Slide source: 2018.9.14 Kafka Tutorial by Erhwen Kuo

數據串流平臺：Kafka

授課講師 蔡昀翰

教材編寫 郭二文原著、蔡昀翰編修

緯育 TibaMe

課程大綱

- Module 1：Introduction to Kafka and the ecosystem
- Module 2：Basic Concepts of Apache Kafka
- Module 3：Hands-on Practice of Apache Kafka
- Module 4：Basic Topic Configurations
- Module 5：Log Cleanup Policies
- Module 6：Guidelines of Topic Configurations
- Module 7：Hands-on Practice of Topic Configurations
- Module 8：Broker Configurations Example
- Module 9：Kafka Producers
- Module 10：Consumer Groups
- Module 11：Rebalance
- Module 12：Kafka Consumers

授課講師介紹

蔡昀翰

專長

Data Pipeline Development in Big Data.

簡歷

Inno Technology Co., Ltd - Data Engineer

老師的話

大家加油。

聯絡方式

urmfriend71@gmail.com

學習本課程須知

先備知識

https://tinyurl.com/vqx7s5d

學習目標 (1)

A. 了解 Kafka Ecosystem 及基礎操作。

B. Kafka 參數設定

C. Kafka 生產者與消費者應用

學習方式

A. 依課堂講授內容進行研讀

B. 依課堂 Hands-On 操作

須完成哪些作業或考試

A. 末節課堂完成 25 題測驗

一起修煉吧！

Slide source: 2018.9.14 Kafka Tutorial by Erhwen Kuo

學習目標： A

Module 1：Introduction to Kafka and the Ecosystem

1-1：Why Apache Kafka

1-2：Use cases of Apache Kafka

1-3：Kafka ecosystem
Architecture & Administration
Monitoring Tools

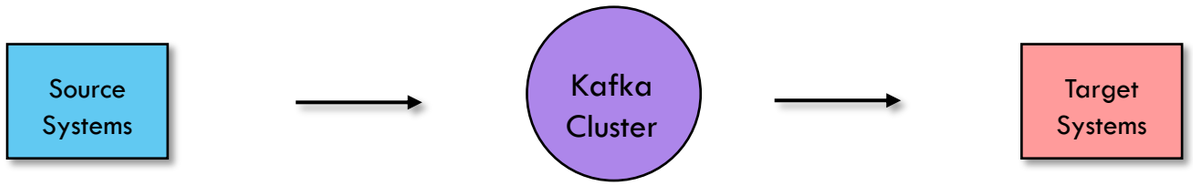
Course Objectives

- What is Kafka
- Learn about the Kafka ecosystem (high level)
- Learn Apache Kafka core API
  - Topics, Partitions
  - Brokers, Replication, Zookeeper
  - Producers, Consumers, Consumer Groups
- Get started with a broker setup on own machine using Docker-Compose

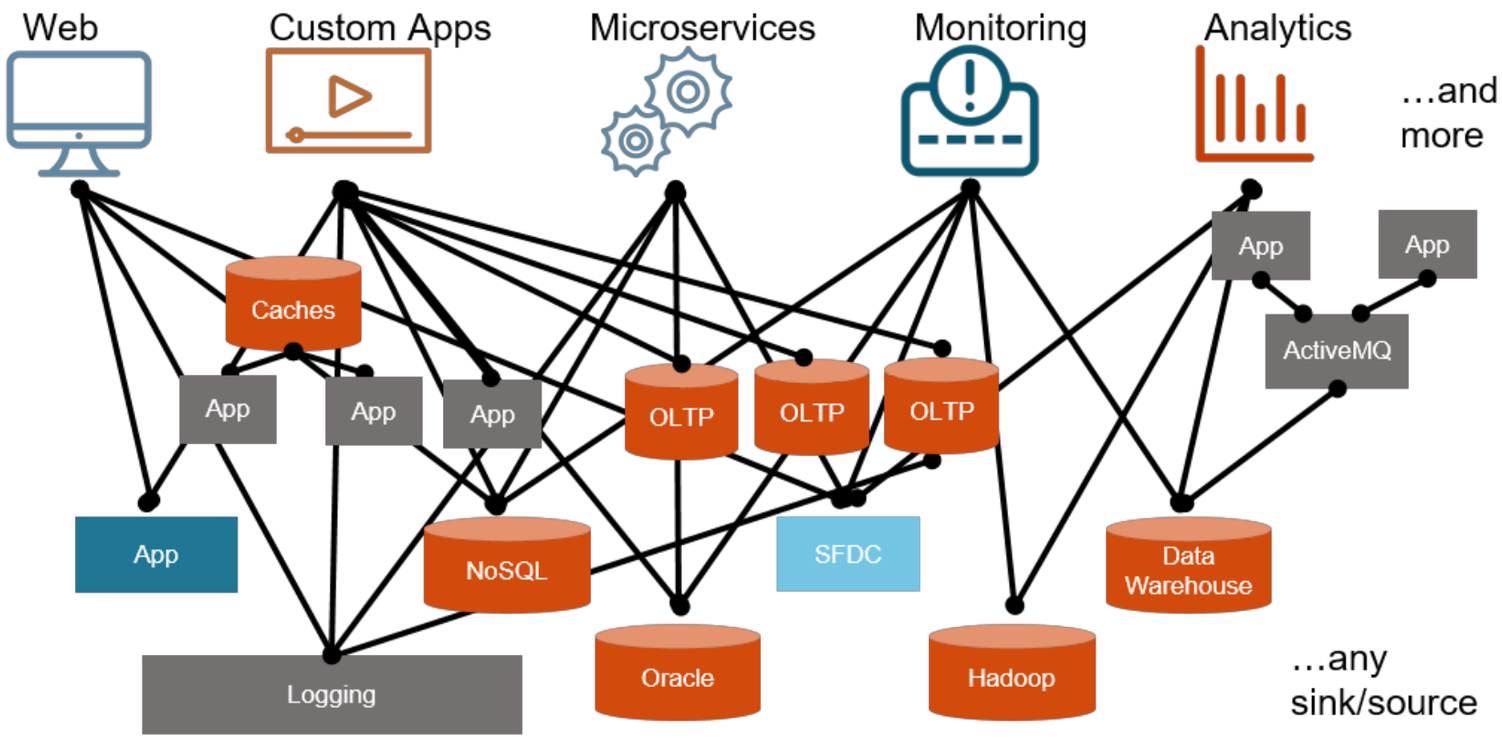
What is Apache Kafka

Kafka was developed at LinkedIn back in 2010

- Apache Kafka was originally developed by LinkedIn, and open sourced in early 2011.
- Written in Scala and Java.
- The project aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds

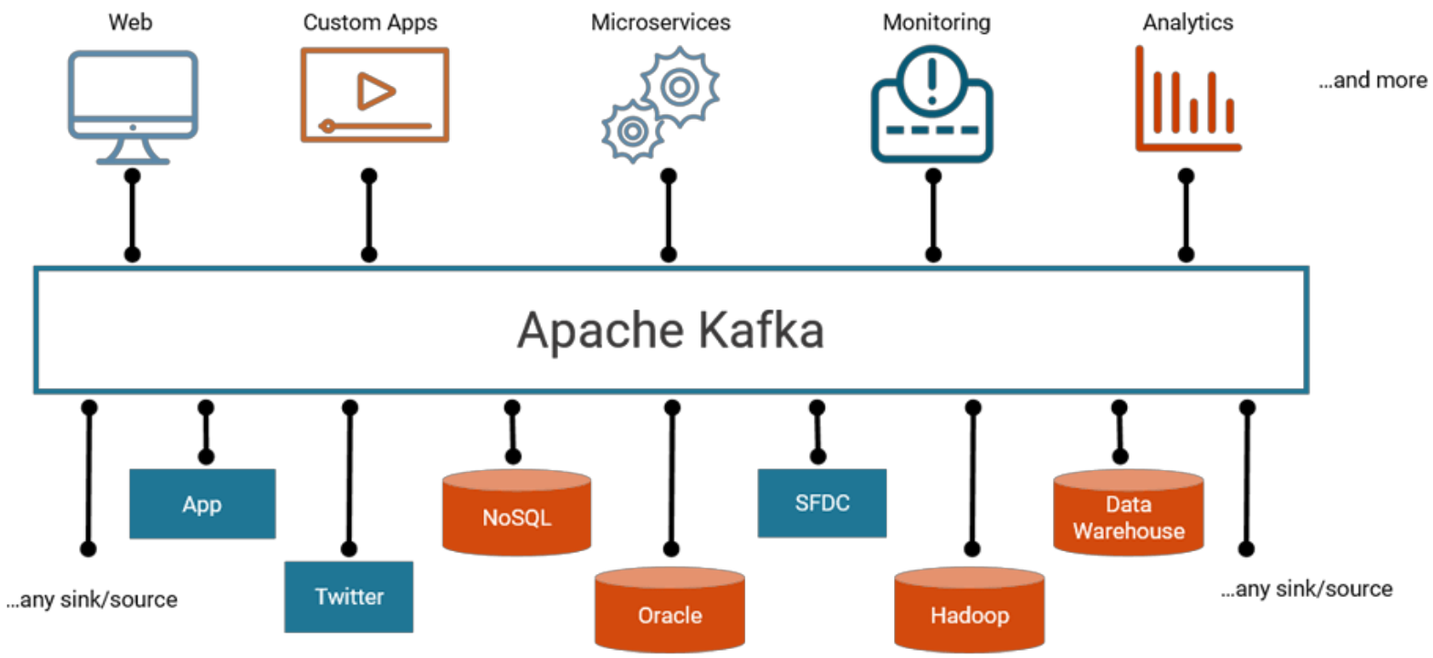


Why Apache Kafka
Integration become very complicate



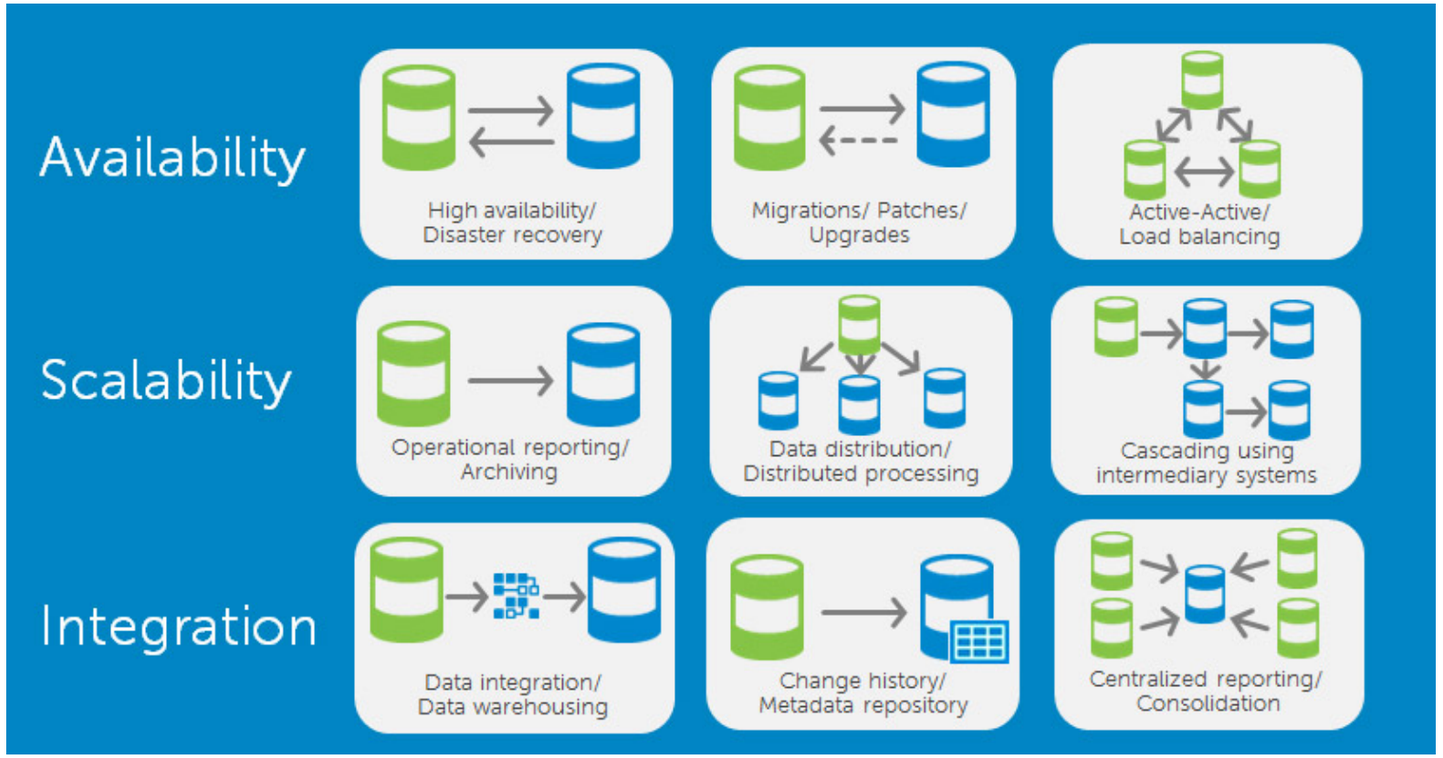
Ref# 4

Why Apache Kafka:
Decoupling of data streams

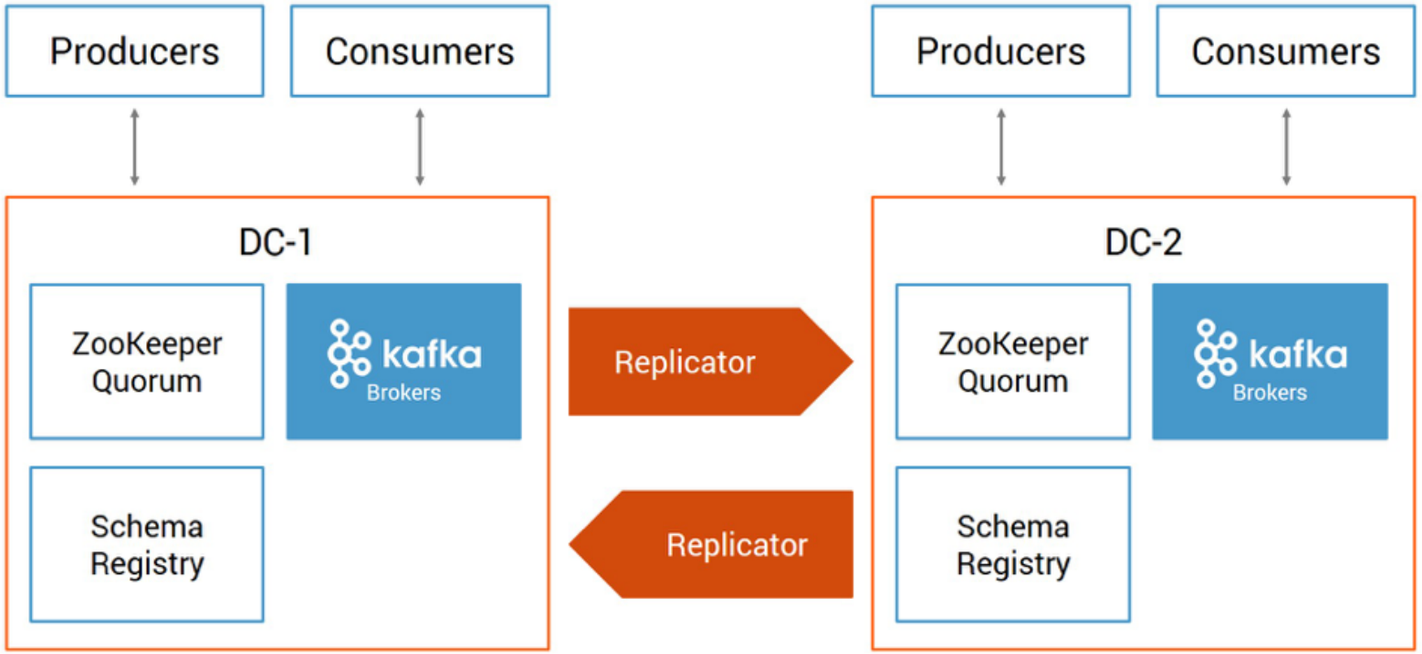


Ref# 4

Why Apache Kafka
Data is very important



Why Apache Kafka:
Decoupling of data location



Ref# 4

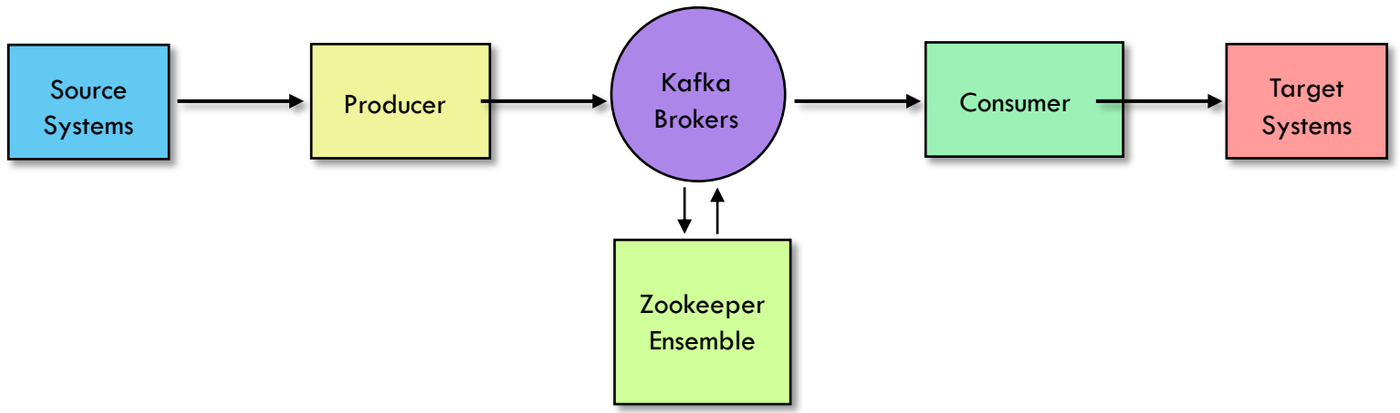
Why Apache Kafka

- Distributed, resilient architecture, fault tolerant
- Horizontal scalability
- High performance (latency of less than 10 ms) – real time
- Used by the 2000+ firms:
  - LinkedIn
  - Netflix
  - AirBnb
  - Yahoo
  - Walmart

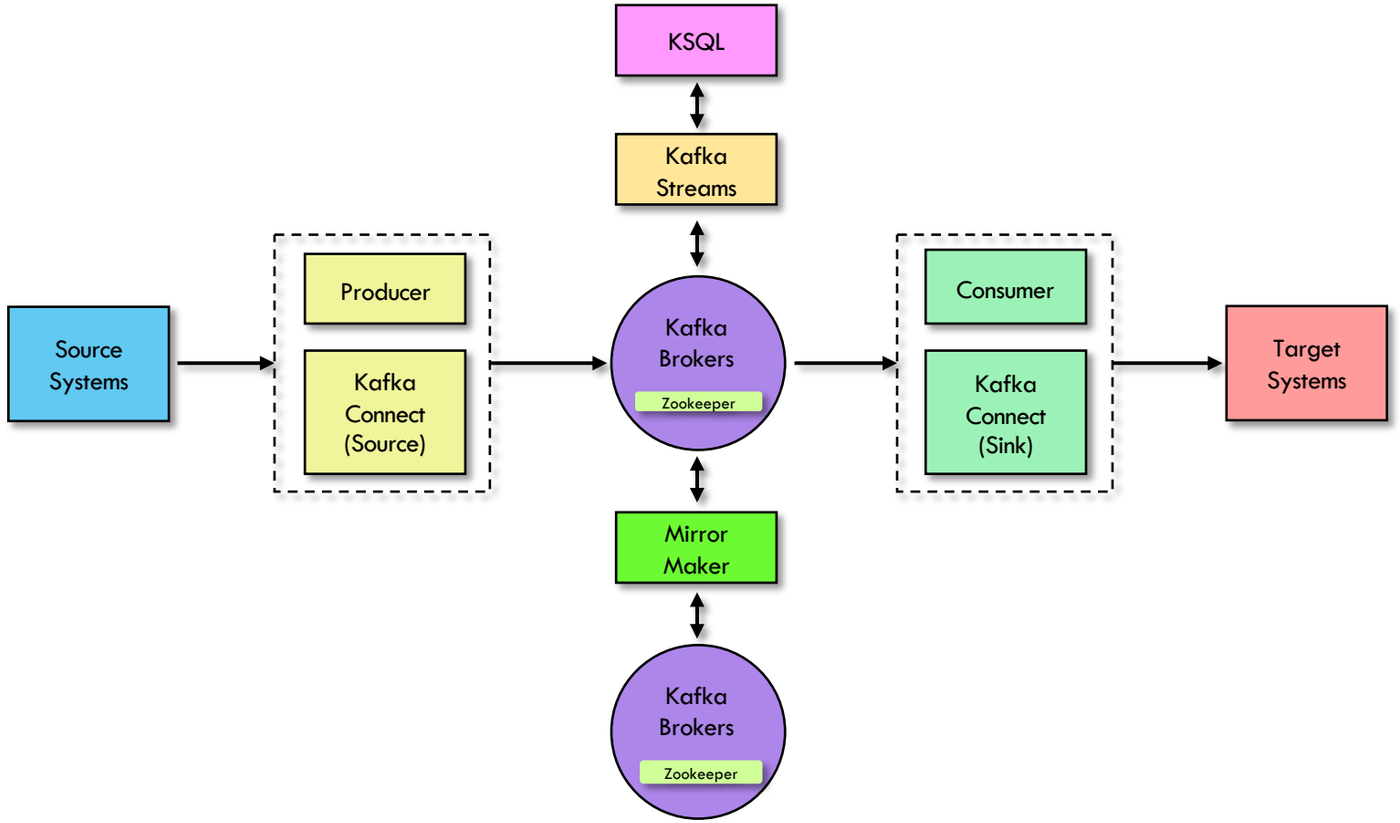
Apache Kafka: Use cases

- Messaging System
- Activity Tracking
- Gather metrics from many different locations (IoT / Industry 4.0)
- Application Logs gathering (Industry 4.0)
- Stream processing (with the Kafka Streams API or Spark for example)
- De-coupling of system dependencies
- Integration with Spark, Flink, Storm, Hadoop, and many other Big Data technologies

