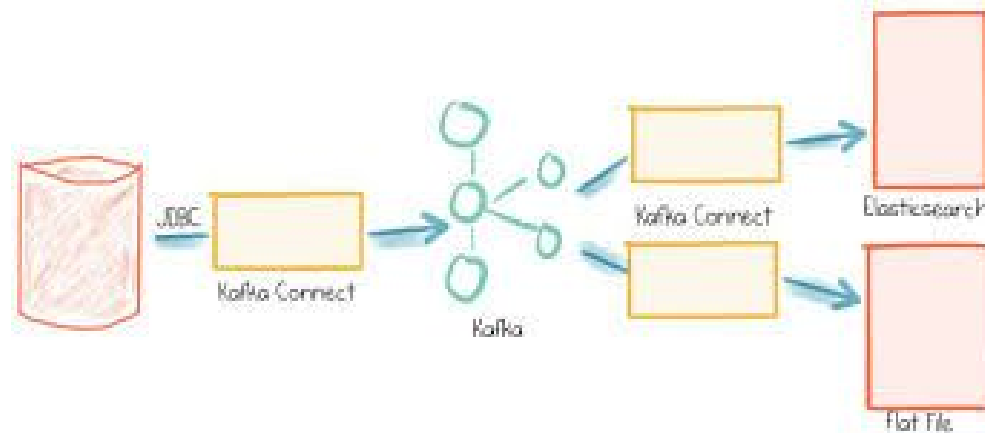
- To start the file sink connector, run the following command:

```
echo '{"name":"dump-kafka-config", "config":
{"connector.class":"FileStreamSink","file":"copy-of-sample-
data.txt","topics":"kafka-config-topic"}}' | curl -X POST -d @-
http://localhost:8083/connectors --header
"content-Type:application/json"
```

5.9 Summary

- Kafka can be integrated with existing open source frameworks, such as Storm, Hadoop, Spark, and ELK

- You can use various ready-made plugins or write your own product/consumer to integrate Kafka with any system.

5.10 Review Questions

⋄ What is the advantage/disadvantage of Kafka Connect?

⋄ What is file sink and file source used for in Kafka?

⋄ What formats are supported for storing data objects in Kafka?

Objectives
Key objectives of this chapter

Key objectives of this chapter

- Encryption and Authentication using SSL
- Configuring Kafka Brokers
- Authorization and ACLs
- Securing a Running Cluster
- ZooKeeper Authentication

6.1 Kafka Security

- When designed and developed at LinkedIn, security was kept out to a large extent.

- Security for Kafka was an afterthought after it became the main project at Apache.

- Later on in the year 2014 (v 0.9.0.0), various security discussions were considered for Kafka, especially data at rest security and transport layer security.

- Kafka broker allows clients to connect to multiple ports and each port supports a different security mechanism, such as:

  - No wire encryption and authentication

  - SSL: wire encryption and authentication

  - SASL: Kerberos authentication

  - SSL + SASL: SSL is for wire encryption and SASL for authentication

  - Authorization similar to Unix permissions for read/write by a client

6.2 Encryption and Authentication using SSL

- Apache Kafka allows clients and brokers to communicate over SSL using a dedicated port, although this is not enabled by default.

6.3 Authenticating Using SASL

Kafka Security – SASL

Client — Broker

Connection

Mechanism list

Selected Mechanism & sasl data

Evaluate and Response

Sasl data

Client Authenticated

Hortonworks

- Simple Authentication and Security Layer (SASL) is a framework for authentication and data security in Internet protocols.

- SASL decouples authentication mechanisms from application protocols, in theory allowing any authentication mechanism supported by SASL to be used in any application protocol that uses SASL.

- Authentication mechanisms can also support proxy authorization, a facility allowing one user to assume the identity of another.

- If your organization is already using a Kerberos server (for example, by using Active Directory), there is no need to install a new server just for Kafka.

6.4 Authorization and ACLs

- Kafka ships with a pluggable and out-of-box Authorizer that uses ZooKeeper to store all the ACLs.

- It is important to set ACLs because otherwise access to resources is limited to super users when an authorizer is configured.

6.5 Summary

- Kafka broker allows clients to connect to multiple ports and each port supports a different security mechanism, such as:

  ◇ No wire encryption and authentication

  ◇ SSL: wire encryption and authentication

  ◇ SASL: Kerberos authentication

  ◇ SSL + SASL: SSL is for wire encryption and SASL for authentication

Chapter 7 - Monitoring Kafka

Objectives
Key objectives of this chapter
• Metrics basics
• Broker metrics
• JVM monitoring
• Producer metrics
• Consumer metrics

7.1 Introduction

- The Apache Kafka applications have numerous measurements for their operation, such as:

    ◊ the overall rate of traffic,

    ◊ timing metrics for every request type

    ◊ per-topic and per-partition metrics.

