



# Confluent Developer

## Skills for Building

## Apache Kafka®

## Student Handbook

Version 5.3.0-v1.0.1

# Table of Contents

---

01 Introduction . . . . .	1
02 Fundamentals of Apache Kafka . . . . .	9
03 Kafka's Architecture . . . . .	31
04 Developing With Kafka . . . . .	67
05 More Advanced Kafka Development . . . . .	116
06 Schema Management In Kafka . . . . .	170
07 Data Pipelines with Kafka Connect . . . . .	209
08 Stream Processing with Kafka Streams . . . . .	262
09 Stream Processing with KSQL . . . . .	309
10 Event Driven Architecture . . . . .	335
11 Confluent Cloud . . . . .	364
12 Conclusion . . . . .	396
Appendix A: Basic Kafka Administration . . . . .	401
Appendix B: Apache Kafka driving ML & Data Analytics . . . . .	433

ybhandare@greendotcom.com



### Introduction

ybhandare@greendotcorp.com

Copyright © Confluent, Inc. 2014-2019. [Privacy Policy](#) | [Terms & Conditions](#).

Apache, Apache Kafka, Kafka and the Kafka logo are trademarks of the  
[Apache Software Foundation](#)

ybhandare@greendotcorp.com

# Agenda

---



1. Introduction ... ←
2. Fundamentals of Apache Kafka
3. Kafka's Architecture
4. Developing With Kafka
5. More Advanced Kafka Development
6. Schema Management In Kafka
7. Data Pipelines with Kafka Connect
8. Stream Processing with Kafka Streams
9. Stream Processing with KSQL
10. Event Driven Architecture
11. Confluent Cloud
12. Conclusion
13. Appendix: Basic Kafka Administration
14. Appendix: Apache Kafka driving ML & Data Analytics

ybhandare@greendotcorp.com

## Prerequisite

---

This course requires a working knowledge of the Kafka architecture.

New to Kafka? Need a refresher?

Sign up for free "Confluent Fundamentals for Apache Kafka" course at <https://confluent.io/training>

Attendees should have a working knowledge of the Kafka architecture, either from prior experience or the recommended prerequisite course Confluent Fundamentals for Apache Kafka®.

This free course is available at <http://confluent.io/training> for students who need to catch up.

ybhandare@greendotcorp.com

## Course Objectives

---

After this course, you will be able to:

- Write Producers and Consumers to send data to & read data from Kafka
- Integrate Kafka with external systems using Kafka Connect and MQTT
- Write streaming applications with Kafka Streams & KSQL
- Write event driven, semantic applications that leverage Kafka
- Use and integrate with Confluent Cloud

Throughout the course, Hands-On Exercises will reinforce the topics being discussed

ybhandare@greendotcorp.com

## Class Logistics

---



- Start and end times
- Can I come in early/stay late?
- Breaks
- Lunch
- Restrooms
- Wi-Fi and other information
- Emergency procedures



No video & recording please!

ybhandare@greendotcorp.com

## How to get the courseware?

---



# Introductions

---



- About you:
  - Your Name, Company and Role
  - Your Experience with Kafka
  - Other Messaging or Big Data Systems, you use
  - OS, Programming Languages
  - Your Course Expectations
- About your instructor

ybhandare@greendotcorp.com



### Fundamentals of Apache Kafka

ybhandare@greendotcorp.com

# Agenda

---



1. Introduction
2. Fundamentals of Apache Kafka ... ←
3. Architecture of Apache Kafka
4. Developing with Apache Kafka
5. Advanced Development with Apache Kafka
6. Schema Management in Apache Kafka
7. Data Pipelines with Kafka Connect
8. Stream Processing with Kafka Streams
9. Stream Processing with KSQL
10. Event Driven Architecture
11. Confluent Cloud
12. Conclusion
13. Appendix: Basic Kafka Administration
14. Appendix: Apache Kafka driving ML & Data Analytics

ybhandare@greendotcorp.com

## Learning Objectives

---

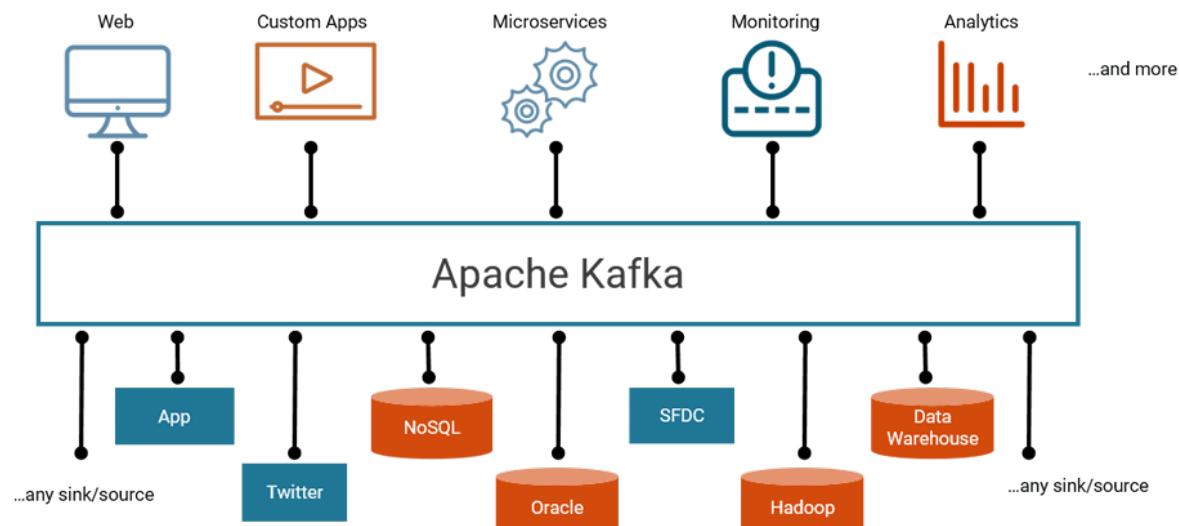


After this module you will be able to:

- explain how Kafka powers the **Distributed Streaming Platform**
- describe the main characteristics of a **log**
- sketch the high level architecture of a Kafka producer
- list key characteristics of a Kafka consumer group

ybhandare@greendotcorp.com

# The Streaming Platform



Apache Kafka is the foundation of what we call a **Distributed Streaming Platform**. Kafka is used for building real-time data pipelines and streaming apps. It is horizontally scalable, fault-tolerant, wicked fast, and runs in production in thousands of companies.

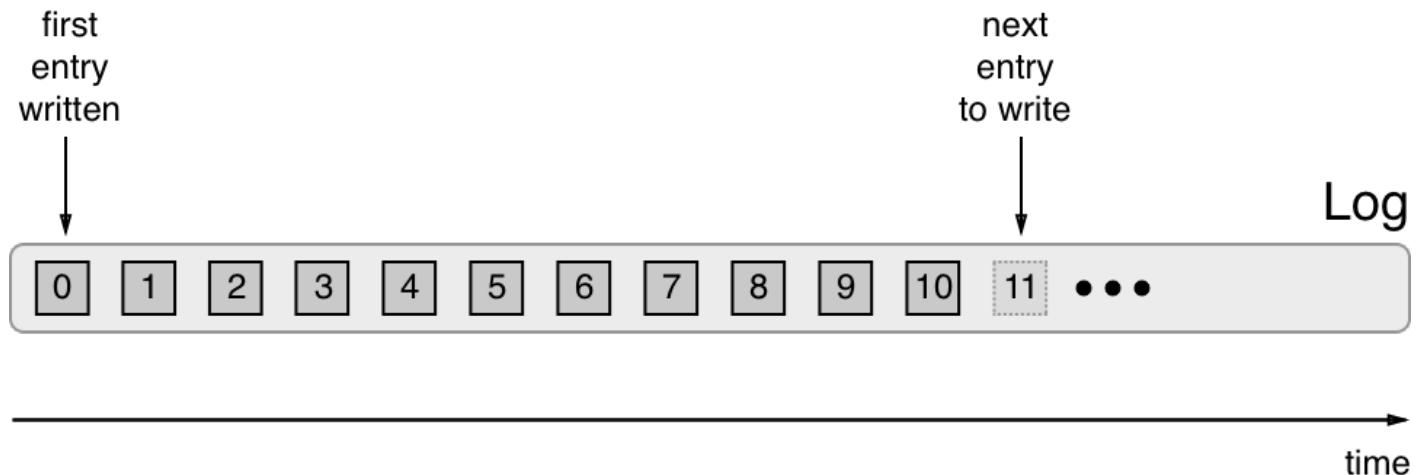
A streaming platform has three key capabilities:

- Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.
- Store streams of records in a fault-tolerant durable way.
- Process streams of records as they occur, in real-time.

Kafka is generally used for two broad classes of applications:

- Building real-time streaming data pipelines that reliably get data between systems or applications
- Building real-time streaming applications that transform or react to the streams of data

## The Commit Log



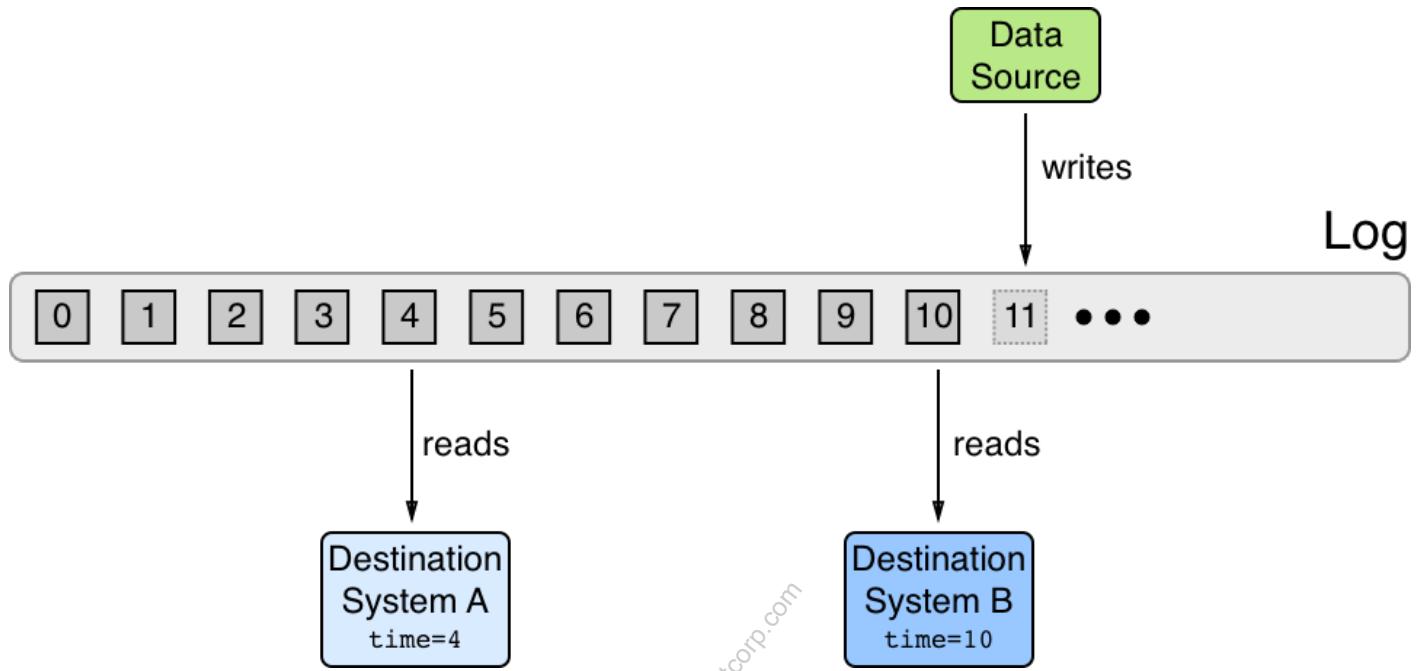
To understand Kafka as a **Streaming Platform**, it is important that we first discuss a few basics. One central element that enables Kafka and stream processing is the so called "log"; in the context of Kafka it is also called the "commit log".

A log is a data structure that is like a stack of elements. New elements are always appended at the end of the stack, and once written, they are never changed. In this regard one talks of an append only, write once data structure.

Elements that are added to the log are strictly ordered in time. The first element added to the stack is older than the second one, which in turn is older than the third one.

In the image the time axis reflects that fact. The index of the elements, in that sense, can be viewed as a time scale.

## Log Structured Data Flow



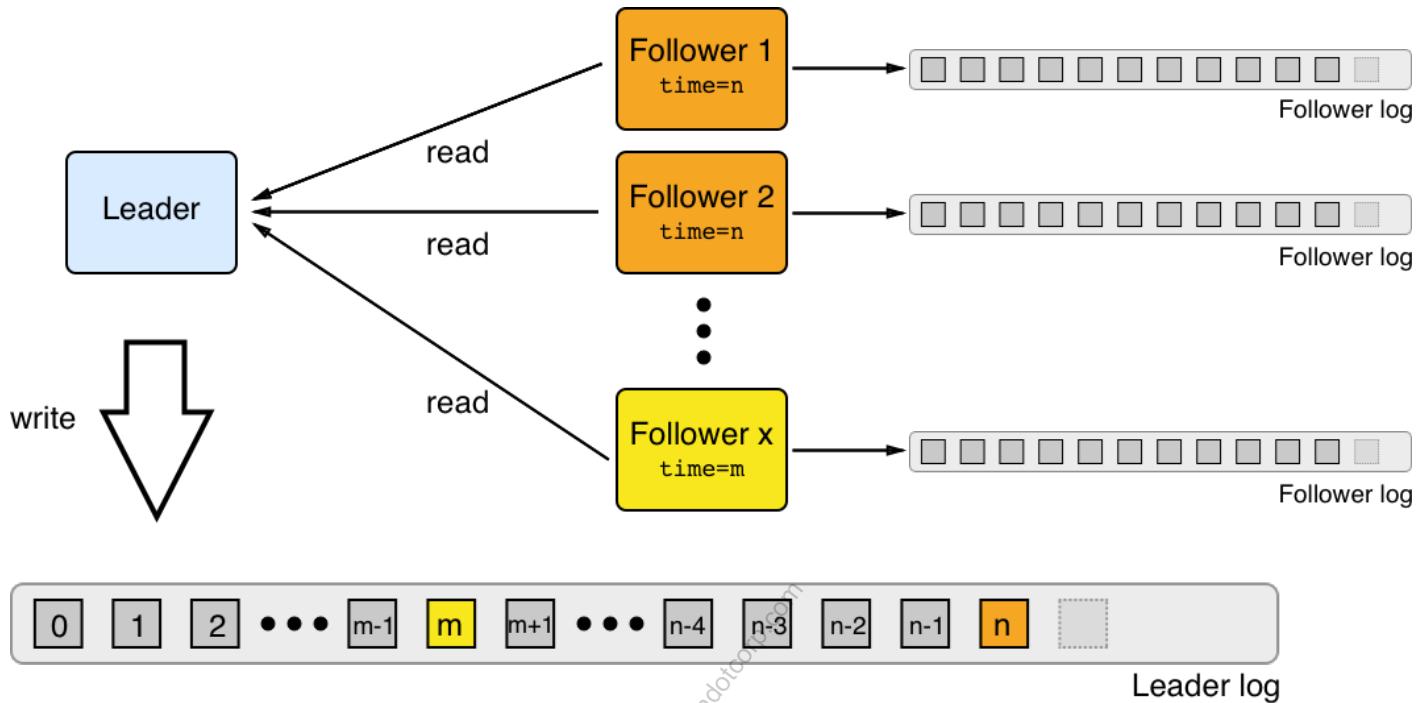
The log is produced by some data source. In the image the data source has already produced elements with index 0 to 10, and the next element the source produces will be written at index 11.

The data source can write at its own speed since it is totally decoupled from any destination system. In fact, the source system does not know anything about the consuming applications.

Multiple destination systems can independently consume from the log, each at its own speed. In the sample we have two consumers that read from different positions at the same time. This is totally fine and an expected behavior.

Each destination system consumes the elements from the log in temporal order, that is from left to right in our image.

# Log Replication



For reliability reasons one wants to keep around more than one copy of the log. This process is called replication. With a log this is straight forward to implement.

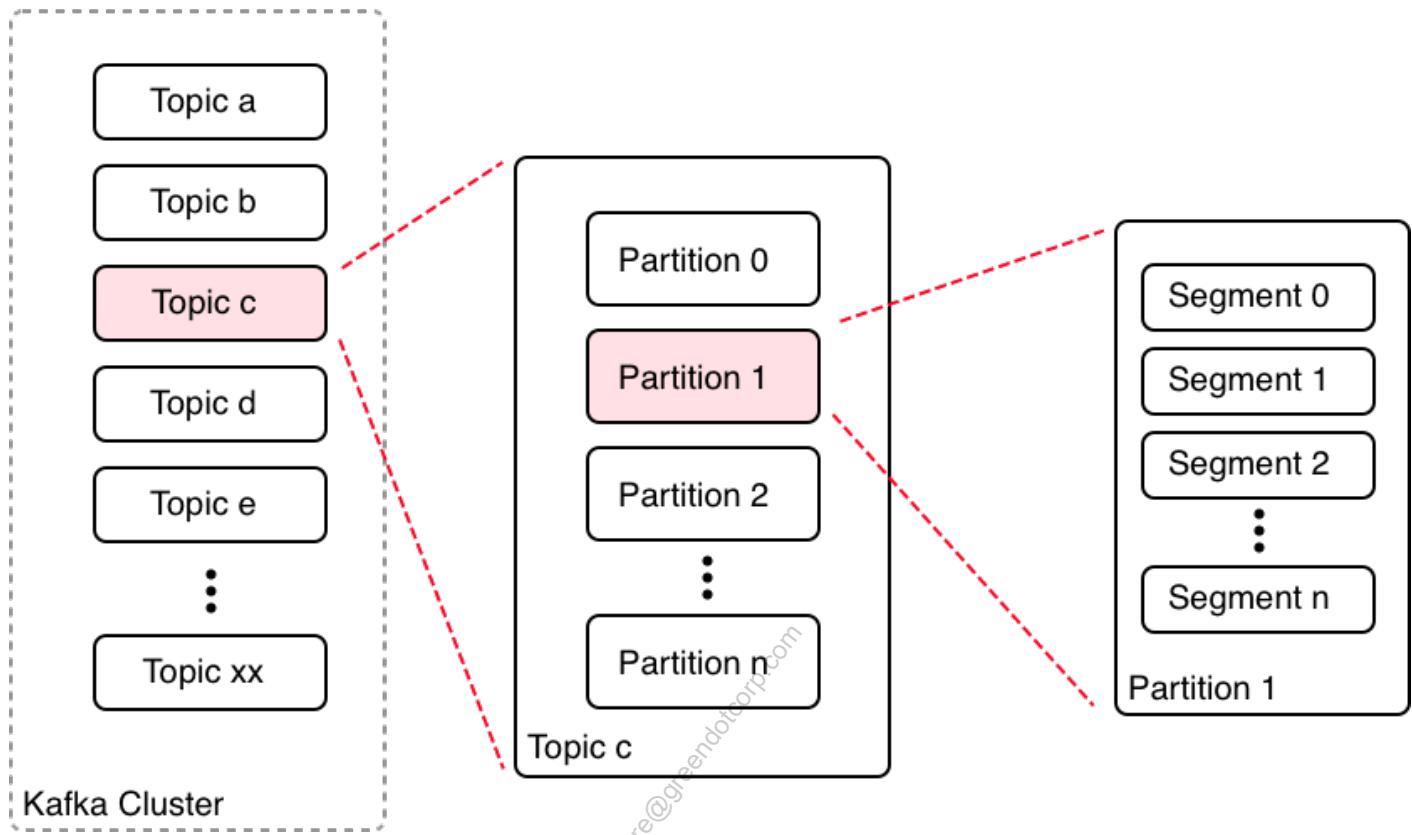
One possible way of doing a log replication is to define a (single) leader and a number of followers (Note: this is what Kafka does). The leader writes and keeps the principal copy of the log. The followers read from the leader's log (managed by the leader) and update their own local log accordingly, as shown in the image.

The followers are up to date with the leader if their last offset (or time) in their private log corresponds to the last offset of the leader's log. A follower is behind, or **out of sync** if its offset is behind the last offset of the leader. By default, for Kafka this is true, when a follower is more than 10s behind the end of the leader's log. Adjust the parameter `replica.lag.time.max.ms` to change this default.

The followers whose log is up to date with the leader are called **in-sync replicas**. Any of those followers can take over the role of a leader if the current leader fails.

Only the leader writes data originating from a data source to the log. The followers only ever read from the leader.

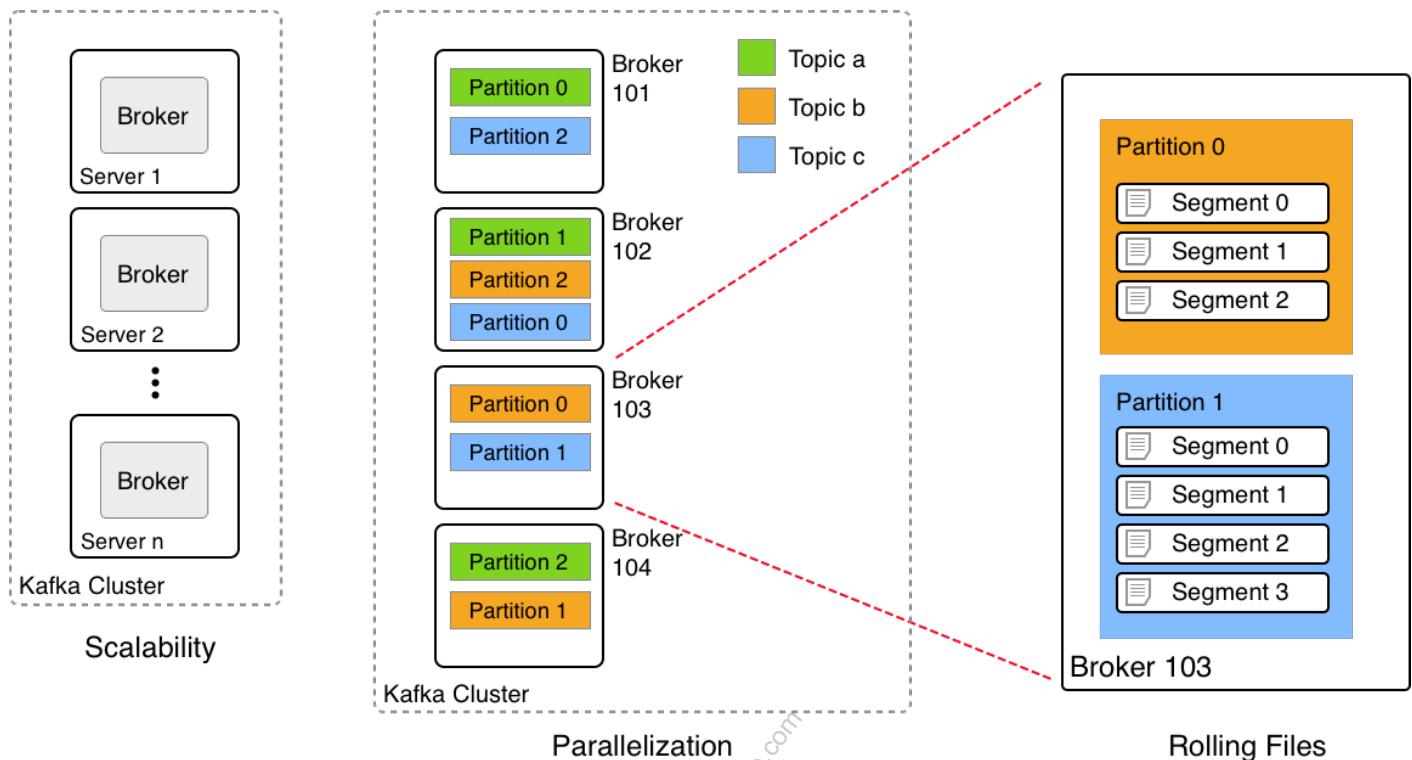
### Logical View



Let's now transfer what we have learned about the log to Kafka. In Kafka we have the three terms **Topic**, **Partition** and **Segment**. Let's review each of them:

- **Topic:** A topic comprises all messages of a given category. We could e.g. have the topic "temperature\_readings" which would contain all messages that contain a temperature reading from one of the many measurement stations the company has around the globe.
- **Partition:** To parallelize work and thus increase the throughput Kafka can split a single topic into many partitions. The messages of the topic will then be split between the partitions. The default algorithm used to decide to which topic partition a message goes uses the hash code of the message key. A partition is handled in its entirety by a single Kafka broker. A partition can be viewed as a "log".
- **Segment:** The broker stores the messages as they come in in a physical file. Since the data can potentially be endless the broker is using a "rolling-file" strategy. It creates/allocates a new file and fills it with messages. When the segment is full (or the given max. time per segment expires), the next one is allocated by the broker.

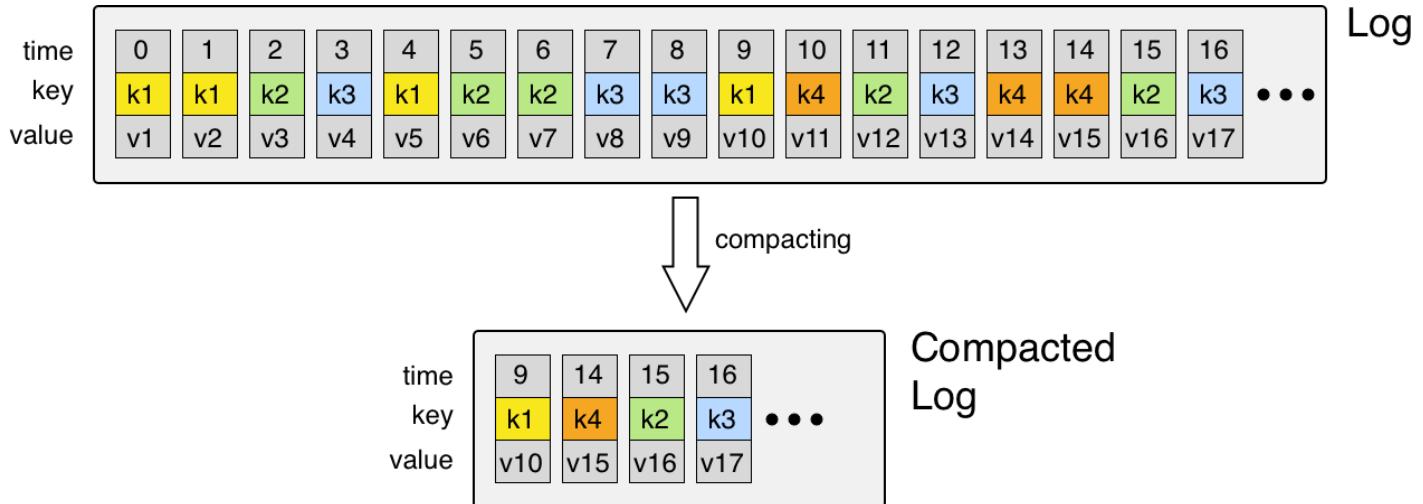
# Topics, Partitions and Segments



Normally one uses several Kafka brokers that form a cluster to scale out. If we have three topics (that is 3 categories of messages) then they may be organized physically as shown in the graphic.

- Partitions of each topic are distributed among the brokers
- Each partition on a given broker results in one to many physical files called segments
- Segments are treated in a rolling file strategy. Kafka opens/allocates a file on disk and then fills that file sequentially and in append only mode until it is full, where "full" depends on the defined max size for the file, (or the defined max time per segment is expired). Subsequently a new segment is created.

# Log Compaction



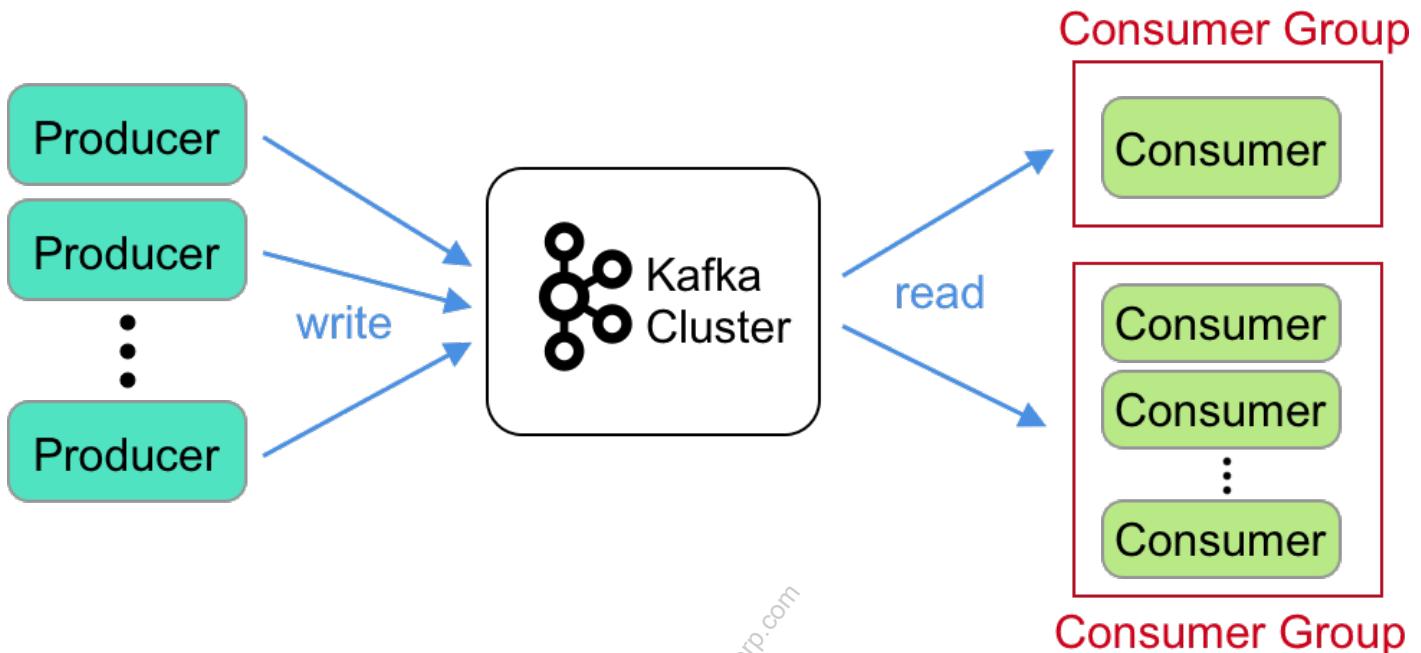
The log on top contains all events/records that are produced by the data source (each event in this slide has an offset [or a time], a key and a value). Events with the same key are color coded with the same color for easier readability.

If we create a new stream that only ever contains the last event per key of the full log then we call this "log compaction". The lower part of the slide shows the result of such a compaction.

Compacted logs are useful for restoring state after a crash or system failure. Kafka log compaction also allows downstream consumers to restore their state from a log compacted topic.

They are useful for in-memory services, persistent data stores, reloading a cache, etc. An important use case of data streams is to log changes to keyed, mutable data changes to a database table or changes to object in in-memory microservice.

Log compaction is a granular retention mechanism that retains the last update for each key. A log compacted topic log contains a full snapshot of final record values for every record key not just the recently changed keys.

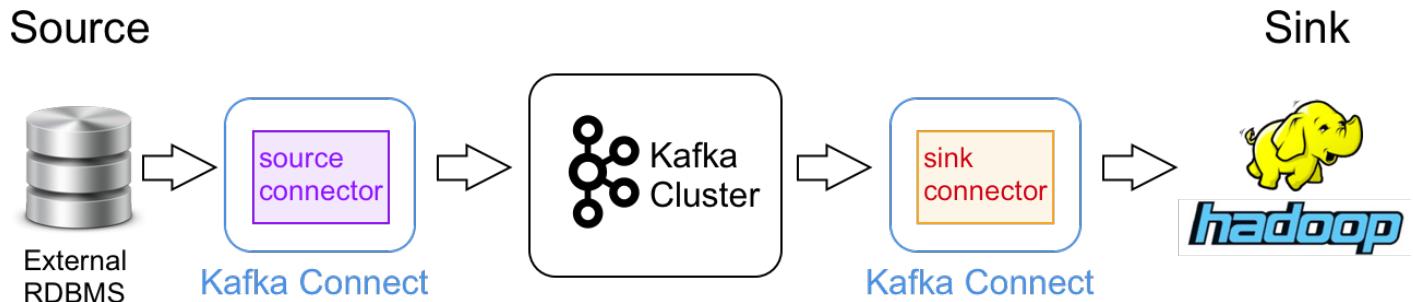


Now that we've talked about topics, partitions and segments we need to have a word about the entities that produce the data and those that consume data.

- **Producers:** these are applications that write data to one or many topics.
- **Consumers & Consumer Groups:** Consumers are organized in consumer groups. Members of a consumer group collaborate and as such allow the parallel processing of data. Consumers in a consumer group are instances of the same application with the same application ID.

Producers and consumers are both so called **Kafka clients**.

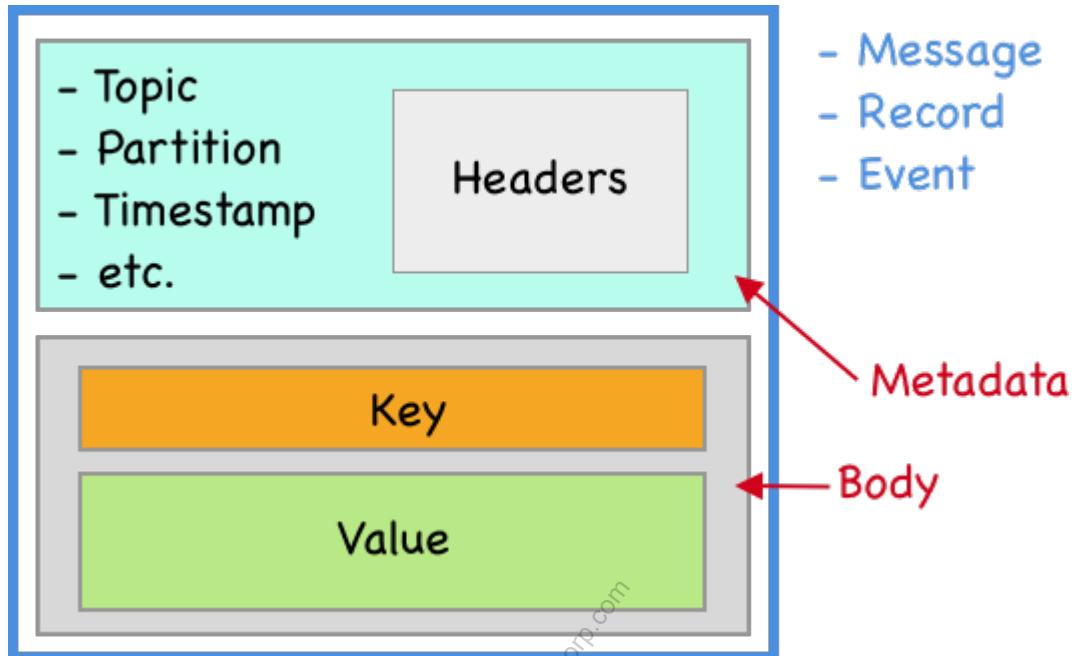
At a high-level, Kafka is a pub-sub messaging system that has producers that capture events. Events are sent to and stored locally on a central cluster of brokers. And consumers subscribe to topics or named categories of data. End-to-end, producers to consumer data flow is real-time.



Kafka Connect is a standard framework for source and sink connectors. It makes it easy to import data from and export data to standard data sources such as relational DBs, HDFS, cloud based blob storage, etc.

Confluent's goal is to provide a comprehensive list (>100) of supported source and sink connectors.

ybhandare@greendotcorp.com



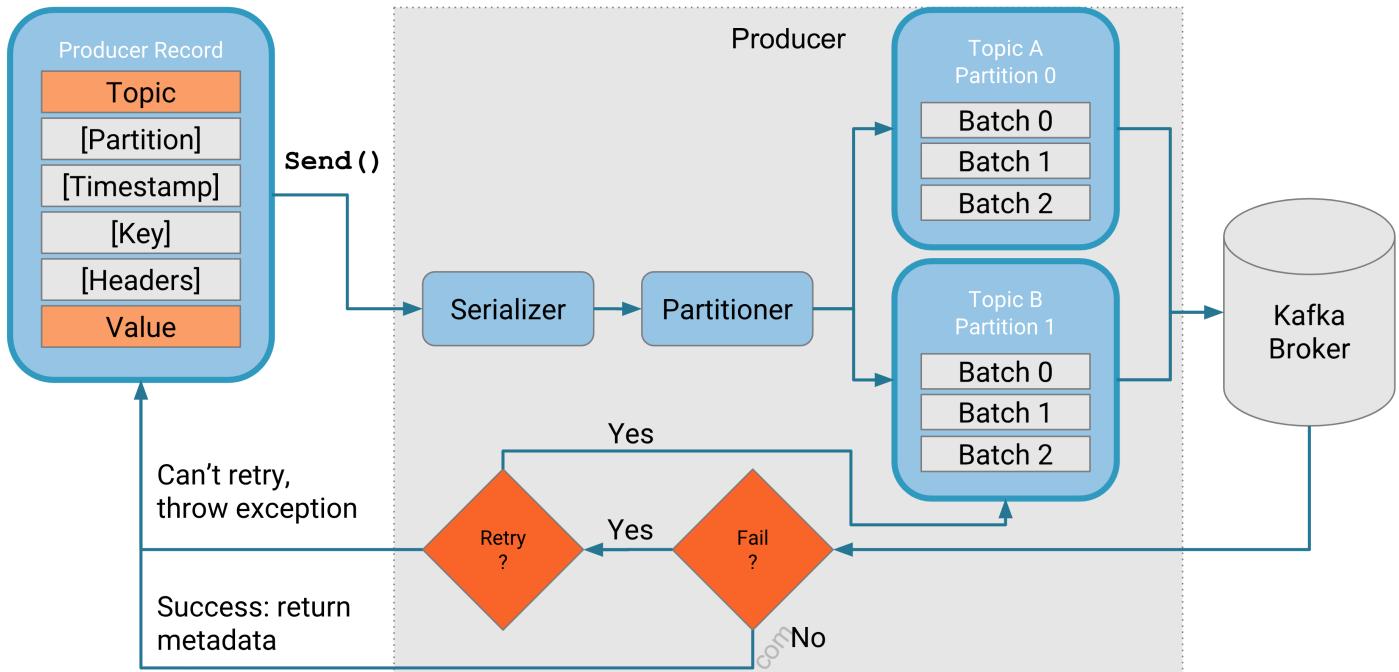
A data element in a topic (or log) is called a **record** in the Kafka world. Often we use equivalent words for a record. The most common ones are **message** and **event**.

A record in Kafka consists of **Metadata** and a **Body**.

- The metadata contains offset, compression, magic byte, timestamp and an optional **headers** collection of 0 to many key value pairs.
- The body consists of a **Key** and a **Value** part
- The value part is usually containing the **business relevant** data
- The key by default is used to decide into which partition a record is written to. As a consequence all records with identical key go into the same partition. This is important in the downstream processing, since ordering is (only) guaranteed on a partition and **not** on a topic level!

As noted, there is also a **timestamp** in the message but we will talk later about the concept of time...

# Producer Design

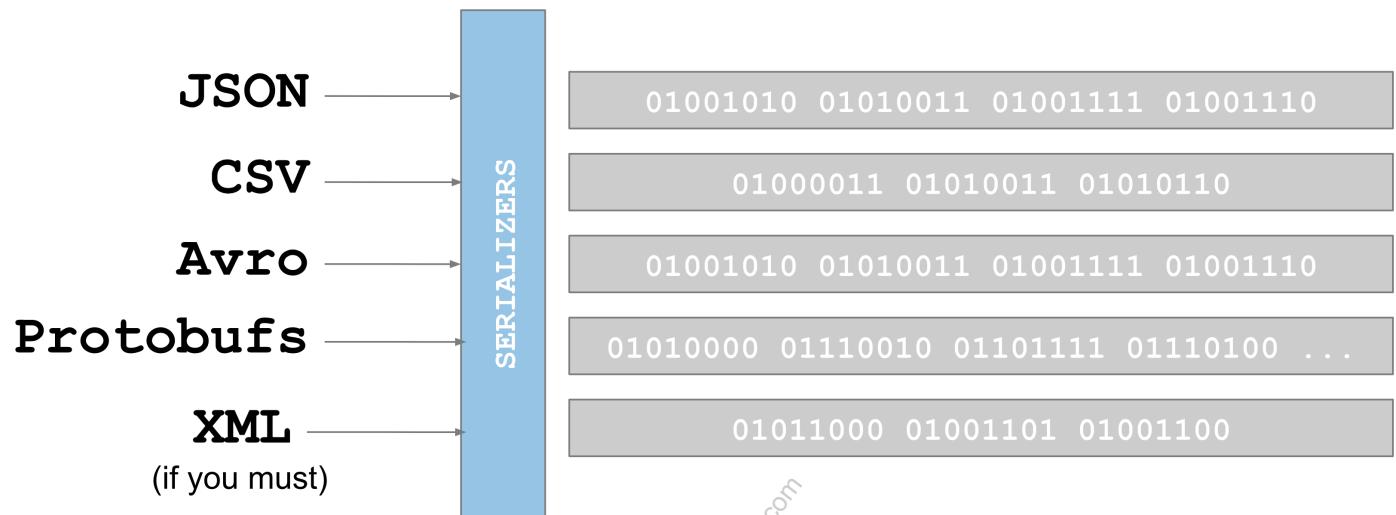


In this graphic you can see the high level architecture of a Kafka producer.

- On the left hand side you see a producer record, that is the data the producer wants to send to Kafka. It has the known elements:
  - key, value and headers, as well as more meta information such as
  - the topic name and partition number into which to write the record to
  - the timestamp of the record. Usually that is the time when the record is ingested by the broker into the commit log
- Once your code sends the record it goes through the serializer and the partitioner
- After those steps individual records are batched for improved throughput. Batching happens on a per topic partition level (i.e. only messages written to the same partition are batched together)
- Optionally there might also compression involved after the above steps - it is not shown here on this slide
- The producer then tries to write the batch to the Kafka broker. The broker will answer either with ACK or NACK:
  - If ACK then all is good and success metadata is returned to the producer
  - IF NACK then the producer transparently retries until the max number of retries is reached, in which case an exception is returned to the producer

## The Serializer

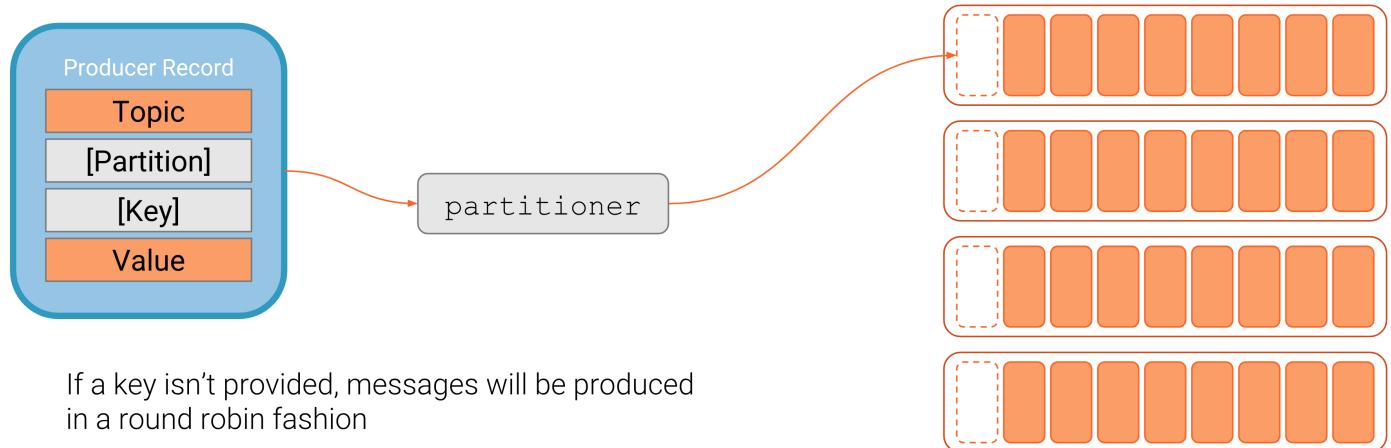
Kafka doesn't care about what you send to it as long as it's been converted to a byte stream beforehand.



Let's look at the serializer. The data that is stored on the Kafka brokers in the topics is binary in nature. Kafka does not care what you send. It treats your data just as streams of bytes. Thus you have the option to use many different data formats in your Kafka client applications, as long as there is a serializer that is supporting your data format. On the slide you see some popular data formats such as JSON, AVRO, and yes, ProtoBuf.

Of course, on the other side of the pipeline, in the consumer, the arrays of bytes will have to be deserialized such that you can continue to work with your familiar data format in the consumer application as well.

# Partitioning



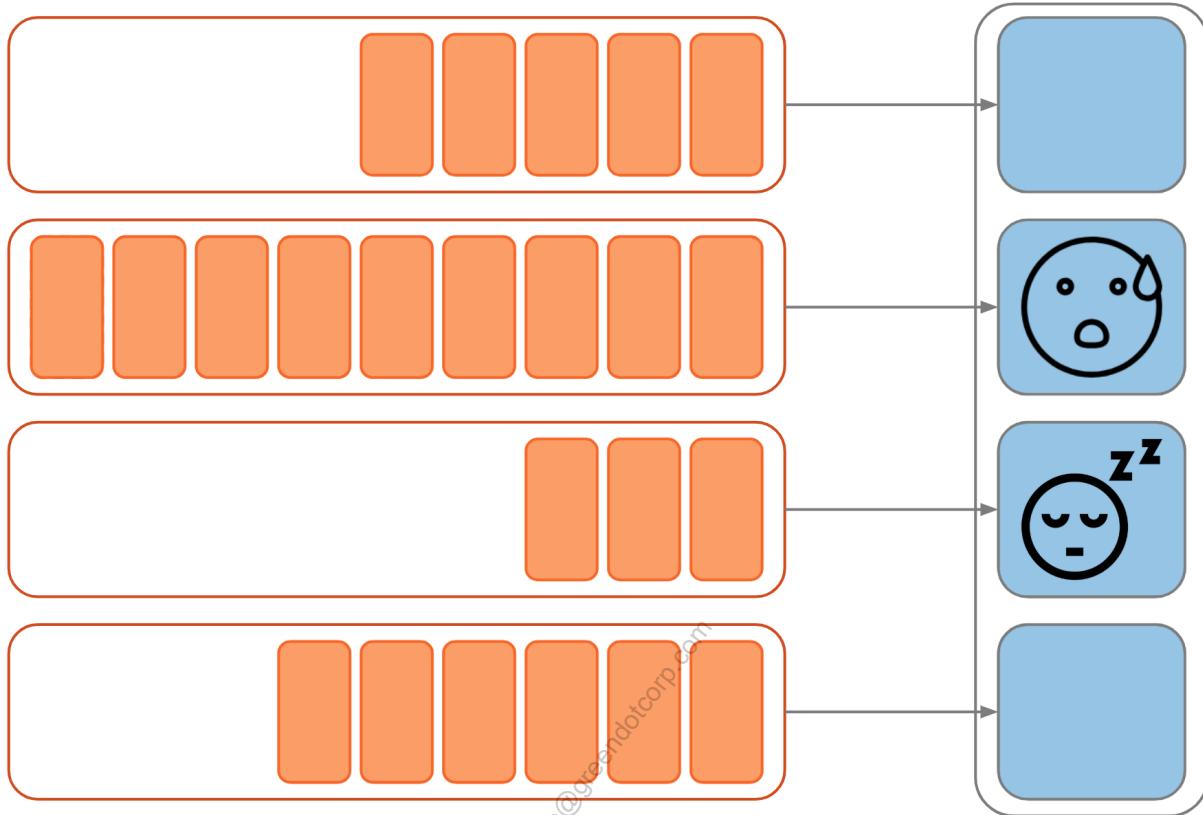
Default Partitioner: `partition = hash(key) % numPartitions`

Record keys determine the partition with the default kafka partitioner. Keys are used in the default partitioning algorithm:

```
partition = hash(key) % numPartitions
```

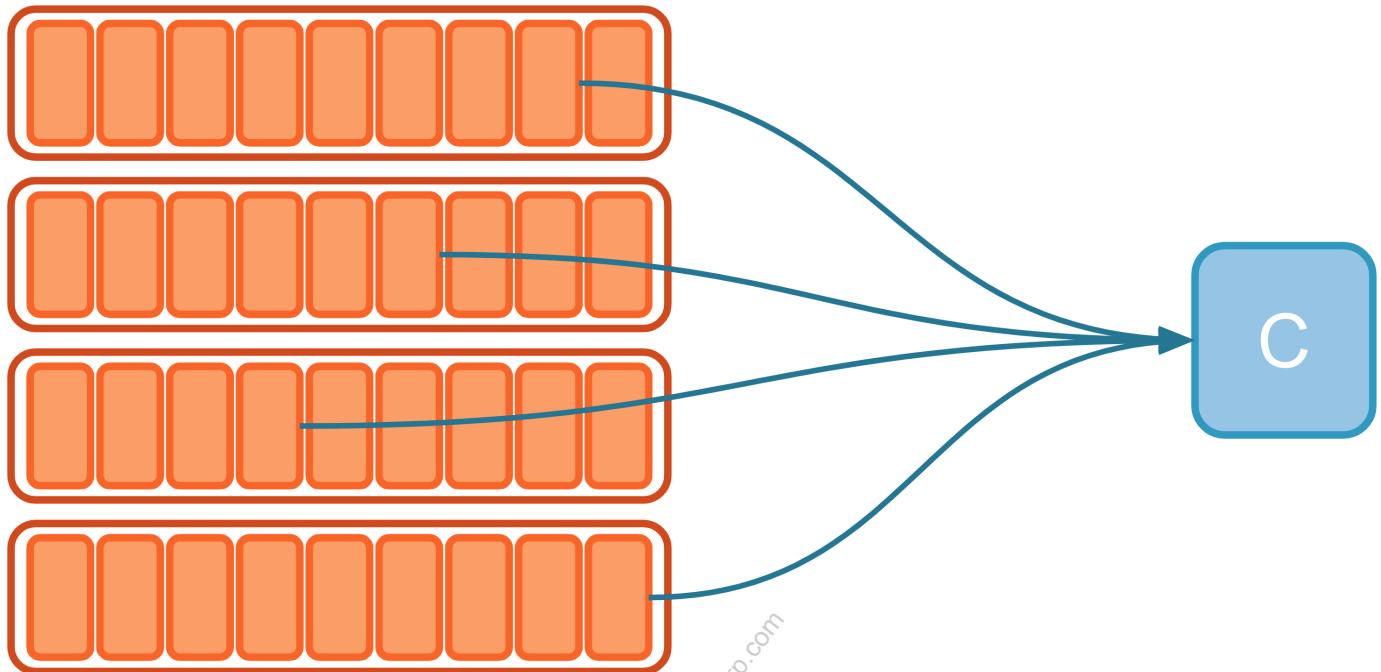
## Key Cardinality

### Consumers



- **Cardinality:** the number of elements in a set or other grouping, as a property of that grouping.
- Key cardinality affects the amount of work done by the individual consumers in a group. Poor key choice can lead to uneven workloads.
- Keys in Kafka don't have to be simple types like Integer, String, etc. They can be complex objects with multiple fields. For example, in some cases, a compound key can be useful, where part of the key is used for partitioning, and the rest used for grouping, sorting, etc. So, create a key that will evenly distribute groups of records around the partitions.

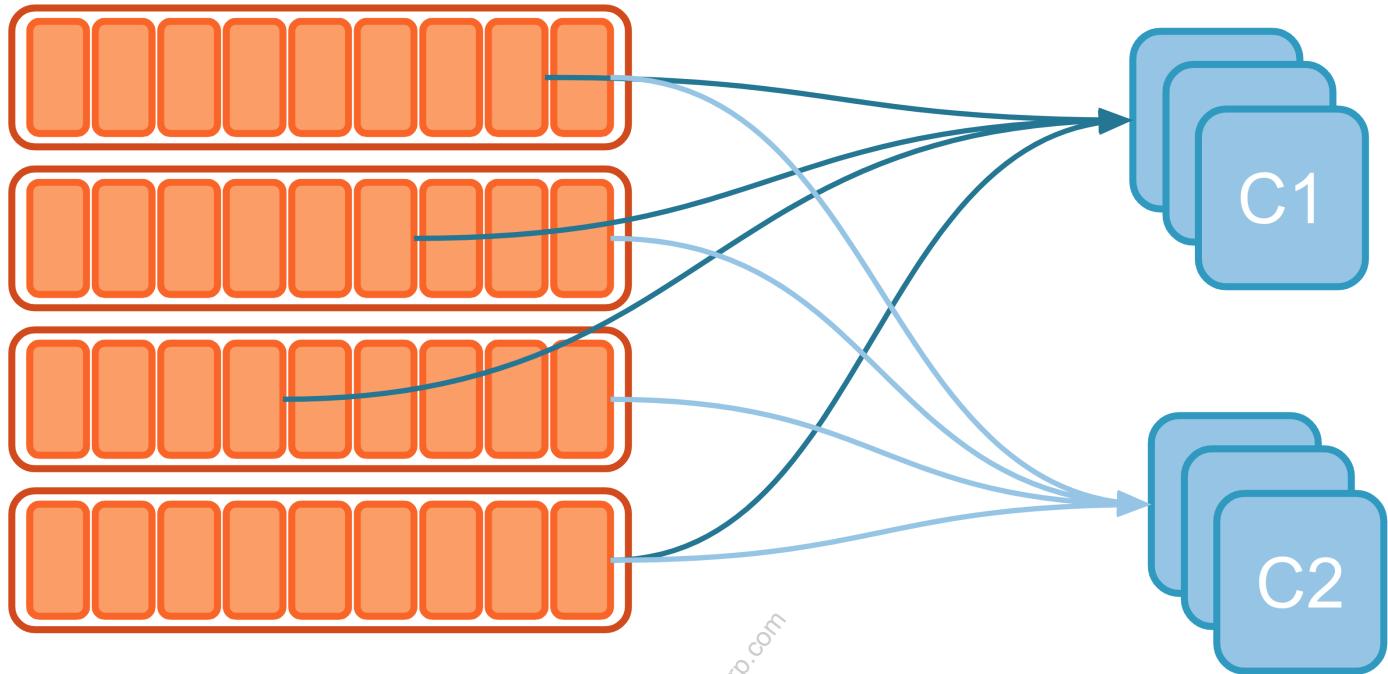
## Consuming from Kafka - Single Consumer



If you have a single consumer that consumes data from a topic, here with 4 partitions, then this consumer will consume all records from all partitions of the topic.

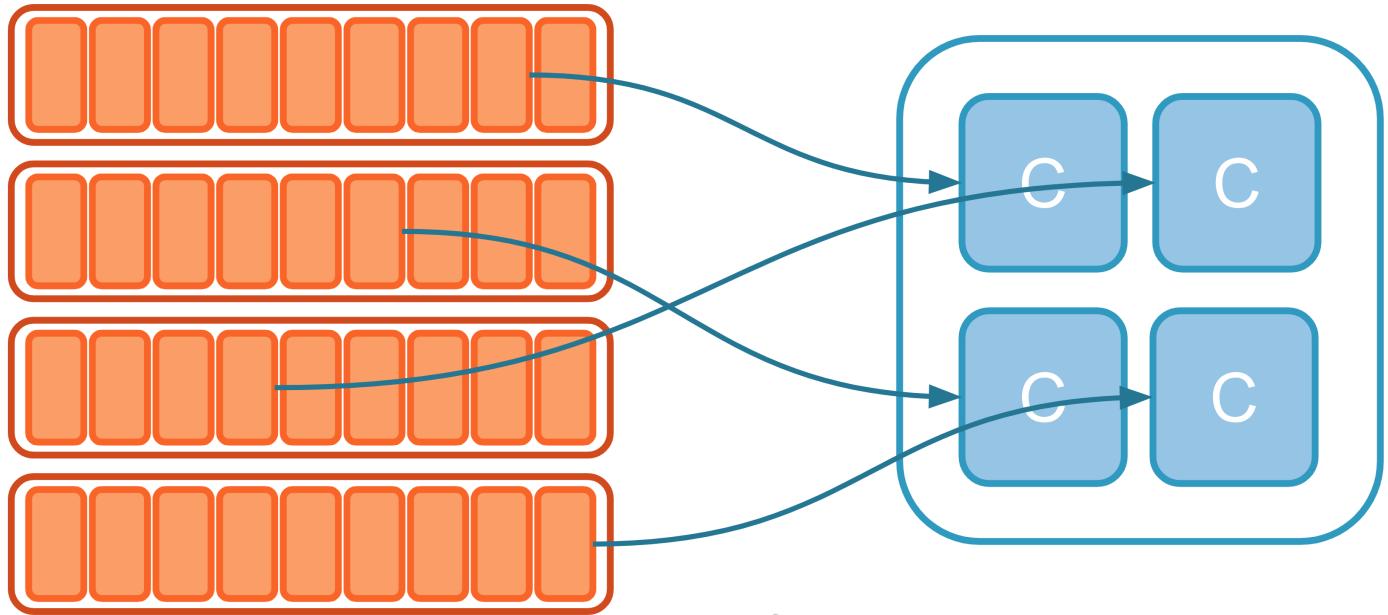
ybhandarkar@greenidotcorp.com

## Consuming from Kafka - Grouped Consumers



If there are more than one consumer collaborating, we call them a consumer group. The consumers in a consumer group will split up the workload among themselves in a somewhat even fashion. Note that, as indicated here on the slide, we can have multiple consumer groups consuming from the same topic(s).

## Consuming from Kafka - Grouped Consumer



As we said before, a consumer group transparently load balances the work among the participating consumer instances. In this image we have 4 consumers that consume a topic with 4 partitions. Thus each consumer consumes records from exactly one partition.

A partition is always consumed as a whole by a single consumer of a consumer group. A consumer in turn can consume from 0 to many partitions of a given topic.

### Questions:



1. Explain the relations between **topics** and **partitions**.
2. How does Kafka achieve **high availability**?
3. What is the **easiest** way to integrate your RDBMS with Kafka?

### Answers:

1. Topics are logical containers in Kafka and (usually) contain records or data of same type. Partitions are used to parallelized work with topics. Thus one topic has 1...n partitions.
2. Kafka uses replication to achieve HA. Each partition of a topic is available in the cluster on say 3 distinct brokers. In case of failure of one broker, still two additional copies of the data are available.
3. By using Kafka Connect with a plugin/connector that suits your needs, e.g. a JDBC source or sink connector, depending of the direction the data has to flow.

## Hands-On Lab

---

- In this Hands-On Exercise you will run a simple Kafka cluster and use some of the Kafka command line tools to do basic operations on the cluster.
- Please refer to the lab **Using Kafka's Command-Line Tools** in the exercise book.



ybhandare@greendotcorp.com



### Kafka's Architecture

ybhandare@greendotcorp.com

# Agenda

---



1. Introduction
2. Fundamentals of Apache Kafka
3. Kafka's Architecture ... ↪
4. Developing With Kafka
5. More Advanced Kafka Development
6. Schema Management In Kafka
7. Data Pipelines with Kafka Connect
8. Stream Processing with Kafka Streams
9. Stream Processing with KSQL
10. Event Driven Architecture
11. Confluent Cloud
12. Conclusion
13. Appendix: Basic Kafka Administration
14. Appendix: Apache Kafka driving ML & Data Analytics

ybhandare@greendotcorp.com

## Learning Objectives

---

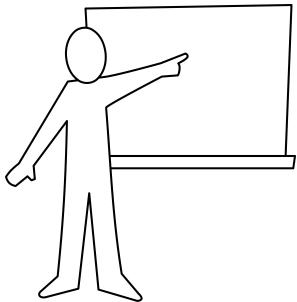


After this module you will be able to:

- Explain how Kafka log files are stored on the Kafka Brokers
- Describe how Kafka uses replicas for reliability
- Illustrate how Consumer Groups and Partitions provide scalability
- List the essential elements that contribute to a secure Kafka cluster

ybhandare@greendotcorp.com

# Module Map



- Kafka Log Files ...
- Replicas for Reliability
- Partitions and Consumer Groups for Scalability
- Security
- Hands-on Lab

ybhandare@greendotcorp.com

## What is a Commit Log?

---

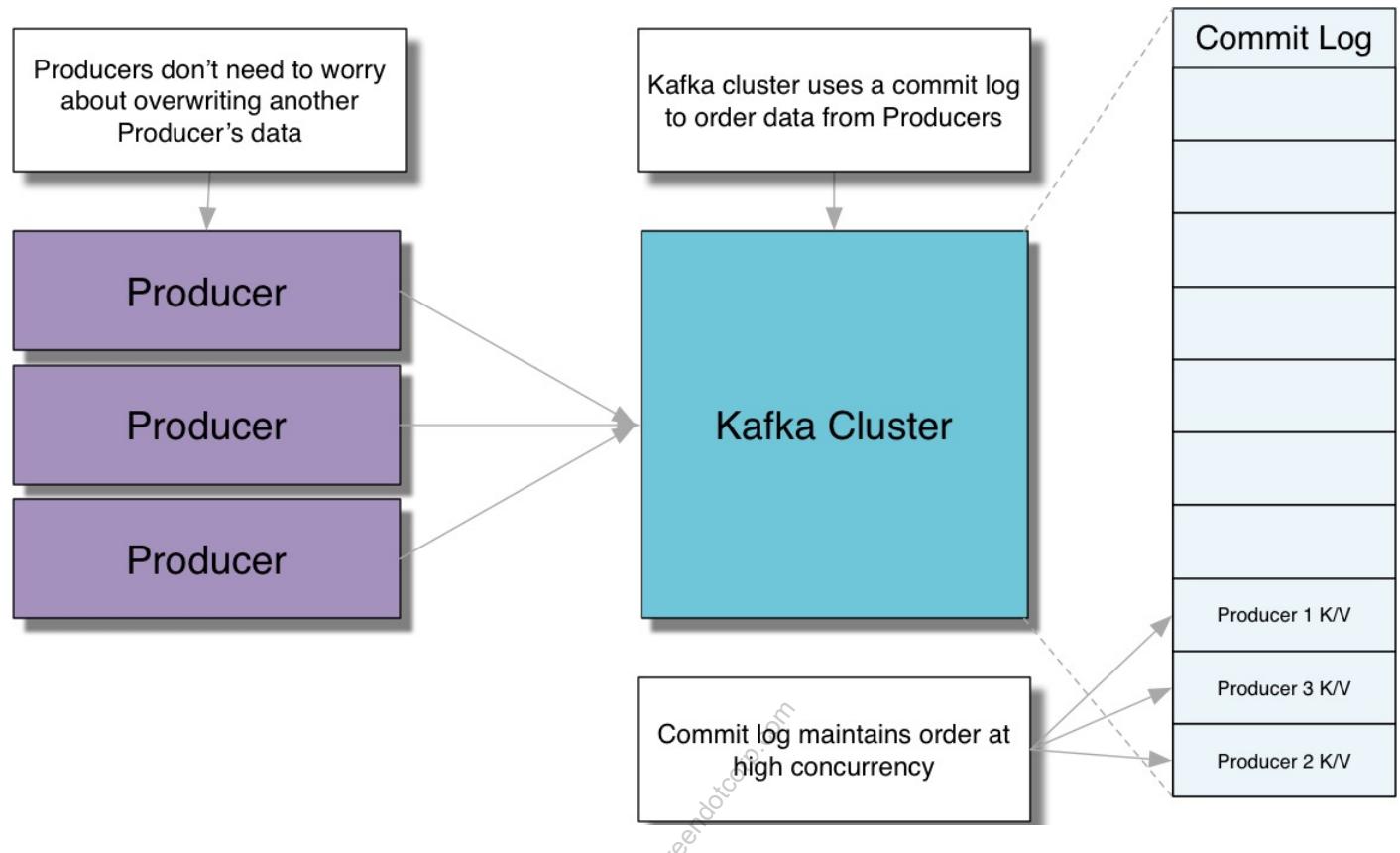
- A Commit Log is a way to keep track of changes as they happen
- Commonly used by databases to keep track of all changes to tables
- Kafka uses commit logs to keep track of all messages in a particular Topic
  - Consumers can retrieve previous data by backtracking through the commit log



**Backtracking** means going backwards and we cannot read backward through the log. Consumers can retrieve data from any point in the commit log

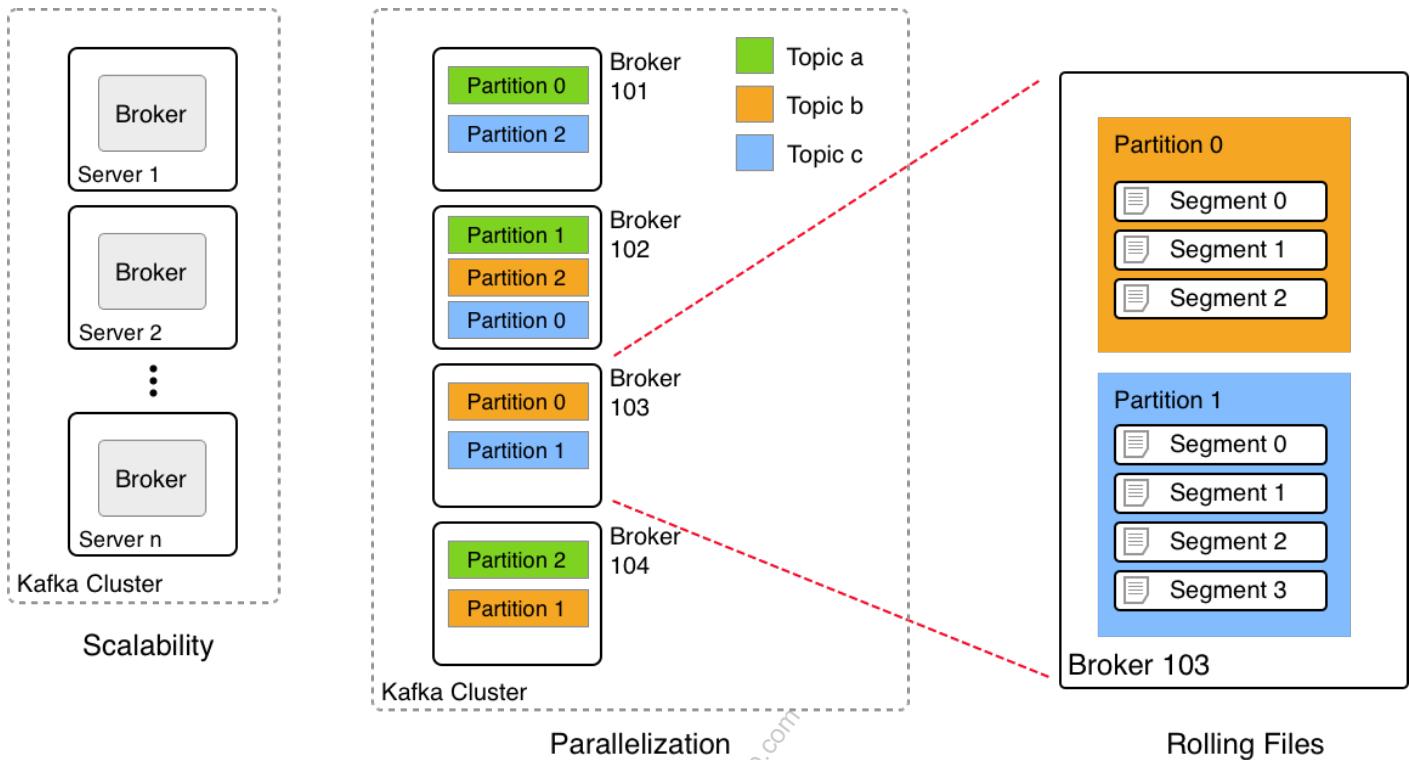
ybhandare@greendotcorp.com

## The Commit Log for High Concurrency



Inbound producer requests are placed in a request queue when received by the broker. This allows the broker to handle "simultaneous" messages without loss of data and to avoid throttling inbound traffic if possible. The broker moves the data from the request queue into the commit log which resides in the page cache.

## Partitions Are Stored as Separate Logs

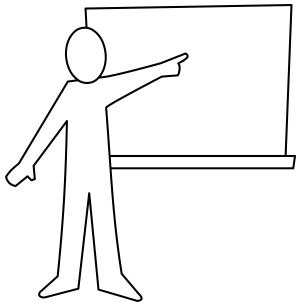


Each partition within a topic is a subset of the data held by that topic. Therefore, each partition will have its own commit log, each with its own set of messages and offset numbers.

Each commit log is subdivided into physical files on disk called segments. This is needed due to the fact that the amount of data that can be written to a partition is unbounded.