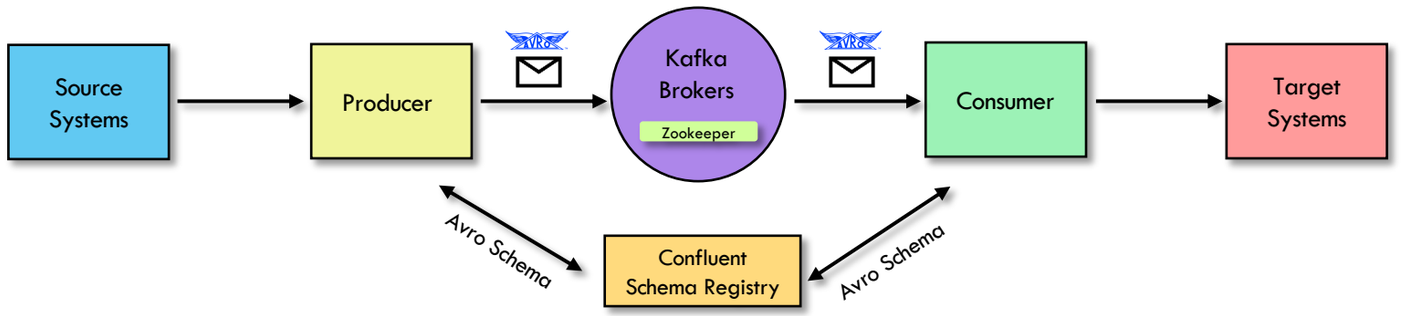
Kafka Ecosystem:
Kafka Core



Kafka Ecosystem:
Kafka Extended API



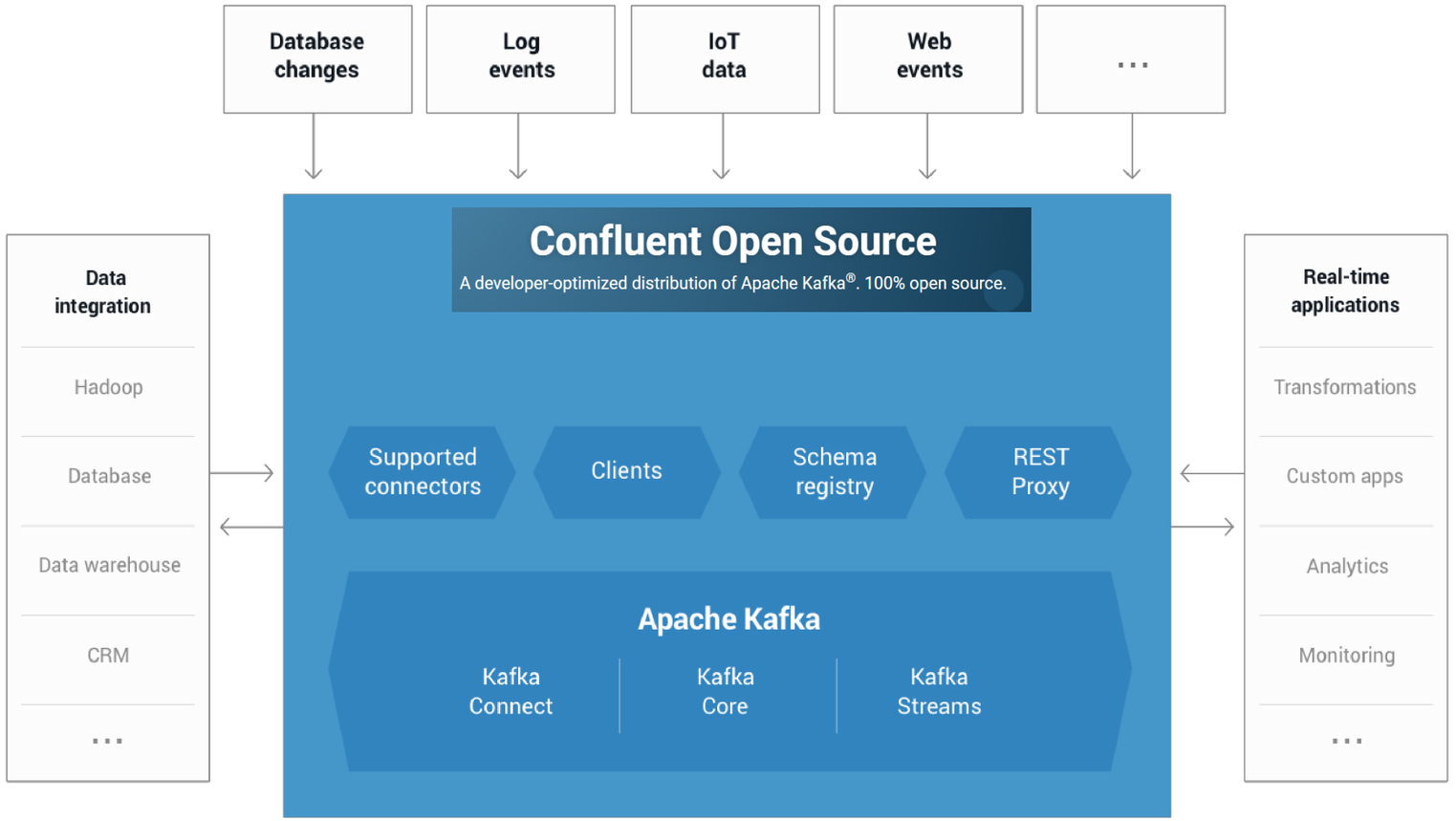
Kafka Ecosystem:
Confluent Components - Schema Registry



Kafka Ecosystem：
Administration and Monitoring Tools

- Topics UI (Landoop): View the topics content
- Schema UI (Landoop): Explore the Schema registry
- Connect UI (Landoop): Create and monitor Connect tasks
- Kafka Manager (Yahoo): Overall Kafka Cluster Manager
- Burrow (LinkedIn): Kafka Consumer Lag Checking
- Exhibitor (Netflix): Zookeeper Configuration, Monitoring, Backup
- Kafka Tools (LinkedIn): Broker and topics admin tasks simplified
- Kafkat (Airbnb): More broker and topics admin tasks simplified
- JMX Dump: Dump JMX metrics from Brokers

Kafka in the Enterprise Architecture
Confluent Open Source Version



Ref#4

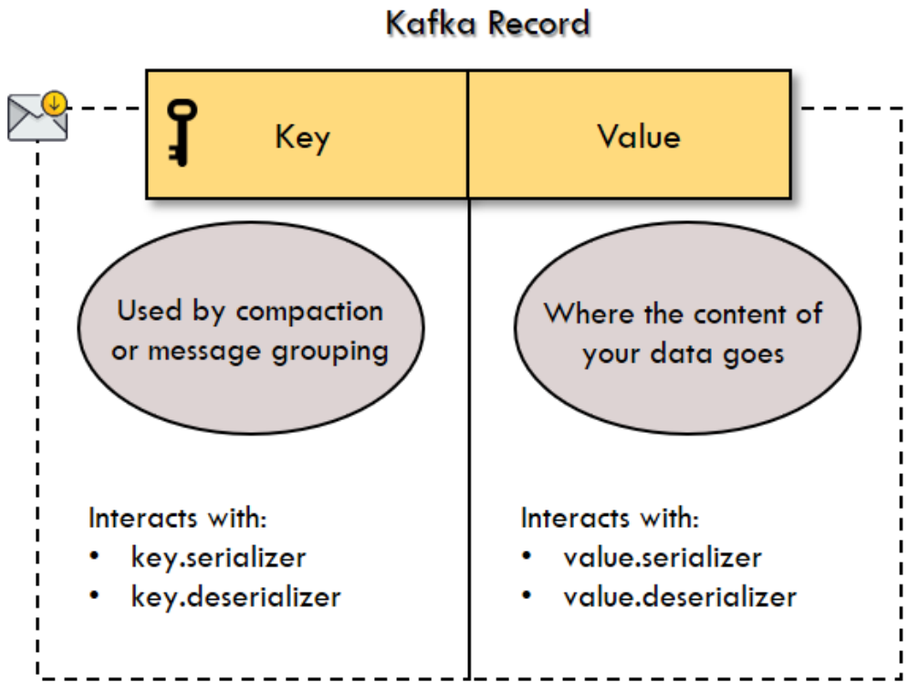
學習目標： A

Module 2：Basic Concepts of Apache Kafka

2-1：Topics and partitions
2-2：Broker
2-3：Producer & Consumer

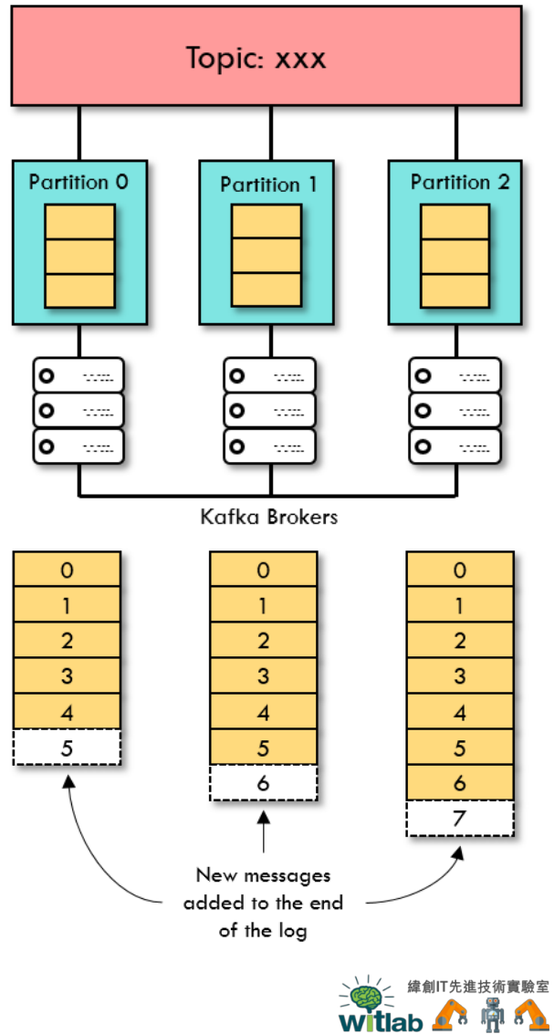
Kafka Record

- Every message publish to Kafka called “**Record**”
- **Record** contain two parts:
  - **Key**
    - Used by compaction or for message grouping
  - **Value**
    - The content of data goes



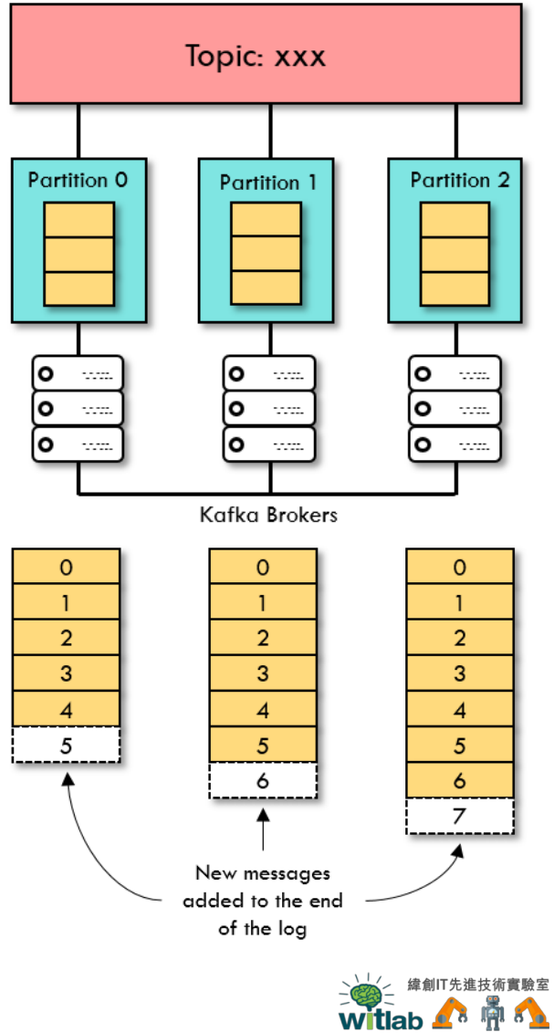
Topics and partitions

- **Topics**: a particular stream of data
  - Similar to a table in a database (without all the constraints)
  - You can have as many topics as you want
  - A topic is identified by its **name**
- Topics are split in **partitions**
  - Each partition is ordered
  - Each message within a partition gets an incremental id, called **offset**



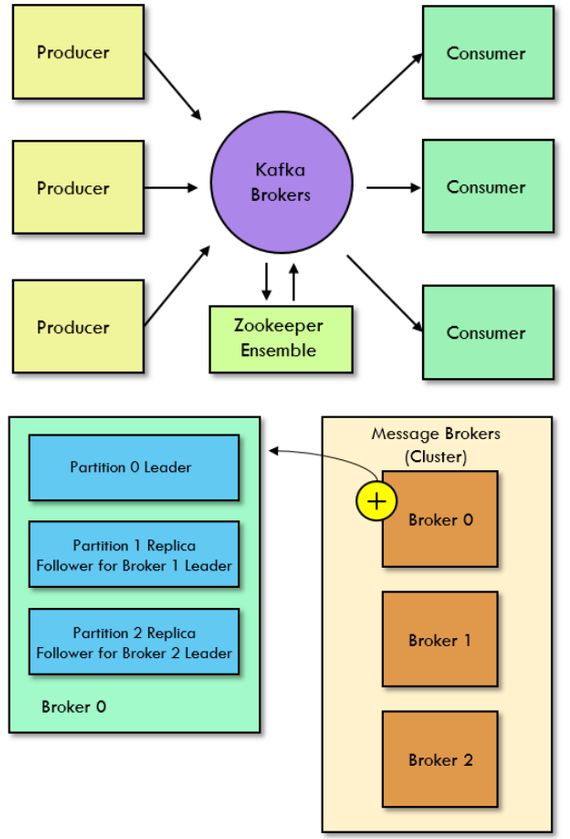
Topics and partitions

- **Offset** only have a meaning for a specific partition.
  - E.g. offset 3 in partition doesn't represent the same data as offset 3 in another partition
- Order is guaranteed only within a partition (not across partitions)
- Data is kept only for a limited time (default is **one** weeks)
- Once the data is written to a partition, it can't be changed (immutability)
- Data is assigned randomly to a partition unless a **key** is provided (more on this later)
- You can have as many partitions per topics as you want)



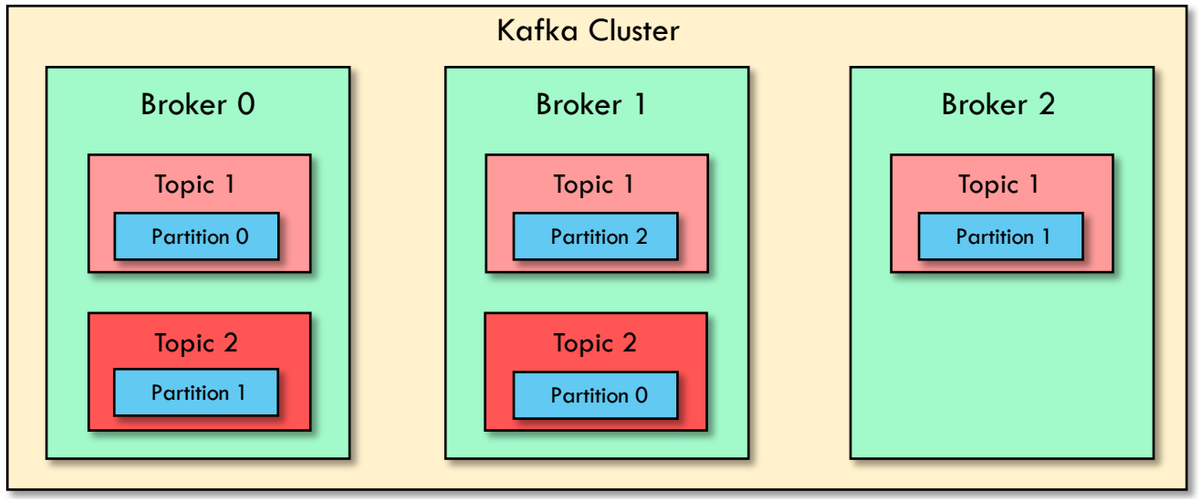
Brokers

- A Kafka cluster is composed of multiple **brokers** (servers)
- Each broker is identified with its ID (integer)
- Each broker contains certain topic partitions
- After connecting to any broker (called a bootstrap broker), you will be connected to the entire cluster
- A good number to get started is **3** brokers, but some big clusters have over 100 brokers



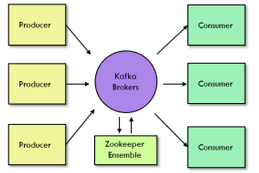
Brokers and topics

- Example of 2 topics (3 partitions and 2 partitions)

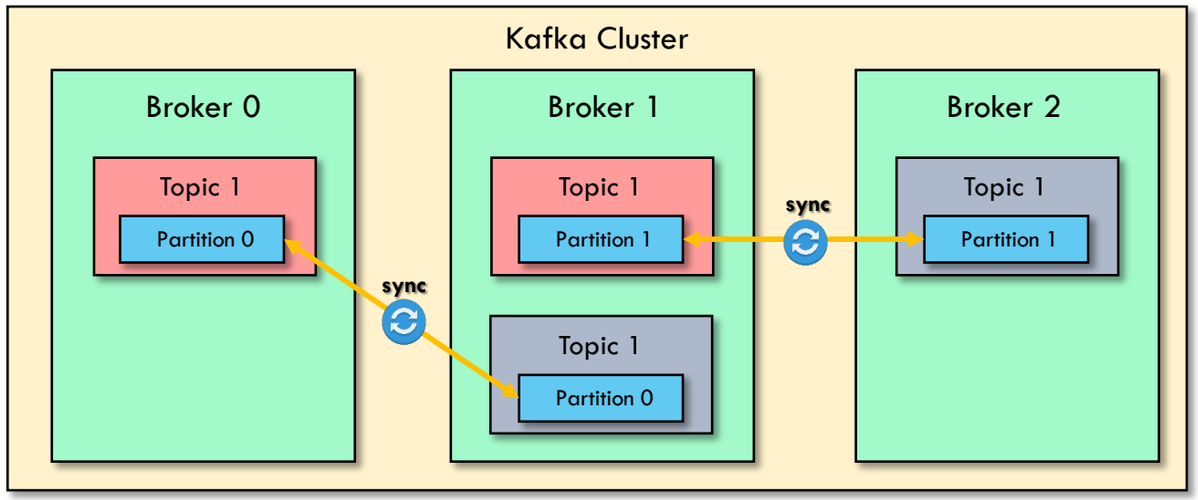


- Data is distributed and Broker 2 doesn't have any Topic 2 data

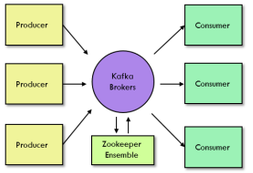
Topic replication factor



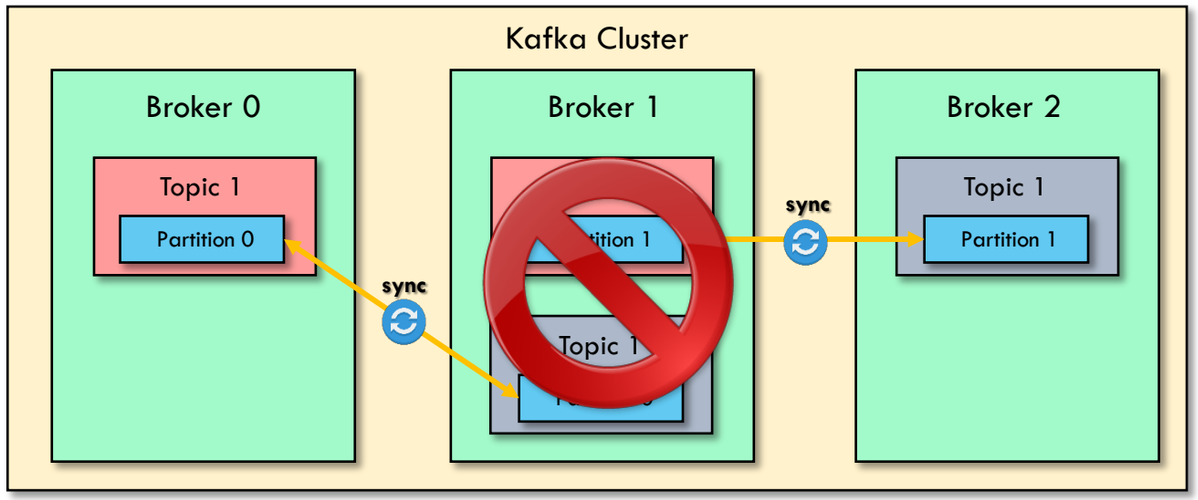
- Topics should have a replication factor > 1 (usually between 2 and 3)
- This way if a broker is down, another broker can serve the data
- Example: 1 **topic** with 2 **partitions** and **replication factor** of 2



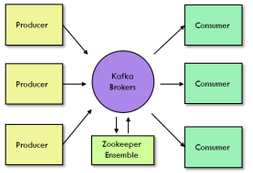
Topic replication factor



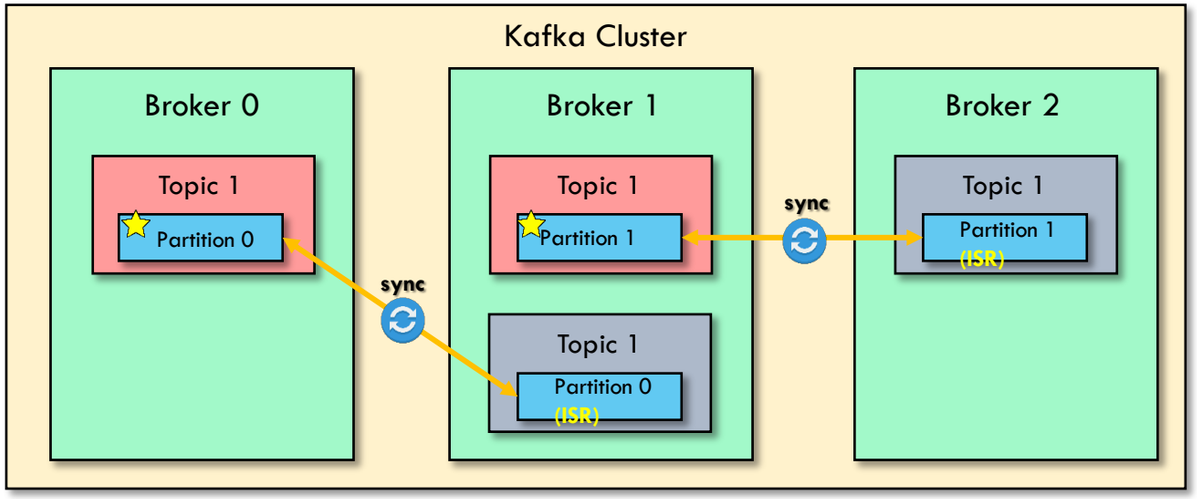
- Example: we lost Broker 1
- Result: Broker 0 and 2 can still serve the data



Concept of Leader for a partition

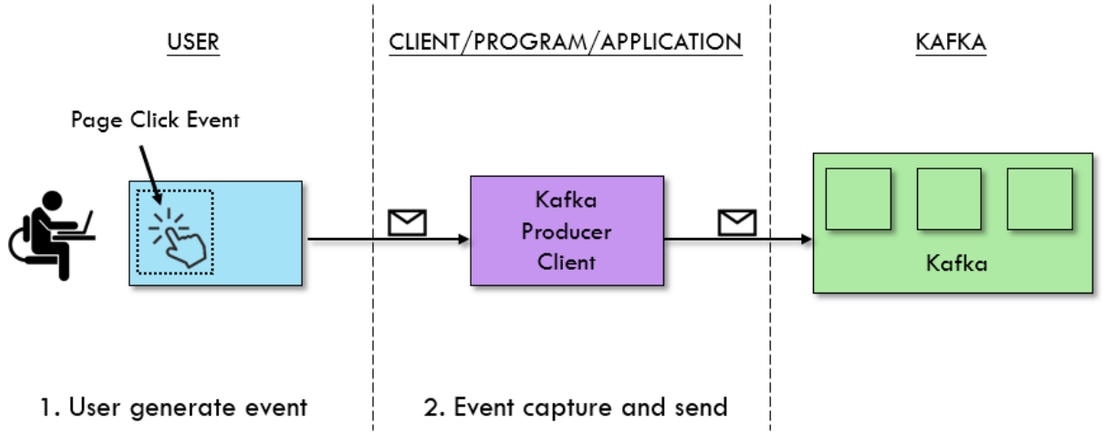


- At any time only 1 broker can be a leader for a given partition
- Only that leader can retrieve and serve data for a partition
- The other brokers will synchronize the data
- There each partition has: one **leader**, and multiple **ISR** (in-sync replica)



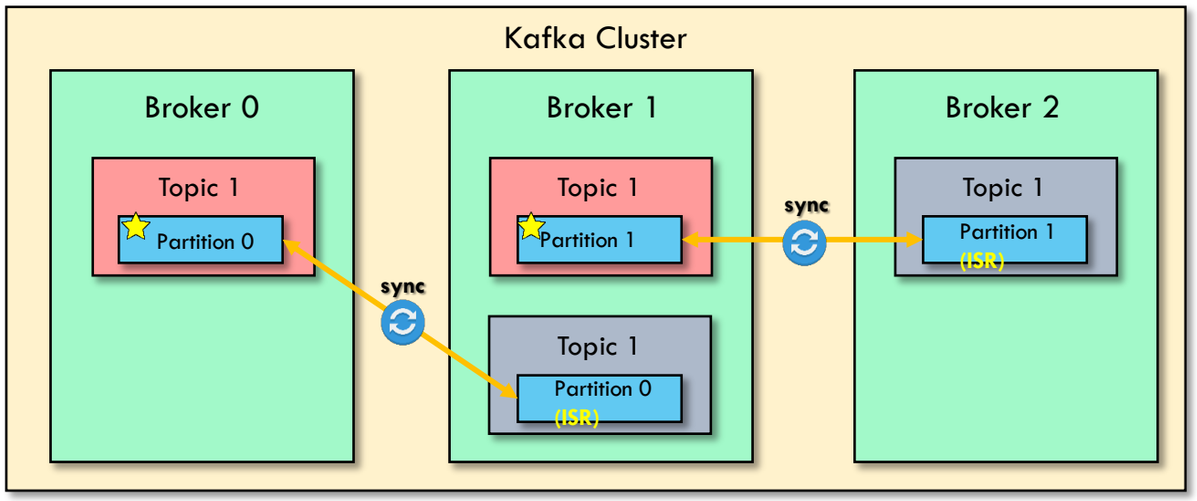
Producers

- **Producers** write data to **topics**.
- They only have to specify the topic name and one broker to connect to, and Kafka will automatically take care of routing the data to the right brokers.