- They provide a detailed view of every operation in the broker

- Certain metrics are useful when debugging problems.

7.2 Metrics Basics

- The easiest way to access them in an external monitoring system is to use a collection agent provided by your monitoring system and attach it to the Kafka process. For example:
  - ◊ Nagios check_jmx plugin
  - ◊ jmxtrans
  - ◊ MX4J.

7.3 OS Monitoring

- OS should also be monitored.

- The main areas that are necessary to watch are CPU usage, memory usage, disk usage, disk IO, and network usage.

- The Kafka broker uses a significant amount of processing for handling requests.

- For CPU utilization, you will want to look at the system load average at the very least.

7.4 Kafka Broker Metrics

- Most broker metrics are low level

- The metrics provide information about nearly every function within the broker, but the most common ones provide the information needed to run Kafka on a daily basis.

- There are various broker metrics:

  ◊ under replicated partitions

  ◊ all topics bytes in

  ◊ all topics bytes out

  ◊ all topics messages in

  ◊ partition count

  ◊ leader count

  ◊ offline partitions

  ◊ request metrics

7.5 Under-Replicated Partitions

- If there is only one metric that you are able to monitor from the Kafka broker, it should be the number of under-replicated partitions.

- This measurement, provided on each broker in a cluster, gives a count of the number of partitions for which the broker is the leader replica, where the follower replicas are not caught up.

- This single measurement provides insight into a number of problems with the Kafka cluster, from a broker being down to resource exhaustion.

- It indicates that one of the brokers in the cluster is offline.

- The count of under-replicated partitions across the entire cluster will equal the number of partitions that are assigned to that broker, and the broker that is down will not report a metric.

7.6 All topics bytes in

- The all topics bytes in rate, expressed in bytes per second, is useful as a measurement of how much message traffic your brokers are receiving from producing clients.

- This is a good metric to trend over time to help you determine when you need to expand the cluster or do other growth-related work.

- It is also useful for evaluating if one broker in a cluster is receiving more traffic than the others, which would indicate that it is necessary to rebalance the partitions in the cluster.

7.7 All topics bytes out

- The all topics bytes out rate, similar to the bytes in rate, is another overall growth metric.

- In this case, the bytes out rate show the rate at which consumers are reading messages out.

- The outbound bytes rate may scale differently than the inbound bytes rate

7.8 All topics messages in

- The messages in rate shows the number of individual messages, regardless of their size, produced per second.

- This is useful as a growth metric as a different measure of producer traffic.

- It can also be used in conjunction with the bytes in rate to determine an average message size.

- You may also see an imbalance in the brokers, just like with the bytes in rate, that will alert you to maintenance work that is needed.

7.9 Partition count

- It is the total number of partitions assigned to that broker.

- This includes every replica the broker has, regardless of whether it is a leader or follower for that partition.

7.10 Leader count

- The leader count metric shows the number of partitions that the broker is currently the leader for.

- It should be generally even across the brokers in the cluster.

- It is much more important to check the leader count on a regular basis, possibly alerting on it, as it will indicate when the cluster is imbalanced even if the number of replicas is perfectly balanced in count and size across the cluster.

7.11 Offline partitions

- Along with the under-replicated partitions count, the offline partitions count is a critical metric for monitoring

- This measurement is only provided by the broker that is the controller for the cluster (all other brokers will report 0) and shows the number of partitions in the cluster that currently have no leader.

- Partitions without leaders can happen for two main reasons:

  ◊ All brokers hosting replicas for this partition are down

  ◊ No in-sync replica can take leadership due to message-count mismatches (with unclean leader election disabled).

7.12 Request Metrics (Contd.)

- The time metrics each provide a set of percentiles for requests, as well as a discrete Count attribute.

- The metrics are all calculated since the broker was started

- The parts of request processing they represent are:

  ◊ **Total time** - Measures the total amount of time the broker spends processing the request, from receiving it to sending the response back to the requestor.

  ◊ **Request queue time** - The amount of time the request spends in a queue after it has been received but before processing starts.

  ◊ **Local time** - The amount of time the partition leader spends processing a request, including sending it to disk (but not necessarily flushing it).

  ◊ **Remote time** - The amount of time spent waiting for the followers before request processing can complete.

  ◊ **Throttle time** - The amount of time the response must be held in order to slow the requestor down to satisfy client quota settings.

  ◊ **Response queue time** - The amount of time the response to the request spends in the queue before it can be sent to the requestor.

  ◊ **Response send time** - The amount of time spent actually sending the response.

7.13 Client Monitoring

- All applications need monitoring.

- Those that instantiate a Kafka client, either a producer or consumer, have metrics specific to the client that should be captured.