# Module Map

---



- Kafka Log Files
- Replicas for Reliability ... ↙
- Partitions and Consumer Groups for Scalability
- Security
-  Hands-on Lab

ybhandare@greendotcorp.com

## Problems With our Current Model

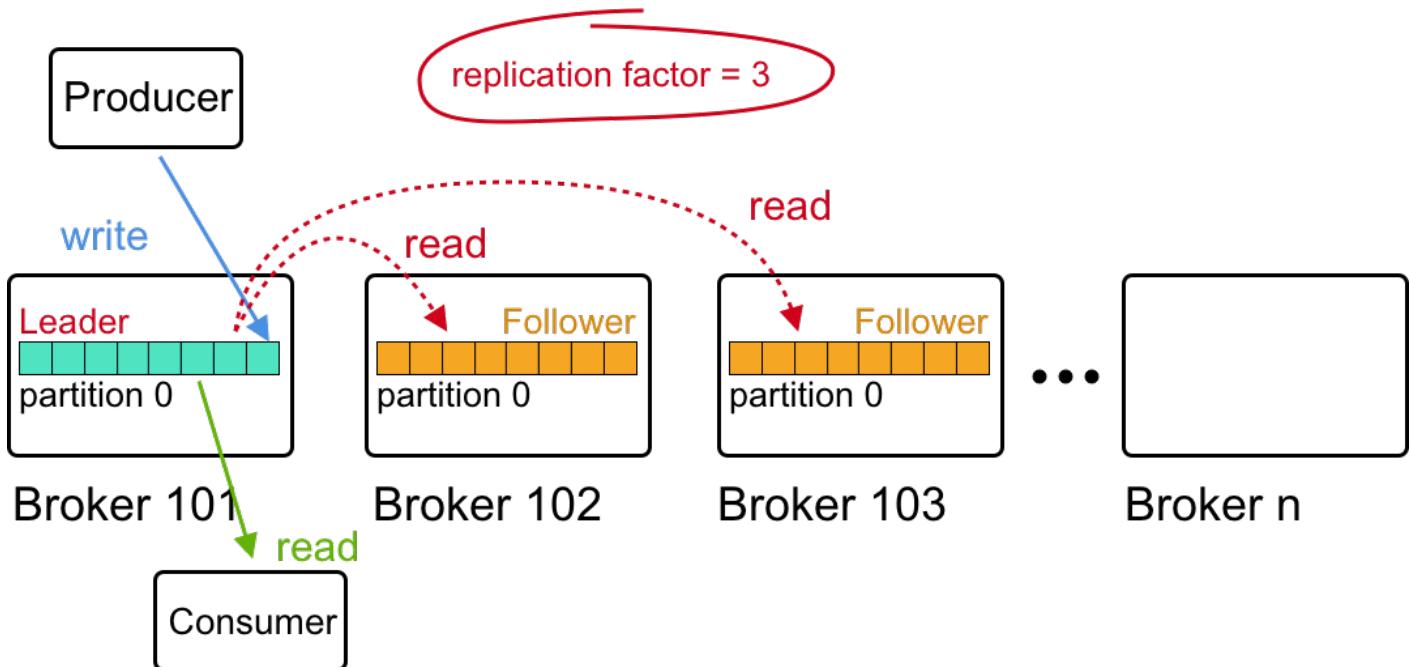
---

- So far, we have said that each Broker manages one or more Partitions for a Topic
- This does not provide reliability
  - A Broker failing would result in all of those Partitions being unavailable
- Kafka takes care of this by replicating each partition
  - The replication factor is configurable

Partitioning data is good for performance but not for reliability. The more parts there are in any system, the higher the probability that one of them will fail. Kafka addresses this problem by included a built-in replication solution to maintain multiple copies of each partition.

ybhandare@greendotcorp.com

## Replication of Partitions

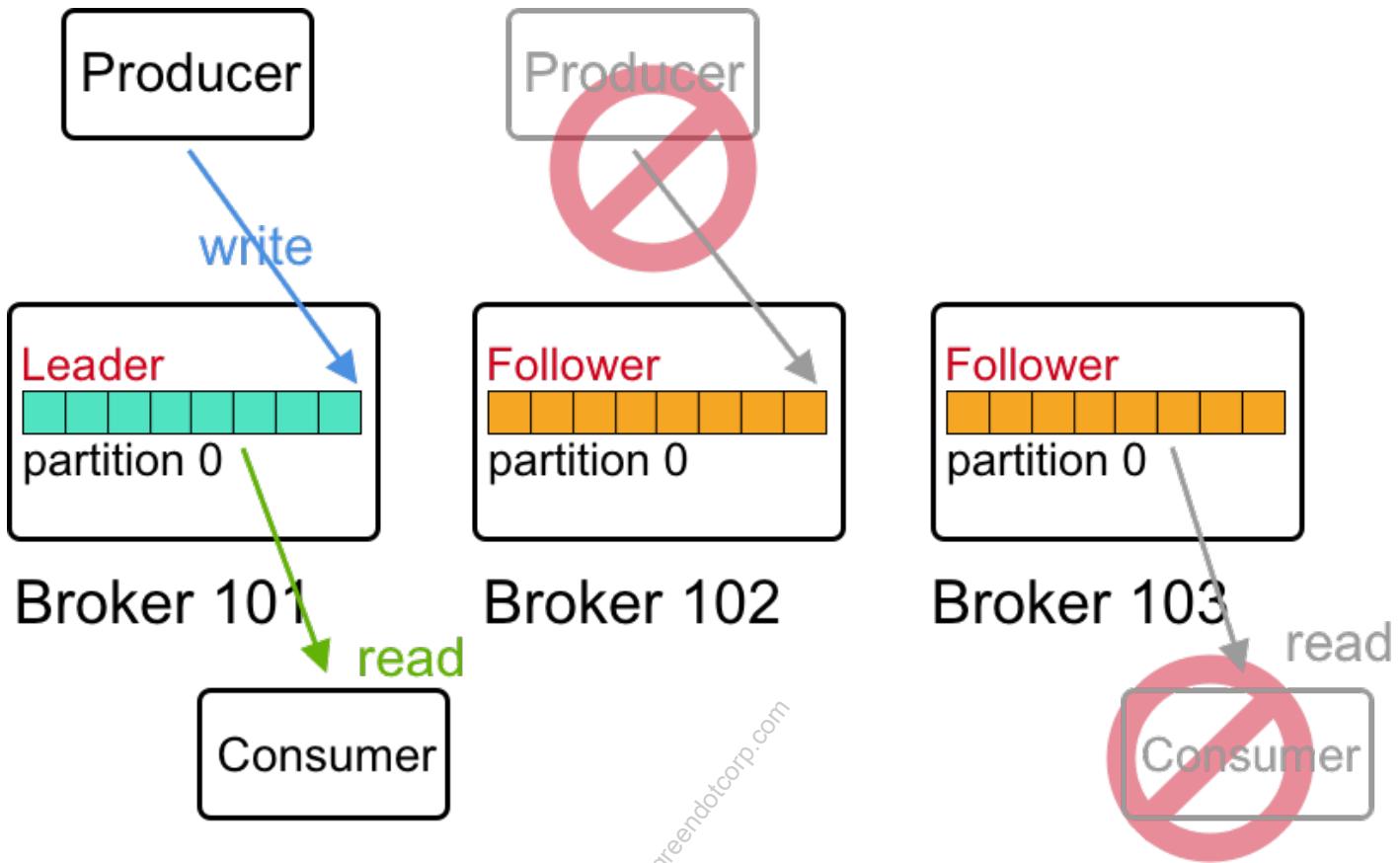


- Kafka maintains replicas of each partition on other Brokers in the cluster
  - Number of replicas is configurable
- One Broker is in the role of **leader** for a particular partition
  - All writes and reads go to and from the leader
  - Other Brokers are in the role of **follower** for that partition

At any given point of time, all the replicas are byte wise identical (except for where one hasn't caught up). To facilitate that, the ordering of the messages must be identical on all replicas.

For any given replicated partition, one replica is a leader and the rest are followers. All I/O (produce and consumer requests) go to the leader. In the case of write requests, this allows just one replica (the leader) to determine the ordering of messages. Once the messages are written to the local log of the leader, all the replicas can get the data.

## Important: Clients Do Not Access Followers

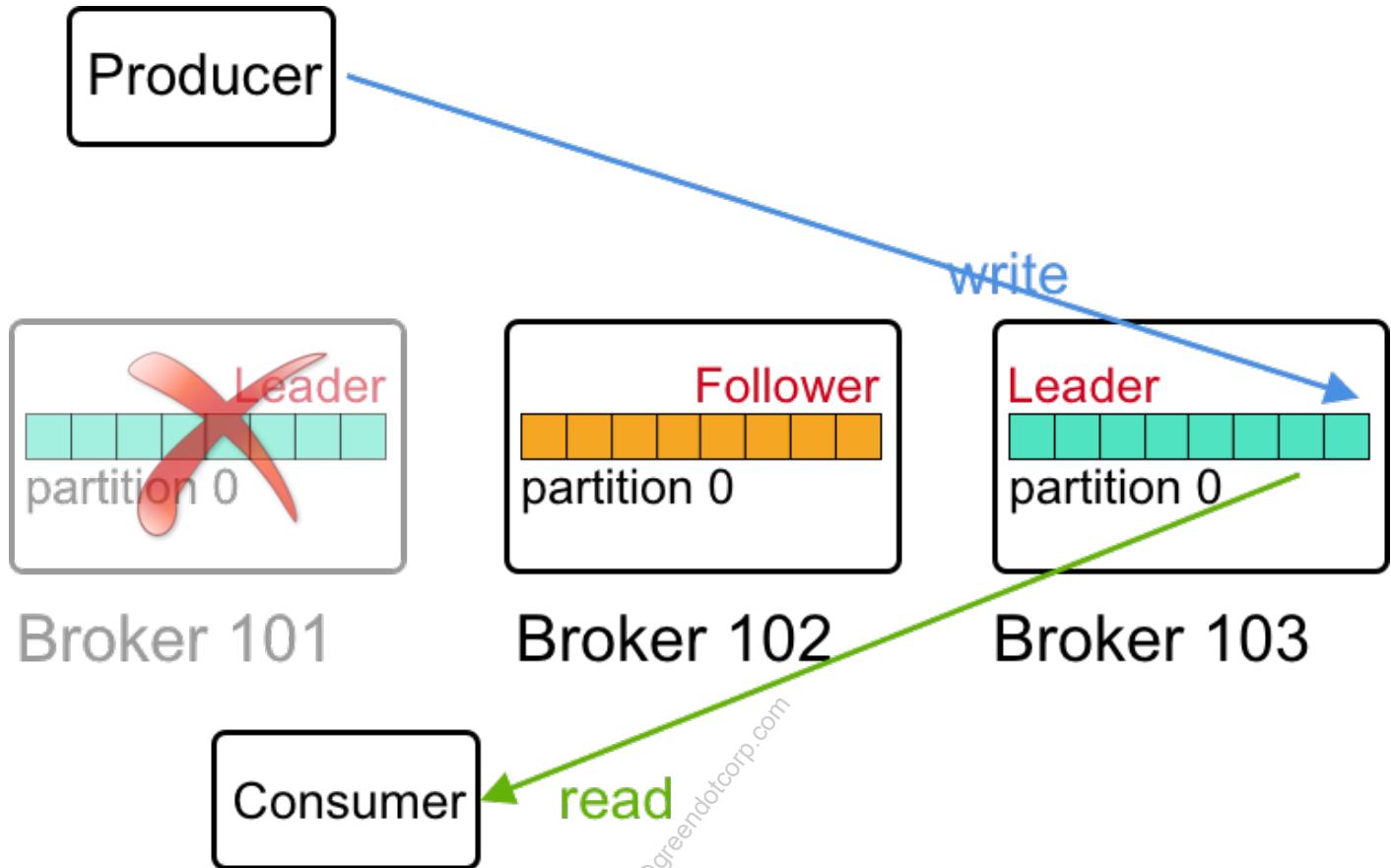


- It is important to understand that Producers only write to the leader
- Likewise, Consumers *only* read from the leader
  - They do not read from the replicas
  - Replicas only exist to provide reliability in case of Broker failure
- The followers just copy the data from the commit log of the leader as a pull request; they do not interact with any external clients.



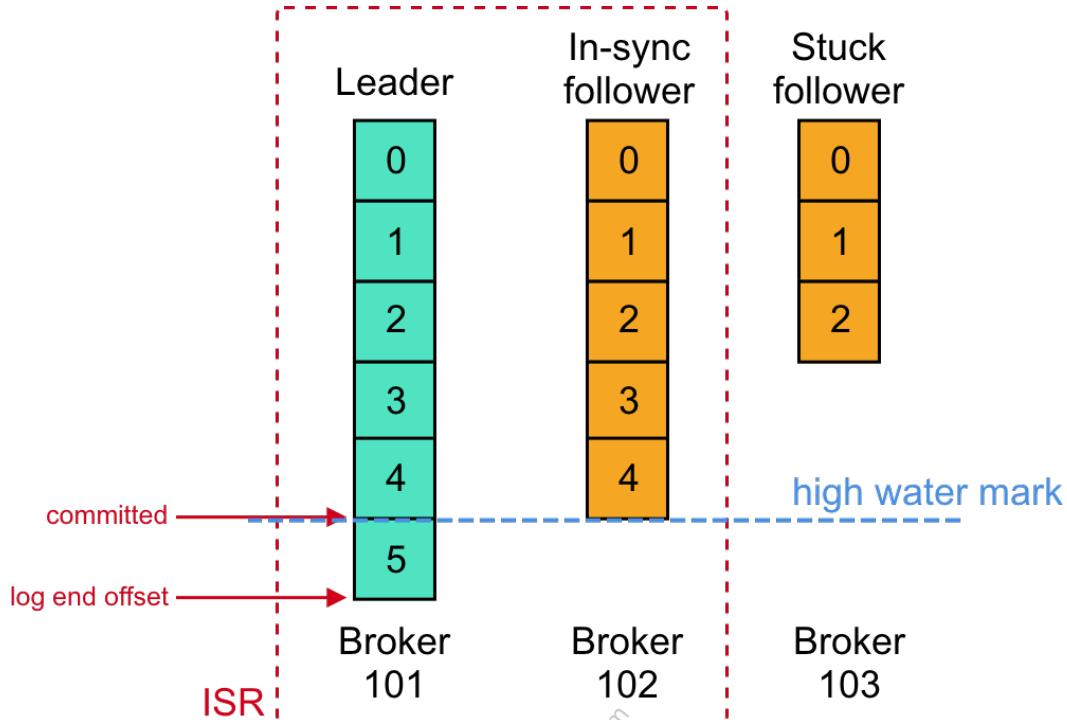
KIP-392 may change the behavior of clients (consumers) being able to access followers in a future version: <https://cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replica>

## Leader Failover



- If a leader fails, the Kafka cluster will elect a new leader from among the followers
- The clients will automatically switch over to the new leader

## In-Sync Replicas



- We have seen that Kafka uses replication to guarantee the durability (and HA) of data. Each partition can be replicated. Each replica is on a different broker. There is one leader and the other replicas are followers
- The In-Sync Replicas (ISR) is a list of the replicas - both leader and followers - that are identical up to a specified point called the high-water mark.
- If the leader fails, it is the list of ISRs which is used to elect a new leader
- Although this is more of an administration Topic, it helps to be familiar with the term ISR



A follower does not have to have all messages from the leader to be considered in-sync. An ISR follower is only guaranteed to have all the **committed** messages

# Managing Broker Failures

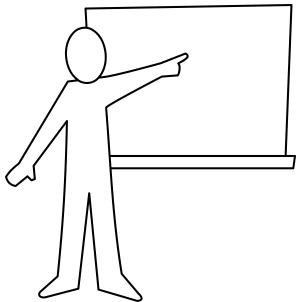
- One Broker in cluster is assigned role of **Controller**
  - Detects Broker failure/restart via ZooKeeper
- Controller action on Broker failure
  - Selects a new leader and updates the ISR list
  - Persists the new leader and ISR list to ZooKeeper
  - Sends the new leader and ISR list changes to all Brokers
- Controller action on Broker restart
  - Sends leader and ISR list information to the restarted Broker
- Current Controller fails → another Broker is assigned role of **Controller**

The Controller is a thread off the main broker software which maintains the list of partition information, such as leaders and ISR lists. The Controller runs on exactly one Broker in the cluster at any point in time.

The Controller monitors the health of every Broker by monitoring their interaction with ZooKeeper. If a Broker registration in ZK goes away, that indicates a Broker failure. The Controller elects new leaders for partitions whose leaders are lost due to failures and then broadcasts the metadata changes to all the Brokers so that clients can contact any Broker for metadata updates. The Controller also stores this data in ZooKeeper so that if the Controller itself fails, the new Controller elected by ZooKeeper will have a set of data to start with.

If the Controller fails, ZooKeeper will assign the role to the next Broker to check in with ZooKeeper after the failure.

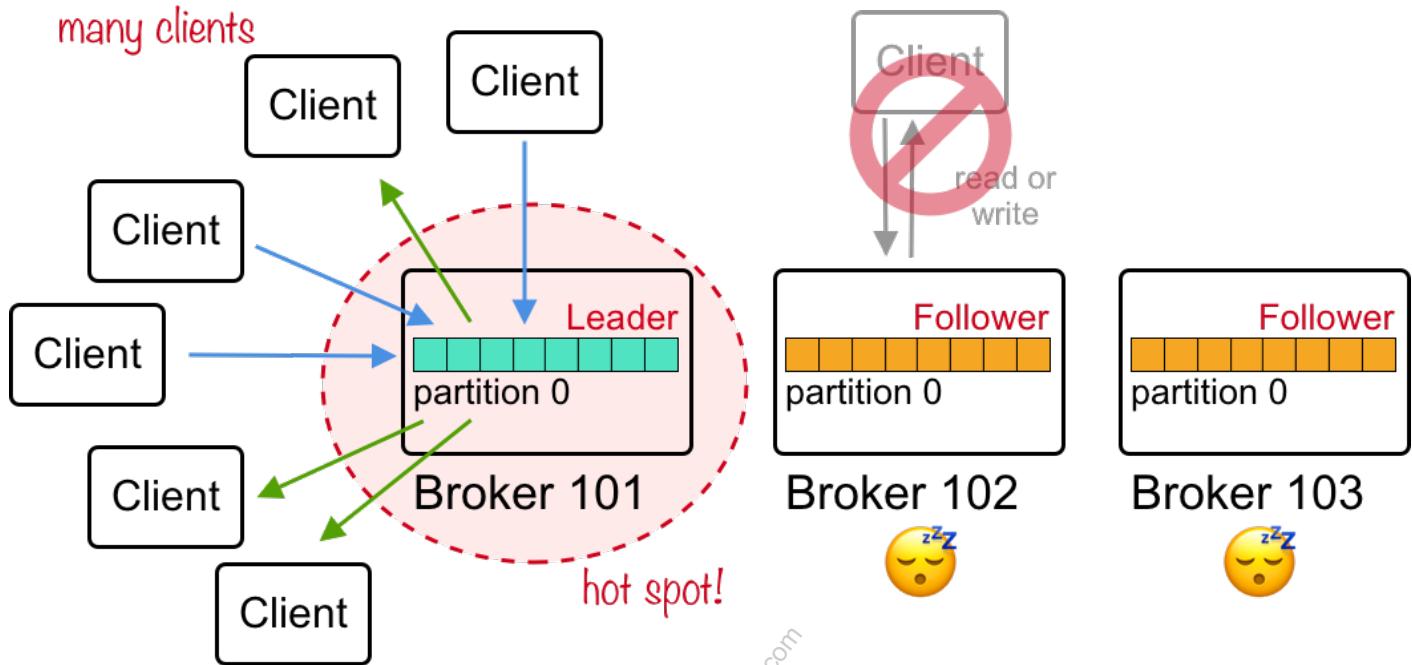
# Module Map



- Kafka Log Files
- Replicas for Reliability
- Partitions and Consumer Groups for Scalability ... ↵
- Security
-  Hands-on Lab

ybhandare@greendotcorp.com

## Scaling using Partitions

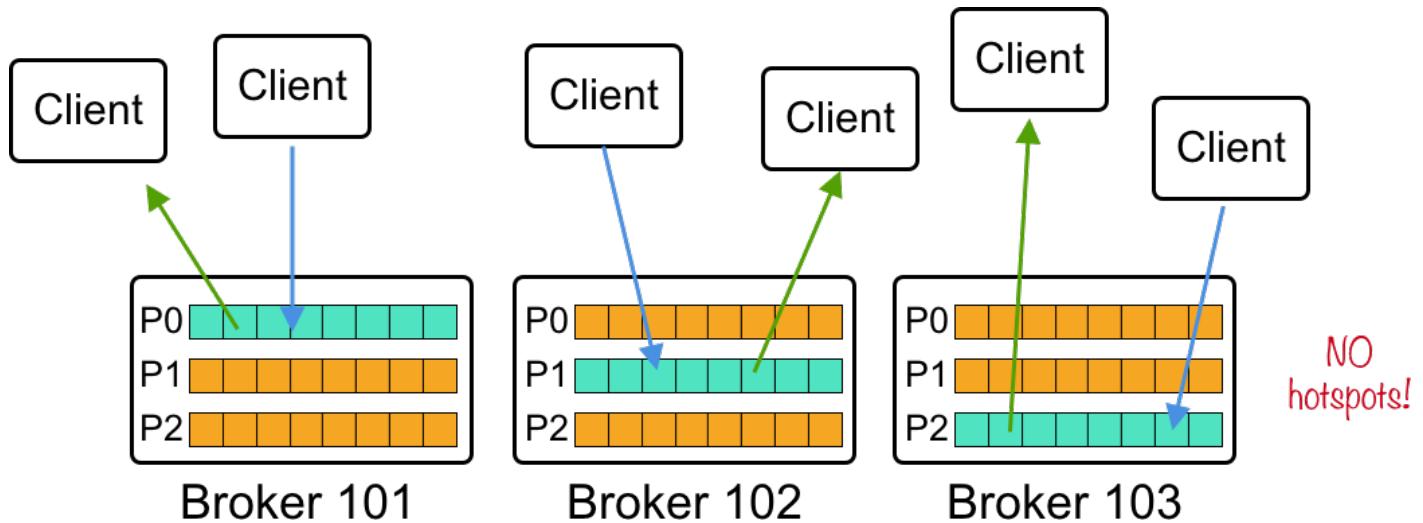


- **Recall:** All Consumers read from the leader of a Partition
- No clients write to, or read from, followers
- This can lead to congestion on a Broker if there are many Consumers



If it looks like the followers are completely passive, then this is certainly not the case, but in fact they are be actively polling the leader. On the other hand, the followers do not have to work hard...

## Scaling using Partitions



In a multi-partition topic with replication, each partition will have its own leader. The leader of a partition serves more traffic than the followers since it handles all produce and consume (or fetch) requests. Kafka will spread the leaders across the available brokers to balance the load.

A Broker is usually both a leader and a follower, depending on which partition you are talking about.

Both Producers and Consumers need to know which Broker is the leader for the partition they need to contact. This is done through a metadata request. When the client starts, it does not know which Brokers are the leaders of which partitions so it issues a metadata request to a Broker (any Broker). This is why the Controller caches the partition information on all the Brokers - so any Broker can respond to a metadata request. After receiving the updated metadata, the client will know who the leader is for each partition and will communicate with the appropriate Brokers. If a Broker fails, the client will get an error and will refresh its metadata.

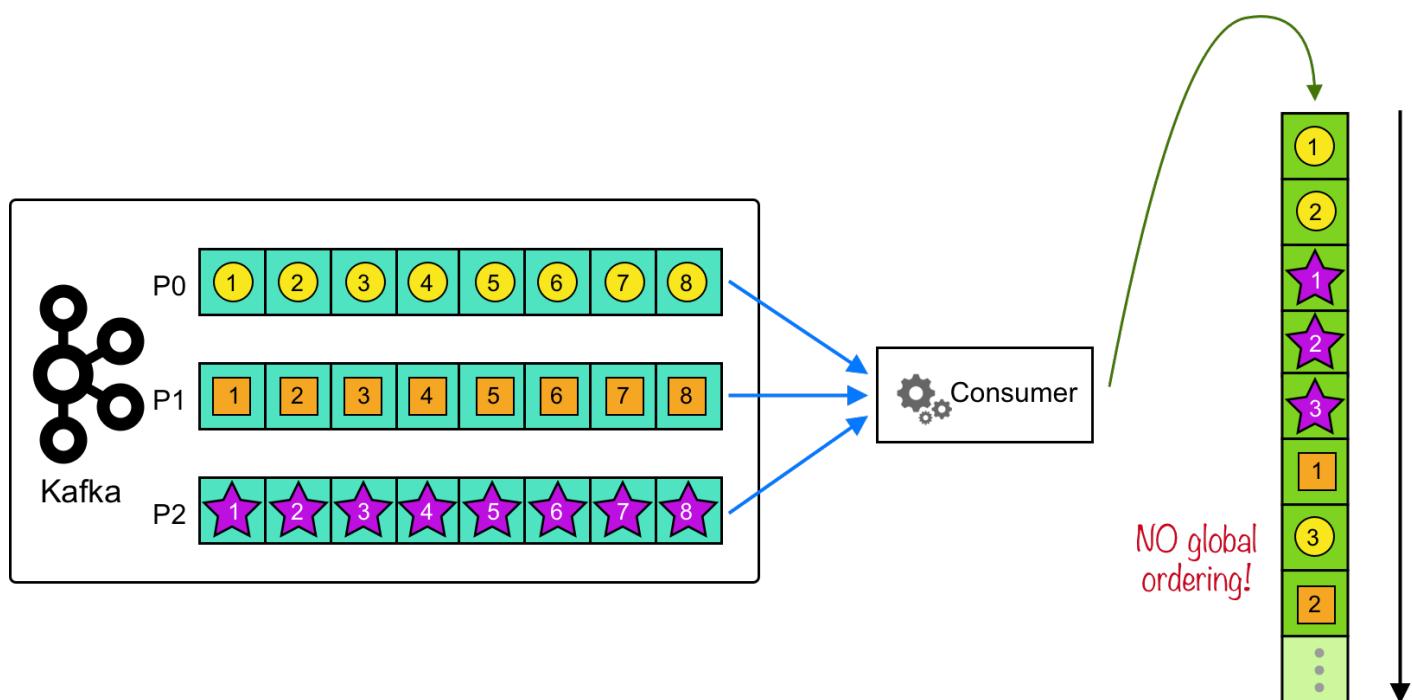
## Preserve Message Ordering

---

- Messages with the same key, from the same Producer, are delivered to the Consumer in order
  - Kafka hashes the key and uses the result to map the message to a specific Partition
  - Data within a Partition is stored in the order in which it is written
  - Therefore, data read from a Partition is read in order *for that partition*
- If the key is null and the default Partitioner is used, the record is sent to a random Partition

ybhandare@greendotcorp.com

## An Important Note About Ordering



- **Question:** how can you preserve message order if the application requires it?

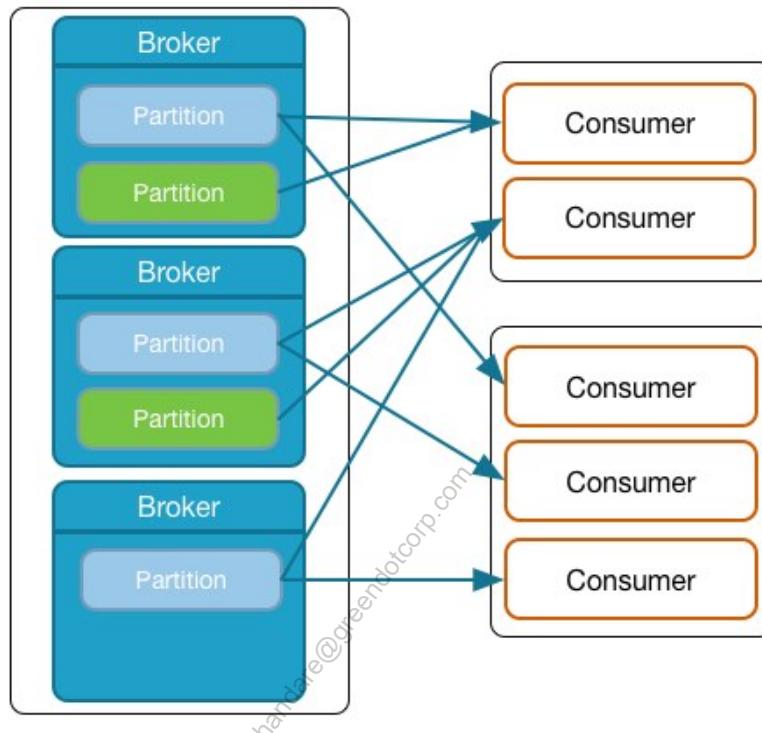
- If there are multiple Partitions, you **will not get total ordering across all messages** when reading data

Answer:

- Be selective when choosing the message key
  - Messages with the same key are delivered to the Consumer in order
- Use a single Producer
  - There are no guarantees that different producers writing to the same topic with the same key will send in order, due to batching, CPU, etc
- Ensure that the application calling the Kafka Producer is preserving message order as well
  - Send synchronously to the producer, e.g. wait till the first message is received before sending the second
- Make the applications responsible for ordering outside of Kafka
  - Have producers include information that can be used by the consumers to reorder the messages after receipt

## Group Consumption Balances Load

- Multiple Consumers in a *Consumer Group*
  - The `group.id` property is identical across all Consumers in the group
- Consumers in the group are typically on separate machines
- Automatic failure detection and load rebalancing



Kafka Topics allow the same message to be consumed multiple times by different Consumer Groups.

A Consumer Group binds together multiple consumers for parallelism. Members of a Consumer Group are subscribed to the same topic(s). The partitions will be divided among the members of the Consumer Group to allow the partitions to be consumed in parallel. This is useful when processing of the consumed data is CPU intensive or the individual Consumers are network-bound.

Within a Consumer Group, a Consumer can consume from multiple Partitions, but a Partition is only assigned to one Consumer to prevent repeated data.

Consumer Groups have built-in logic which triggers partition assignment on certain events, e.g., Consumers joining or leaving the group.

## Partition Assignment within a Consumer Group

- Partitions are ‘assigned’ to Consumers
  - A single Partition is consumed by only one Consumer in any given Consumer Group
  - *i.e.*, you are guaranteed that messages with the same key will go to the same Consumer
    - Unless you change the number of partitions (see later)
  - `partition.assignment.strategy` in the Consumer configuration

The Consumer Group is managed by a process called a Group Coordinator. Like the Controller, it is a thread from the main Broker software. Unlike the Controller, there are typically multiple Coordinators running in the Cluster at any given time.

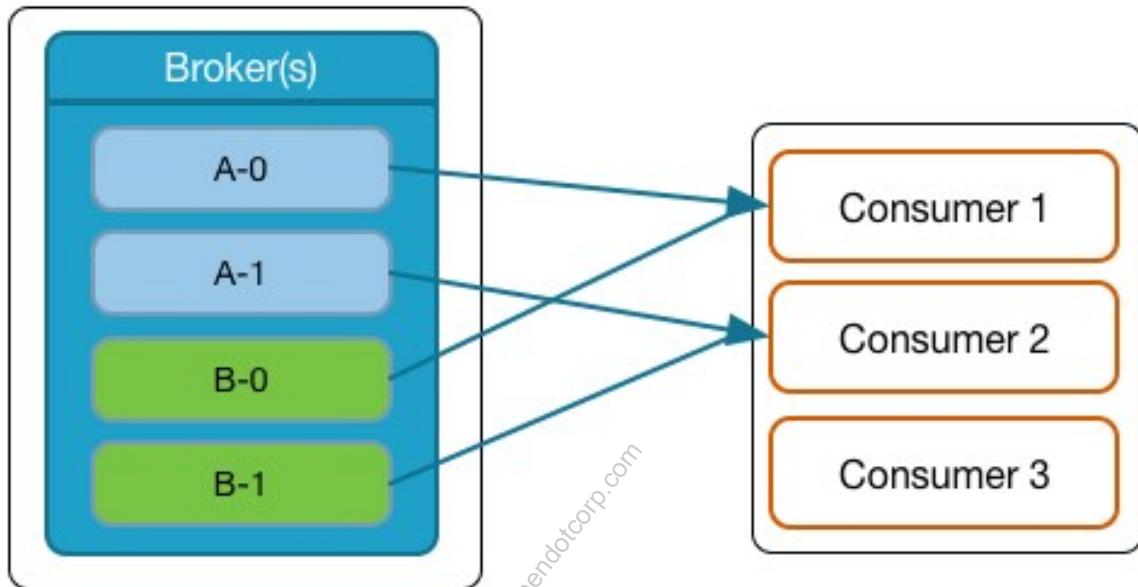
The main tasks of the Group coordinator are handling the consumer offsets, managing the group membership, and coordinating partition assignments and rebalances.

While the Group Coordinator manages the partition assignment, it does not actually create the mapping of partitions to Consumers. To avoid putting unnecessary calculations on the Brokers, the Coordinator elects one of the Consumers in the group to be the Group Leader. Additionally, delegating partition assignment to the Consumer group leader allows different groups to utilize different partition assignment strategies. A single Group Coordinator could be coordinating many consumer groups, so performing partition assignment there would require all groups managed by that coordinator to use the same strategy.

When the Coordinator detects that a rebalance is required, it sends the current list of partitions and Consumers to the Group Leader. The Group Leader generates the map and sends it back to the Coordinator, who then distributes it to the members of the group.

## Partition Assignment Strategy: Range

- Range (default)
  - Topic/Partition: topics A and B, each with two Partitions 0 and 1
  - Consumer Group: Consumers c1, c2, c3
  - c1: {A-0, B-0} c2: {A-1, B-1} c3: {}



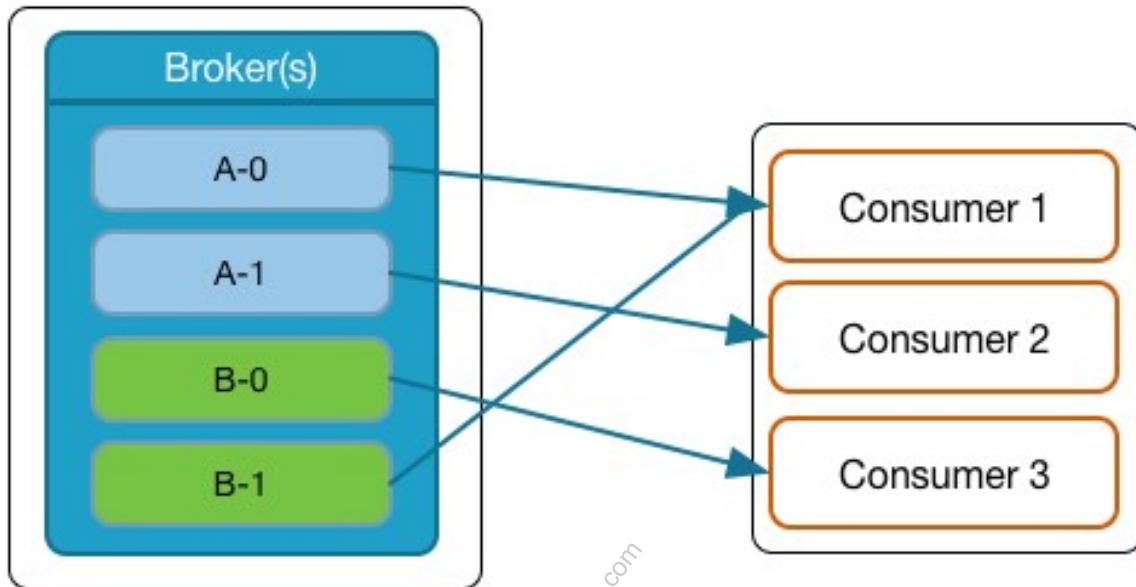
There are three built-in Partition Assignment strategies: Range (default), RoundRobin, and Sticky.

In Range, the Partition assignment assigns matching partitions to the same Consumer. In this example, there are two 2-partition Topics and three Consumers. The first partition from each Topic is assigned to one Consumer, the second partition from each is assigned to another Consumer, repeating until there are no more Partitions to assign. Since we have more Consumers than Partitions in any Topic, one Consumer will be idle.

The Range strategy is useful for "co-partitioning", which is particularly useful for Topics with keyed messages. Imagine that these two Topics are using the same key - for example, a userid. Topic A is tracking search results for specific user IDs; Topic B is tracking search clicks for the same set of user IDs. By using the same user IDs for the key in both Topics, messages with the same key would land in the same numbered Partition in both Topics (assuming both topics had the same number of Partitions) and so will land in the same Consumer.

## Partition Assignment Strategy: RoundRobin

- RoundRobin
  - c1: {A-0, B-1} c2: {A-1} c3: {B-0}



- Partition assignment is automatically recomputed on changes in Partitions/Consumers

The RoundRobin strategy is much simpler. Partitions are assigned one at a time to Consumers in a rotating fashion until all the Partitions have been assigned. This provides much more balanced loading of Consumers than Range.

## Partition Assignment Strategy: Sticky

- Preserves existing Partition assignments to Consumers during reassignment to reduce overhead
  - Kafka Consumers retain pre-fetched messages for Partitions assigned to them before a reassignment
- Reduces the need to cleanup local Partition state between rebalances

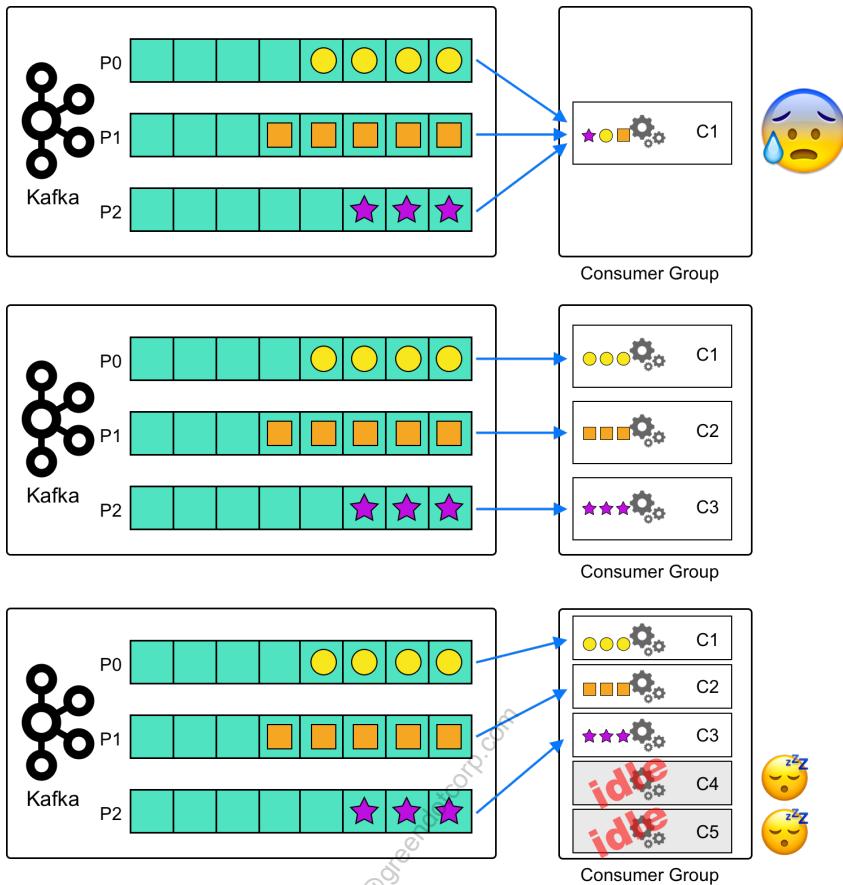
Sticky assignment strategy was introduced in Kafka 0.11 (Confluent 3.3)

An important note about Range and RoundRobin: Neither strategy guarantees that Consumers will retain the same Partitions after a reassignment. In other words, if a Consumer 1 is assigned to Partition A-0 right now, Partition A-0 may be assigned to another Consumer if a reassignment were to happen. Most Consumer applications are not locality-dependent enough to require that Consumer-Partition assignments be static.

If your application requires Partition assignments to be preserved across reassignments, use the Sticky strategy. Sticky is RoundRobin with assignment preservation.

A full description of this feature can be found at <https://cwiki.apache.org/confluence/display/KAFKA/KIP-54--+Sticky+Partition+Assignment+Strategy>

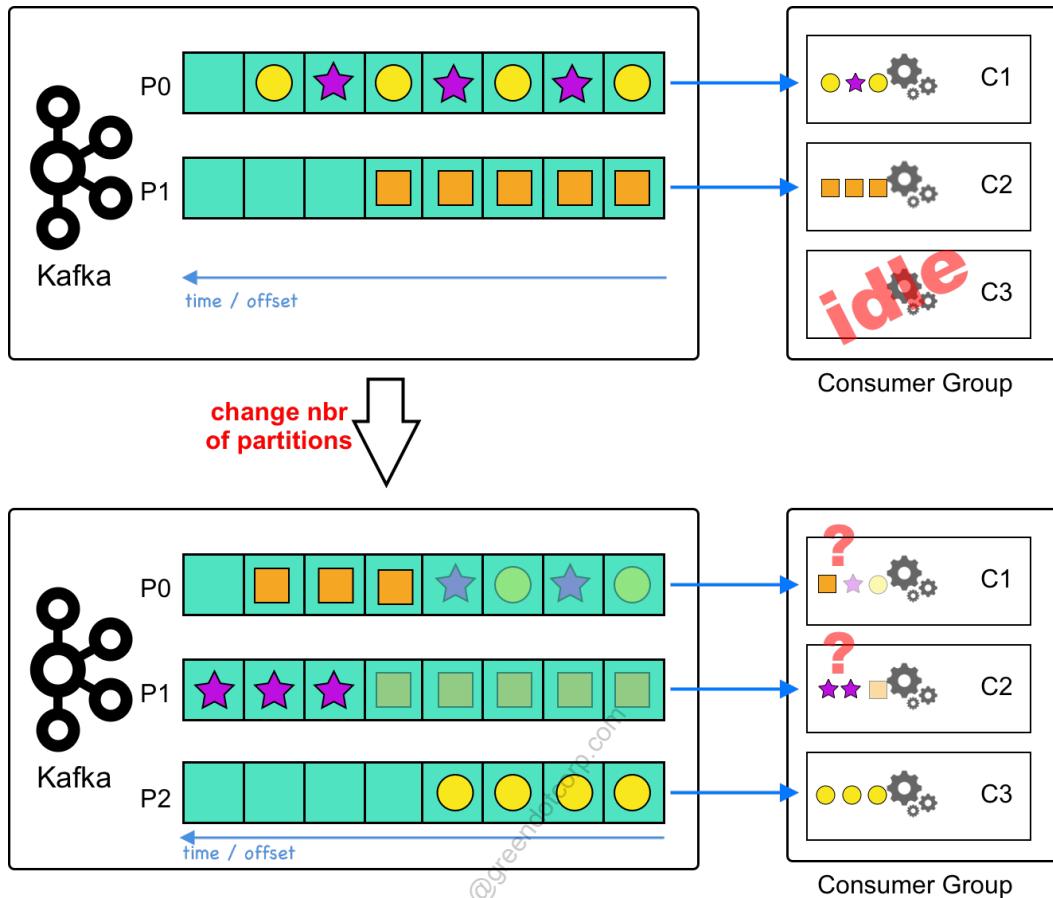
## Consumer Groups: Limitations



The number of useful Consumers in a Consumer Group is constrained by the number of Partitions on the Topic. **Example:** If you have a Topic with three partitions, and ten Consumers in a Consumer Group reading that Topic, only three Consumers will receive data. One for each of the three Partitions.

If there are more consumers than partitions, the additional consumers will sit idle. The idea of a hot-standby consumer is not required but can prevent performance differentials during client failures.

## Consumer Groups: Caution When Changing Partitions



On this slide icons with same shapes represent records with same key.

Recall that semantic partition works on the idea that a message will be sent to the partition determined by the formula  $\text{hash}(\text{key}) \% n$ , where  $n$  is the number of partitions. Changing the  $n$  number could change the output of the formula, resulting in messages with the same key being sent to different partitions. This would defeat the purpose of semantic partitioning.

Example: Using Kafka's default Partitioner, Messages with key **K1** were previously written to Partition 0 of a Topic. After repartitioning, new messages with key **K1** may now go to a different Partition. Therefore, the Consumer which was reading from Partition 0 may not get those new messages, as they may be read by a new Consumer

## Consumer Groups: Caution When Changing Partitions

There are strategies to mitigate this problem (e.g., migrating to a larger topic rather than just expanding the existing topic) but the best option is to plan appropriately so that you do not have to resize your topic. Selecting an appropriate number of partitions for your topic is something that will be discussed later in the course.

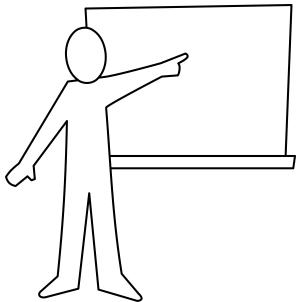


In the lower part of the slide the intent to show is that the partitions may after the change contain values with the keys that were assigned prior to the change and new key values, after the change. The downstream consumers might be confused by that (that's why I put a question mark near to the consumers C1 and C2)...

ybhandare@greendotcorp.com

# Module Map

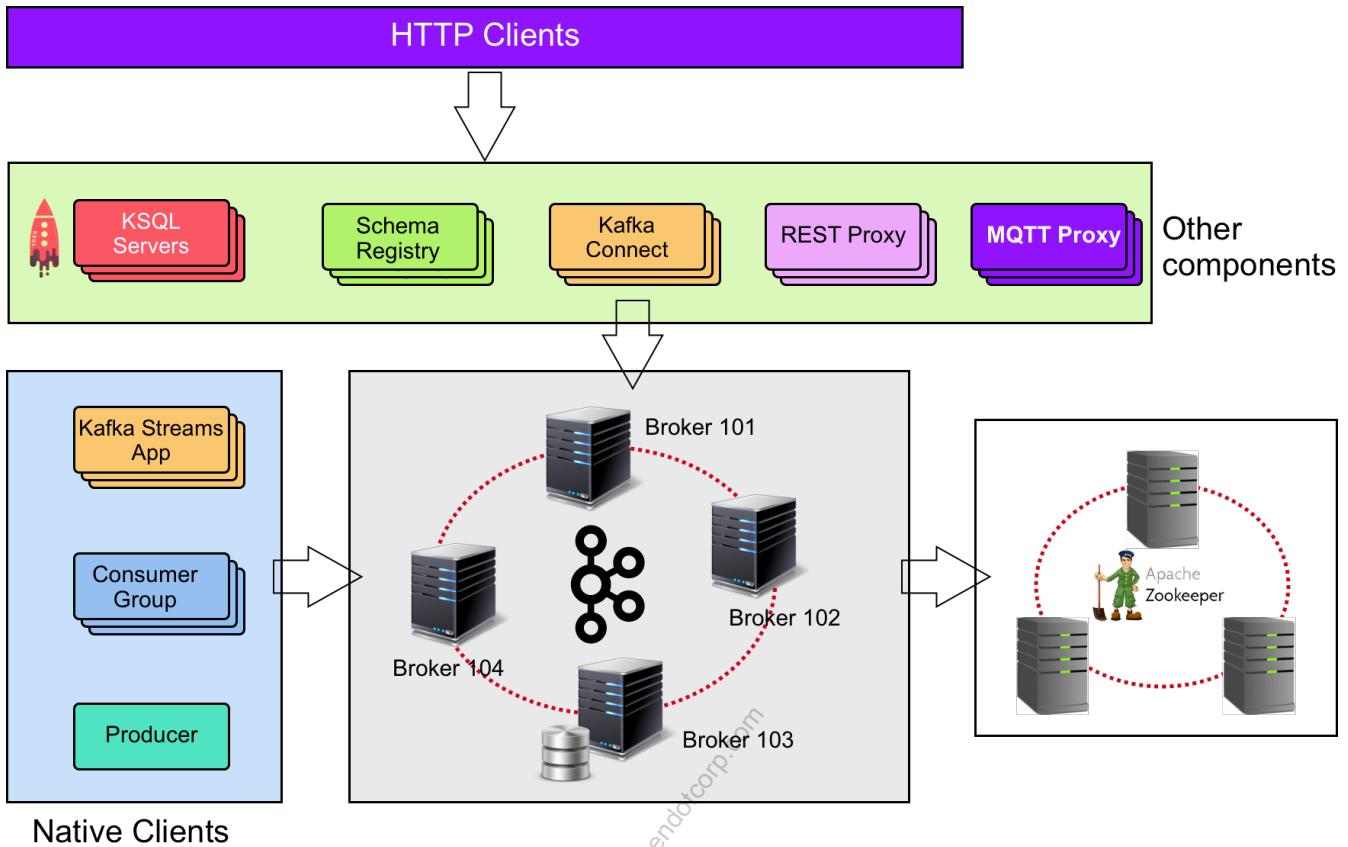
---



- Kafka Log Files
- Replicas for Reliability
- Partitions and Consumer Groups for Scalability
- Security ... ↵
- 🧐 Hands-on Lab

ybhandare@greendotcorp.com

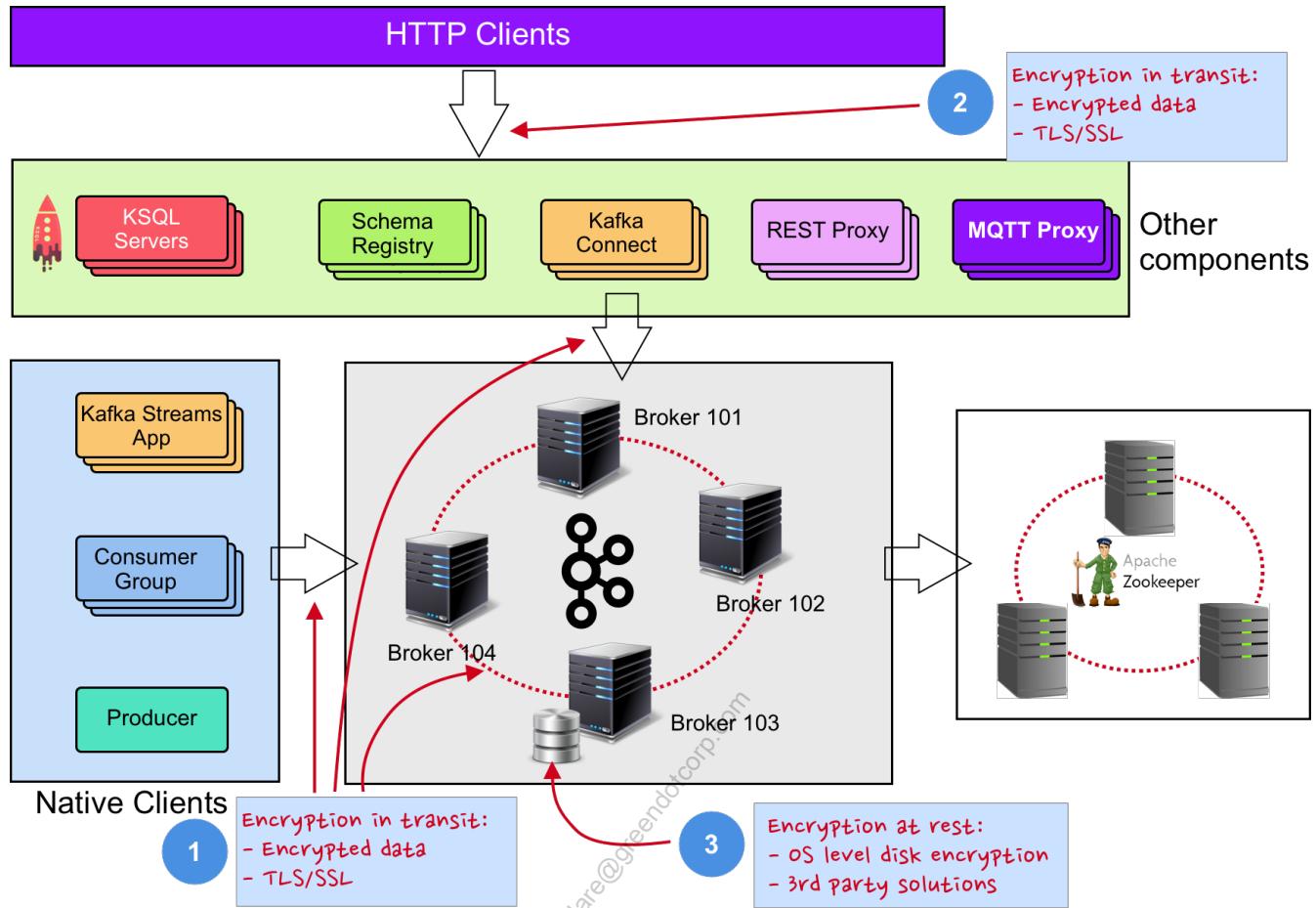
# Security - Architecture



Here we show an overview over a potential Confluent real-time streaming platform. We have the following components:

- At the center lies the Kafka cluster with its 1 to many brokers
- Right of it we have the ZooKeeper ensemble, typically 3 to 5 instances for high availability
- On the left hand side we have the so called native Kafka clients: producers, consumer groups, Kafka Streams applications and KSQL Server clusters
- On top we have, what I call, middle ware: the Confluent Schema Registry, Kafka Connect and the Confluent REST Proxy
- Finally at the very top, in purple we have the HTTP clients that access the middle ware via REST API

# Security - Encryption at Rest & in Transit



The Kafka cluster, including the ZooKeeper ensemble can be secured as follows:

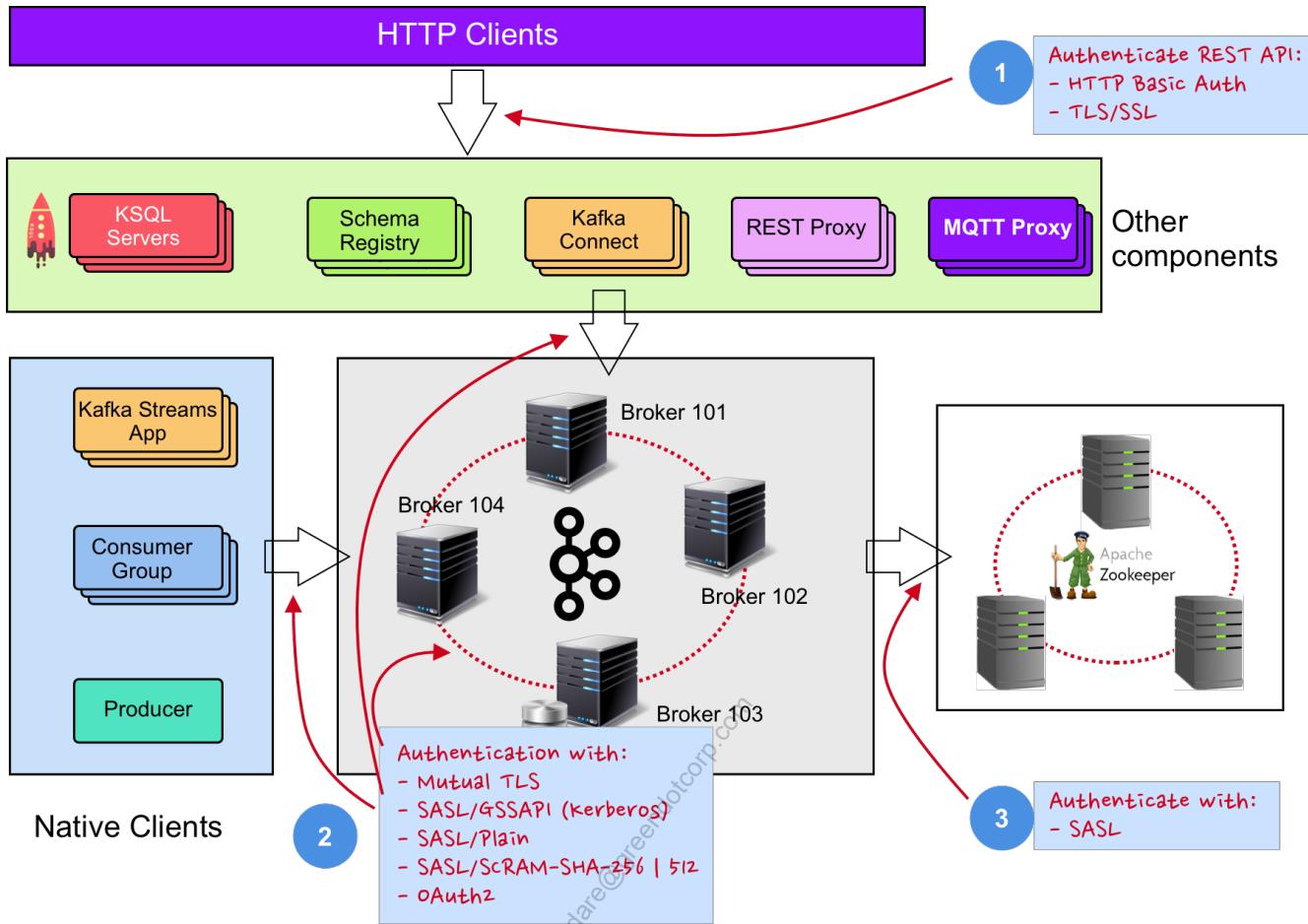
1. Encryption **in-transit** (or **in-flight**) is mainly achieved in the following 2 ways:
  - the producer encrypts the data before sending it to the broker. The consumer decrypts this data
  - using (mutual) TLSThe above applies for both situations:
  - client → broker
  - broker → broker
2. Encryption between HTTP clients and Confluent Platform services such as Schema Registry happens in the following ways
  - Using TLS/SSL
  - Using the Confluent Enterprise security plugin
3. Encryption at rest is implemented either by using OS level disk partition encryption or by using 3rd party services such as the Vorometric Data Security manager



Although using encryption/decryption at the client level normally inhibits us from working with KSQL, an engineer from Confluent created UDFs which support at-rest encryption/decryption at the field level in a proof of concept (POC), showing that it can be done.

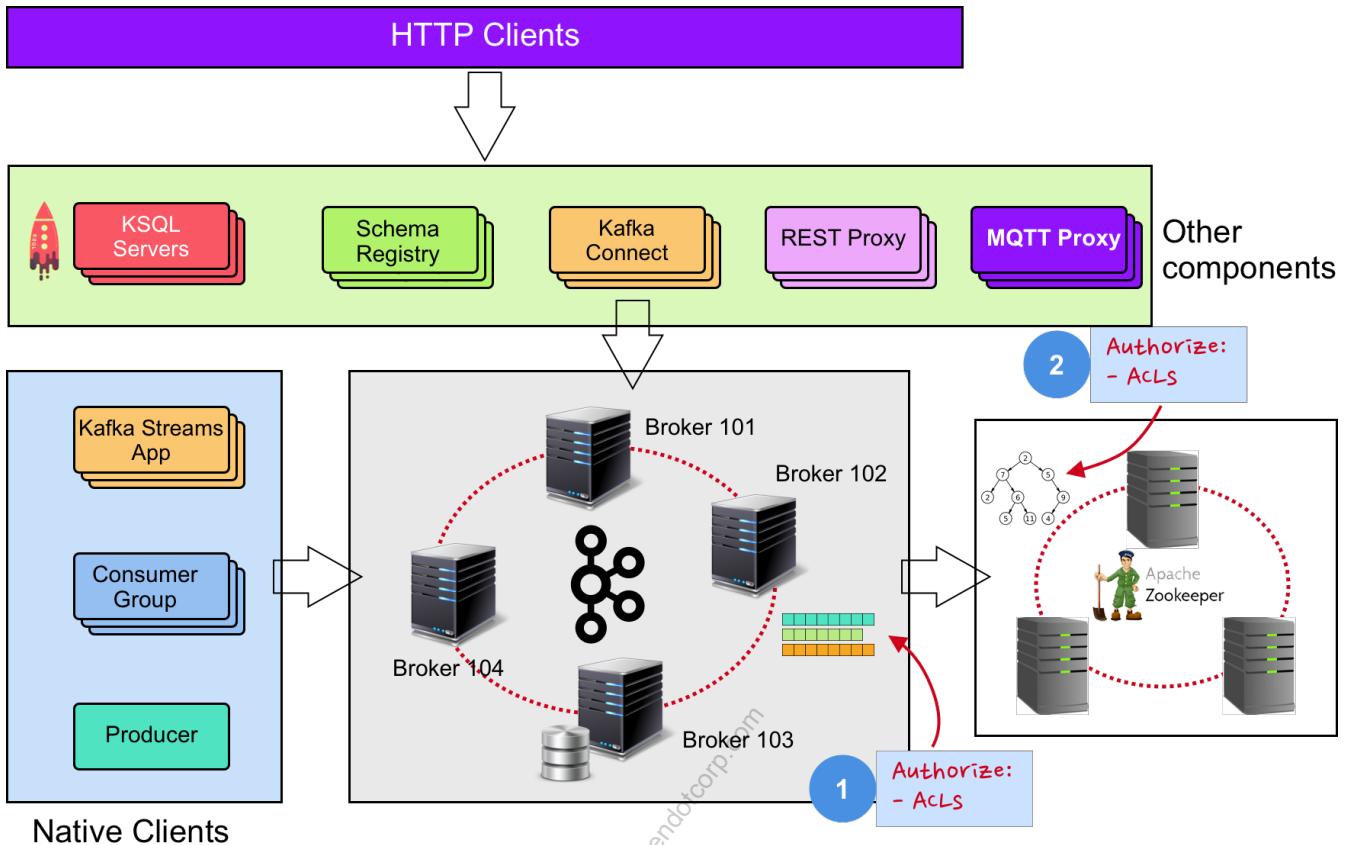
ybhandare@greendotcorp.com

# Security - Authentication



1. HTTP Clients can authenticate themselves to the REST APIs of the Schema Registry, Kafka Connect Workers, REST Proxy or KSQL Server via:
  - HTTP Basic Auth
  - SASL
2. Intra-broker and client to broker authentication happens via:
  - **MTLS**: Mutual TLS/SSL
  - SASL/Plain:
  - SASL/GSSAPI (Kerberos):
  - SASL/SCRAM-SHA-256 and SASL/SCRAM-SHA-512:
  - OAuth2
3. Brokers authenticate themselves to the ZooKeeper ensemble via SASL

# Security - Authorization



1. All relevant resources on the Kafka cluster, such as topics can be protected by Access Control Lists (ACLs). All clients accessing the Kafka cluster need the corresponding rights to execute `create`, `read` and/or `write` operations on those resources.

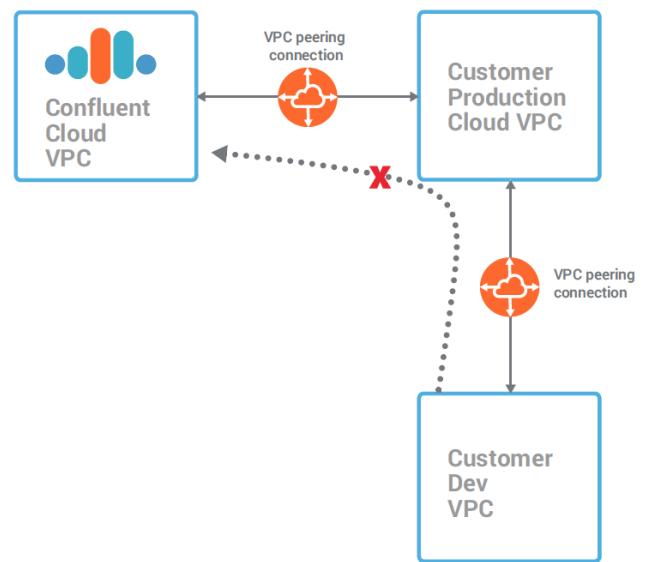


Impersonation is not possible at this time, that is, e.g. the identity of a client accessing the KSQL API is not passed through to the Kafka cluster, but rather the KSQL Server identifies itself (with its technical account) to the Kafka cluster.

2. All nodes containing meta data about the Kafka cluster, stored in ZooKeeper, can be protected by ACLs. The brokers need the corresponding rights to read and/or write from and to those nodes

## Security - Confluent Cloud

- Security enabled **Out-of-the-Box**
- Storage isolation
- VPC peering possible
- User → Cloud communication secured by TLS
- Data encrypted in motion & at rest
- CCloud is hosted in multiple Azure, AWS & GCP regions



- Confluent Cloud has all security features enabled out-of-the-box, with no extra effort and at no extra cost, and provides full SOC-2 and PCI Level-1 compliance (HIPAA coming soon!).
  - Confluent Cloud Enterprise customers also have storage isolation and they can choose to create VPC peering connection between Confluent Cloud and their own VPCs for an extra layer of security.
  - Customers access Confluent Cloud via three main interfaces:
    - Confluent Cloud web-based user interface
    - Confluent Cloud command line client
    - Apache Kafka protocol
- All communication between users and Confluent Cloud is secured using TLS encryption.
- **Encryption:** Confluent Cloud encrypts all data in motion and at rest.
  - **Physical security:** Confluent Cloud is hosted in multiple AWS and GCP regions



### Questions:

- How does Kafka store its data?
- How does Kafka achieve:
  - scalability
  - reliability
- What happens if you have more consumers in a consumer group than partitions in the consumed topic?

- Kafka uses commit logs to store all its data
  - These allow the data to be read back by any number of Consumers
- Topics can be split into Partitions for scalability
- Partitions are replicated for reliability
- Consumers can be collected together in Consumer Groups
  - Data from a specific Partition will go to a single Consumer in the Consumer Group
- If there are more Consumers in a Consumer Group than there are Partitions in a Topic, some Consumers will receive no data

ybhandare@gridnode.com

## Hands-On Lab

---

- In this Hands-On Exercise, you will create a Topic with multiple Partitions, write data to the Topic, then read the data back to see how ordering of the data is affected
- Please refer to the lab **Consuming from Multiple Partitions** in Exercise Book



ybhandare@greendotcorp.com



### Developing With Kafka

ybhandare@greendotcorp.com

# Agenda

---



1. Introduction
2. Fundamentals of Apache Kafka
3. Kafka's Architecture
4. Developing With Kafka ... ←
5. More Advanced Kafka Development
6. Schema Management In Kafka
7. Data Pipelines with Kafka Connect
8. Stream Processing with Kafka Streams
9. Stream Processing with KSQL
10. Event Driven Architecture
11. Confluent Cloud
12. Conclusion
13. Appendix: Basic Kafka Administration
14. Appendix: Apache Kafka driving ML & Data Analytics

ybhandare@greendotcorp.com

## Learning Objectives

---



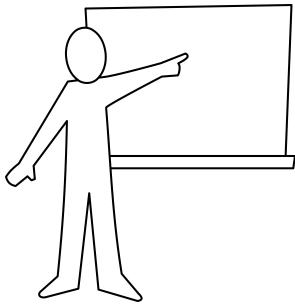
After this module you will be able to:

- Write a Producer using the Java API
- Write a Consumer using the Java API
- Use the Confluent REST proxy to access Kafka from other languages

ybhandare@greendotcorp.com

## Module Map

---



- Programmatically Accessing Kafka ... ←
- Writing a Producer in Java
- Using the REST Proxy to Write a Producer
- Hands-on Lab
- Writing a Consumer in Java
- Using the REST Proxy to Write a Consumer
- Hands-on Lab

ybhandare@greendotcorp.com

## Programmatically Accessing Kafka - The Kafka API

- Kafka includes Java clients in the `org.apache.kafka.clients` package
  - They are available in a JAR which has as few dependencies as possible, to reduce code size
- The non-Java clients are based on `librdkafka` which provides consistent APIs and semantics, high performance and high-quality clients in various programming languages. C/C++ client calls the library methods directly (e.g. developer calls `rd_kafka_consumer_poll()`), whereas Python, Go, .NET clients use librdkafka internally (e.g. developer calls `consumer.poll()` which calls `rd_kafka_consumer_poll()` underneath).
- C# is sometimes used interchangeably with .NET. C# is the programming language, .NET is the application framework and runtime



All features are supported by the Java clients. Other clients have varying levels of feature support (e.g., EOS only available for the Java client).

- Since Kafka 0.9, Kafka includes Java clients in the `org.apache.kafka.clients` package. These are intended to supplant the older Scala clients, which required multiple libraries.

More info here: <https://docs.confluent.io/current/clients/index.html#feature-support>)

# Native Clients

**Confluent Platform**



**Apache Kafka**



**Community Supported**

Proxy http/REST



---



---

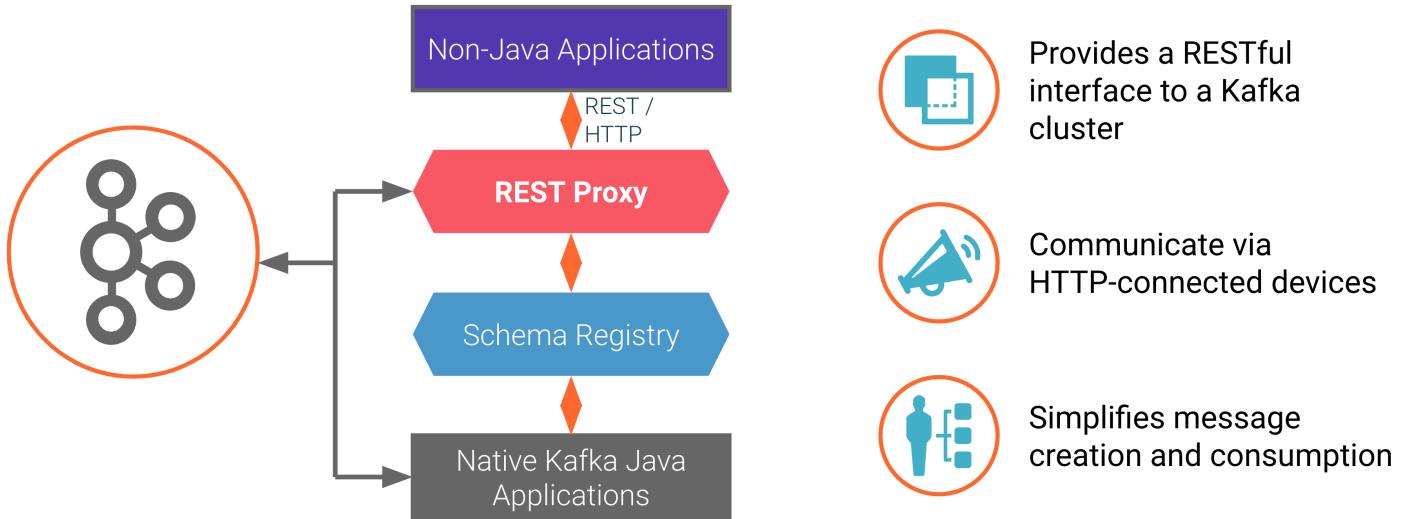
stdin/stdout



**Confluent Platform** Clients developed and fully supported by Confluent

In Kafka the communication between the clients and the servers is done with a simple, high-performance, language agnostic TCP protocol. This protocol is versioned and maintains backwards compatibility with older version.