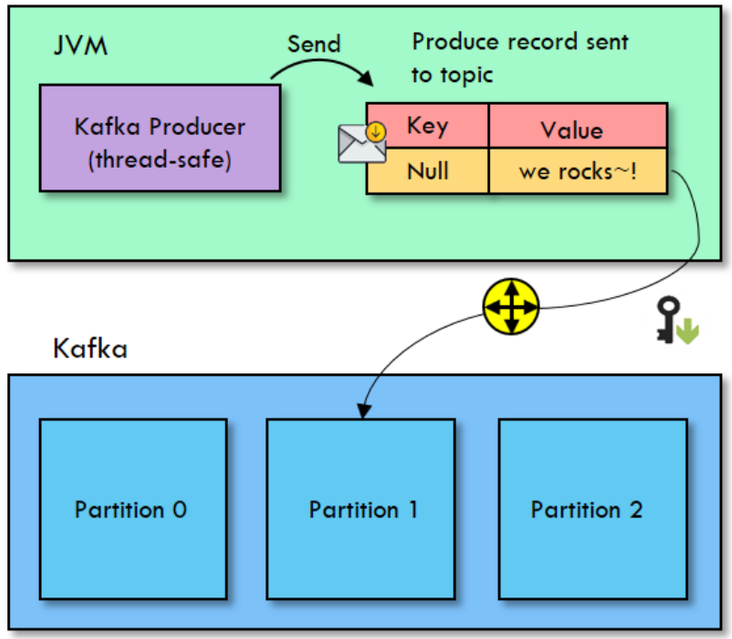
Producers: Broker acknowledgement

- Producer can choose to receive acknowledgement of data writes:
  - **acks** = 0 : Producer won't wait for acknowledgement (possible data loss)
  - **acks** = 1 : Producer will wait for leader acknowledgement (limited data loss)
  - **acks** = all (-1) : Leader + replicas acknowledgment (no data loss)

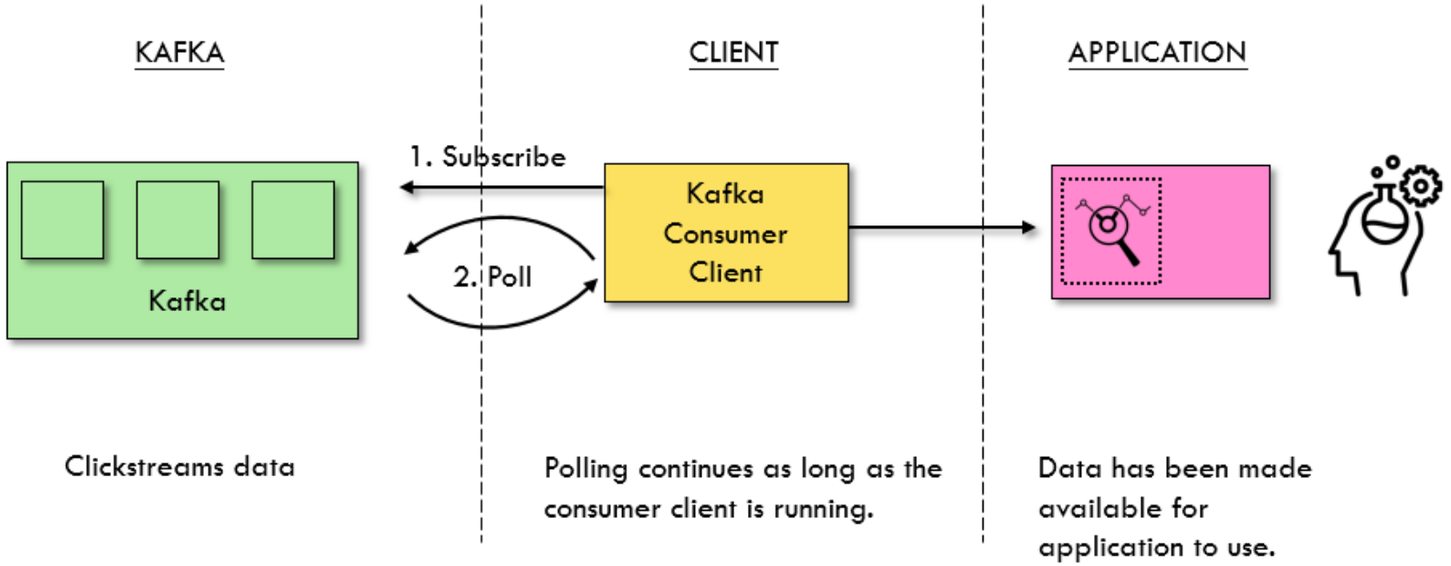


Producers: Message keys

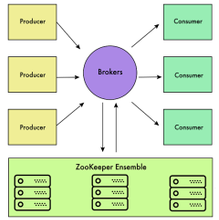
- Producers can choose to send a key with the message
- If a **key** is sent, then the producer has the guarantee that all messages for that key will always go to the same partition
- This enables to guarantee ordering for a specific **key**



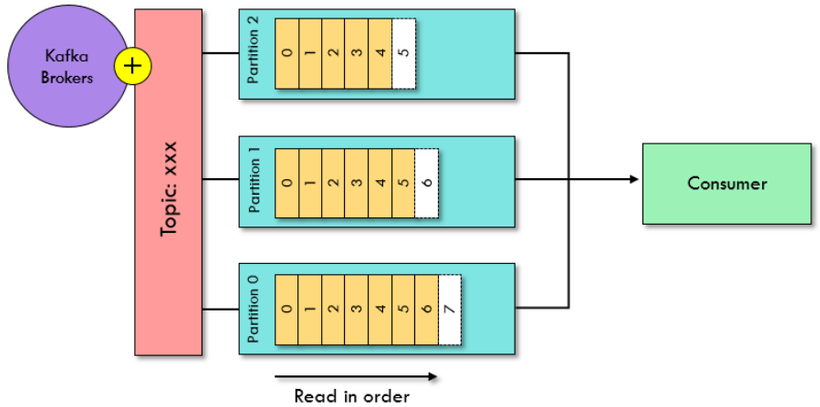
Kafka Consumer



Consumers

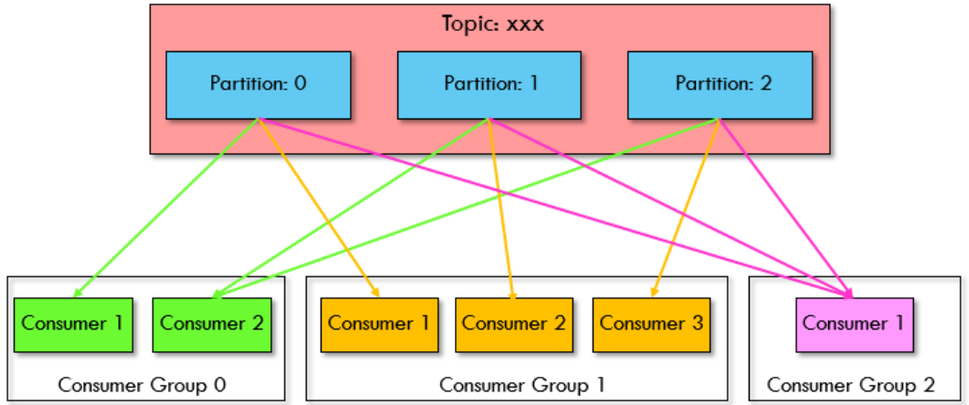


- **Consumers** read data from a topic
- They only have to specify the topic name and one broker to connect to, and Kafka will automatically take care of pulling the data from the right brokers
- Data is read in order **for each partitions**



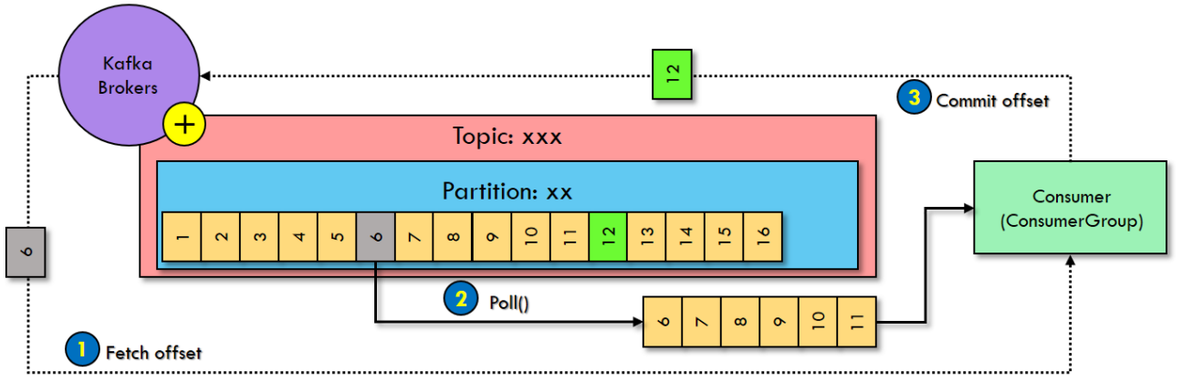
Consumer Groups

- Consumers read data in **consumer groups**
- Each consumer within a group reads from exclusive partitions
- You should control consumers instances less or equal than partitions (otherwise some will be inactive)



Consumer Offsets

- Kafka stores the offsets at which a **consumer group** has been reading
- The **offsets** commit live in a Kafka topic named "**__consumer_offsets**"
  - Key = [group, topic, partition] , Value=offset
- When a consumer has processed data received from Kafka, it should be **committing** the **offsets**
- If a consumer process dies, it will be able to read back from where it left off



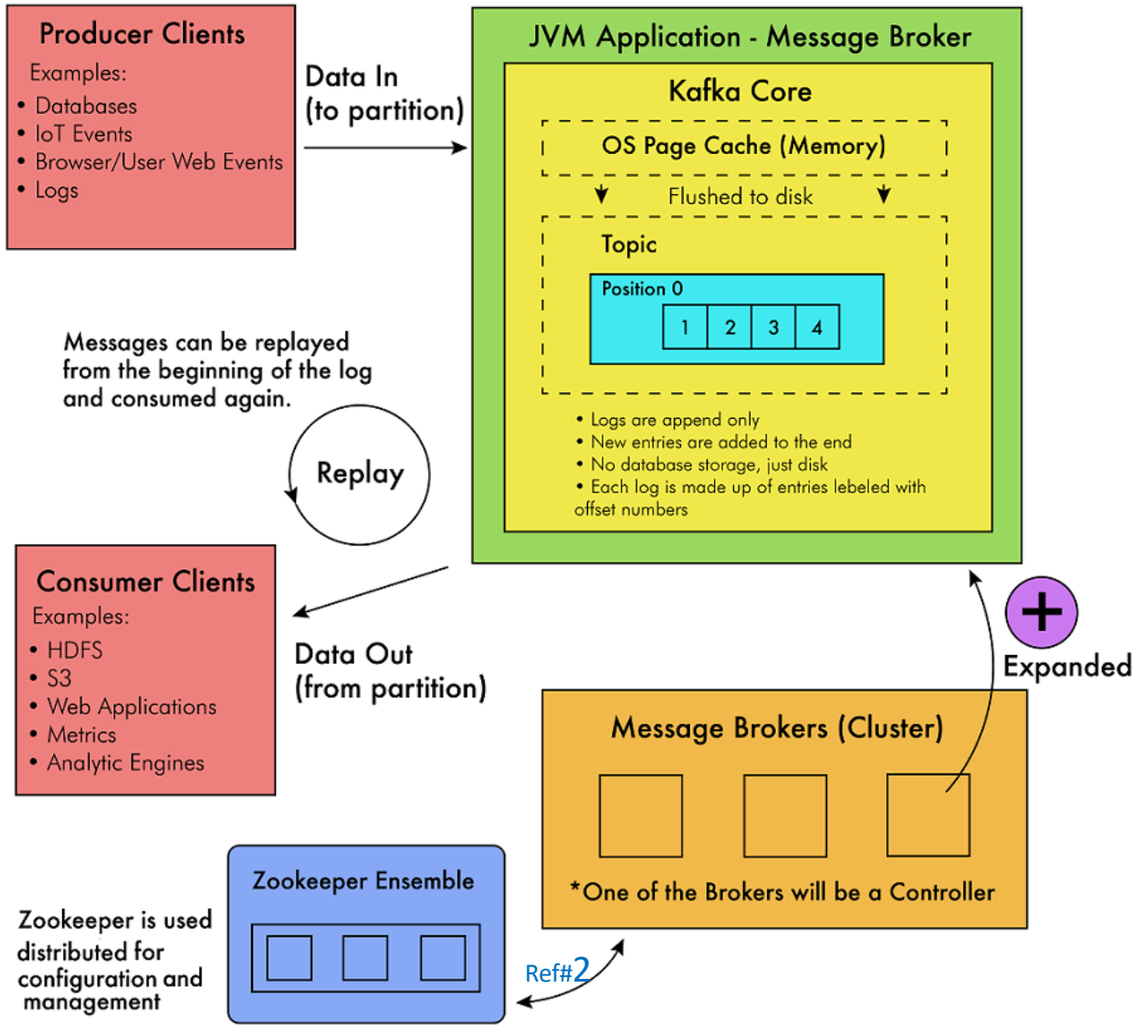
Kafka Guarantees

- Messages are appended to a **topic-partition** in the order they are sent
- Consumers read messages in the order stored in a **topic-partition**
- With a replication factor of N, producers and consumers can tolerate up to N-1 brokers being down
- This is why a replication factor of 3 is a good idea:
  - Allows for one broker to be taken down for maintenance
  - Allows for another broker to be taken down unexpectedly
- As long as the number of partitions remains constant for a topic (no new partitions), the same key will always go to the same partition

Delivery semantics for consumers

- As learned before, consumers choose when to commit offsets.
- **At most once**: offsets are committed as soon as the message is received. If the processing goes wrong, the message will be lost (it won't be read again).
- **At least once**: offsets are committed after the message is processed. If the processing goes wrong, the message will be read again. This can result in duplicate processing of messages. Make sure your processing is idempotent (i.e. processing again the message won't impact your systems)
- **Exactly once**: Very difficult to achieve / need strong engineering. Since version 0.11, Kafka start to support Exactly-once.

Terminology of Kafka



Module 3：Hands-on Practice of Apache Kafka

3-1：Kafka Topic Operations

3-2：Producer

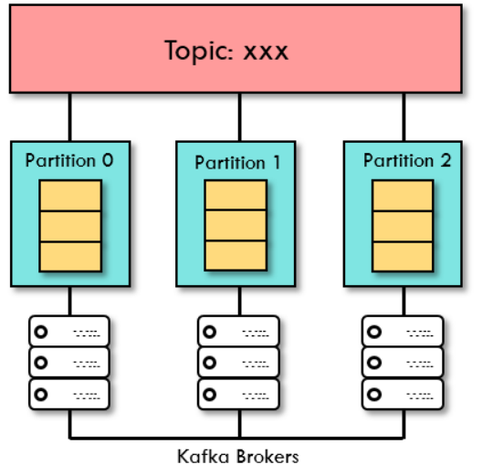
3-3：Consumer

Hands-on Practice

- In your Centos 7 VM - Go into Kafka docker container
  - $ su root
  - $ systemctl start docker
  - $ cd ~
  - $ ls
  - $ git clone https://github.com/semicolon1709/kafka-tutorial-docker-env.git
  - $ cd kafka-tutorial-docker-env
  - $ docker-compose up -d
  - $ docker exec -it kafka bash

Kafka Topic operations
create, list, delete, describe

- Create
  - kafka-topics --create --zookeeper zookeeper:2181
    --replication-factor 1 --partitions 1
    --topic test
- List
  - kafka-topics --list --zookeeper zookeeper:2181
- Describe
  - kafka-topics --describe --zookeeper zookeeper:2181 --topic test
- Delete
  - kafka-topics --delete --zookeeper zookeeper:2181 --topic test



Kafka Topic: create

```
$ kafka-topics \
  --create \
  --zookeeper zookeeper:2181 \
  --replication-factor 1 --partitions 1 \
  --topic test
```

```
root@kafka:/# kafka-topics --create \
>   --zookeeper zookeeper:2181 \
>   --replication-factor 1 \
>   --partitions 1 \
>   --topic test
Created topic "test".
```

在container裡頭的zookeeper服務的Hostname

Topic “test” is created!

Kafka Topic: list

```
$ kafka-topics \
  --list \
  --zookeeper zookeeper:2181
```

```
root@kafka:/# kafka-topics --list --zookeeper zookeeper:2181
__confluent.support.metrics
test
```

The command will list out all existing “topics”

Kafka Topic: describe

```
$ kafka-topics \
  --describe \
  --zookeeper zookeeper:2181 \
  --topic test
```

```
root@kafka:/# kafka-topics --describe --zookeeper zookeeper:2181 --topic test
Topic:test      PartitionCount:1        ReplicationFactor:1     Configs:
        Topic: test     Partition: 0    Leader: 1       Replicas: 1     Isr: 1
```

The command tells many details of a "topic"

Kafka Topic: delete

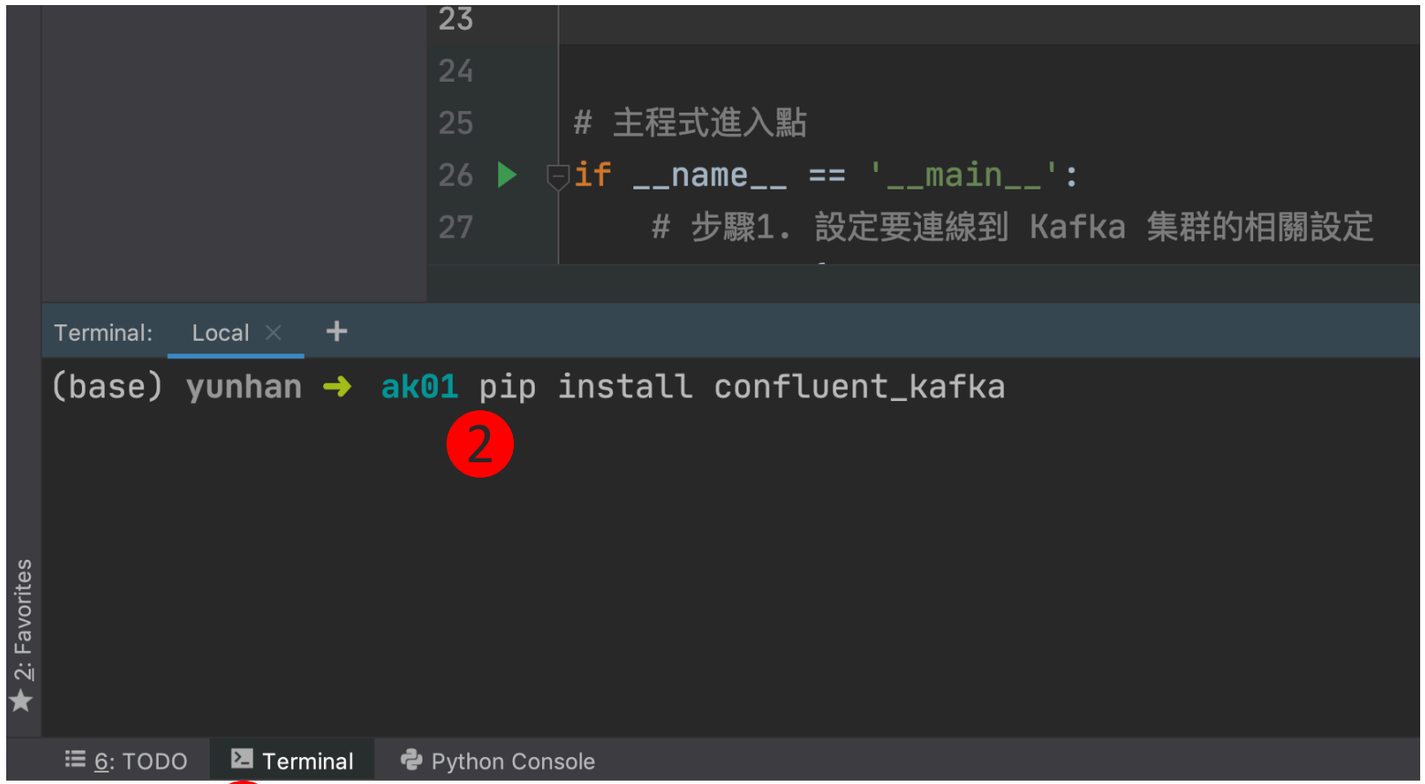
```
$ kafka-topics \
  --delete \
  --zookeeper zookeeper:2181 \
  --topic test
```

```
root@kafka:/# kafka-topics --delete --zookeeper zookeeper:2181 --topic test
Topic test is marked for deletion.
Note: This will have no impact if delete.topic.enable is not set to true.
```

The command will has the "topic" marked for deletion!