- The Kafka producer client metrics are available as attributes on a small number of mbeans.

7.14 Overall producer metrics

- The overall producer metrics bean provides attributes describing everything from the sizes of the message batches to the memory buffer utilization.

- **record-error-rate** – it is one attribute that you will want to set an alert for.

  ◊ This metric should always be zero, and if it is anything greater than that, the producer is dropping messages it is trying to send to the Kafka brokers.

- **record-retry-rate attribute** – it can also be tracked, but it is less critical than the error rate because retries are normal.

- **request-latency-avg** – another attribute you will want to set an alert for.

  ◊ This is the average amount of time a produce request sent to the brokers takes.

  ◊ You should be able to establish a baseline value for what this number should be in normal operations, and set an alert threshold above that.

- **outgoing-byte-rate** - it describes the messages in absolute size in bytes per second.

7.15 Overall producer metrics (Contd.)

- **record-send-rate** – it describes the traffic in terms of the number of messages produced per second.

- **request-rate** – it provides the number of produce requests sent to the brokers per second.

- **request-size-avg metric** – it provides the average size of the produce requests being sent to the brokers in bytes.

- **batch-size-avg** – it provides the average size of a single message batch (which, by definition, is comprised of messages for a single topic partition) in bytes.

- **record-size-avg** – it shows the average size of a single record in bytes.

- **records-per-request-avg** – this metric describes the average number of messages that are in a single produce request.

- **record-queue-time-avg** – it is the average amount of time, in milliseconds, that a single message waits in the producer after the application sends it before it is actually produced to Kafka.

7.16 Consumer Metrics

- The consumer in Kafka consolidates many of the metrics into attributes on just a few metrics.

- The overall consumer bean has metrics regarding the lower-level network operations

7.17 Per-broker and per-topic metrics

- The metrics that are provided by the consumer client for each of the broker connections and each of the topics being consumed are useful for debugging issues with consumption

- The incoming-byte-rate and request-rate metrics break down the consumed message metrics provided by the fetch manager into per-broker bytes per second and requests per second measurements, respectively.

7.18 Consumer coordinator metrics

- Consumer clients generally work together as part of a consumer group.

- This group has coordination activities, such as group members joining and heartbeat messages to brokers to maintain group membership.

- The consumer coordinator is the part of the consumer client that is responsible for handling this work, and it maintains its own set of metrics.

- The biggest problem that consumers can run into due to coordinator activities is a pause in consumption while the consumer group synchronizes.

- The coordinator provides the metric attribute sync-time-avg, which is the average amount of time, in milliseconds, that the sync activity takes.

- It is also useful to capture the sync-rate attribute, which is the number of group syncs that happen every second.

- For a stable consumer group, this number should be zero most of the time.

7.19 Quotas

- Apache Kafka has the ability to throttle client requests in order to prevent one client from overwhelming the entire cluster.

- This is configurable for both producer and consumer clients and is expressed in terms of the permitted amount of traffic from an individual client ID to an individual broker in bytes per second.

- There is a broker configuration, which sets a default value for all clients, as well as per-client overrides that can be dynamically set.

- When the broker calculates that a client has exceeded its quota, it slows the client down by holding the response back to the client for enough time to keep the client under the quota.

7.20 Lag Monitoring

- For Kafka consumers, one of the most important things to monitor is the consumer lag.

- Measured in number of messages, this is the difference between the last message produced in a specific partition and the last message processed by the consumer.

- The records-lag-max metric from the consumer client can provide a partial view of the consumer status.

- External monitoring is better than what is available from the client itself because if the consumer is broken or offline, the metric is either inaccurate or not available.

- An external process should watch both the state of the partition on the broker, tracking the offset of the most recently produced message, and the state of the consumer, tracking the last offset the consumer group has committed for the partition.

7.21 Summary

- In this chapter, we covered the basics of how to monitor the Kafka applications.

- Monitoring is a key aspect of running Apache Kafka properly

Chapter 8 - Apache Kafka Best Practices

Objectives

Key objectives of this chapter

- Working with Partitions
- Working with Consumers
- Working with Producers
- Working with Brokers

8.1 Best Practices for Working with Partitions

- Understand the data rate of your partitions to ensure you have the correct retention space.

  ◇ The data rate of a partition is the average message size times the number of messages per second.

  ◇ The data rate dictates how much retention space, in bytes, is needed to guarantee retention for a given amount of time.

  ◇ The data rate also specifies the minimum performance a single consumer needs to support without lagging.

8.2 Best Practices for Working with Partitions (Contd.)

- Unless you have architectural needs that require you to do otherwise, use random partitioning when writing to topics.

  - When you're operating at scale, uneven data rates among partitions can be difficult to manage.