- Kafka provides a Java client
- Confluent Platform provides Kafka integration using C/C++, Python, Go, and .NET
- There are additional “community supported” clients in different languages

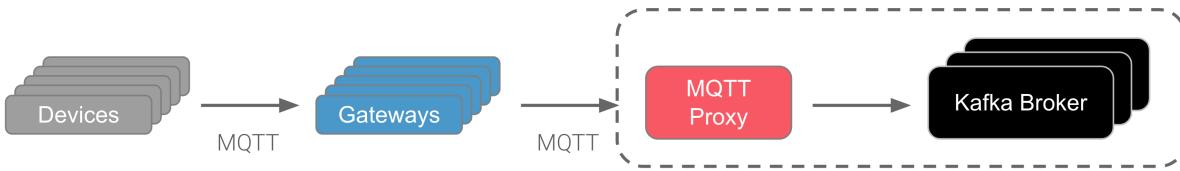


- Confluent Open Source also includes a REST Proxy for Kafka. This allows **any language to access Kafka** via REST. We will discuss REST Proxy in much detail in a later module
- Confluent Enterprise also includes an **MQTT proxy** to allow direct ingestion of IoT data. This will also be discussed in a later module

Three uses for the REST Proxy:

1. For remote clients (such as outside the datacenter including over the public internet)
2. For internal client apps that are written in a language not supported by native Kafka client libraries (i.e. other than Java, C/C++, Python, and Go)
3. For developers or existing apps for which REST is more productive and familiar than Kafka Producer/Consumer API.

## MQTT Proxy



**Connect IoT data sources** leveraging all of your infrastructure investments



**Ensure IoT data delivery** at all QoS levels (QoS0, QoS1 and QoS2) of the MQTT protocol



**Reduce operational cost and complexity** by eliminating third party MQTT brokers and their intermediate storage and lag

- Confluent Enterprise includes an MQTT (Message Queuing Telemetry Transport) proxy to allow direct ingestion of IoT data
- The MQTT proxy is intended to allow customers to add Kafka to their IoT architecture to enable stream processing. MQTT is a pub/sub messaging transport protocol (the de facto standard for IoT devices) that is designed to support thin lightweight clients that run in a very diverse set of devices and communicate asynchronously with scalable, more heavy weight, protocol aware MQTT brokers. Without this proxy, customers would have to maintain separate MQTT clients and brokers, adding layers and complexity to their architectures. The current implementation of the proxy is read-only (i.e., producer only).

## Our Class Environment

---

- During the course this week, we anticipate that you will be writing code in...

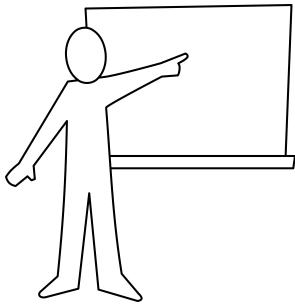
Java	using Kafka's Java API
.NET	using Confluent's .NET client
Python	using Confluent's Python client
Node JS	using the Node JS community client
Other	<i>If you wish to use some other programming language to access the Confluent REST Proxy, you can do so, but be aware that your instructor may not be familiar with your language of choice, though</i>

All the Python examples in these slides use Python as a way to send REST calls to the REST proxy, and the examples are NOT using the Python client

ybhandare@greendotcorp.com

# Module Map

---



- Programmatically Accessing Kafka
- Writing a Producer in Java ... ←
- Using the REST Proxy to Write a Producer
- Hands-on Lab
- Writing a Consumer in Java
- Using the REST Proxy to Write a Consumer
- Hands-on Lab

ybhandare@greendotcorp.com

## Writing a Producer in Java

---

- To create a Producer, use the `KafkaProducer` class
- This is thread safe; sharing a single Producer instance across threads will typically be faster than having multiple instances
- Create a `Properties` object, and pass that to the Producer
  - You will need to specify one or more Broker host/port pairs to establish the initial connection to the Kafka cluster
    - The property for this is `bootstrap.servers`
    - This is only used to establish the initial connection
    - The client will use all servers, even if they are not all listed here
    - Question: why not just specify a single server?

You don't want to specify just a single server for `bootstrap.servers` in case that system is down when the client starts – in which case, the client won't be able to connect at all. Specifying more than one gives you a better chance that at least one will be available. You don't need to specify all of them, but it certainly doesn't hurt to do so.

## Producers - Important Configuration Elements (1)

Name	Description
<code>bootstrap.servers</code>	List of Broker host/port pairs used to establish the initial connection to the cluster
<code>key.serializer</code>	Class used to serialize the key. Must implement the <code>Serializer</code> interface
<code>value.serializer</code>	Class used to serialize the value. Must implement the <code>Serializer</code> interface
<code>compression.type</code>	How data should be compressed. Values are <code>none</code> , <code>snappy</code> , <code>gzip</code> , <code>lz4</code> , <code>zstd</code> . Compression is performed on batches of records

- Compression is end-to-end
  - Compressed on the Producer, stored in compressed format on the Broker, decompressed on the Consumer
- Question: can you enable compression on existing topic with uncompressed data?
  - Answer: Yes, thought this is a trick question because compression is enabled per-message, not per-topic. You can have one producer compressing and another producer not compressing messages, both writing to same topic.

A key serializer must be specified even if you do not intend to use keys.

## Producers - Important Configuration Elements (2)

Name	Description
acks	<p>Number of acknowledgment the Producer requires the leader to have before considering the request complete. This controls the <b>durability</b> of records.</p> <ul style="list-style-type: none"><li>• <b>acks=0</b>: Producer will <b>not wait</b> for any acknowledgment from the server</li><li>• <b>acks=1</b>: Producer will wait until the <b>leader</b> has written the record to its local log</li><li>• <b>acks=all</b>: Producer will wait until <b>all in-sync replicas</b> have acknowledged receipt of the record</li></ul>

The **acks** setting is used to determine if messages are transferred successfully from the producer to the brokers. If a setting other than 0 is used, it is a best practice to set the **retries** to a non-zero value to handle transient failures.

ybhandare@greendotcorp.com

## Creating the Properties and KafkaProducer Objects

```
1 Properties props = new Properties();
2 props.put("bootstrap.servers", "broker1:9092");
3 props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
4 props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
5
6 KafkaProducer<String, String> producer = new KafkaProducer<>(props);
```

- Other serializers available: `ByteArraySerializer`, `IntegerSerializer`, `LongSerializer`
- `StringSerializer` encoding defaults to UTF8
  - Can be customized by setting the property `serializer.encoding`

<https://www.javadoc.io/doc/org.apache.kafka/kafka-clients/2.3.0>

Note that the key and value data types are specified as part of the `KafkaProducer`. For the initial examples and exercises, we will use simple data types. Complex data types will be addressed when we discuss the Schema Registry later in the course.

## Helper Classes



### Dangerous

```
props.put("bootstrap.servers", "broker1:9092");
props.put("key.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");
```



### Safer

```
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
```

- Helper classes `ProducerConfig`, `ConsumerConfig`, and `StreamsConfig`

The use of helper classes allow the IDE to provide suggestions as the code is generated so that all the variables do not need to be memorized. Additionally, the helper classes will make errors apparent at compile time rather than run time (as would be the case if the direct variable names were used).

Even worse, with literal strings, it's possible that no runtime error occurs, and the property specified is simply ignored because of a typo and a default value used instead, resulting in a running application with incorrect behavior.

## Sending Messages to Kafka

```
1 String k = "mykey";
2 String v = "myvalue";
3 ProducerRecord<String, String> record = new ProducerRecord<String, String>("my_topic", k, v); ①
4 producer.send(record); ②
```

- ① `ProducerRecord` can take an optional timestamp if you don't want to use current system time
- ② Alternatively:

```
producer.send(new ProducerRecord<String, String>("my_topic", k, v));
```

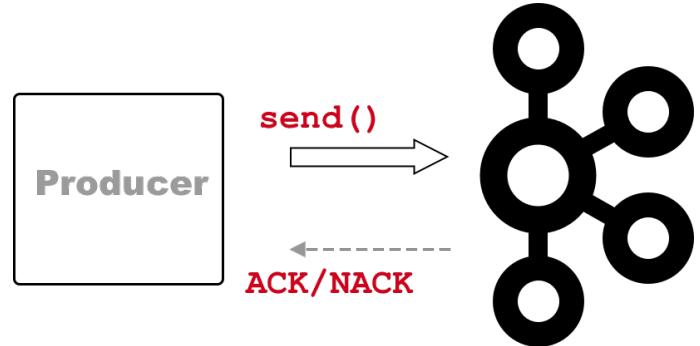
The `ProducerRecord` constructor encapsulates the provided key and value into a record (message) complete with headers. When `send()` is called, the `KafkaProducer` serializes the key and value and runs the default Partitioner (to determine which partition the message will store the message in the topic) using the data provided in the `ProducerRecord`.

ybhandare@greendotcorp.com

# Asynchronous Sending

KafkaProducer::send()...

- is **asynchronous**
- thus **non blocking**
- returns **Future** with **RecordMetadata**
- force blocking:  
`producer.send(...).get()`



- The **send()** call is asynchronous
  - It does not block; it returns immediately and your code continues
- It returns a **Future** which contains a **RecordMetadata** object
  - The Partition the record was put into and its offset
- To force **send()** to block, call `producer.send(record).get()`

The **send()** method does not actually push the message to the brokers but instead places the message in a binary queue in local memory. There is a separate binary queue for each partition in the topics that the producer communicates with. This approach enables the producer to handle messages either individually (for real-time) or as groups (for batching). Batching is also the reason why it is non-blocking - multiple sends need to happen in order to have multiple messages to batch together.

If a return value is needed for the **send**, the method can be configured with a **Future** which will either return the metadata for the successfully written data or an exception.

## When Do Producers Actually Send?

- A Producer `send()` returns immediately after it has added the message to a local buffer of pending record sends
  - This allows the Producer to send records in batches for better performance
- Then the Producer flushes multiple messages to the Brokers based on batching configuration parameters
- You can also manually flush by calling the Producer method `flush()`

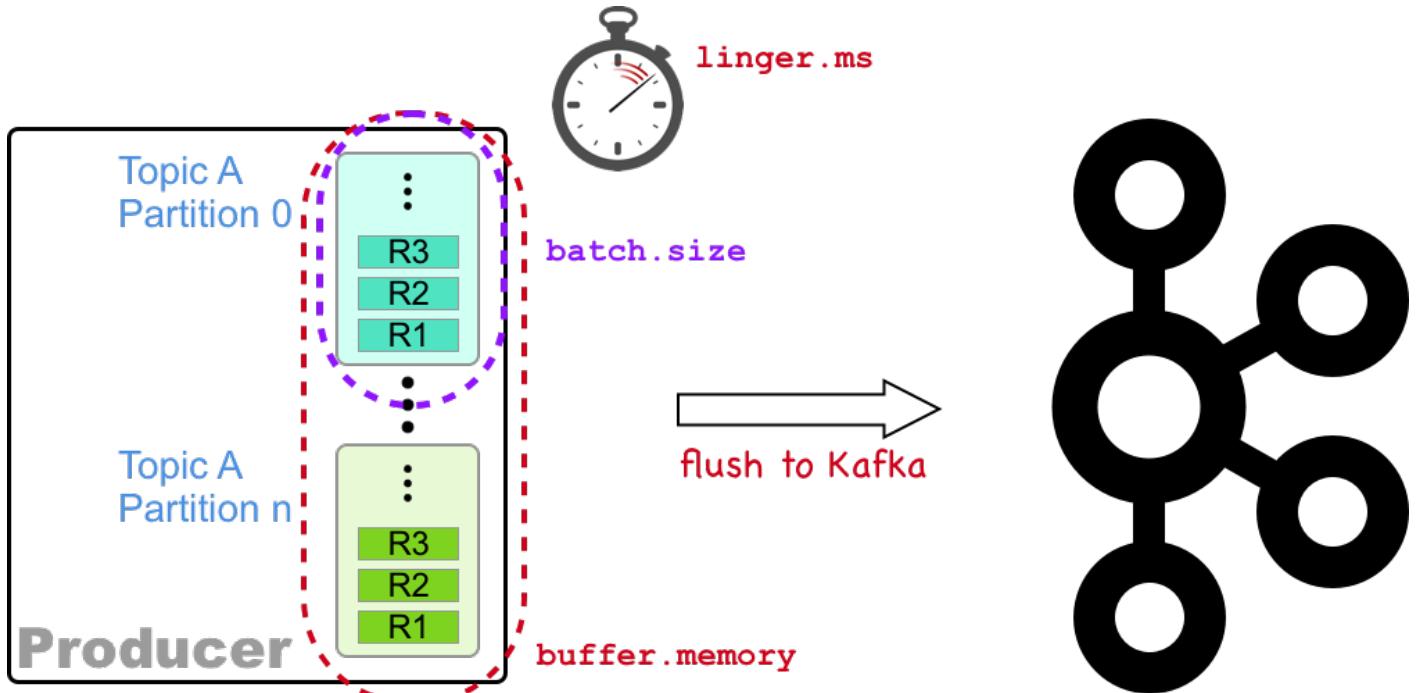
The records are actually pushed to the brokers by an internal background thread that will be triggered by configurable batching thresholds. It is this action that is waiting for the acks from the brokers (as configured by the `acks` setting), not the `send()` method.

There is also a manual `flush()` method that can be called in the Producer code. However, this method pushes all the binary queues at once and is synchronous. The internal background thread can push individual queues as they hit their batching thresholds.



- Do not confuse the Producer method `flush()` in a Producer context with the term `flush` in a Broker context
- `flush` in a Broker context refers to when messages get written **from the page cache to disk**

## Performance Tuning Batching



- Producers can adjust batching configuration parameters
  - `batch.size` message batch size in bytes (Default: 16KB)
  - `linger.ms` time to wait for messages to batch together (Default: 0, i.e., send immediately)
    - High throughput: large `batch.size` and `linger.ms`, or flush manually
    - Low latency: small `batch.size` and `linger.ms`
  - `buffer.memory` (Default: 32MB)
    - The Producer's buffer for messages to be sent to the cluster
    - Increase if Producers are sending faster than Brokers are acknowledging, to prevent blocking

The internal thread which pushes data from the producer to the brokers is triggered by two thresholds: `batch.size` and `linger.ms`. Batching provides higher throughput due to larger data transfers and fewer RPCs but at the cost of higher latency due to the time it takes to accumulate the batches.



We have multiple buffers, one for each partition, but they all fit in the same area as defined by the `buffer.memory` parameter.

## Performance Tuning Batching

---

The `buffer.memory` setting defines how much of the producer's memory can be used for the binary queues. This setting may need to be adjusted for reasons including:

- Caching because of retries and/or slow brokers
- Larger batching size (since more data will be buffered on the Producer)
- Large message sizes (above the default maximum message size)
- Topics with large partition counts (since queues are defined per partition)

ybhandare@greendotcorp.com

## Producer Retries

---

- Developers can configure the `retries` configuration setting in the Producer code
  - `retries`: how many times the Producer will attempt to retry sending records
    - Only relevant if `acks` is not 0
  - Hides transient failure
  - Ensure lost messages are retried rather than just throwing an error
    - `retries`: number times to retry (Default: `MAX_INT`)
    - `retry.backoff.ms`: pause added between retries (Default: 100)
    - For example, `retry.backoff.ms=100` and `retries=600` will retry for 60 seconds

Retries only matter for producers with `acks` set to 1 or all. The action that is retried is the push from the producer buffer, not the `send()`.

The example `retries` and `retry.backoff.ms` combination assumes reasonable Broker failure time of 10 seconds

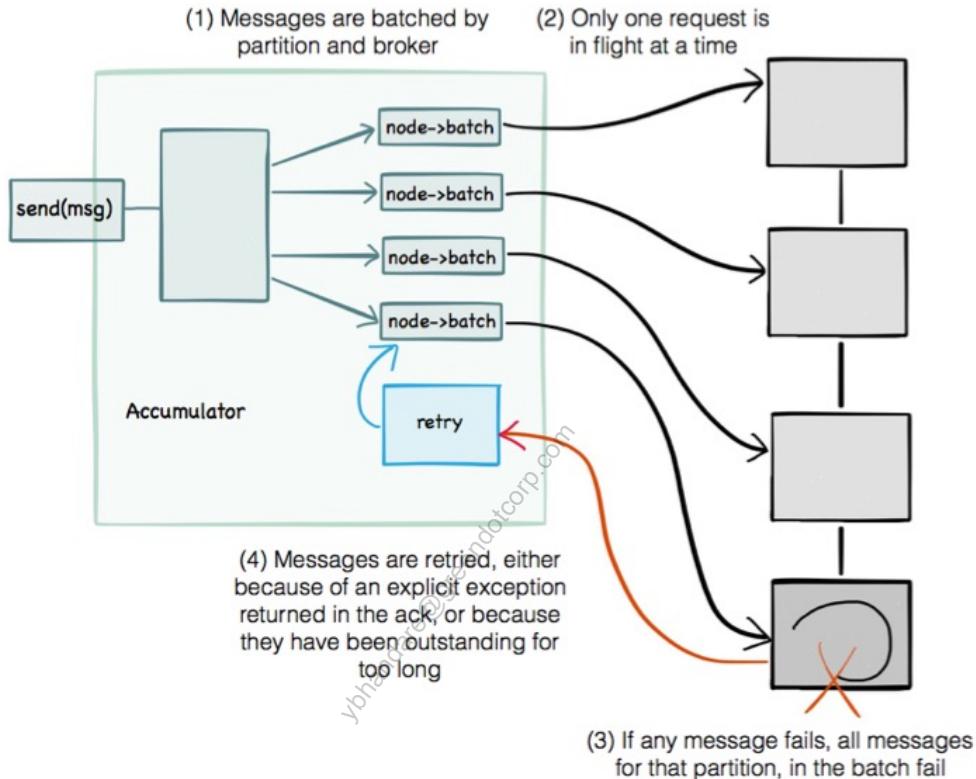
ybhandare@greendotcorp.com

## Preserve Message Send Order

- Use **idempotent** producers, or
- Set `max.in.flight.requests.per.connection=1` (Default: 5)



May impact throughput due to lack of request pipelining



The term "in-flight request" means a produce request that has not been acknowledged by the broker.

If `retries > 0` and `max.in.flight.requests.per.connection > 1`, then out-of-order messages may happen. For example, assume you send message batches m1, m2, and m3 with `max.in.flight.requests.per.connection=3`. If all three message batches are in-flight but only m2 fails, then only m2 is retried. However, since m1 and m3 are already in-flight they will likely arrive first. The message batches will arrive in the order m1, m3, m2 – out of order.

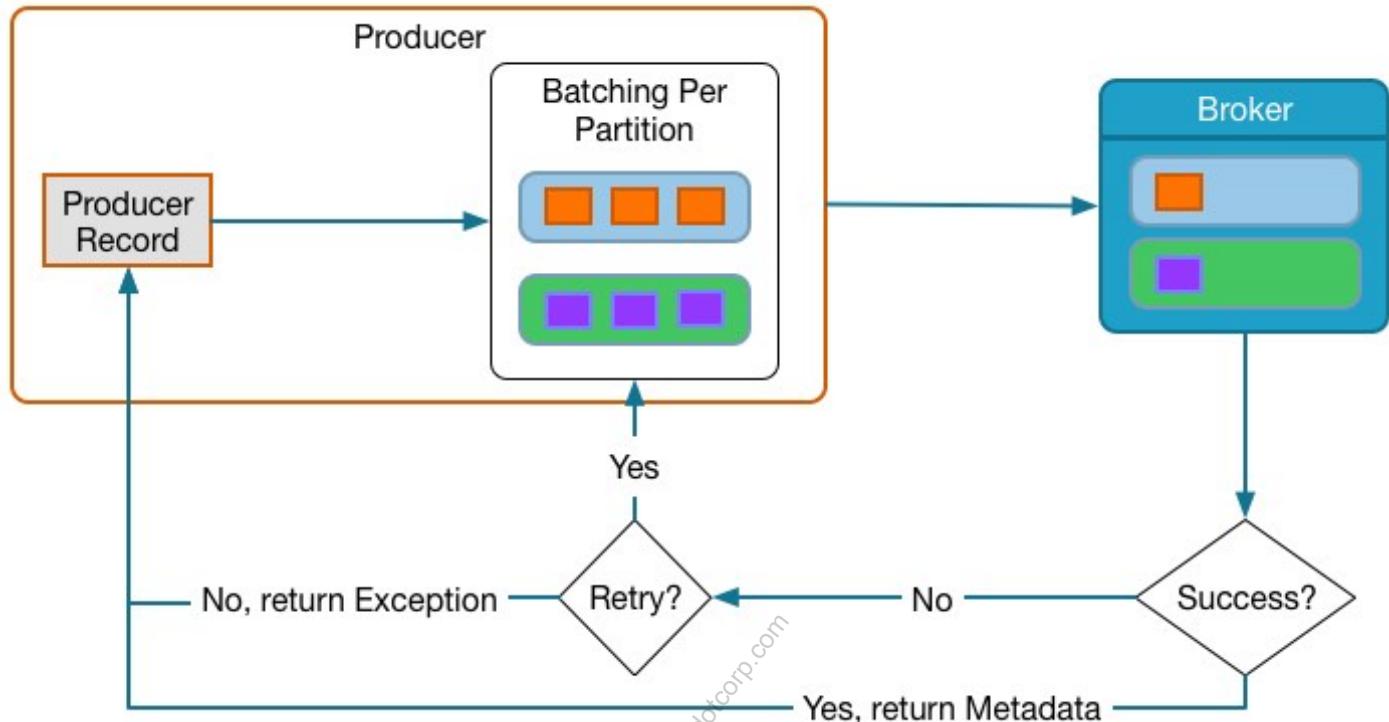
Note that `retries > 0` is "at least once" delivery and may introduce duplicates, especially if used with `acks=all`.



If using an idempotent producer `max.inflight.requests` does not have to be set to 1. This setting has no noticeable effect on throughput and latency under normal circumstances.

ybhandare@greendotcorp.com

## Batching and Retries



This diagram summarizes the flow described in the previous slides. The mechanism which results in the return values going to the Producer Record object is actually the `Future` that is returned to the `send()` method. The implementation of the `Future` will be discussed in detail later in this chapter.

## send() and Callbacks (1)

- `send(record)` is equivalent to `send(record, null)`
- Instead, it is possible to supply a `Callback` as the second parameter
  - This is invoked when the send has been acknowledged
  - It is an Interface with an `onCompletion` method:

```
onCompletion(RecordMetadata metadata, java.lang.Exception exception)
```

- `Callback` parameters
  - `metadata` will be `null` if an error occurred
  - `exception` will be `null` if no error occurred

For situations where an application needs to verify the successful transfer of individual messages, the `send()` method can be configured with an `onCompletion()` method as the callback. This method will return one of two results: a `RecordMetadata` object if the message was written successfully to a partition or an `Exception` if the transfer fails. You will never get back both.

## send() and Callbacks (2)

- Parameters correlated to a particular record can be passed into the Callback's constructor
- Example code, with lambda function and closure instead of requiring a separate class definition

```
1 producer.send(record, (recordMetadata, e) -> {
2     if (e != null) {
3         e.printStackTrace();
4     } else {
5         System.out.println("Message String = " + record.value() +
6                             ", Offset = " + recordMetadata.offset());
7     }
8});
```

To correlate the callback to a particular record (e.g. if `send()` fails), you can use any information available in the closure for the callback you instantiate for the record (a closure is a function that can be passed as an object and has access to previously scoped variables), or passed directly to the callback's constructor if you don't use a lambda function.

## Closing the Producer

```
producer.close();
```

blocks until all previously sent requests complete

```
producer.close(timeout, timeUnit);
```

wait until complete or given timeout expires

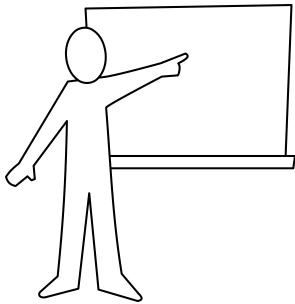
**close(timeout, timeUnit):**

- waits up to timeout for the producer to complete the sending of all incomplete requests
- If the producer is unable to complete all requests before the timeout expires, this method will fail any unsent and unacknowledged records immediately

ybhandare@greendotcorp.com

# Module Map

---

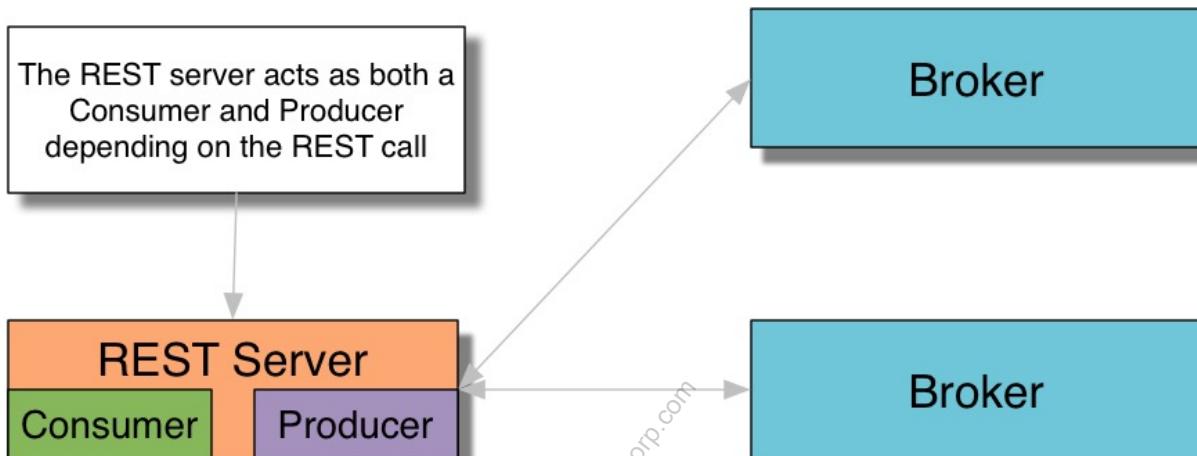


- Programmatically Accessing Kafka
- Writing a Producer in Java
- Using the REST Proxy to Write a Producer ... ←
- Hands-on Lab
- Writing a Consumer in Java
- Using the REST Proxy to Write a Consumer
- Hands-on Lab

ybhandare@greendotcorp.com

## About the Confluent REST Proxy

- The Confluent REST Proxy allows you to use HTTP to perform actions on the Kafka cluster
- The REST calls are translated into native Kafka calls
- This allows virtually any language to access Kafka
- Uses POST to send data to Kafka
  - Embedded formats: JSON, base64-encoded JSON, or Avro-encoded JSON
- Uses GET to retrieve data from Kafka



The Confluent REST Proxy is part of Confluent Open Source and Confluent Enterprise distributions. The proxy provides a RESTful interface to a Kafka cluster, making it easy to produce and consume messages, view the state of the cluster, and perform administrative actions without using the native Kafka protocol or clients.

Some example use cases are:

- Reporting data to Kafka from any frontend app built in any language not supported by official Kafka clients
- Ingesting messages into a stream processing framework that doesn't yet support Kafka
- Scripting administrative actions



The configuration files for the REST proxy are preconfigured in our lab environments. Configuring the REST proxy is not part of this course. Refer students to the documentation for more details if they ask:  
<https://docs.confluent.io/current/kafka-rest/docs/deployment.html>

# A Python Producer Using the REST Proxy

```
1 #!/usr/bin/python
2
3 import requests
4 import json
5
6 url = "http://restproxy:8082/topics/my_topic"
7 headers = {
8     "Content-Type" : "application/vnd.kafka.json.v2+json"
9 }
10 # Create one or more messages
11 payload = {"records":
12     [
13         {"key": "firstkey",
14          "value": "firstvalue"
15     }]
16 # Send the message
17 r = requests.post(url, data=json.dumps(payload), headers=headers)
18 if r.status_code != 200:
19     print "Status Code: " + str(r.status_code)
20     print r.text
```

The target topic is specified as part of the URL, after the REST proxy.

This example shows how to use the JSON format. Refer to the [docs.confluent.io](https://docs.confluent.io) for examples using the other formats (e.g., Avro-encoded JSON).



A question that often comes up during class wrt REST Proxy - how are all the producer/consumer config settings we talked about used through the REST Proxy?

**Answer:** in almost all cases, the configs have to be defined on the REST server, and cannot be specified as part of the client request.

## Hands-On Lab

---

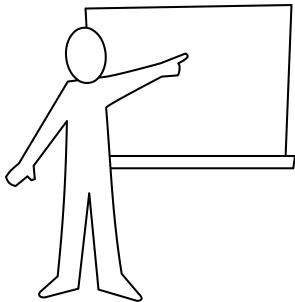
- In this Hands-On Exercise, you will write a Kafka Producer either in Java or Node JS
- Please refer to the following labs in the Exercise Book:
  - **Creating a Kafka Producer in Java**
  - Optional: **Creating a Kafka Producer in Node JS**



ybhandare@greendotcorp.com

# Module Map

---



- Programmatically Accessing Kafka
- Writing a Producer in Java
- Using the REST Proxy to Write a Producer
- Hands-on Lab
- Writing a Consumer in Java ... ←
- Using the REST Proxy to Write a Consumer
- Hands-on Lab

ybhandare@greendotcorp.com

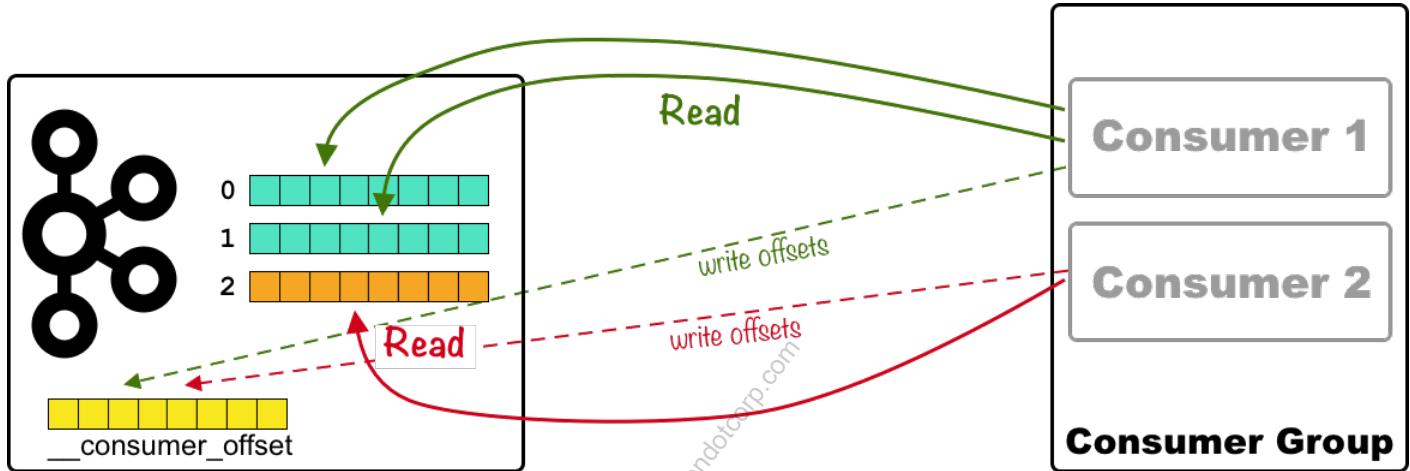
## Consumers and Offsets

- Each message in Partition has offset
- Kafka tracks consumer offset
- Offset for each triple (`group.id,topic,partition`) is tracked
- Offsets tracked in topic `__consumer_offsets`

- Consumer offsets **auto-committed** by default



Consumer Offset refers to **next message** to read



- Each message in a Partition has an offset, the numerical value indicating where the message is in the log
- Kafka tracks the Consumer Offset for each partition of a Topic the Consumer (or Consumer Group) has subscribed to in the topic `__consumer_offset` (the pair `<Consumer Group, partition>` is tracked)
- Consumer offsets are committed automatically by default. We will see later how to manually commit offsets if you need to do that



The Consumer Offset is the value of the next message the Consumer will read, **not** the last message that has been read. For example, if the Consumer Offset is 9, this indicates that messages 0 to 8 have already been processed, and that message 9 will be the next one sent to the Consumer

Automatic consumer offset commits are the default behavior for the consumer API. Some students may ask about other options - defer those questions until the next chapter (advanced development).

## Important Consumer Properties

- Important Consumer properties include:

Name	Description
<code>bootstrap.servers</code>	List of Broker host/port pairs used to establish the initial connection to the cluster
<code>key.deserializer</code>	Class used to deserialize the key. Must implement the <code>Deserializer</code> interface
<code>value.deserializer</code>	Class used to deserialize the value. Must implement the <code>Deserializer</code> interface
<code>group.id</code>	A unique string that identifies the Consumer Group this Consumer belongs to.
<code>enable.auto.commit</code>	When set to <code>true</code> (the default), the Consumer will trigger offset commits based on the value of <code>auto.commit.interval.ms</code> (default 5000ms)

- As with the producers, even if the key is not used by your environment, the clients must be configured as if it is so the `key.deserializer` is a required setting.
- The `group.id` is used to create the consumer groups – this must be the same on all members of the group.
- For students wishing to disable the automatic offset commit feature, this can be done by setting `enable.auto.commit=false`. The default interval is 5 seconds.

## Consumer Configuration

```
1 Properties props = new Properties();
2 props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
3 props.put(ConsumerConfig.GROUP_ID_CONFIG, "samplegroup");
4 props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
5 props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
```

As with the `KafkaProducer` class, the `KafkaConsumer` is configured with a `Properties` object which supports helper classes.

ybhandare@greendotcorp.com

## Creating the Consumer and Subscribing

---

```
1 KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
2 consumer.subscribe(Arrays.asList("my_topic", "my_other_topic"));
```

- Can subscribe to 1...n Topics
- Calling `subscribe` again will replace existing list of topics
- Can use regular expressions for topic subscription



the `subscribe()` method will replace the entire list of subscribed topics; there is no append option for this command.

ybhandare@greendotcorp.com

## Reading Messages from Kafka with `poll()`

```
1 while (true) { ①
2     ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100)); ②
3     for (ConsumerRecord<String, String> record : records)
4         System.out.printf("offset = %d, key = %s, value = %s\n",
5                            record.offset(), record.key(), record.value());
6 }
```

- ① Loop forever
- ② Each call to `poll` returns a (possibly empty) list of messages.

The `poll(Duration)` method has many functions but the main ones to know about are:

- Fetch records from assigned partitions
- Trigger partition assignment (if necessary)
- Commit consumer offsets if automatic offset commit is enabled and the `auto.commit.interval.ms` is exceeded

ybhandare@greendotcorp.com

## Controlling the Number of Messages Returned

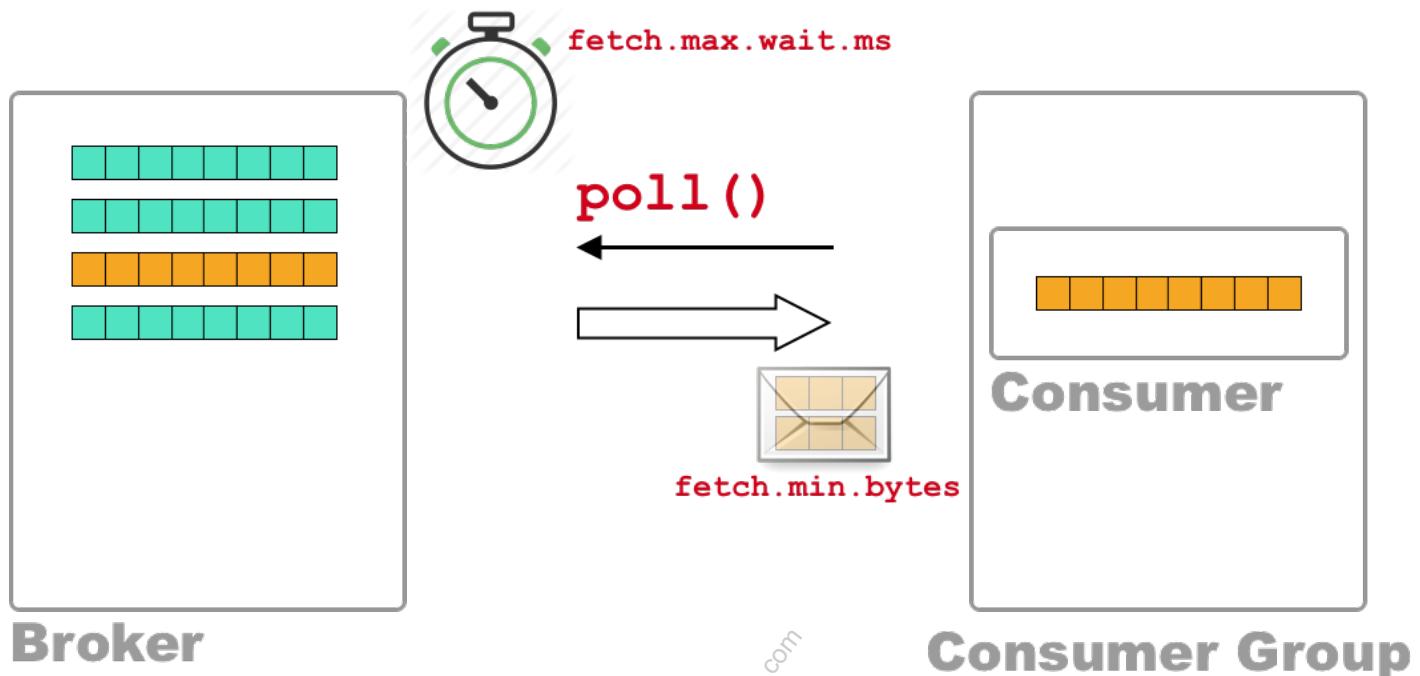
- `poll(Duration)` fetches from **multiple partitions** across **multiple brokers**
- Max. amount of data fetched **per partition**:  
`max.partition.fetch.bytes` (default 1MB)
- Alternatively manage **total** size with:  
`max.poll.records` (default 500)

The goal of a consumer is to consume all messages up to the end of the log. To prevent the consumer from fetching more data than it can reasonably or physically handle, there are two limits that can be set on the consumers:

- `max.partition.fetch.bytes`: Limits the total amount of data that can be fetched per partition
- `max.poll.records`: Limits the number of records that a `poll()` can return, regardless of size, across all partitions assigned to the consumer

The timeout parameter in the call to `poll(Duration)` is across all fetch requests

- This controls the maximum amount of time in milliseconds that the Consumer will block if no new records are available
- This method returns immediately if there are records available. Otherwise, it will await the passed timeout



When tuning the performance of a consumer there are two main properties to configure:

`fetch.min.bytes` minimal amount of data the `poll()` method should wait for, before returning it to the consumer

`fetch.max.wait.ms`: If there is not enough data, then this is the maximum time interval after which the `poll()` request returns with whatever has been accumulated so far. This can be an empty array of messages, if not new data is inflowing.

## High throughput

- get more data in one batch
- large `fetch.min.bytes`
- reasonable `fetch.max.wait.ms`

## Low latency

- get data as quickly as possible
- set `fetch.min.bytes=1` (default)

- Consumption goal: high throughput or low latency?
  - High throughput: get more data at a given time
  - Low latency: send immediately when data is available
- `fetch.min.bytes`, `fetch.max.wait.ms`
  - `fetch.min.bytes` (Default: 1)
    - Broker waits for messages to accumulate to this size batch before responding
  - `fetch.max.wait.ms` (Default: 500)
    - Broker will not wait longer than this duration before returning a batch
- High throughput
  - Large `fetch.min.bytes`, reasonable `fetch.max.wait.ms`
- Low latency
  - `fetch.min.bytes=1`

As with the producer, the consumer can be tuned for batch or real-time processing, depending on whether high throughput or low latency is your goal.

There are additional settings for tuning consumer performance. For further learning, refer to the Operations training class or the Confluent white paper “Optimizing Your Apache Kafka Deployment”.

# Preventing Resource Leaks

- Wrap the code in a `try{ }` block
- **Avoid resource leaks:** Close consumer in `finally{ }` block



`KafkaConsumer` is not thread-safe

```
1 try {  
2     while (true) {  
3         ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));  
4         for (ConsumerRecord<String, String> record : records)  
5             System.out.printf("offset = %d, key = %s, value = %s\n",  
6                                 record.offset(), record.key(), record.value());  
7     }  
8 } finally {  
9     consumer.close();  
10 }
```

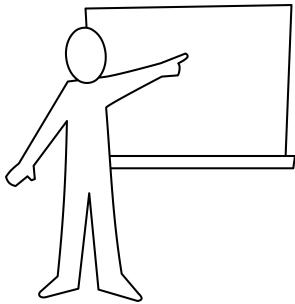
Note: This slide illustrates a simple example. A more complete example would add a shutdown hook and call `consumer.close()` there rather than using a simple `finally{}` block.

Introduced in Kafka 0.10.2 (Confluent 3.2), you can close the consumer with a timeout (consistent with what was already available with the producer):

- `close()`: waits up to the default timeout of 30 seconds for any needed cleanup. If auto-commit is enabled, this will commit the current offsets if possible within the default timeout
- `close(timeout, timeUnit)`: waits up to timeout for the Consumer to complete pending commits and leave the group. If auto-commit is enabled, this will commit the current offsets if possible within the timeout. If the Consumer is unable to complete offset commits and gracefully leave the group before the timeout expires, the Consumer is force closed

# Module Map

---



- Programmatically Accessing Kafka
- Writing a Producer in Java
- Using the REST Proxy to Write a Producer
- Hands-on Lab
- Writing a Consumer in Java
- Using the REST Proxy to Write a Consumer ... ↪
- Hands-on Lab

ybhandare@greendotcorp.com

# A Python Consumer Using the REST Proxy (1)

Main Logic:

```
1 import requests
2 import json
3 import sys
4
5 FORMAT = "application/vnd.kafka.v2+json"
6 POST_HEADERS = { "Content-Type": FORMAT }
7 GET_HEADERS = { "Accepts": FORMAT }
8
9 base_uri = create_consumer_instance("group1", "my_consumer")
10 subscribe_to_topic(base_uri, "hello_world_topic")
11 consume_messages(base_uri)
12 delete_consumer(base_uri)
```

Using the REST Proxy as a consumer has a few more steps than the producer.

First we define a few global variables:

1. `FORMAT`: defines Kafka's specific JSON format in version 2, used in HTTP request
2. `POST_HEADERS`: headers used for `POST` requests
3. `GET_HEADERS`: headers used for `GET` requests

The actual application logic:

1. Create a consumer instance called `my_consumer` in consumer group `group1`
2. Subscribe the consumer to the topic `hello_world_topic`
3. Now consume messages
4. When done, delete the consumer instance to avoid resource leaks



`base_uri` is used to identify the consumer instance

## A Python Consumer Using the REST Proxy (2)

Creating the Consumer Instance:

```
1 def create_consumer_instance(group_name, instance_name):
2     url = f'http://restproxy:8082/consumers/{group_name}'
3     payload = {
4         "name": instance_name,
5         "format": "json"
6     }
7     r = requests.post(url, data=json.dumps(payload), headers=POST_HEADERS)
8
9     if r.status_code != 200:
10        print ("Status Code: " + str(r.status_code))
11        print (r.text)
12        sys.exit("Error thrown while creating consumer")
13
14    return r.json()["base_uri"]
```

In this part of the example, we are creating the instance of the consumer. Notice that the `url` is referencing the consumers rather than a topic name as we saw in the producer example.

1. We define the `url` to the desired consumer group
2. The payload defines among other things the instance name we want to use
3. Now we do an HTTP POST request to the `url`
4. In case of an error we stop the application with `sys.exit(...)`
5. As a last step, we return `base_uri`, which is used to identify the consumer instance

## A Python Consumer Using the REST Proxy (3)

### Subscribing to a topic

```
1 def subscribe_to_topic(uri, topic_name):
2     payload = {
3         "topics": [topic_name]
4     }
5
6     r = requests.post(uri + "/subscription",
7                         data=json.dumps(payload),
8                         headers=POST_HEADERS)
9
10    if r.status_code != 204:
11        print("Status Code: " + str(r.status_code))
12        print(r.text)
13        delete_consumer(uri)
14        sys.exit("Error thrown while subscribing the consumer to the topic")
```

In the body of the POST request we send the list of topics (here only a single one) that we want this consumer to subscribe to.

If there is an error, we delete the consumer and then exit the application

## A Python Consumer Using the REST Proxy (4)

Consuming and processing messages:

```
1 def consume_messages(uri):
2     r = requests.get(base_uri + "/records", headers=GET_HEADERS, timeout=20)
3
4     if r.status_code != 200:
5         print ("Status Code: " + str(r.status_code))
6         print (r.text)
7         sys.exit("Error thrown while getting message")
8
9     for message in r.json():
10        if message["key"] is not None:
11            print ("Message Key:" + message["key"])
12            print ("Message Value:" + message["value"])
```

In a more realistic example you would loop indefinitely, since a topic is an open ended stream of data.

ybhandare@greendotcorp.com

## A Python Consumer Using the REST Proxy (5)

Deleting the consumer:

```
1 def delete_consumer(uri):
2     r = requests.delete(uri, headers=GET_HEADERS)
3
4     if r.status_code != 204:
5         print ("Status Code: " + str(r.status_code))
6         print (r.text)
```

To verify that the consumer instances are removed properly, explicitly remove them using the **DELETE** call before exiting your code.

ybhandare@greendotcorp.com

### Questions:



- What is the **default language** to write Kafka clients?
- What **other languages** does Confluent support?
- How can **non-supported languages** programmatically access Kafka?

- The Kafka API provides Java clients for Producers and Consumers
- Client libraries for other languages are available
  - Confluent provides and supports client libraries for C/C++, Python, Go, and .NET
- The Confluent REST Proxy allows other languages to access Kafka without the need for native client libraries

ybhandare@greendotcorp.com

## Hands-On Lab

---

- In this Hands-On Exercise, you will write a basic Kafka Consumer in Java and optionally in .NET or Python.
- Please refer to the following labs in the handbook:
  - **Creating a Kafka Consumer in Java**
  - **OPTIONAL: Creating a Kafka Consumer in C#/NET**



ybhandare@greendotcorp.com



### More Advanced Kafka Development

ybhandare@greendotcorp.com

# Agenda

---



1. Introduction
2. Fundamentals of Apache Kafka
3. Kafka's Architecture
4. Developing With Kafka
5. More Advanced Kafka Development ... ←
6. Schema Management In Kafka
7. Data Pipelines with Kafka Connect
8. Stream Processing with Kafka Streams
9. Stream Processing with KSQL
10. Event Driven Architecture
11. Confluent Cloud
12. Conclusion
13. Appendix: Basic Kafka Administration
14. Appendix: Apache Kafka driving ML & Data Analytics

ybhandare@greendotcorp.com

## Learning Objectives

---



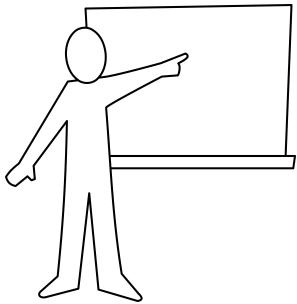
After this module you will be able to:

- Control message durability
- Enable exactly once semantics (EOS)
- Specify the offset to read from
- Manually commit reads from the Consumer
- Create a custom Partitioner

ybhandare@greendotcorp.com

# Module Map

---



- Message Size & Durability ... ←
- Exactly Once Semantics
- Specifying Offsets
- Consumer 'Liveness' and Rebalancing
- Manually Committing Offsets
- Partitioning Data
-  Hands-on Lab

ybhandare@greendotcorp.com

# Message Size Limit



Don't change max message size!

## Limit Message Batch size

- **Globally** - `message.max.bytes` (Default: 1MB)
- **Per topic** - `max.message.bytes` (Default: 1MB)

Kafka is not optimized for super-large messages. When a large message is received, a byte buffer is allocated to receive the entire message. The larger the message, the more likely that performance issues might occur (e.g., fragmentation in the heap). Rather than increasing the maximum message size, consider other options such as:

- Break up the message and send as a transaction with the EOS framework
- Compress the message
- Store the object and just pass a reference to the object
- If you must change the maximum size for a batch of messages that the Broker can receive **from a Producer**,
- **globally**, adjust `message.max.bytes` on broker
- **per topic**, adjust `max.message.bytes`



If the first non-empty Partition of the fetch is larger than the limit, the message will still be returned to ensure that the topic and clients can make progress.

## Message Durability - Replication Factor

- Topics can be **replicated** for durability
- **Default** replication factor is **1**
- Specify when **created**, or **modified**

Create:

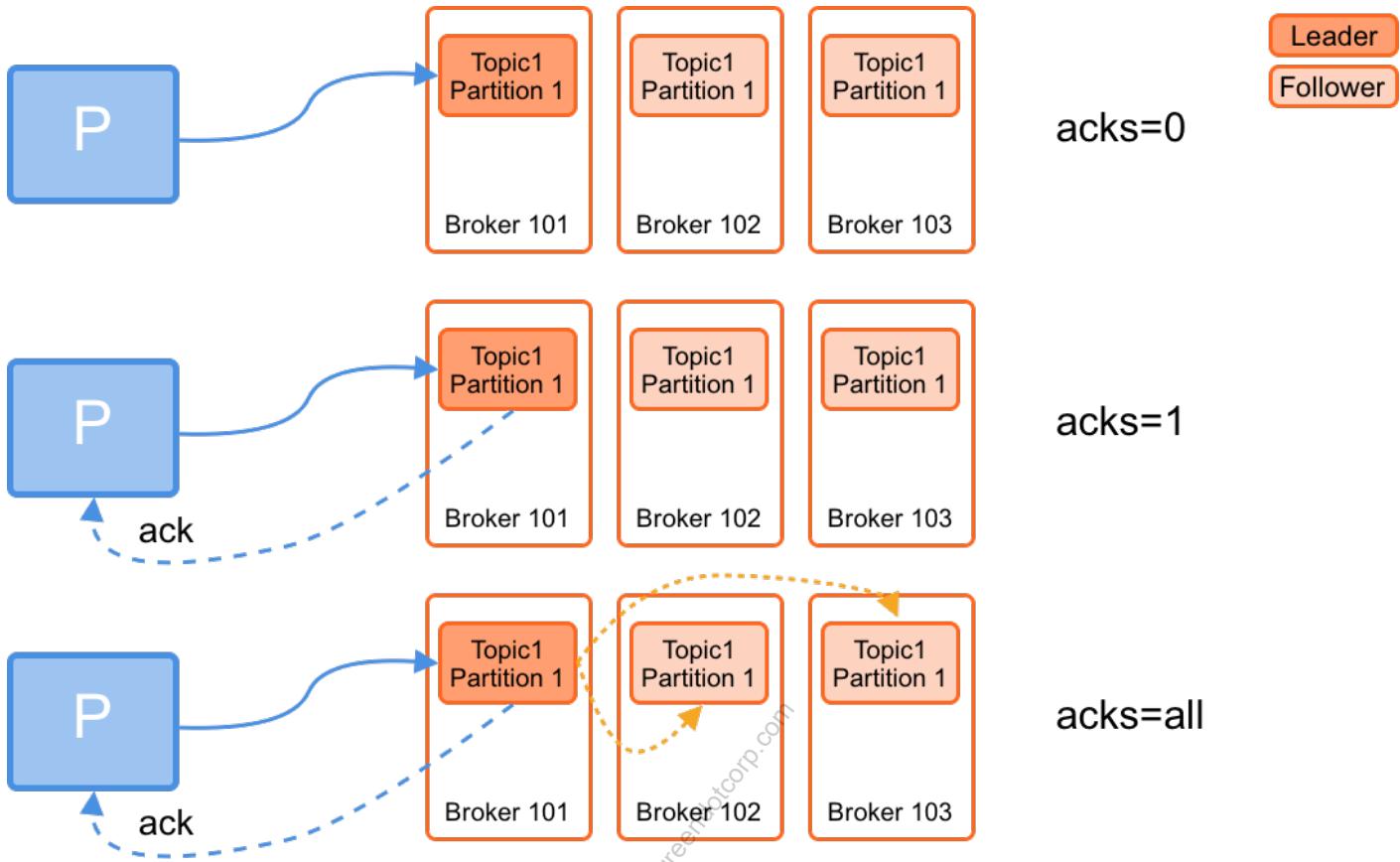
```
kafka-topics --bootstrap-server kafka:9092 \  
  --create \  
  --topic vehicle-positions \  
  --partitions 12 \  
  --replication-factor 3
```

Modify:

```
kafka-reassign-partitions \  
  --bootstrap-server kafka:9092 \  
  ...
```

Increasing the replication factor is a function of the `kafka-reassign-partitions` command. This command is discussed in detail in the Operations course appendix.

## Message Durability - Delivery Acknowledgment



How durable a message is, is determined by the producer. The producer uses the `acks` setting to determine one of three cases shown on the slide.

- **acks=0:** The producer sends messages and **does not** wait for acknowledgment from the broker(s)
- **acks=1:** The producer waits for an ACK from the broker which is leader for a given partition. The leader sends an ACK once it has stored the message(s) in the commit log
- **acks=all:** The producer also waits for an ACK from the broker which is leader for a given partition. But this time the leader only sends an ACK if the replicas are up to date and have replicated the respective messages into their own commit log.



the setting `min.insync.replica` determines how many followers need to have ACK'ed the messages. Also remember that the leader is included in the number of `min.insync.replica`. Thus `min.insync.replica=2` means the leader and one follower

For more details see the next slide...

## Message Durability - Delivery Acknowledgment

Producer defines durability requirements!

<code>acks=0</code>	Producer will not wait for any acknowledgment. The message is placed in the Producer's buffer, and immediately considered sent
<code>acks=1</code>	The Producer will wait until the leader acknowledges receipt of the message
<code>acks=all</code>	The leader will wait for acknowledgment from all in-sync replicas before reporting the message as delivered to the Producer



Use `min.insync.replicas` with `acks=all`

- `acks` config parameter determines behavior of Producer when sending messages
- Use this to configure the durability of messages being sent
- `min.insync.replicas` with `acks=all` defines min. number of replicas in ISR needed to satisfy a produce request

## Message Durability - Important Timeouts

### request.timeout.ms

- Default: 30s
- max time waiting for the response
- after timeout → **resend**
- throw error if **retries** exhausted



**retries** defaults to `MAX_INT` and is only relevant if `acks!=0`

### max.block.ms

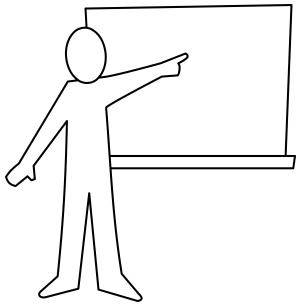
- Default: 60s
- how long `send()` maximally blocks
- after timeout → `TimeoutException`

### delivery.timeout.ms

- Upper limit for delivery, includes
  - batching in accumulator
  - retries
  - in-flight

- `request.timeout.ms` is the amount of time the producer will wait for the ack from the Brokers.
- `max.block.ms` is how long the `send` call will wait for a message to be placed in the buffer pool. This situation happens if the buffer is full.
- `delivery.timeout.ms`: The window of enforcement includes batching in the accumulator, retries, and the inflight segments of the batch. With this config, the user has a guaranteed upper bound on when a record will either get sent, fail or expire from the point when `send` returns. In other words we no longer overload `request.timeout.ms` to act as a weak proxy for accumulator timeout and instead introduce an explicit timeout that users can rely on without exposing any internals of the producer such as the accumulator.

## Module Map



- Message Size & Durability
- Exactly Once Semantics ... ←
- Specifying Offsets
- Consumer 'Liveness' and Rebalancing
- Manually Committing Offsets
- Partitioning Data
-  Hands-on Lab

ybhandare@greendotcorp.com

*Writing real-time, mission-critical streaming applications, requiring **exactly once** processing guarantees.*

## Use Cases:

- Financial transactions
- Tracking ad views
- Kafka Streams & KSQL apps

## Supported Clients

- Producer and Consumer (Java only)
- Kafka Streams API
- Confluent KSQL

- Write real-time, mission-critical streaming applications that require guarantees that data is processed “exactly once”
  - Complements the “at most once” or “at least once” semantics that exist today
- Exactly Once Semantics (EOS) bring strong transactional guarantees to Kafka
  - Prevents duplicate messages from being processed by client applications
    - Even in the event of client retries and Broker failures
- Use cases:
  - tracking ad views,
  - processing financial transactions,
  - stream processing with e.g. aggregates only really makes sense with EOS



For EOS to work, the producer must be set with `acks=all`. Through the use of header fields and other improvements, EOS can take the "at least once" guarantee of this `acks` setting and dedupe messages as they pass through Kafka.



At this time EOS is **not** supported in `librdkafka` based clients. Only idempotent producers are supported.



This section is only a high-level overview of EOS. For a more complete discussion, refer to the online talk and the original KIP: <https://cnfl.io/EOS-KIP-98>

ybhandare@greendotcorp.com

# EOS - In 3 Steps

1

Use **idempotent** producer(s)

`enable.idempotence=true`

2

Write **atomically** across multiple partitions

use **Transaction API**

```
1 producer.initTransactions();
2 try {
3     producer.beginTransaction();
4     producer.send(record1);
5     producer.send(record2);
6     producer.sendOffsetsToTransaction(...);
7     producer.commitTransaction();
8 } catch(ProducerFencedException e) {
9     producer.close();
10 } catch(KafkaException e) {
11     producer.abortTransaction();
12 }
```

3

**Fence off** open transactions

Use **transactional.id**

- Configure the Producer send operation to be **idempotent**
  - An idempotent operation is one which can be performed many times without causing a different effect than only being performed once
    - Removes the possibility of duplicate messages being delivered to a particular Topic Partition due to Producer or Broker errors
  - Set `enable.idempotence` to `true` (Default: `false`)
- A Producer writes atomically across multiple Partitions using "transactional" messages
  - Either all or no messages in a batch are visible to Consumers
  - Use the new transactions API
- Allow reliability semantics to span multiple producer sessions
  - Set `transactional.id` to guarantee that transactions using the same Transactional ID have completed prior to starting new transactions

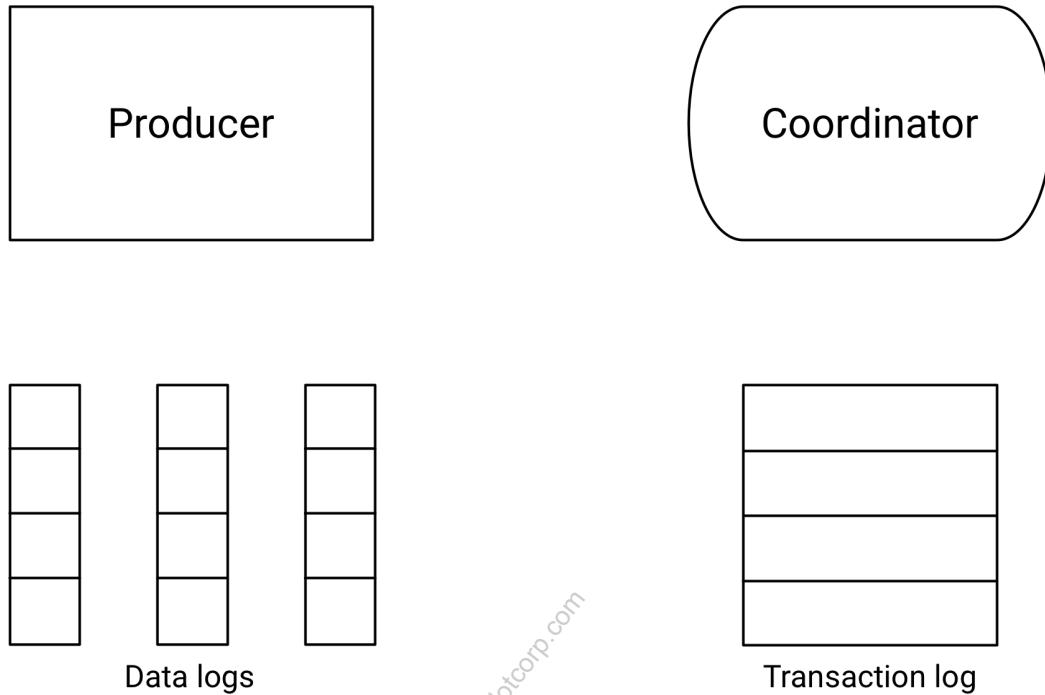
Transactional guarantees enable applications to batch messages into a single atomic unit. In particular, a "batch" of messages in a transaction can be written to multiple partitions, and are "atomic" in the sense that writes will fail or succeed as a single unit.

Additionally, stateful applications will also be able to ensure continuity across multiple sessions of the application. In other words, Kafka can guarantee idempotent production and transaction recovery across application bounces.

To achieve this, Kafka requires that the application provide a unique id which is stable across all sessions of the application – the `transactional.id`. While there may be a 1-1 mapping between an Transactional ID and the internal Producer ID, the main difference is the the Transactional ID is provided by the Producer configurations, and is what enables idempotent guarantees across producers sessions.

The method `sendOffsetsToTransaction` sends the consumed offsets to the offsets topic as part of the transaction. With this guarantees, we know that the offsets and the output records will be committed as an atomic unit. This applies in **produce-consume-produce** scenarios.

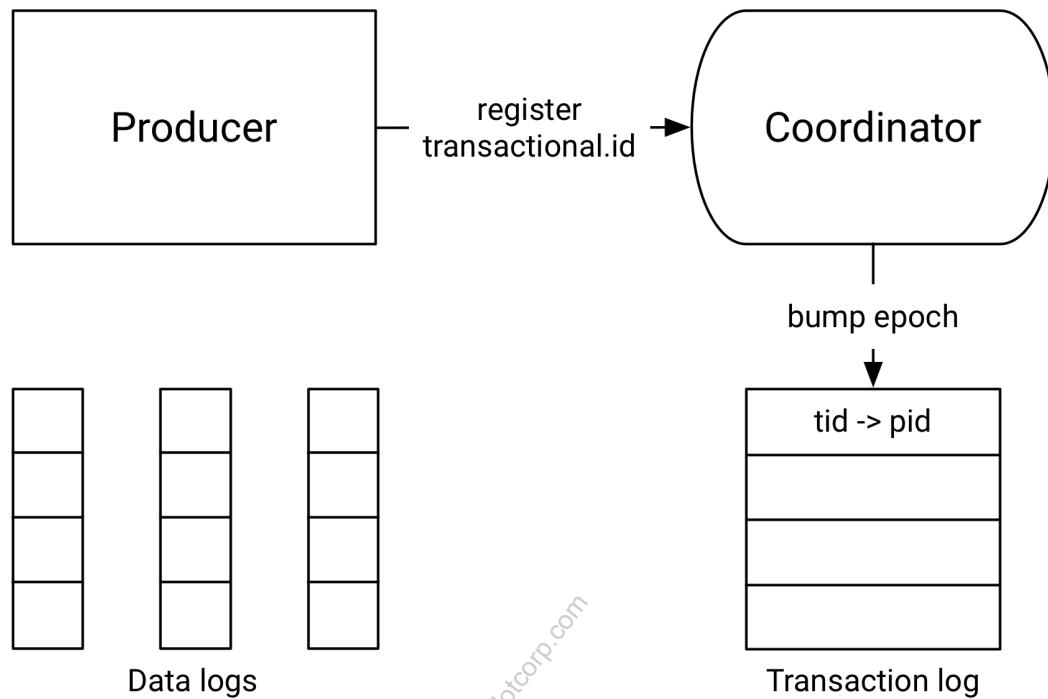
ybhandare@greenlotcc.com



Producers can send data atomically across multiple Partitions to guarantee the receipt of sets of messages. This is a transaction.

A Transaction Coordinator is a module that is available on any Broker. The Transaction Coordinator is responsible for managing the lifecycle of a transaction in the "Transaction Log" – an internal Kafka Topic partitioned by `transactional.id`. The Broker that acts as the Transaction Coordinator is not necessarily a Broker that the Producer is sending messages to. For a given Producer (identified by `transactional.id`), the Transaction Coordinator is the leader of the Partition of the Transaction Log where `transactional.id` resides. Because the Transaction Log is a Kafka Topic, it has durability guarantees.

producer.initTransactions()

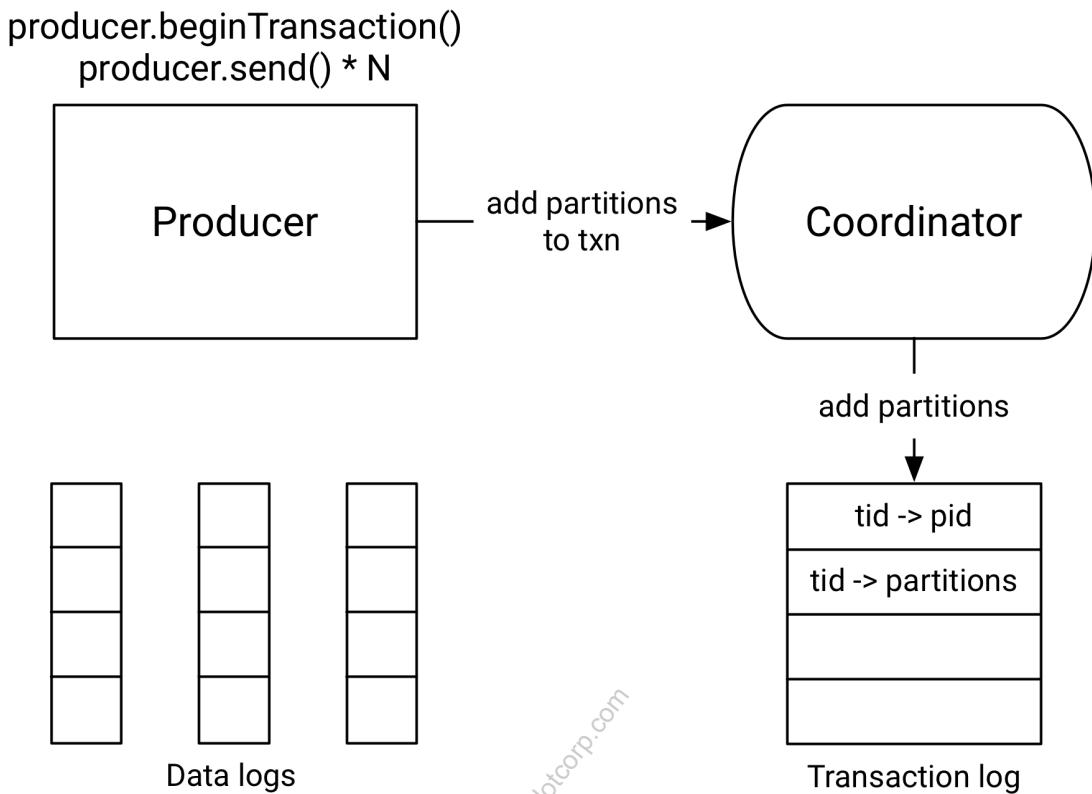


The Producer initiates a transaction. The Producer registers itself to the Transaction Coordinator with `transactional.id`. The Transaction Coordinator records a mapping `{ Transactional ID : Producer ID }`. If the Transactional ID already exists, it means two Producers are fighting for the same identity. The Transaction Coordinator will "fence off" Producers with old PIDs (so-called "zombie Producers") and only allow the Producer with the newest PID to participate in transactions identified by `transactional.id`.



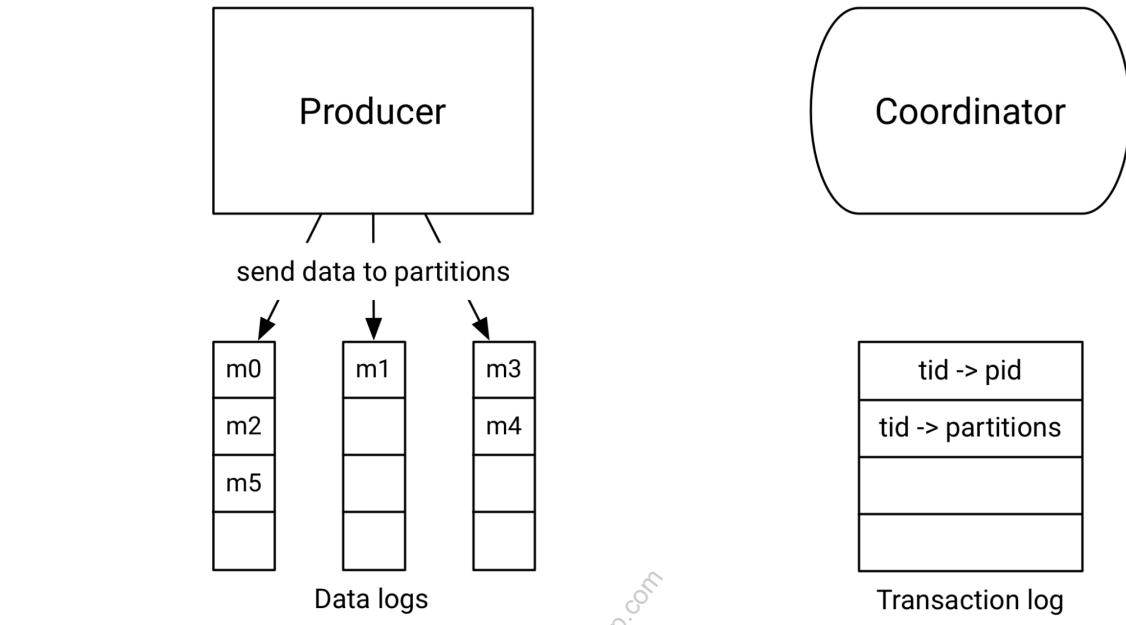
The Transaction Coordinator also increments an **epoch** associated with the `transactional.id`. The epoch is an internal piece of metadata stored for every `transactional.id`.