Hands-on Practice

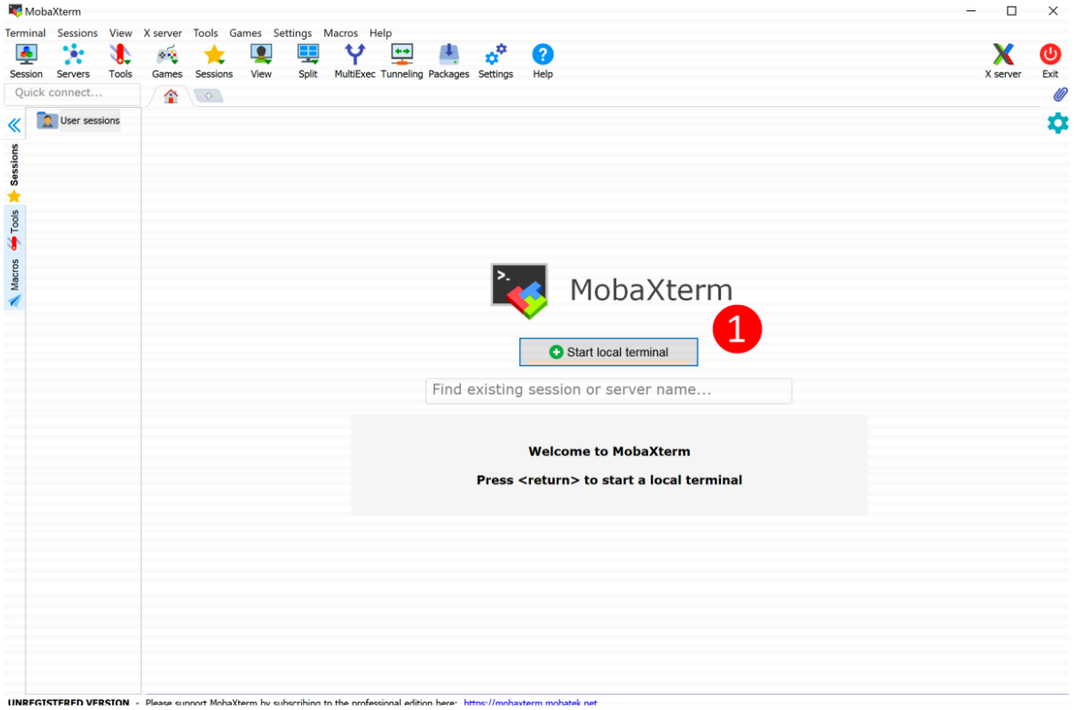
- Python (In your Windows Host)
  - Install PyCharm (https://www.jetbrains.com/pycharm/)
  - Open Project with PyCharm: Kafka-Tutorial/02_document/ak01
  - install package confluent_kafka



Hands-on Practice

Connecting to Kafka on VM from Windows – SSH Tunneling

- MobaXterm (command) - Kafka-Tutorial/01_software/MobaXterm_Portable_v12.4



Hands-on Practice

Connecting to Kafka on VM from Windows – SSH Tunneling

- MobaXterm (command) - Kafka-Tutorial/01_software/MobaXterm_Portable_v12.4

Hands-on Practice

創建一個新的Topic – test3 - In Kafka docker (Centos 7 VM)

- kafka-topics \
  --create \
  --zookeeper zookeeper:2181 \
  --replication-factor 1 --partitions 1 \
  --topic test3

Hands-on Practice

- Producer - console: In Kafka docker (Centos 7 VM)
  kafka-console-producer \

  --broker-list localhost:9092 \

  --topic test3

- Consumer - python: PyCharm (Windows Host)
  RUN Python Code kafka_consumer.py

Hands-on Practice

- Producer - console: In Kafka docker (Centos 7 VM)
  kafka-console-producer \
  --broker-list localhost:9092 \
  --topic test3

- Consumer - console: In Kafka docker (Centos 7 VM)
  kafka-console-consumer \
  --bootstrap-server localhost:9092 \
  --topic test3

Hands-on Practice

- Producer - python: PyCharm (Windows Host)
  RUN Python Code kafka_producer.py

- Consumer - console: In Kafka docker (Centos 7 VM)
  kafka-console-consumer \

  --bootstrap-server localhost:9092 \

  --topic test3

Hands-on Practice

- Producer - python: PyCharm (Windows Host)
  RUN Python Code kafka_producer.py

- Consumer - python : PyCharm (Windows Host)
  RUN Python Code kafka_consumer.py

學習目標： A

Module 4：Basic Topic Configurations

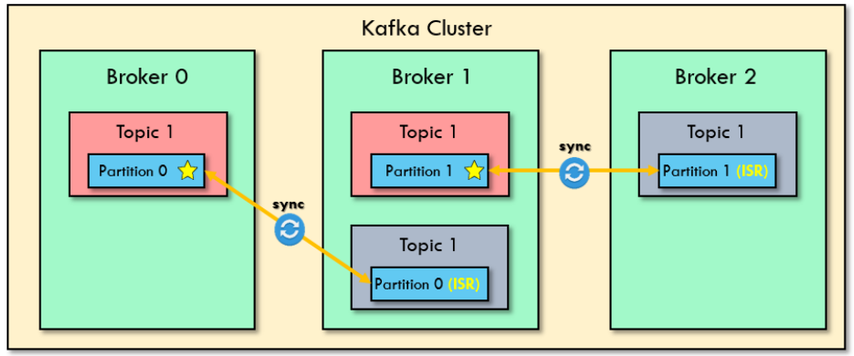
4-1：Partitions

4-2：Replication Factors

4-3：Segments

Partitions Count, Replication Factor

- The two **most important** parameters when creating a topic.
- The impact performance and durability of the system overall



- It is best to get the parameters right the first time!
  - If the **Partitions Count** increases during a topic life-cycle, you will break your keys ordering guarantees
  - If the **Replication Factor** increases during a topic life-cycle, you put more pressure on your cluster, which can lead to unexpected performance decrease

Partitions Count

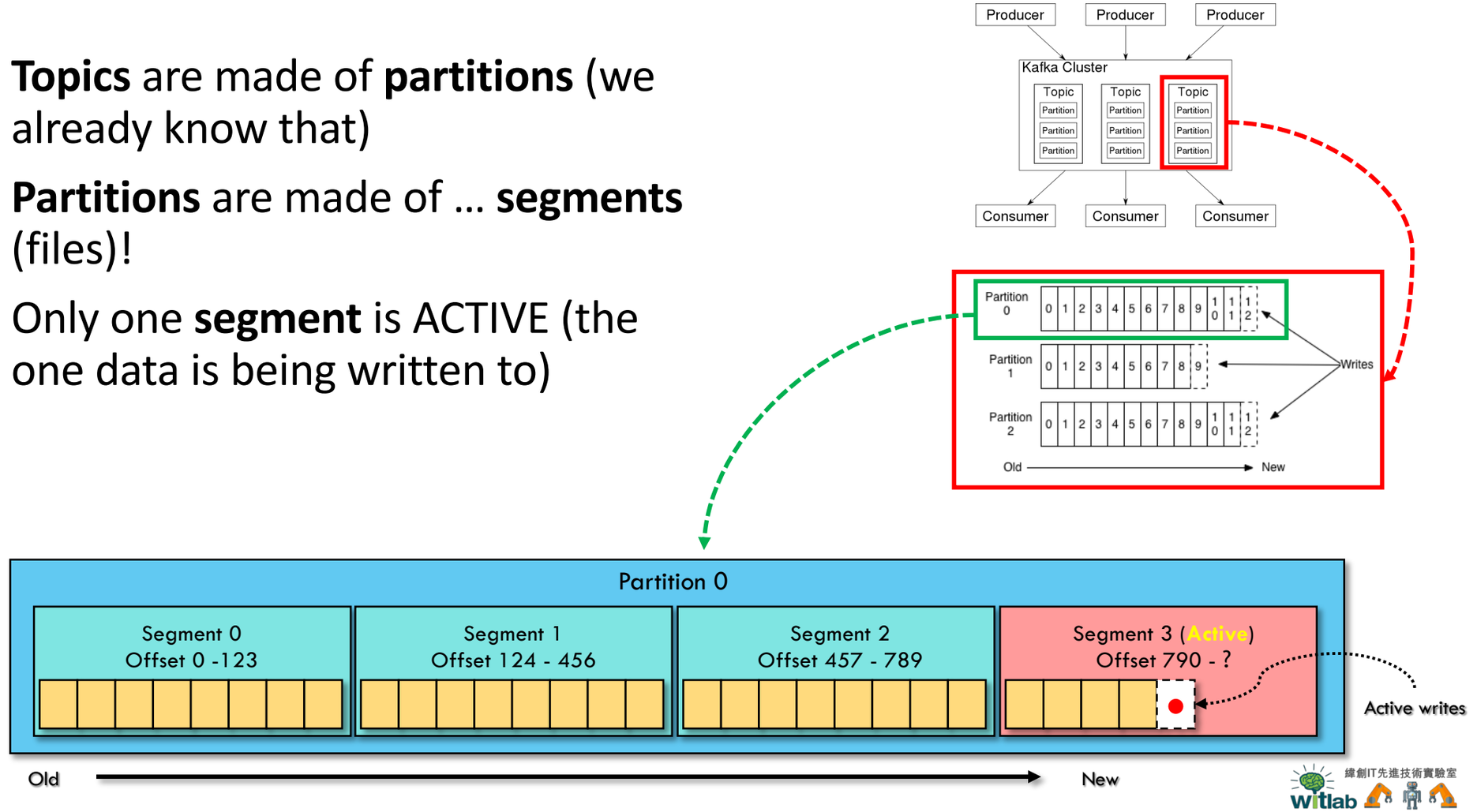
- Roughly, each partition can get a throughput of 10 MB / sec
- More partitions implies:
  - Better parallelism, better throughput
  - BUT more files opened on your system
  - BUT if a broker fails (unclean shutdown), lots of concurrent leader elections
  - BUT added latency to replicate (in the order of milliseconds)
- Guidelines:
  - Partitions per topic = (1 to 2) x (# of brokers), max 10 partitions
  - Example: in a 3 brokers setup, 3 or 6 partitions is a good number to start with

Replication Factor

- Should be at least **2**, maximum of **3**
- The higher the replication factor:
  - Better resilience of your system (N-1 brokers can fail)
  - BUT longer replication (higher latency is acks=all)
  - BUT more disk space on your system (50% more if RF is 3 instead of 2)
- Guidelines:
  - **Set it to 2** (if you have 3 brokers)
  - **Set it to 3** (if you have greater than 5 brokers)
  - If replication performance is an issue, get a better broker instead of less replication factor

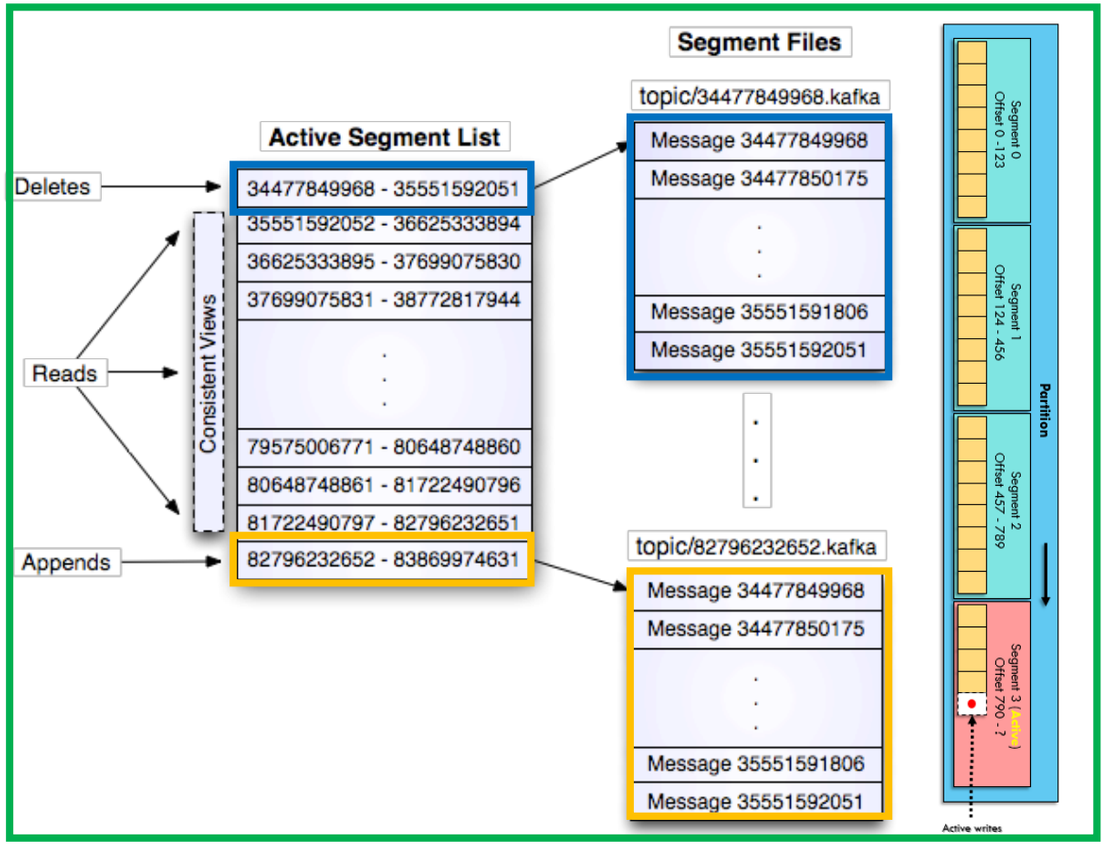
Partitions and Segments

- **Topics** are made of **partitions** (we already know that)
- **Partitions** are made of … **segments** (files)!
- Only one **segment** is ACTIVE (the one data is being written to)

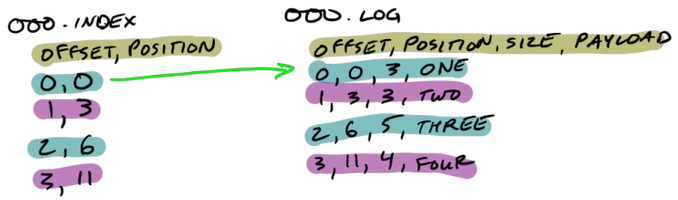


Partitions and Segments

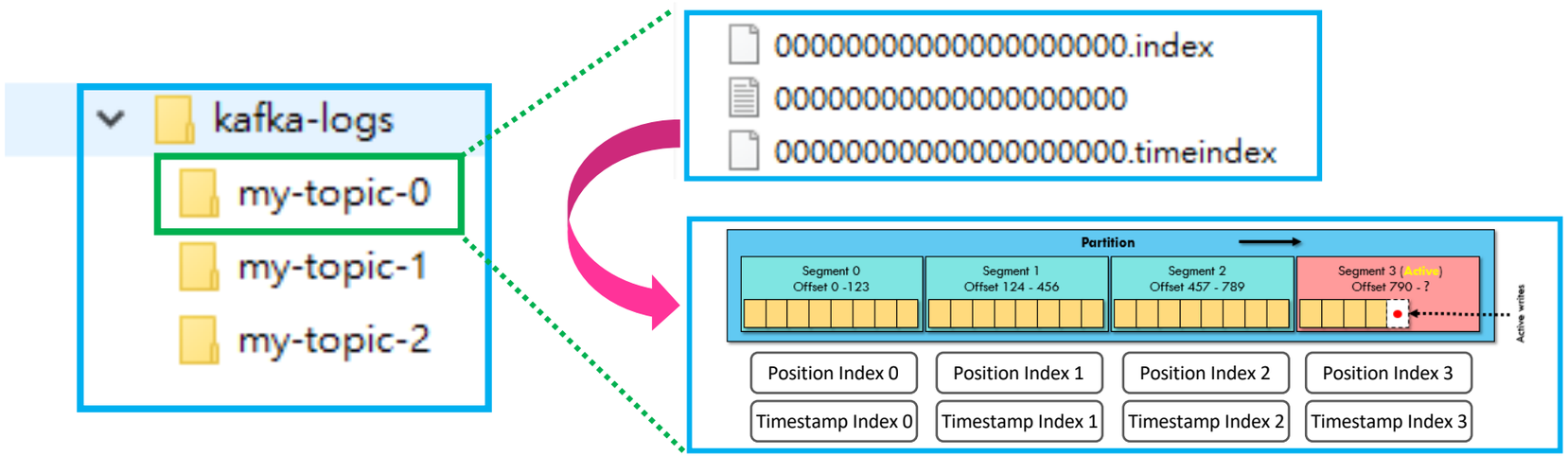
- Two segment settings:
  - **log.segment.bytes**: the max size of a single segment in bytes
  - **segment.ms**: the time kafka will wait before committing the segment if not full



Segments and Indexes



- Segments come with **two** indexes (files):
  - An offset to position index: allows Kafka where to read to find a message
  - A timestamp to offset index: allow Kafka to find messages with a timestamp
- Therefore, Kafka knows where to find data in a constant time!



Segments and Indexes
Create Topic (test4)

```
$ kafka-topics
    --create
    --zookeeper zookeeper:2181
    --replication-factor 1 --partitions 1
    --topic test4
```

```
root@kafka:/# kafka-topics --create --zookeeper zookeeper:2181 \
>    --replication-factor 1 --partitions 1 --topic test4
Created topic "test4".
```

請注意partitions的數量與topic名稱

在kafka的資料目錄下會找到對應的folder名稱

```
root@kafka:/# cd /var/lib/kafka/data
root@kafka:/var/lib/kafka/data#
root@kafka:/var/lib/kafka/data# ls -l test4-0
total 0
-rw-r--r-- 1 root root 10485760 Sep  6 08:57 00000000000000000000.index
-rw-r--r-- 1 root root        0 Sep  6 08:57 00000000000000000000.log
-rw-r--r-- 1 root root 10485756 Sep  6 08:57 00000000000000000000.timeindex
-rw-r--r-- 1 root root        0 Sep  6 08:57 leader-epoch-checkpoint
```

Segments: Why should I care?

- A small **log.segment.bytes** (size, default: 1GB) means:
  - More segments per partitions
  - Log Compaction happens more often
  - BUT Kafka has to keep more files opened (Error: Too many open files)
- Ask yourself: how fast will I have new segments based on throughput?
- A small **log.roll.hours** or **log.roll.ms**(time, default 1 week) means:
  - You set a max frequency for log compaction (more frequent triggers)
  - Maybe you want daily compaction instead of weekly?
- Ask yourself: how often do I need log compaction to happen?

學習目標： B

Module 5：Log Cleanup Policies

5-1：Delete

5-2：Compact

5-3：Clean up Policy Configurations

Log Cleanup Policies

- Many Kafka clusters make data expire, according to a policy
- That concept is called "**log cleanup**".
  - Policy 1: **log.cleanup.policy**=**delete** (Kafka default for all user topics)
    - Delete based on age of data (default is a week)
    - Delete based on max size of log (default is -1 == infinite)
  - Policy 2: **log.cleanup.policy**=**compact** (Kafka default for topic **__consume_offsets**)
    - Delete based on keys of your messages
    - Will delete old duplicate keys **after** the active segment is committed
    - Infinite time and space retention

Log Cleanup: Why and When?

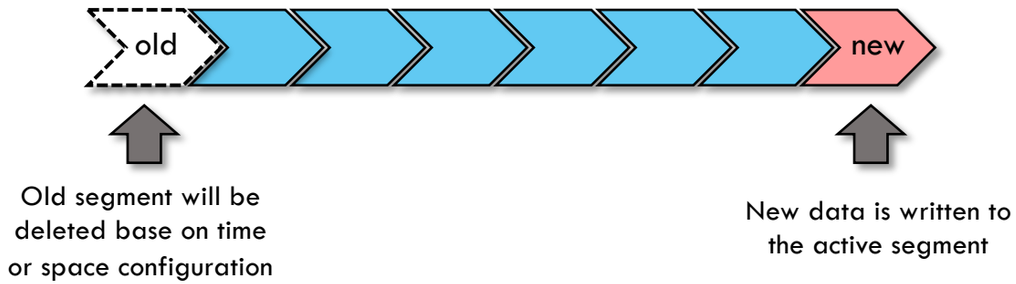
- Deleting data from Kafka allows you to:
  - Control the size of the data on the disk, delete obsolete data
  - Overall: Limit maintenance work on the Kafka Cluster

- How often does log cleanup happen?
  - Log cleanup happens on your **partition segments**!
  - Smaller / More segments means that log cleanup will happen more often!
  - Log cleanup shouldn’t happen too often => takes CPU and RAM resources
  - The cleaner checks for work every 15 seconds (**log.cleaner.backoff.ms**)

Log Cleanup Policy: log.cleanup.policy=delete

- **log.retention.hours**:
  - Number of hours to keep data for (default is 168 – one week)
  - Higher number means more disk space
  - Lower number means that less data is retained (your consumers may need to replay more data than less)
- **log.retention.bytes**:
  - Max size in Bytes for each partition (default is -1 == infinite)
  - Useful to keep the size of a log under a threshold

Log Cleanup Policy: Delete
log.cleanup.policy=delete



Use cases – two common pair of options:

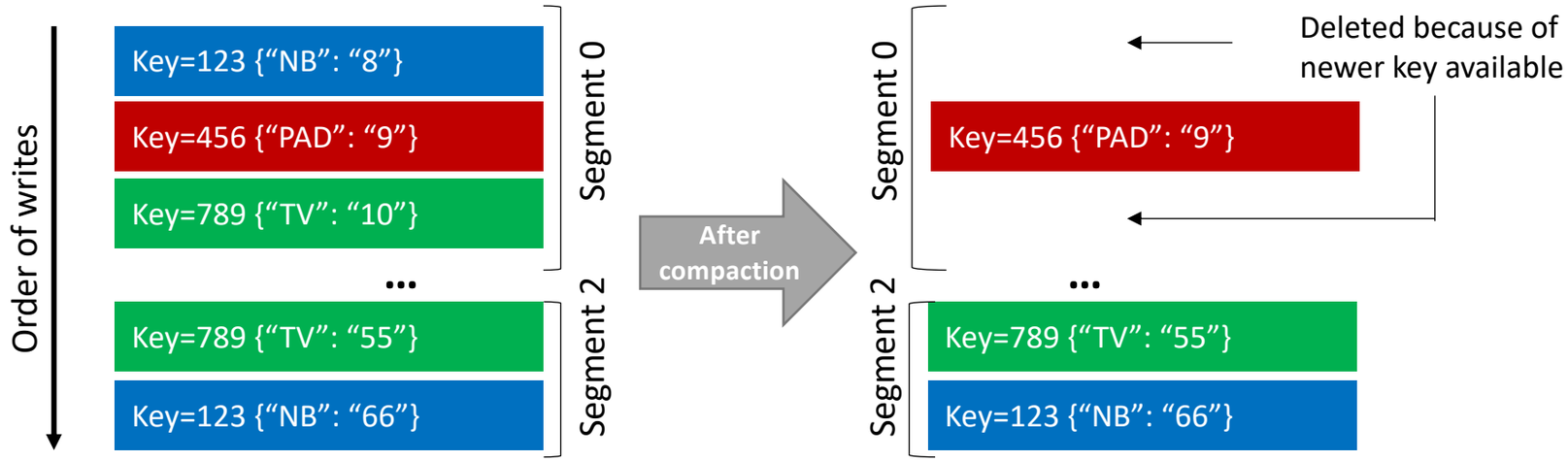
- One week of retention:
  - **log.retention.hours = 168** and **log.retention.bytes = -1**
- Infinite time retention bounded by 500 MB:
  - **log.retention.hours = 17520** and **log.retention.bytes = 524288000**

Log Cleanup Policy: Compact
log.cleanup.policy=compact

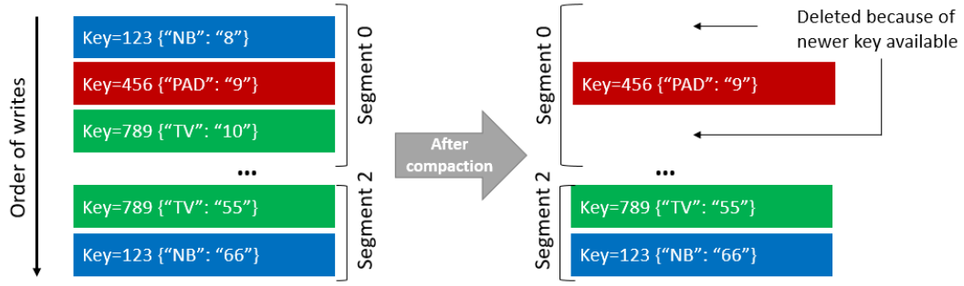
- Log compaction ensures that your log contains *at least the last know value for a specific key within a partition*
- Very useful if we just require a SNAPSHOT instead of full history (such as for a data table in a database)
- The idea is that we only keep the latest “update” for a key in our log

Log Cleanup Policy: Example
log.cleanup.policy=compact

- Our topic is: **product-inventory**
- We want to keep the most recent inventory for our products



Log Cleanup Policy: Example
log.cleanup.policy=compact



1

```
root@kafka:/# kafka-topics --zookeeper zookeeper:2181 --create \
>   --topic product-inventory \
>   --partitions 1 --replication-factor 1 \
>   --config cleanup.policy=compact \
>   --config min.cleanable.dirty.ratio=0.00001 \
>   --config segment.ms=1000
Created topic "product-inventory".
```

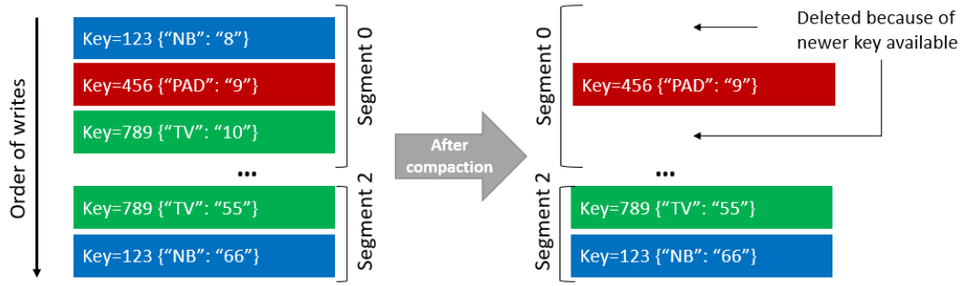
特別去設定這個topic的一些參數來展現log-compaction的效果

2

```
root@kafka:/# kafka-configs --zookeeper zookeeper:2181 \
>   --entity-type topics  \
>   --entity-name product-inventory \
>   --describe
Configs for topic 'product-inventory' are min.cleanable.dirty.ratio=0.00001,cleanup.policy=compact,segment.ms=1000
```

檢查topic的參數

Log Cleanup Policy: Example
log.cleanup.policy=compact



2 Producer#1

再開一個Console-producer來發佈資料

```
root@kafka:/# kafka-console-producer --broker-list kafka:9092 \
> --topic product-inventory \
> --property parse.key=true \
> --property key.separator=,
>123,{"NB":"8"}
456,{"PAD":"9"}
789,{"TV":"10"}
456,{"PAD":"7"}
789,{"TV":"11"}
456,{"PAD":"6"}
789,{"TV":"12"}
456,{"PAD":"5"}
789,{"TV":"13"}
```

3 貼上範例資料

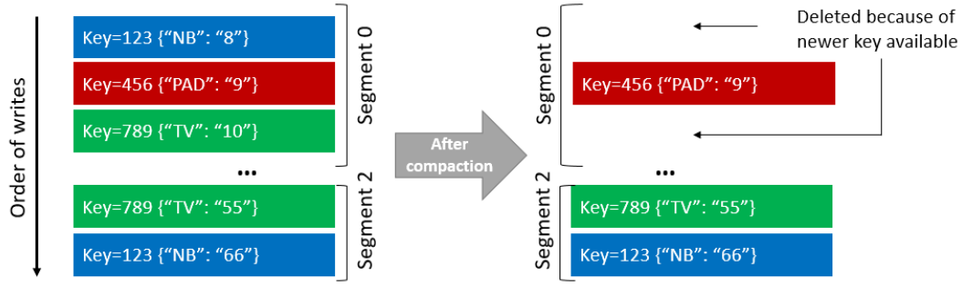
1 Consumer#1

開一個Console-consumer訂閱這個“product-inventory”的topic

```
root@kafka:/# kafka-console-consumer \
>    --bootstrap-server kafka:9092 \
>    --topic product-inventory \
>    --from-beginning \
>    --property print.key=true \
>    --property key.separator=,
123,{"NB":"8"}
456,{"PAD":"9"}
789,{"TV":"10"}
456,{"PAD":"7"}
789,{"TV":"11"}
456,{"PAD":"6"}
789,{"TV":"12"}
456,{"PAD":"5"}
789,{"TV":"13"}
```

4 即時收到發佈的範例資料

Log Cleanup Policy: Example
log.cleanup.policy=compact



5

等待10秒後....