  - There are three main reasons for this:

    - Consumers of the "hot" (higher throughput) partitions will have to process more messages than other consumers in the consumer group, potentially leading to processing and networking bottlenecks.

    - Topic retention must be sized for the partition with the highest data rate, which can result in increased disk usage across other partitions in the topic.

    - Attaining an optimum balance in terms of partition leadership is more complex than simply spreading the leadership across all brokers. A "hot" partition might carry 10 times the weight of another partition in the same topic.

8.3 Best Practices for Working with Consumers

- Tune your consumer socket buffers for high-speed ingest.

  - In Kafka, the parameter is **receive.buffer.bytes**, which defaults to 64kB.

  - The default value is too small for high-throughput environments, particularly if the network's bandwidth-delay product between the broker and the consumer is larger than a local area network (LAN).

  - For high-bandwidth networks (10 Gbps or higher) with latencies of 1 millisecond or more, consider setting the socket buffers to 8 or 16 MB.

  - If memory is scarce, consider 1 MB.

  - You can also use a value of -1, which lets the underlying operating system tune the buffer size based on network conditions.

  - The automatic tuning might not occur fast enough for consumers that need to start "hot."

8.4 Best Practices for Working with Consumers (Contd.)

- Design high-throughput consumers to implement back-pressure when warranted

  - It is better to consume only what you can process efficiently

  - Consumers should consume into fixed-sized buffers

8.5 Best Practices for Working with Producers

- Configure your producer to wait for acknowledgments

  ◇ It lets the producer know that the message has actually made it to the partition on the broker.

  ◇ In Kafka, the setting is acks (without acks, it's known as. "fire and forget")

  ◇ Kafka provides fault-tolerance via replication so the failure of a single node or a change in partition leadership does not affect availability.

- Configure retries on your producers.

  ◇ The default value is 3, which is often too low.

  ◇ The right value will depend on your application

  ◇ For applications where data-loss cannot be tolerated, consider Integer.MAX_VALUE (infinity).

  ◇ This guards against situations where the broker leading the partition isn't able to respond to a produce request right away.

8.6 Best Practices for Working with Producers (Contd.)

- For high-throughput producers, tune buffer sizes

  - buffer.memory and batch.size (which is counted in bytes).

  - Because batch.size is a per-partition setting, producer performance and memory usage can be correlated with the number of partitions in the topic.

- Instrument your application to track metrics

  - e.g. the number of produced messages, average produced message size, and the number of consumed messages.

8.7 Best Practices for Working with Brokers

- Compacted topics require memory and CPU resources on your brokers.

    ◇ Log compaction needs both heap (memory) and CPU cycles on the brokers to complete successfully

    ◇ Failed log compaction puts brokers at risk from a partition that grows unbounded.

8.8 Best Practices for Working with Brokers (Contd.)

- Distribute partition leadership among brokers in the cluster.

  ◇ Leadership requires a lot of network I/O resources.

- Don't neglect to monitor your brokers for in-sync replica (ISR) shrinks, under-replicated partitions, and unpreferred leaders.

  ◇ These are signs of potential problems in your cluster.

  ◇ Frequent ISR shrinks for a single partition can indicate that the data rate for that partition exceeds the leader's ability to service the consumer and replica threads.

8.9 Best Practices for Working with Brokers (Contd.)

- Either disable automatic topic creation or establish a clear policy regarding the cleanup of unused topics.

  ◇ If no messages are seen for x days, consider the topic defunct and remove it from the cluster to avoid the creation of additional metadata within the cluster that you'll have to manage.

- For sustained, high-throughput brokers, provision sufficient memory to avoid reading from the disk subsystem.

  ◇ Partition data should be served directly from the operating system's file system cache whenever possible.

  ◇ Ensure your consumers can keep up; a lagging consumer will force the broker to read from disk.

8.10 Best Practices for Working with Brokers (Contd.)

- For a large cluster with high-throughput service level objectives (SLOs), consider isolating topics to a subset of brokers.

  ◇ If you have multiple online transaction processing (OLTP) systems using the same cluster, isolating the topics for each system to distinct subsets of brokers can help to limit the potential blast radius of an incident.

- Using older clients with newer topic message formats, and vice versa, places extra load on the brokers

  ◇ they convert the formats on behalf of the client which slows down the performance.

- Don't assume that testing a broker on a local desktop machine is representative of the performance you'll see in production.

  ◇ Testing over a loopback interface to a partition using replication factor 1 is a very different topology from most production environments.

  ◇ The network latency is negligible via the loopback and the time required to receive leader acknowledgments can vary greatly when there is no replication involved.

8.11 Summary

- In this chapter, we discussed the practices for working with Kafka partitions, consumers, producers, and brokers