Once the epoch is bumped, any producers with same `transactional.id` and an older epoch are considered zombies and are fenced off, ie. future transactional writes from those producers are rejected.



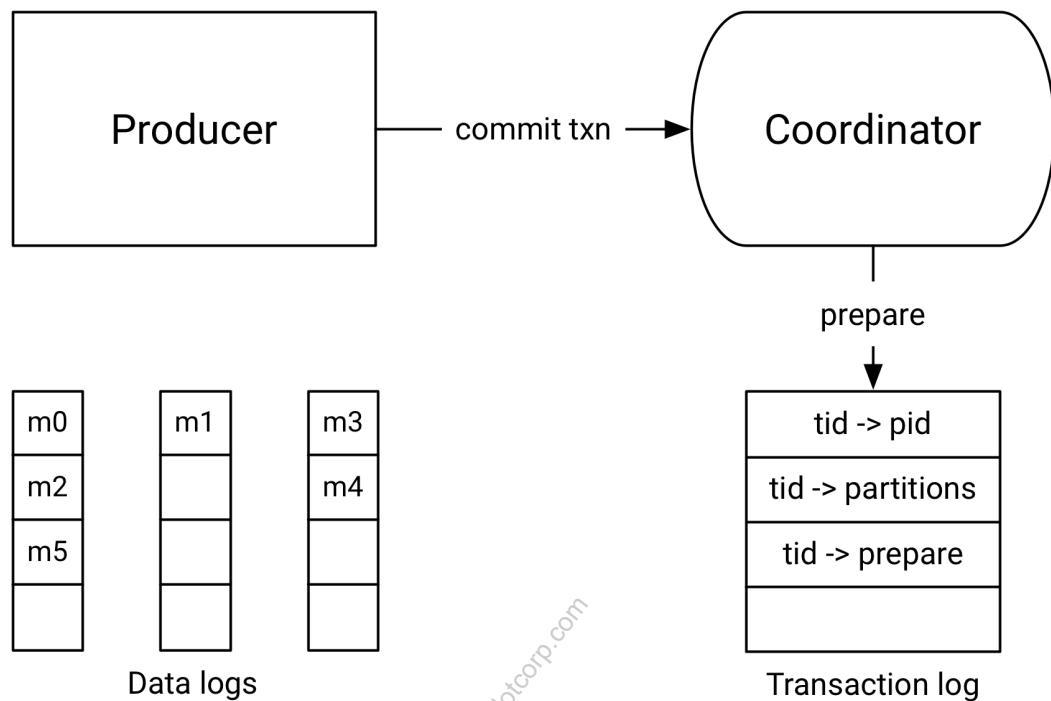
The Producer sends information to the Transaction Coordinator about the Partitions it will write to.

**i** To be precise - "Register Partitions" would happen after producer starts sending data to various partitions (see next slide)  
 The producer sends this request to the transaction coordinator the first time a new TopicPartition is written to as part of a transaction. The addition of this TopicPartition to the transaction is logged by the coordinator in step 4.1a. We need this information so that we can write the commit or abort markers to each TopicPartition (see section 5.2 for details). If this is the first partition added to the transaction, the coordinator will also start the transaction timer.



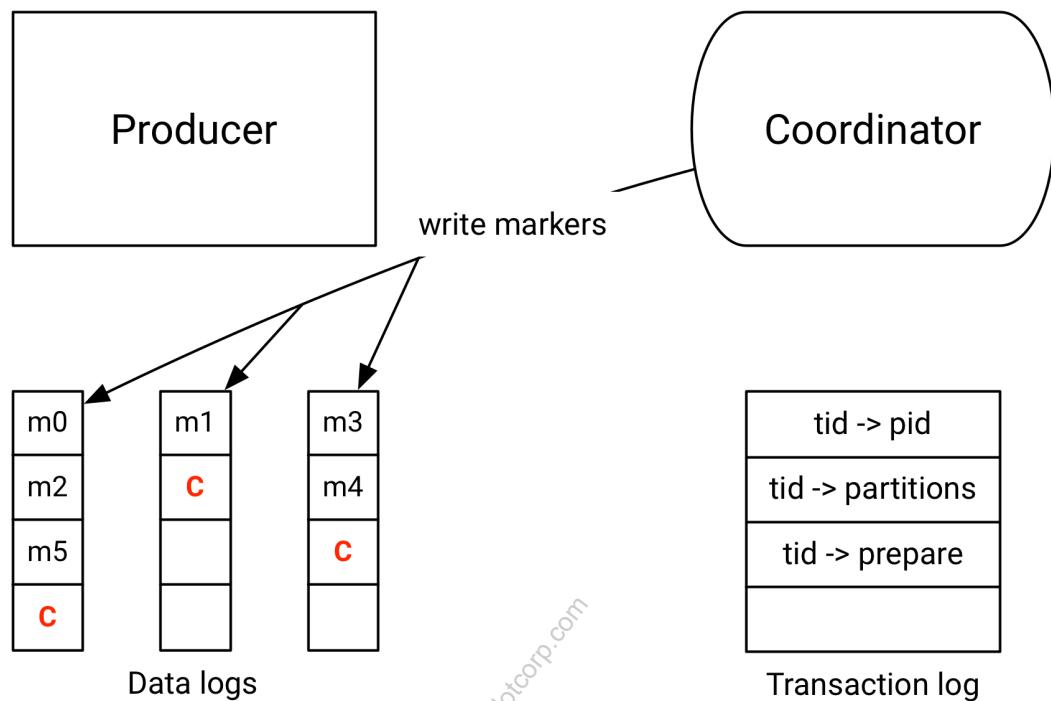
The Producer sends transactional messages which may be on multiple Partitions on multiple Brokers.

producer.commitTransaction()



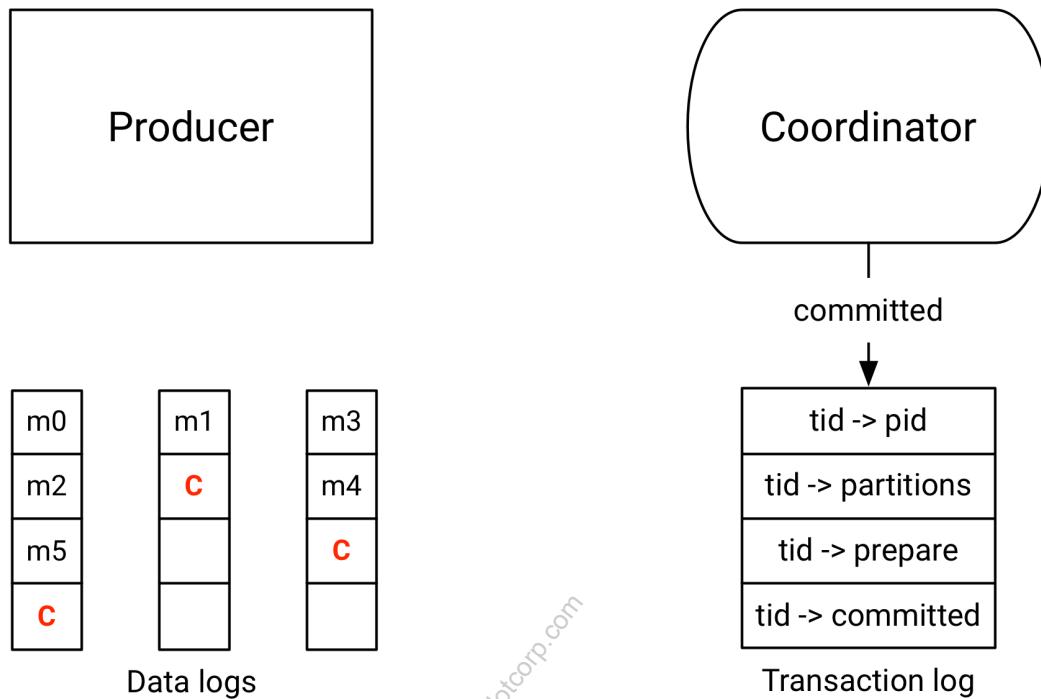
Producer commits the transaction. The transaction coordinator marks the transaction as in status of "preparing".

producer.commitTransaction()



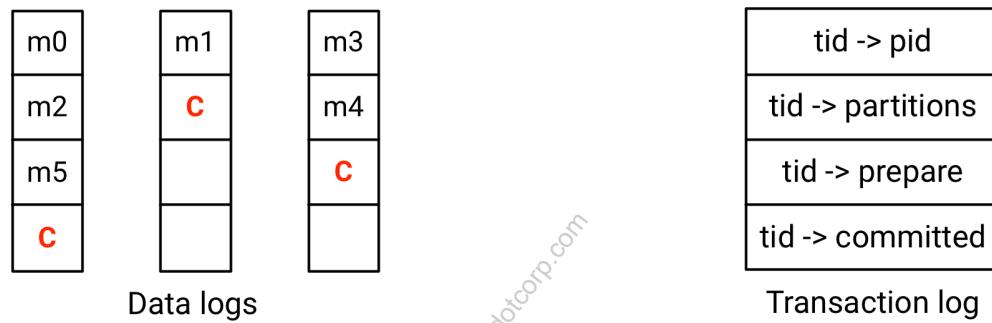
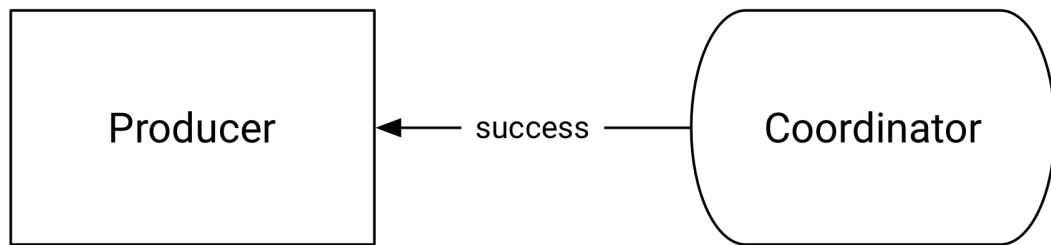
The Transaction Coordinator writes commit markers to the Partitions the Producer writes to. Commit markers are special messages which log the producer id and the result of the transaction (committed or aborted). These messages are internal only and are not exposed by standard consumer operations.

producer.commitTransaction()



The Transaction Coordinator marks the transaction as committed.

producer.commitTransaction()



As a final step the transaction coordinator sends an acknowledgment to the producer.

```
1 producer.initTransactions();
2 try {
3     producer.beginTransaction();
4     producer.send(record1);
5     producer.send(record2);
6     producer.sendOffsetsToTransaction(...);
7     producer.commitTransaction();
8 } catch(ProducerFencedException e) {
9     producer.close();
10 } catch(KafkaException e) {
11     producer.abortTransaction();
12 }
```

What are these methods doing?

### initTransactions

Needs to be called before any other methods when the `transactional.id` is set in the configuration. This method does the following:

1. Ensures any transactions initiated by previous instances of the producer with the same `transactional.id` are completed. If the previous instance had failed with a transaction in progress, it will be aborted. If the last transaction had begun completion, but not yet finished, this method awaits its completion.
2. Gets the internal producer id and epoch, used in all future transactional messages issued by the producer. Note that this method will raise `TimeoutException` if the transactional state cannot be initialized before expiration of `max.block.ms`. Additionally, it will raise `InterruptException` if interrupted. It is safe to retry in either case, but once the transactional state has been successfully initialized, this method should no longer be used.

<code>beginTransaction</code>	<p>Must be called to signal the start of a new transaction. The producer records local state indicating that the transaction has begun, but the transaction won't begin from the coordinator's perspective until the first record is sent.</p> <p>Throws a <code>ProducerFencedException</code> if another producer with the same <code>transactional.id</code> is active</p> <p>Note that prior to the first invocation of this method, you must invoke <code>initTransactions()</code> exactly one time.</p>
<code>commitTransaction</code>	<p>Commits the ongoing transaction. This method will flush any unsent records before actually committing the transaction. Further, if any of the <code>send(ProducerRecord)</code> calls which were part of the transaction hit irrecoverable errors, this method will throw the last received exception immediately and the transaction will not be committed. So all <code>send(ProducerRecord)</code> calls in a transaction must succeed in order for this method to succeed.</p>
<code>sendOffsetsToTransaction</code>	<p>Although not on the slide, this method is important in more complex scenarios (e.g. stream processing).</p> <p>Sends a list of specified offsets to the consumer group coordinator, and also marks those offsets as part of the current transaction. These offsets will be considered committed only if the transaction is committed successfully. The committed offset should be the next message your application will consume, i.e.</p> <p><code>lastProcessedMessageOffset + 1</code>.</p> <p>This method should be used when you need to batch consumed and produced messages together, typically in a consume-transform-produce pattern. Thus, the specified <code>consumerGroupId</code> should be the same as config parameter <code>group.id</code> of the used consumer. Note, that the consumer should have <code>enable.auto.commit=false</code> and should also not commit offsets manually (via sync or async commits).</p>



When using transactions, it's not necessary to register a **callback** to be notified of failed acks - there will be an exception thrown in that case.

ybhandare@greendotcorp.com

## Transactional

- `isolation.level=read_committed`
- reads only committed transactional messages and all non-transactional messages

## Non-Transactional (default)

- `isolation.level=read_uncommitted`
- reads all transactional messages and all non-transactional messages

If using EOS-enabled producers, consumers running older versions (i.e., 0.10.x or older) should be updated as soon as possible. When a consumer sends a request for an older version, the broker assumes the READ\_UNCOMMITTED isolation level and converts the message set to the appropriate format before sending back the response, essentially disabling zero-copy for those fetch requests. This conversion can be costly when compression is enabled, so it is important to update the client as soon as possible.

If you design an application as a consume-process-produce pipeline (e.g. read from one topic, write to another), you can combine the consumer offset commits with the producer writing to the output topic in a single atomic transaction, which means each message is effectively processed only once. However, if failure occurs, it's possible messages could be processed multiple times. But in that case, the output of the processing never got committed the first time around, so the semantics are exactly-once, even if it's not truly exactly once delivery and processing under the covers.

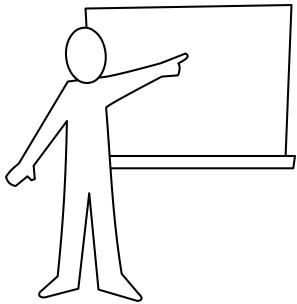
One exception is when rebalancing happens in a consumer group - that could result in moving processing of messages from a partition to a different process with a different transactional.id, and EOS is not guaranteed in that case.

Kafka Streams has some special functionality to make the rebalancing case work with EOS, but it's not built-in at this level. EOS was really designed with Kafka Streams in mind, where consume-process-produce with Kafka input and output is the standard execution model.



## Module Map

---



- Message Size & Durability
- Exactly Once Semantics
- Specifying Offsets ... ↙
- Consumer 'Liveness' and Rebalancing
- Manually Committing Offsets
- Partitioning Data
-  Hands-on Lab

ybhandare@greendotcorp.com

## Determining the Offset When a Consumer Starts

### 3 Scenarios

- Consumer Group starts first time
- Consumer offset < smallest offset
- Consumer offset > last offset

### Values for `auto.offset.reset`

<code>earliest</code>	Reset offset to earliest available
<code>latest</code>	Reset offset to latest available
<code>none</code>	Throw exception if no prev. offset found for Consumer Group

- The Consumer property `auto.offset.reset` determines what to do if there is no valid offset in Kafka for the Consumer's Consumer Group
  - When a particular Consumer Group starts the first time
  - If the Consumer offset is less than the smallest offset
  - If the Consumer offset is greater than the last offset
- The value can be one of:
  - `earliest`: Automatically reset the offset to the earliest available
  - `latest`: Automatically reset to the latest offset available
  - `none`: Throw an exception if no previous offset can be found for the ConsumerGroup
- The default is `latest`



this setting does not affect offsets that have been deleted or compacted due to log cleanup. In those cases, if the offset specified in the offsets topic does not exist, the broker will advance to the next highest offset and proceed from there.

# Accessing & Changing Offset Within Consumer (1)

## Get Offsets

<code>position(TopicPartition)</code>	Offset of next record that will be fetched
<code>offsetsForTimes(Map&lt;TopicPartition, Long&gt; timestampsToSearch)</code>	Offsets for given Partitions by timestamp

## Change Offsets

<code>seekToBeginning(Collection&lt;TopicPartition&gt;)</code>	Seek to first offset of each specified Partition
<code>seekToEnd(Collection&lt;TopicPartition&gt;)</code>	Seek to last offset of each specified Partition
<code>seek(TopicPartition, offset)</code>	Seek to specific offset in specified Partition

- The `KafkaConsumer` API provides ways to view offsets and dynamically change the offset from which the Consumer will read
- View offsets
  - `position(TopicPartition)` provides the offset of the next record that will be fetched
  - `offsetsForTimes(Map<TopicPartition, Long> timestampsToSearch)` looks up the offsets for the given Partitions by timestamp
- Change offsets
  - `seekToBeginning(Collection<TopicPartition>)` seeks to the first offset of each of the specified Partitions
  - `seekToEnd(Collection<TopicPartition>)` seeks to the last offset of each of the specified Partitions
  - `seek(TopicPartition, offset)` seeks to a specific offset in the specified Partition

## Accessing & Changing Offset Within Consumer (1)

Kafka 0.10.1.0 introduced `offsetsForTimes`, a timestamp search method, to the Consumer. The returned offset for each Partition is the earliest offset whose timestamp is greater than or equal to the given timestamp in the corresponding Partition. This is useful to rewind offsets if applications need to re-consume messages for a certain period of time, or in a multi-datacenter environment because the offset between two different Kafka clusters are independent and users cannot use the offsets from the failed datacenter to consume from the DR datacenter. In this case, searching by timestamp will help because the messages should have same timestamp if users are using `CreateTime`. Even if users are using `LogAppendTime`, a more granular search based on timestamp can still help reduce the amount of messages to be re-consumed.

Messages written to a topic using replication tools (e.g., MirrorMaker, Confluent Replicator) will retain their original timestamps when copied to the new location.

As of Kafka 0.11.0, the `kafka-consumer-groups` command has the ability to reset the entire Consumer Group to the offsets corresponding to specific timestamps within the partitions.

```
kafka-consumer-groups \
  --bootstrap-server broker101:9092 \
  --group my-group \
  --topic my_topic \
  --reset-offsets \
  --execute \
  --to-datetime 2017-08-01T17:14:23.933`
```

## Accessing & Changing Offset Within Consumer (2)

Seek to beginning of all partitions for topic `my_topic`:

```
1 ConsumerRebalanceListener listener = new
2     ConsumerRebalanceListener() {
3         @Override
4             public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
5                 // nothing to do...
6             }
7
8         @Override
9             public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
10                 consumer.seekToBeginning(partitions);
11             }
12     };
13
14 consumer.subscribe(Arrays.asList("my_topic"), listener); ①
15 consumer.poll(Duration.ofMillis(0)); ②
```

- ① Subscribe to topics & register `ConsumerRebalanceListener`
- ② `poll(Duration.ofMillis(0))` retrieves metadata from broker

- **Example:** to seek to the beginning of all Partitions that are being read by a Consumer for a particular Topic, you might do something like:

The `poll` method has several functions:

- Fetch messages from assigned partitions
- Trigger partition assignment (if necessary)
- Commit offsets if auto offset commit is enabled

In this example, the data returned by `poll` (i.e., messages fetched from the partitions) is not important since we are resetting the place that it is reading from so the data is not assigned to an object as we saw in the previous chapter. Calling `poll` immediately after the `subscribe` triggers the partition assignment so that the methods in line 3 will work.

## Changing the Offset Within the Consumer (3)

Reset offset to **particular timestamp**:

```
1 for (TopicPartition partition : partitions) {  
2     timestampsToSearch.put(partition, MY_TIMESTAMP); ①  
3 }  
4  
5 Map<TopicPartition, OffsetAndTimestamp> result = consumer.offsetsForTimes(timestampsToSearch); ②  
6  
7 for (Map.Entry<TopicPartition, OffsetAndTimestamp> entry : result.entrySet()) {  
8     consumer.seek(entry.getKey(), entry.getValue().offset()); ③  
9 }
```

- ① Add each Partition and timestamp to HashMap
- ② Get offset for each Partition
- ③ Seek to specified offset for each Partition

**Question:** what is a use case where this would be useful?

**Answer:** a Multi Datacenter DR failover. If the active datacenter fails and a consumer spins up in the passive datacenter, it cannot use simple offset numbers to determine where to read from since offset numbers are not consistent when topics are replicated using MirrorMaker or Confluent Replicator. The consumer can use this code to determine where to restart consumption based on timestamp.

## Resetting Consumer Group Offsets via the CLI

Sample commands:

```
kafka-consumer-groups \
  --bootstrap-server broker101:9092 \
  --group my-group \
  --topic my_topic \
  --reset-offsets \
  --execute \
  --to-earliest # --to-latest
```

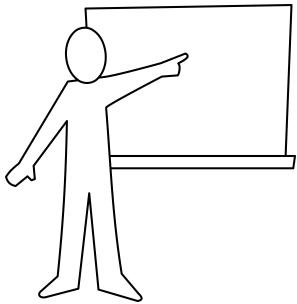
```
kafka-consumer-groups \
  --bootstrap-server broker101:9092 \
  --group my-group \
  --topic my_topic \
  --reset-offsets \
  --execute \
  --to-datetime 2018-08-01T17:14:23.933
```

```
kafka-consumer-groups \
  --bootstrap-server broker101:9092 \
  --group my-group \
  --topic my_topic \
  --reset-offsets \
  --execute \
  --to-offset 9876
```

- The command `kafka-consumer-groups` can be used to set the starting offset in a topic for a consumer group
- Uses the `--reset-offsets` option with a modifier
- Can reset to earliest or latest offset, a specific offset, or a specific timestamp (ISO 8601)
- Can reset to a specific offset (long)

# Module Map

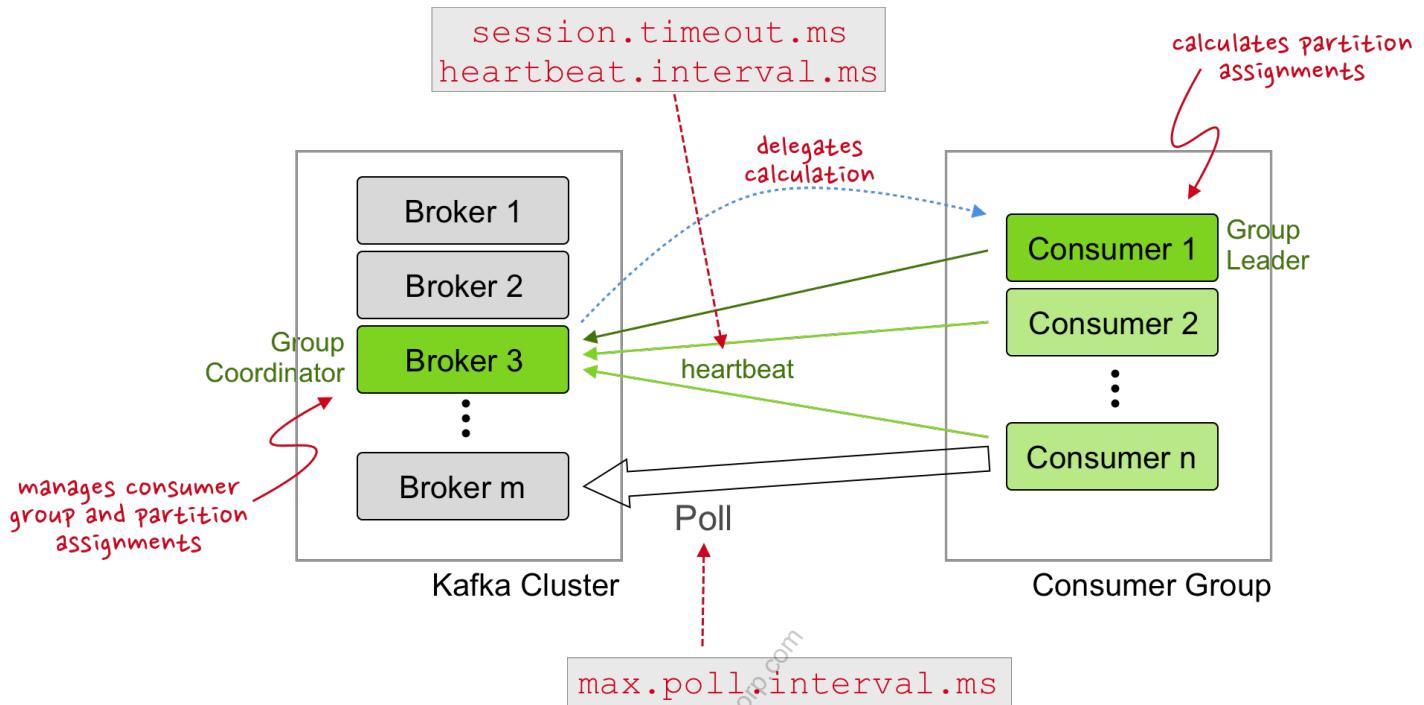
---



- Message Size & Durability
- Exactly Once Semantics
- Specifying Offsets
- Consumer 'Liveness' and Rebalancing ... 
- Manually Committing Offsets
- Partitioning Data
-  Hands-on Lab

ybhandare@greendotcorp.com

## Consumer Liveness



Here is an image that depicts how the liveness of a consumer in a consumer group is determined.

- Each consumer sends a periodic liveness signal to the group coordinator (broker) on a dedicated thread
- If no liveness signal is received for more than `session.timeout.ms` milliseconds (default: 10s) then the consumer is considered to be dead and the group coordinator triggers a partition reassignment
- The calculation of the partition assignment to the remaining consumer group member is delegated to the **group leader**
- The group coordinator then communicates the new partition assignments to each consumer of the group
- If the liveness thread of a consumer is still working but its main thread, where the polling for data is happening, "hangs" then there is another timeout time called `max.poll.interval.ms` (default: 5min) after exceeded the corresponding consumer is considered dead and the group coordinator triggers a partition reassignment. The parameter `heartbeat.interval.ms` (default: 3s) in turn defines how long the interval between two successive heartbeat signals are.

## Partition Assignments for Consumers

- All data of partition consumed by **same consumer**
- Except on rebalance!
- Rebalance moves partition ownership
- Goal → **spread load equally**



Consumption **pauses** during rebalance

- We have said that all the data from a particular Partition will go to the same Consumer
- This is true as *long as* the number of Consumers in a Consumer Group does not change
- If the number of Consumers changes, a Partition rebalance occurs
  - Partition ownership is moved around between the Consumers to spread the load evenly
    - No guarantees that a Consumer will get the same or different Partition
- Consumers cannot consume messages during the rebalance, so this results in a short pause in message consumption

ybhandare@greendotcorp.com

# Consumer Rebalancing

Rebalancing on:

- Consumer **leaves** group
- Consumer **joins** group
- Consumer **changes topic subscription**
- Number of partitions changes



Consumption is **stopped** during rebalance

- **Question:** could adding a Consumer to a Consumer Group cause Partition assignment to change?

- Consumer rebalances are initiated when
  - A Consumer leaves the Consumer group (either by failing to send a timely heartbeat or by explicitly requesting to leave)
  - A new Consumer joins the Consumer Group
  - A Consumer changes its Topic subscription
  - The Consumer Group notices a change to the Topic metadata for any subscribed Topic (e.g. an increase in the number of Partitions)
- Rebalancing does not occur if
  - A Consumer calls **pause**
- During rebalance, consumption is paused

**Answer:** Yes!

Once a rebalance has begun, the coordinator starts a timer which is set to expire after the group's session timeout. Each member in the previous generation detects that it needs to rejoin with its periodic heartbeats sent to the coordinator. The coordinator uses the **REBALANCE\_IN\_PROGRESS** error code in the heartbeat response so the Consumer knows to rejoin. "

# The Case For and Against Rebalancing

- Usually Rebalancing is **desired**
- But → Rebalancing is like **fresh start**
- Consumers manage rebalancing with:

```
public interface ConsumerRebalanceListener {  
    void onPartitionsAssigned(Collection<TopicPartition> partitions)  
    void onPartitionsRevoked(Collection<TopicPartition> partitions)  
}
```

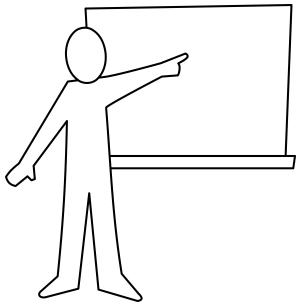
- Typically, Partition rebalancing is a good thing
  - Allows you to add more Consumers to a Consumer Group dynamically, without having to restart all the other Consumers in the group
  - Automatically handles situations where a Consumer in the Consumer Group fails
- However a rebalance is like a fresh restart
  - Consumers may or may not get a different set of Partitions
    - If your Consumer is relying on getting all data from a particular Partition, this could be a problem
  - A previously-paused Partition is no longer paused
- Managing rebalances where Partition assignment matters
  - Option 1: only have a single Consumer for the entire topic
  - Option 2: provide a `ConsumerRebalanceListener` when calling `subscribe()`
    - Implement `onPartitionsRevoked` and `onPartitionsAssigned` methods
  - Soft option 3: Sticky partition assignment strategy (KIP-54) does not guarantee that assignments do not change. Assignments can and do change, because the goals are, in priority order:
    - Topic partitions are still distributed as evenly as possible
    - Topic partitions stay with their previously assigned consumers as much as possible



Transactions will not automatically work across rebalances in a consumer group. Kafka Streams has special functionality to support this, but it's not built-in at this level

# Module Map

---



- Message Size & Durability
- Exactly Once Semantics
- Specifying Offsets
- Consumer 'Liveness' and Rebalancing
- Manually Committing Offsets ... 
- Partitioning Data
-  Hands-on Lab

ybhandare@greendotcorp.com

## Default Behavior: Automatically Committed Offsets

---

- Default: `enable.auto.commit=true`
- Commit happens **automatically** ...
- ... during `poll()` call
- Typically **desired** behavior

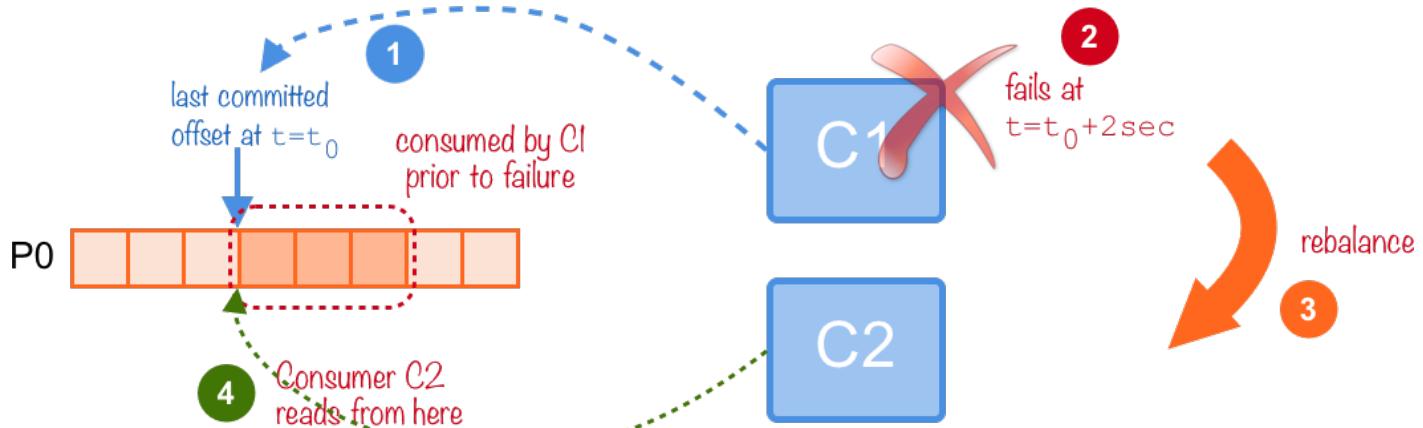


Can sometimes be problematic

- By default, `enable.auto.commit` is set to `true`
  - Consumer offsets are periodically committed in the background
    - This happens during the `poll()` call
- This is typically the desired behavior
- However, there are times when it may be problematic

ybhandare@greendotcorp.com

## Problems With Automatically Committed Offsets



### Avoid reprocessing with **idempotency**

- By default, automatic commits occur every five seconds
- Imagine that two seconds after the most recent commit, a rebalance is triggered
  - After the rebalance, Consumers will start consuming from the latest committed offset position
- In this case, the offset is two seconds old, so all messages that arrived in those two seconds will be processed twice
  - This provides “at least once” delivery



“At least once” delivery works if the application is idempotent: a property such that messages can be applied multiple times without changing the result

## Manually Committing Offsets

- Set: `enable.auto.commit=false`

### Synchronous

- `commitSync()`
- blocks** until success or exception
- At most once:

```
var records=consumer.poll();
consumer.commitSync();
... // process records
```

- At least once:

```
var records=consumer.poll();
... // process records
consumer.commitSync();
```

### Asynchronous

- `commitAsync()`
- Non-blocking**
- Optional: add **callback**



Higher Throughput



Delayed notification about failure

- A Consumer can manually commit offsets to control the committed position
  - Disable automatic commits: set `enable.auto.commit` to `false`
- `commitSync()`
  - Blocks until it succeeds, retrying as long as it does not receive a fatal error
  - For “at most once” delivery, call `commitSync()` immediately after `poll()` and then process the messages
  - Consumer should ensure it has processed all the records returned by `poll()` or it may miss messages
- `commitAsync()`
  - Returns immediately
  - Optionally takes a callback that will be triggered when the Broker responds
  - Has higher throughput since the Consumer can process next message batch before commit returns
    - Trade-off is the Consumer may find out later the commit failed
- `commitAsync` is also useful for slow-performing consumers (e.g. consumers who need to call an external service API), as an alternative to increasing the session timeout

## Manually Committing Offsets



One thing this slide doesn't show explicitly is that `commitSync`/`commitAsync` can be called with parameters to specify topic/partition/offset to commit, allowing more fine-grained offset updates, not just committing the whole batch returned in the last poll call.

- A consumer can combine both `commitSync` and `commitAsync`:
  - `commitAsync()` during normal processing of messages and
  - `commitSync()` just before exiting the Consumer or before a rebalance (in the "finally" clause)



Developers should make sure to commit offsets before a partition rebalance by using the sync methods as part of the `onPartitionsRevoked` method.

ybhandare@greendotcorp.com

## Manually Committing Offsets From the REST Proxy

- It is possible to manually commit offsets from the REST Proxy

```
1 payload = {  
2     "format": "binary",  
3     # Manually/Programmatically commit offset  
4     "auto.commit.enable": "false"  
5 }  
6  
7 headers = {  
8     "Content-Type" : "application/vnd.kafka.v2+json"  
9 }  
10  
11 r = requests.post(baseurl, data=json.dumps(payload), headers=headers)  
12  
13 # Commit the offsets  
14 if shouldCommit() == True:  
15     r = requests.post(base_uri + "/offsets", headers=headers, timeout=20)  
16     if r.status_code != 200:  
17         print ("Status Code: " + str(r.status_code))  
18         print (r.text)  
19         sys.exit("Error thrown while committing")  
20     print "Committed"
```

Note the difference in parameter name between REST and Java. REST: `auto.commit.enable` ; Java `enable.auto.commit`

You can specify offsets to commit, or you can commit all records that have been fetched by the REST Proxy for that consumer instance

## Aside: What Offset is Committed?

---



Offset committed == offset of **next record to read**

- The offset committed (whether automatically or manually) is the offset of the next record to be read
  - Not the offset of the last record which was read

ybhandare@greendotcorp.com

## Storing Offsets Outside of Kafka



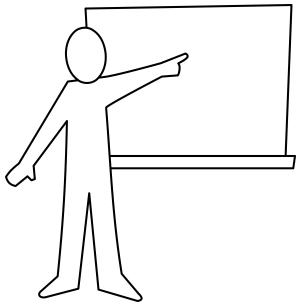
Topic `__consumer_offsets`

- By default, Kafka stores offsets in a special Topic
  - Called `__consumer_offsets`
- In some cases, you may want to store offsets outside of Kafka
  - For example, in a database table
- If you do this, you can read the value and then use `seek()` to move to the correct position when your application launches

Storing offsets outside of Kafka can in some cases allow exactly-once processing to be implemented (e.g., when working with a transactional external resource like an RDBMS).

## Module Map

---



- Message Size & Durability
- Exactly Once Semantics
- Specifying Offsets
- Consumer 'Liveness' and Rebalancing
- Manually Committing Offsets
- Partitioning Data ...   Hands-on Lab

ybhandare@greendotcorp.com

## Kafka's Default Partitioning Scheme

- `partition_index = hash(key) % number_of_partitions`
- All messages with **same key** go to **same partition**
- If `key=NULL`: use **Round Robin**
- Override with **custom partitioner** if needed

- Recall that by default, if a message has a key, Kafka will hash the key and use the result to map the message to a specific Partition
- This means that all messages with the same key will go to the same Partition
- If the key is null and the default Partitioner is used, the record will be sent to a random Partition (using a round-robin algorithm)
- You may wish to override this behavior and provide your own Partitioning scheme
  - For example, if you will have many messages with a particular key, you may want one Partition to hold just those messages



Kafka uses its own hash algorithm, so the hash will not change if the version of Java on the machine is upgraded and a new hashing algorithm is introduced.

## Creating a Custom Partitioner

- Implement interface **Partitioner**

```
public interface Partitioner extends Configurable, ... {  
    void configure(java.util.Map<java.lang.String,?> configs)  
  
    int partition(java.lang.String topic,  
                 java.lang.Object key,  
                 byte[] keyBytes,  
                 java.lang.Object value,  
                 byte[] valueBytes,  
                 Cluster cluster)  
  
    void close();  
}
```



**configure** is inherited from **Configurable**

- Mostly implement method **partition**
- Return number of message **target partition**

- To create a custom Partitioner, you should implement the **Partitioner** interface
  - This interface includes **configure**, **close**, and **partition** methods, although often you will only implement **partition**
  - **partition** is given the Topic, key, serialized key, value, serialized value, and cluster metadata
- It should return the number of the Partition this particular message should be sent to (0-based)



The part we don't show on the slide is that you need to register the custom partitioner with the **partitioner.class** producer config property. Also, **configure** and **close** are optional lifecycle method called once by the producer client library when the producer starts and is closed, and **partition** is called for every message. So whatever the **partition** method does, it should ideally be fast, otherwise it can slow down the entire producer

## Custom Partitioner: Example

- Assume we want to store all messages with a particular key in one Partition, and distribute all other messages across the remaining Partitions

```
1 public class MyPartitioner implements Partitioner {  
2     public void configure(Map<String, ?> configs) {}  
3     public void close() {}  
4  
5     public int partition(String topic, Object key, byte[] keyBytes,  
6                           Object value, byte[] valueBytes, Cluster cluster) {  
7         List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);  
8         int numPartitions = partitions.size();  
9  
10        if ((keyBytes == null) || (!(key instanceof String)))  
11            throw new InvalidRecordException("Record did not have a string Key");  
12  
13        if (((String) key).equals("OurBigKey")) ①  
14            return 0; // This key will always go to Partition 0  
15  
16        // Other records will go to the rest of the Partitions using a hashing function  
17        return (Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1)) + 1;  
18    }  
19 }
```

- ① This is the key we want to store in its own Partition

This sample partitioner verifies that messages have a non-null String-type key. Then it returns 0 for messages with the specified key and a non-0 partition number for any other key, based on the standard hashing function.

This custom partitioner `MyPartitioner` would then be passed into the Producer Java code using the `partitioner.class` config parameter and is by the `KafkaProducer` as part of the `send()` call. The developer code does not call the partitioner directly.

## An Alternative to a Custom Partitioner

- Specify Partition when defining `ProducerRecord`

```
ProducerRecord<String, String> record  
= new ProducerRecord<String, String>("my_topic", 0, key, value);
```

Writes message to Partition `0`

**Question:** Which method is preferable?

- It is also possible to specify the Partition to which a message should be written when creating the `ProducerRecord`
- Answer:** Writing the partitioning logic directly into the producer code is simpler than creating a new class. However, the custom partitioner approach is more portable and makes it easier to share the custom code with other producer applications.

ybhandare@greendotcorp.com



1. How can you influence the reliability of message delivery?
2. Explain in a nutshell what EOS brings to the table
3. In your consumer code, how can you react to rebalancing?
4. How can you influence the partitioning of your data?

5. A team responsible for consuming data from a Kafka cluster is reporting that the cluster is not properly managing offsets. They describe that when they add Consumers to a Consumer Group, all the Consumers see some previously-consumed messages. What is happening?

1. You can configure the reliability of message delivery by specifying different values for the `acks` configuration parameter
2. Exactly Once Semantics (EOS) enables single delivery of Kafka messages
3. Your Consumer can move through the data in the Cluster, reading from the beginning, the end, or any point in between. You may need to take Consumer Rebalancing into account when you write your code
4. It is possible to specify your own Partitioner if Kafka's default is not sufficient for your needs
5. When the Consumer Group membership changes, it triggers a rebalance
  - Existing Partitions may be reassigned to different Consumers
    - Offsets are managed per Partition not per Consumer
  - This works only if the Consumer code is written well and the offsets are committed upon consumption
    - Action:
  - Review Consumer client code for committing offsets during consumption
    - Commit intervals
    - `commitSync()` vs `commitAsync()`
  - Consumers should implement `ConsumerRebalanceListener` code
    - Consumer can commit offsets in reaction to receiving an `onPartitionsRevoked` event

## Hands-On Lab

---

- In this Hands-On Exercise you will create a Consumer which will access data already stored in the cluster. Either the consumer always starts from the very beginning or it starts from an offset stored outside of Kafka
- Please refer to the labs
  - **Accessing Previous Data**
  - **Managing Consumer Offsets in Code - Java**
  - **OPTIONAL: Managing Consumer Offsets in Code - C#/.NET**

in Exercise Book



ybhandare@greendotcorp.com

## Further Reading

---

- KIP-98 on EOS: <https://cnfl.io/KIP-98>

ybhandare@greendotcorp.com



### Schema Management In Kafka

ybhandare@greendotcorp.com

# Agenda

---



1. Introduction
2. Fundamentals of Apache Kafka
3. Kafka's Architecture
4. Developing With Kafka
5. More Advanced Kafka Development
6. Schema Management In Kafka ... ←
7. Data Pipelines with Kafka Connect
8. Stream Processing with Kafka Streams
9. Stream Processing with KSQL
10. Event Driven Architecture
11. Confluent Cloud
12. Conclusion
13. Appendix: Basic Kafka Administration
14. Appendix: Apache Kafka driving ML & Data Analytics

ybhandare@greendotcorp.com

## Learning Objectives

---



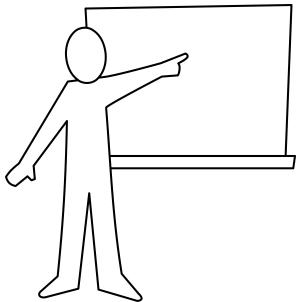
After this module you will be able to:

- Explain what Avro is, and how it can be used for data with a changing schema
- Write Avro messages to Kafka
- Use the Confluent Schema Registry for better performance

ybhandare@greendotcorp.com

# Module Map

---



- An Introduction to Avro ...
- Avro Schemas
- Using Confluent Schema Registry
- Hands-on Lab

ybhandare@greendotcorp.com

## Motivation

---

Always have a schema evolution plan in place...

*Chris Smith, VP, Engineering Data Science, Ticketmaster*

The quote is from the Web Cast found here: <https://www.confluent.io/online-talks/an-intro-to-ksql-and-kafka-streams-processing-with-ticketmaster>

When you're a company using Kafka to power your real-time event streaming platform, then sooner or later you will find out that your data schemas are not static. New (business) requirements ask for an evolution of your schemas. In this module we're discussing ways to achieve that goal.

ybhandare@greendotcorp.com

## The Need for a More Complex Serialization System

So far, all our data has been **plain text**...



PROS

- Supported by most programming languages
- **Easy to inspect** files for debugging

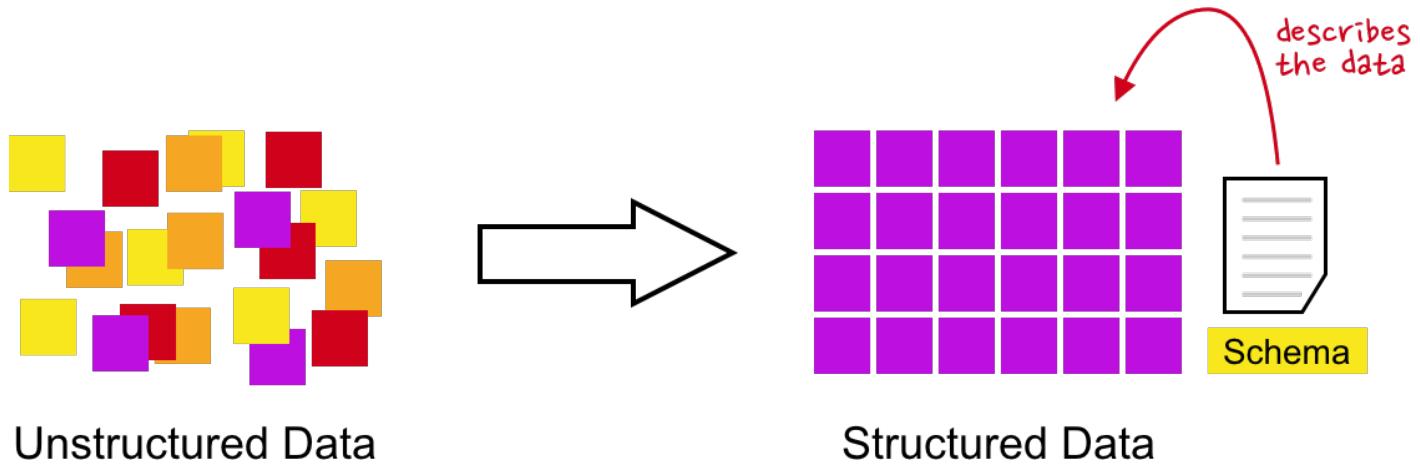


CONS

- Data is stored **inefficiently**
- Non-text data must be converted to strings
  - No type checking is performed
  - It is inefficient to convert binary data to strings
- No **schema evolution**

There are a few pros and cons when using a purely text based data format. One of the biggest drawbacks is the lack of a well defined **data schema**. We're going to talk about this in detail on the next slide.

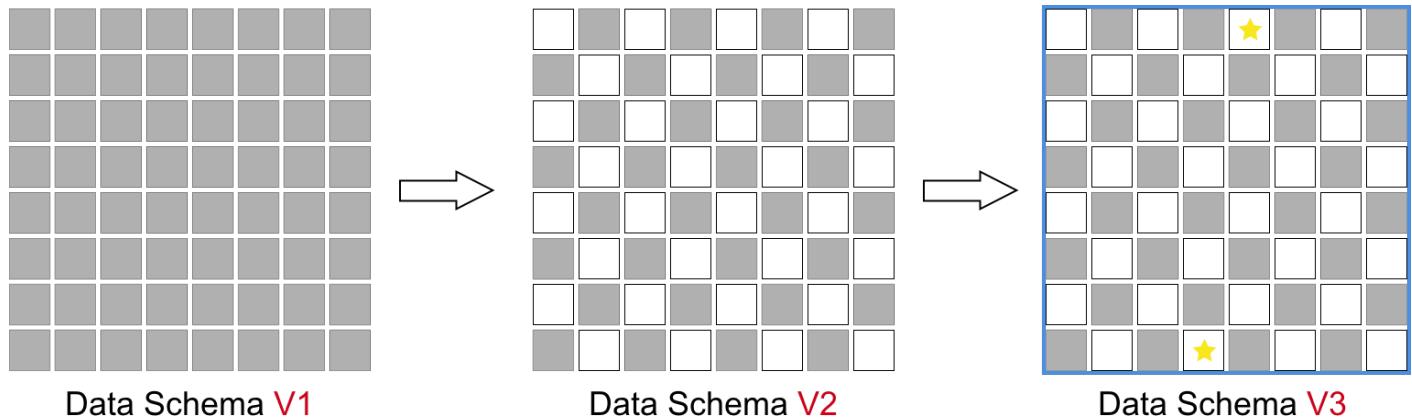
## The Data Schema



The **data schema** is what describes the structure of the data. This can be used as the contract between two parties, the producer of the data and the consumer(s) of the data. With this data the consumer(s) always know what to expect from the producer side. There are no surprises.

Furthermore, when you have no data schema then you have no means of evolving the structure of your data over time in a controlled way. This is called **schema evolution**.

## Data Schema Evolution



The image on the slide conceptually describes what goes on in "real-life". A structure (or schema) is conceived (v1). Over time new (business) requirements come into play and the structure/schema has to be adapted to the situation(v2). But that might not be the end, further refinements are required, resulting in further modifications (v3).

As business changes or more applications want to leverage the same data, there is a need for the existing data structures to evolve. We need what's called a **schema evolution**.

In asynchronous event-driven architecture, the data format is the API, and having support for schema evolution is as important for preserving compatibility as API compatibility is in RPC/request-response architecture. With event-first design, the data becomes the API which, like any production system, needs to support change and evolution (i.e., Avro or Protobuf)

## What is Serialization?



**i** Kafka has its own serialization classes in  
`org.apache.kafka.common.serialization`

First, let's start with some basics, what is **serialization** and where do we need it in our Kafka cluster?

- Serialization is a way of representing data in memory as a series of bytes
  - Needed to transfer data across the network, or store it on disk
- Deserialization is the process of converting the stream of bytes back into the data object
- Serialization is important since Kafka uses byte arrays as the format to transport and store data on the brokers
- However, serializers and deserializers are not standard across operating systems and programming languages
- Kafka tries to mitigate this issue by providing serialization classes but these are based on simple types and may not be enough for more complex data types.

# Avro: An Efficient Data Serialization System

- Avro is an Apache OSS project
- Provides data serialization
- Data is defined with a self-describing schema
- Supported by many programming languages, including Java
- Provides a data structure format
- Supports code generation of data types
- Provides a container file format
- Avro data is binary, so stores data efficiently
- Type checking is performed at write time

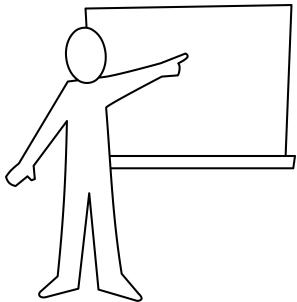


```
{  
  "namespace": "model",  
  "type": "record",  
  "name": "SimpleCard",  
  "fields": [  
    {  
      "name": "suit",  
      "type": "string",  
      "doc" : "The suit of the card"  
    },  
    {  
      "name": "denomination",  
      "type": "string",  
      "doc" : "The card number"  
    }  
  ]  
}
```

From clouddurable.com: "Apache Avro is a data serialization system. Avro provides data structures, binary data format, container file format to store persistent data, and provides RPC capabilities. Avro does not require code generation to use and integrates well with JavaScript, Python, Ruby, C, C#, C++ and Java. Avro gets used in the Hadoop ecosystem as well as by Kafka."

# Module Map

---



- An Introduction to Avro
- Avro Schemas ...
- Using Confluent Schema Registry
- Hands-on Lab

ybhandare@greendotcorp.com

# Avro Schemas

- Avro schemas define the **structure** of your data
- Schemas are represented in **JSON** or **IDL** format
- Avro has three different ways of creating records:

<b>Generic</b>	<ol style="list-style-type: none"><li>1. Manually create data type</li><li>2. Manually create schema</li></ol>
<b>Reflection</b>	<ol style="list-style-type: none"><li>1. Manually create data type</li><li>2. Generate schema from code</li></ol>
<b>Specific</b>	<ol style="list-style-type: none"><li>1. Manually write schema</li><li>2. Generate code to include in your program</li></ol> <p><i>Most common way to use Avro classes</i></p>

For Avro schemas to be useful in your environment, you must have data types in your programming language of choice and the schema to describe it in a JSON format for compatibility reasons (more on that later).



Schemas can also be defined in an IDL-based format. In fact, many customers use that option instead of JSON. <https://avro.apache.org/docs/1.8.2/idl.html>

IDL, short for **Interactive Data Language**, is a programming language used for data analysis.

What is the difference between the three ways of creating records?

- Generic: Manually create both the data type (Java class) and the schema (\*.avsc file)
- Reflection: Manually create the data type and then generate a schema from that code
- Specific: Manually write the schema and then generate the code to include in your program

## Avro Data Types (Simple)

- Avro supports several simple and complex data types
  - Following are the most common

Name	Description	Java equivalent
boolean	True or false	boolean
int	32-bit signed integer	int
long	64-bit signed integer	long
float	Single-precision floating-point number	float
double	Double-precision floating-point number	double

ybhandare@greendotcorp.com

## Avro Data Types (Simple)

Name	Description	Java equivalent
string	Sequence of Unicode characters	<code>java.lang.CharSequence</code>
bytes	Sequence of bytes	<code>java.nio.ByteBuffer</code>
null	The absence of a value	<code>null</code>

ybhandare@greendotcorp.com

## Avro Data Types (Complex)

Name	Description
record	A user-defined field comprising one or more simple or complex data types, including nested records
enum	A specified set of values
union	Exactly one value from a specified set of types
array	Zero or more values, each of the same type
map	Set of key/value pairs; key is always a <b>string</b> , value is the specified type
fixed	A fixed number of bytes
logical	Avro primitive or complex type with extra attributes to represent a derived type

**record** is the most important of these, as we will see

The complex types can be nested (e.g., a union of arrays, a map of enums).

Example for the use of **union**:

```
{  
  "type" : "record",  
  "namespace" : "tutorialspoint",  
  "name" : "empdetails ",  
  "fields" :  
  [  
    { "name" : "experience", "type": ["int", "null"] }, { "name" : "age", "type": "int" }  
  ]  
}
```

In this case the field **experience** can either be undefined (**null**) or an integer.

Regarding **logical** type: Refer to <https://avro.apache.org/docs/1.8.2/spec.html#Logical+Types> for more details

## Example Avro Schemas (1)

```
{  
  "namespace": "model",  
  "type": "record",  
  "name": "SimpleCard",  
  "fields": [  
    {  
      "name": "suit",  
      "type": "string",  
      "doc": "The suit of the card"  
    },  
    {  
      "name": "denomination",  
      "type": "string",  
      "doc": "The card number"  
    }  
  ]  
}
```

Schema of a **TrainArrived** event:

```
{  
  "namespace": "model",  
  "type": "record",  
  "name": "TrainArrived",  
  "fields": [  
    { "name": "trainId", "type": "int" },  
    { "name": "stationId", "type": "int" },  
    {  
      "name": "arrivalTime",  
      "type": "int",  
      "doc": "Time in ms since the epoch"  
    }  
  ]  
}
```

- This example creates a record object with two fields: a String for the name of the card and a String for the card number.
- **doc** allows you to place comments in the schema definition
- The **namespace** field is explained on the next slide.



Here **denomination** of type **string** may seem not the best choice. An **enum** would be preferable. But we will discuss the **enum** on a subsequent slide.

## Example Avro Schema (2)

---

- By default, the schema definition is placed in `src/main/avro`
  - File extension is `.avsc`
- The `namespace` is the Java package name, which you will import into your code

ybhandare@greendotcorp.com

## Example Schema with **array** and **map**

### Array

```
{  
  "namespace": "example.avro",  
  "name": "Parent",  
  "type": "record",  
  "fields": [  
    {  
      "name": "children",  
      "type": {  
        "type": "array",  
        "items": {  
          "name": "Child",  
          "type": "record",  
          "fields": [  
            {  
              "name": "name",  
              "type": "string"  
            }  
          ]  
        }  
      }  
    }  
  ]  
}
```

### Map

```
{  
  "namespace": "example.avro",  
  "type": "record",  
  "name": "Log",  
  "fields": [  
    {"name": "ip", "type": "string"},  
    {"name": "timestamp", "type": "string"},  
    {"name": "message", "type": "string"},  
    {  
      "name": "additional",  
      "type": {  
        "type": "map",  
        "values": "string"  
      }  
    }  
  ]  
}
```

Note that the **map** type is a complex type and could handle such things as optional fields in data structured like this:

```
{ "ip": "172.18.80.109", "timestamp": "2015-09-17T23:00:18.313Z", "message": "blahblahblah" }  
{ "ip": "172.18.80.112", "timestamp": "2015-09-17T23:00:08.297Z", "message": "blahblahblah",  
"microseconds": 223 }  
{ "ip": "172.18.80.113", "timestamp": "2015-09-17T23:00:08.299Z", "message": "blahblahblah",  
"thread": "http-apr-8080-exec-1147" }
```

where now the fields **microseconds** and **thread** are not mentioned explicitly...

## Example Schema with enum

```
{  
  "type": "record",  
  "name": "Test",  
  "namespace": "com.acme",  
  "fields": [  
    {  
      "name": "name",  
      "type": "string"  
    },  
    {  
      "name": "suit_type",  
      "type": {  
        "type": "enum",  
        "name": "Suit",  
        "symbols": ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]  
      },  
      "doc": "The suit of the card"  
    },  
  ],  
}
```

To allow for `null` values define the type of the enum like this:



```
"type": [  
  "null",  
  {  
    "type": "enum",  
    "name": "Suit",  
    "symbols": ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]  
  }  
]
```

Note that we define the value of `type` as an array including `null`.

# Logical Data Types

## Decimal

```
{  
  "type": "bytes",  
  "logicalType": "decimal",  
  "precision": 4,  
  "scale": 2  
}
```

## Date

```
{  
  "type": "int",  
  "logicalType": "date"  
}
```

## Timestamp (ms precision)

```
{  
  "type": "long",  
  "logicalType": "timestamp-millis"  
}
```

- The `decimal` logical type represents an arbitrary-precision signed `decimal` number of the form `unscaled × 10^(-scale)`
  - `scale`, a JSON integer representing the scale (optional). If not specified the scale is 0.
  - `precision`, a JSON integer representing the (maximum) precision of decimals stored in this type
- A `date` logical type annotates an Avro `int`, where the `int` stores the number of days from the unix epoch, 1 January 1970 (ISO calendar).
- A `timestamp-millis` logical type annotates an Avro `long`, where the `long` stores the number of milliseconds from the unix epoch, **1 January 1970 00:00:00.000 UTC**.

## Default and Null Values

### Default Value

```
{  
  "name": "suit_type",  
  "type" : {  
    "type" : "enum",  
    "name" : "Suit",  
    "symbols" : ["SPADES", "HEARTS",  
                "DIAMONDS", "CLUBS"],  
    "default": "SPADES"  
  },  
}
```

### Null Value

```
{  
  "name" : "experience",  
  "type": ["int", "null"]  
}
```

- The field `suit_type`, if missing in the source data, will be assigned the **default** value of `SPADES`
- The value of the field `experience` can be either undefined (`null`) or of type `int`

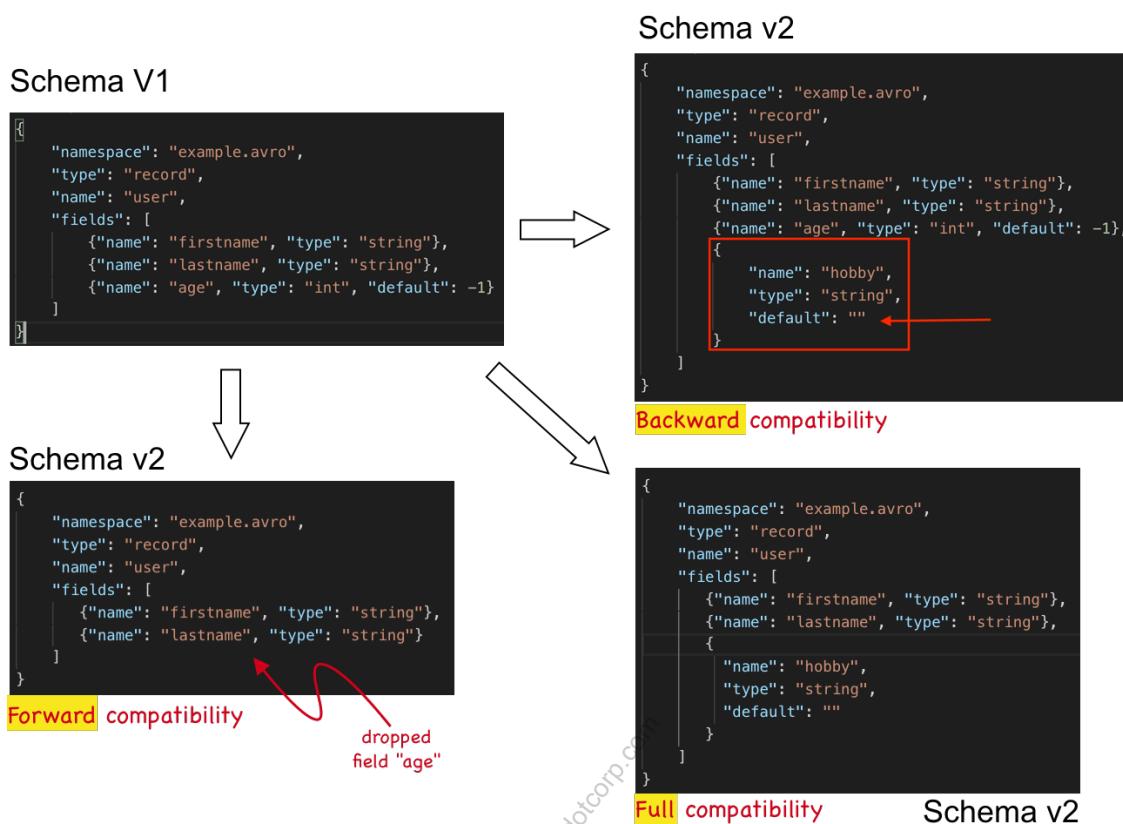
## Schema Evolution

- Avro schemas may evolve as updates to code happen
  - Schema Registry allows schema evolution
- We often want compatibility between schemas

Compatibility	Description
Backward	<p>Code with a <b>new version</b> of the schema:</p> <ul style="list-style-type: none"><li>• can read data written in the <b>old</b> schema</li><li>• assumes <b>default values</b> if fields are not provided</li></ul>
Forward	<p>Code with <b>previous versions</b> of the schema:</p> <ul style="list-style-type: none"><li>• can read data written in a <b>new</b> schema</li><li>• <b>ignores</b> new fields</li></ul>
Full	Forward and Backward

- **Backward compatibility example:** schema is written with fields (A, B); on the other side, schema is expecting to read fields (A, B, C); code reads (A, B) and assumes default value for C
- **Forward compatibility example:** schema is written with fields (A, B, C); on the other side, schema is expecting to read fields (A, B); code reads (A, B) and ignores C
- **Forward & Backward:** Only check compatibility against **previous** schema version
- **Forward transient:** Check compatibility against **all** previous schema versions
- **Backward transient:** Check compatibility against **all** previous schema versions

# Compatibility Examples



The various possible compatibility modes for schema evolution are defined as follows:

- **BACKWARD**: (default) consumers using the new schema can read data written by producers using the latest registered schema
- **BACKWARD\_TRANSITIVE**: consumers using the new schema can read data written by producers using all previously registered schemas
- **FORWARD**: consumers using the latest registered schema can read data written by producers using the new schema
- **FORWARD\_TRANSITIVE**: consumers using all previously registered schemas can read data written by producers using the new schema
- **FULL**: the new schema is forward and backward compatible with the latest registered schema
- **FULL\_TRANSITIVE**: the new schema is forward and backward compatible with all previously registered schemas
- **NONE**: schema compatibility checks are disabled

## Compatibility Examples

We recommend keeping the default **BACKWARD** compatibility.

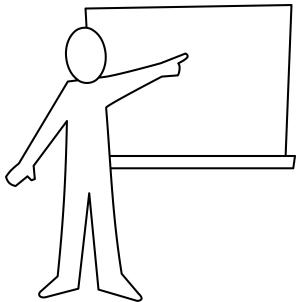


The supported schema compatibility settings mentioned here are more an SR concept, not settings within Avro itself!

ybhandare@greendotcorp.com

# Module Map

---



- An Introduction to Avro
- Avro Schemas
- Using Confluent Schema Registry ... 
-  Hands-on Lab

ybhandare@greendotcorp.com

## What Is Confluent Schema Registry?

- Confluent Schema Registry provides centralized management of schemas
  - Stores a versioned history of all schemas
  - Provides a RESTful interface for storing and retrieving Avro schemas
  - Checks schemas and throws an exception if data does not conform to the schema
  - Allows evolution of schemas according to the configured compatibility setting
- Sending the Avro schema with each message would be inefficient
  - Instead, a globally unique ID representing the Avro schema is sent with each message
- The Schema Registry stores schema information in a special Kafka topic
- The Schema Registry is accessible both via a REST API and a Java API
  - There are also command-line tools, `kafka-avro-console-producer` and `kafka-avro-console-consumer`

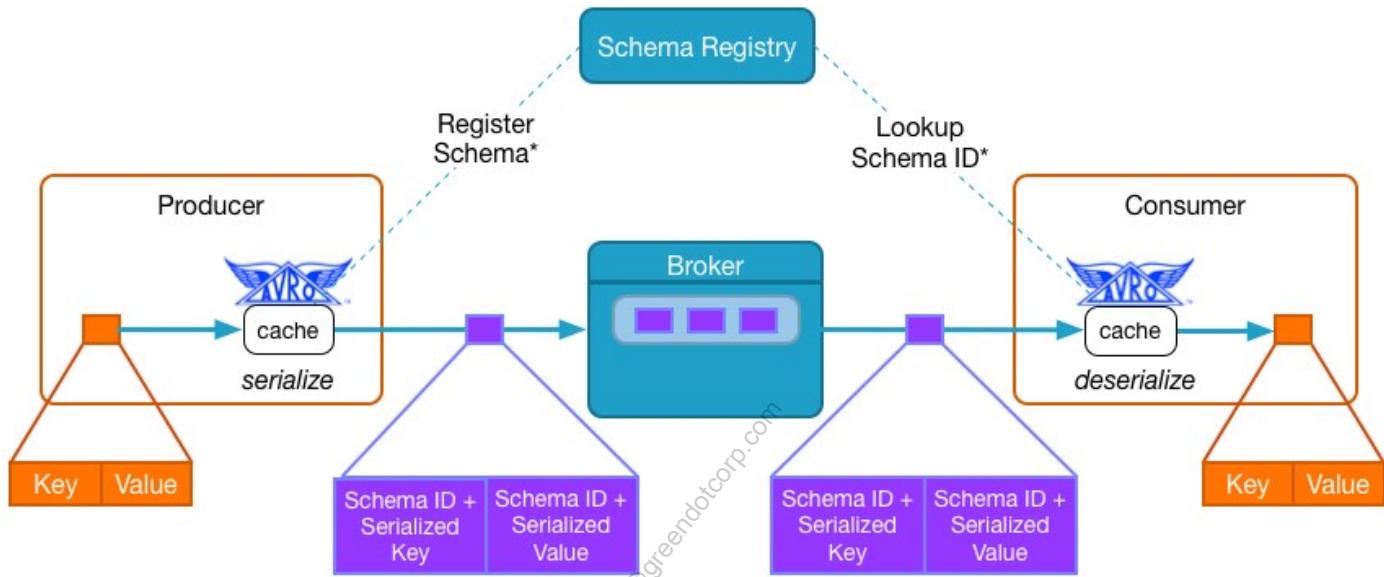
Schema Registry is a component that provides a management and lookup service for schemas. It provides a RESTful interface for storing and retrieving Avro schemas. It stores a versioned history of all schemas, provides multiple compatibility settings and allows evolution of schemas according to the configured compatibility setting. It provides serializers that plug into some of the Kafka clients (Java, Python, and .NET) that handle schema storage and retrieval for Kafka messages that are sent in the Avro format.

A special Kafka topic (default name is `"_schemas"`) is required and can be reconfigured with the `kafkastore.topic` parameter.

Spanning multiple data centers with your Schema Registry provides additional protection against data loss and improved latency. The recommended multi-datacenter deployment designates one datacenter as “master” and all others as “slaves”.