再開一個Console-consumer訂閱這個“product-inventory”的topic

8

Producer#1

```
root@kafka:/# kafka-console-producer --broker-list kafka:9092 \
> --topic product-inventory \
> --property parse.key=true \
> --property key.separator=,
>789,{"TV":"55"}
123,{"NB":"66"}
```

6

貼上最後2筆範例資料

Consumer#1

```
root@kafka:/# kafka-console-consumer \
>    --bootstrap-server kafka:9092 \
>    --topic product-inventory \
>    --from-beginning \
>    --property print.key=true \
>    --property key.separator=,
123,{"NB":"8"}
456,{"PAD":"9"}
789,{"TV":"10"}
456,{"PAD":"7"}
789,{"TV":"11"}
456,{"PAD":"6"}
789,{"TV":"12"}
456,{"PAD":"5"}
789,{"TV":"13"}
789,{"TV":"55"}
123,{"NB":"66"}
```

7

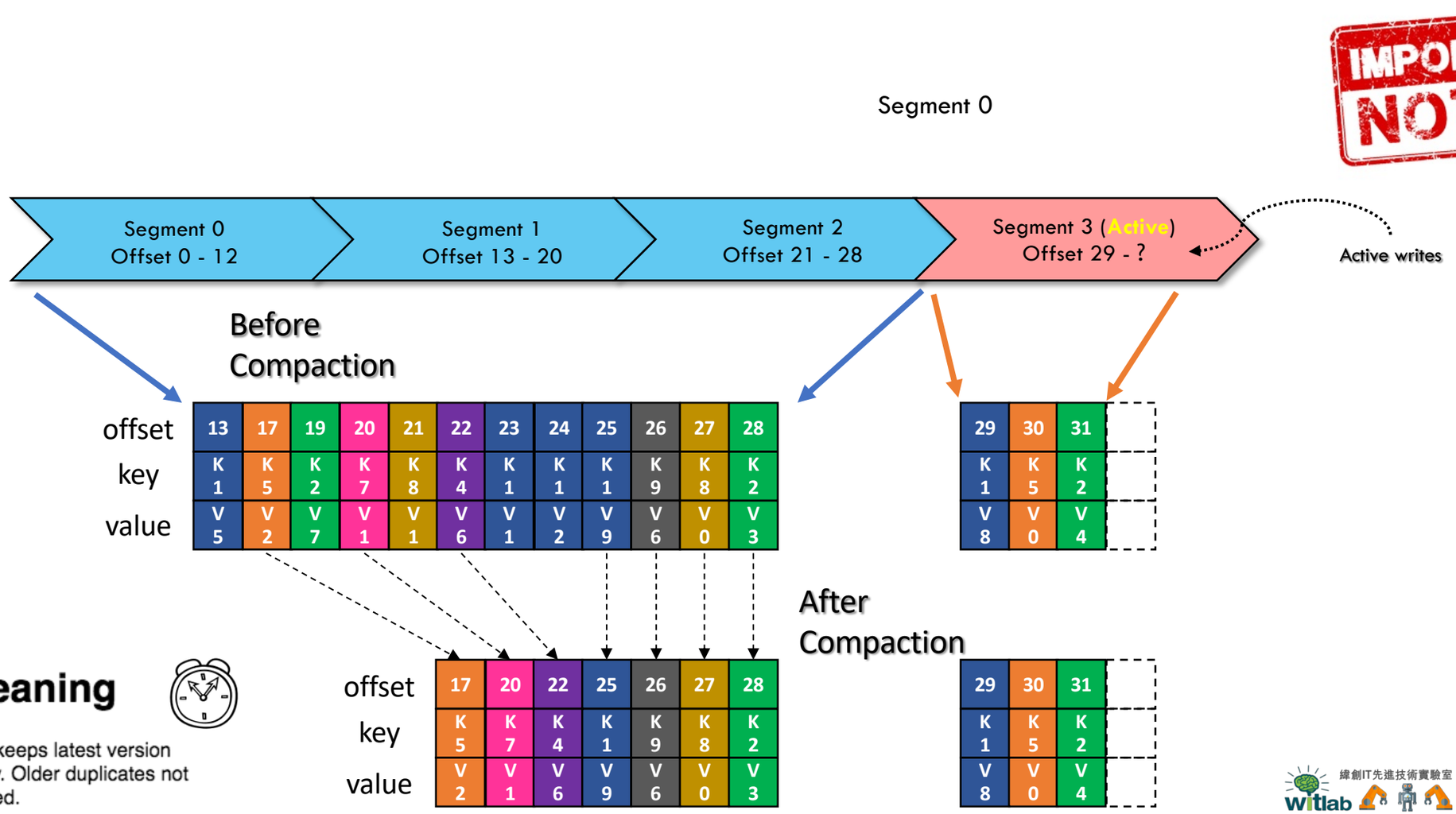
Console-consumer確實地收到了最後發佈的最後2筆範例資料

Consumer#2

```
root@kafka:/# kafka-console-consumer \
>    --bootstrap-server kafka:9092 \
>    --topic product-inventory \
>    --from-beginning \
>    --property print.key=true \
>    --property key.separator=,
123,{"NB":"8"}
456,{"PAD":"5"}
789,{"TV":"13"}
789,{"TV":"55"}
123,{"NB":"66"}
```

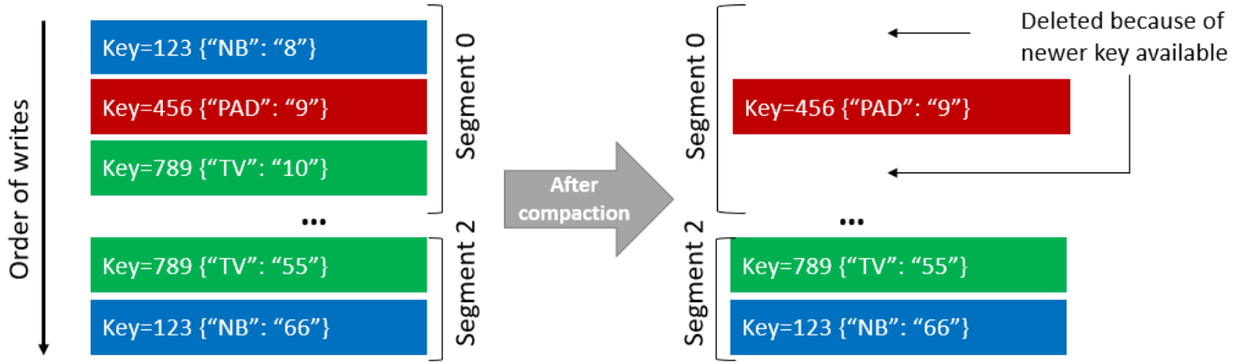
經過compact之後的結果！

Log Cleanup Policy: Example
log.cleanup.policy=compact



Log Compaction: Example
log.cleanup.policy=compact

- Any consumer that is reading *from the head of a log* will still see all the messages sent to the topic
- Ordering of message is kept, log compaction only removes some messages, but does not re-order them
- The offset of a message is immutable (it never changes). Offsets are just skipped if a message is missing



學習目標： B

Module 6：Guidelines of Topic Configurations

6-1：Log Compression

6-2：Other advanced configurations

6-3：Why should I care abouttopic configuration?

Log Compression

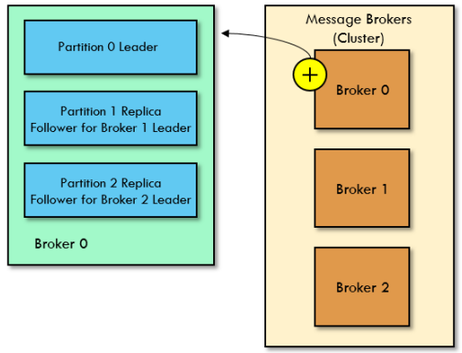
- Topics can be compressed using **compression.type** setting.
- Options are '**gzip**', '**snappy**', '**lz4**', '**uncompressed**', '**producer**'
- If you need compression, ideally you keep default as 'producer'.
  - The producer will perform the compression on its side
  - The broker will take the data as is => Saves CPU resources on the broker
- If compression is enabled, make sure you're sending batches of data
- Data will be uncompressed by the consumer!
- Compression only makes sense if you're sending non-binary data (json, xml, text…), don't enable compression for binary data (parquet, protobuf, avro..)

Other advanced configurations

- **max.message.bytes** (default is **1MB**): if your messages get bigger than 1MB, increase this parameter on the topic and your consumers buffer
- **min.insync.replicas** (default is **1**): if using acks=all, specify how many brokers need to acknowledge the write
- **unclean.leader.election.enable** (danger zone! – default **false**): if set to true, it will allow replicas which are not in sync to become leader as a last resort if all ISRs are offline. This can lead to data loss. If set to false, the topic will go offline until the ISRs come back up

Why should I care about topic configuration?

- Brokers have defaults for all the topic configuration parameters
- These parameters impact **performance** and **topic behavior**
- Some topics may need different values than the defaults
  - Replication Factor
  - Number of Partitions
  - Message size
  - Compression level
  - Log Cleanup Policy
  - Other configurations
- A list of configuration can be found at:
  - https://kafka.apache.org/documentation/#topicconfigs



學習目標： B

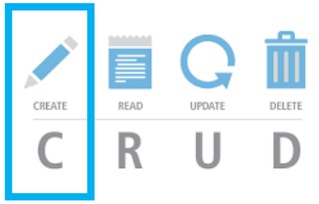
Module 7：Hands-on Practice of Topic Configurations

7-1：Create Topic
7-2：Alter Topic
7-3：Delete Topic

Topic configuration example:

- Configurations pertinent to topics have both a **server default** as well an optional **per-topic override**.
- *If no **per-topic configuration** is given the **server default** is used.*
- The override can be set at topic creation time by giving one or more **--config** options.

Topic configuration example: (Create)



- Configurations pertinent to topics have both a server default as well an optional per-topic override.
- If no per-topic configuration is given the server default is used.

```
kafka-topics \
--zookeeper zookeeper:2181 \
--create \
--topic my-topic \
--partitions 3 \
--replication-factor 1 \
--config max.message.bytes=64000 \
--config flush.messages=1
```

Topic configuration example: (Query)


```
kafka-topics \
--zookeeper zookeeper:2181 \
--topic my-topic  \
--describe
```

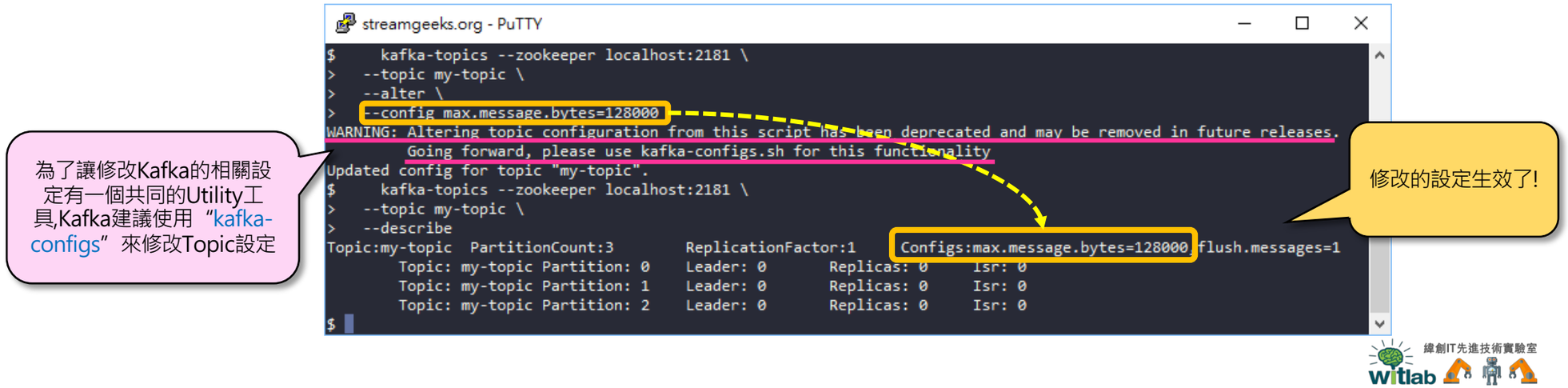
```
streamgeeks.org - PuTTY
$     kafka-topics --create \
>   --zookeeper localhost:2181 \
>   --topic my-topic \
>   --partitions 3 \
>   --replication-factor 1 \
>   --config max.message.bytes=64000 \
>   --config flush.messages=1
Created topic "my-topic".
$
```

```
streamgeeks.org - PuTTY
$     kafka-topics --zookeeper localhost:2181 \
>   --topic my-topic \
>   --describe
Topic:my-topic  PartitionCount:3        ReplicationFactor:1     Configs:max.message.bytes=64000,flush.messages=1
        Topic: my-topic Partition: 0    Leader: 0       Replicas: 0     Isr: 0
        Topic: my-topic Partition: 1    Leader: 0       Replicas: 0     Isr: 0
        Topic: my-topic Partition: 2    Leader: 0       Replicas: 0     Isr: 0
$
```

Kafka會先載入預設的config, 然後再加上特別設定的config!

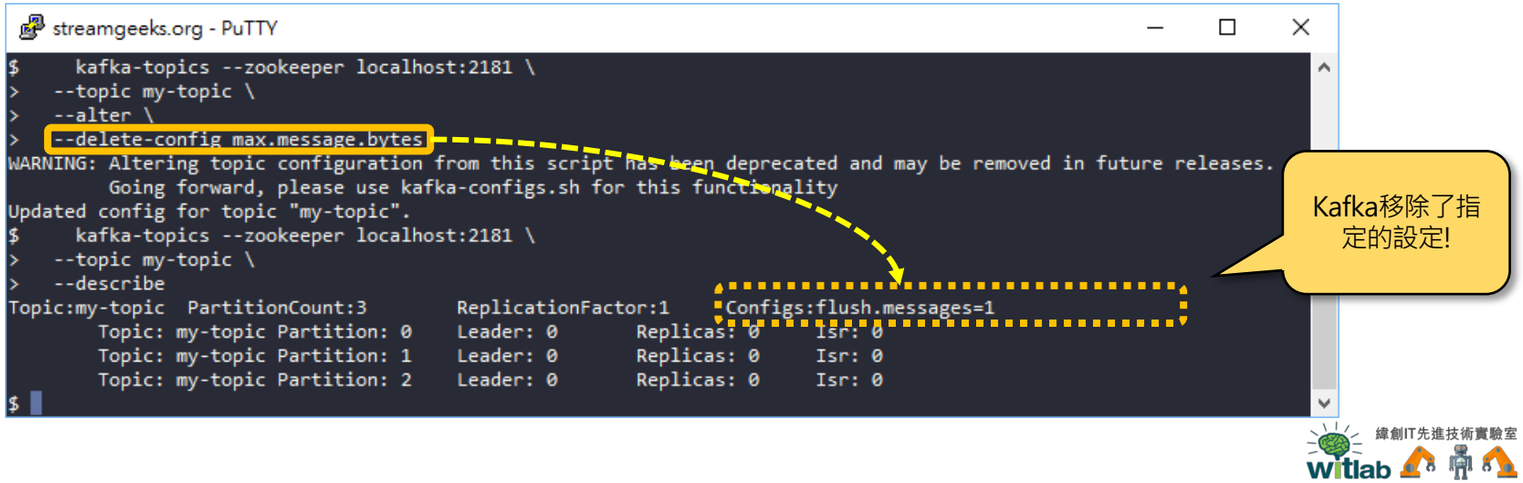
Topic configuration example: (Update)


```
kafka-topics \
--zookeeper zookeeper:2181  \
--topic my-topic \
--alter \
--config max.message.bytes=128000
```



Topic configuration example: (Remove)


```
kafka-topics \
--zookeeper zookeeper:2181 \
--topic my-topic \
--alter \
--delete-config max.message.bytes
```

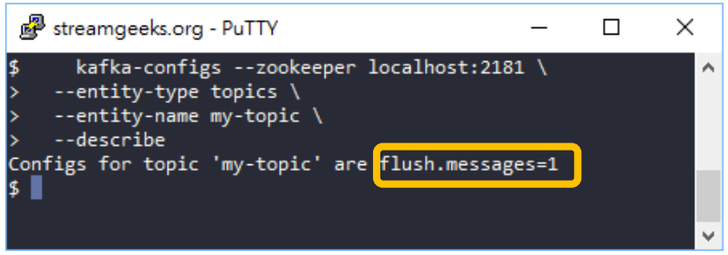


Topic configuration example: (Query)
** 2nd method - preferred**

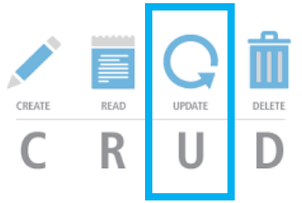

```
kafka-configs \
--zookeeper zookeeper:2181 \
--entity-type topics \
--entity-name my-topic \
--describe
```

使用kafka-configs的工具可以讓我們對Kafka的不同的entity-type來進行設定。包括: (topics/clients/users/brokers)

指定要修改的entity-name來進行設定。包括: (topic name/client id/user principal name/broker id)



Kafka configuration
kafka-configs

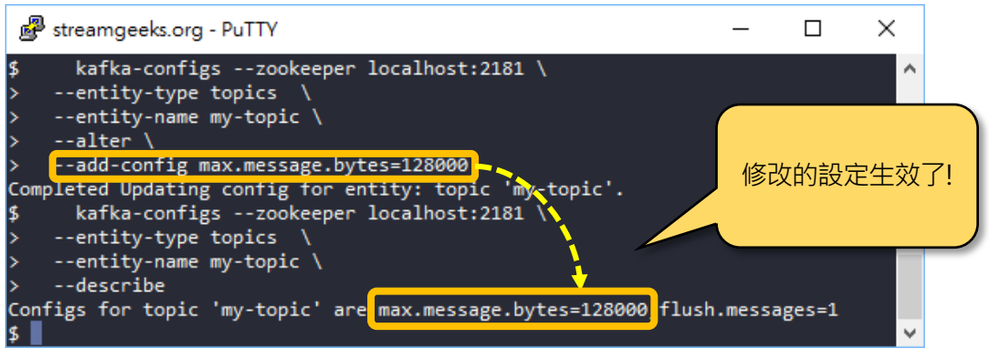


entity-type: **topics**

- cleanup.policy
- compression.type
- delete.retention.ms
- file.delete.delay.ms
- flush.messages
- flush.ms
- follower.replication.throttled.replicas
- index.interval.bytes
- leader.replication.throttled.replicas
- max.message.bytes
- message.format.version
- message.timestamp.difference.max.ms
- message.timestamp.type
- min.cleanable.dirty.ratio
- min.compaction.lag.ms
- min.insync.replicas
- preallocate
- retention.bytes
- retention.ms
- segment.bytes
- segment.index.bytes
- segment.jitter.ms
- segment.ms
- unclean.leader.election.enable

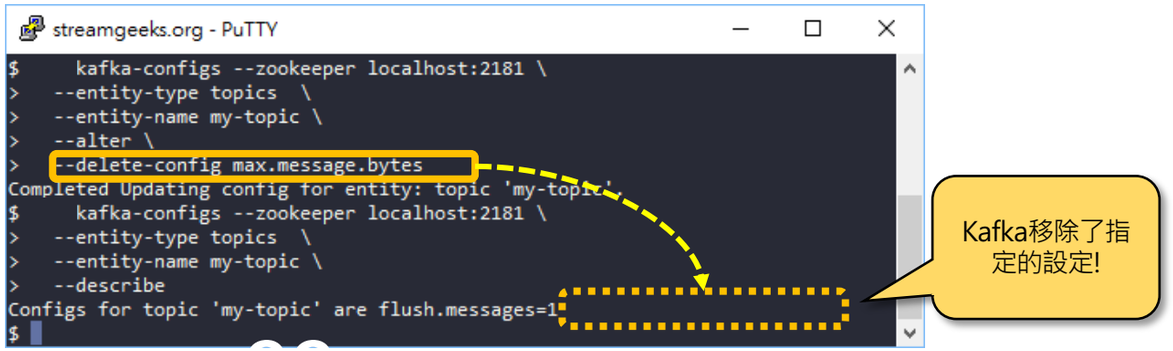
Topic configuration example: (Update)
** 2nd method - preferred**


```
kafka-configs \
--zookeeper zookeeper:2181 \
--entity-type topics \
--entity-name my-topic \
--alter  \
--add-config max.message.bytes=128000
```



Topic configuration example: (Remove)
** 2nd method - preferred**


```
kafka-configs \
--zookeeper zookeeper:2181 \
--entity-type topics \
--entity-name my-topic \
--alter  \
--delete-config max.message.bytes
```



學習目標： B

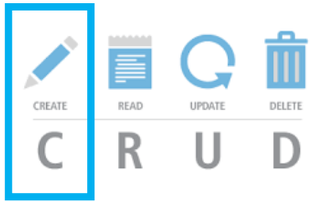
Module 8：Broker Configurations Example

8-1：Update

8-2：Query

8-3：Remove

Broker configuration



- From Kafka version **1.1** onwards, some of the broker configs can be updated *without* restarting the broker.
- See the *Dynamic Update Mode* column in Broker Configs for the update mode of each broker config.
  - read-only: Requires a broker restart for update
  - per-broker: May be updated dynamically for each broker
  - cluster-wide: May be update dynamically as a cluster-wide default

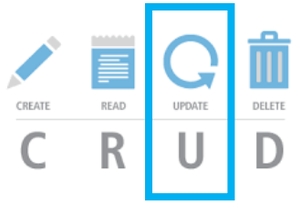
advertised.listeners

Listeners to publish to ZooKeeper for clients to use, if different than the `listeners` config property. In IaaS environments, this may need to be different from the interface to which the broker binds. If this is not set, the value for `listeners` will be used. Unlike `listeners`, it is not valid to advertise the 0.0.0.0 meta-address.
Also unlike `listeners`, there can be duplicated ports in this property, so that one listener can be configured to advertise another listener's address. This can be useful in some cases where external load balancers are used.