## Schema Registration and Data Flow

- The message key and value can be independently serialized
- Producers serialize data and prepend the schema ID
- Consumers use the schema ID to deserialize the data
- Schema Registry communication is only on the first message of a new schema
  - Producers and Consumers cache the schema/ID mapping for future messages



Traditionally, the full JSON schema is included with any data that is created with the associated data types. In a Kafka environment, this could significantly increase the size of the key and/or value of a message and so is unacceptable. The Schema Registry solves this issue by storing schemas in a special topic and identifying them by a ID number. That ID is included with the keys and values instead of the JSON schema.

Note the **\*** in the diagram next to "Register Schema" and "Lookup Schema ID". This is a reminder that this happens only on new schemas/IDs. Otherwise, previously used schemas are cached locally on the client.

## Integration with Schema Registry

- Java clients (producer, consumer)
- KSQL
- Kafka Streams
- Kafka Connect
- Confluent REST Proxy
- Non-Java clients based on [librdkafka](#)  
(except Go)

- **Java** based Kafka clients directly integrate with SR, that is it supports automated schema registration and lookup
- **KSQL** supports topics whose record values are encoded in AVRO out of the box
- **Kafka Streams** applications are written in Java and thus support the SR
- **Kafka Connect** directly integrates with SR via simple configuration settings
- **REST Proxy** also integrates with SR via simple configuration settings
- [librdkafka](#): Currently Confluent supports direct access to SR for Python, .NET and C/C++. Go is not yet offering direct SR support.

ybhandare@greendotcorp.com

## Other Clients & Schema Registry

- Other clients use the Schema Registry REST API to manually pre-register schemas and use the IDs in the requests

```
$ curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \
--data '{"schema": "{\"type\": \"string\"}'}' \
http://schemaregistry1:8081/subjects/<topic-name>-key/versions

$ curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \
--data '{"schema": "{\"type\": \"string\"}'}' \
http://schemaregistry1:8081/subjects/<topic-name>-value/versions

$ curl -X GET http://schemaregistry1:8081/schemas/ids/1
```



**<topic-name>** is the name of the topic where the schema is used.

Schema Registry is supported with other clients, but requires the registration and ID lookups to be done manually through the REST interface.

## Different Versions of Schemas in the Same Topic

- Different versions of schemas in the same Topic can be Backward, Forward, or Full compatible
  - Default is **BACKWARD**
- If they are neither, set compatibility to **NONE**
  - Code has no assumptions on schema as long as it is valid Avro
  - Code has full burden to read and process data
- Configuring compatibility
  - Use REST API

```
$ curl -X PUT -i -H "Content-Type: application/vnd.schemaregistry.v1+json" \
--data '{"compatibility": "NONE"}' \
http://schemaregistry1:8081/config/my_topic
```

Some people call a Topic that has multiple schemas a "fat" Topic. For a detailed discussion of when to use this approach, and the use of **custom subject naming strategies**, refer to:  
<https://www.confluent.io/blog/put-several-event-types-kafka-topic/>

When Confluent's Avro serializer registers a schema in the registry, it does so under a subject name. By default, that subject is **<topic>-key** for message keys and **<topic>-value** for message values. The schema registry then checks the mutual compatibility of all schemas that are registered under a particular subject.

# Subject Naming Strategies

Configurations	Naming Strategies
<ul style="list-style-type: none"><li>• <code>key.subject.name.strategy</code></li><li>• <code>value.subject.name.strategy</code></li></ul>	<ul style="list-style-type: none"><li>• <code>TopicNameStrategy</code> (default)</li><li>• <code>RecordNameStrategy</code></li><li>• <code>TopicRecordNameStrategy</code></li></ul>

## Configurations

- `key.subject.name.strategy`
- `value.subject.name.strategy`

3 possible settings are available for each of the above config property:

- `TopicNameStrategy` (default):  $\langle\text{subject-name}\rangle = \langle\text{topic}\rangle\text{-key}|\langle\text{topic}\rangle\text{-value}$
- `RecordNameStrategy`:  $\langle\text{subject-name}\rangle = \langle\text{type}\rangle$   
*This setting also allows any number of event types in the same topic*
- `TopicRecordNameStrategy`:  $\langle\text{subject-name}\rangle = \langle\text{topic}\rangle\text{-}\langle\text{type}\rangle$   
*This setting also allows any number of event types in the same topic, and further constrains the compatibility check to the current topic only*

Where `<topic>` is the topic name and `<type>` is the fully qualified Avro record type name



Alternatively users can create **different Topics** for different schemas.

## Java Avro Producer example

```
1 Properties props = new Properties();
2 props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
3 // Configure serializer classes
4 props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
5           KafkaAvroSerializer.class);
6 props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
7           KafkaAvroSerializer.class);
8 // Configure schema repository server
9 props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
10           "http://schemaregistry1:8081");
11 // Create the producer expecting Avro objects
12 KafkaProducer<Object, Object> avroProducer = new KafkaProducer<Object, Object>(props);
13 // Create the Avro objects for the key and value
14 CardSuit suit = new CardSuit("spades");
15 SimpleCard card = new SimpleCard("spades", "ace");
16 // Create the ProducerRecord with the Avro objects and send them
17 ProducerRecord<Object, Object> record = new
18 ProducerRecord<Object, Object>("my_avro_topic", suit, card);
19 avroProducer.send(record);
```

Why is ProducerRecord typed `<Object, Object>` on this slide if consumer.poll() returns `<CardSuit, SimpleCard>` on the next slide?

This illustrates that the Producer can send different types of Avro objects to different topics without having to instantiate different Producers for each topic and type of object. However on the Consumer side in this example, the Consumer is subscribing to a single topic, which is only going to have one type of Avro object, so it can be more specific in the typing. You can set the types to be more specific in the Producer, but it has to be done consistently throughout - in the KafkaProducer declaration and in the ProducerRecord declaration.

## Java Avro Consumer Example

```
1 public class CardConsumer {
2     public static void main(String[] args) {
3         Properties props = new Properties();
4         props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
5         props.put(ConsumerConfig.GROUP_ID_CONFIG, "testgroup");
6         props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
7                 KafkaAvroDeserializer.class);
8         props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
9                 KafkaAvroDeserializer.class);
10        props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
11                  "http://schemaregistry1:8081");
12        props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG, "true");
13
14        KafkaConsumer<CardSuit, SimpleCard> consumer = new KafkaConsumer<>(props);
15        consumer.subscribe(Arrays.asList("my_avro_topic"));
16
17        while (true) {
18            ConsumerRecords<CardSuit, SimpleCard> records = consumer.poll(Duration.ofMillis(100));
19            for (ConsumerRecord<CardSuit, SimpleCard> record : records) {
20                System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(),
21                                  record.key().getSuit(), record.value().getCard());
22            }
23        }
24    }
25 }
26 }
```

Note that in this code and the producer code on the previous page, there do not appear to be any methods which communicate directly with the Schema Registry. The interaction with the Schema Registry is abstracted into the `KafkaAvroSerializer` and `KafkaAvroDeserializer`.

Line 12 is required for the preferred behavior of "specific" code generation. If this line is omitted, the default of "generic" is used instead.



the `CardSuit` and `SimpleCard` types shown in these examples could possibly be generated from different schema versions, with compatibility enforced performed by the serializer on the Producer side before data is sent to Kafka (and on to Consumers).

## REST API Avro Producer Example

```
1 # Read in the Avro files
2 key_schema = open("my_key.avsc", 'rU').read()
3 value_schema = open("my_value.avsc", 'rU').read()
4
5 producerurl = "http://kafkarest1:8082/topics/my_avro_topic"
6 headers = {
7     "Content-Type" : "application/vnd.kafka.avro.v2+json"
8 }
9 payload = {
10     "key_schema": key_schema,
11     "value_schema": value_schema,
12     "records":
13     [
14         {
15             "key": {"suit": "spades"},
16             "value": {"suit": "spades", "denomination": "ace"}
17         }
18     ]
19 }
20 # Send the message
21 r = requests.post(producerurl, data=json.dumps(payload), headers=headers)
22 if r.status_code != 200:
23     print "Status Code: " + str(r.status_code)
24     print r.text
```

Note that the producer code sends its payload as JSON (and **not** as AVRO). The communication between the producer and the REST Proxy is over HTTP. Only the REST Proxy deals with AVRO serialization/deserialization.



The examples on this page and the next - although written in Python - use the REST Proxy, **not** the Python client.

This is just an example on **how to use** the REST API and not a recommendation for a Python client...

## REST API Avro Consumer Example

```
1 # Get the message(s) from the consumer
2 headers = {
3     "Accept" : "application/vnd.kafka.avro.v2+json"
4 }
5 # Request messages for the instance on the topic
6 r = requests.get(base_uri + "/topics/my_avro_topic", headers=headers, timeout=20)
7 if r.status_code != 200:
8     print "Status Code: " + str(r.status_code)
9     print r.text
10    sys.exit("Error thrown while getting message")
11 # Output all messages
12 for message in r.json():
13     keysuit = message["key"]["suit"]
14     valuesuit = message["value"]["suit"]
15     valuecard = message["value"]["denomination"]
16     # Do something with the data
```

Don't be surprised to see the client code deal with JSON and **not** with AVRO. AVRO serialization and deserialization is handled by the REST Proxy. Our client communicates via HTTP REST calls with the Proxy. From a consumer's perspective AVRO is hidden away. Only the producer needs to know about AVRO, as it needs to send the schemas with the POST request to the REST Proxy.

# AVRO Command-line Tools

## Producer

```
$ kafka-avro-console-producer \
--broker-list ${YOUR_BOOTSTRAP_SERVER} \
--property schema.registry.url=schemaregistry1:8081 \
--topic my_avro_topic \
--property value.schema="${MY_AVRO_SCHEMA}"
```

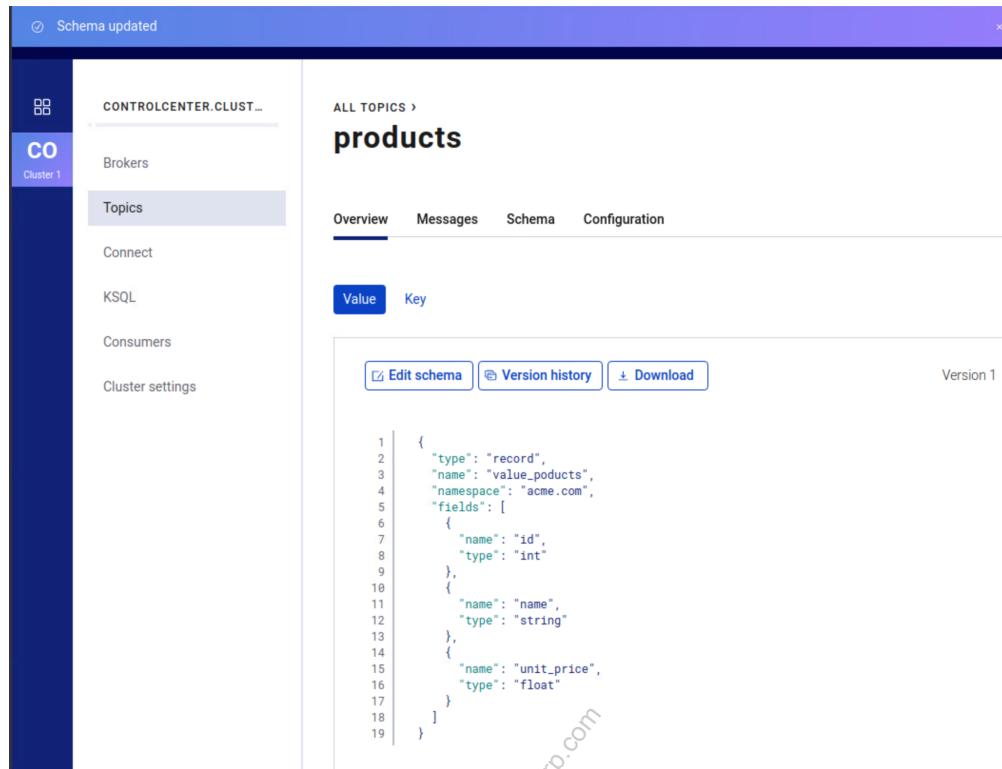
## Consumer

```
$ kafka-avro-console-consumer \
--bootstrap-server broker1:9092 \
--property schema.registry.url=schemaregistry1:8081 \
--from-beginning \
--topic my_avro_topic
```

In the producer example, `MY_AVRO_SCHEMA` is an environment variable that contains the schema. It may be defined like this:

```
$ MY_AVRO_SCHEMA='{
  "type": "record",
  "namespace": "example.avro",
  "name": "product",
  "fields": [{"name": "id", "type": "int"},
             {"name": "name", "type": "string"},
             {"name": "unit_price", "type": "float"}]
}'
```

# Viewing Schemas in Confluent Control Center



The screenshot shows the Confluent Control Center interface. On the left, a sidebar for 'Cluster 1' lists 'Brokers', 'Topics' (which is selected), 'Connect', 'KSQL', 'Consumers', and 'Cluster settings'. The main area is titled 'products' under 'ALL TOPICS'. It has tabs for 'Overview', 'Messages', 'Schema', and 'Configuration', with 'Schema' selected. Below the tabs, there are buttons for 'Edit schema', 'Version history', and 'Download'. The schema is displayed as a JSON-like code block:

```
1  {
2   "type": "record",
3   "name": "value_products",
4   "namespace": "acme.com",
5   "fields": [
6     {
7       "name": "id",
8       "type": "int"
9     },
10    {
11      "name": "name",
12      "type": "string"
13    },
14    {
15      "name": "unit_price",
16      "type": "float"
17    }
18  ]
19 }
```

The ability to view Schema Registry information was introduced in Confluent Enterprise 5.0.

## Features:

- View key & value schemas
- Edit key & value schemas
- View previous schema versions & compare against current
- Download schemas

### Questions:

- When does the use of Avro make sense?
- What does the Confluent Schema Registry do for you?



- Using a serialization format such as Avro makes sense for complex data
- The Schema Registry makes it easy to efficiently write and read Avro data to and from Kafka by centrally storing the schema. It also supports schema evolution

ybhandare@greendotcorp.com

## Hands-On Lab

---

- In this Hands-On Exercise, you will write and read Kafka data with Avro
- Please refer to the lab **Using Kafka with Avro** in Exercise Book



ybhandare@greendotcorp.com



### Data Pipelines with Kafka Connect

ybhandare@greendotcorp.com

# Agenda

---



1. Introduction
2. Fundamentals of Apache Kafka
3. Kafka's Architecture
4. Developing With Kafka
5. More Advanced Kafka Development
6. Schema Management In Kafka Streams
7. Data Pipelines with Kafka Connect ... 
8. Stream Processing with Kafka Streams
9. Stream Processing with KSQL
10. Event Driven Architecture
11. Confluent Cloud
12. Conclusion
13. Appendix: Basic Kafka Administration
14. Appendix: Apache Kafka driving ML & Data Analytics

ybhandare@greendotcorp.com

## Learning Objectives

---

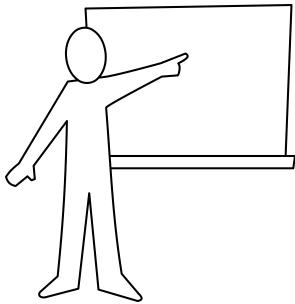


After this module you will be able to:

- Explain the motivation for Kafka Connect
- Explain the differences between standalone and distributed mode
- Configure and use Kafka Connect
- List standard Connectors that are provided
- Compare Kafka Connect to writing your own data transfer system

ybhandare@greendotcorp.com

# Module Map



- The Motivation for Kafka Connect ... ←
- Types of Connectors
- Kafka Connect Implementation
- Standalone and Distributed Modes
- Configuring the Connectors
-  Hands-on Lab

ybhandare@greendotcorp.com

## GOALS

---



1. Focus on copying
2. Batteries included
3. Standardize
4. Parallelism
5. Scale

The goals of a framework that provides integration with Kafka are:

1. **Focus on copying:** Its all about getting data in and getting data out. Nothing more
2. **Batteries included:** The Kafka platform should include this framework and with it the most popular plugins for import and export of data
3. **Standardize:** The framework should standardize the task of importing and exporting data. No 1-off solutions!
4. **Parallelism:** The framework should be able to parallelize the work of say importing a bunch of tables or files, as much as possible
5. **Scale:** We want to be able to scale out the workload, similar to how we can e.g. scale out brokers or consumer groups

YBHAZIAREDBRNDONDOTCOM

## What is Kafka Connect?

- Kafka Connect is a framework for streaming data between Apache Kafka and other data systems
- Kafka Connect is open source, and is part of the Apache Kafka distribution
- It is simple, scalable, and reliable



Kafka Connect is not an API like the Client API (which implements Producers and Consumers within the applications you write) or Kafka Streams. It is a reusable framework that uses plugins called Connectors to customize its behavior for the endpoints you choose to work with.

ybhandare@greendotcorp.com

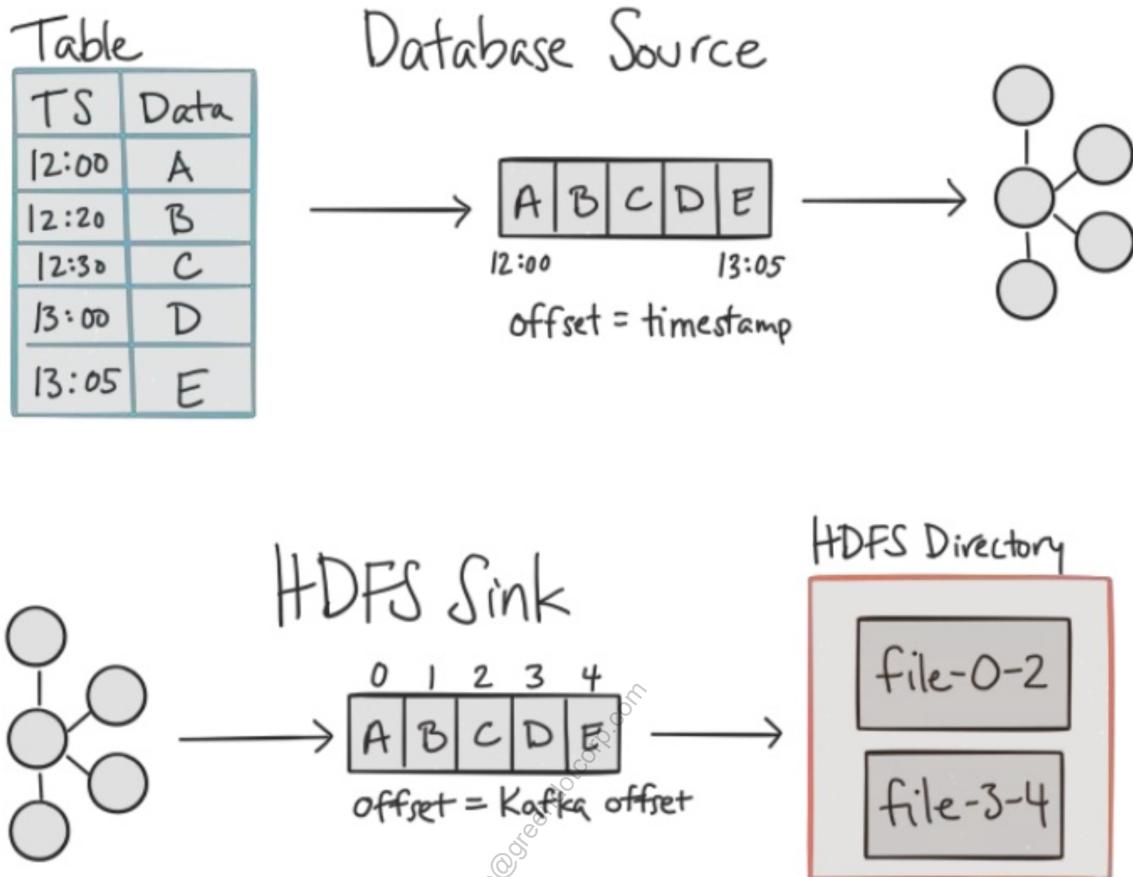
## Connect Basics

- Connectors are logical jobs responsible for managing the copying of data between Kafka and another system
- Connector Sources read data *from* an external data system into Kafka
  - Internally, source connector is a Kafka Producer
- Connector Sinks write Kafka data *to* an external data system
  - Internally, sink connector is a Kafka Consumer Group

An instance of Connect (called a worker) can function as both a producer and consumer, depending on the type of connector you install. In fact, internally connectors are running the standard client code: Producer (source connectors) and Consumer (sink connectors).

Source connectors read from external data stores (e.g., databases). Sink Connectors pull data from a Kafka topic and write it out to an external application (e.g., HDFS, Elasticsearch).

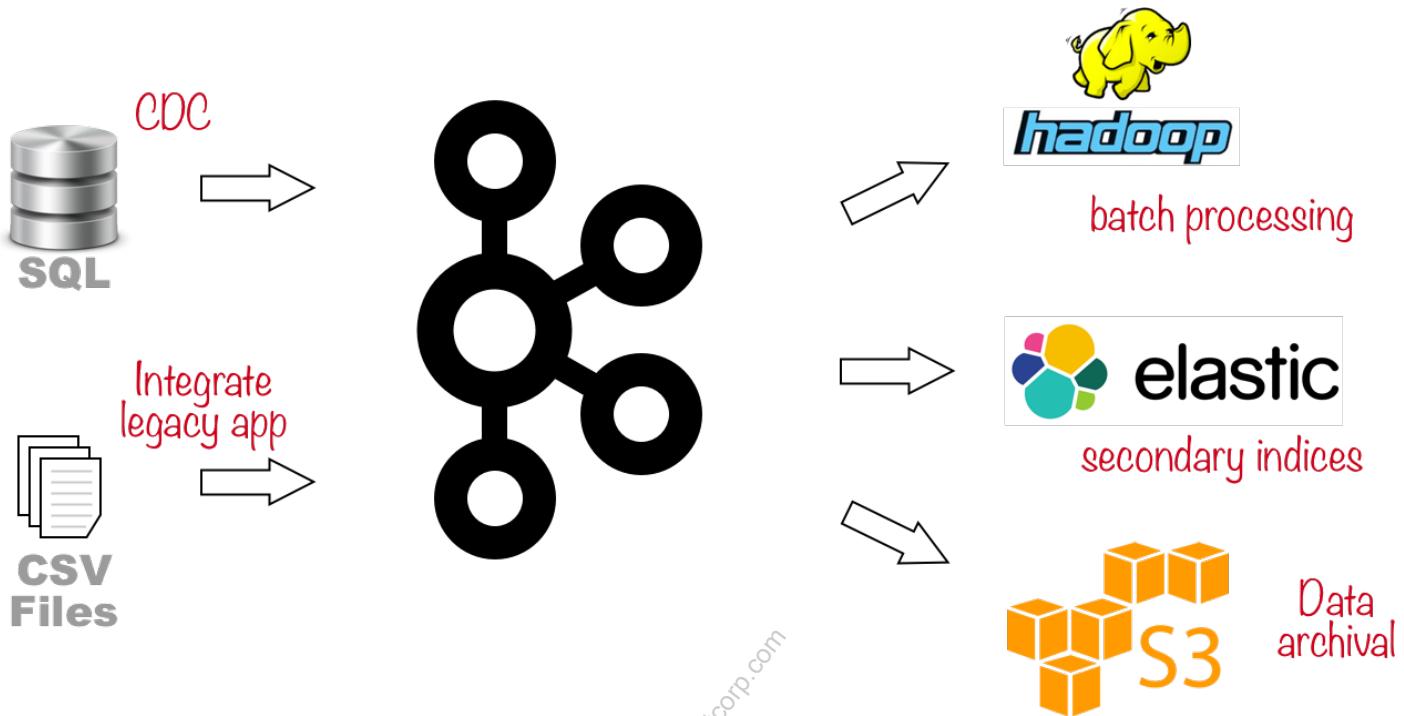
The term "Connector" can be used for either the type of plugin (e.g., the HDFS Connector) or a configured instance of a connector (e.g., the mycompany\_HDFS connector).



Here we see a sample source and a sample sink.

- **Source:** Data originating in a relational DB is imported into Kafka. The timestamp from the table is "translated" into what's the offset in Kafka
- **Sink:** Data from Kafka is exported to a HDFS directory. The Kafka offset becomes part of the filenames in the sink (i.e. from → to)

## Example Use Cases



- Example use cases for Kafka Connect include:
  - Stream an entire SQL database into Kafka
  - Import CSV files generated by legacy app into Kafka
  - Stream Kafka Topics into HDFS for batch processing
  - Stream Kafka Topics into Elasticsearch for secondary indexing
  - Many companies have huge amount of data they want to archive and saving cost

## Why Not Just Use Producers and Consumers?



### Anti Patterns

1. One-off tools
2. Kitchen sink tools
3. Stream processing frameworks



### Kafka Connect

- Off-the-shelf
- Tested
- Connectors for common sources & sinks
- Fault tolerance
- Automatic load balancing
- No coding required
- Pluggable/extendable by developers



Internally, Kafka Connect is just a Kafka client using the standard Producer and Consumer APIs

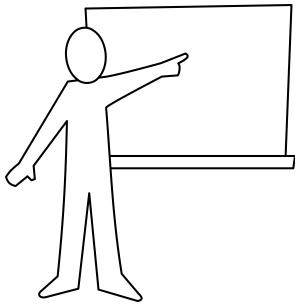
Producers and consumers provide complete flexibility to send any data to Kafka or process it in any way. This flexibility means you do everything yourself.

Typically the Client API is included in your application as a library. But what if you don't have code access? You'll need to write a client which can behave as a producer or consumer between the external applications and the cluster. Writing your own producers and consumers provides complete flexibility to send any data to Kafka or process it in any way, but this flexibility means you do everything yourself.

Connect is a prebuilt framework which provides the basic client architecture (using the standard Kafka Client API). This framework is written to make Connect highly available and scalable. However, it relies on plugins called **Connectors** to provide the specific functionality to communicate with your chosen endpoints.

Connect is restricted to data copying, with light data transformations.

# Module Map



- The Motivation for Kafka Connect
- Types of Connectors ... ←
- Kafka Connect Implementation
- Standalone and Distributed Modes
- Configuring the Connectors
-  Hands-on Lab

ybhandare@greendotcorp.com

# Confluent Platform Connectors

## Source Connectors

Verified Standard	Verified Standard	Confluent Supported	Confluent Supported
 CockroachDB Change Data Capture Cockroach Labs <a href="#">Read More</a>	 Vertical Analytics Platform Vertica <a href="#">Read More</a>	 Debezium MySQL CDC Connector Debezium Community <a href="#">Read More</a>	 Debezium PostgreSQL CDC Connector Debezium Community <a href="#">Read More</a>
Confluent Supported	Confluent Supported	Confluent Supported	Confluent Supported
 Debezium SQL Server CDC Connector Debezium Community <a href="#">Read More</a>	 Debezium MongoDB CDC Connector Debezium Community <a href="#">Read More</a>	 Kafka Connect Data Diode Confluent, Inc. <a href="#">Read More</a>	 Kafka Connect ActiveMQ Source Confluent, Inc. <a href="#">Read More</a>
Confluent Supported	Confluent Supported	Confluent Supported	Confluent Supported
 Kafka Connect Syslog Confluent, Inc. <a href="#">Read More</a>	 Kafka Connect JDBC Confluent, Inc. <a href="#">Read More</a>	 Kafka Connect Salesforce Confluent, Inc. <a href="#">Read More</a>	 Kafka Connect Kafka Replicator Confluent, Inc. <a href="#">Read More</a>
Confluent Supported	Confluent Supported	Confluent Supported	Confluent Supported
 Kafka Connect S3 Source Confluent, Inc. <a href="#">Read More</a>	 Kafka Connect Datanan Confluent, Inc. <a href="#">Read More</a>	 Kafka Connect IBM MQ Source Confluent, Inc. <a href="#">Read More</a>	 Kafka Connect MQTT Confluent, Inc. <a href="#">Read More</a>

## Sink Connectors

Verified Standard	Confluent Supported	Confluent Supported	Confluent Supported
 Vertical Analytics Platform Vertica <a href="#">Read More</a>	 Kafka Connect Data Diode Confluent, Inc. <a href="#">Read More</a>	 Kafka Connect OmniSci Sink Confluent, Inc. <a href="#">Read More</a>	 Kafka Connect MapRDB Confluent, Inc. <a href="#">Read More</a>
Confluent Supported	Confluent Supported	Confluent Supported	Confluent Supported
 Kafka Connect JDBC Confluent, Inc. <a href="#">Read More</a>	 Kafka Connect Salesforce Confluent, Inc. <a href="#">Read More</a>	 Kafka Connect ActiveMQ Sink Confluent, Inc. <a href="#">Read More</a>	 Kafka Connect Azure Blob Storage Sink Connector Confluent, Inc. <a href="#">Read More</a>
Confluent Supported	Confluent Supported	Confluent Supported	Confluent Supported
 Kafka Connect Elasticsearch Confluent, Inc. <a href="#">Read More</a>	 Kafka Connect HDFS Confluent, Inc. <a href="#">Read More</a>	 Kafka Connect Http Sink Connector Confluent, Inc. <a href="#">Read More</a>	 Kafka Connect Vertica Confluent, Inc. <a href="#">Read More</a>
Confluent Supported	Confluent Supported	Confluent Supported	Confluent Supported
 Kafka Connect Cassandra Confluent, Inc. <a href="#">Read More</a>	 Kafka Connect JMS Sink Confluent, Inc. <a href="#">Read More</a>	 Kafka Connect IBM MQ Sink Confluent, Inc. <a href="#">Read More</a>	 Kafka Connect HDFS 3 Confluent, Inc. <a href="#">Read More</a>

The slide shows a fraction of the source and sink connectors available on <https://confluent.io/hub>

Some of the connectors are included as part of the CP download and do not require additional steps to install.

- Confluent Community Edition ships with commonly used Connectors
  - FileStream
  - JDBC
  - HDFS
  - AWS S3
  - Elasticsearch
- Confluent Enterprise includes additional connectors
  - Replicator
  - Google Cloud Storage (GCS)
  - JMS
  - IBM MQ
  - ActiveMQ
  - Cassandra

Customers who are looking for other connectors will have to check Confluent Hub (<https://www.confluent.io/hub/>) or the Connectors page.

## Confluent Platform Connectors

---

For a detailed description of Confluent Hub:

<https://www.confluent.io/blog/introducing-confluent-hub/>



The goal of Confluent is to provide a comprehensive set of supported connectors.

ybhandare@greendotcorp.com

# Installing New Connectors

## From Confluent Hub

- Use `confluent-hub` client included with Confluent Platform

```
$ confluent-hub install debezium/debezium-connector-mysql:latest
```

## From other sources

- Package Connector in JAR file
- Install JAR file on **all** Kafka Connect worker machines
  - Connectors are installed as plugins
  - There is **library isolation** between plugins

```
plugin.path=/path/to/my/plugins
```

Prior to Kafka 0.11.0, JARs were placed in a path specified by `CLASSPATH` and there was no library isolation between connectors

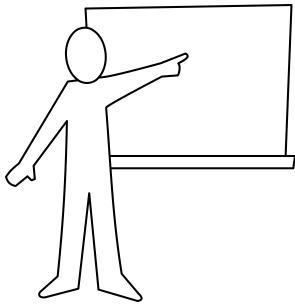
```
$ export CLASSPATH="$CLASSPATH:/path/to/my/connectors/*"
```

**QUESTION:** Why is library isolation important?

Answer: *Connectors built by different developers might use different versions of the same library.*

# Module Map

---



- The Motivation for Kafka Connect
- Types of Connectors
- Kafka Connect Implementation ...
- Standalone and Distributed Modes
- Configuring the Connectors
- Hands-on Lab

ybhandare@greendotcorp.com

# Kafka Connect Reliability and Scalability

- Elasticity and scalability
  - To add resources to Connect simply start more workers
  - Connect Workers discover each other and load-balance the work automatically
  - Allows running more connectors and more tasks
- Reliability and fault-tolerance
  - Failed connector will automatically restart
  - Tasks and connectors on failed worker will automatically (re)start on another worker
- ALL connectors support at-least-once semantics
- SOME connectors support exactly-once semantics
- MANY connectors support schema evolution
  - Data format changes in source system will reflect in Kafka and all target systems

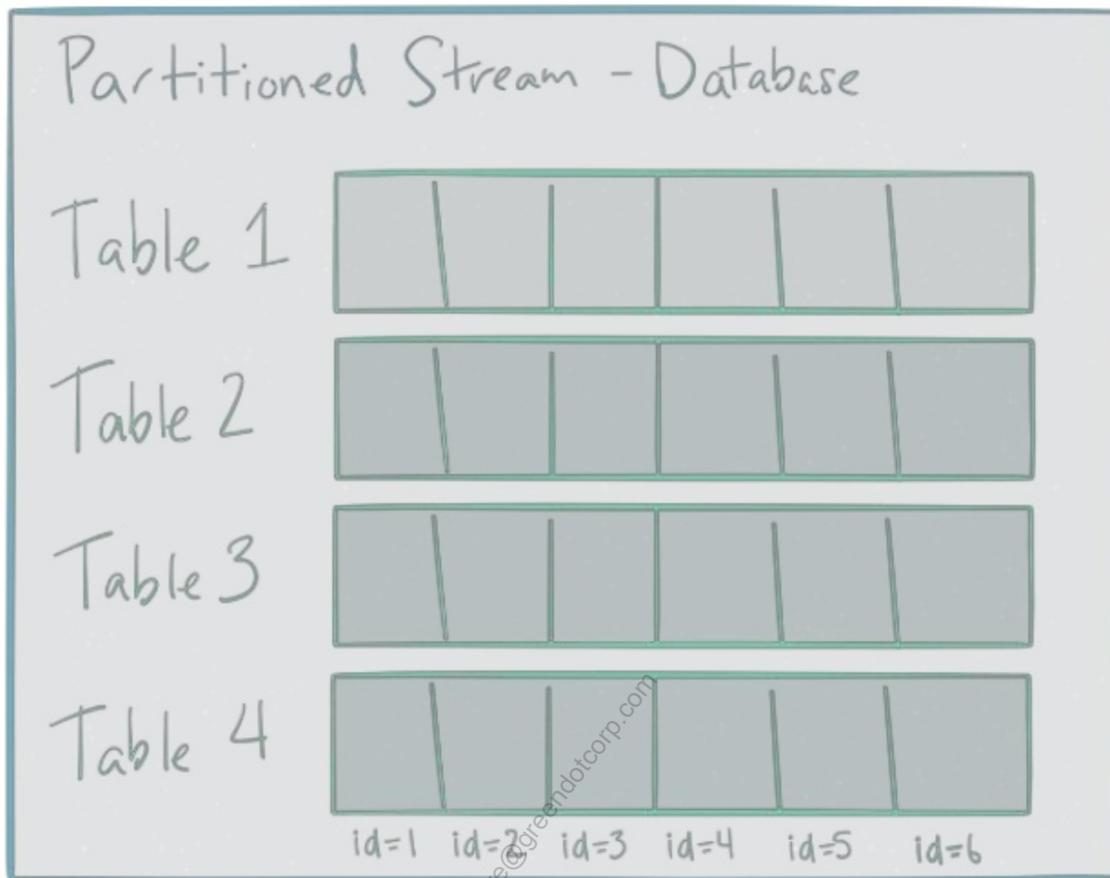


To clarify the point "**Failed Connector will automatically restart**": If a worker fails, any connectors running on that worker will automatically be restarted. If a connector thread or task fails (while the worker continues to run), fault tolerance is not automatic!



In Kafka we partition a topic (or a stream) into partitions to parallelize work.

ybhandare@greendotcorp.com



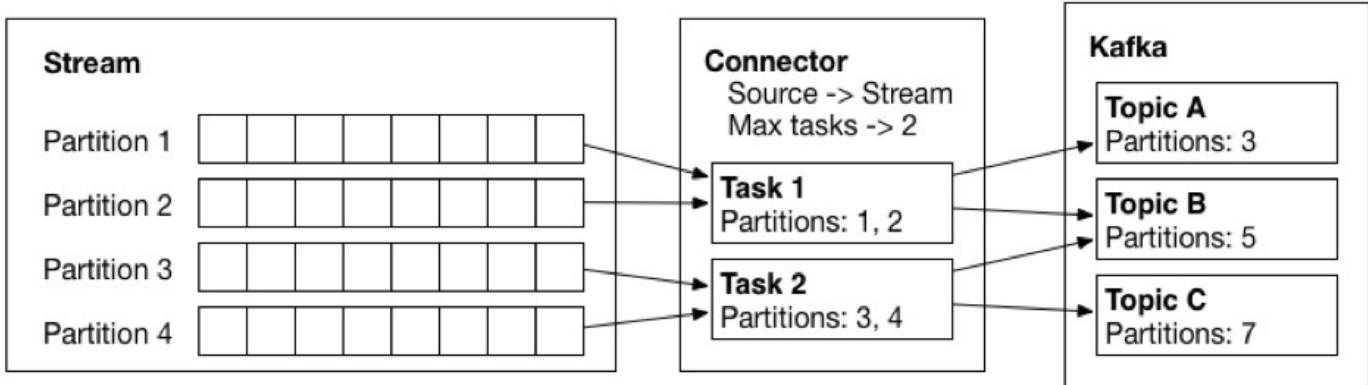
Kafka Connect also uses the concept of partitioning the input stream of data. Admittedly it is a bit **confusing** to use the same terminology for two different things...

In case of a relational DB as source, the **partitions** of the input stream of data would correspond to the **tables** of the DB

In case of importing CSV files, the **partitions** would correspond to **files**

# Providing Parallelism and Scalability

- Partitioning → parallelism & scalability
- 1 Connector job → 1..n **tasks**
- 1 Worker → 1..n **tasks**
- 1 Task per Thread



- Splitting the workload into smaller pieces provides the parallelism and scalability
- Connector jobs are broken down into **tasks** that do the actual copying of the data
- **Workers** are processes running one or more tasks, each in a different thread

A Connector is a Connector class and a configuration. Each connector defines and updates a set of tasks that actually copy the data. Connect distributes these tasks across workers.

In the case of Connect, the term **partition** can mean any subset of data in the source or the sink. How a partition is represented depends on the type of Connector (e.g., tables are partitions for the JDBC connector, files are the partitions for the FileStream connector).

Worker processes are not managed by Kafka. Other tools can do this: Kubernetes, Mesos, YARN, Chef, Puppet, custom scripts, etc.



not all connectors support multiple tasks and parallelism. For example, the **syslog** source connector only supports one task.

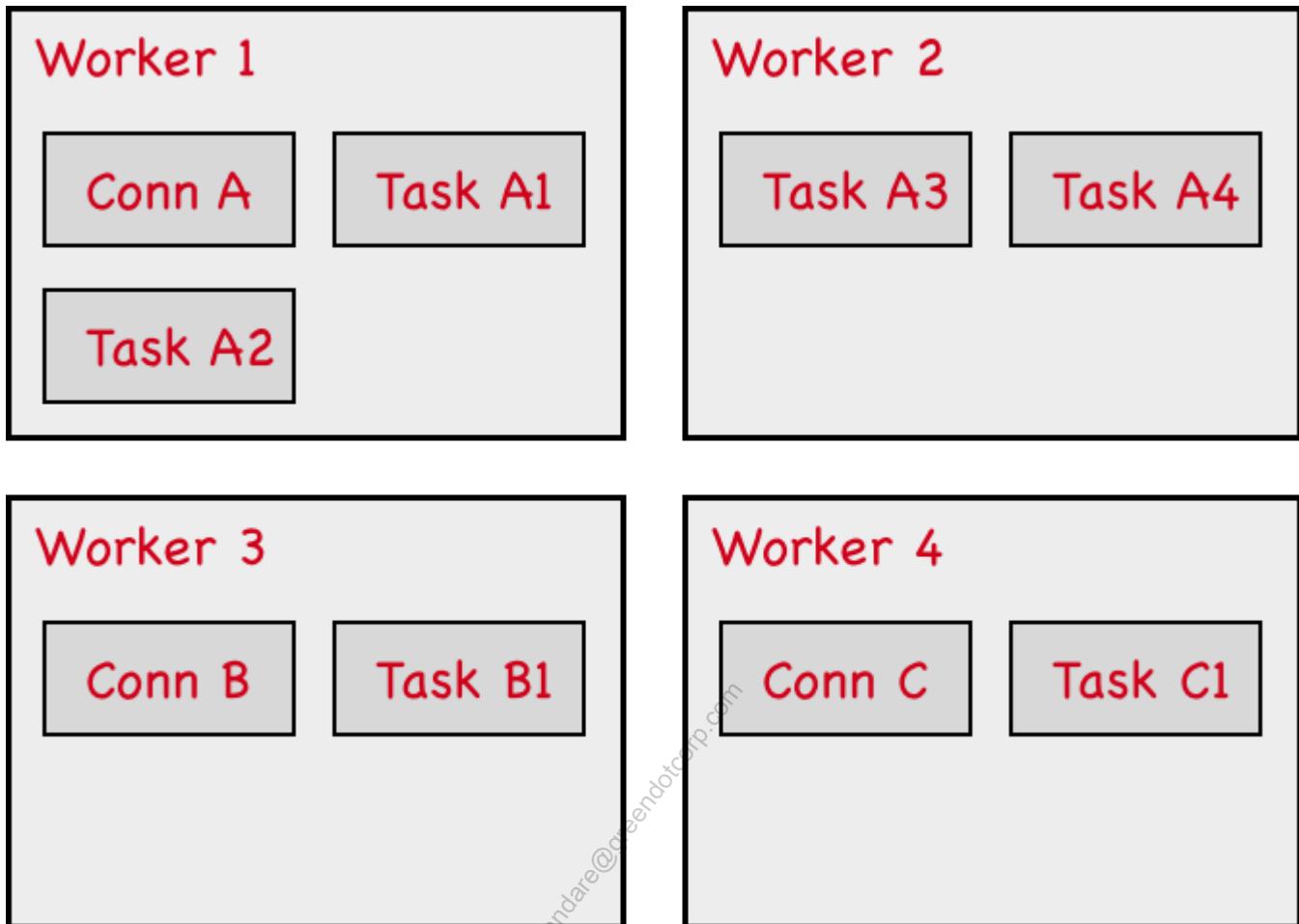
## Source and Sink Offsets

---

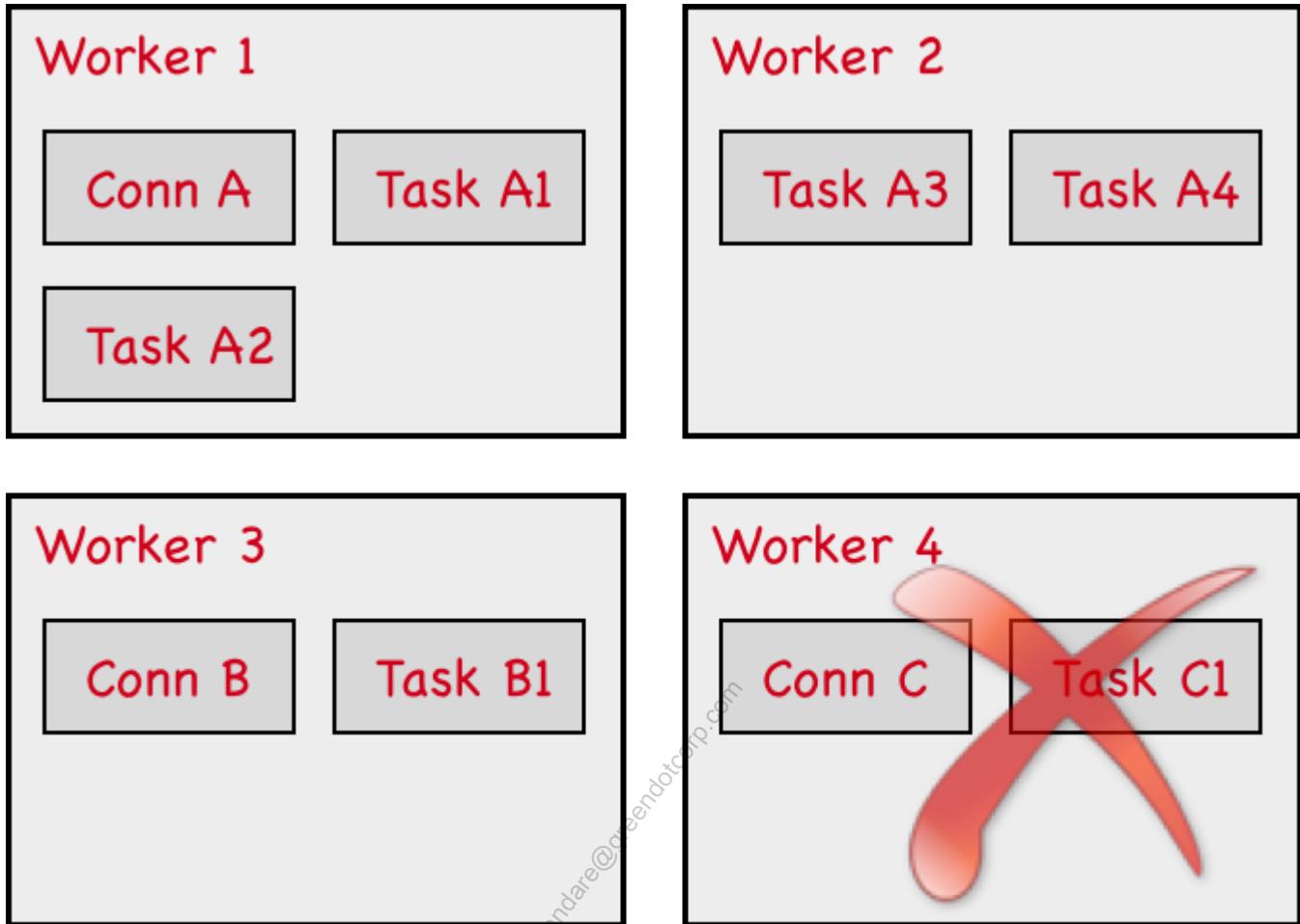
- Kafka Connect tracks the produced and consumed offsets so it can restart at the correct place, in case of failure
- What the offset corresponds to depends on the Connector, for example:
  - File input: offset → position in file
  - Database input: offset → timestamp or sequence id
- The method of tracking the offset depends on the specific Connector
  - Each Connector can determine its own way of doing this
- Examples of source and sink offset tracking methods:
  - JDBC source in distributed mode: a special Kafka Topic
  - HDFS sink: an HDFS file
  - FileStream source: a separate local file

As with Kafka topics, the position within the partitions used by Connect must be tracked as well to prevent the unexpected replay of data. As with the partitions, the object used as the offset varies from connector to connector, as does the method of tracking the offset. The most common way to track the offset is through the use of a Kafka topic, though the Connector developer can use whatever he or she wants.

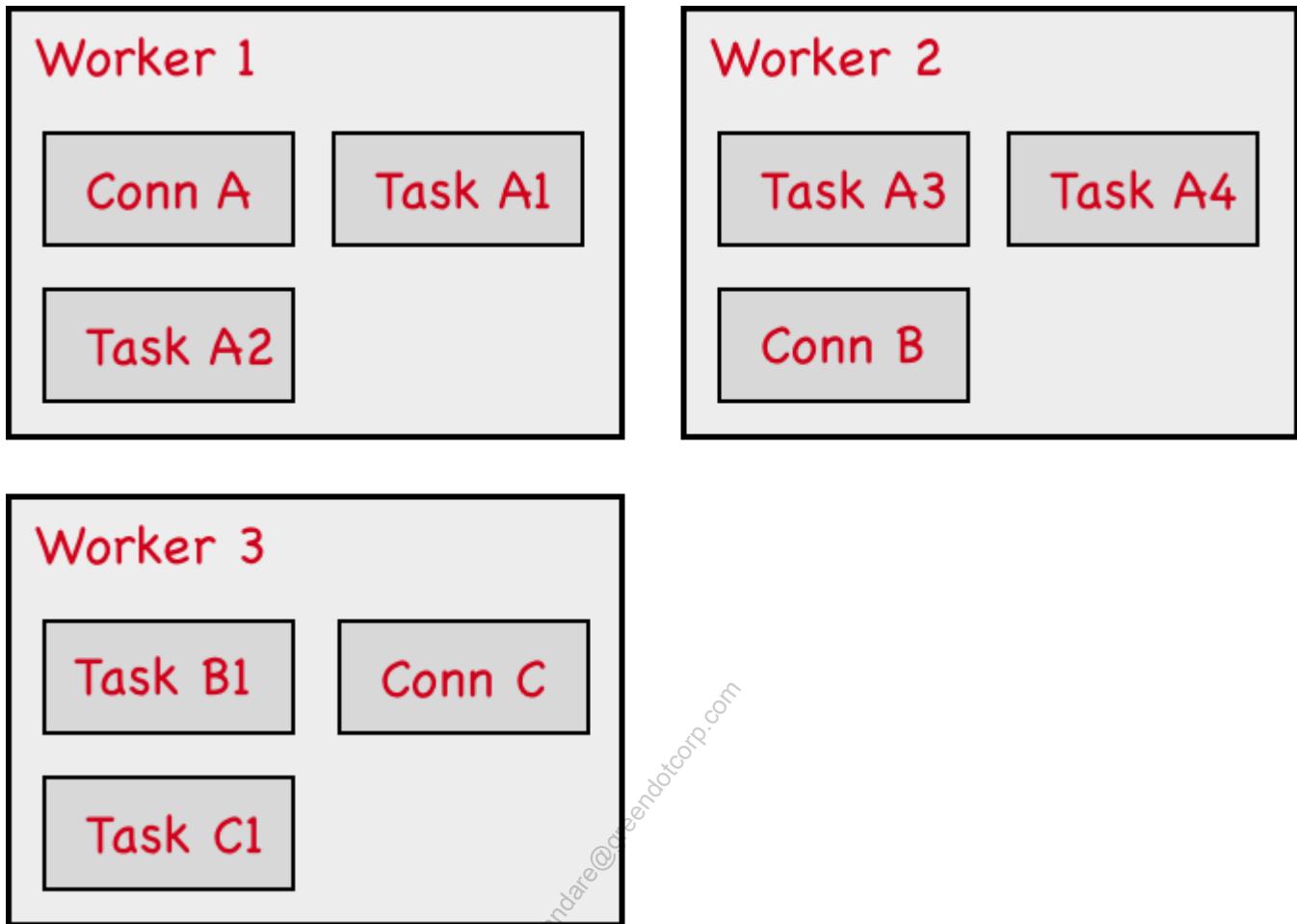
The variation is primarily with source connectors. With sink connectors, the offsets are Kafka topic partition offsets, and they're usually stored in the external system where the data is being written.



Kafka Connect, when run in distributed mode, automatically and transparently does the failover for you. Here we have 4 workers with 3 connectors A, B, C. Connector A has 4 tasks running on workers 1 and 2 whilst connectors B and C each have a single task running on worker 3 and 4 respectively.



Let's now assume that worker 4 fails... what happens to connector C with its task C1?

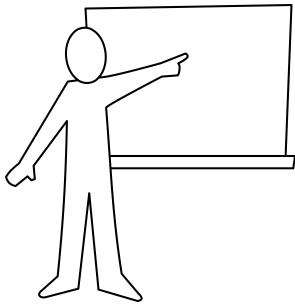


As we can see, Kafka Connect uses a similar technique that we know from consumer groups to reschedule or reallocate workload evenly across the available workers. In our case the connector job B has been moved to worker 2 whilst the connector job C and task C1 have been moved to worker 3.

Rebalancing/task assignment is performed by the Worker, not by the Connector instance thread. Much like consumer groups, there's a concept of a "leader" Worker which performs task reassessments whenever rebalancing is triggered, which can involve movement of connector threads in addition to tasks. See <https://github.com/apache/kafka/blob/2.2/connect/runtime/src/main/java/org/apache/kafka/connect/runtime/distributed/WorkerCoordinator.java#L220-L258> for more detail.

# Module Map

---



- The Motivation for Kafka Connect
- Types of Connectors
- Kafka Connect Implementation
- Standalone and Distributed Modes ... ↩
- Configuring the Connectors
-  Hands-on Lab

ybhandare@greendotcorp.com

## Two Modes: Standalone and Distributed

### Standalone mode

- 1 worker on 1 machine
- When to use:
  - testing and development
  - non distributable process

### Distributed mode

- $n$  workers on  $m$  machines
- When to use:
  - production environments
  - for **fault tolerance**
  - and **scalability**

Kafka Connect can be run in two modes

- **Standalone**
  - 1 worker process on single machine
  - Use cases:
    - testing and development, or
    - when a process should not be distributed (e.g. `tail` a log file)
- **Distributed mode**
  - Multiple worker processes on one or more machines
  - Use Case: requirements for fault tolerance and scalability

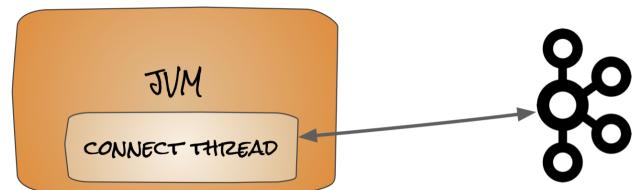
## Running in Standalone Mode

To run in standalone mode, start a process by providing as arguments

- Standalone config properties file
- 1...n connector config files



Each connector instance runs in own thread



```
$ connect-standalone connect-standalone.properties \
  connector1.properties [connector2.properties connector3.properties ...]
```

A single worker can run as many connectors as you wish. Once the connectors are running, changes to their configuration files will only take effect on restart of the worker. Making changes to the connectors without a restart will be discussed later in the chapter.

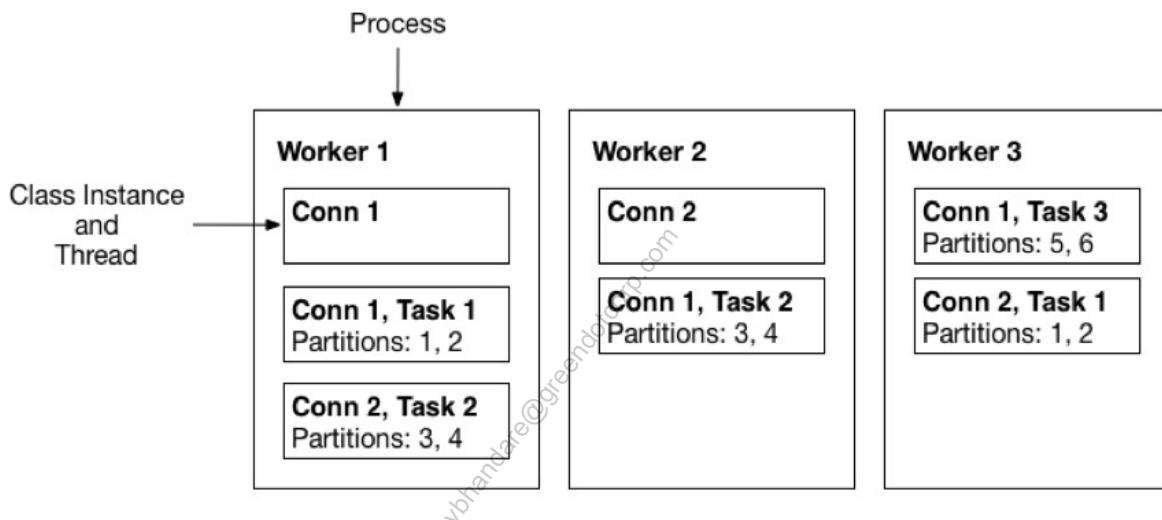
# Running in Distributed Mode

Start Kafka Connect **on each** worker node

```
$ connect-distributed connect-distributed.properties
```

## Group coordination

- Connect leverages Kafka's group membership protocol
  - Configure workers with the same `group.id`
- Workers distribute load within this Kafka Connect "cluster"



In distributed mode, the connector configurations **cannot** be kept on the Worker systems; a failure of a worker should not make the configurations unavailable. Instead, distributed workers keep their connector configurations in a special Kafka topic which is specified in the worker configuration file.

Workers coordinate their tasks to distribute the workload using the same mechanisms as Consumer Groups.

## Configuring Workers: Both Modes

- You can modify Connect configuration settings
  - Distributed mode in `/etc/kafka/connect-distributed.properties`
  - Standalone mode in `/etc/kafka/connect-standalone.properties`
  - <http://docs.confluent.io/current/connect/>
- Important configuration options common to all workers:

Parameter	Description
<code>bootstrap.servers</code>	A list of host/port pairs to use to establish the initial connection to the Kafka cluster
<code>key.converter</code>	Converter class for the key
<code>value.converter</code>	Converter class for the value

Because Connect Workers are based on the Client API, some of the same settings (e.g., `bootstrap.servers`) will be used.

Connect Converters are like wrappers around Serializers and Deserializers. Since a Worker can be both a Producer and a Consumer (depending on the Connector), it was decided that having one object to specify rather than two a more efficient way to configure the Worker.



See <https://docs.confluent.io/current/connect/references/allconfigs.html> for a comprehensive list of configuration settings.

## Configuring Workers: Standalone Mode

Parameter	Description
<code>offset.storage.file.filename</code>	The filename in which to store offset data for the Connectors (Default: ""). This enables a standalone process to be stopped and then resume where it left off.

The provided file `/etc/kafka/connect-standalone.properties` sets this parameter to `/tmp/connect.offsets` which is not a very durable location on disk

ybhandare@greendotcorp.com

## Configuring Workers: Distributed Mode

Parameter	Description
<code>group.id</code>	A unique string that identifies the Kafka Connect cluster group the worker belongs to
<code>session.timeout.ms</code>	Timeout used to detect failures when using Kafka's group management facilities
<code>heartbeat.interval.ms</code>	Expected time between heartbeats to the group coordinator when using Kafka's group management facilities. Must be smaller than <code>session.timeout.ms</code>
<code>config.storage.topic</code>	Topic in which to store Connector & task config. data
<code>offset.storage.topic</code>	Topic in which to store offset data for Connectors
<code>status.storage.topic</code>	Topic in which to store connector & task status

As with Consumer Groups, the `group.id` will determine which Workers will cooperate as a team.

Connect Worker Groups will use unique administrative topics to hold connector configurations, offsets for both source and sinks, and status (if necessary) of the external data sources/sinks. Different worker groups can use different administrative topics if you need to keep their configurations separate.



Make sure that all Workers with the same `group.id` are using the same names for the administrative topics or you will get unpredictable results.

## Creating Kafka Connect's Topics Manually

- The Topics used in Kafka Connect distributed mode will be **automatically created**
- They are created with **recommended values** for
  - replication factor
  - partition counts
  - cleanup policy

The following number of Partitions:

Topic	Partitions
config.storage.topic	1
offset.storage.topic	25
status.storage.topic	5

Notice that the recommendations for `offset.storage.topic` are similar to the `__consumer_offsets` Topic since they do similar jobs.



In general, it's usually ok to let Connect create these topics for us. Connect will create these topics with the recommended replication factor, partition counts, and cleanup policy. The only time you might want to create these topics manually would be when you need custom configuration, or Connect doesn't have appropriate ACL privileges to create them. See <https://docs.confluent.io/current/connect/userguide.html#distributed-worker-configuration>

## Enabling Security with Connect

- If you have security enabled in the Kafka cluster (e.g. SSL or SASL), you need to enable it in Connect as well
  - Configure security on a per-worker basis
  - Override the default Producer or Consumer configurations that the worker uses
  - Configurations apply to *all* connectors running on the worker

You need the configuration at the worker top level because Connect leverages Kafka's group membership protocol to coordinate the workers and distribute work between them (as it does for Consumer Groups). The configuration at the producer/consumer level is required to interact with the Kafka cluster as a client.

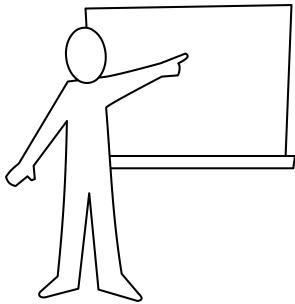
If using SASL for authentication, you must use the same principal for workers and connectors as only a single JAAS is currently supported on the client side.

Securing passwords used for individual connectors is handled differently and will be discussed later in the module.

ybhandare@greendotcorp.com

# Module Map

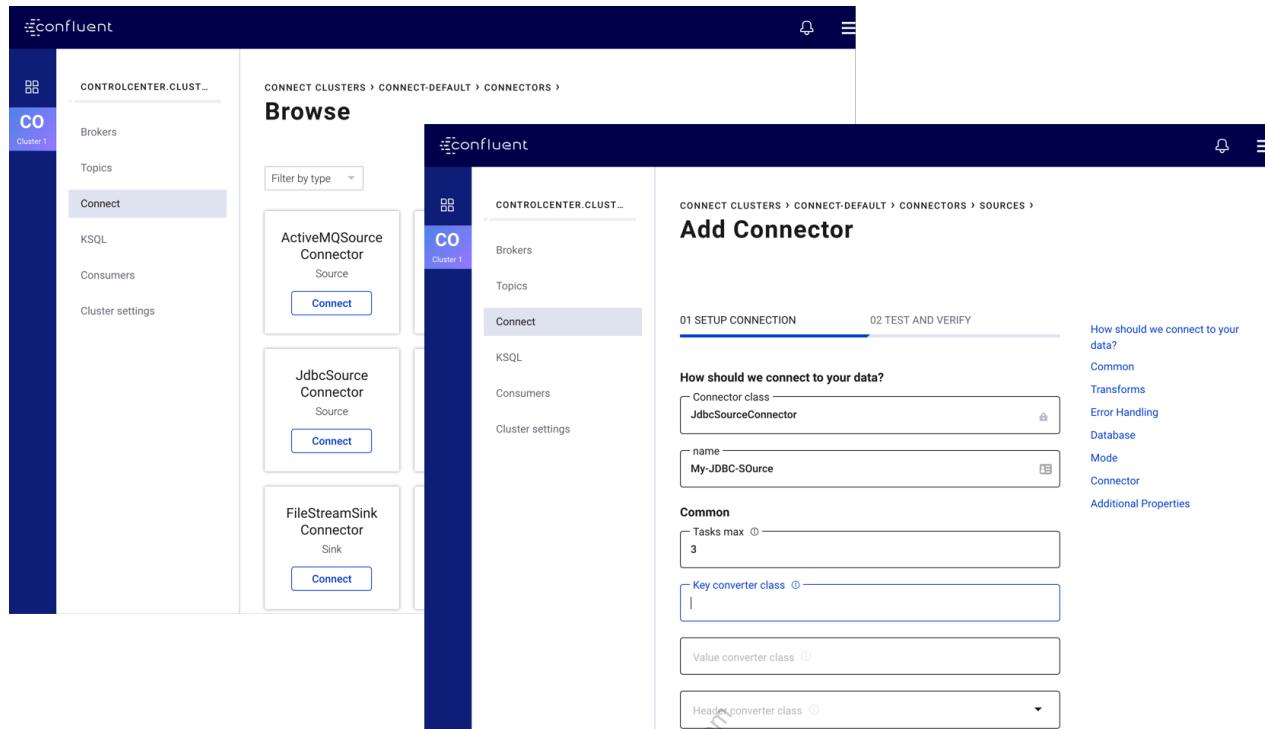
---



- The Motivation for Kafka Connect
- Types of Connectors
- Kafka Connect Implementation
- Standalone and Distributed Modes
- Configuring the Connectors ... 
-  Hands-on Lab

ybhandare@greendotcorp.com

# Configuring Connectors with Confluent Control Center



The image displays two screenshots of the Confluent Control Center interface. The left screenshot shows the 'Browse' page, which lists various connectors. The right screenshot shows the 'Add Connector' page for a JdbcSource Connector, with fields for configuration. A sidebar on the right lists connector types: Common, Transforms, Error Handling, Database, Mode, Connector, and Additional Properties.

Connectors can also be configured in Confluent Control Center. We can define source and sink connectors there.



Not all connectors may be fully configurable in the Control Center. In this case one has to use the Connect REST API to create and configure the connector. In the hands-on lab we give an example of this.

# Configuring Connectors with the REST API

- Add, modify delete connectors
- Distributed Mode:
  - Config **only** via REST API
  - Config stored in Kafka topic
  - REST call to **any** worker
- Standalone Mode:
  - Config also via REST API
  - Changes **not persisted!**
- Control Center uses REST API

```
1  {
2      "name": "Operators-Connector",
3      "config": {
4          "connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",
5          "connection.url": "jdbc:postgresql://postgres:5432/postgres",
6          "connection.user": "postgres",
7          "table.whitelist": "operators",
8          "mode": "incrementing",
9          "incrementing.column.name": "id",
10         "table.types": "TABLE",
11         "topic.prefix": "pg-",
12         "numeric.mapping": "best_fit",
13         "transforms": "createKey,extractInt",
14         "transforms.createKey.type": "org.apache.kafka.connect.transforms.ValueMapper$ValueToKey",
15         "transforms.createKey.fields": "id",
16         "transforms.extractInt.type": "org.apache.kafka.connect.transforms.ExtractValue$IntegerExtractor",
17         "transforms.extractInt.field": "id"
18     }
19 }
```

- Connectors can be added, modified, and deleted via a REST API on port 8083
- In distributed mode, configuration can be done only via this REST API
  - Changes made this way will persist after a worker process restart
  - Connector configuration data is stored in a special Kafka Topic
  - The REST requests can be made to any worker
- In standalone mode, configuration can also be done via a REST API
  - However, typically configuration is done via a properties file
    - Changes made via the REST API when running in standalone mode will not persist after worker restart
- Confluent Control Center leverages this REST API to let users configure and manage connectors through the GUI

To make changes to connectors, the Worker will also run a REST API as part of its code. This allows HTTP calls to be sent to any of the Workers that will then configure the connectors. Changes made via the REST API take effect immediately and without a reboot.



The REST API included with a Connect Worker is different from the Confluent REST Proxy.

# Using the REST API

Some important REST endpoints

Method	Path	Description
GET	/connectors	Get a list of active connectors
POST	/connectors	Create a new Connector
GET	/connectors/(string: name)/config	Get configuration information for a Connector
PUT	/connectors/(string: name)/config	Create a new Connector, or update the configuration of an existing Connector

More information on the REST API can be found at <https://docs.confluent.io/current/connect/references/restapi.html>

ybhandare@greendotcorp.com

## Configuring the Connector

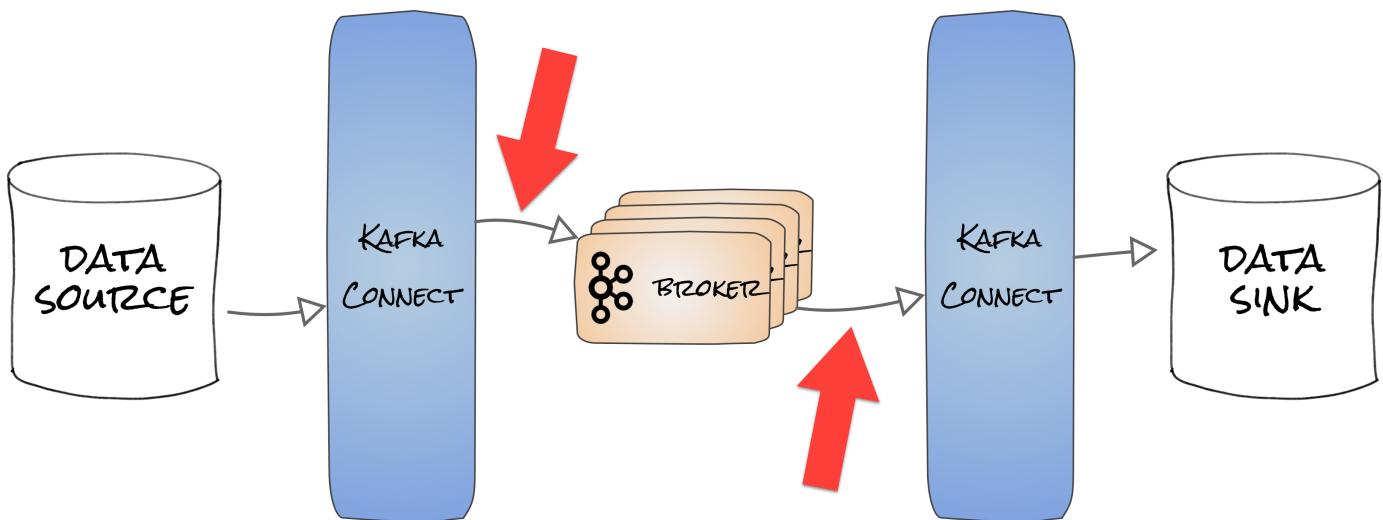
Parameter	Description
<code>name</code>	Connector's unique name
<code>connector.class</code>	Connector's Java class
<code>tasks.max</code>	Maximum tasks to create. The Connector may create fewer if it cannot achieve this level of parallelism (Default: 1)
<code>key.converter</code>	(optional) Override the worker key converter
<code>value.converter</code>	(optional) Override the worker value converter
<code>topics</code> (Sink connectors only)	List of input Topics (to consume from)

The number of tasks that a Connector should start will be dependent on the number of cores and workers available.

The reason that the `topics` parameter only exists for sink connectors is that source connectors typically have logic built in that will create new topic names based on a specific naming convention.

Key and value converters were added to the connector configurations in AK 0.10.1. Prior to that, there was a single converter setting that was specified in the worker configurations. Additional converters can be installed via Confluent Hub (when available).

## Transforming Data with Connect



Now, recall the architecture. Imagine we wanted to change messages in there.