Type: string
Default: null
Valid Values:
Importance: high
Update Mode: per-broker

https://kafka.apache.org/documentation/#brokerconfigs

Kafka configuration
kafka-configs

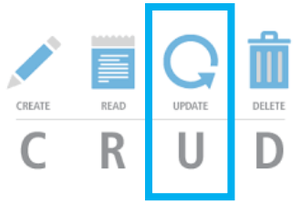


- advertised.listeners
- background.threads
- compression.type
- follower.replication.throttled.rate
- leader.replication.throttled.rate
- listener.security.protocol.map
- listeners
- log.cleaner.backoff.ms
- log.cleaner.dedupe.buffer.size
- log.cleaner.delete.retention.ms
- log.cleaner.io.buffer.load.factor
- log.cleaner.io.buffer.size

entity-type: **brokers**

- log.cleaner.io.max.bytes.per.second
- log.cleaner.min.cleanable.ratio
- log.cleaner.min.compaction.lag.ms
- log.cleaner.threads
- log.cleanup.policy
- log.flush.interval.messages
- log.flush.interval.ms
- log.index.interval.bytes
- log.index.size.max.bytes
- log.message.timestamp.difference.max.ms
- log.message.timestamp.type
- log.preallocate

Kafka configuration (Cont.)
kafka-configs

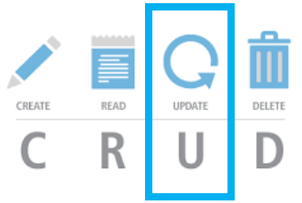


- log.retention.bytes
- log.retention.ms
- log.roll.jitter.ms
- log.roll.ms
- log.segment.bytes
- log.segment.delete.delay.ms
- message.max.bytes
- metric.reporters
- min.insync.replicas
- num.io.threads
- num.network.threads
- num.recovery.threads.per.data.dir

entity-type: **brokers**

- num.replica.fetchers
- principal.builder.class
- replica.alter.log.dirs.io.max.bytes.per.second
- sasl.enabled.mechanisms
- sasl.jaas.config
- sasl.kerberos.kinit.cmd
- sasl.kerberos.min.time.before.relogin
- sasl.kerberos.principal.to.local.rules
- sasl.kerberos.service.name
- sasl.kerberos.ticket.renew.jitter
- sasl.kerberos.ticket.renew.window.factor
- sasl.mechanism.inter.broker.protocol

Kafka configuration (Cont.)
kafka-configs



- ssl.cipher.suites
- ssl.client.auth
- ssl.enabled.protocols
- ssl.endpoint.identification.algorithm
- ssl.key.password
- ssl.keymanager.algorithm
- ssl.keystore.location
- ssl.keystore.password
- ssl.keystore.type
- ssl.protocol
- ssl.provider
- ssl.secure.random.implementation

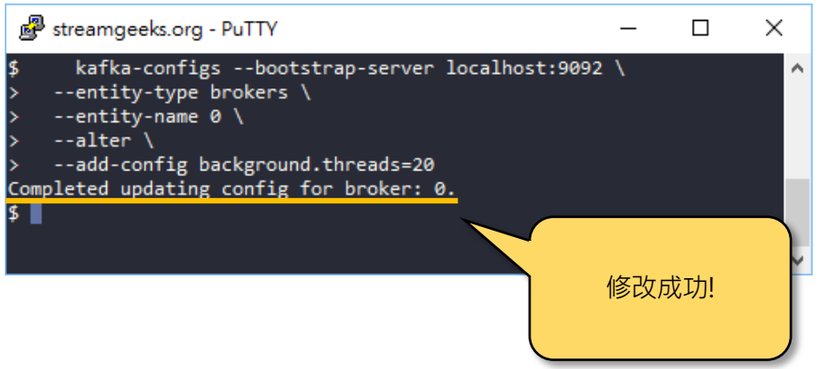
entity-type: **brokers**

- ssl.trustmanager.algorithm
- ssl.truststore.location
- ssl.truststore.password
- ssl.truststore.type
- unclean.leader.election.enable

Broker configuration example: (Update)

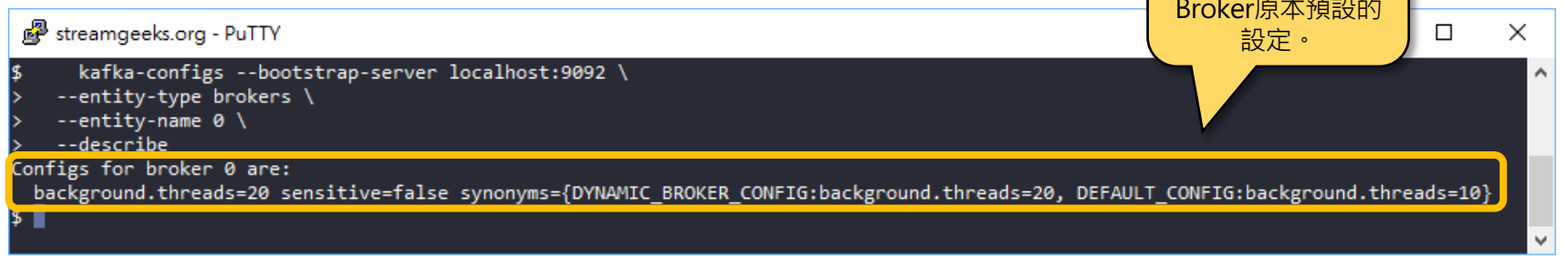
- To alter the current broker configs for broker id 0 (for example, the number of log cleaner threads):

```
kafka-configs \
--bootstrap-server localhost:9092 \
--entity-type brokers \
--entity-name 1 \
--alter \
--add-config background.threads=20
```



Broker configuration example: (Query)

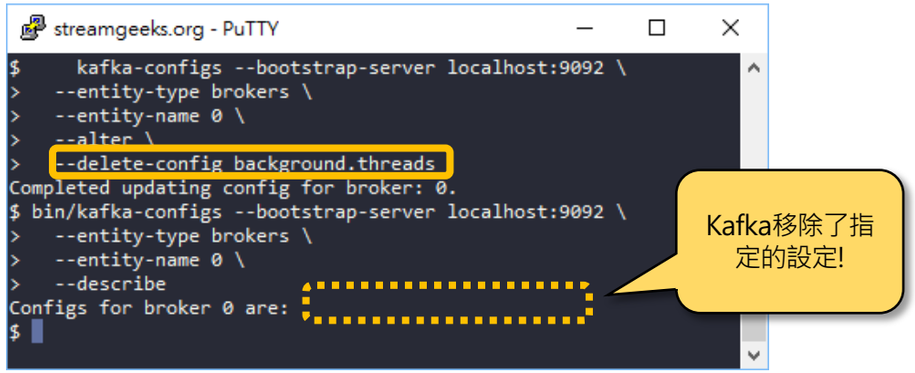

```
kafka-configs \
--bootstrap-server localhost:9092 \
--entity-type brokers \
--entity-name 1 \
--describe
```



Broker configuration example: (Remove)

- To delete a config override and revert to the statically configured or default value for broker id 0 (for example, the number of log cleaner threads):

```
kafka-configs \
--bootstrap-server localhost:9092 \
--entity-type brokers \
--entity-name 1 \
--alter  \
--delete-config background.threads
```



Where are these configuration stored?

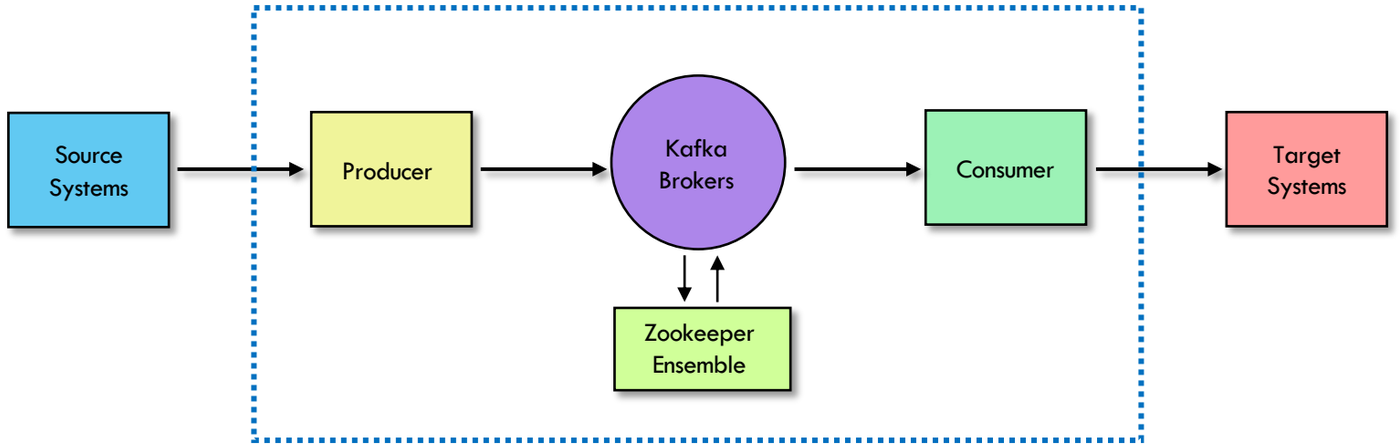
- Dynamic per-broker configurations stored in ZooKeeper
- Dynamic cluster-wide default configurations stored in ZooKeeper
- Static broker configurations from server.properties
- Kafka default, see broker configurations
  - https://kafka.apache.org/documentation/#brokerconfigs

學習目標： C

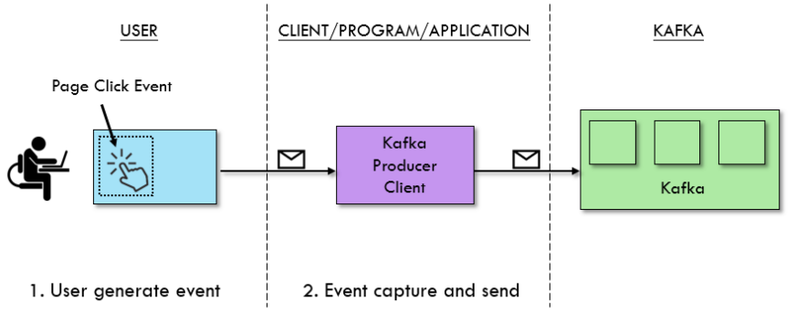
Module 9：Kafka Producers

9-1：Fire-and-forget
9-2：Synchronous send
9-3：Asynchronous send

Kafka Ecosystem:
Kafka Core



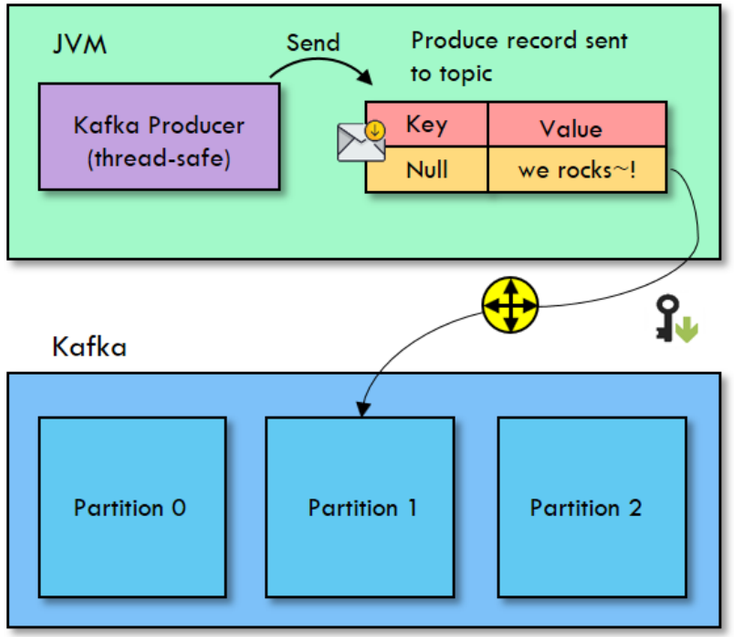
Open Project with PyCharm: Kafka-Tutorial/02_document/ak03

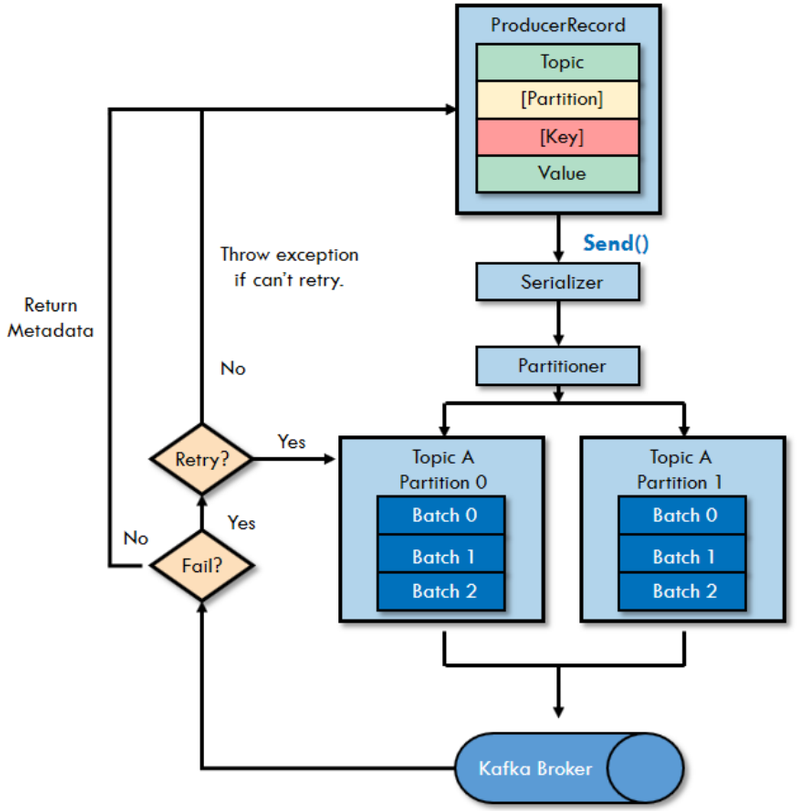


Kafka Producers: Writing Messages to Kafka

101

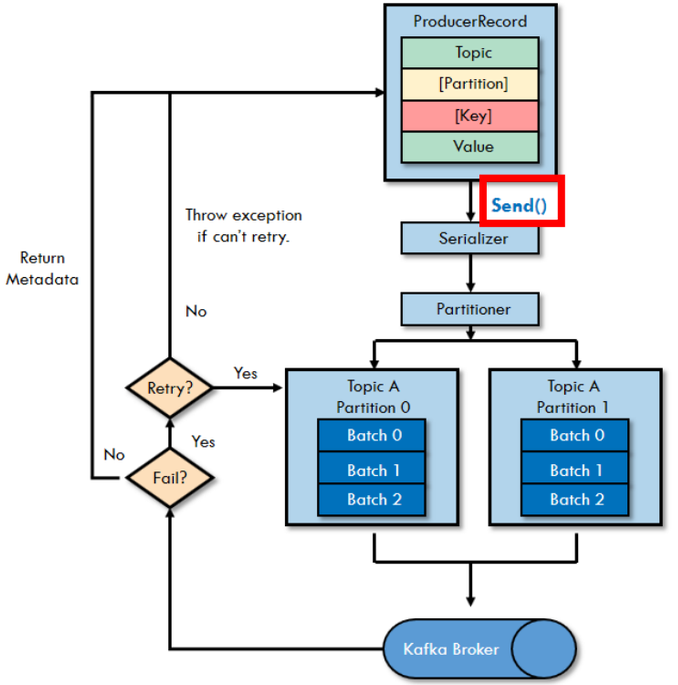
Producer Overview





Three primary methods of sending messages

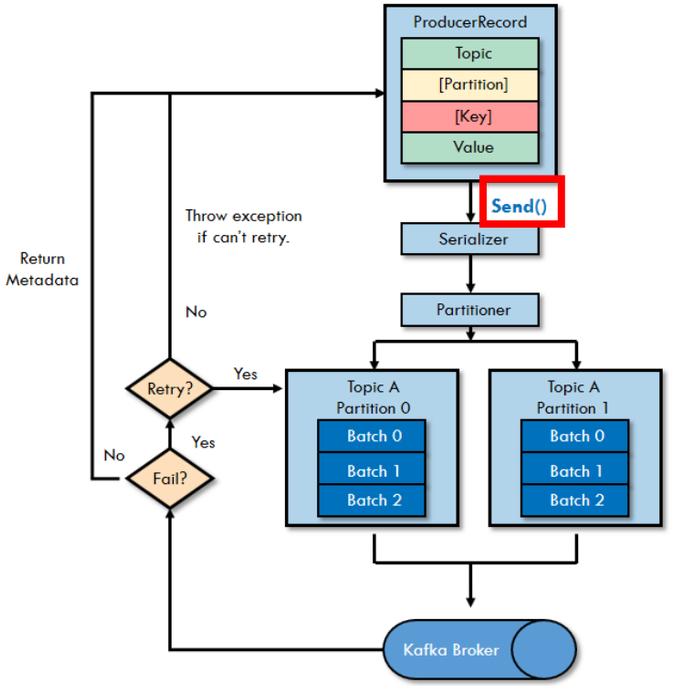
- **Fire-and-forget**
  - We send a message to the server and don't really care if it arrives successfully or not.
- **Synchronous send**
  - We send a message, and wait to see if the **produce()** was successful or not.
- **Asynchronous send**
  - We call the **produce()** method with a **callback function**, which gets triggered when it receives a response from the Kafka broker.



Three primary methods of sending messages (Fire-and-forget)

- Producer_01_FireAndForget

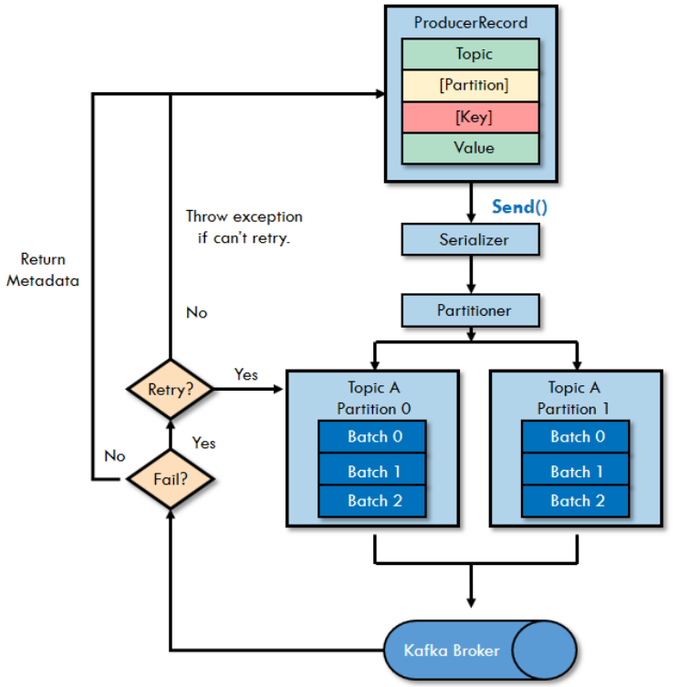
```
# ** 示範: Fire - and -forget **
# 在以下的"produce()"過程，我們並沒有去檢查訊息發佈的結果
# 因此這種方法的throughput最高，但也不知道訊息是否發佈成功或失敗

for i in range(msgCount):
    producer.produce(topicName, key=str(i), value='msg_{}'.format(i))
```



Three primary methods of sending messages (Synchronous send)

- Producer_02_Sync

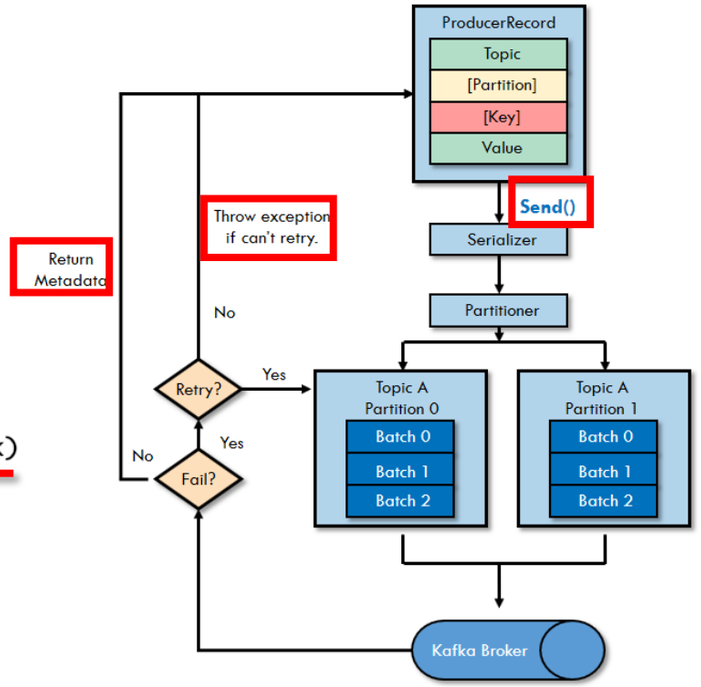
```
# produce(topic, [value], [key], [partition], [on_delivery], [timestamp], [headers])

# ** 示範: Synchronous Send **
# librdkafka(C++)的produce都是用async實作,
# 所以使用confluent_kafka的client作sync發佈的話,
# 只能採produce()搭配flush()實作。
# 由於這種方法是一筆一筆的送。因此這種方法的throughput最差
# (通常不會用Kafka來做這種事)

for i in range(msgCount):
    producer.produce(topicName, key=str(i), value='msg_{}'.format(i))
    producer.flush()    # <-- 每次produce()後就呼叫flush(), 就實作了Sync Producer的功能
    if i % 10000 == 0:
        print('Send {} messages'.format(i))
```



Three primary methods of sending messages (Asynchronous send)

- Producer_03_Async

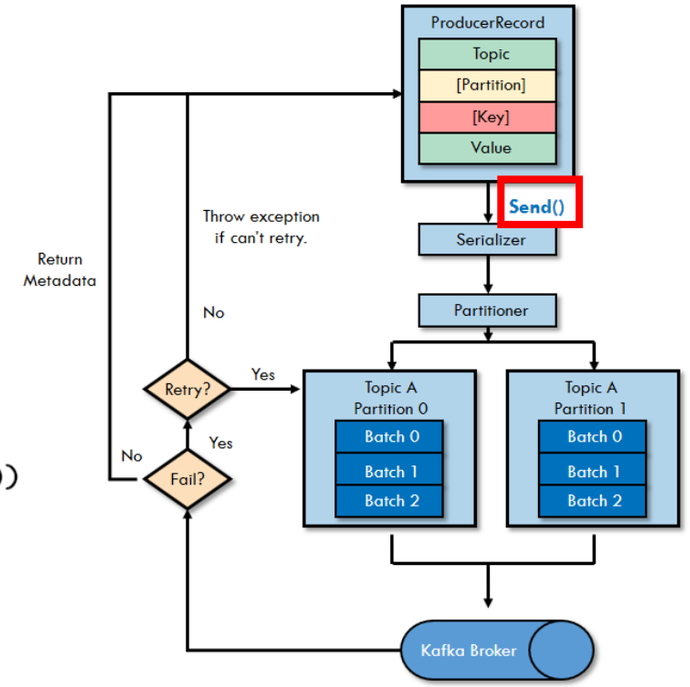
```
'''
// ** 示範: Asynchronous Send **
// 透過一個Callback函式我們可以非同步地取得由Broker回覆訊息發佈的ack結果
// 這種方法可以取得Broker回覆訊息發佈的ack結果，同時又可以取得好的throughput（建議的作法）
'''

for i in range(msgCount):
    producer.produce(topicName, key=str(i), value='msg_'+str(i), callback=delivery_callback)

    producer.poll(0)  # 呼叫poll來讓client程式去檢查內部的Buffer，並觸發callback
```



Three primary methods of sending messages (Asynchronous send)

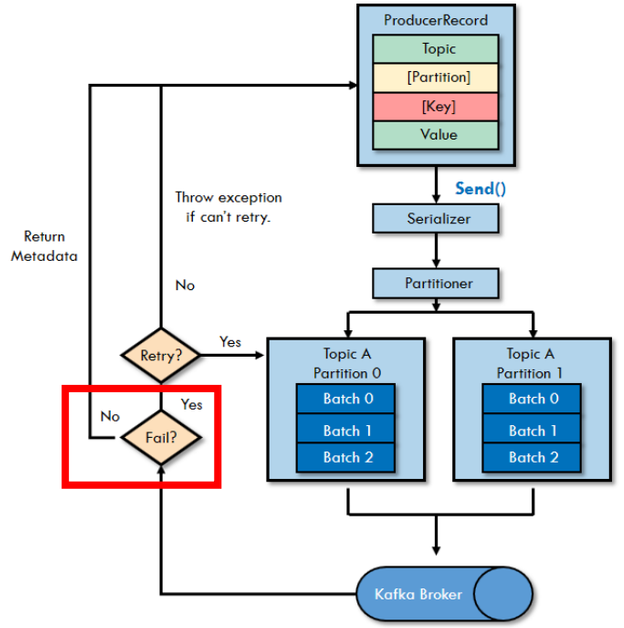
- Producer_03_Async
  - Callback function

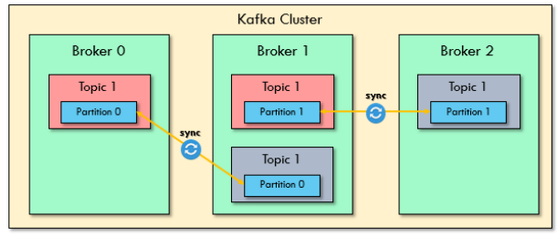
```
def delivery_callback(err, msg):
    if err:
        sys.stderr.write('%% Message failed delivery: %s\n' % err)
    else:
        # 為了不讓打印訊息拖慢速度，我們每1萬打印一筆record Metadata來看
        if int(msg.key()) % 10000 == 0:
            sys.stderr.write('%% Message delivered to topic:[{}]\n'.format(msg.topic()))
```



Configuring Producers

- **acks=0**, the producer will not wait for a reply from the broker before assuming the message was sent successfully.
- **acks=1**, the producer will receive a success response from the broker the moment the leader replica received the message.
- **acks=all**, the producer will receive a success response from the broker once all in-sync replicas received the message.