There are at least two options discussed on the following slides...

ybhandare@greendotcorp.com

## Single Message Transform (SMT)

---

Modify events before storing in Kafka:

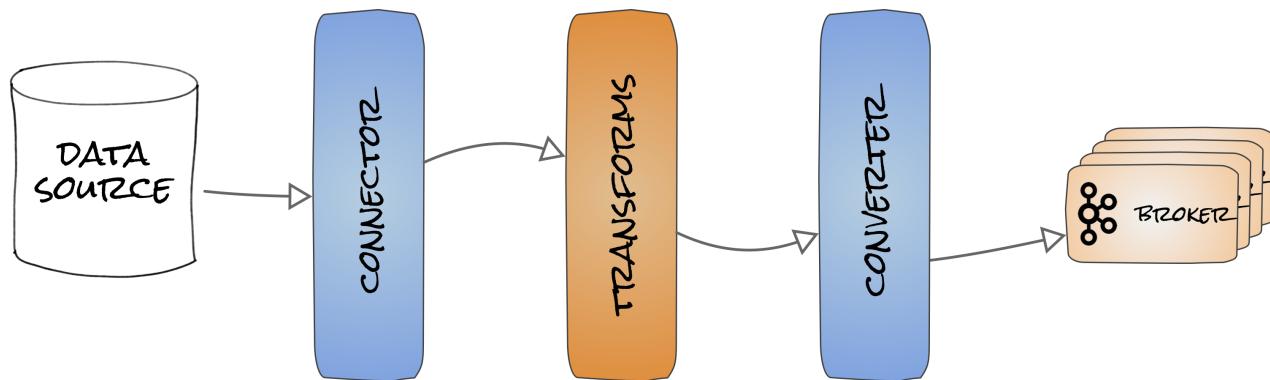
- Mask sensitive information
- Add identifiers
- Tag events
- Lineage/provenance
- Remove unnecessary columns

Modify events going out of Kafka:

- Route high priority events to faster data stores
- Direct events to different Elasticsearch indices
- Cast data types to match destination
- Remove unnecessary columns

ybhandare@greendotcorp.com

## Where SMTs Live



Single message transforms are a new stage in the Kafka Connect pipeline. Here we can see where they fit in for a source connector.

Source connectors read data from another system and store the data in Kafka. The connector (or more accurately, the connector's tasks) read data from the data source. At this stage the data will be in whatever format the source system exposes, but the connector will translate that into Kafka Connect's generic data API. Once it is in this generic format, the Kafka Connect single message transformations can be applied. Because they use the generic data format, any transformation can be used with any connector, independent of the original format of the data. The transformation is applied and the resulting data is still in a generic data API format. Finally, the data is passed to a converter, which serializes it into a format like Avro or JSON before it is passed to Kafka for storage. The pipeline for sink connectors looks similar, just reversed.

As you can see, by breaking the processing down into separate steps and leveraging Connect's data API, we can make transformations very general and applicable regardless of which connector you're using, much like we make serialization formats reusable across all connectors.

## Single Message Transform - Details

InsertField	insert a field using attributes from the message metadata or from a configured static value
ReplaceField	rename fields, or apply a blacklist or whitelist to filter
MaskField	replace field with valid <code>null</code> value for the type (0, empty string, etc)
ValueToKey	replace the key with a new key formed from a subset of fields in the value payload
HoistField	wrap the entire event as a single field inside a <code>Struct</code> or a <code>Map</code>
ExtractField	extract a specific field from <code>Struct</code> and <code>Map</code> and include only this field in results
SetSchemaMetadata	modify the schema name or version
TimestampRouter	modify the Topic of a record based on the original Topic name and timestamp
RegexRouter	modify the topic of a record based on original topic, replacement string and a regular expression

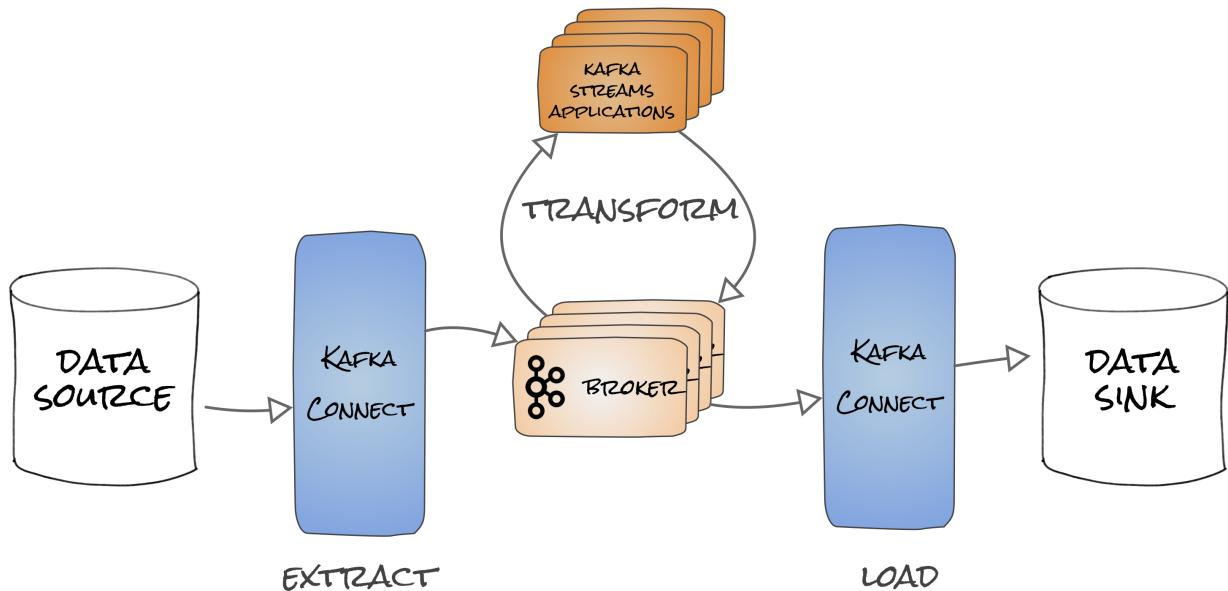
Many customers need a way to perform simple filtering and transformations for which building a streams application is too much work. For example, a company might want to take order information and send that to a team for analysis but needed to redact sensitive information such as social security numbers or credit card numbers.

The above properties can be configured at the **connector level** and be applied to either **key** and/or **value**.

TimestampRouter - Modify the topic of a record based on original topic and timestamp. Useful when using a sink that needs to write to different tables or indexes based on timestamps

More info: [http://kafka.apache.org/documentation.html#connect\\_transforms](http://kafka.apache.org/documentation.html#connect_transforms)

## Beware of SMTs!



We've described SMTs and explained why we constrained them the way we did. Why then, should you be wary of using SMTs too heavily? Why are we suggesting avoiding them for anything but the lightest weight transformations? Why would we add a feature to Kafka Connect and then warn people **not** to use it?

Three reasons: Levels of abstraction, Order of operations and readability, Schemas & compatibility

The first is really the most important and has a lot of impact. The latter two could be considered "usability warts", but can also be significant pitfalls.

Many of us are trying to get to **streaming ETL**. But be careful! SMTs are not what streaming ETL is for. Anything complex should be a **Streams application**.

# Programming with Configuration

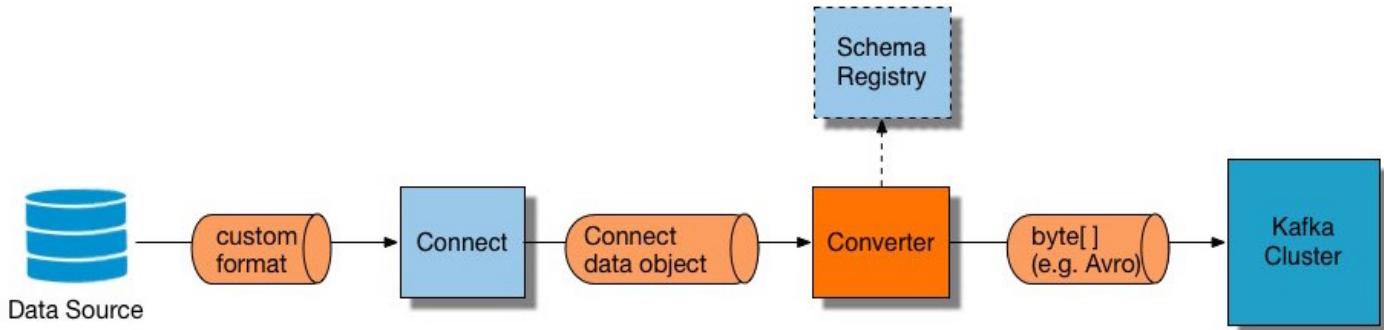
```
name=local-file-source
connector.class=FileStreamSource
tasks.max=1
file=test.txt
topic=connect-test
transforms=MakeMap,InsertSource, InsertKey, ExtractStoreId, MessageTypeRouter
transforms.MakeMap.type=org.apache.kafka.connect.transforms.HoistField$Value
transforms.MakeMap.field=line
transforms.InsertSource.type=org.apache.kafka.connect.transforms.InsertField$Value
transforms.InsertSource.static.field=data_source
transforms.InsertSource.static.value=test-file-source
transforms.InsertKey.type=org.apache.kafka.connect.transforms.ValueToKey
transforms.InsertKey.fields=storeId
transforms.ExtractStoreId.type=org.apache.kafka.connect.transforms.ExtractField$Key
transforms.ExtractStoreId.field=storeId
transforms.MessageTypeRouter.type=org.apache.kafka.connect.transforms.RegexRouter
transforms.MessageTypeRouter.regex=(foo|bar|baz)-.*
transforms.MessageTypeRouter.replacement=$1-logs
```

This starts to look kind of unpleasant!

1. Hoist Field: message sent **Foo** → transformed value `{"line": "Foo"}`
2. Insert Source: Add field **data\_source** with value **test-file-source** to the imported records
3. Insert Key & ExtractStoreId: Take the value of the field **storeId** and use this as value for the Kafka record **key**
4. MessageTypeRouter: Update the record's topic using the configured regular expression and replacement string

## Converting Data

- Converters provide the data format written to or read from Kafka (like Serializers)
- Converters are decoupled from Connectors to enable reuse of converters
  - Allows any connector to work with any serialization format
- Example of format conversion in a source converter (sink converter is the reverse)



Serialization formats may seem like a minor detail, but **not** separating the details of data serialization in Kafka from the details of source or sink systems would result in a lot of inefficiency:

- First, a lot of code for doing simple data conversions are duplicated across a large number of ad hoc connector implementations.
- Second, each connector ultimately contains its own set of serialization options as it is used in more environments—JSON, Avro, Thrift, ProtoBuf, and more.

Much like the serializers in Kafka's producer and consumer, the **Converters** abstract away the details of serialization. Converters are different because they guarantee data is transformed to a common data API defined by Kafka Connect. This API supports both schema and schema-less data, common primitive data types, complex types like structs, and logical type extensions. By sharing this API, connectors write one set of translation code and Converters handle format-specific details. For example, the JDBC connector can easily be used to produce either JSON or Avro to Kafka, without any format-specific code in the connector.

## Converter Data Formats

---

- Converters apply to both the key and value of the message
  - Key and value converters can be set independently
    - `key.converter`
    - `value.converter`
- Pre-defined data formats for Converter
  - Avro: `AvroConverter`
  - JSON: `JsonConverter`
  - String: `StringConverter`
  - Byte Array: `ByteArrayConverter`

The `ByteArrayConverter` (detailed in KIP-128, introduced in Kafka 0.11) provides an option for dealing with raw data, in contrast to structured data. It saves cost (CPU, memory, garbage collection overhead) of deserializing/convert data to Connect format.

For example, this is particularly useful with **Replicator**. This connector doesn't need to convert to/from byte array format and objects - it just copies the raw data as bytes for better performance.

ybhandare@greendotcorp.com

## Avro Converter as a Best Practice

---

### Use an Avro Converter and Schema Registry!

```
key.converter=io.confluent.connect.avro.AvroConverter
key.converter.schema.registry.url=http://schemaregistry1:8081
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://schemaregistry1:8081
```

ybhandare@greendotcorp.com

## JDBC Source Connector: Overview

- JDBC Source periodically polls a relational database for new or recently modified rows
  - Creates a record for each row, and Produces that record as a Kafka message
- Records from each table are Produced to their own Kafka Topic
- New and deleted tables are handled automatically

The JDBC source connector allows you to import data from any relational database with a JDBC driver into Kafka topics. By using JDBC, this connector can support a wide variety of databases without requiring custom code for each one.

Data is loaded by periodically executing a SQL query and creating an output record for each row in the result set. By default, all tables in a database are copied, each to its own output topic. The database is monitored for new or deleted tables and adapts automatically. When copying data from a table, the connector can load only new or modified rows by specifying which columns should be used to detect new or modified data.



Here is a good place to discuss CDC versus query based source connectors (Pros and Cons of each)!

## Detecting New and Updated Rows

Incremental query mode	Description
Incrementing column	Check a single column where newer rows have a larger, auto-incremented ID. Does not support updated rows
Timestamp column	Checks a single 'last modified' column. Can't guarantee reading all updates
Timestamp and incrementing column	Combination of the two methods above. Guarantees that all updates are read
Custom query	Used in conjunction with the options above for custom filtering



**bulk** mode for one-time load, not incremental, unfiltered

The Connector can detect new and updated rows in several ways as described on the slide.

For the reasons stated on the slides, many environments will use both the timestamp and the incrementing column to capture all updates.

Note: Because timestamps are not necessarily unique, the timestamp column mode cannot guarantee all updated data will be delivered. If two rows share the same timestamp and are returned by an incremental query, but only one has been processed before a crash, the second update will be missed when the system recovers.

The custom query option can only be used in conjunction with one of the other incremental modes as long as the necessary WHERE clause can be appended to the query. In some cases, the custom query may handle all filtering itself.

## JDBC Source Connector: Configuration

Parameter	Description
<code>connection.url</code>	The JDBC connection URL for the database
<code>topic.prefix</code>	The prefix to prepend to table names to generate the Kafka Topic name
<code>mode</code>	The mode for detecting table changes. Options are <code>bulk</code> , <code>incrementing</code> , <code>timestamp</code> , <code>timestamp+incrementing</code>
<code>query</code>	The custom query to run, if specified
<code>poll.interval.ms</code>	The frequency in milliseconds to poll for new data in each table
<code>table.blacklist</code>	A list of tables to ignore and not import. If specified, <code>table.whitelist</code> cannot be specified
<code>table.whitelist</code>	A list of tables to import. If specified, <code>table.blacklist</code> cannot be specified



See <https://docs.confluent.io/current/connect/connect-jdbc/docs/index.html> for a complete list

Setting both `table.whitelist` and `table.blacklist` does not fail any upfront configuration validation checks but will fail when starting the connector at runtime.

## HDFS Sink Connector: Overview

- Continuously polls from Kafka and writes to HDFS (Hadoop Distributed File System)
- Integrates with Hive
  - Auto table creation
  - Schema evolution with Avro
- Works with secure HDFS and the Hive Metastore, using Kerberos
- Provides exactly once delivery
- Data format is extensible
  - Avro, Parquet, custom formats
- Pluggable Partitioner, supporting:
  - Kafka Partitioner (default)
  - Field Partitioner
  - Time Partitioner
  - Custom Partitioners

The HDFS connector allows you to export data from Kafka topics to HDFS files in a variety of formats and integrates with Hive to make data immediately available for querying with HiveQL.

The connector periodically polls data from Kafka and writes them to HDFS. The data from each Kafka topic is partitioned by the provided partitioner and divided into chunks. Each chunk of data is represented as an HDFS file with topic, kafka partition, start and end offsets of this data chunk in the filename. If no partitioner is specified in the configuration, the default partitioner which preserves the Kafka partitioning is used. The size of each data chunk is determined by the number of records written to HDFS, the time written to HDFS and schema compatibility.

The HDFS connector integrates with Hive and when it is enabled, the connector automatically creates an external Hive partitioned table for each Kafka topic and updates the table according to the available data in HDFS.

## FileStream Connector

- FileStream Connector acts on a local file
  - Local file Source Connector: tails local file and sends each line as a Kafka message
  - Local file Sink Connector: Appends Kafka messages to a local file

The FileSource Connector reads data from a file and sends it to Kafka. This connector will read only one file and send the data within that file to Kafka. It will then watch the file for appended updates only. Any modification of file lines already sent to Kafka will not be reprocessed.

The FileSink Connector reads data from Kafka and outputs it to a local file. Multiple topics may be specified as with any other sink connector. As messages are added to the topics specified in the configuration, they are produced to a local file as specified in the configuration.

The FileStream Connector examples on [docs.confluent.io](https://docs.confluent.io) are intended to show how a simple connector runs for those first getting started with Kafka Connect as either a user or developer. It is not recommended for production use. Instead, we encourage users to use them to learn in a local environment. The examples include both a file source and a file sink to demonstrate an end-to-end data flow implemented through Kafka Connect. The FileStream Connector examples are also detailed in the developer guide as a demonstration of how a custom connector can be implemented.

### Questions:



- Why would you use Kafka Connect?
- What kind of supported connectors are you aware of?
- What about if no connector exists for your source or sink?

- Kafka Connect provides a scalable, reliable way to transfer data from external systems into Kafka, and vice versa
- Many off-the-shelf Connectors are provided by Confluent, and many others are under development by third parties
- To create a new connector is well documented and straight forward. Write a Java application that implements pre-defined interfaces and register the resulting JAR with Kafka Connect

ybhandare@grendel:~/tmp\$

## Hands-On Lab

---

Please refer to the following list of labs in the exercise book:

- **Using Kafka Connect**
- **Using the Syslog Connector**
- **Using Kafka Connect with MQTT**
- **OPTIONAL: Using the Confluent MQTT Proxy**



ybhandare@greendotcorp.com



### Stream Processing with Kafka Streams

ybhandare@greendotcorp.com

# Agenda

---



1. Introduction
2. Fundamentals of Apache Kafka
3. Kafka's Architecture
4. Developing With Kafka
5. More Advanced Kafka Development
6. Schema Management In Kafka
7. Data Pipelines with Kafka Connect
8. Stream Processing with Kafka Streams ... 
9. Stream Processing with KSQL
10. Event Driven Architecture
11. Confluent Cloud
12. Conclusion
13. Appendix: Basic Kafka Administration
14. Appendix: Apache Kafka driving ML & Data Analytics

ybhandare@greendotcorp.com

## Learning Objectives

---

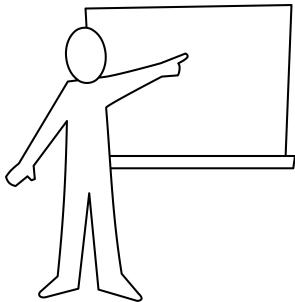


After this module you will be able to:

- Explain the motivation behind the Kafka Streams API
- List the features provided by the Streams API
- Write an application using the Streams DSL (Domain-Specific Language)

ybhandare@greendotcorp.com

# Module Map



- An Introduction to the Kafka Streams API ...
- Kafka Streams Concepts
- Creating a Kafka Streams Application
- Kafka Streams by Example
- Managing Kafka Stream Processing
-  Hands-on Lab



ybhandare@greendotcorp.com

# Kafka Streams

Write standard Java apps and microservices to process your data in real-time...

- No separate processing cluster required
- Develop on Mac, Linux, Windows
- Deploy to containers, VMs, bare metal, cloud
- Powered by Kafka: elastic, scalable, distributed, battle-tested
- Perfect for small, medium, large use cases
- Fully integrated with Kafka security
- Exactly-once processing semantics
- Part of Apache Kafka®

```
KStream<User, PageViewEvent> pageViews = builder.stream("pageviews-topic");
KTable<Windowed<User>, Long>> viewsPerUserSession = pageViews
    .groupByKey()
    .windowedBy(TimeWindows.of(Duration.ofMinutes(5)))
    .count(...)
```

The code snippet does the following:

- use the `builder` object to create a `KStream` from the topic `pageviews-topic`
- Group the `KStream` by its key (as defined in the record key)
- Define time windows of 5 minutes (starting from the Linux epoch)
- Count the records per key and per time window

# What Is the Kafka Streams API?

- **Transforms** and **enriches** data
  - per-record stream processing
  - millisecond latency
  - stateless & stateful processing
  - windowing operations
- **Fault-tolerant** and **distributed processing**
- Has **Domain-Specific Language (DSL)**
  - High level operations: `map`, `flatMap`, `count`, etc.
- **Processor API** for even more flexibility

This module is intended as a very basic overview of Kafka Streams and KSQL due to time constraints. More advanced topics (e.g., Global KTables, User Defined Functions (UDFs), STRUCT data type) are beyond the scope of this course and are discussed in the *Stream Processing with Kafka Streams and KSQL* course.

Additional information:



- In Kafka Streams there's also a Processor API in addition to the DSL providing even more flexibility.
- Windowing operations are not the equivalent of micro batching found in other streaming frameworks
  - The term “micro-batch” is frequently used to describe scenarios where batches are small and/or processed at small intervals. Even though processing may happen as often as once every few minutes, data is still processed a batch at a time. Spark Streaming is an example of a system that supports micro-batch processing.
  - Stream processing with windowing in turn does not wait until the end of a time window until results are published. Results are continuously updated (e.g. aggregated) and published for downstream processing.

# A Library, Not a Framework

- Kafka Streams provides **streaming capabilities**
- Other streaming frameworks:
  - Apache Flink
  - Spark Streaming
  - Apache Storm
  - Apache Samza
  - etc.
- **BUT:** Kafka Streams does not require its own cluster
  - It's just a **Java library**
  - Runs on **1...n** machines
- Using Kafka Streams may reduce load on remote DBMS
  - Local data accesses via the local persistent datastore



What is meant with "Kafka Streams does not require its own cluster"? It is clear that we always need a cluster of brokers, no matter whether we use Kafka Streams or say Flink for stream processing. The meaning is that Flink needs its own "Flink Cluster infrastructure" whilst a Kafka Streams application is just a Java application and thus can run without extra clustering infrastructure. Of course one can use e.g. Kubernetes to run the streams app at scale.

Regarding the reduction on load on a remote DBMS:

- **Old way:** processor looks up reference data in a remote database management system (e.g. Cassandra) or even uses that remote system to merge joined results.
- **New way:** Migrate some of the processing jobs to Kafka Streams API to reduce the load on the remote database management system by turning remote data accesses into cheaper local data accesses.

## Why Not Just Build Your Own?

---

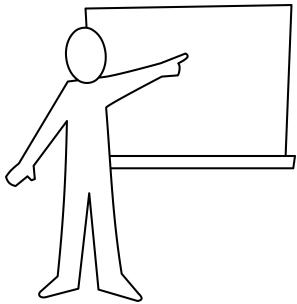
- Many people are currently building their own stream processing applications
  - Using the Producer and Consumer APIs
- Using Kafka Streams API is much easier than taking the 'do it yourself' approach
  - Well-designed, well-tested, robust
  - Means you can focus on the application logic, not the low-level plumbing

Kafka Streams is part of the open source Apache Kafka distribution and so is available for free as part of that package.

ybhandare@greendotcorp.com

# Module Map

---

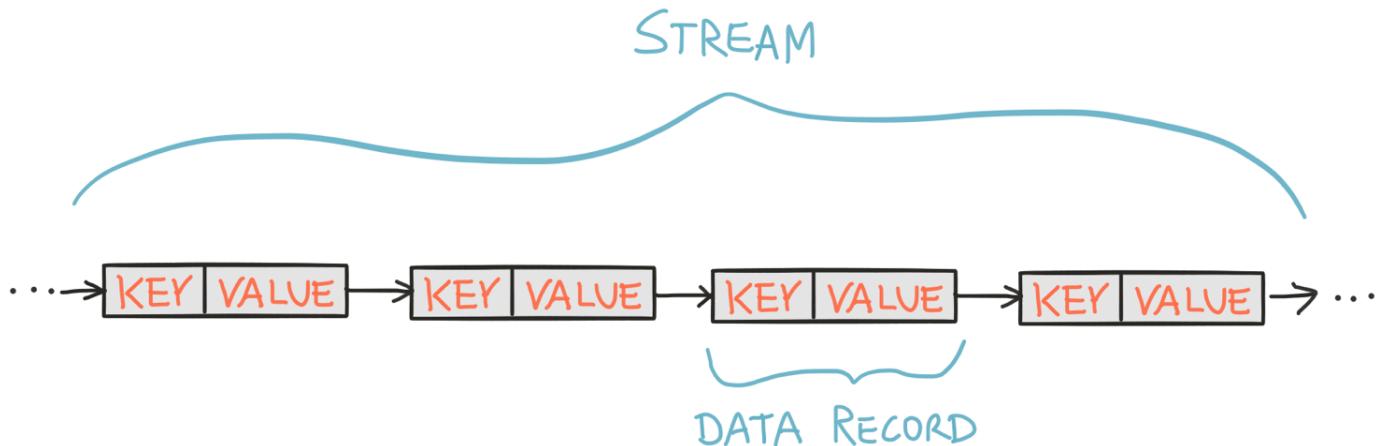


- An Introduction to the Kafka Streams API
- Kafka Streams Concepts ... ←
- Creating a Kafka Streams Application
- Kafka Streams by Example
- Managing Kafka Stream Processing
-  Hands-on Lab

ybhandare@greendotcorp.com

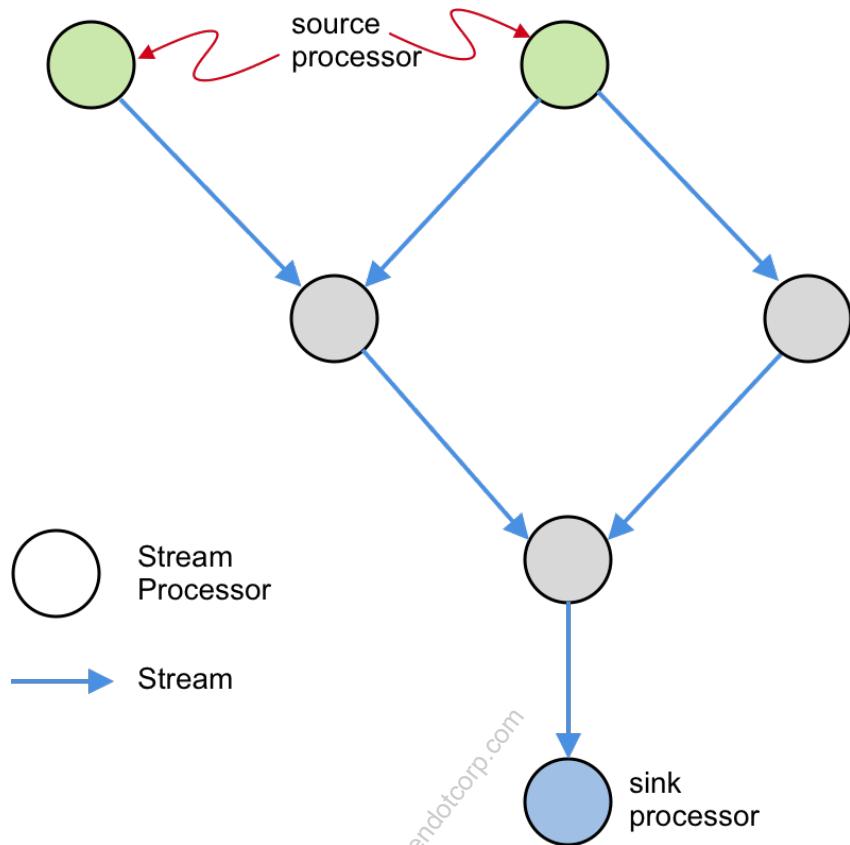
## What is a Stream?

- Think of a stream as an unbounded, continuous real-time flow of records
  - You don't need to explicitly request new records, you just receive them
- Records are key-value pairs



ybhandare@greendotcorp.com

## Stream Processor and Topology



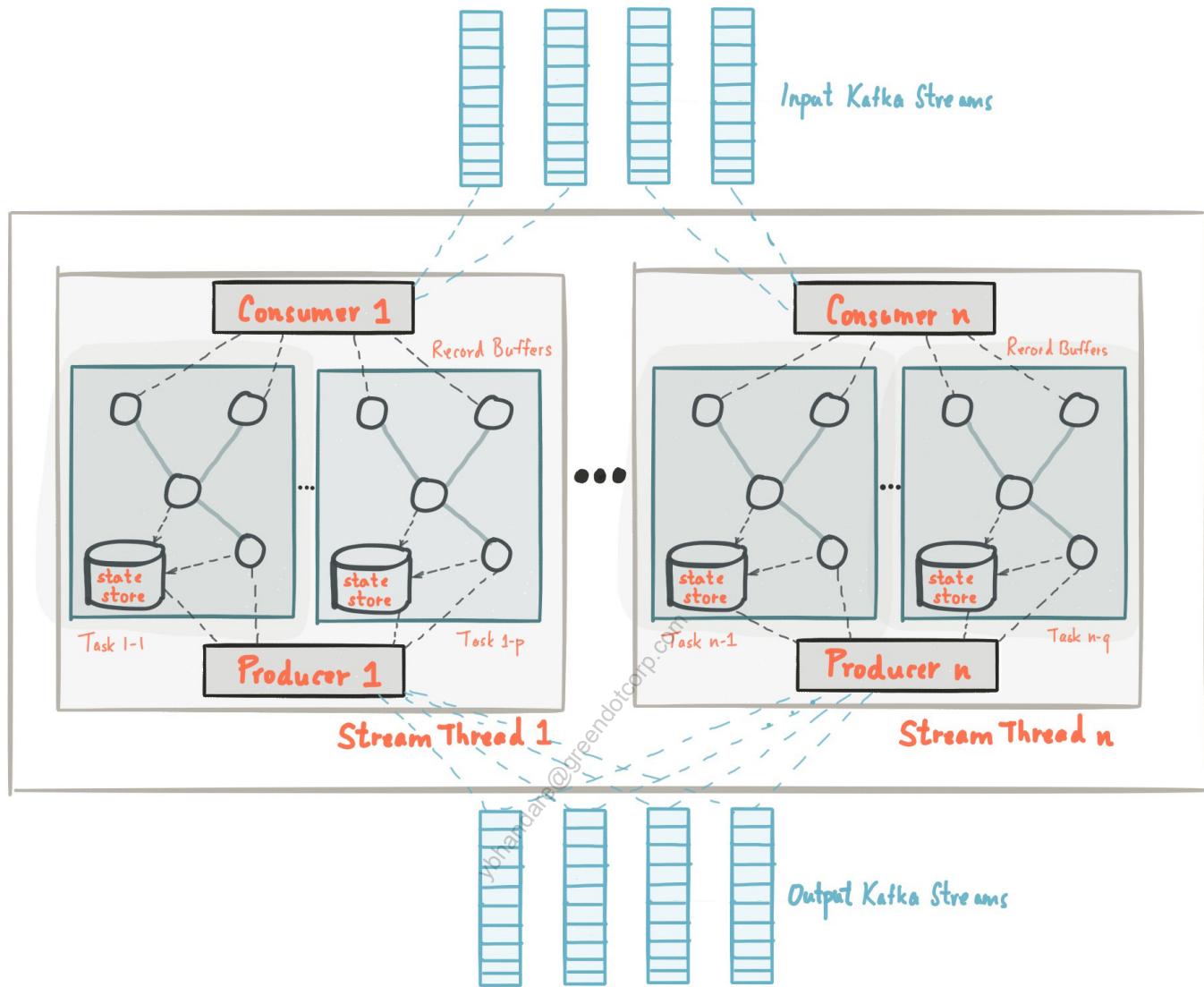
- A *stream processor* transforms data in a stream
- A *processor topology* defines the data flow through the stream processors

A common misconception with the processor topology is that the "stream processors" are different systems. This is incorrect!

### Correct definition:

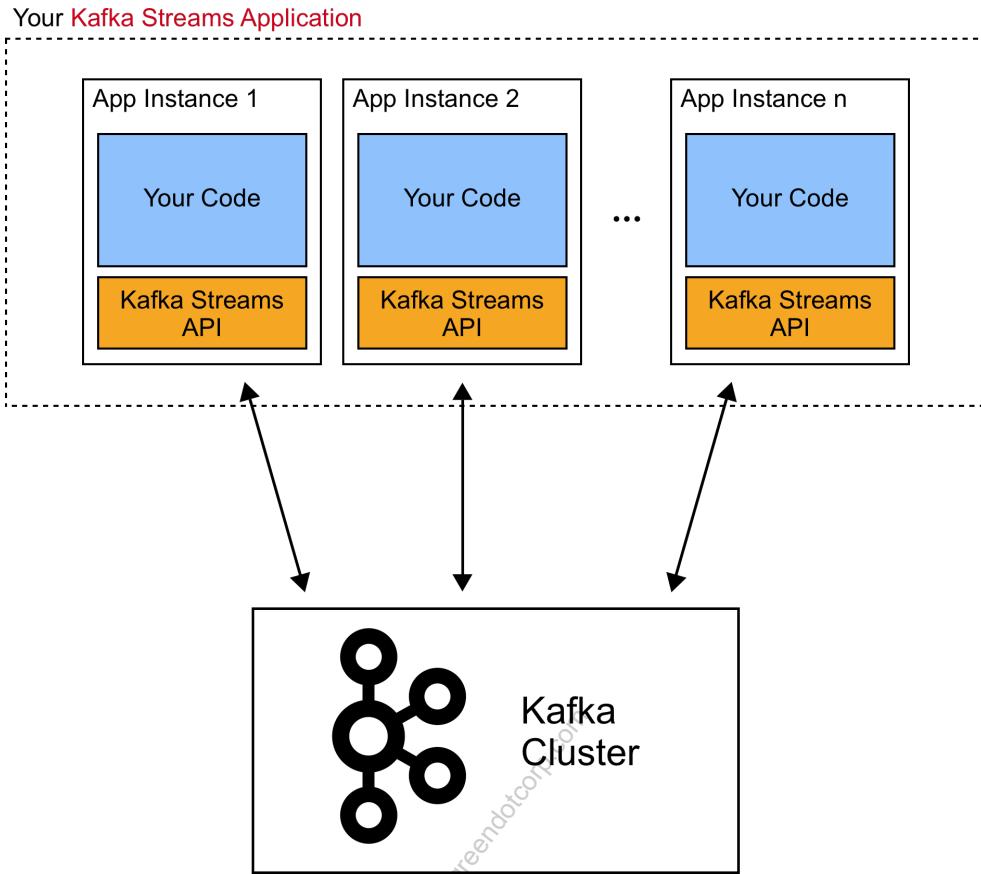
*A stream processor is a node in the processor topology that represents a single processing step. With the **Processor API**, you can define arbitrary stream processors that processes one received record at a time, and connect these processors with their associated state stores to compose the processor topology.*

## Logical View of Kafka Streams Application



This diagram is a logical view of a Kafka Streams application that contains multiple stream threads, each of which in turn containing multiple stream tasks. Stream threads can run in parallel on a single system or distributed across multiple systems for performance and availability. Stateful information is stored locally in a RocksDB temporarily before being persisted to a special Kafka topic.

# Kafka Streams Application Architecture

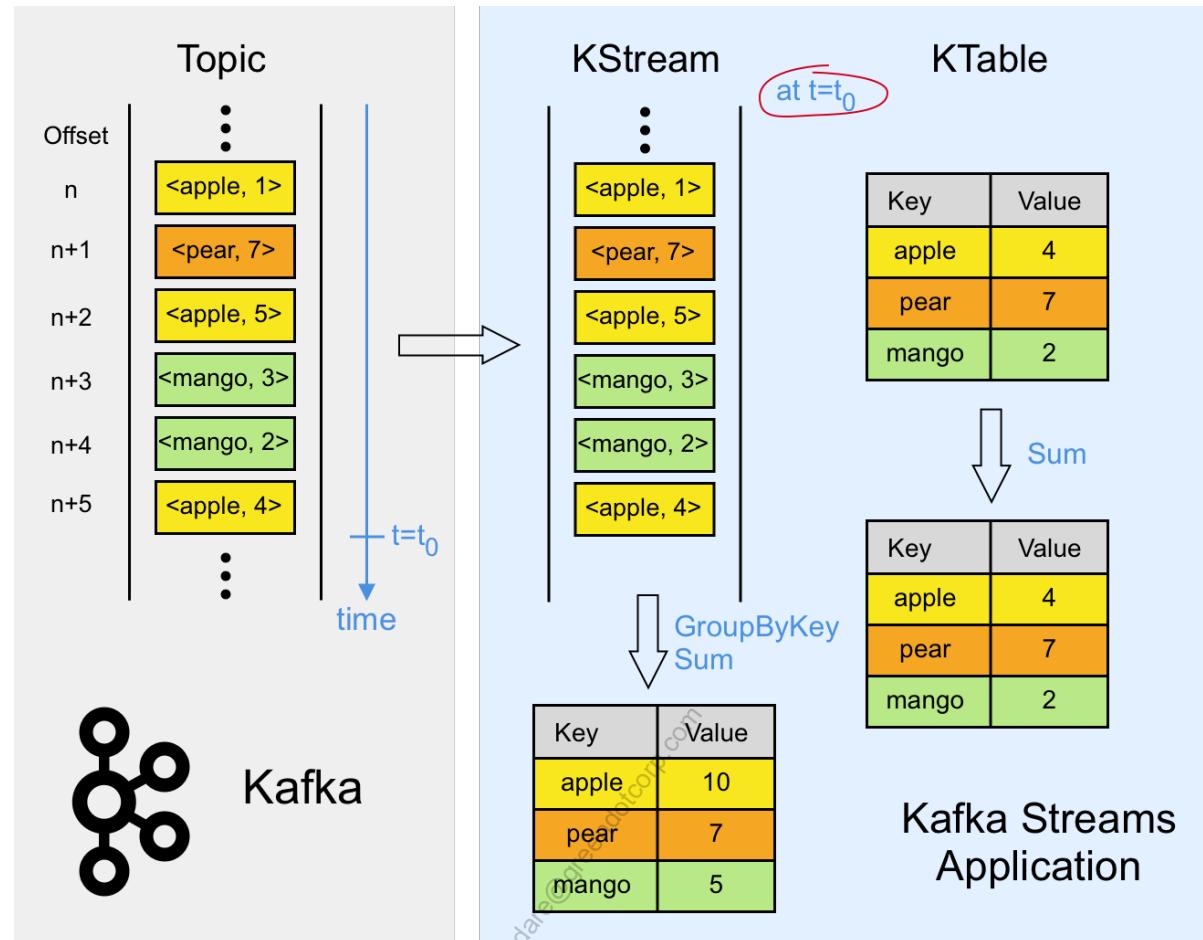


- The Streams API of Apache Kafka, available through a Java library, can be used to build highly scalable, elastic, fault-tolerant, distributed applications and microservices.
- A unique feature of the Kafka Streams API is that the applications we build with the Kafka Streams library are normal Java applications. These applications can be packaged, deployed, and monitored like any other Java application – there is no need to install separate processing clusters or similar special-purpose and expensive infrastructure!

- We can run more than one instance of the application. The instances run independently but will automatically discover each other and collaborate.
- We can elastically add or remove instances during live operation.
- If one instance (unexpectedly dies) the others will automatically take over its work and continue where it left off.

ybhandare@greendotcorp.com

## KStreams and KTables



- A **KStream** is an abstraction of a record stream
  - Each record represents a self-contained piece of data in the unbounded data set
- A **KTable** is an abstraction of a changelog stream
  - Each record represents an update
- Example: We send two records to the stream
  - ('apple', 1), and ('apple', 5)
- If we were to treat the stream as a **KStream** and sum up the values for **apple**, the result would be 6
- If we were to treat the stream as a **KTable** and sum up the values for **apple**, the result would be 5
  - The second record is treated as an update to the first, because they have the same key



It is clear that summing a table (as indicated at the center right of the slide) is not possible. It is only a conceptual comparison between a **KStream** and a **KTable**.

## KStreams and KTables



Typically, if you are going to treat a Topic as a **KTable** it makes sense to configure log compaction on the Topic



You might realize that you cannot really "Sum" a KTable as indicated on the slide. You would have to first group it by (another) key. But the slide is used merely to illustrate the difference between a **KStream** and a **KTable**...

ybhandare@greendotcorp.com

## Windowing, Joining, and Aggregations

- Kafka Streams API allows us to *window* the stream of data by time
  - To divide it up into 'time buckets'
- We can aggregate records
  - Combine multiple input records together in some way into a single output record
  - Examples: sum, count
  - This is usually done on a windowed basis
- We can join, or *i.e.*, merge, data from different sources

Note: the stateful transformations are beyond the scope of this introduction to Kafka Streams. Refer the students to the [docs.confluent.io](https://docs.confluent.io) for more additional information.

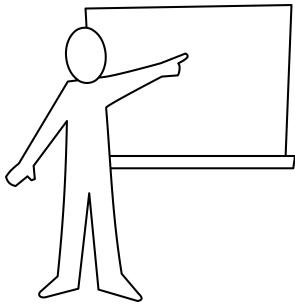


Usually it makes sense to use **compacted topics** as the backing "data store" of a **KTable**

ybhandare@greendotcorp.com

# Module Map

---



- An Introduction to the Kafka Streams API
- Kafka Streams Concepts
- Creating a Kafka Streams Application ... ↵
- Kafka Streams by Example
- Managing Kafka Stream Processing
-  Hands-on Lab

ybhandare@greendotcorp.com

# Configuring a Kafka Streams Application

- Kafka Streams configuration is specified with a `StreamsConfig` instance
  - This is typically created from a `java.util.Properties` instance
- Specify configuration parameters
  - `APPLICATION_ID_CONFIG`: all app instances use the same ID
  - `BOOTSTRAP_SERVERS_CONFIG`: where to find Kafka broker(s)
- Example:

```
Properties config = new Properties();
config.put(StreamsConfig.APPLICATION_ID_CONFIG, "simple-example");
config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
config.put(..., ...); ①
```

- ① Continue to specify configuration options

- `APPLICATION_ID_CONFIG`: Each stream processing application must have a unique ID. The same ID must be given to all instances of the application.



since Streams reads in from a topic and outputs to a topic, it will require Consumer and Producer configurations in addition to the Streams configurations.

## Serializers and Deserializers (SerDes)

- Use Serializers and Deserializers (SerDes) to convert bytes of the record to a specific type
  - SERializer
  - DESerializer
- Key SerDes can be independent from value SerDes
- There are many many built-in SerDes (e.g. `Serdes.String`, etc.)
- You can also define your own
- Configuration example:

```
config.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());  
config.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.Long().getClass());
```

Because Streams applications behave as both Producers and Consumers, a combination object called a SerDe is used instead of serializers and deserializers.

The SerDes defined in the Properties object are typically typed to match either the inbound or outbound messages. If the data types change due to the transforms, the SerDes can be overridden in the application code (examples will be shown later in the chapter).



Usually it is not recommended to define default, application-scoped SerDes, unless you're sure they're the only SerDes needed for the whole application!

## Available SerDes

- Kafka includes a variety of SerDes in the `kafka-clients` Maven artifact

Data type	SerDes
byte[]	<code>Serdes.ByteArray()</code> , <code>Serdes.Bytes()</code> ( <code>Bytes</code> wraps Java's <code>byte[]</code> and supports equality and ordering semantics)
ByteBuffer	<code>Serdes.ByteBuffer()</code>
Double	<code>Serdes.Double()</code>
Integer	<code>Serdes.Integer()</code>
Long	<code>Serdes.Long()</code>
String	<code>Serdes.String()</code>

- If you are using Avro, you can use Avro SerDes provided by the `kafka-streams-avro-serde` Maven artifact

ybhandare@googlecorp.com

## Creating the Processing Topology

- Create a `KStream` or `KTable` object from one or more Kafka Topics, using `StreamsBuilder`
- Example:

```
StreamsBuilder builder = new StreamsBuilder();  
  
KStream<String, Long> purchases = builder.stream("PurchaseTopic");
```

The code to create a `KTable` is the same as shown for the `KStream` – just replace `KStream` with `KTable` and `builder.stream` with `builder.table`.

The `builder.stream()` starts the flow of data into the `KTable` or `KStream` by consuming messages from the specified topic. `KStreams` can specify multiple topics for `stream()`; `KTables` can only specify a single topic.

ybhandare@greendotcorp.com

## Transforming a Stream

---

- Data can be transformed using a number of different operators
- Some operations result in a new `KStream` object
  - For example, `filter` or `map`
- Some operations result in a `KTable` object
  - For example, an aggregation operation

ybhandare@greendotcorp.com

## Some Stateless Transformation Operations (1)

### filter

Creates a new **KStream** containing only records from the previous **KStream** which meet some specified criteria

```
var smallPurchases = purchases
    .filter((key,value)->value.amount<50.0)
```

### map

Creates a new **KStream** by transforming each element in the current stream into a different element in the new stream

```
var upper = words
    .map((key,value)->new KeyValue<>(key, value.toUpperCase()));
```

### mapValues

Creates a new **KStream** by transforming the value of each element in the current stream into a different element in the new stream

```
var upper = words
    .mapValues(value->value.toUpperCase());
```



**filter** is the equivalent of **where** and **map** the equivalent of **select** in a SQL statement

## Some Stateless Transformation Operations (2)

### flatMap

Creates a new `KStream` by transforming each element in the current stream into zero or more different elements in the new stream

### flatMapValues

Creates a new `KStream` by transforming the value of each element in the current stream into zero or more different elements in the new stream

```
var pattern = Pattern.compile("\\W+", ...);

var words = textLines
    .flatMapValues(v -> Arrays.asList(pattern.split(v)));
```



A word to the difference between `map` and `mapValues` and `flatMap` and `flatMapValues`: `map` and `flatMap` will always mark the output stream for repartitioning, even if you don't change the key. So don't use those variants **unless** you really want to change the key, because repartitioning is a relatively expensive operation.

## Some Stateful Transformation Operations

```
stream  
  .groupByKey()  
  .count()
```

Counts the number of instances of each key in the stream; results in a new, ever-updating **KTable**

```
stream  
  .groupByKey()  
  .reduce(...)
```

Combines values of the stream using a supplied Reducer into a new, ever-updating **KTable**

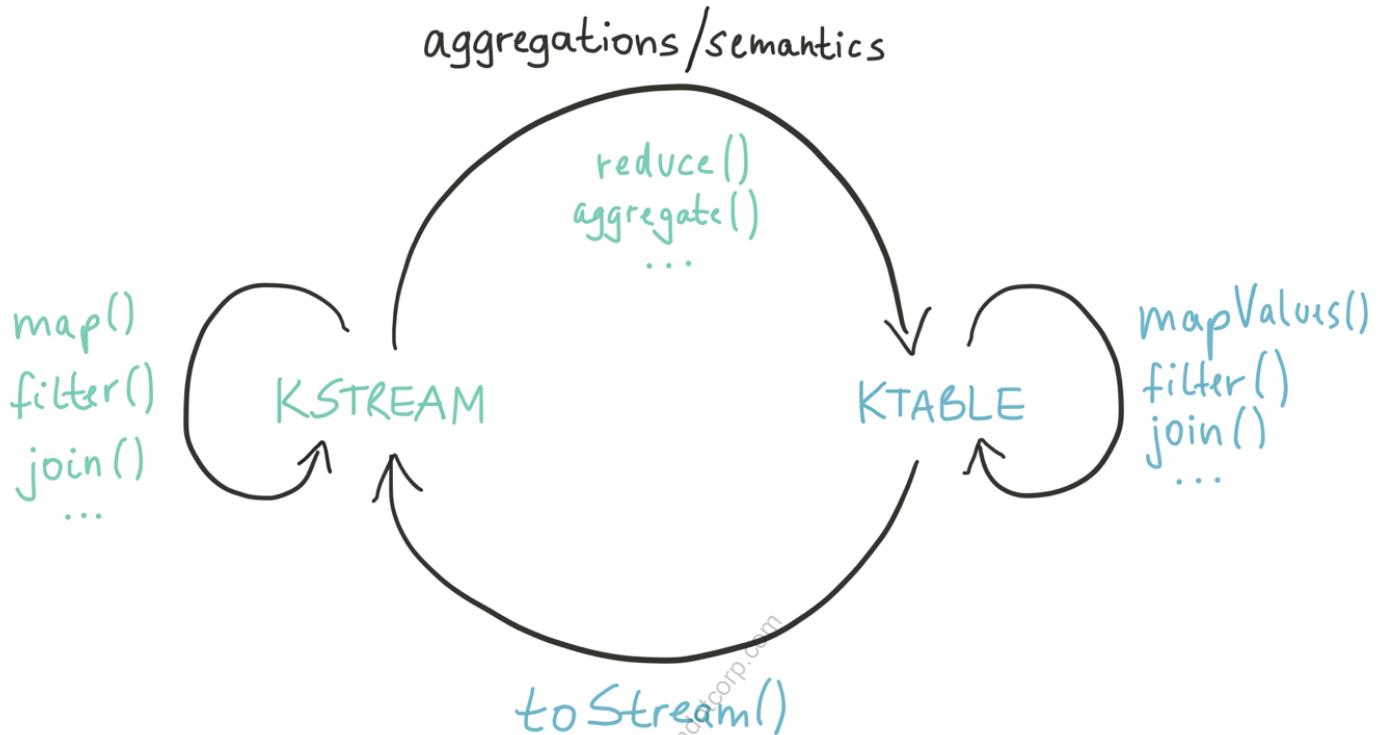
For a full list of operations, see:

<https://www.javadoc.io/doc/org.apache.kafka/kafka-streams/2.3.0>

ybhandare@greendotcorp.com

## Mixed Processing

- You can build stream processing topologies with mixed stateless and stateful processing



It is important to familiarize yourself with the output types of the various stream processors so that your code can appropriately handle the data.

- **join**: used to join two **KStream** objects, a **KStream** and a **KTable**, or two **KTable** objects. This is often used to enrich data coming from one stream with the data of another stream or table (such as a lookup table)
- **aggregate**, **reduce**: the two functions are similar in what they're doing. We can apply any aggregation function such as summing, averaging, counting, etc. to a group of records. The group is by key and time window (if time windows are used)

## Writing Streams Back to Kafka

- Streams can be written to Kafka Topics using the `to` method
  - Example:

```
myNewStream.to("NewTopic");
```

- We often want to write to a Topic but then continue to process the data
  - Do this using the `through` method

```
myNewStream.through("NewTopic").flatMap(...)...;
```

The typical output of a Stream Topology is to a topic.

If you want to have intermediate states to work with as well, the `through()` method outputs to a topic while continuing the topology.



`through` incurs a **round-trip** (produce/consume) to Kafka, and the data read back from Kafka feeds into the next operator in the topology.

## Printing A Stream's Contents

---

- It is sometimes useful to be able to see what a stream contains
  - Especially when testing and debugging
- Use `print()` to write the contents of the stream

```
myNewStream.print(Printed.toSysOut());
```

ybhandare@greendotcorp.com

## Running the Application

---

- To start processing the stream, create a `KafkaStreams` object
  - Configure it using `StreamsBuilder`
- Then call the `start()` method
- Example:

```
KafkaStreams streams = new KafkaStreams(builder.build(), config);
streams.start();
```

ybhandare@greendotcorp.com

## Application Graceful Shutdown

---

- Allow your application to gracefully shutdown
  - For example, in response to SIGTERM
- Add a shutdown hook to stop the application

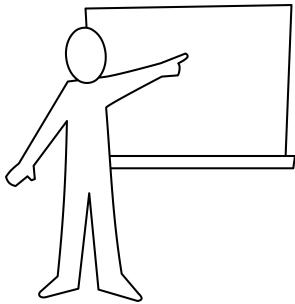
```
Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
```

- After an application is stopped, Kafka migrates any tasks that had been running in this instance to available remaining instances

ybhandare@greendotcorp.com

# Module Map

---



- An Introduction to the Kafka Streams API
- Kafka Streams Concepts
- Creating a Kafka Streams Application
- Kafka Streams by Example ... ←
- Managing Kafka Stream Processing
-  Hands-on Lab

ybhandare@greendotcorp.com

## A Simple Kafka Streams API Example (1)

- Define the Kafka Streams configuration properties

```
public class SimpleStreamsExample {  
  
    public static void main(String[] args) throws Exception {  
        Properties config = new Properties();  
        // Give the Streams application a unique name. The name must be unique in the Kafka cluster  
        config.put(StreamsConfig.APPLICATION_ID_CONFIG, "simple-streams-example");  
        config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");  
        config.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");  
        // Specify default (de)serializers for record keys and for record values.  
        config.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,  
                   Serdes.ByteArray().getClass());  
        config.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,  
                   Serdes.String().getClass());  
    }  
}
```

The Streams application as a whole can be launched just like any normal Java application that has a `main()` method.

Things to point out:

- Line 8 shows that the KafkaStreams type uses some of the same configurations as the standard clients
- Lines 9 and 10 show the SerDes key and value; in this example they will be the same for the inbound and outbound messages and so will not have to be overridden

## A Simple Kafka Streams Example (2)

- Create the processing topology, transform the data, run the application

```
StreamsBuilder builder = new StreamsBuilder();

// Construct a KStream from the input Topic "TextLinesTopic"
KStream<byte[], String> textLines = builder.stream("TextLinesTopic");

// Convert to upper case
KStream<byte[], String> uppercasedWithMapValues =
    textLines.mapValues(value -> value.toUpperCase());

// Write the results to a new Kafka Topic called "UppercasedTextLinesTopic".
uppercasedWithMapValues.to("UppercasedTextLinesTopic");

// Run the Streams application via `start()`
KafkaStreams streams = new KafkaStreams(builder.build(), config);
streams.start();

// Stop the application gracefully
Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
}

}
```

In this example, the `textLines` KStream object is created by consuming the topic `TextLinesTopic`. `textLines` is transformed by `mapValues` to convert all of the characters in the message to uppercase, resulting in the KStream object called `uppercasedWithMapValues`. That resulting KStream is then produced to the topic `UppercasedTextLinesTopic`.



The `key` is `null` in this case, that's the reason why we're using the `ByteArray` SerDes for it (on the previous slide)

## Kafka Streams With Stateful Processing (1)

- Define the Kafka Streams configuration properties

```
public class SimpleStreamsExample {  
  
    public static void main(String[] args) throws Exception {  
        Properties config = new Properties();  
        // Give the Streams application a unique name. The name must be unique in the Kafka cluster  
        config.put(StreamsConfig.APPLICATION_ID_CONFIG, "simple-stateful-example");  
        config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");  
        config.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");  
        // Specify default (de)serializers for record keys and for record values.  
        config.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,  
                   Serdes.ByteArray().getClass());  
        config.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,  
                   Serdes.String().getClass());  
    }  
}
```

This code is not new and very similar to the one we saw 2 slides ago... only the name of the app and the SerDes for the Key has changed.

ybhandare@greendotcorp.com

## Kafka Streams With Stateful Processing (2)

```
StreamsBuilder builder = new StreamsBuilder();

// Construct a KStream from the input Topic "TextLinesTopic"
KStream<byte[], String> textLines = builder.stream("TextLinesTopic");

final KTable<String, Long> wordCounts = textLines
    .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
    // Count the occurrences of each word (record key).
    .groupBy((key, word) -> word,
        Serialized.with(Serdes.String(), Serdes.String()))
    .count("Counts");

// Write the 'KTable<String, Long>' to an output Topic
wordCounts.toStream()
    .to("WordsWithCountsTopic",
        Produced.with(Serdes.String(), Serdes.Long()));

// Run the Streams application via 'start()'.
KafkaStreams streams = new KafkaStreams(builder.build(), config);
streams.start();

Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
}

}
```

In this example, the TextLinesTopic is read into the `textLines` KStream object. `textLines` is run through several stream processors which break each message into separate messages containing single words (based on whitespace) converted to lowercase (`flatMapValues()`). Each message is converted to a new message retaining the same value but also using the value as the key (`groupBy()`) before being counted based on the key and values kept in a topic called "Counts" (`count()`). `groupBy()` transparently repartitioned the data into a new internal topic before running the stateful `count()` operation to ensure that all occurrences of a given word are on the same partition. Note that because the `count()` output is stateful, a final type of KTable was specified for the generated object, `wordCounts`.

## Kafka Streams With Stateful Processing (2)

`wordCounts` is converted to a KStream (`toStream()`) before being produced to a topic called "WordWithCountsTopic". However, since the `count()` transform also changed the data types for the message, `Produced.with()` is used to override the SerDes specified in the Properties object.

ybhandare@greendotcorp.com

## Join Example

- You can enrich records by merging data, *i.e.*, joining data, from different sources
- The example below does a join based on key
  - Effectively, where `leftStream.key == rightStream.key`, use the value from `leftStream`

```
final KStream<Long, String> joinedStream =  
    leftStream.join(rightStream, (leftValue, rightValue) -> leftValue,  
    JoinWindows.of(TimeUnit.MINUTES.toMillis(5)),  
    Joined.with(Serdes.Long(), Serdes.String(), Serdes.String()));
```

ybhandare@greendotcorp.com

## Converting Our Avro Example to Kafka Streams

```
import solution.model.PositionValue;

// Specify Avro Serde and Schema Registry
config.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
           SpecificAvroSerde.class);
config.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
           "http://schema-registry:8081");

StreamsBuilder builder = new StreamsBuilder();

KStream<String, String> positions = builder.stream("vehicle-positions",
    Consumed.with(Serdes.String(), getJsonSerde())); ①
KStream<String, PositionValue> converted = positions
    .mapValues(vp -> getPositionValue(vp)); ②

converted.to("vehicle-positions-avro",
    Produced.with(Serdes.String(), getPositionValueSerde()));

// Continue with stream builder, start, and shutdown hook
```

- ① `getJsonSerde` creates a SerDes to convert JSON formatted data
- ② `getPositionValue` is the method you created in the exercise

- In a previous Hands-On Exercise, you wrote an application to write JSON formatted data into an Avro Topic
- With the Streams API in Kafka, it is very simple to convert this topic into an AVRO topic
- We assume that the `key` is of type `string` and left untouched

## Kafka Streams Processing Guarantees

- Supports **at-least-once** processing
- With no failures, it will process data exactly once
- Upon failure, some records may be processed **more than once**
- Supports **Exactly Once** processing semantics (EOS)
  - Set `processing.guarantee=exactly_once` (Default: `at_least_once`)

Whether or not it is acceptable to process certain records **more than once** depends on the use-case!



It is worth mentioning the **robustness** of exactly once processing (EOS) in KStreams vs plain consumer groups!

ybhandare@greendotcorp.com

## More Kafka Streams API Features

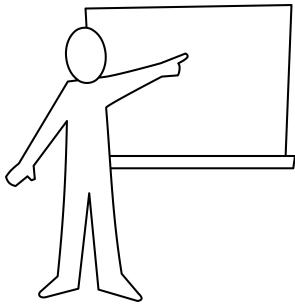
---

- We have only scratched the surface of Kafka Streams
- Check the documentation to learn about processing windowed tables, joining tables, joining streams and tables...

ybhandare@greendotcorp.com

# Module Map

---



- An Introduction to the Kafka Streams API
- Kafka Streams Concepts
- Creating a Kafka Streams Application
- Kafka Streams by Example
- Managing Kafka Stream Processing ...
-  Hands-on Lab



ybhandare@greendotcorp.com

## Parallelizing Kafka Streams

---

- You can run Kafka streaming applications across multiple machines
- Parallelizing Kafka Streams or KSQL applications can improve throughput and performance
- Kafka handles dividing the work across the machines
- It also provides fault tolerance
  - If a machine fails, the task it was working on is automatically restarted on one of the remaining instances
- The number of tasks in the Kafka Streams application may be increased by increasing the number of Partitions

Kafka Streams is designed around parallelization of work loads. Systems running Streams applications should be multi-core in order to take best advantage of this design.

ybhandare@greendotcorp.com

## Fault Tolerance

---

- Clients can enable standby replica tasks for the streaming applications
  - These standby replica tasks have fully replicated copies of the state
  - When a task migration happens, Kafka Streams attempts to assign a task to an application instance where such a standby replica already exists
    - Reduces restoration time for a failed task
- `num.standby.replicas`: number of standby replicas for each task (Default: 0)

The use of standby replicas is recommended for performance sensitive environments.

ybhandare@greendotcorp.com

## Client Resource Utilization

---

- Stateful client applications may use Streams operations (e.g., aggregates, joins, windows)
- Clients keep state in local state stores (*i.e.*, RocksDB)
  - The state stores use local disk on the client machines
  - The state stores have 50MB - 100MB memory overhead
  - Clients persist state stores to a compacted Kafka Topic
    - If the client state store fails, the data is recoverable
  - Using standby replicas for Kafka Streams tasks will result in extra copies of state stores
- If client performance is bound by CPU
  - Add cores or machines, and increase the number of threads
    - `num.stream.threads` (Default: 1)
- If client performance is bound by network, memory, or disk
  - Scale out the clients on to additional machines

ybhandare@greendotcorp.com

### Questions:



What is the main advantage of Kafka Streams versus other stream processing software?

- Kafka Streams API provides a DSL for writing Kafka stream processing applications in Java
  - It is a lightweight library
  - No external dependencies other than core Kafka
- No external cluster is required, unlike most stream processing frameworks

ybhandare@greendotcorp.in

## Hands-On Lab

---

- In this Hands-On Exercise, you will write an application which uses the Streams API in Kafka to process data
- Please refer to the lab **Writing a Kafka Streams Application** in Exercise Book



ybhandare@greendotcorp.com



### Stream Processing with KSQL

ybhandare@greendotcorp.com

# Agenda

---



1. Introduction
2. Fundamentals of Apache Kafka
3. Kafka's Architecture
4. Developing With Kafka
5. More Advanced Kafka Development
6. Schema Management In Kafka
7. Data Pipelines with Kafka Connect
8. Stream Processing with Kafka Streams
9. Stream Processing with KSQL ... ←
10. Event Driven Architecture
11. Confluent Cloud
12. Conclusion
13. Appendix: Basic Kafka Administration
14. Appendix: Apache Kafka driving ML & Data Analytics

ybhandare@greendotcorp.com

## Learning Objectives

---

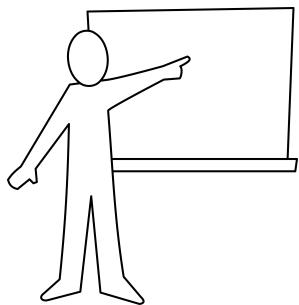


After this module you will be able to:

- Use Confluent KSQL to filter and map a stream
- Write a KSQL query that joins two streams or a stream and a table
- Write a KSQL query that aggregates values per key and time window
- Explain when to use KSQL versus Kafka Streams

ybhandare@greendotcorp.com

## Module Map



- KSQL for Apache Kafka ...
- Writing KSQL Queries
- Hands-on Lab

ybhandare@greendotcorp.com

# Event Transformation with Stream Processing



Apache Kafka® library to write real-time applications and microservices in Java and Scala

```
object FraudFilteringApplication extends App {  
  
  val config = new java.util.Properties  
  config.put(StreamsConfig.APPLICATION_ID_CONFIG, "fraud-filtering-app")  
  config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka-broker1:9092,kafka-broker2:9092")  
  
  val builder: StreamsBuilder = new StreamsBuilder()  
  val fraudulentPayments: KStream[String, Payment] = builder  
    .stream[String, Payment]("payments-kafka-topic")  
    .filter((_, payment) => payment.fraudProbability > 0.8)  
  
  val streams: KafkaStreams = new KafkaStreams(builder.build(), config)  
  streams.start()  
}
```



## Confluent KSQL

The streaming SQL engine for Apache Kafka®

```
CREATE STREAM fraudulent_payments AS  
SELECT * FROM payments  
WHERE fraudProbability > 0.8;
```

**You write only SQL.** No Java, Python, or other boilerplate to wrap around it!

Both, Kafka Streams and KSQL can be used to transform streams of events. KSQL is way more approachable than Kafka streams for a big population of non-Java developer (e.g. data analysts, engineers, business analysts, etc.)

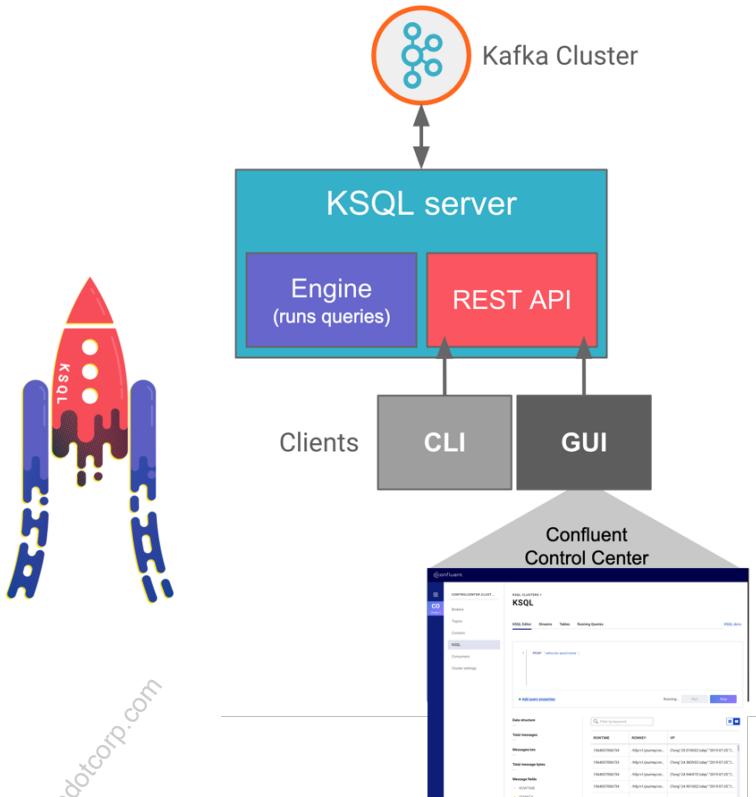
On the slide we see the same result achieved in Java and in KSQL. It is clear that KSQL is free from any boilerplate code. On the other hand we need to admit that Kafka Streams applications offer much more fine grained control to the developer when it comes to stream processing.

## Enable Stream Processing using SQL-like Semantics

 Leverage Kafka Streams API without **any coding required**

 Use **any programming language**

 Connect via **Control Center UI, CLI, REST or deploy in headless mode**



KSQL is really two components: the KSQL Server and the KSQL CLI.

KSQL Server is a pre-built instance of a Kafka Streams application. Rather than coding the streams application in Java, you can develop your processing topology using a higher-level SQL API. The same deployment recommendations for Kafka Streams apply to the instances of the KSQL Server.



**Data exploration**



**Data enrichment**



**Streaming ETL**



**Filter, cleanse, mask**



**Real-time monitoring**



**Anomaly detection**

Here are a few typical use cases our customers report on how they use KSQL

ybhandare@greendotcom

# Using KSQL to Query & Enrich Data

- **Confluent KSQL** is the streaming SQL engine for Apache Kafka
  - transforms & enriches data in a Kafka cluster
  - does real-time stream processing
  - uses Kafka Streams underneath
- KSQL is an alternative to writing a Java application
- KSQL **STREAM** and **TABLE** abstractions are analogous to Kafka Streams API **KStream** and **KTable**
- KSQL is included in **Confluent Community edition**

KSQL officially launched as part of Confluent Community edition 4.1 and is part of the standard download from that version. Confluent Community edition is free and does not require an enterprise license.