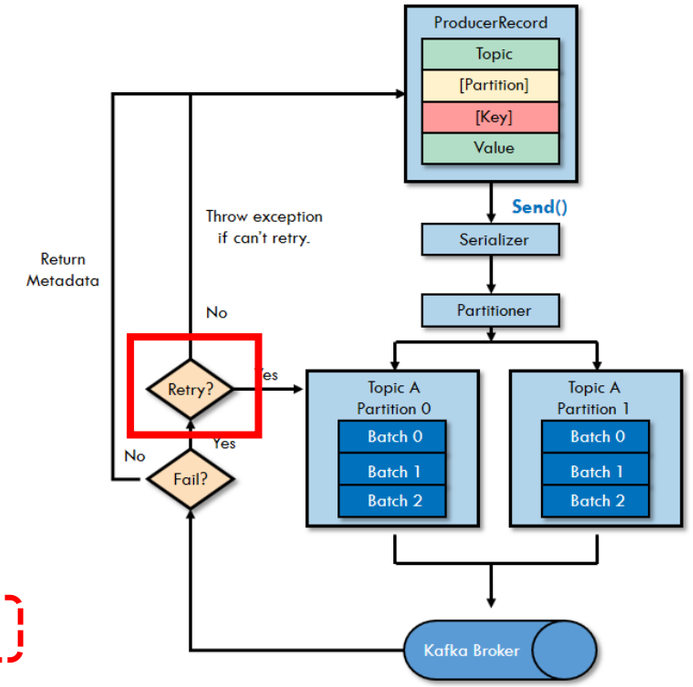



```
props = {
    # Kafka集群在那裡?
    'bootstrap.servers': 'localhost:9092',
    'error_cb': error_cb,
    'acks': "all"
}
```

Configuring Producers

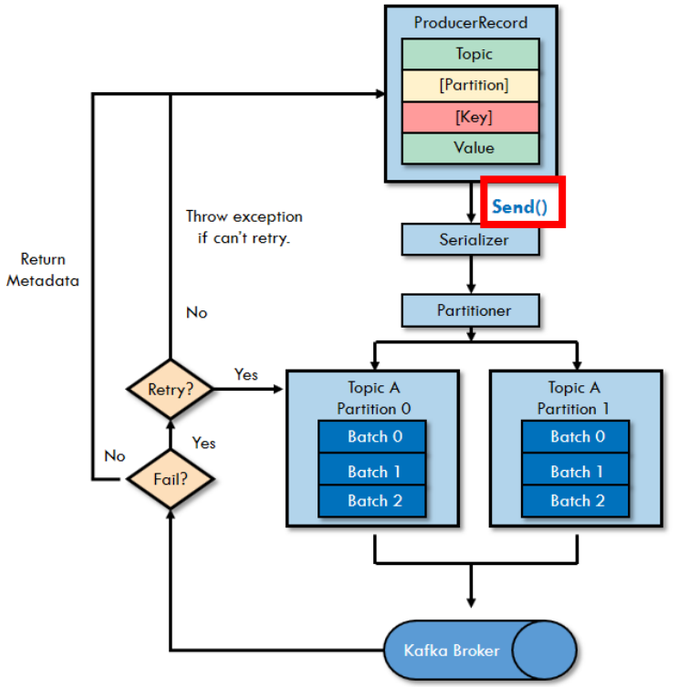
- **retries**
  - control how many times the producer will retry sending the message before giving up and notifying the client of an issue.

```
props = {
    # Kafka集群在那裡?
    'bootstrap.servers': 'localhost:9092',
    'error_cb': error_cb,
    'retries': '10'
  }
```



Configuring Producers

- **max.in.flight.requests.per.connection**
  - controls how many messages the producer will send to the server without receiving responses.
  - Setting this to **1** will guarantee that messages will be written to the broker in the order in which they were sent, even when retries occur.



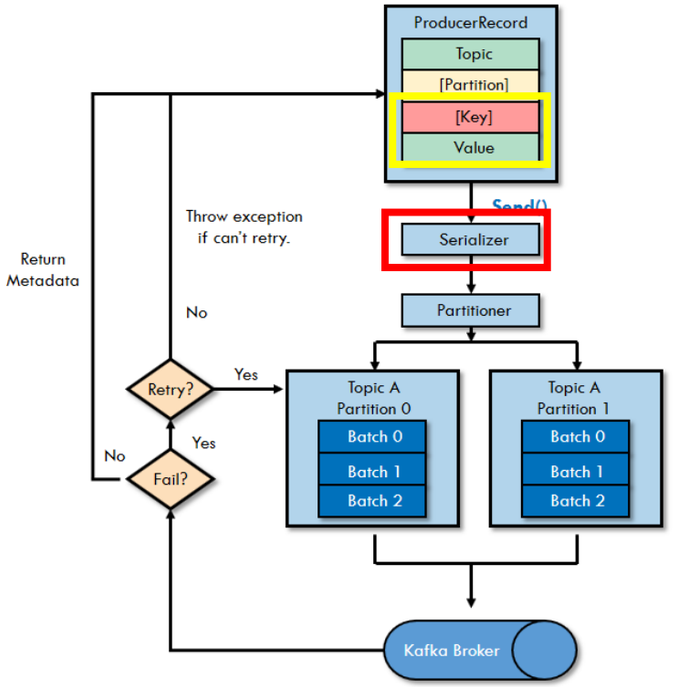

```
props = {
    # Kafka集群在那裡?
    'bootstrap.servers': 'localhost:9092',     # <
    'error_cb': error_cb,                      # 設
    'max.in.flight.requests.per.connection': '1'
}
```

Configuring Producers - Serializers

Kafka lets us publish and subscribe to streams of records and the records can be of any type (JSON, String, POJO, etc.)

JAVA

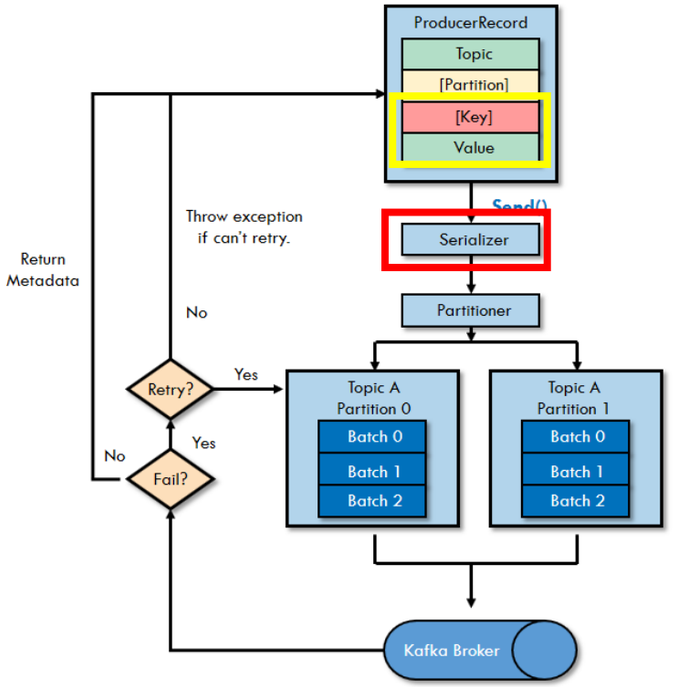
- **key.serializer & value.serializer**
  - **String – StringSerializer**
    - **JSON, XML, Delimited Text, etc..**
  - Integer - IntegerSerializer
  - Long - LongSerializer
  - Double - DoubleSerializer
  - ByteBuffer - ByteBufferSerializer
  - **byte[] – ByteArraySerializer**
    - **Avro, ProtoBuf, etc..**



Configuring Producers - Serializers

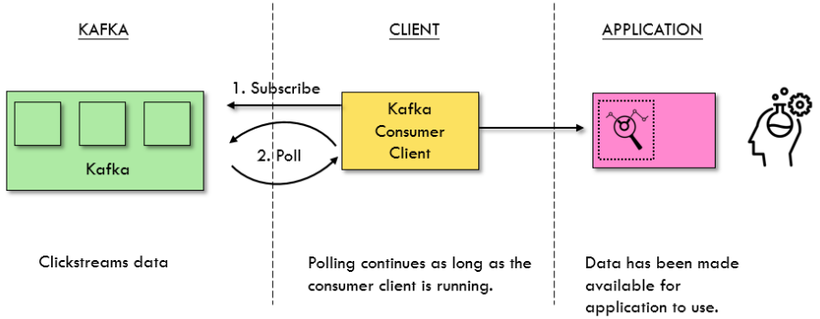
- Producer_04_Json

```
/ ** 示範: 如何將資料包成物件並轉換成JSON字串送入Kafka **
// 由於一般應用場景的資料都是包括了很多的資料欄位及不同的資料型別。通常都會使用一個類別物件來做為資料傳遞的容器。
// 因此如何把一個Data Transfer Object (DTO)序列化送進Kafka是相當關鍵的步驟
'''
for i in range(msgCount):
    fakeNumber = str(i)
    # 讓我們產生假的Employee資料
    employee = Employee(id='emp_id_'+fakeNumber,
                        firstName='fn_'+fakeNumber,
                        lastName='ln_'+fakeNumber,
                        deptId='dept_id_'+str(i % 10),
                        hireDate=epoch_now_mills(),
                        wage=float(i),
                        sex=True
                        )
    # 轉換成JSON字串
    employeeJson = json.dumps(employee.__dict__)
```



Module 10：Consumer Groups

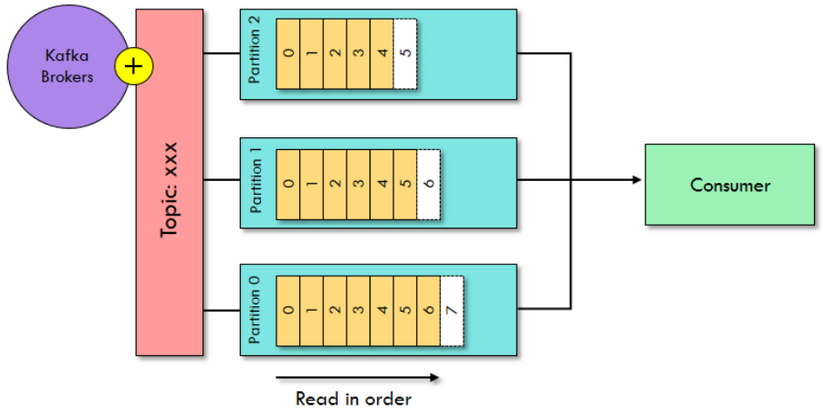
10-1：Groups Coordinator
10-2：Consumer Leader
10-3：Consumer Groups Example



Kafka Consumers: Reading Messages from Kafka

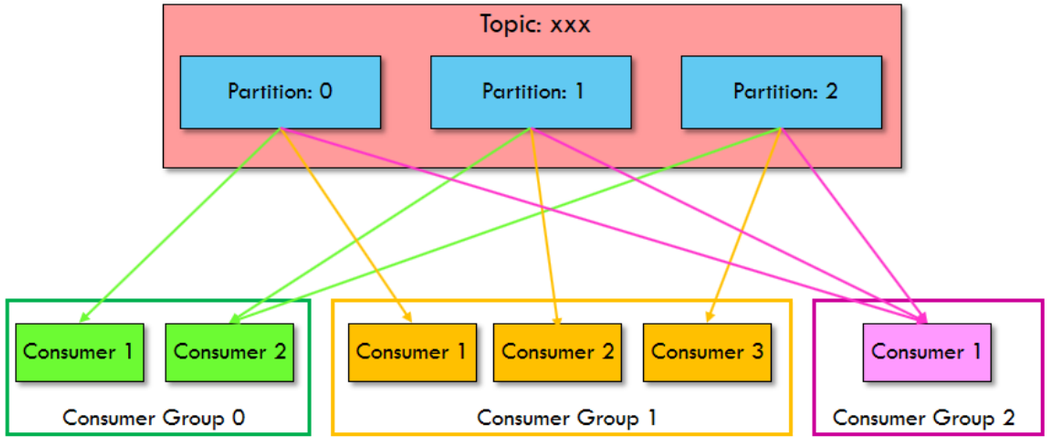
Consumers

- **Consumers** read data from a topic
- They only have to specify the topic name and one broker to connect to, and Kafka will automatically take care of pulling the data from the right brokers
- Data is read in order **for each partitions**



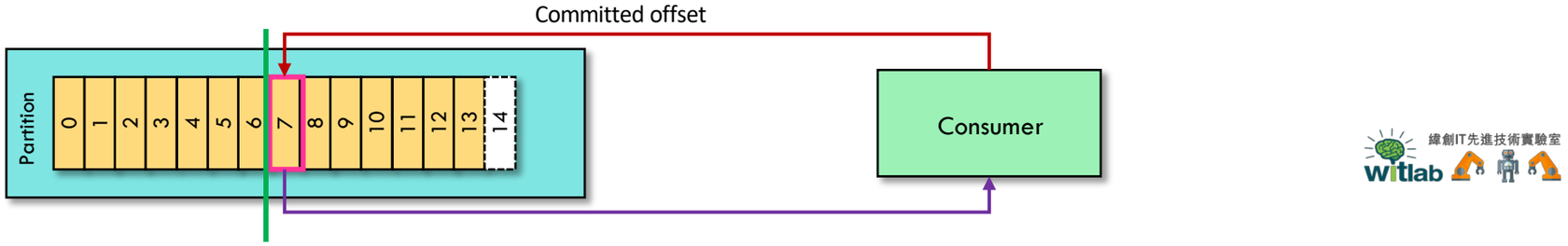
Consumer Groups

- Consumers read data in **consumer groups**
- Each consumer within a group reads from exclusive partitions
- You cannot have more consumers than partitions (otherwise some will be inactive)



Consumer Offsets

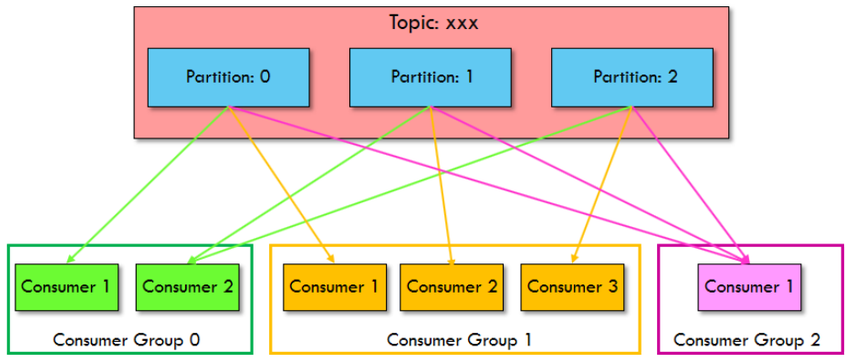
- Kafka stores the offsets at which a **consumer group** has been reading
- The **offsets** commit live in a Kafka topic named "**__consumer_offsets**"
- When a consumer has processed data received some Kafka, it should be **committing** the **offsets**
- If a consumer process dies, it will be able to read back from where it left off thanks to consumer offsets!



Configuring Consumers

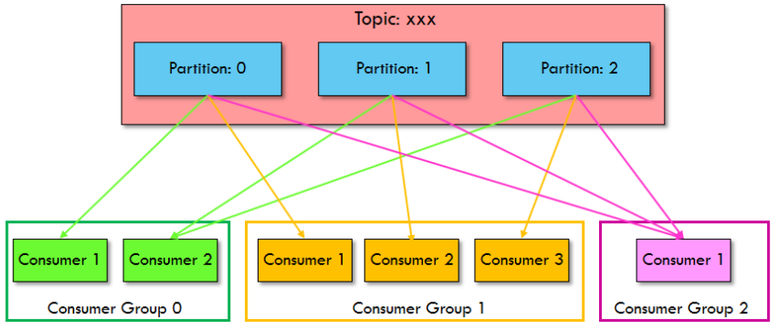
- **group.id**
  - Kafka consumers are typically part of a **consumer group**. When multiple consumers are subscribed to a topic and belong to the same consumer group, each consumer in the group will receive messages from a different subset of the partitions in the topic.

```
props = {
    'bootstrap.servers': 'localhost:9092',
    'group.id': 'iii',
    'auto.offset.reset': 'earliest',
    'error_cb': error_cb
}
```



Consumers and Consumer Groups
Create a topic with 4 partitions

```
$ kafka-topics \
    --create \
    --zookeeper zookeeper:2181 \
    --replication-factor 1 --partitions 4 \
    --topic ak03.four_partition
```

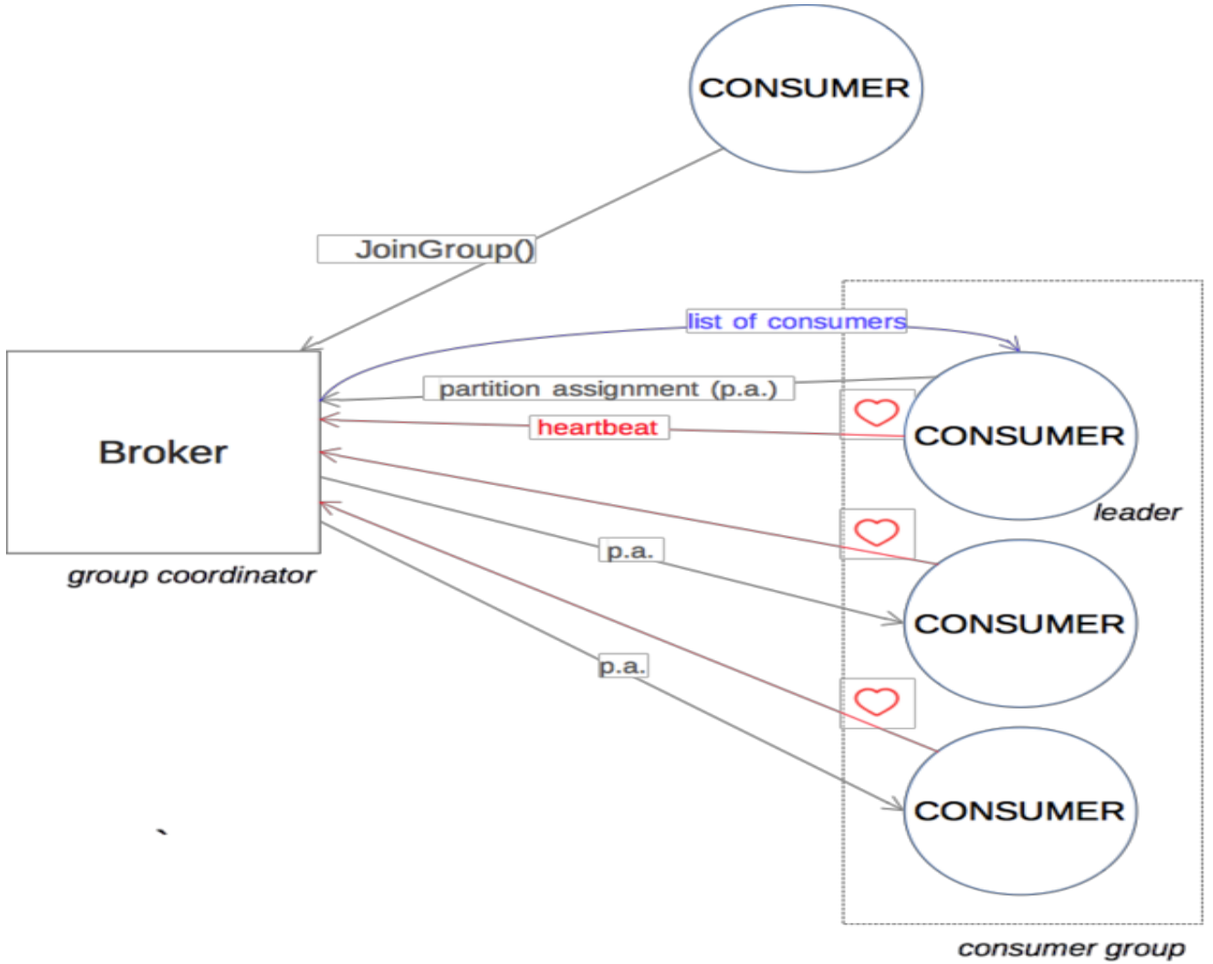



```
root@kafka:/# kafka-topics --create \
> --zookeeper zookeeper:2181 \
> --replication-factor 1 \
> --partitions 4 \
> --topic ak03.four_partition
WARNING: Due to limitations in metric names,
ues it is best to use either, but not both.
Created topic "ak03.four_partition".
```

在container裡頭的zookeeper服務的Hostname

Topic is created!

Consumers and Consumer Groups
ConsumerCoordinator



Consumers and Consumer Groups
Producer_01_CGroup : Publish Event:

```
# produce(topic, [value], [key], [partition], [on_delivery], [timestamp], [heade
for i in range(msgCount):
    producer.produce(topicName, key=str(i), value='msg_'+str(i))
    producer.poll(0)  # <-- (重要) 呼叫poll來讓client程式去檢查內部的Bu
    print('key={}, value={}'.format(str(i), 'msg_' + str(i)))
    time.sleep(3)  # 讓主執行緒停個3秒
```

這個Producer的目的是持續發佈有序列號的訊息來觀察ConsuermGroup的概念!

每3秒發佈一筆訊息 !