This section is intended to be an introduction the KSQL. More in-depth discussions and examples can be found in the *Stream Processing with Kafka Streams and KSQL* course, as well as the Confluent Blog and Online Talks: <https://www.confluent.io/blog/>  
<https://www.confluent.io/online-talks/>

ybhandare@greendotcorp.com

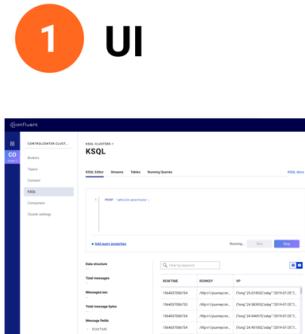
- Identify patterns or anomalies in real-time data, surfaced in milliseconds

```
CREATE TABLE possible_fraud AS
  SELECT card_number, count(*)
  FROM authorization_attempts
  WINDOW TUMBLING (SIZE 5 SECONDS)
  GROUP BY card_number
  HAVING count(*) > 3;
```

This example creates a new topic called "possible\_fraud" by looking at the number of times a field called "card\_number" appears in a topic called "authorization\_attempts" over 5 second windows. If a "card\_number" appears more than 3 times in that window, create a message with the "card\_number" as the key and the count as the value in "possible\_fraud".

ybhandare@greendotcorp.com

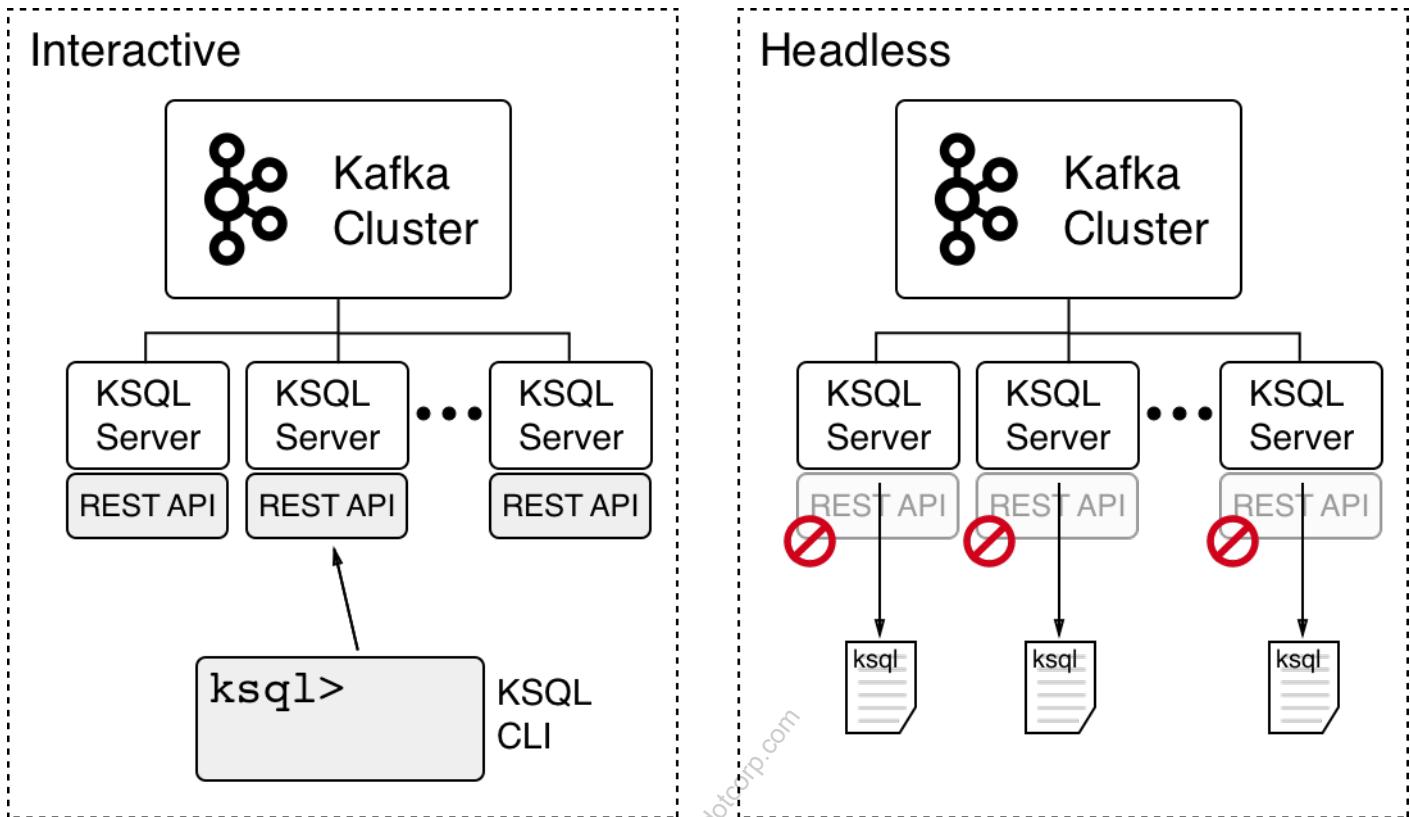
## KSQL Can be used Interactively + Programmatically



The ability to use the KSQL Client from Confluent Control Center was added in CE 5.0.

ybhandare@greendotcorp.com

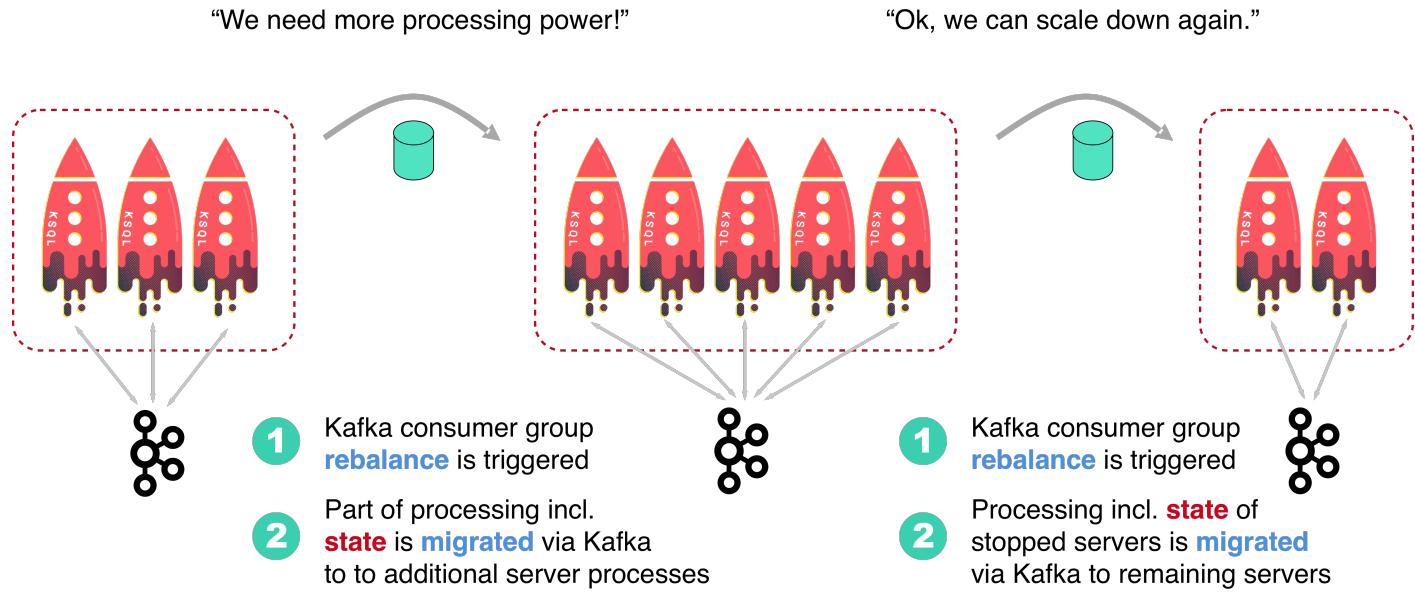
## KSQL Server Modes



KSQL can be run in two modes, interactive and headless

- Interactive mode: a user can access a KSQL server being part of a resource group. The access to the KSQL server happens via its REST API
- Headless mode: The KSQL servers are configured to use a file containing KSQL statements. In this case the KSQL server automatically disables the REST API and thus cannot be accessed from outside anymore. This is the recommended mode when running in production.

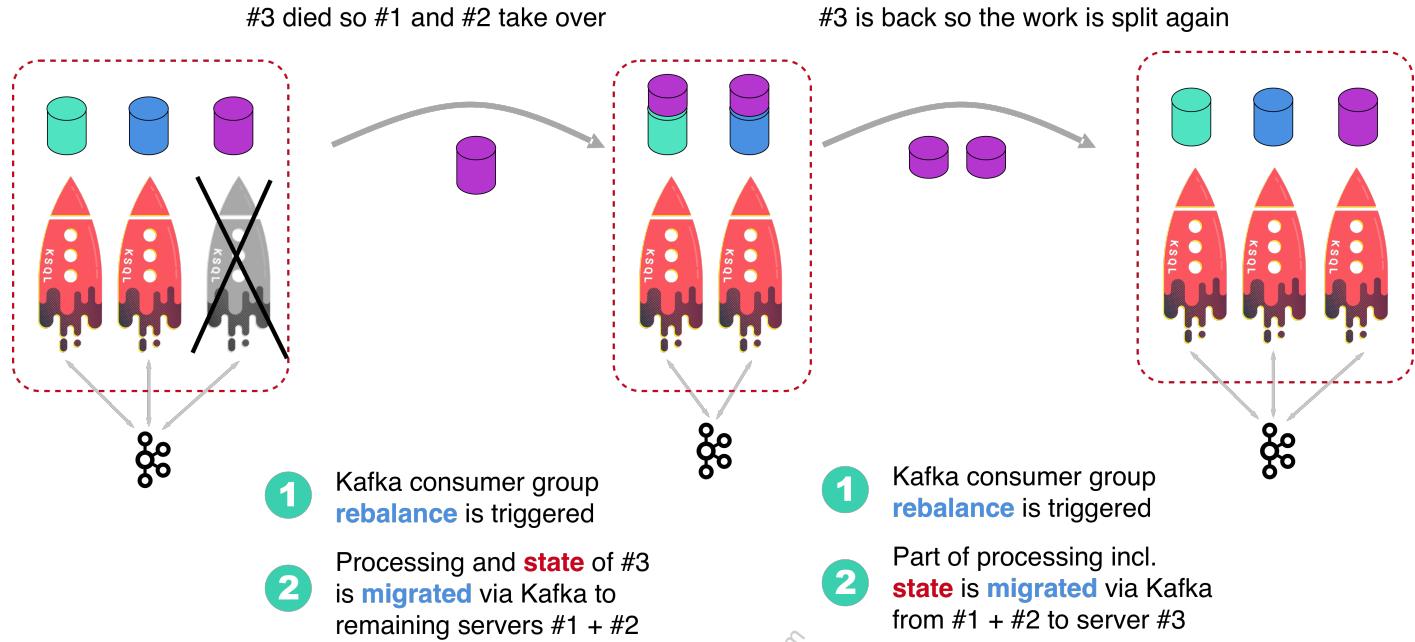
# Elasticity & Scalability



We can add, remove, restart servers in KSQL clusters during live operations.

ybhandare@greendotcom

# Failure Tolerance



Processing fails over automatically, without data loss or miscomputation.

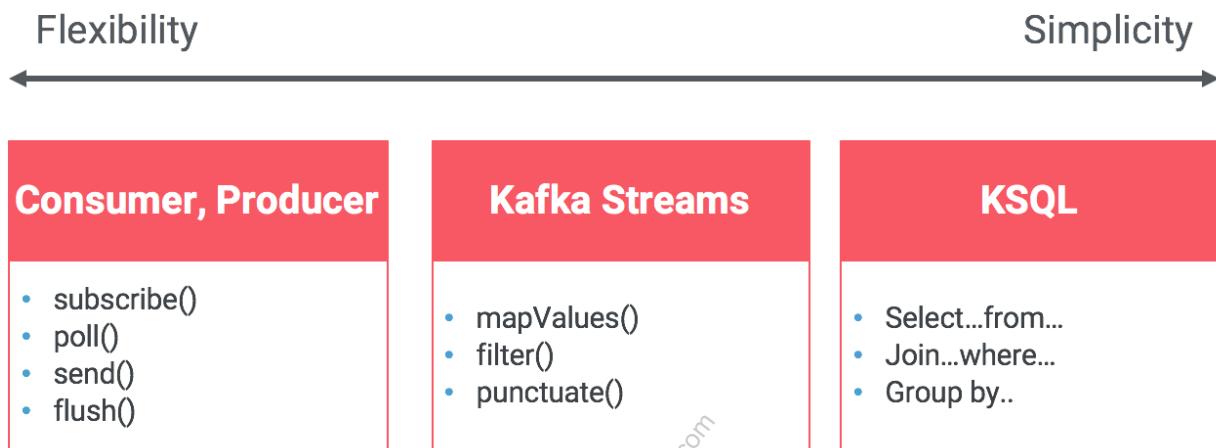
When one member of a KSQL cluster is failing the mechanism of Kafka Consumer Group is used to automatically trigger a rebalancing of work among the remaining group members.

When a new member is added to the cluster once again the rebalancing happens yet again and the workload is equally distributed to all members of the group.

## Which Approach to Choose?

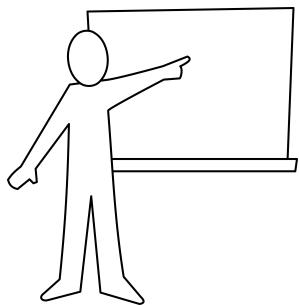
### Comparing KSQL to Kafka Streams to Producer/Consumer

- KSQL is an API around Kafka Streams, and
- Kafka Streams is an API around Producer and Consumer
- Choose the right API for your business



Depending on the level of control required for your environment, you can leverage the appropriate tool.

# Module Map



- KSQL for Apache Kafka
- Writing KSQL Queries ...
- 

ybhandare@greendotcorp.com

## KSQL Syntax

---

- KSQL has similar semantics to SQL
- KSQL statements must be terminated with a semicolon ;
- Multi-line statements

ybhandare@greendotcorp.com

## General KSQL Commands

Command	Description
<code>DESCRIBE</code>	List the columns in a stream or table along with their data type and other attributes
<code>CREATE STREAM</code>	Create a new stream with the specified columns and properties
<code>CREATE TABLE</code>	Create a new table with the specified columns and properties
<code>SHOW TOPICS</code>	List the available Topics in the Kafka cluster that KSQL is configured to connect to
<code>SHOW STREAMS</code>	List the defined streams
<code>SHOW TABLES</code>	List the defined tables
<code>DROP</code>	Drop an existing stream or table

ybhandare@greenelcorp.com

## Non-persistent and Persistent Queries

- Non-persistent query
  - The result of a non-persistent query will not be persisted into a Kafka Topic and will only be printed out in the console
- Persistent query
  - The result of a persistent query will be persisted into a Kafka Topic

Command	Description
<code>SELECT</code>	Non-persistent
<code>CREATE STREAM AS SELECT</code>	Persistent
<code>CREATE TABLE AS SELECT</code>	Persistent

ybhandare@greendotcorp.com

## Stopping Queries

---

- By default, **SELECT** won't stop on its own
  - Since these are never-ending *streams* of data, there is no inherent "end"
- There are differences in stopping a non-persistent versus persistent query
  - Stop a non-persistent query with **CTRL-c** to the console
  - Stop a persistent query with the **TERMINATE** command
- You may also limit the number of records returned with the **LIMIT** clause

ybhandare@greendotcorp.com

## Explore Data in Kafka Topics

- View the content of the Kafka topic `my-pageviews-topic`, which has a JSON payload, in the form of a stream

```
CREATE STREAM pageviews (viewtime BIGINT, user_id VARCHAR, page_id VARCHAR)
  WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'my-pageviews-topic');
```

- View the content of the Kafka topic `my-users-topic`, which has a JSON payload, in the form of a changelog

```
CREATE TABLE users (usertimestamp BIGINT, user_id VARCHAR, gender VARCHAR, region_id VARCHAR)
  WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'my-users-topic', KEY = 'user_id');
```



you can't query a topic directly in KSQL. You have to define a `STREAM` or `TABLE` against that topic first, which maps a schema on top of the raw binary data stored in the topic.

## KSQL Functions and Clauses

- KSQL supports many popular scalar functions that have similar meaning to its counterpart in SQL
  - `ABS`, `CEIL`, `CONCAT`, `LEN`, `ROUND`, `TRIM`, `SUBSTRING`, and many more
  - `EXTRACTJSONFIELD`: Given a string column in JSON format, extract the field that matches
- You can `JOIN` streams and tables
- KSQL supports aggregation functions as well
  - `COUNT`, `MAX`, `MIN`, `SUM`, etc
- Re-key the streams
  - With the `PARTITION BY` clause, the resulting stream will have a key with the specified column
- Extend KSQL with UDFs and UDAFs

Most of the commands available are the same as are available in standard SQL.

One that is specific to KSQL is the `PARTITION BY` clause which allows the use of specific fields as keys for the new messages.

If the functions provided out of the box do not suit your needs then it is possible to extend KSQL by authoring user defined functions (UDFs) and user defined aggregate functions (UDAFs) in Java.

## Persistent Query With Join Example

- Create a persistent query that does a **JOIN** between a **stream** and a **table** based on a field **user\_id**

```
CREATE STREAM vip_actions AS
  SELECT use_id, pageid, action
  FROM clickstream c
  LEFT JOIN users u
  ON c.user_id = u.user_id
  WHERE u.level = 'Platinum';
```

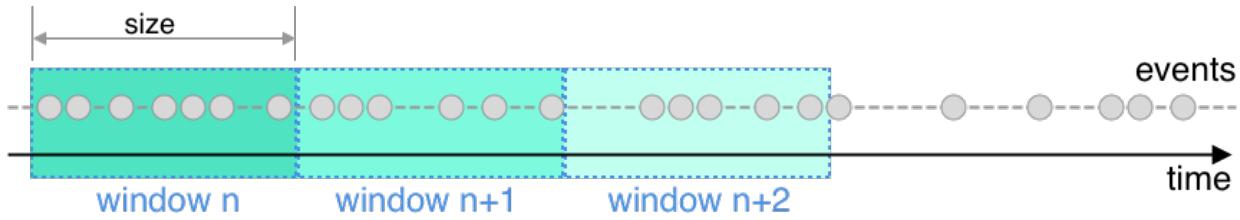


At this time you can't do **subqueries** in KSQL

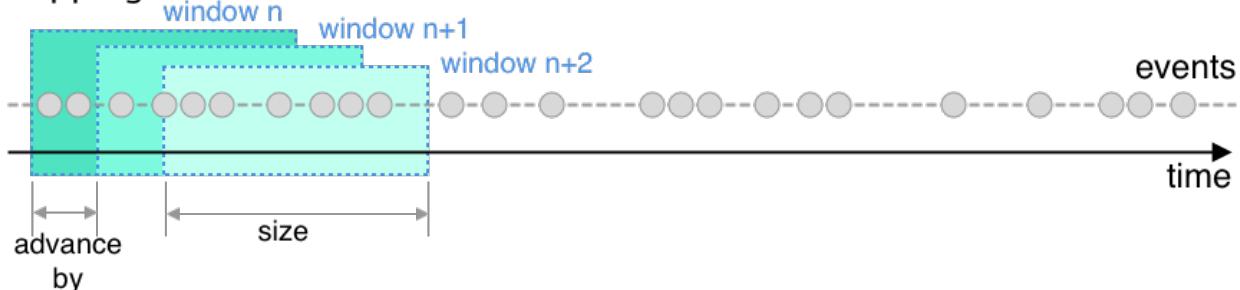
ybhandare@greendotcorp.com

# Windows in KSQL

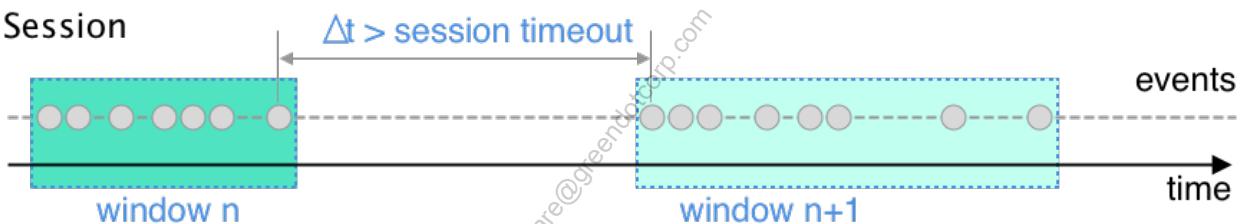
## Tumbling



## Hopping



## Session



- Use the **WINDOW** clause to group input records for aggregations or joins into
  - **TUMBLING**: fixed-sized, non-overlapping windows based on the records' timestamps
  - **HOPPING**: fixed-sized, (possibly) overlapping windows based on the records' timestamps
  - **SESSION**: sessions with a period of activity and gaps in between
- Specify window size, *i.e.*, period of time

## Persistent Query With Windowing Example

- Create a persistent query that counts data in a tumbling window.

```
CREATE TABLE pageviews_regions AS
  SELECT gender, regionid , COUNT(*) AS num_users
  FROM pageviews
  WINDOW TUMBLING (size 30 second)
  GROUP BY gender, regionid
  HAVING COUNT(*) > 1;
```

ybhandare@greendotcorp.com

### Questions:

Who's the the target audience of KSQL?



Confluent KSQL is a declarative stream processing language built on top of Kafka Streams API. Thus non-Java programmers that have some experience with SQL, and want to do stream processing, are the target audience.

ybhandare@greendotcorp.com

## Hands-On Lab

---

- In this Hands-On Exercise, you will use KSQL to transform data and identify patterns
- Please refer to the lab **Using Confluent KSQL** in Exercise Book



ybhandare@greendotcorp.com



**Event Driven Architecture**

ybhandare@greendotcorp.com

# Agenda

---



1. Introduction
2. Fundamentals of Apache Kafka
3. Kafka's Architecture
4. Developing with Apache Kafka
5. Advanced Development with Apache Kafka
6. Schema Management in Apache Kafka
7. Data Pipelines with Kafka Connect
8. Stream Processing with Kafka Streams
9. Stream Processing with KSQL
10. Event Driven Architecture ... ←
11. Confluent Cloud
12. Conclusion
13. Appendix: Basic Kafka Administration
14. Appendix: Apache Kafka driving ML & Data Analytics

ybhandare@greendotcorp.com

## Learning Objectives

---

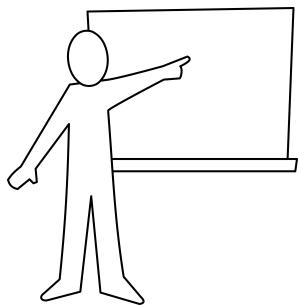


After this module you will be able to:

- Explain the role of events in the Confluent Streaming Platform
- Sketch an image explaining CQRS
- Describe what a microservice is in the context of the Kafka powered event streaming platform

ybhandare@greendotcorp.com

## Module Map



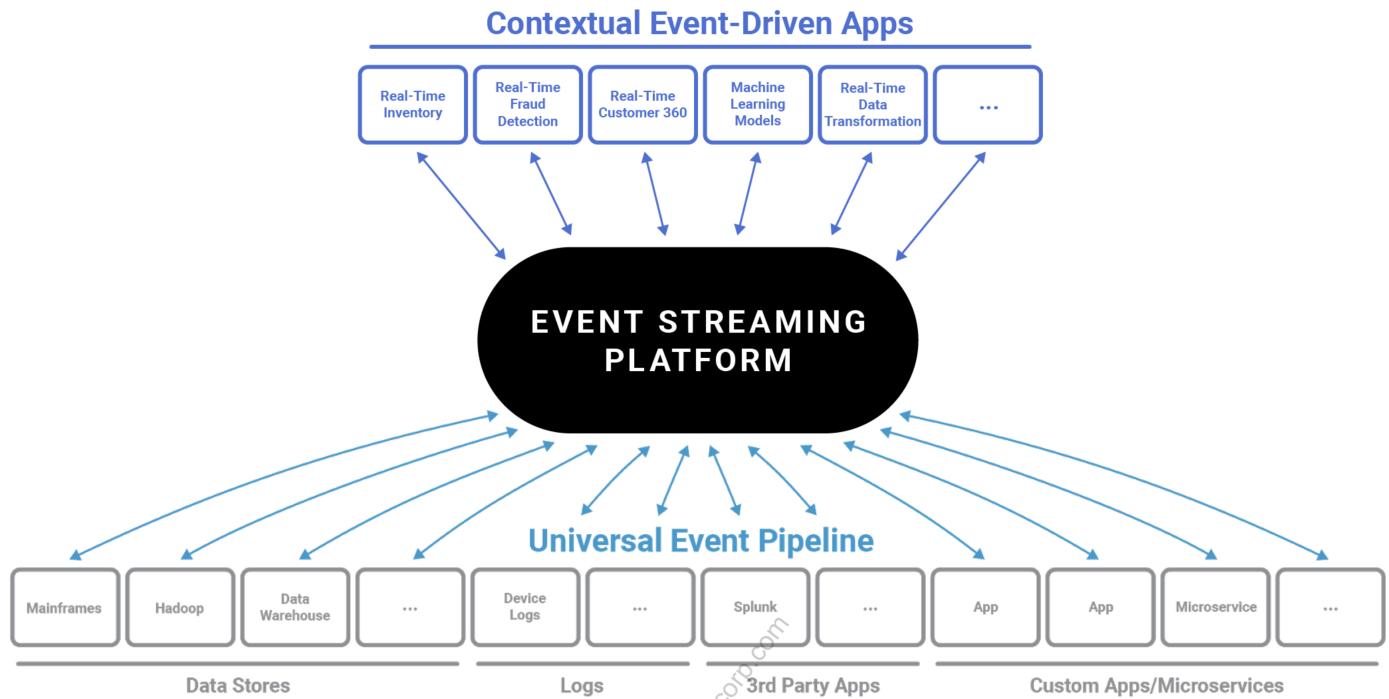
- Event driven Platform ... ←
- From CQRS to Event Sourcing
- Microservices

ybhandare@greendotcorp.com

To rethink data as not stored records or transient messages, but instead as a continually updating stream of events

ybhandare@greendotcorp.com

# Event Driven Platform



Taking this back to the enterprise world, here are some examples of Confluent customers building contextual event-driven applications:

- When an online retailer sells a product to a customer in one part of the world, they must be able to access accurate inventory data to make sure a customer in another part of the world did not buy the last item in stock one second ago. To do this, retailers leverage Confluent Platform to build real-time inventory applications --- marrying real time information with inventory information stored in a database.
- When a bank wants to reduce incidence of online account fraud, they build a login verification service to blacklist IPs from fraudulent bots in real-time. To do this, banks leverage Confluent Platform to build real-time fraud detection applications --- marrying real-time login information with past login attempts, all while cross-referencing a GeoIP database. This application has saved one leading bank tens of millions of dollars per year in reduced fraud losses.

These are just a few examples of contextual event-driven applications. Other examples including building real-time customer 360 applications, or machine learning models. In fact, there are an infinite number of applications that can be built based on what a business needs.

Seeing these examples, it's hopefully clear how event streaming is so much more than "faster ETL" or "messaging that persists". It's an entirely new way to run a business.

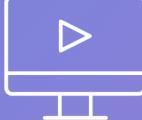
And, seeing these examples, this might be a good time to talk about:

- What event-driven applications would deliver impactful outcomes to your business?
- From a pipeline perspective, which of the six benefits of a universal event pipeline are most relevant to you?
- What would you imagine to be the best place to start?

Now, moving on, in looking at this diagram, you'll also see we've basically drawn out an architecture of an Event Streaming Platform.

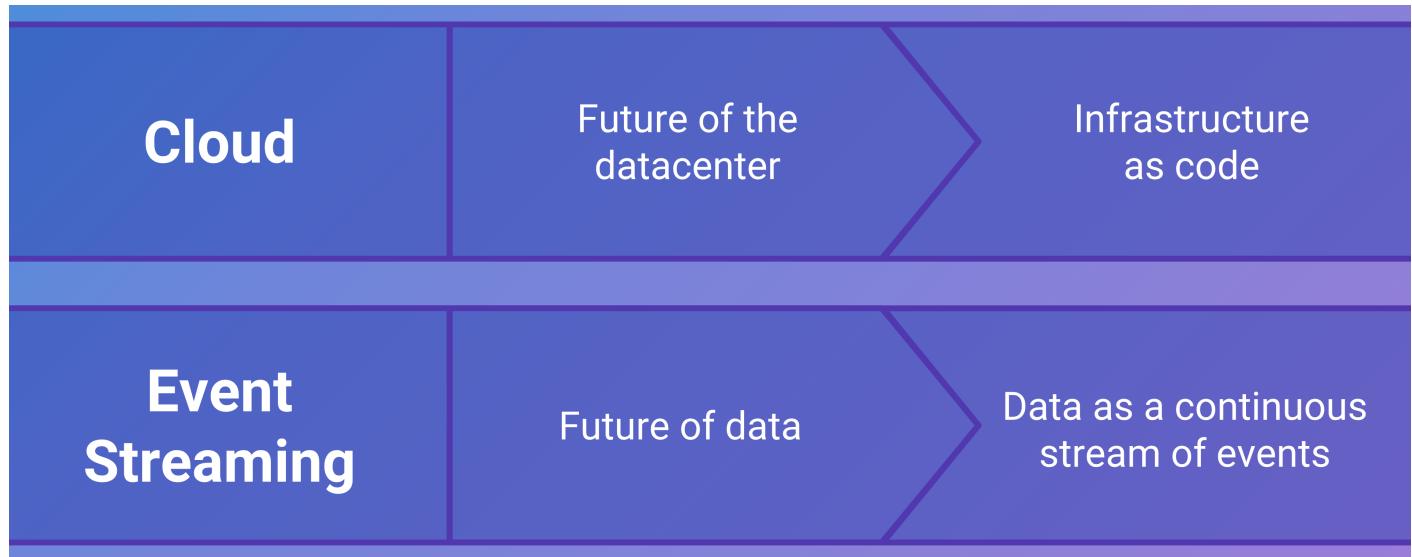
You'll see how an event streaming platform sits at the center of your applications and data infrastructure, connecting to other systems such as Data Warehouse, Hadoop, or custom apps, 3rd party apps, as well as to your event-driven applications. For this reason, customers have referred to it as the Central Nervous System of their enterprise.

ybhandare@greendotcom.co

			
Transportation	Banking	Retail	Entertainment
ETA	Fraud detection	Real-time inventory	Real-time recommendations
Real-time sensor diagnostics	Trading and risk systems	Real-time POS reporting	Personalized news feed
Driver-rider match	Mobile applications / customer experience	Personalization	In-app purchases

Here on the slide we have four sectors with samples where event streaming enabled new kinds of apps and experiences.

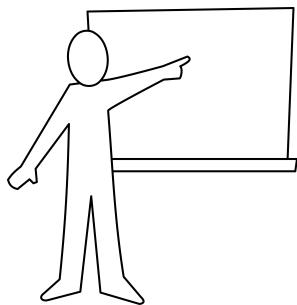
- **Transportation:** ETA (estimated time of arrival), real-time diagnostic of the various parts of a car via connected diagnostic sensors (IoT) and for companies like Uber and Lyft that have to match drivers to riders
- **Banking:** Real-time fraud detection and prevention, real-time trading and risk analysis, customer 360 experiences
- **Retail:** Have a real-time overview of inventory, real-time reporting and personalization a la Amazon or Ebay
- **Entertainment:** Recommendation systems like Netflix, Spotify and others, personalized news feed like Google or Apple News, in-app purchases e.g. in games



The introduction of event streaming, as the future of data streaming, is a fundamental paradigm shift, similar to what was the cloud was to the future of the datacenter

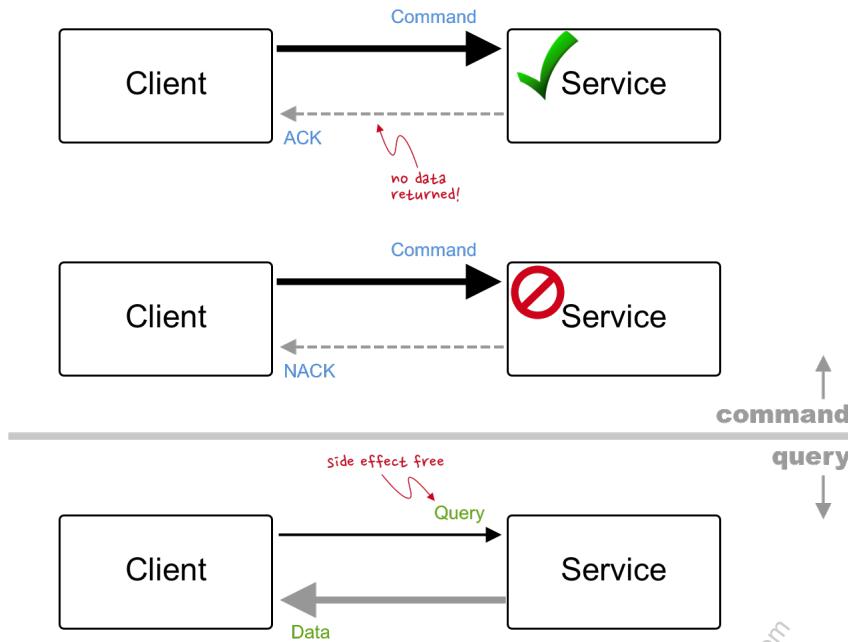
Now let's step back for a moment and talk about from a developers perspective **event streaming** is enabled and done the right way.

# Module Map



- Event driven Platform
- From CQRS to Event Sourcing ... ↙
- Microservices

ybhandare@greendotcorp.com



## Sample Commands

- AddItemToShoppingCart
- CheckoutShoppingCart
- ValidateCreditCard
- ReduceTemperature
- ...

## Sample Queries

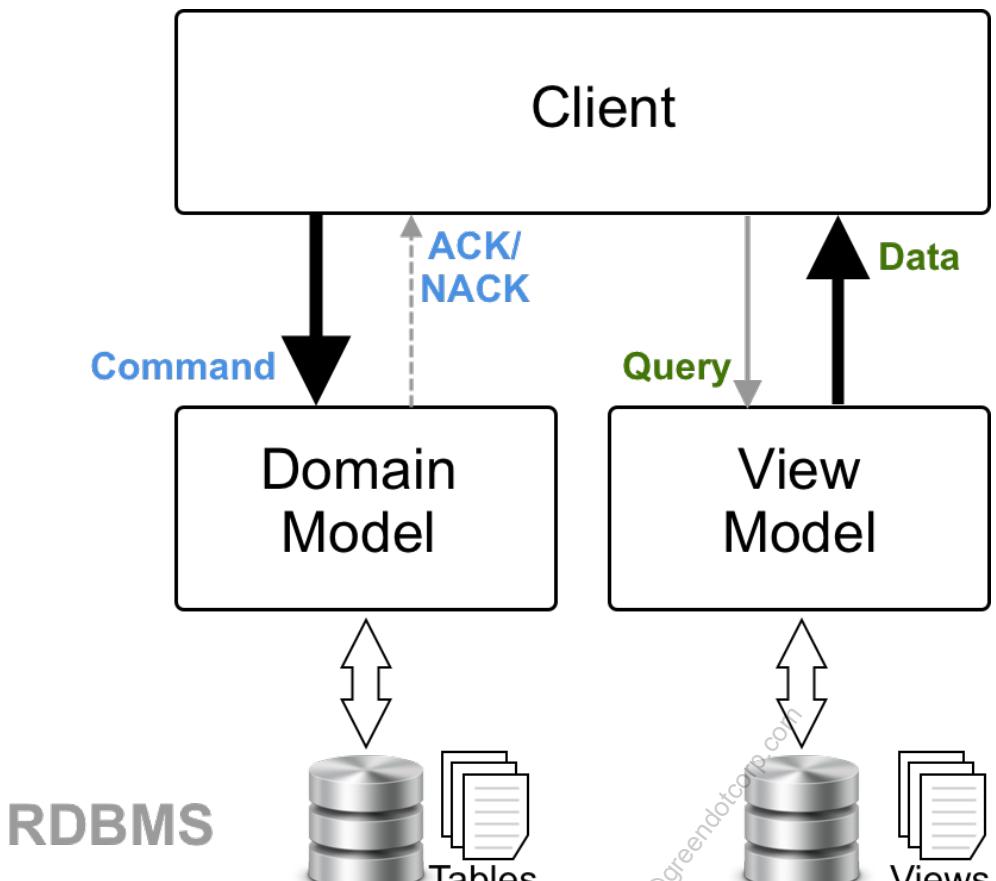
- GetTop10Songs
- GetRelatedProducts
- GetAccountBalances
- ...

One of the building blocks that makes it easier to later on move to an event streaming platform is CQRS.

Command Query Responsibility Segregation (CQRS) is an important architectural pattern that has the following characteristics:

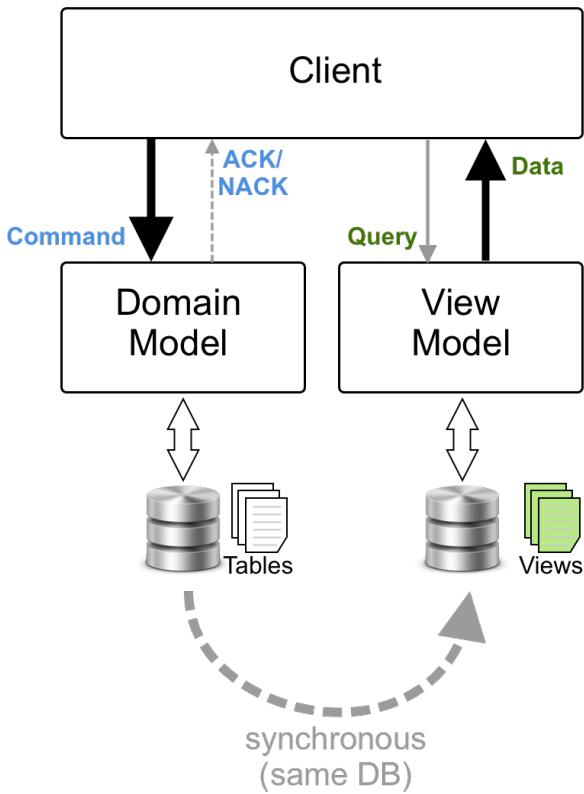
- When talking about CQRS we picture a **client** and a **service**
- **Commands** are sent by the client to the service
- The service tries to execute the desired **action** and answers with
  - **ACK** if the command could be executed successfully
  - **NACK** if the command could not be executed
- The service never returns any data other than **ACK/NACK** and maybe some additional meta data about the result of the invoked action
- The client uses **queries** to get data from the service
- Queries are **side-effect free**, that is, they do not cause any changes in state on the side of the service
- Since queries are side-effect free, they **can be repeated** multiple times, always delivering the same result

## Domain Model and View Model

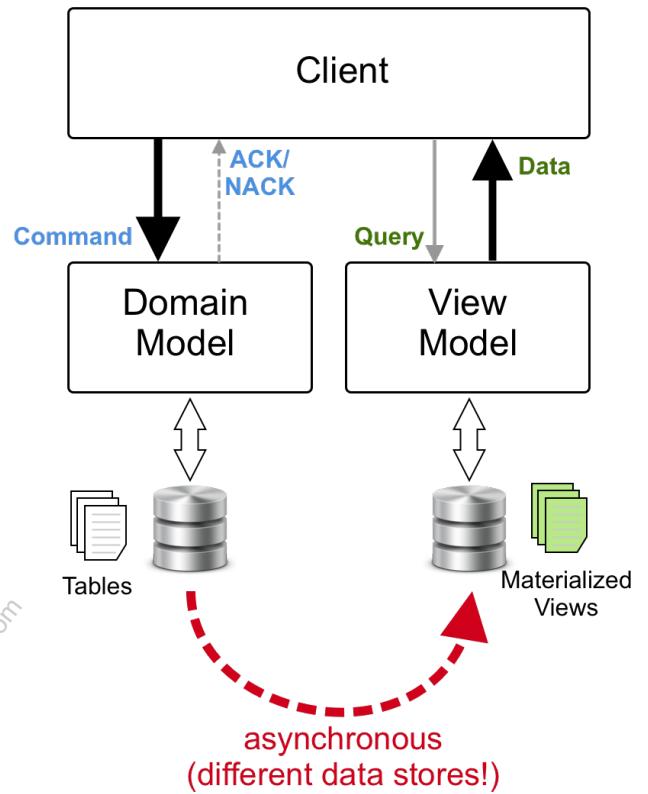


- Commands or actions are executed against a domain model
- The resulting state changes are often persisted to a relational DB
- Queries are executed against a denormalized view of the current state of the domain
- Often these denormalized views are provided by database views in an RDBMS

## Consistent



## Eventually Consistent



The model of CQRS that we saw on the previous slide does not scale well because everything has to be consistent and thus synchronous. The next step was to introduce **eventual consistency** into the picture.

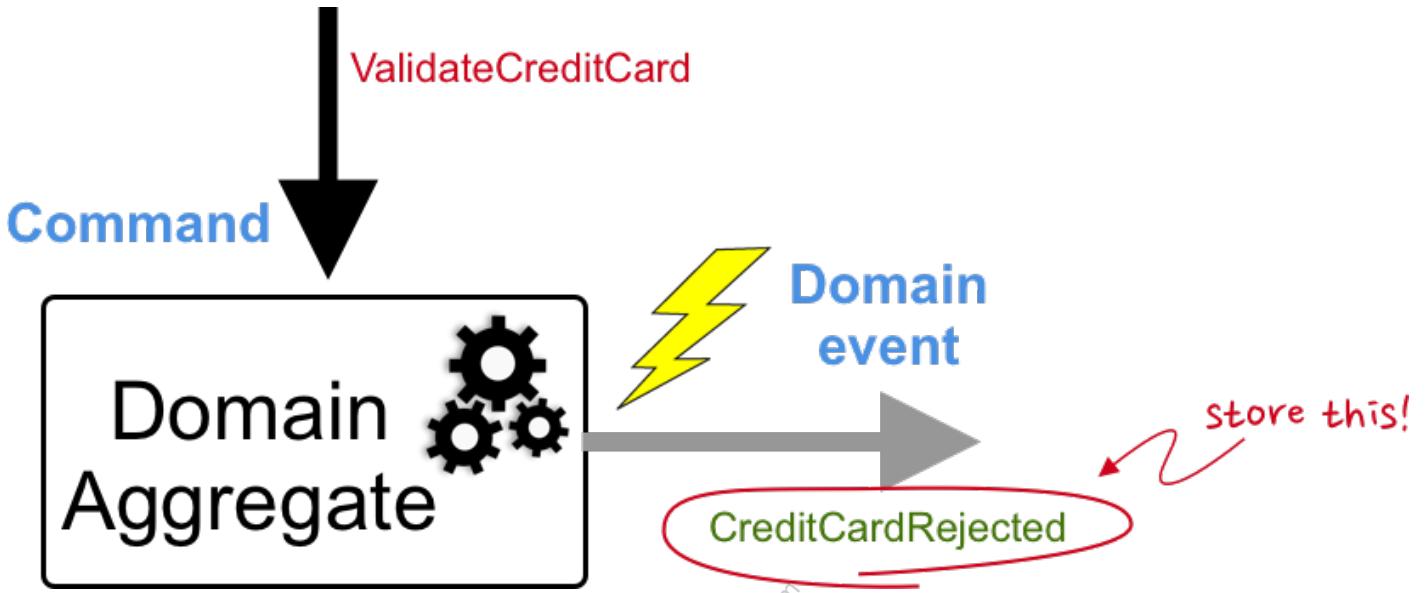
- On the left side we have the fully consistent view where the query that is executed right after an update caused by a command provides the new state
- This is often realized by having a single transactional DB as the target of a change triggered by a command and the source of data requested by a query
- Table changes triggered by the command are reflected immediately in the database views that are providing the data to the queries
- Now we can decouple the data store for the state of the domain model and the data store for the view model as shown on the right side of the image on the slide
- The source of the data requested by the queries in this slightly changed architectural pattern are often called **materialized views** (instead of only **views**)

## Consistent versus eventual-Consistent

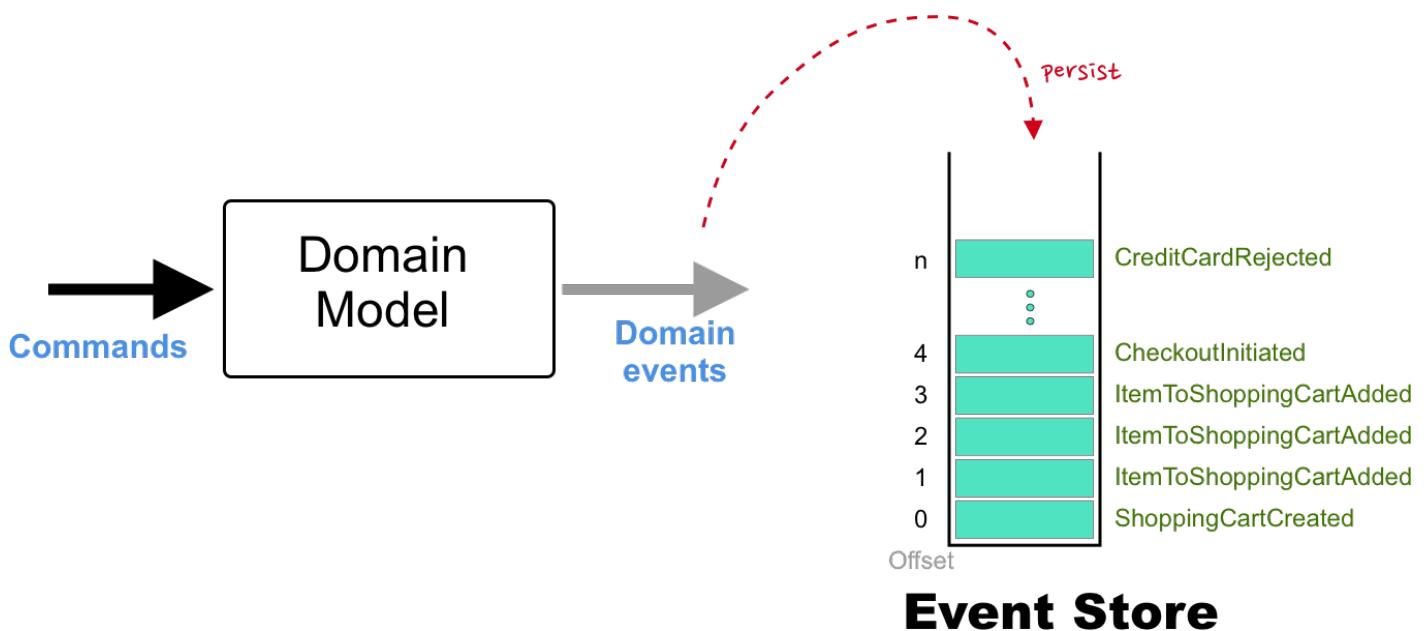
---

- An asynchronous process updates the materialized views each time a change in the domain model (and its supporting DB) is happening
- With this there is a short delay until the materialized views reflect the new state of the world
- We also say that the view model is **eventual consistent**
- The client has to cope with this fact - namely that the data queried may be stale!

ybhandare@greendotcorp.com



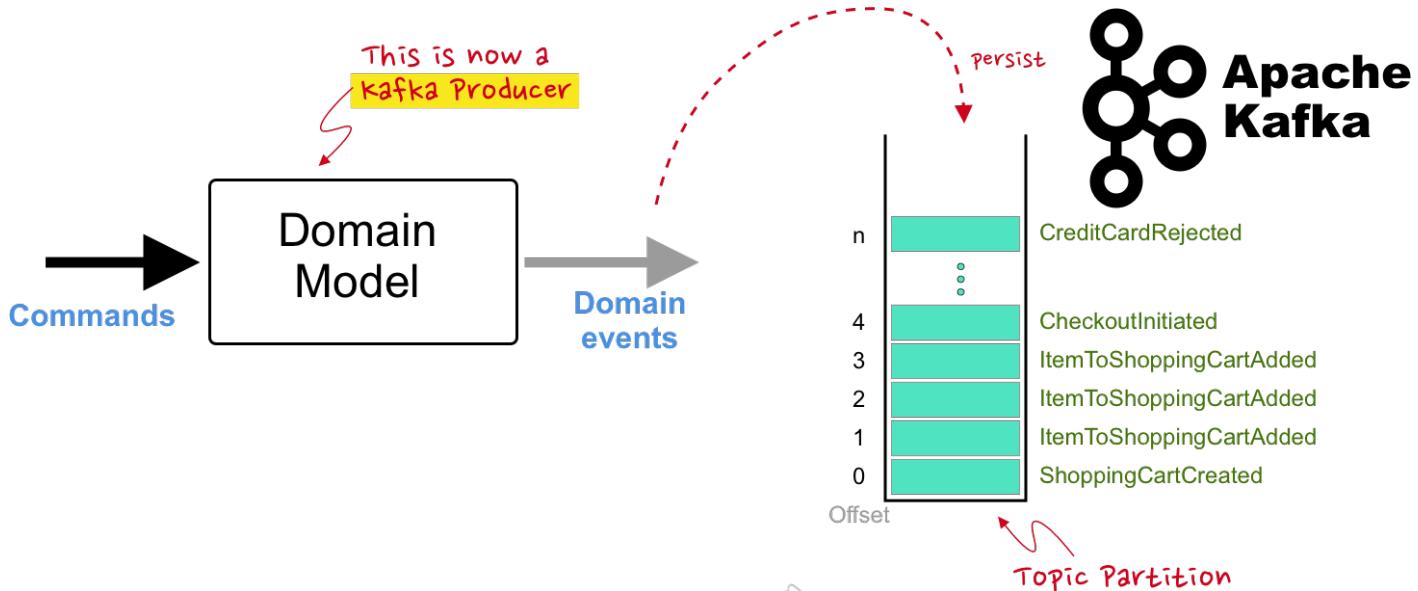
- Instead of only working with the state of the world, as it is "now", we want to see how the world changes.
- We introduce the the concept of an **event**
- **Events** tell us what changed
- A **command** is applied to a **domain aggregate** (or **domain object**)
- A **command**, if successfully applied to the **domain aggregate** always triggers a corresponding **event**
- Let's look at a few samples:
  - AddItemToShoppingCart → ItemToShoppingCartAdded
  - CheckoutShoppingCart → CheckoutInitiated
  - ValidateCreditCard → CreditCardValidated or CreditCardRejected
- Instead of updating the current state (of the domain aggregate) in a database we store the events in a (commit) log



- Events are stored in a structure often called a **commit log**
- A new event is appended **at the end** of the log
- Existing events are immutable, that is, they are never updated or deleted
- The effect of wrong events must be corrected by so called **compensating events**
- The **current state** of the world can be achieved by replaying all events
- We can get the **state at any time** in the past by replaying the events up to that particular point in time



Broken down to the Kafka world: All these events got to the same topic and the key is e.g. `shoppingCartId` such as that all events belonging to the same cart go to the same partition and thus retain ordering.



Of course, when observing the previous slide with the **event store** we should immediately feel familiar. Doesn't this resemble to what Kafka offers us in the first place? And indeed, Apache Kafka is the ideal event store with all the guarantees we expect:

- High volume
- Durability
- Availability
- Append only commit log

Perfect! Here we thus see the connection between the generic event driven architecture and the Confluent Platform driven by Apache Kafka.

# What is an Event?

- ItemAdded
- OrderPlaced
- EmployeeHired
- CreditCardRejected
- TrainArrived
- PlaneDocked

```
public class TrainArrived {  
    public long trainId;  
    public long stationId;  
    public DateTime arrivalTime;  
}
```

- Naming convention: <noun><verb>
- What happened?
- What changed?
- Is immutable
- **Cannot be rejected**
- Often contains **event time**
- Only essential data in event body
- Corrections: **Compensating Events**

An event is:

- always named in **past tense**, ideally with <noun><verb>
- an event tells us what happened and what changed (in the state of the "world") when it happened
- an event is **immutable**, that is, it cannot be changed or deleted, since it was in the past
- it contains the delta that has changed from the previous state as payload
- it often contains the timestamp of **when** the event happened
- "wrong" events must be corrected by **compensating** events
- The body of an event should only contain the data that is essential. Avoid unnecessary data in the events! See the sample **TrainArrived** event.



In the Kafka world, an **event** corresponds to a **Kafka record**



When dealing with an event driven architecture then time really matters. But what do we mean with **time**?

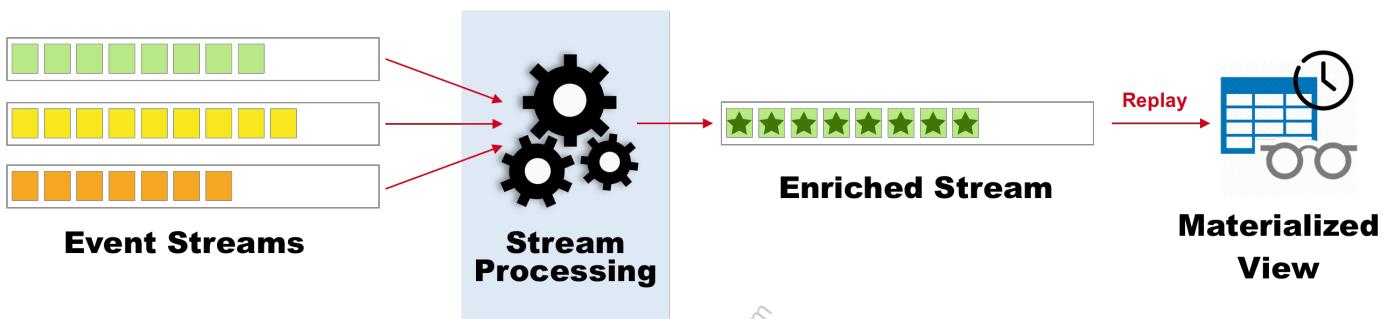
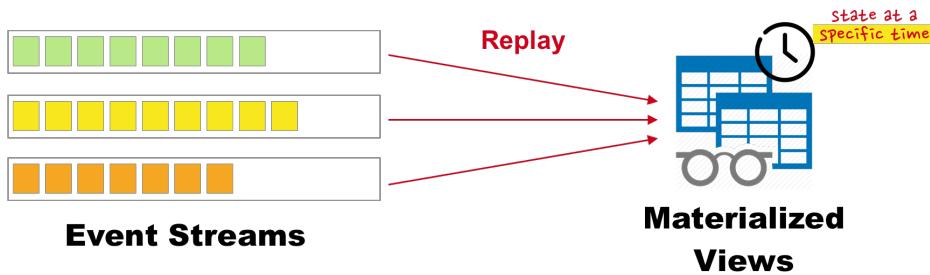
- There are at least 3 notions of time in such an architecture
  - **Event Time:** This is the point in time when the actual event happened of which this event object is a result of
  - **Ingestion Time:** This is the time an event is persisted on the system, e.g. in the commit log of a Kafka broker
  - **Processing Time:** This is the point in time when the event is processed downstream



In a more and more globalized economy we need an **absolute time scale**.

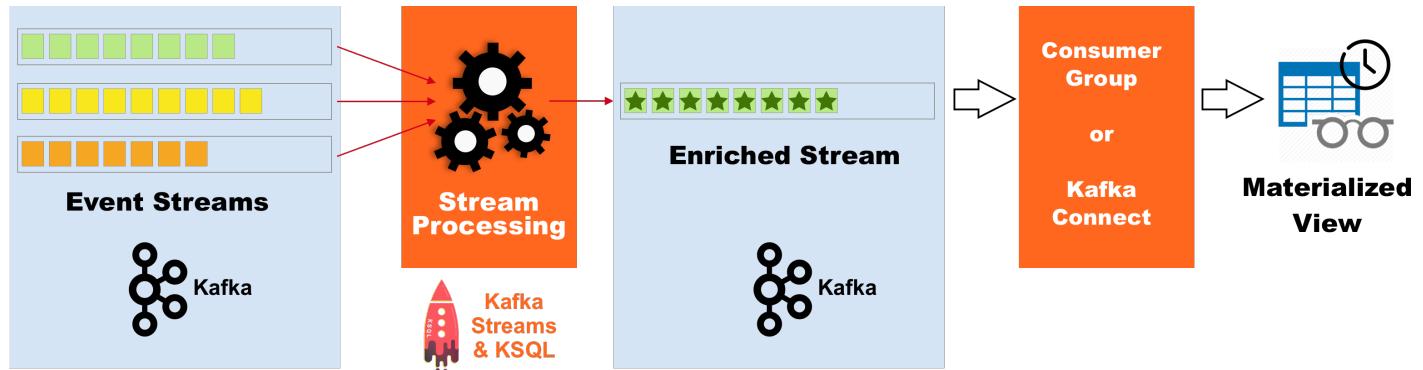
Make sure to store the time in a format that is absolute, e.g. UTC.

## Materialized Views



- Querying a commit log is not efficient and should be avoided at all cost
- But how can we then query the **state of the world**?
- We are creating **materialized views** (sometimes called **projections**) for this purpose
- A materialized view shows the **denormalized state** as of now or any arbitrary point in the past
- materialized views are generated by **replaying streams of events** up to the desired point in time
- Several event streams can be **joined** to achieve denormalization or data enrichment

## Materialized Views

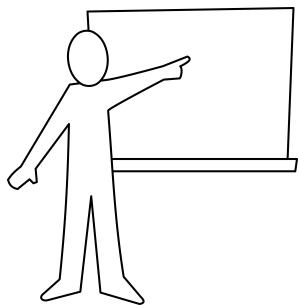


- To generate denormalized or enriched materialized views we can use stream processing such as Kafka Streams or KSQL
- To export the enriched stream and create a materialized view in an external data store we can use either a custom Kafka consumer group or we can use Kafka Connect with the appropriate sink connector

ybhandare@grendelnp:~

## Module Map

---

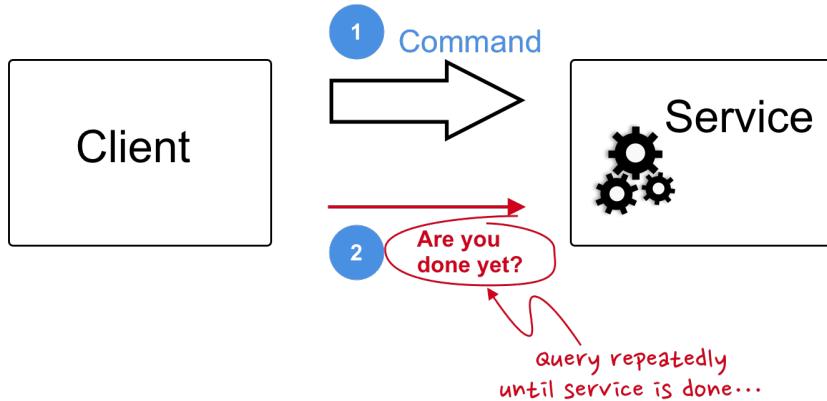


- Event driven Platform
- From CQRS to Event Sourcing
- Microservices ... ↪

ybhandare@greendotcorp.com

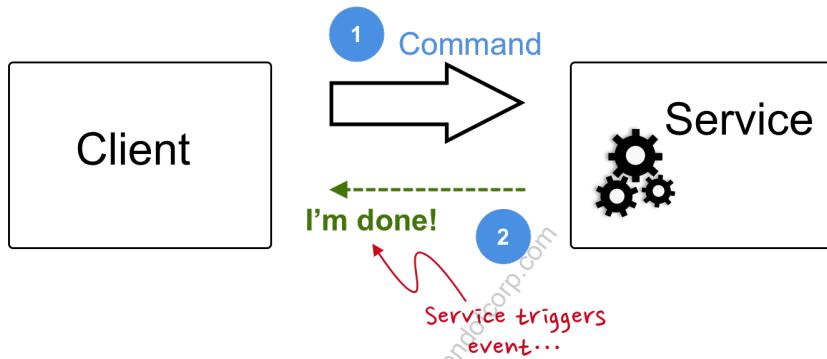
# Tell, Don't Ask!

**ASK**  
✗



**Tight coupling**

**TELL**  
✓



**Loose coupling**

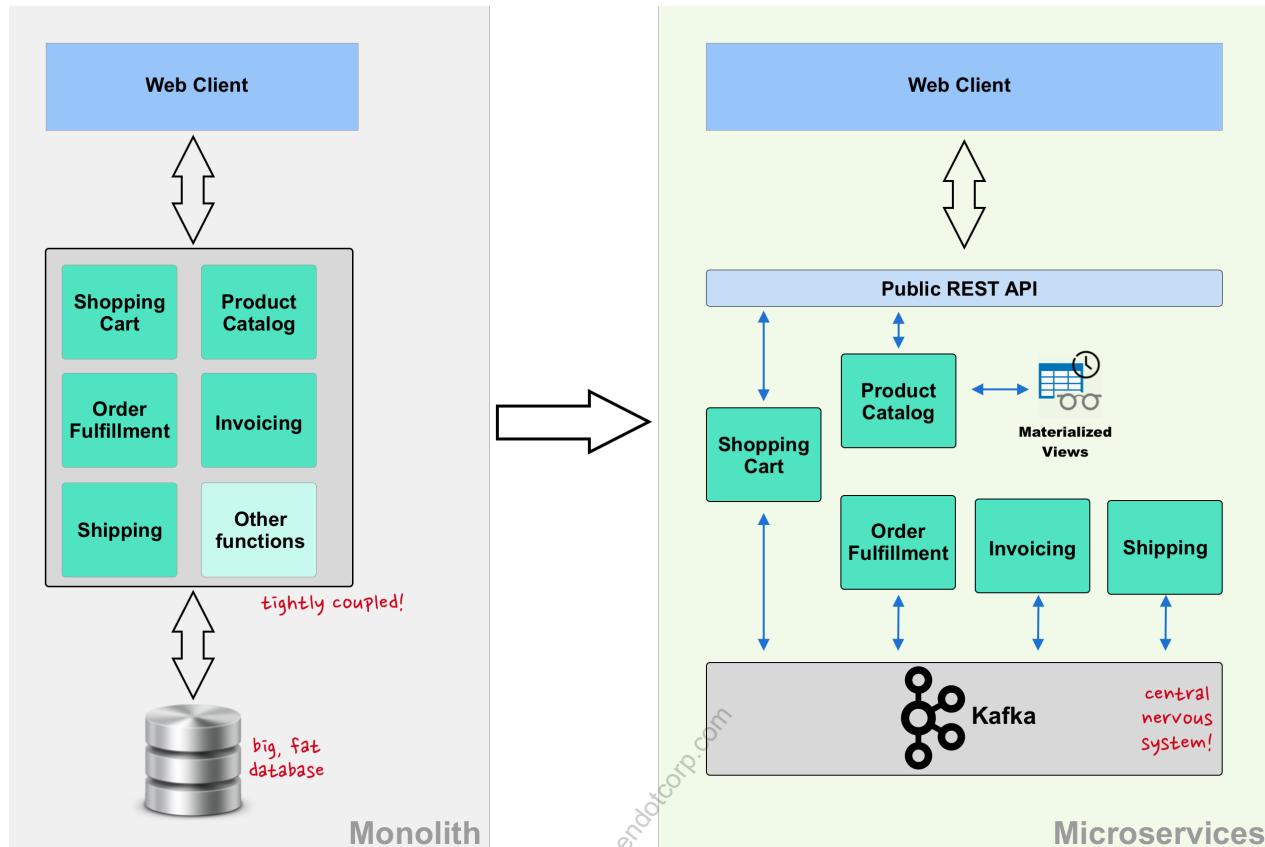
- To decouple individual parts in our application landscape we ideally adhere to the principle of **Tell, don't ask**
- On the slide we see in the upper part a client that requests an action from a service, and then repeatedly asks the service for the result of the action (are you done yet?)
- This is the "ask" case, and it is bad, since it tightly couples the client to the service
- A much better way, as shown in the lower half of the slide, is to have the client wait for a notice from the service when it is done. We call this the "tell" case
- This means that, instead a client to repeatedly ask a service "are you done?", the client waits for an event from the service telling that **it is done**
- Other than indicated on the slide, where the service reacts on a command from the client with an event, the same applies also when the client just is interested in events that happened on the service. E.g. an email service which consumes an event from the shipping service that an order has been shipped. Now the email service can send a confirmation email to the customer...
- The **tell don't ask** principle fits very well into the event driven architecture



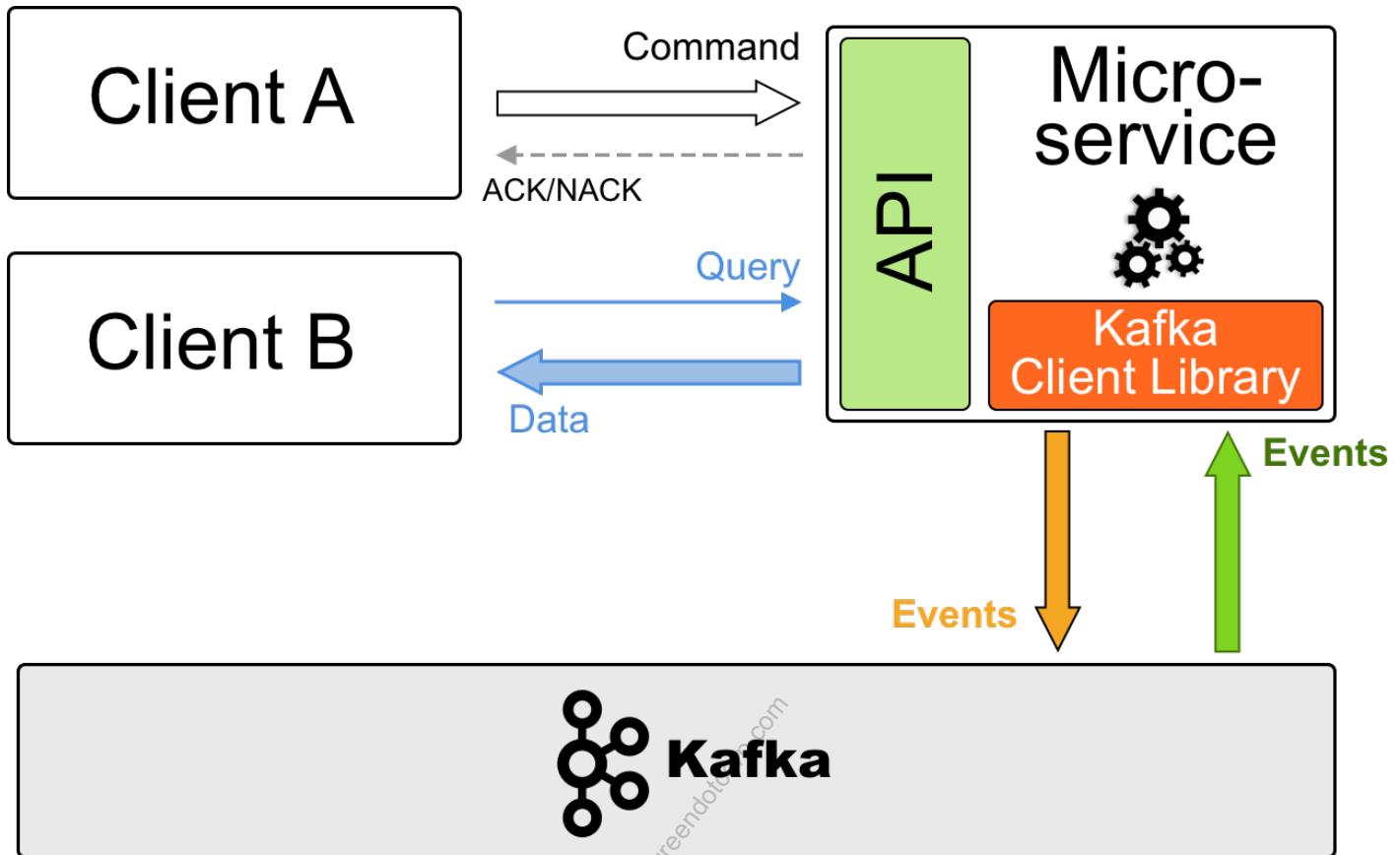
on the **Tell** side, it is worth mentioning, that clients can also just send events (facts) about what's happening, without knowing exactly which service(s) downstream will process those events. In other words, moving away from using events as "commands", and more for "notification".

ybhandare@greendotcorp.com

# Microservices



- The idea of a microservice based architecture is to separate concerns
- Instead of building a huge monolith that does everything (e.g. an ordering system) we break the overall domain down into loosely coupled sub-domains
- Each microservice is responsible for a particular sub-domain only
- Individual microservices should be only loosely coupled and ideally communicate with each other via events, adhering to the **Tell, don't ask** principle
- Each microservice that needs to be directly accessed by other microservices, has a well defined public API
- The public API should follow the CQRS pattern
- Each microservice should be able to have its very own and independent release cycle



- A microservice, depending on its functionality, can have a public API
- Clients can send commands to this service or query this service via the API
- The microservice can be a consumer of events
- The microservice can be a producer of events (trigger events)
- In the Kafka world consuming events would mean that the microservice is a Kafka consumer
- Similarly if the microservice is triggering/producing events then it is a Kafka producer
- Kafka serves as event store and as message bus between the microservices.



Please explain how a microservice can best leverage the Kafka powered real-time event streaming platform.

ybhandare@greendotcorp.com

## Hands-On Lab

---

Please refer to the lab **Writing a Microservice** in the Exercise Book



ybhandare@greendotcorp.com

## Further Reading

---

- Designing Event-Driven Systems (Book):  
<https://www.confluent.io/designing-event-driven-systems>

ybhandare@greendotcorp.com



**Confluent Cloud**

ybhandare@greendotcorp.com

# Agenda

---



1. Introduction
2. Fundamentals of Apache Kafka
3. Kafka's Architecture
4. Developing with Apache Kafka
5. Advanced Development with Apache Kafka
6. Schema Management in Apache Kafka
7. Data Pipelines with Kafka Connect
8. Stream Processing with Kafka Streams
9. Stream Processing with KSQL
10. Event Driven Architecture
11. Confluent Cloud ... ←
12. Conclusion
13. Appendix: Basic Kafka Administration
14. Appendix: Apache Kafka driving ML & Data Analytics

ybhandare@greendotcorp.com

## Learning Objectives

---

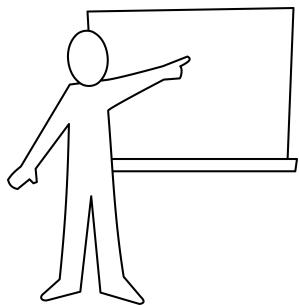


After this module you will be able to:

- Create an Account on Confluent Cloud
- Create an environment and a cluster in CCloud
- Manage topics on your cluster
- Write Kafka clients for your cluster
- Configure and use the Schema Registry in CCloud

ybhandare@greendotcorp.com

## Module Map



- Confluent Cloud Overview ... ←
- Using the Cloud CLI and Web UI
- Configuring Kafka Clients

ybhandare@greendotcorp.com

## Two Ways to Deploy Confluent Platform

Self-Managed Software

### Confluent Platform

The Enterprise Distribution of  
Apache Kafka



Deploy on any platform, on-prem or cloud



Fully-Managed Service

### Confluent Cloud

Apache Kafka Re-engineered  
for the Cloud



Available on the leading public clouds



Currently there are two ways to deploy the Confluent Streaming Platform. You can either self manage the software on premise or in the cloud of your choice (AWS, GCP and Azure), or you can decide to go with a fully manage service, the Confluent Cloud.

Confluent Cloud comes in two flavors,

- **Confluent Cloud:** Confluent Cloud™ is a fully-managed streaming data service based on open-source Apache Kafka. Pricing is consumption based
- **Confluent Cloud Enterprise:** Same as above, but:
  - Unlimited data throughput and retention
  - 99.95% uptime SL
  - Enterprise level security & compliance
  - etc.