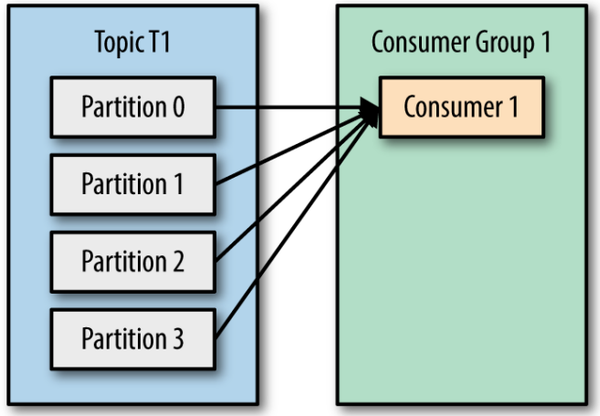
Consumers and Consumer Groups

- Consumer_01_Cgroup
  - One Consumer instance

跑第1個instance, 並觀察log

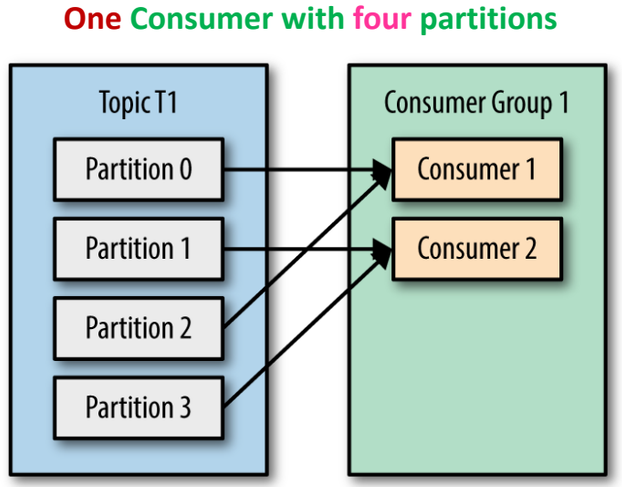
```
python (python3.7)
Setting newly assigned partitions: [ak03.fourpartition-0, ak03.fourpartition-1, ak03.fourpartition-2, ak03.fourpartitio
n-3]
ak03.fourpartition-0-0 : (4 , msg_4)
ak03.fourpartition-0-1 : (6 , msg_6)

ak03.fourpartition-1-0 : (0 , msg_0)
ak03.fourpartition-1-1 : (2 , msg_2)
ak03.fourpartition-2-0 : (5 , msg_5)
ak03.fourpartition-3-0 : (1 , msg_1)
ak03.fourpartition-3-1 : (3 , msg_3)
```

One Consumer with four partitions



Consumers and Consumer Groups

- Consumer_01_Cgroup
  - Two Consumer instance




```
python (python3.7)

Revoking previously assigned partitions: [ak03.fourpartition-0, ak03.fourpartition-1, ak03.fourpartition-2, ak03.fourpa
rtition-3]
ak03.fourpartition-1-5 : (19 , msg_19)

Setting newly assigned partitions: [ak03.fourpartition-2, ak03.fourpartition-3]


ak03.fourpartition-2-4 : (20 , msg_20)
```

```
python (python3.7)
Setting newly assigned partitions: [ak03.fourpartition-0, ak03.fourpartition-1]


ak03.fourpartition-0-4 : (21 , msg_21)

ak03.fourpartition-0-5 : (23 , msg_23)

ak03.fourpartition-1-6 : (25 , msg_25)
```

跑第2個instance, 並觀察log

Consumers and Consumer Groups

- Consumer_01_Cgroup
  - Four Consumer instance

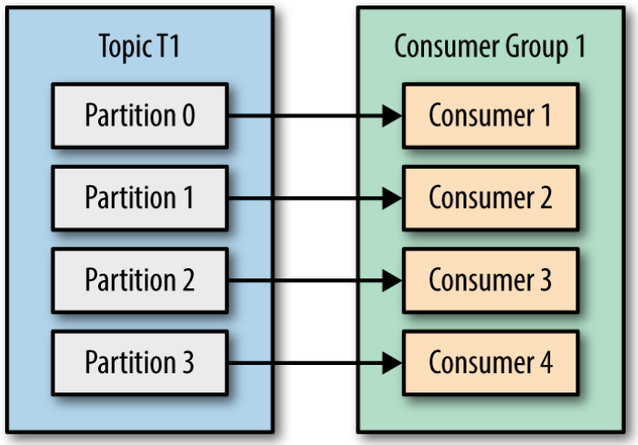
```
python (python3.7)
Revoking previously assigned partitions: [ak03.fourpartition-3]
Setting newly assigned partitions: [ak03.fourpartition-3]

ak03.fourpartition-3-10 : (39 , msg_39)
```

```
python (python3.7)
Revoking previously assigned partitions: [ak03.fourpartition-0, ak03.fourpartition-1]
Setting newly assigned partitions: [ak03.fourpartition-1]

ak03.fourpartition-1-10 : (38 , msg_38)
```

```
python (python3.7)
Revoking previously assigned partitions: [ak03.fourpartition-2]
Setting newly assigned partitions: [ak03.fourpartition-2]

ak03.fourpartition-2-8 : (36 , msg_36)
```

```
python (python3.7)
Setting newly assigned partitions: [ak03.fourpartition-0]

ak03.fourpartition-0-7 : (35 , msg_35)

ak03.fourpartition-0-8 : (37 , msg_37)
```

Three Consumer with four partitions



跑第4個instance, 並觀察log

Consumers and Consumer Groups

- Consumer_01_Cgroup
  - Five Consumer instance

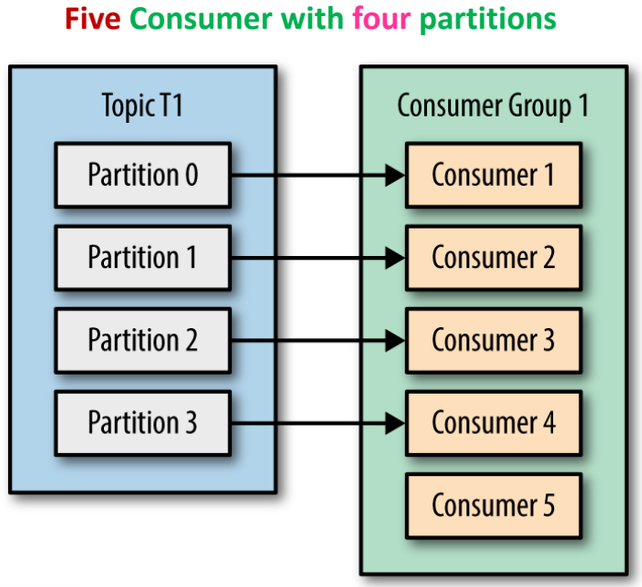
```
python (python3.7)

Revoking previously assigned partitions: [ak03.fourpartition-3]
Setting newly assigned partitions: []
_
```

```
python (python3.7)
Revoking previously assigned partitions: [ak03.fourpartition-1]
Setting newly assigned partitions: [ak03.fourpartition-1]

ak03.fourpartition-1-11 : (44 , msg_44)
```

```
python (python3.7)
Revoking previously assigned partitions: [ak03.fourpartition-2]
Setting newly assigned partitions: [ak03.fourpartition-2]

ak03.fourpartition-2-9 : (41 , msg_41)
```

```
python (python3.7)
Revoking previously assigned partitions: [ak03.fourpartition-0]
Setting newly assigned partitions: [ak03.fourpartition-0]

ak03.fourpartition-0-9 : (40 , msg_40)

ak03.fourpartition-0-10 : (42 , msg_42)
```

```
python (python3.7)
Setting newly assigned partitions: [ak03.fourpartition-3]

ak03.fourpartition-3-11 : (45 , msg_45)

ak03.fourpartition-3-12 : (47 , msg_47)
```



跑第5個instance, 並觀察log

Module 11：Rebalance

11-1：What is Rebalance
11-2：Rebalance Process
11-3： Rebalance Listener

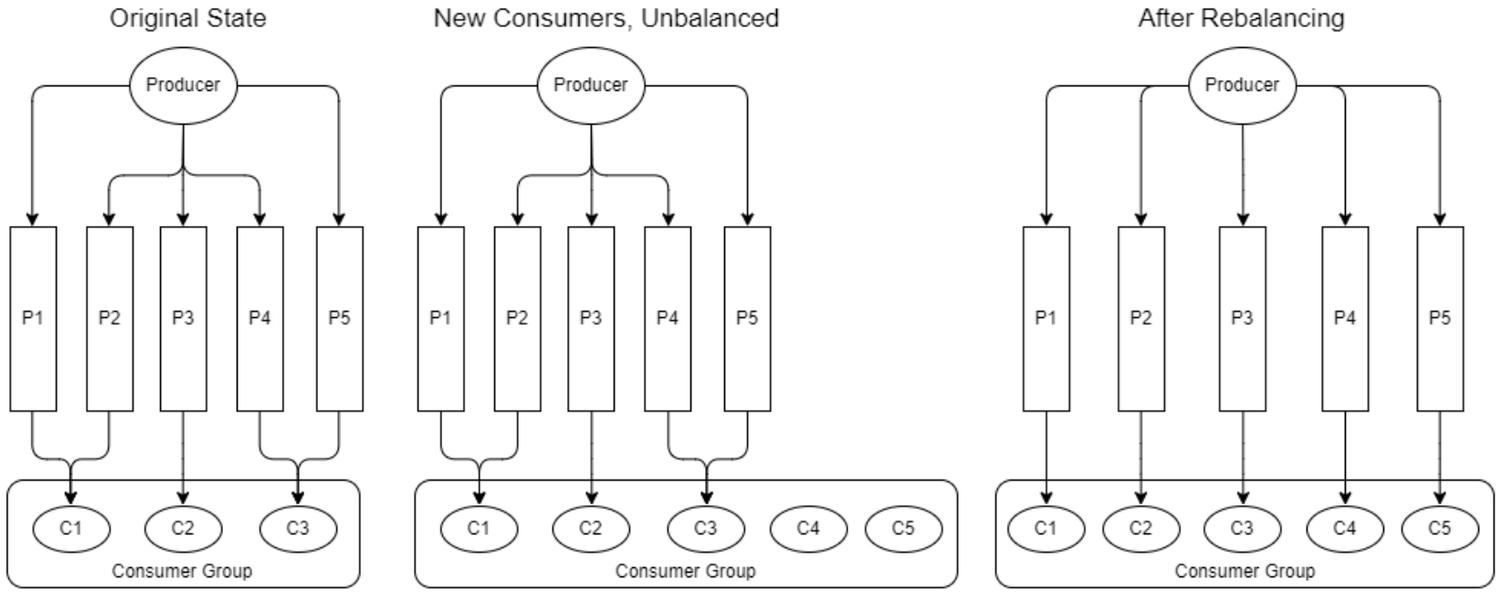
What is Rebalance?

- Every consumer in a **consumer group** is assigned one or more topic partitions exclusively, and **Rebalance** is the re-assignment of partition ownership among consumers.
- A Rebalance happens when:
  - a consumer JOINS the group
  - a consumer SHUTS DOWN cleanly
  - a consumer is considered DEAD by the group coordinator. This may happen after a crash or when the consumer is busy with a long-running processing
  - new partitions are added

Rebalance process

- **Rebalance** can be more or less described as follows:
  - The leader receives a list of all consumers in the group from the group coordinator and is responsible for assigning a subset of partitions to each consumer.
  - After deciding on the partition assignment the group leader sends the list of assignments to the group coordinator, which sends this information to all the consumers.

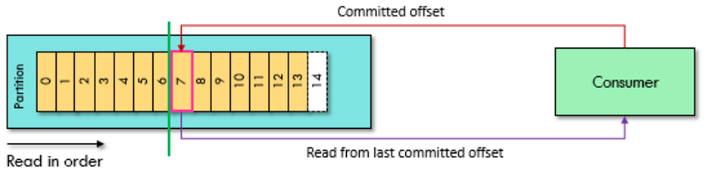
Rebalance process



Module 12：Kafka Consumers

12-1：Automatic Commit
12-2：Synchronous Commit
12-3：Asynchronous Commit

Commits and Offsets



- Whenever we call **consume()**, broker returns records that consumers in our group have not read yet.
- This means that Kafka client tracking which records were read by a consumer of the group.
- We call the action of updating the current position in the partition a **commit**.

```
while True:
    # 請求Kafka把新的訊息吐出來
    records = consumer.consume(num_messages=500, timeout=1.0)
    if records is None:
        continue
```

Commits and Offsets

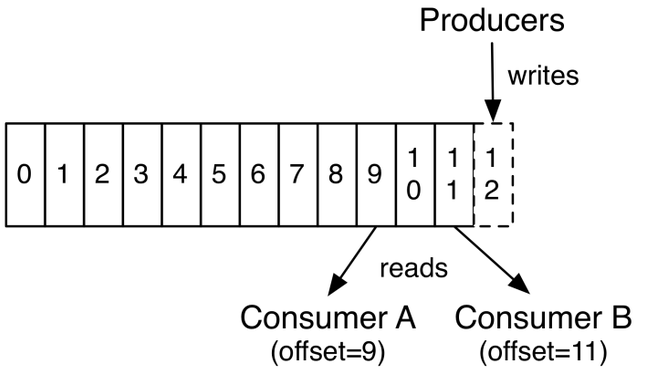
- How does a consumer commit an offset?
- Kafka client produces a message to Kafka, to a special **__consumer_offsets** topic, with the committed offset for each partition.
- If a consumer crashes or a new consumer joins the consumer group, it will trigger a **partition rebalance**.
- After a **rebalance**, each consumer may be assigned a new set of partitions than the one it processed before.
- The consumer will read the latest committed offset of each partition and continue from there.

標題

- Managing offsets has a big impact on the client application.
- The Kafka Consumer API provides multiple ways of committing offsets:
  - Automatic Commit
  - Synchronous Commit (previous **consume()** offset)
  - Asynchronous Commit (previous **consume()** offset)
  - Combining Synchronous and Asynchronous Commits (previous **consume()** offset)
  - Commit Specified Offset

Configuring Consumers

- **auto.offset.reset**
  - This property controls the behavior of the consumer when it starts reading a partition for which it doesn't have a committed offset.
  - The default is "**latest**," which means that lacking a valid offset, the consumer will start reading from the newest records
  - The alternative is "**earliest**," which means that lacking a valid offset, the consumer will read all the data in the partition, starting from the very beginning.



Commits and Offsets: Automatic Commit

- Configure below configurations:
  - **enable.auto.commit=true**
  - **auto.commit.interval.ms=5000 (default)**
- Every **five** seconds the consumer will commit the largest offset your client received from consume().
- Whenever you consume(), the consumer checks if it is time to commit, and if it is, it will commit the offsets it returned in the last consume().

Commits and Offsets: Automatic Commit

- Consumer_02_AutoCommit

```
props = {
    'bootstrap.servers': 'localhost:9092',
    'group.id': 'iii',
    'auto.offset.reset': 'earliest',
    'enable.auto.commit': True,
    'auto.commit.interval.ms': 5000,
    'on_commit': print_commit_result,
    'error_cb': error_cb
}
```

```
while True:
    # 請求Kafka把新的訊息吐出來
    records = consumer.consume(num_messages=500, timeout=1.0)  # 批次讀取
    if records is None:
        continue
```

```
ak03.test-0-1 : (1 , msg_1)
ak03.test-0-2 : (2 , msg_2)
# Committed offsets for: ak03.test-0 {offset=3}
ak03.test-0-3 : (3 , msg_3)
ak03.test-0-4 : (4 , msg_4)
# Committed offsets for: ak03.test-0 {offset=5}
ak03.test-0-5 : (5 , msg_5)
```

Commits and Offsets: Automatic Commit

- With **auto-commit** enabled, a call to consume() *will always commit the last offset returned by the previous consume()*. It doesn't know which events were actually processed, so it is critical to always process all the events returned by consume() before calling consume() again.
- Automatic commits are convenient, but they don't give developers enough control to avoid duplicate messages.

Commits and Offsets: Manual Commit

- The consumer API has the option of committing the current offset at a point that makes sense to the application developer rather than based on a timer.
- Configure below configurations:
  - **auto.commit.offset=false**
- Offsets will only be committed when the application explicitly chooses to do so.
- The simplest and most reliable commit method is Synchronous Commit. With confluent-kafka lib, **commit(asynchronous=False)**, default value for asynchronous is True, represents synchronous commit,

Commits and Offsets: Sync Commit

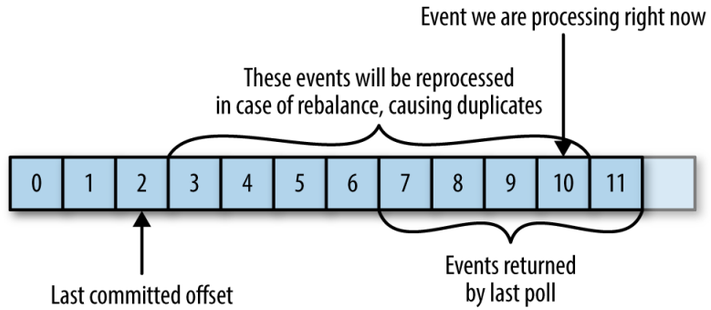
- Consumer_03_CommitSync

```
props = {
    'bootstrap.servers': 'localhost:9092',
    'group.id': 'iii',
    'auto.offset.reset': 'earliest',
    'enable.auto.commit': False,
    'error_cb': error_cb
}
```

```
while True:
    recrods_pulled = False  # 用來檢查是否有有效的record被取出來

    # 請求Kafka把新的訊息吐出來
    records = consumer.consume(num_messages=500, timeout=1.0)  # 批次讀取
    if records is None:
        continue

    for record in records:
        # 檢查是否有錯誤
        if record is None:
            continue
        if record.error():...
        else:
            recrods_pulled = True
            # ** 在這裡進行商業邏輯與訊息處理 **
            # 取出相關的metadata
            topic = record.topic()
            partition = record.partition()
            offset = record.offset()
            timestamp = record.timestamp()
            # 取出msgKey與msgValue
            msgKey = try_decode_utf8(record.key())
            msgValue = try_decode_utf8(record.value())

            # 秀出metadata與msgKey & msgValue訊息
            print('%s-%d-%d : (%s , %s)' % (topic, partition, offset, msgKey, msgValue))

    # 同步地執行commit (Sync commit)
    if recrods_pulled:
        offsets = consumer.commit(asynchronous=False)
```

Commits and Offsets: Sync Commit

- This **commit(asynchronous=False)** will commit the latest offset returned by **consume()** and return once the offset is committed, throwing an exception if commit fails for some reason.
- Make sure you call **commit(asynchronous=False)** after you are done processing all the records in the collection.
- When **rebalance** is triggered, all the messages from the beginning of the most recent batch until the time of the rebalance will be processed twice.



Commits and Offsets: Asynchronous Commit

- One drawback of manual commit is that the application is ***blocked*** until the broker responds to the commit request.
- Another option is the **commit(asynchronous=True)**. Since default value is True, you can just call **commit()**. Instead of waiting for the broker to respond to a commit, we just send the request and continue on:

Commits and Offsets: Asynchronous Commit

- Consumer_04_CommitAsync

```
props = {
    'bootstrap.servers': 'localhost:9092',
    'group.id': 'iii',
    'auto.offset.reset': 'earliest',
    'enable.auto.commit': False,
    'on_commit': print_commit_result,
    'error_cb': error_cb
}
```

```
while True:
    recrods_pulled = False # 用來檢查是否有有效的record被取出來
    # 請求Kafka把新的訊息吐出來
    records = consumer.consume(num_messages=500, timeout=1.0)  # 批次讀取
    if records is None:
        continue
    for record in records:
        # 檢查是否有錯誤
        if record is None:
            continue
        if record.error():...
        else:
            recrods_pulled = True
            # ** 在這裡進行商業邏輯與訊息處理 **
            # 取出相關的metadata
            topic = record.topic()
            partition = record.partition()
            offset = record.offset()
            timestamp = record.timestamp()
            # 取出msgKey與msgValue
            msgKey = try_decode_utf8(record.key())
            msgValue = try_decode_utf8(record.value())

            # 秀出metadata與msgKey & msgValue訊息
            print('%s-%d-%d : (%s , %s)' % (topic, partition, offset, msgKey, msgValue))
    # 異步地執行commit (Async commit)
    if recrods_pulled:
        consumer.commit()
```

Commits and Offsets: Asynchronous Commit

- **Synchronous commit** will retry the commit until it either succeeds or encounters a non-retriable failure.
- **Asynchronous commit** will not retry.
- **Asynchronous commit** gives you an option to pass in a callback that will be triggered when the broker responds.

```
def print_commit_result(err, partitions):
    if err is not None:
        print('# Failed to commit offsets: %s: %s' % (err, partitions))
    else:
        for p in partitions:
            print('# Committed offsets for: %s-%s {offset=%s}' % (p.topic, p.partition, p.offset))
```

Commits and Offsets:
Combining Sync and Async Commits

- Consumer_05_CommitSyncAsync

```
        while True:
            recrods_pulled = False # 用來檢查是否有有效的record被取出來
            # 請求Kafka把新的訊息吐出來
            records = consumer.consume(num_messages=500, timeout=1.0)  # 批次讀取
            if records is None: continue
            for record in records:
                if record is None: continue
                if record.error():...
                else:
                    recrods_pulled = True
                    topic = record.topic()
                    partition = record.partition()
                    offset = record.offset()
                    timestamp = record.timestamp()
                    msgKey = try_decode_utf8(record.key())
                    msgValue = try_decode_utf8(record.value())
                    print('%s-%d-%d : (%s , %s)' % (topic, partition, offset, msgKey, msgValue))
            # 異步地執行commit (Async commit)
            if recrods_pulled:
                consumer.commit()
    except KeyboardInterrupt as e:
        sys.stderr.write('Aborted by user\n')
    except Exception as e:
        sys.stderr.write(str(e))

    finally:
        # 步驟6.關掉Consumer實例的連線
        consumer.commit(asynchronous=False)
        consumer.close()
```

Commits and Offsets: Commit Specified Offset

- Committing the latest offset only allows you to commit as often as you finish processing batches.
- What if **consume()** returns a huge batch and you want to commit offsets in the middle of the batch to avoid having to process all those rows again if a rebalance occurs?
- You can’t just call **commit()** or **commit(asynchronous=False)** — this will commit the last offset returned, which you didn’t get to process yet.
- Consumer API allows you to call **commit()** and **commit(asynchronous=False)** and pass a map of **partitions** and **offsets** that you wish to commit.

Commits and Offsets: Commit Specified Offset

- Consumer_06_CommitSpecified

```
while True:
    recrods_pulled = False  # 用來檢查是否有有效的record被取出來
    # 請求Kafka把新的訊息吐出來
    records = consumer.consume(num_messages=500, timeout=1.0)  # 批次讀取
    if records is None:
        continue
    for record in records:
        # 檢查是否有錯誤
        if record is None:
            continue
        if record.error():...
        else:
            # ** 在這裡進行商業邏輯與訊息處理 **
            # 取出相關的metadata
            topic = record.topic()
            partition = record.partition()
            offset = record.offset()
            timestamp = record.timestamp()
            # 取出msgKey與msgValue
            msgKey = try_decode_utf8(record.key())
            msgValue = try_decode_utf8(record.value())

            # 秀出metadata與msgKey & msgValue訊息
            print('%s-%d-%d : (%s , %s)' % (topic, partition, offset, msgKey, msgValue))

            consumer.commit(record)  # 非同步的commit
```

Rebalance Listeners

- If you know your consumer is about to lose ownership of a partition, you will want to commit offsets of the last event you've processed.
- The consumer API allows you to run your own code when partitions are added or removed from the consumer.
- You do this by passing on_revoke and on_assign when calling the subscribe() method

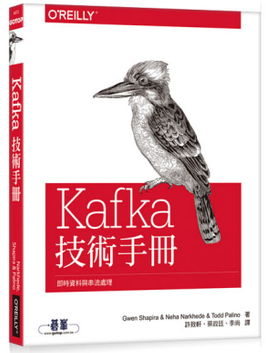
Commits and Offsets: RebalanceListener

- Consumer_07_CommitSpecified

```python
# 當發生Re-balance時, 如果有partition被assign時被呼叫
def print_assignment(consumer, partitions):
    result = '[{}]'.format(','.join([p.topic + '-' + str(p.partition) for p in partitions]))
    print('Setting newly assigned partitions:', result)


# 當Re-balance被觸發後, Commit現在process的offets
def commit_on_revoke(consumer, partitions):
    global offsets_dict
    for k, v in offsets_dict.items():
        consumer.commit(v)  # 進行異步的commit

    result = '[{}]'.format(','.join([p.topic + '-' + str(p.partition) for p in partitions]))
    print('Revoking previously assigned partitions: ' + result)


# 當發生commit時被呼叫
def print_commit_result(err, partitions):...


if __name__ == '__main__':
    ...
    props = {...}

    # 步驟2. 產生一個Kafka的Consumer的實例
    consumer = Consumer(props)
    # 步驟3. 指定想要訂閱訊息的topic名稱
    topicName = 'ak03.four_partition'
    # 步驟4. 讓Consumer向Kafka集群訂閱指定的topic
    consumer.subscribe([topicName], on_assign=print_assignment, on_revoke=commit_on_revoke)
```

Reference

1. Kafka 技術手冊｜即時資料與串流處理 (Kafka: The Definitive Guide) – O’REILLY



2. Kafka In Action - MANNING

3. Learn Apache Kafka for Beginners – Udemy (Stephane Maarek)

4. Confluent Document of Kafka – confluent.io

confluent

36