## What is Confluent Cloud?

---

- **Fully managed** Streaming Platform based on Apache Kafka
- Resilient and scalable
- Manage via Web UI
- Automate via CLI

Confluent Cloud is a resilient, scalable streaming data service based on Apache Kafka®, delivered as a fully managed service. Confluent Cloud has a web interface and local command line interface. You can manage cluster resources, settings, and billing with the web interface. You can use Confluent Cloud CLI to create and manage Kafka topics.

ybhandare@greendotcorp.com

## Comparing Confluent Cloud Editions

Description	CCloud Enterprise	Confluent Cloud
Support	24x7 by Confluent	Community
Data retention	Unlimited	1-30 days
Cluster throughput	Write: 0.5 MBps - 1 GBps Read: 1 MBps - 2 GBps	Write: max. 100 MBps Read: max. 100 MBps
Data isolation	physical	logical
Multi-zone HA	yes	no
Max clusters	unlimited	5
Max partitions	100,000	2,048
Schema Registry	1000 (5000) schema versions	200 schema versions
KSQL	Yes	Yes
GDPR readiness	Yes	Yes
SOC-2 Type II	Yes (AWS only)	Yes (AWS only)
PCI level 1, HIPAA	Yes	No

On this slide we present a comparison between the Confluent Cloud and the enterprise edition of Confluent Cloud.

- **Confluent Cloud:** Targeted mostly at developers. Billing is usage based
- **Enterprise:** For use in production and production like environments that need maximum scalability, availability and isolation and security

Supported Features:

- A single Schema Registry is available per environment (e.g. Prod, Dev).
- Up to 1,000 schema versions are included in CCE. The quota **can** be increased up to 5,000 schema versions.
- Access Control to Schema Registry is based on API key and secret.
- Each environment must have at least one Kafka cluster to enable Schema Registry

## Confluent Cloud - Professional versus Enterprise

---

- Your VPC must be able to communicate with the Confluent Cloud Schema Registry public internet endpoint.  
For more information, see: *"Using Confluent Cloud Schema Registry in a VPC Peered Environment"*
- Only available in the US region.

ybhandare@greendotcorp.com

# CCE - Full feature set

Environment overview

CLUSTERS SCHEMA REGISTRY

cluster-name\_01

Cloud Provider: Amazon AWS / Region: US-West1 (Oregon) / Availability: Single Zone

Status: Running

Read	Write	Storage
Provisioned: 300 MB/s	Provisioned: 1 MB/s	Provisioned: 50 GB
Peak usage: 200 MB/s	Peak usage: 0.9 MB/s	Current usage: 34.11 GB
Average: 123 MB/s	Average: 0.45 MB/s	% available: 15.23%

cluster-name\_02

Cloud Provider: Amazon AWS / Region: US-West1 (Oregon) / Availability: Single Zone

Status: Running

Read	Write	Storage
Provisioned: 300 MB/s	Provisioned: 1 MB/s	Provisioned: 50 GB
Peak usage: 202.123 MB/s	Peak usage: 0.888 MB/s	Current usage: 40.239 GB
Average: 2.091 MB/s	Average: 0.751 MB/s	% available: 19.522%

Search clusters Show past 15 mins of activity + Add cluster

<b>Scale</b>	Unlimited throughput, unlimited retention
<b>Availability</b>	99.95% uptime SLA
<b>Durability</b>	Multi-AZ with 3 availability zones (option)
<b>Connection</b>	VPC peering (option)
<b>Support</b>	24x7 Gold SVPC peering (option)
<b>Terms</b>	1 year commitment; flexible payment options
<b>Cloud</b>	AWS and GCP

On this slide we provide an overview over the most relevant features offered by **Confluent Cloud Enterprise**.

- Multiple environments, such as DEV, STAGING, PROD
- Multiple clusters per environments
- Nearly unlimited scalability in **throughput** and **data retention**
- Very high guaranteed availability of at least 99.95%
- Very strong durability guarantees by providing multi-AZ deployments
- 24x7 Gold Support by true Kafka and cloud experts
- No cloud vendor lock-in; CCloud runs on AWS, GCP and Azure
- flexible commitment options



01

## Control access to cluster and resources

- SASL
- Dedicated VPC, cluster, storage
- VPC peering for private endpoint



02

## Protect users, applications and access to data

- ACL Support in Kafka (coming soon)
- GDPR ready
- SOC-2 Type II Compliant
- HIPAA ready



03

## Secure data against unwanted access (Encryption)

- Data at rest encryption
- Over the wire encryption (TLS Support)

Today, security is top of mind for us.

Authentication:

We have three ways

1. Plaintext if you're in dev or prototype
2. Multiple SASL implementations including Kerberos, PLAIN, SCRAM, OAuth  
NOTE: This is in flux and changing fast at this time. SASL/PLAIN are currently available
3. TLS certificates

Authorization:

1. We support ACLs at topic granularity
  - a. Can apply on topics, consumer groups, clusters
2. With 5.0, we support LDAP and AD group support (AD - active directory) and ACL wildcards
  - a. Today, we only support prefix based ACL's

Encryption:

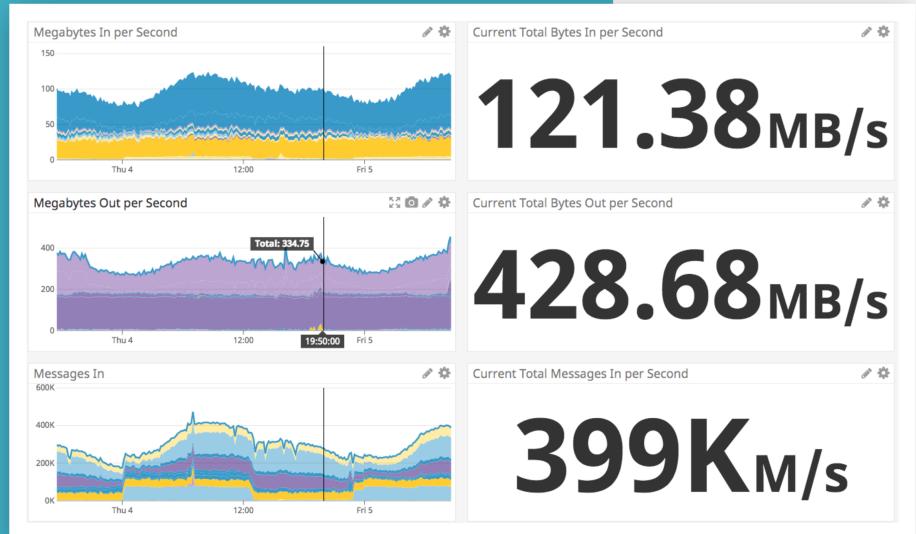
1. We support TLS over the wire
2. Data at rest encryption - we support this today through solutions like Gemalto and Vormetric which allows you to encrypt at the disk or operating system level.

For **Confluent Cloud** the following applies:

- All traffic over the wire requires TLS/SSL encryption and SASL\_PLAIN authentication.
- All data is encrypted at rest on encrypted volumes.
- You are provided auth keys specific to your cluster which you can revoke or reissue if necessary.
- All data is stored on secure infrastructure, with access controls that are restricted to Confluent engineers, inside a Confluent controlled VPC.
- **Confluent Cloud Enterprise** is a private cluster product. Resources are allocated specifically to each cluster. There is no shared data from other customers in your cluster.
- VPC Peering (optional) provides network-level security for **Confluent Cloud Enterprise** customers.

ybhandare@greendotcorp.com

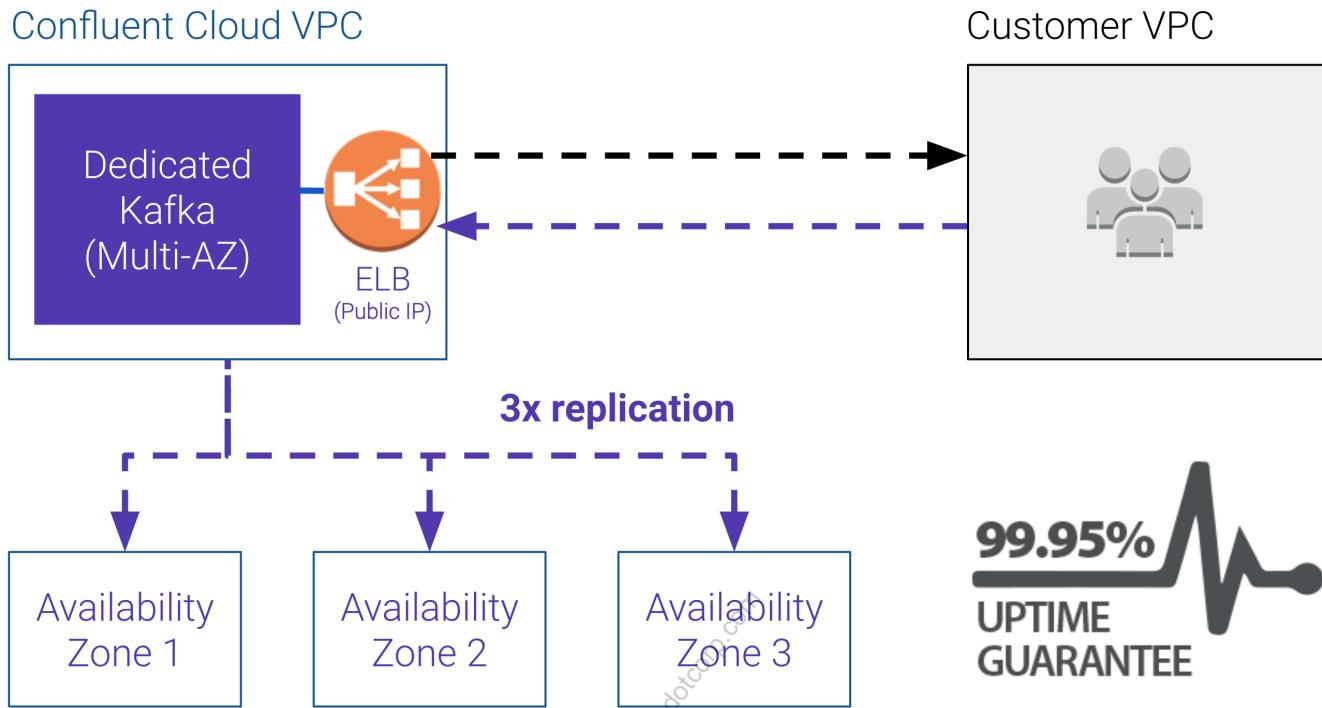
- **Sub-25 ms**  
latencies\* at massive scale
- **3 days**  
or less to get up and running
- **Unlimited**  
throughput and fanout
- **Infinite**  
retention



Confluent Cloud Enterprise has a number of very demanding customers. Their expectations are:

- **minimum latency** and **maximum throughput** respectively
- unlimited fanout of messages
- have their data retained infinitely - implying **unbounded amounts of storage**
- be up and running in no time - ideally in **less than 3 days**

## CCE - High availability



Customers expect high availability even under the worst circumstances. CCloud optionally allows you to configure your hosted clusters to span 3 availability zones. This leads to an **uptime guarantee of 99.95%**.

When a cluster is configured this way then it is guaranteed that replicas of each partition are distributed across those AZ.

The fact that the load balancer on the image is named **ELB**, and other AWS specific names appear on the slide, it does not mean that HA is tied to AWS. HA is supported in a similar way on all supported cloud providers (AWS, GCP, Azure).

# Kafka Expertise, and Why it matters?

<b>Complexity</b>	<ul style="list-style-type: none"><li>• Streaming systems are distributed<ul style="list-style-type: none"><li>◦ Many components with complex interactions</li><li>◦ Challenging to optimize and troubleshoot</li></ul></li></ul>	DOWNTIME RISK		
<b>Non-trivial capacity planning</b>	<ul style="list-style-type: none"><li>• Streaming systems are stateful<ul style="list-style-type: none"><li>◦ Capacity planning is non-trivial</li><li>◦ Retention, memory, compute and n/w need sizing</li></ul></li></ul>	DATA LOSS	LATENCY	
<b>Large surface area to cover</b>	<ul style="list-style-type: none"><li>• They require many APIs, metrics, systems, and configs<ul style="list-style-type: none"><li>◦ Difficult to secure and monitor</li><li>◦ Time-consuming, difficult to learn and manage</li></ul></li></ul>	SECURITY RISK	TIME TO MARKET	SLOW TO LAUNCH

Behind Apache Kafka's power is its architecture. Distributed, stateful systems have a number of characteristics that make them difficult to operate and maintain. Even though Kafka's architecture is elegant and simple compared to competitors, some of the complexity cannot be avoided without deep understanding of the system. You want to focus on building applications, not infrastructure.

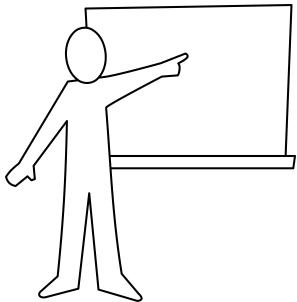
# Confluent Cloud, What does Fully-managed Mean?

<b>Infrastructure management (commodity)</b>	<ul style="list-style-type: none"><li>• Upgrades (latest stable version of Kafka)</li><li>• Patching</li><li>• Maintenance</li></ul>	Infra-as-a-Service
<b>Kafka-specific management</b>	<ul style="list-style-type: none"><li>• Sizing (retention, latency, throughput, storage, etc.)</li><li>• Data balancing for optimal performance</li><li>• Performance tuning for real-time and latency requirements</li><li>• Fixing Kafka bugs</li><li>• Uptime monitoring and proactive remediation of issues</li><li>• Recovery support from data corruption</li></ul>	Platform-as-a-Service Harness full power of Kafka Mission-critical reliability
<b>Scaling</b>	<ul style="list-style-type: none"><li>• Scaling the cluster as needed</li><li>• Data balancing the cluster as nodes are added</li><li>• Support for any Kafka issue with less than 60 minute response time</li></ul>	Future-proof Evolve as you need

Many vendors offer Kafka as a Service. But most of them remain at the level of **Infra-as-a-service**.

Confluent Cloud in addition to that, offers all the other important services listed on the slide.

# Module Map



- Confluent Cloud Overview
- Using the Cloud CLI and Web UI ... ←
- Configuring Kafka Clients

ybhandare@greendotcorp.com

# Confluent Cloud Web Interface - Clusters

The screenshot shows the Confluent Cloud Web Interface for managing clusters. The top navigation bar includes the Confluent Cloud logo and a 'Training-gns' section. The left sidebar has tabs for 'DE' (selected), 'OT', and 'Schemas'. The main content area is titled 'Clusters' and features a search bar, a time filter set to 'Last hour', and a button to '+ Add cluster'. Two clusters are listed: 'demo-cluster' (Cloud Provider: GCP, Region: us-central1, Availability: Single Zone) and 'other-cluster' (Cloud Provider: AZURE, Region: westus2, Availability: Single Zone). Each cluster card shows 'Read' and 'Write' metrics (Peak and Average) and 'Storage' usage (Used and Available). The status for both clusters is 'Running'.

Cluster	Cloud Provider	Region	Availability	Status
demo-cluster	GCP	us-central1	Single Zone	Running
other-cluster	AZURE	westus2	Single Zone	Running

After logging in you first have to select the environment you want to work with. Then you'll be presented with a list of clusters defined in this environment.

On this screen you can also add additional clusters if needed.

Also note the tab **Schema Registry** which will provide you the ability to enable a Schema Registry for this instance. Note that SR instances are per environment.

# Confluent Cloud Web Interface

The screenshot shows the Confluent Cloud Web Interface. On the left, the 'Cluster Settings' page is displayed, featuring a sidebar with clusters DE (Cluster 1), OT (Cluster 2), and Schemas. The main content includes sections for Kafka, API access, Pricing, Usage Limits, and Billing Schedule. A 'Change settings' button is at the bottom. On the right, the 'Topics' page is shown, displaying a list of topics (orders, products, test-topic) with their partitions. A 'Topics' sidebar on the right lists various configuration parameters such as partitions, replication.factor, and cleanup.policy.

In the cluster overview you see important pricing and billing informations as well as usage limits.

You can also navigate to **Topics** to get an overview of all topics defined on your cluster. Clicking on one of the entries will show you similar information as you are used to from Confluent Control Center (Settings, Schema and Inspect)

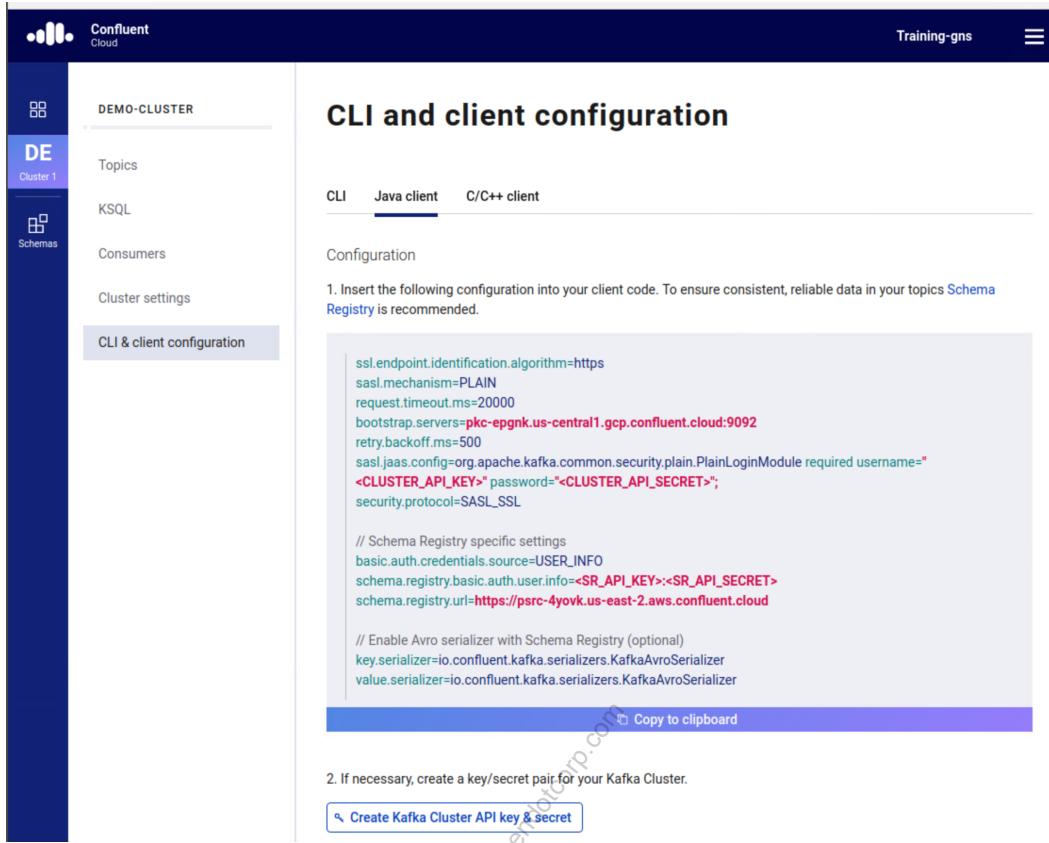
# Confluent Cloud Web Interface - Consumer Lag

The screenshot shows the Confluent Cloud Web Interface for the 'demo-consumer-group' on the 'test-topic' topic. The summary section indicates a total of 391,190 messages behind, with a note that the data is 'Calculating change'. A chart shows the current progress in processing. The detailed table below lists the consumer ID, topic, partition, and lag for each consumer. The consumer ID is repeated for each partition, and the lag values are as follows:

Consumer Id	Topic Partition Topic	Partition	Lag Messages
client-1-cab20bde-9122-4511-bd02-219e16...	test-topic	2	67759
client-1-cab20bde-9122-4511-bd02-219e16...	test-topic	3	58658
client-1-cab20bde-9122-4511-bd02-219e16...	test-topic	4	56709
client-1-cab20bde-9122-4511-bd02-219e16...	test-topic	5	69473
client-1-cab20bde-9122-4511-bd02-219e16...	test-topic	0	68840

The CCloud UI also offers the important capability of monitoring consumer lags.

# Confluent Cloud Web Interface - Clients



**DE Cluster 1**

## CLI and client configuration

CLI   Java client **C/C++ client**

Configuration

1. Insert the following configuration into your client code. To ensure consistent, reliable data in your topics [Schema Registry](#) is recommended.

```
ssl.endpoint.identification.algorithm=https
sasl.mechanism=PLAIN
request.timeout.ms=20000
bootstrap.servers=pkc-epgnk.us-central1.gcp.confluent.cloud:9092
retry.backoff.ms=500
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username="<CLUSTER_API_KEY>" password="<CLUSTER_API_SECRET>";
security.protocol=SASL_SSL

// Schema Registry specific settings
basic.auth.credentials.source=USER_INFO
schema.registry.basic.auth.user.info=<SR_API_KEY>:<SR_API_SECRET>
schema.registry.url=https://psrc-4yovk.us-east-2.aws.confluent.cloud

// Enable Avro serializer with Schema Registry (optional)
key.serializer=io.confluent.kafka.serializers.KafkaAvroSerializer
value.serializer=io.confluent.kafka.serializers.KafkaAvroSerializer
```

[Copy to clipboard](#)

2. If necessary, create a key/secret pair for your Kafka Cluster.

[Create Kafka Cluster API key & secret](#)

If you navigate to your cluster and then **Data In/Out → Clients** you will see the necessary client configurations (including Schema Registry) that you can use when using your cluster with Kafka clients

# Confluent Cloud CLI - Getting Started

## Getting help:

```
$ ccloud help
```

Manage your Confluent Cloud.

Usage:

```
ccloud [command]
```

Available Commands:

api-key	Manage API keys
completion	Output shell completion code
environment	Manage and select ccloud environments
help	Help about any command
kafka	Manage Apache Kafka
login	Login to Confluent Cloud
logout	Logout of Confluent Cloud
service-account	Manage service accounts
update	Update ccloud
version	Print the ccloud version
...	

## Login to your account:

```
$ ccloud login
```

Enter your Confluent Cloud credentials:

Email::: <YOUR\_EMAIL>

Password: <YOUR\_PASSWORD>

Logged in as xyz@confluent.io

Using environment tXXXX ("Training-gns")

## List Environments:

```
$ ccloud environment list
```

## Select your environment:

```
$ ccloud environment use <YOUR_ENVIRONMENT>
```

The tool `ccloud` is the CLI that provides access to your cloud based clusters via command line. You don't have to go far to get help on how to use the tool. Just type `ccloud help` or even better `ccloud help <command>`. You will be provided with a detailed list of all options and their meaning.

By default it uses the configuration settings stored in `~/.ccloud/config`

To login to your account use the command `ccloud login`

Once you're logged in, you can list all available environments and then select the one you want to work with



you can also use `kafka-console-producer|consumer` and `kafkacat` with CCloud - access is **not** limited to the ccloud CLI. Just more security configs are needed for the standard CLI tools.



ACLs/RBAC only supported for **Confluent Cloud Enterprise** at this time, no support in **Confluent Cloud**.

## Confluent Cloud CLI - Manage Topics

List all clusters in your environment

```
$ ccloud kafka cluster list
  Id      ! Name  ! Provider ! Region   ! Durability ! Status
+-----+-----+-----+-----+-----+
  1kc-lovz9 ! Alpha ! gcp      ! europe-west3 ! LOW        ! UP
```

Select your cluster

```
$ ccloud kafka cluster use <CLUSTER_ID>
```

ybhandare@greendotcorp.com

## Confluent Cloud CLI - Manage Topics

List all topics:

```
$ ccloud kafka topic list
demo-topic
other-topic
my-topic
```

Create a new topic **product-topic**:

```
$ ccloud kafka topic create product-topic \
--replication-factor 3 \
--partitions 6
```

Describe topic **product-topic**:

```
$ ccloud kafka topic describe product-topic
Topic: products PartitionCount: 6 ReplicationFactor: 3
  Topic      | Partition | Leader | Replicas |    ISR
+-----+-----+-----+-----+
  product-topic |      0 |      2 | [2 4 9] | [2 4 9]
  product-topic |      1 |      3 | [3 2 0] | [3 2 0]
  product-topic |      2 |      1 | [1 3 8] | [1 3 8]
...
Configuration

  Name          |    Value
+-----+-----+
  compression.type | producer
...
```

We can list, create, update and delete topics using the **ccloud** CLI as shown on the slide.

Alternatively we can do the same from the Web UI, but the CLI approach is preferred for automation.



The tools included with Apache Kafka® such as **kafka-topics**, **kafka-console-consumer**, **kafka-console-producer**, or **kafkacat** can also be used with CCloud.

## Confluent Cloud CLI - Produce and consume data

Producing data:

```
$ ccloud kafka topic produce \
  product-topic
```

Consuming data:

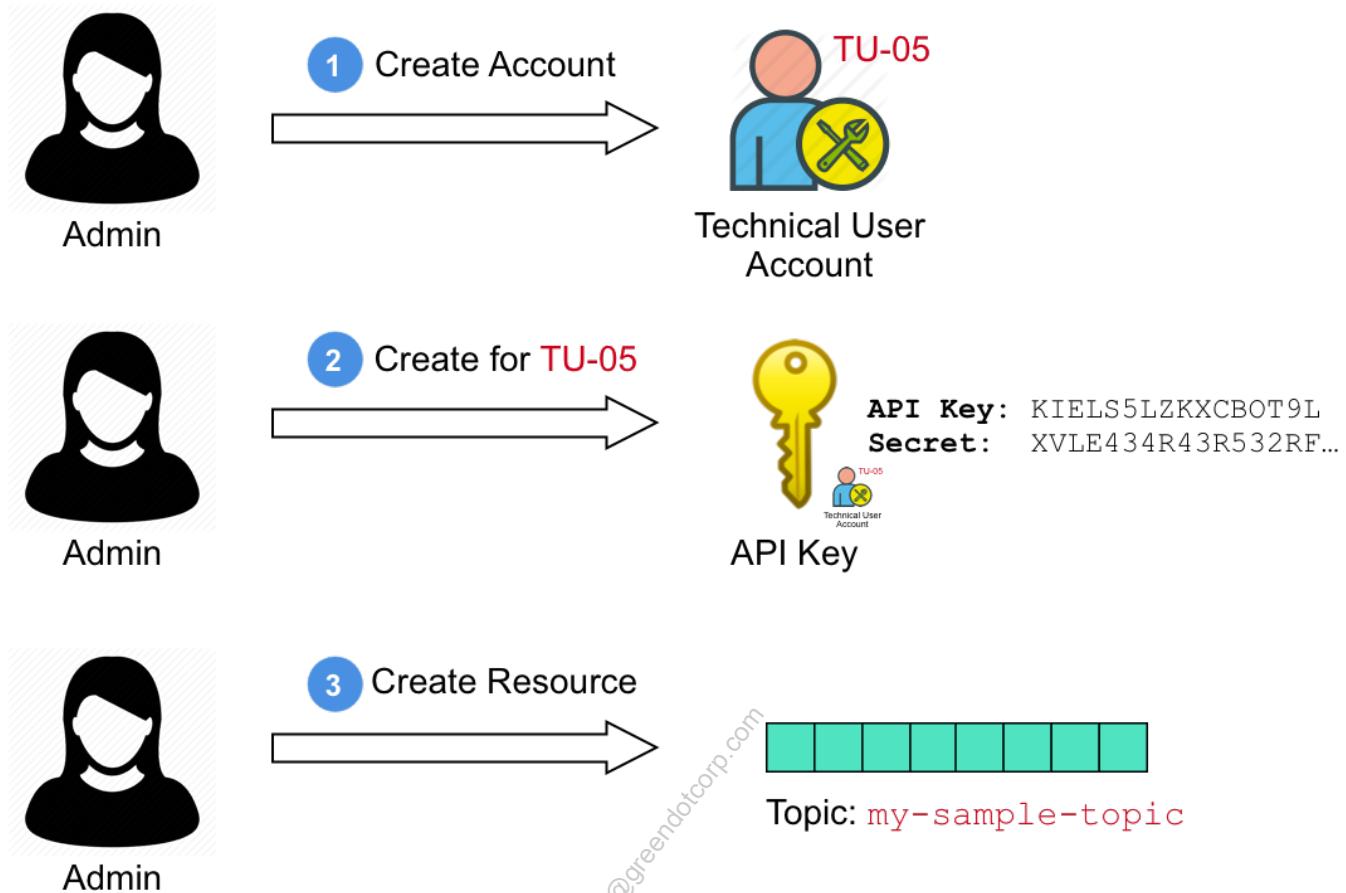
```
$ ccloud kafka topic consume \
  product-topic \
  --from-beginning
```

Add data:

```
1:Kafka
2:is great
3:and I love it
```

```
is great
and I love it
Kafka
```

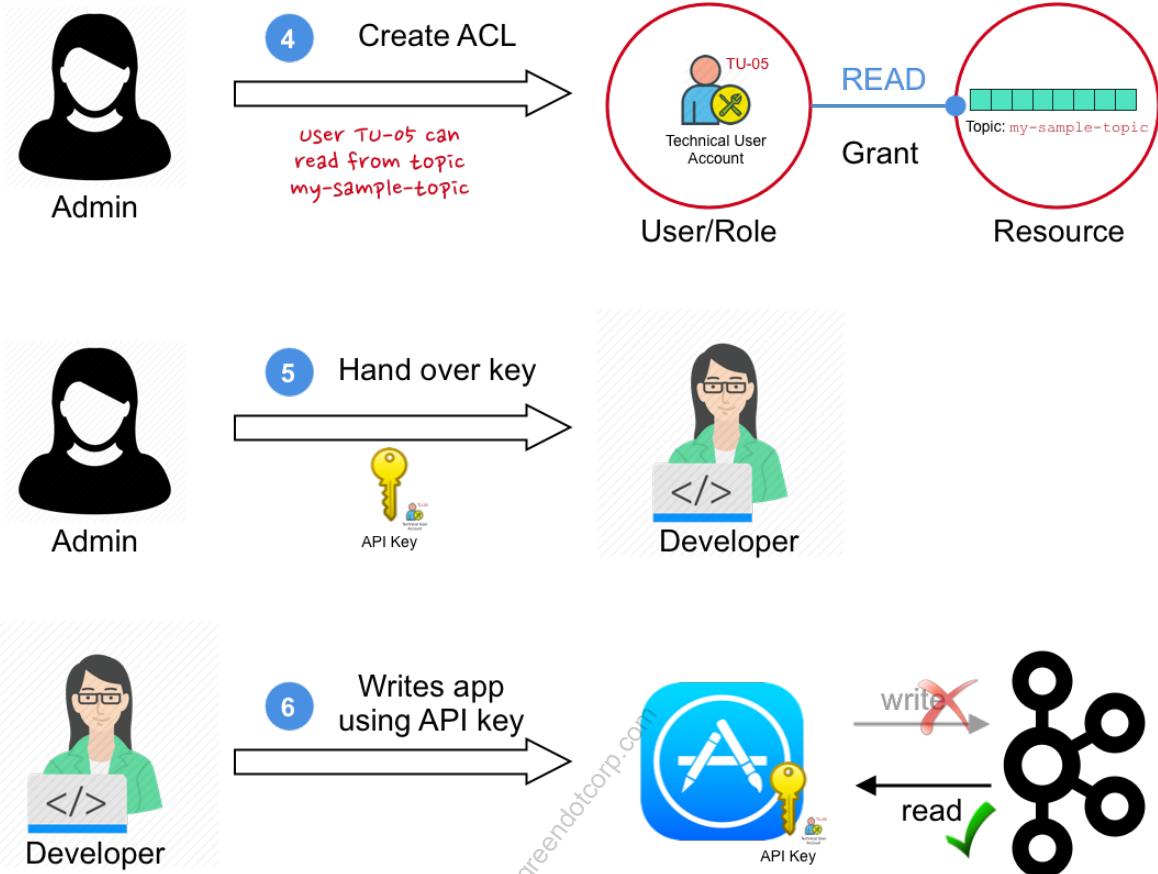
Due to the fact that in the current CLI, `--property` is not supported for the command `consume` we have to use a work-around and put config properties into the environment configuration file. That's why we add `parse.key=true` and `key.separator=`, to the file `~/.ccloud/conf`.



Note that the following only applies to CCE!

1. The admin of your Confluent Cloud creates a technical user account. Here identified with the ID **TU-05**
2. The admin then creates an API-Key and secret for that technical user **TU-05**
3. Then the admin creates resources on Kafka such as this topic **my-sample-topic**

Each of the above operations can be executed by using the CLI **ccloud** (V2)



4. Admin creates ACLs such as:

Grant the technical user account **TU-05** **read** access to the resource **my-sample-topic**

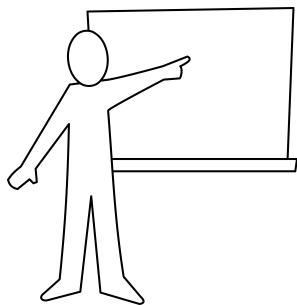
5. Admin hands over the API key and secret to the developer (or operations engineer, depending on who configures the app)

6. The developer writes an application that accesses resources on Kafka using the API key and secret

As we can see on the slide, this app is then only able to read from the topic, but not to write!

## Module Map

---



- Confluent Cloud Overview
- Using the Cloud CLI and Web UI
- Configuring Kafka Clients ... 

ybhandare@greendotcorp.com

## Write a Kafka Avro Producer - Configuration

```
public class ProductProducer {
    public static void main(String[] args) {
        System.out.println("*** Starting VP Producer ***");

        Properties settings = new Properties();
        settings.put(ProducerConfig.CLIENT_ID_CONFIG, "prod-producer");
        settings.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "<BOOTSTRAP_SERVER>");
        settings.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, "20000");
        settings.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG, "500");
        settings.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        settings.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class);
        settings.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SASL_SSL");
        settings.put(SaslConfigs.SASL_MECHANISM, "PLAIN");
        settings.put(SaslConfigs.SASL_JAAS_CONFIG ,
            "org.apache.kafka.common.security.plain.PlainLoginModule " +
            "required username=<API_KEY> password=<API_SECRET>;");

        settings.put("schema.registry.url", "<SR_URL>");
        settings.put("basic.auth.credentials.source", "USER_INFO");
        settings.put("schema.registry.basic.auth.user.info",
            "<SR_KEY>:<SR_SECRET>");
        ...
    }
}
```

We're only showing the configuration part of the code. The remainder of the code is exactly the same as in a (Avro) producer written for the Confluent Platform.

To summarize, you need the following access information and credentials to your cluster in CCloud:

- <BOOTSTRAP\_SERVER>: the URL to the bootstrap server of your hosted Kafka cluster. You can find this info e.g. via the CCloud UI, on the **Clients** view
- <API\_KEY>: The API key to the Kafka cluster
- <API\_SECRET>: The API key secret to the Kafka cluster
- <SR\_URL>: The URL to your hosted Schema Registry
- <SR\_KEY>: The API key to your hosted Schema Registry
- <SR\_SECRET>: The API key secret to your hosted Schema Registry

All this information is available by your Admin or via the CCloud Web UI

## Write a Kafka Avro Producer - Configuration



The **secrets** you have to write down and store safely whilst creating access keys for your Kafka cluster or Schema Registry; they will **not** be accessible from the Web UI once they were created)

For completeness, here is the code to generate a single Avro message based on the schema `product-value.avsc`:

```
final KafkaProducer<String, ProductValue> producer =
    new KafkaProducer<>(settings);

Runtime.getRuntime().addShutdownHook(new Thread(() -> {
    System.out.println("### Stopping prod-producer ###");
    producer.close();
}));

String key = "1";
ProductValue value = new ProductValue(1, "apples", 1.25);
final ProducerRecord<String, ProductValue> record =
    new ProducerRecord<>("products", key, value);
producer.send(record);
```

### Kafka Consumer:

The configuration of a Kafka consumer connecting to your CCloud cluster needs the same connection and authentication tokens as the producer. We are thus not going to repeat the code... here

### Questions:



- What are the main advantages of using Confluent Cloud versus managing a self hosted Confluent Platform?
- Assuming your company decides to use Confluent Cloud. What features and/or possibilities would you miss compared to a self hosted Streaming Platform powered by Kafka?

ybhandare@greendotcorp.com

## Hands-On Lab

---

Please refer to the lab **Using Confluent Cloud** in the Exercise Book



ybhandare@greendotcorp.com

## Further Reading

---

- Security Overview and Recommendations:  
[https://www.confluent.io/wp-content/uploads/CC\\_Security\\_WP.pdf](https://www.confluent.io/wp-content/uploads/CC_Security_WP.pdf)
- What specific security features does Confluent Cloud offer?  
<https://docs.confluent.io/current/cloud/faq.html#what-specific-security-features-does-ccloud-offer>

ybhandare@greendotcorp.com



### Conclusion

ybhandare@greendotcorp.com

## Course Contents

---

After this course, you are now able to:

- Write Producers and Consumers to send data to & read data from Apache Kafka
- Integrate Kafka with external systems using Kafka Connect, Confluent REST Proxy and Confluent MQTT Proxy
- Write streaming applications with Kafka Streams & Confluent KSQL
- Write event driven, semantic applications that integrate with Apache Kafka
- Use and integrate with Confluent Cloud
- Configure your Kafka clients to access a secured Kafka cluster

ybhandari@greendotcorp.com

## Other Confluent Training Courses

- Confluent Operations for Apache Kafka
- Confluent Stream Processing Using KSQL & Apache Kafka Streams
- Confluent Advanced Skills for Optimizing Apache Kafka



For more details, see <https://confluent.io/training>

- **Confluent Operations for Apache Kafka** covers:
  - Data Durability in Kafka
  - Replication and log management
  - How to optimize Kafka performance
  - How to secure the Kafka cluster
  - Basic cluster management
  - Design principles for high availability
  - Inter-cluster design
- **Confluent Stream Processing Using KSQL & Apache Kafka Streams** covers:
  - Installing KSQL containerized and natively
  - Transforming and aggregating data streams with KSQL
  - Writing Kafka Streams App using DSL and Processor API
  - Testing a Kafka Streams App
  - Monitoring Kafka Streams and KSQL Applications
  - Securing Kafka Streams and KSQL Applications
  - Scaling Kafka Streams and KSQL Applications
- **Confluent Advanced Skills for Optimizing Apache Kafka**
  - Monitoring all components of the Confluent Event Streaming Platform (CP)
  - Troubleshooting of all components of CP
  - Tuning the components of CP

## Thank You!

---



- Thank you for attending the course!
- Please complete the course survey
- For any feedback email [training-admin@confluent.io](mailto:training-admin@confluent.io)

- Survey: your instructor will give you details on how to access the survey
- Feedback: Any feedback is welcome. Please do not hesitate to contact us at the given email address.

ybhandare@greendotcorp.com

## Confluent Certified Developer for Apache Kafka

**Duration:** 90 minutes

**Qualifications:** Solid work foundation in Confluent products and 6-to-9 months hands-on experience

**Availability:** Live, online, 24-hours a day!

**Cost:** \$150

**Register online:** [www.confluent.io/certification](http://www.confluent.io/certification)



### Benefits:

- Recognition for your Confluent skills with an official credential
- Digital certificate and use of the official Confluent Certified Developer Associate logo

### Exam Details:

- The exam is linked to the current Confluent Platform version
- 55 multiple choice questions in 90 minutes
- Designed to validate professionals with a minimum of 6 - 12 months of Confluent experience
- Remotely proctored on your computer
- Available globally in English



### Basic Kafka Administration

ybhandare@greendotcorp.com

# Agenda

---



1. Introduction
2. Fundamentals of Apache Kafka
3. Kafka's Architecture
4. Developing with Apache Kafka
5. Advanced Development with Apache Kafka
6. Schema Management in Apache Kafka
7. Data Pipelines with Kafka Connect
8. Stream Processing with Kafka Streams
9. Stream Processing with KSQL
10. Event Driven Architecture
11. Confluent Cloud
12. Conclusion
13. Appendix: Basic Kafka Administration ... ←
14. Appendix: Apache Kafka driving ML & Data Analytics

ybhandare@greendotcorp.com

## Learning Objectives

---



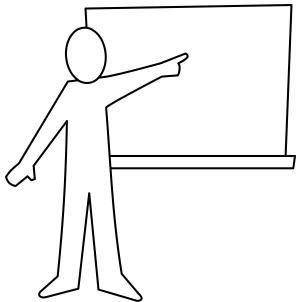
After this module you will be able to:

- Set up Kafka components
- Perform some common Kafka administrative tasks
- Explain how Compacted Logs work
- Determine the number of Partitions to specify for a Topic
- List what security features Kafka provides

ybhandare@greendotcorp.com

# Module Map

---



- Kafka Versions and Software Upgrades ... ←
- Administering Kafka
- Log Management
- Determining How Many Partitions to Specify
- Kafka Security

ybhandare@greendotcorp.com

## Available Package Formats

---

- Confluent provides Kafka in different formats
  - Available as deb, RPM, Zip archive, tarball, Docker images
- Java 8 is required
- If you are running on a Mac, use the Zip or Tar archive
- Running Kafka on Windows may prove problematic
- For development purposes, Confluent CLI can be used as a single host solution
  - <https://docs.confluent.io/current/cli/>

Each download of Confluent Open Source includes all of the components. Not all components are intended to be run on the same system (e.g., the Broker and the REST Proxy) - just download the entire package and activate the components appropriate for that system.

Support for Java 7 ended as of AK 1.1. As of Kafka 1.0, Java 9 is supported but not by Confluent Open Source components, like Schema Registry or REST Proxy. Java 11 is supported beginning with CP 5.2.

ybhandare@greendotcorp.com

## Client and Broker Version Compatibility

- Clients are forwards and backward compatible with Brokers
  - Assumes a minimum broker version of Kafka 0.10.0
  - Non-Java clients must be a minimum version of Kafka 0.10.0
- Some features may not be available if the clients and servers are running different versions

For the latest compatibility information, refer to <https://cwiki.apache.org/confluence/display/KAFKA/Compatibility+Matrix>

Note: Streams API version compatibility is different. Streams API (starting in 0.10.2) needs Brokers to be running at least Kafka 0.10.1 (Confluent 3.1)

If you have customers running older versions, the interoperability rules were more complicated:

- Clients can communicate with Brokers running newer (or same) versions of Kafka
  - Example: a Kafka 0.10.0 (Confluent 3.0) Producer can communicate with a Kafka 0.10.1 (Confluent 3.1) Broker
- Clients can communicate with Brokers running older versions of Kafka
  - Example: a Kafka 0.10.2 (Confluent 3.2) Producer can communicate with a Kafka 0.10.1 (Confluent 3.1) Broker
  - Clients must be running at least Kafka 0.10.2 (Confluent 3.2)
  - Brokers must be running at least Kafka 0.10.0 (Confluent 3.0)

## Upgrading Kafka

---

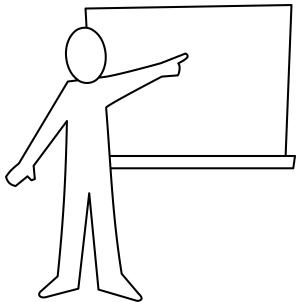
- Kafka supports rolling upgrades
  - No downtime for end users
  - Clients and brokers can be upgraded independently
- Refer to the release notes for version specific instructions
  - <http://kafka.apache.org/documentation.html#upgrade>

The rolling upgrade is only recommended if your mission-critical Topics are replicated. Without replication, taking a Broker down for upgrade will make any un-replicated Partition which is resident on that Broker unavailable.

ybhandare@greendotcorp.com

## Module Map

---



- Kafka Versions and Software Upgrades
- Administering Kafka ... ←
- Log Management
- Determining How Many Partitions to Specify
- Kafka Security

ybhandare@greendotcorp.com

## Administering Kafka - Introduction

---

- Note that we only cover a few common administrative functions here
- For much more in-depth coverage, consider attending *Confluent Operations Training for Kafka*

ybhandare@greendotcorp.com

## Configuring Topics

---

- Methods to create Kafka Topics:
  - Automatic creation
  - Kafka Client APIs
  - Command line
  - Confluent Control Center

ybhandare@greendotcorp.com

## Configuring Topics - Automatic Creation

---

- By default, Topics are automatically created when they are first used by a client
  - Auto-created Topics will have a single Partition and a single replica, by default
  - In production environments, consider disabling automatic Topic generation with the `auto.create.topics.enable` boolean in the broker configurations

By default, Kafka Brokers allow automatic Topic creation. If a Topic is automatically created, it will be configured according to the defaults in the `server.properties` file of the Brokers. The default setting for `default.replication.factor` is 1 - change it to an appropriate value for your environment.

ybhandare@greendotcorp.com

## Configuring Topics - Client API

---

- Supports managing topics from within your applications
- Included as an API with selected clients
  - Java
  - C
  - Python
  - Golang

Minimum broker version is 0.10.0.0. Java client (AdminClient) support started with 0.11.0.0; other clients (TopicAdmin) supported as of 2.0.0.

ybhandare@greendotcorp.com

## Configuring Topics - Command Line

- Create Topics manually on the command line with `kafka-topic`
  - Allows you to set the replication factor and number of Partitions

```
$ kafka-topics \
  --bootstrap-server kafka-1:9092 \
  --create \
  --topic my_topic \
  --partitions 6 \
  --replication-factor 3
```

- Change the number of partitions in a Topic

```
$ kafka-topics \
  --bootstrap-server kafka-1:9092 \
  --alter \
  --topic my_topic \
  --partitions 40
```

- No data is moved from existing Topics



Changing the number of Partitions could cause problems for your application logic!

If manually creating a topic, you can specify the number of partitions and replication factor. Notice that we do not specify which brokers to use - in most cases it is best to let Kafka decide where to place replicas.

Kafka only allows partitions to be added to a Topic; the number of partitions cannot be reduced, since that would technically count as losing data since committed messages are permanently associated with a specific partition.

Additionally, changing the number of partitions will break the guarantee of ordering for semantic partitioning (since the number of partitions is part of the formula). For the workaround (new Topic, copy data), you will have to update all of the clients. Since you will be copying data from the original Topic into a new (larger) Topic in the same cluster, the Topics cannot have the same name. There currently is not a way to rename a Topic.

# Configuring Topics - Confluent Control Center

ALL TOPICS >  
**products**

Overview Messages Schema Configuration

Topic name\* products

Number of partitions\* 1

Cleanup policy

Cleanup policy\* Delete

Time to retain data 1 week

Max size on disk in GB Not set

Message size

Maximum message size in bytes 1000012 bytes

Save changes Cancel

**TOPIC SUMMARY**

- name** products
- partitions** 1
- replication.factor** 1
- cluster** 1
- min.insync.replicas** 1
- cleanup.policy** delete
- retention.ms** 60480000
- retention.bytes** -1
- max.message.bytes** 1000012

A Confluent Enterprise license is required to use Confluent Control Center. The ability to manage topics was added in Confluent Enterprise 5.0.

## Administering Kafka - Deleting Topics

- Topic deletion is enabled by default on Brokers
  - `delete.topic.enable` (Default: true)
- Caveats
  - Stop all Producers/Consumers before deleting
  - All Brokers must be running for the `delete` to be successful

```
$ kafka-topics \
--bootstrap-server kafka-1:9092 \
--delete \
--topic my_topic
```

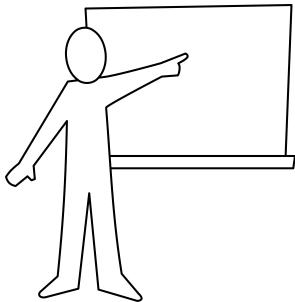
By default, Kafka clusters allow auto Topic creation. This will require that all clients are stopped prior to deleting a Topic – otherwise the Topic will be recreated as soon as a client tries to access it.

When a Topic is deleted, the entries in the `offsets` topic that are associated with that Topic are not immediately removed. We rely on the messages to be garbage collected over time associated with the time retention policy. The issue is, if you recreate the Topic before the retention time has passed and use the same Consumer Group, the offset won't be valid – it will be related to the old messages. So if you delete a Topic, don't recreate it too quickly.

Prior to Kafka 1.0, the default for `delete.topic.enable` was false. To enable topic deletion, the variable needed to be changed in the `server.properties` file on every Broker in the cluster and required a restart.

# Module Map

---



- Kafka Versions and Software Upgrades
- Administering Kafka
- Log Management ... ←
- Determining How Many Partitions to Specify
- Kafka Security

ybhandare@greendotcorp.com

## Log Retention

- Duration default: messages will be retained for seven days
- Duration is configurable per Broker by changing one of, in order of priority:
  - `log.retention.ms`
  - `log.retention.minutes`
  - `log.retention.hours`
- A Topic can override a Broker's configured duration with the property `retention.ms`
- There is no default limit on the size of the log
  - You can set this with `retention.bytes`

Logs are not deleted on consumption because Kafka is multi-subscription. Instead, the Brokers use the retention policy to decide how long to keep the messages. This can be set at the Broker or Topic level.

Cleanup does not affect the currently active segment file (i.e., the segment file which is actively receiving writes from the Producers). Segment files are rolled (i.e., the active segment file is closed and a new segment file is created to receive new data) after a message is appended if one of the following conditions is met:

1. the log segment is full (default 1Gb, configurable with `log.segment.bytes`)
2. the log segment has been open for more than a certain amount of time (default 7 days, configurable with `log.roll.hours`) to ensure that retention can delete or compact old data.  
Relevant for use cases with low data rates
3. the index that maps offsets to file positions is full

The default `delete` policy will retain messages based on time (age) or amount of disk space. The Brokers check the partition directories on a schedule (`log.retention.check.interval.ms`) for files to delete: either the set of segment files exceeds a size (`log.retention.bytes`) or all the messages in a segment file exceed an age (`log.retention.ms`). When this cleanup policy runs, it can only delete whole segment files.

## Log Compaction (1)

- When cleaning up a log, the default policy is `delete`
  - Removes log segments where all messages are older than the `retention.ms` or until the log size is below `retention.bytes` (or both)
- An alternate policy is `compact`
  - A compacted log retains at least the last known message value for each key within the Partition
  - Example usage scenarios
    - A database change log
    - Storing state for external applications
- Configure the retention policy with the Broker-level `log.cleanup.policy` configuration parameter
  - Use the `cleanup.policy` configuration parameter to override at the Topic level
- Both `delete` and `compact` can be enabled at the same time in a single policy

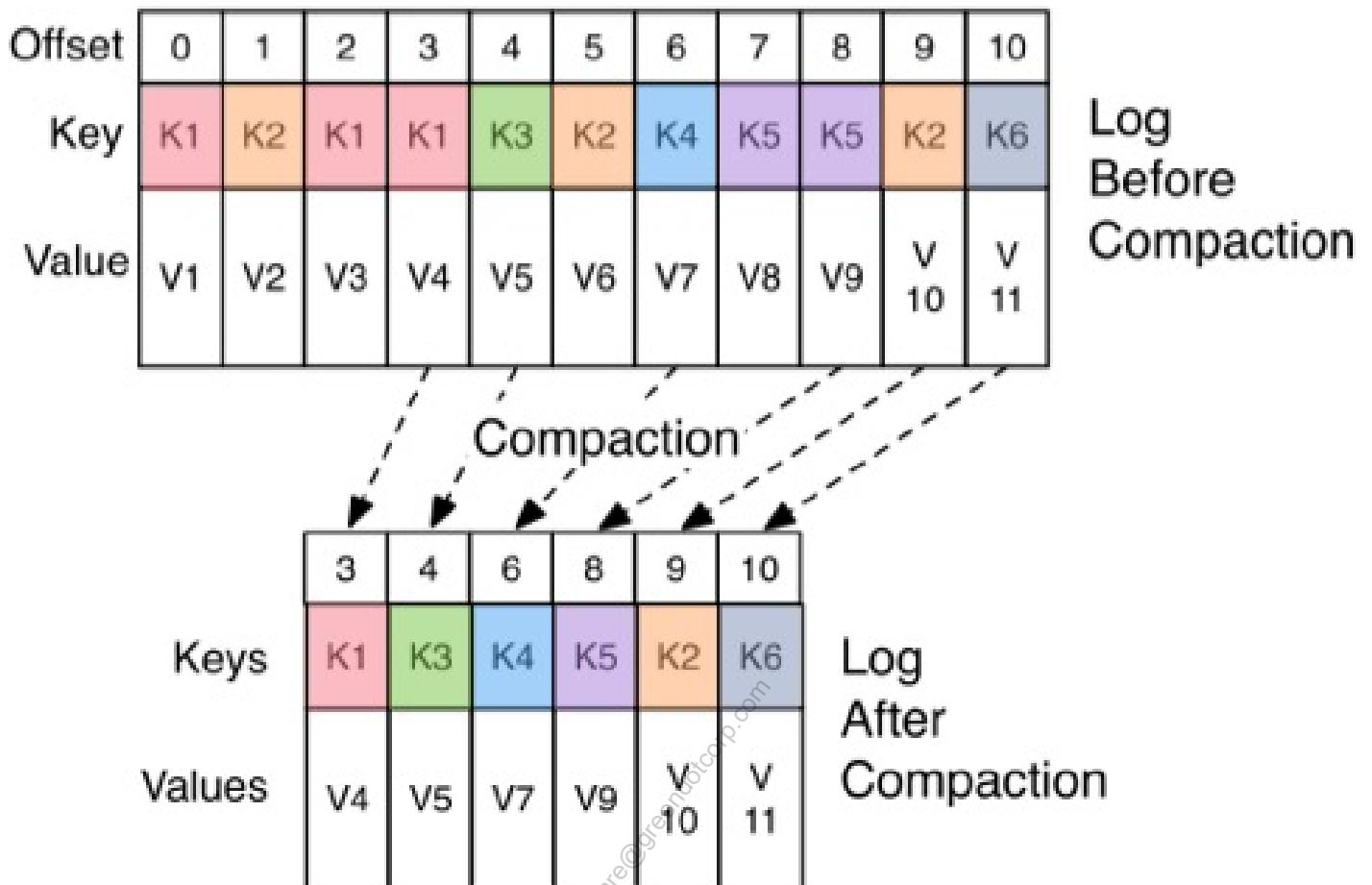
The `compact` cleanup policy is specific to Kafka. It is designed for specific data flows which contains keyed messages in which we only care about the most recent version; i.e., each key message replaces the one that came before it.

Sample use cases:

- \* Database change capture: maintain a replica of the data stream by key (e.g., a search index receiving real-time updates but needing only the most recent entry)
- \* Stateful stream processing: journaling arbitrary processing for high availability (e.g., Samza or anything with "group by"-like processing)
- \* Event sourcing: co-locates query processing with application design and uses a log of changes as the primary store for the application

As of Kafka 0.10.1 (Confluent 3.1), policy supports both `delete` and `compact` to be enabled at the same time. This is useful if your use case generates keys that eventually go stale and are not updated, filling up your available disk space.

## Log Compaction (2)



How compaction works: the original log has the keyed messages assigned to offsets. Some of the messages have the same key. After compaction, we'll just preserve the last message w/ that key; older messages with that key won't be preserved.

The **compact** policy only works for keyed messages. The policy retains the last message associated with a given key. Over time, older messages with a given key will be deleted. Since compaction guarantees to preserve the latest message with a given key, reading from the start of the log guarantees that a Consumer ends up with the latest version. Technically, this is the same behavior that you would see if the topic was not compacted, but compaction means that we have less repeated messages to read to arrive at the same end value.

## Log Compaction (3)

- It is possible that multiple values may exist in the log for a given key
  - If new values are written after the most recent cleanup
- Clients should read the entire log to ensure they have the latest value for each key
- You can configure how aggressive log compaction is
  - `log.cleaner.min.cleanable.ratio`
    - Ratio of “dirty” (un-compacted) log to total log size (excluding the currently active log segment)
    - Trigger log clean if the ratio of dirty/total is larger than this value
    - Default is `0.5`
    - A Topic can override a Broker’s configured ratio with the property `min.cleanable.dirty.ratio`
- Question: if a Consumer offset points to a message that has been compacted away, does `auto.offset.reset` apply?

Answer: the offset remains a valid position in the log, even if the message with that offset has been compacted away. So the parameter `auto.offset.reset` (for invalid offsets) does not apply, the offset is still technically valid. This position (offset) is indistinguishable from the next highest offset that does appear in the log. Since Consumer Offset is the value of the next message the Consumer will read, not the last message that has been read, it will just read the next highest offset that does appear in the log

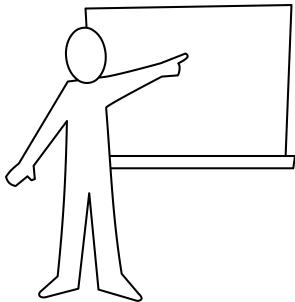
Common log cleaner tunables:

- \* `log.cleaner.min.cleanable.ratio`: trigger the log cleaner when the percentage of dirty data exceeds this value. Defaults to 50%.
- \* `log.cleaner.io.max.bytes.per.second`: throttles the amount of system resources that the cleaner can use. Due to the amount of reads and writes, cleaning is I/O intensive. Default is infinite.

Adjust these parameters carefully - more frequent log cleaning means more I/O time, higher disk utilization.

# Module Map

---



- Kafka Versions and Software Upgrades
- Administering Kafka
- Log Management
- Determining How Many Partitions to Specify ... ←
- Kafka Security

ybhandare@greendotcorp.com

## Specifying the Number of Partitions

- If a Topic is automatically created, it will have a single Partition and a single replica, by default
- If a Topic is manually created, you can specify the number of Partitions, and the number of replicas for each Partition

```
$ kafka-topics \
  --bootstrap-server kafka-1:9092 \
  --create \
  --topic my_topic \
  --partitions 6 \
  --replication-factor 3
```

- The number of Partitions can have an impact on performance

When selecting the number of partitions for a topic, the limiting factor is likely to be the consumers. Topics should be sized so that consumers can keep up with the throughput from a physical (NIC speed) and computational (processing time per poll) standpoint.

ybhandare@greendotcorp.com

## More Partitions → Higher Throughput

- Parallelization
  - Writes to Partitions by the Producer can be performed in parallel
    - Therefore, more Partitions can result in better throughput when sending data to the cluster
  - Reads from Topics by a Consumer Group are bounded by the number of Partitions
    - If you have more Consumers in a Consumer Group than you have Partitions in a Topic, some Consumers will receive no data for that Topic

One of the most common questions asked about Kafka is "How many partitions should my Topic have?"

There is no simple answer. More partitions generally means higher throughput (assuming you have enough Consumers assigned to all the partitions). However, there are downsides to arbitrarily large partition counts that we will discuss on the next slide.

The current limits (2-4K Partitions/Broker, 100s K Partitions per cluster) are maximums. Most environments are well below these values (typically in the 1000-1500 range or less per Broker). Kafka 2.0 enhancements increased these effective limits from previous versions:

<https://issues.apache.org/jira/browse/KAFKA-5642> [https://docs.google.com/document/d/1rLDmzDOGQQeSiMANP0rC2RYp\\_L7nUGHzFD9MQISgXYM/edit](https://docs.google.com/document/d/1rLDmzDOGQQeSiMANP0rC2RYp_L7nUGHzFD9MQISgXYM/edit)

For people still looking for an exact number, they need to measure per-partition throughput on a Producer and on a Consumer, and then factor that into their desired maximum total throughput.

Also a common practice is to over-partition a bit to accommodate growth over next 1-2 years. This will prevent the scenario where added Partitions result in keyed-messages going to new Partitions.

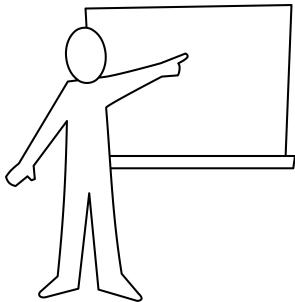
## The Downside of Many Partitions

- Partition unavailability in the event of Broker failure
  - If a Broker fails, and it is a leader for some number of Partitions, each of those Partitions will become unavailable until a new leader is elected
    - The more Partitions for which a Broker is a leader, the longer this will take
- End-to-end latency (time from Producing a message to it being read by a Consumer)
  - A Broker uses a single thread to replicate data from another Broker with which it shares Partitions
    - The more Partitions there are, the longer this can take
  - Alleviated on larger clusters where replicas are distributed amongst more Brokers
- Memory requirements (buffers on clients are per Partition)
  - More Partitions can lead to larger memory requirements on the client
    - The limit on how much data is sent back to the client is a per-Partition limit
    - Producers may locally cache messages for batching purposes
  - Mitigated with `max.poll.records` to reduce the memory requirements

Downsides with too many Partitions: \* You need more open file handles: More Partitions means more directories and segment files on disk. \* Availability issue: Planned failures move Leaders off of a Broker one at a time, with minimal downtime per partition. In a hard failure all the leaders are immediately unavailable. The Controller needs to detect the failure and choose other leaders, but since this happens one at a time, so it can take a long time for them all to be available again. The first partition will be offline for significantly less time than the nth partition. Additionally, if the Controller itself fails, the first thing that has to happen is to failover the Controller. The new Controller needs to initialize by reading a lot of metadata, i.e., the metadata for all Partitions in the cluster. The more Partitions, the longer this recovery takes. \* Latency impact: only really matters when you're talking about millisecond latency. For the message to be seen by a Consumer it must be committed. The Broker replicates data from the leader with a single thread, resulting in overhead per Partition. If you have 1000 Partitions the overhead is about 20 milliseconds. \* Client Memory: Both the Producer and Consumer buffer per-Partition. Increasing the number of partitions would increase the memory requirements on the clients. \* Resizing: A Topic can be expanded if it was created with too few partitions. Topics cannot reduce the number of Partitions they contain.

# Module Map

---



- Kafka Versions and Software Upgrades
- Administering Kafka
- Log Management
- Determining How Many Partitions to Specify
- Kafka Security ...

ybhandare@greendotcorp.com

## Encryption, Authentication, Authorization

- Encryption of Data in Transit
- Client Authentication
- Client Authorization



### Encryption in transit

SSL



### authn & authz

authn: SASL or SSL  
authz: ACLs

Some of the relevant security features are:

- **Encrypt data-in-transit between Kafka Stream applications and Kafka brokers:** You can enable the encryption of the client-server communication between your applications and the Kafka brokers. For example, you can configure your applications to always use encryption when reading and writing data to and from Kafka. This is critical when reading and writing data across security domains such as internal network, public internet, and partner networks.
- **Client authentication:** You can enable client authentication for connections from your application to Kafka brokers. For example, you can define that only specific applications are allowed to connect to your Kafka cluster. Authentication can be done using mutual TLS (SSL) or SASL.
- **Client authorization:** You can enable client authorization of read and write operations by your applications. For example, you can define that only specific applications are allowed to read from a Kafka topic. You can also restrict write access to Kafka topics to prevent data pollution or fraudulent activities. Authentication in Kafka is done via ACLs

Security was added to Kafka in 0.9.0.

# Encryption and Authentication

- Kafka Brokers can listen on multiple ports:
  - Plain text (no wire encryption or authentication)
  - SSL (wire encryption and optional SSL authentication)
  - SASL: Simple Authentication and Security Layer
    - GSSAPI: Kerberos
    - PLAIN: cleartext username/password
    - SCRAM-SHA-256, SCRAM-SHA-512: “salted” and hashed passwords
    - OAUTHBEARER: authentication tokens
  - SSL + SASL (wire encryption and Kerberos authentication)
- Clients choose which ports to use
  - Required credentials are provided through configuration
- Note: encryption is across the wire only
  - No encryption at rest

Kafka brokers can listen on multiple ports at the same time so that different clients can be authenticated appropriately. Clients must choose one port; they cannot failover if their preferred login method is unavailable.

Kerberos (introduced in Kafka 0.9.0/Confluent 2.0) is frequently used because it enables single sign-on. However, as with SSL, there are no Kafka-specific changes that need to be made to the servers. It is beyond the scope of this course to set up the Kerberos environment.

Plain (introduced in Kafka 0.10.0/Confluent 3.0) is the easiest of the SASL options but is not generally used in production environments for reasons that will be discussed later.

The SCRAM implementation (introduced in Kafka 0.10.2/Confluent 3.2) is described in RFC 5802. Support for other SCRAM mechanisms like SHA-224 and SHA-384 are not supported at this time. Strong hash functions combined with strong passwords and high iteration counts protect against brute force attacks if Zookeeper security is compromised. Support for delegation tokens was added in Kafka 2.0.

OAUTHBEARER (introduced in Kafka 2.0/Confluent 5.0) is a self-service token based authentication method. It uses unsecured JSON web tokens by default and should be considered non-production without additional configuration.

## SSL Performance Impact

- Performance was measured on Amazon EC2 r3.xlarge instances
  - Note that these performance tests were not run with Java 9, which has significant performance improvement

	Throughput(MB/s)	CPU on client	CPU on Broker
<b>Producer (plaintext)</b>	83	12%	30%
<b>Producer (SSL)</b>	69	28%	48%
<b>Consumer (plaintext)</b>	83	8%	2%
<b>Consumer (SSL)</b>	69	27%	24%

In an SSL connection, the Producer encrypts the messages before sending to the Brokers. The Brokers decrypt the messages for local storage and then re-encrypt them before sending to the Consumers, who then have to decrypt the messages again.

Note: SSL only provides wire encryption. While the data is resident on the Brokers, it is stored unencrypted. There is currently no native at-rest encryption scheme in Kafka. Currently, customers who require at-rest encryption either use encrypted hardware or encrypt the data before creating the messages at the Producer.

All of this processing will affect end-to-end performance and CPU utilization on all components of the Kafka architecture.

For versions of Kafka and Confluent Open Source which support Java 9, there are significant performance improvements in TLS overhead with Java 9.

These performance numbers are specific to the described environment - actual numbers will vary. The increase in CPU utilization for the consumer connection seems excessive but that is just because those activities require so little CPU in normal circumstances.

## Authorization Using ACLs

- Kafka supports authorization using Access Control Lists (ACLs)
- Common use cases:
  - Allow Producers to write to certain Topics
  - Allow Consumers to read from certain Topics
  - Allow certain users to issue a controlled shutdown
- ACLs are enabled in the Broker's configuration file
- ACLs are configured via the `kafka-acls` command-line tool
  - Example: allow user Bob to write to and read from Topic `my_topic`

```
$ bin/kafka-acls \  
  --authorizer-properties zookeeper.connect=zookeeper1:2181 \  
  --add \  
  --allow-principal User:Bob \  
  --operation Read \  
  --operation Write \  
  --topic my_topic
```

The default authorization plugin implements permissions based on Access Control Lists (ACLs). Authorization is based on a 5-tuple match - we will examine each part individually.

The default authorization plugin only support user-based permissions. For full LDAP integration (i.e., group permissions), Confluent Enterprise includes an additional authorization plugin as of 5.0.0.

Once the first ACL is created, the cluster will deny all access that is not explicitly allowed by the ACLs. This is standard security practice since it is easy to identify users who are accidentally locked out of their objects; users granted too much access rarely report the misconfigurations.

## Securing Confluent Schema Registry

---

- Confluent Schema Registry can be configured with security
- Secure communication and authentication between the Schema Registry and the Kafka cluster
  - SSL
  - SASL
- Secure communication for end-user REST API calls (HTTPS)
  - SSL
- Authentication with ZooKeeper
  - SASL
- The Schema Registry security plugin restricts schema evolution to administrative users
  - Client application users get read-only access only
- Details at <http://docs.confluent.io/current/schema-registry/docs/config.html>

Since Confluent Platform 4.0: Schema Registry security plugin restrict schema evolution to administrative users

ybhandare@greendotcorp.com

## Securing Confluent REST Proxy

---

- Secure communication between REST clients and Confluent REST Proxy (HTTPS)
  - SSL
  - REST proxy security plugin propagates client principal authentication to Kafka brokers
- Secure communication between the REST Proxy and Apache Kafka
  - SSL
  - SASL
- Authentication with ZooKeeper
  - SASL
  - Client application users get read-only access only
- Details at <http://docs.confluent.io/current/kafka-rest/docs/config.html>

As of Kafka 0.10.2 (Confluent 3.2), the REST Proxy can be configured with security

Since Confluent Platform 4.0: REST proxy security plugin propagates client principal authentication to Kafka brokers

ybhandare@greendotcorp.com

- Topics can be manually created, modified and deleted
- Compacted logs are useful in some scenarios
- Increasing the number of Partitions for a Topic can improve performance in some situations
- Kafka supports encryption, authentication, and authorization



ybhandare@greendotcorp.com



**Apache Kafka driving ML & Data Analytics**

ybhandare@greendotcorp.com

# Agenda

---



1. Introduction
2. Fundamentals of Apache Kafka
3. Kafka's Architecture
4. Developing with Apache Kafka
5. Advanced Development with Apache Kafka
6. Schema Management in Apache Kafka
7. Data Pipelines with Kafka Connect
8. Stream Processing with Kafka Streams
9. Stream Processing with KSQL
10. Event Driven Architecture
11. Confluent Cloud
12. Conclusion
13. Appendix: Basic Kafka Administration
14. Appendix: Apache Kafka driving ML & Data Analytics

... ↙

ybhandare@greendotcorp.com

## Learning Objectives

---

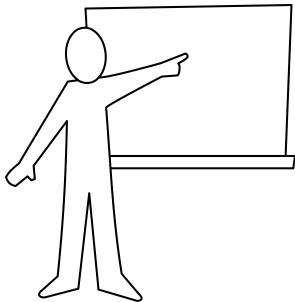


After this module you will be able to:

- Use Kafka to resolve the impedance mismatch between data analysts, data engineers and production engineers
- Use KSQL to filter, transform and enrich your model data
- Use Python and Jupyter Notebooks with KSQL to analyze your data

ybhandare@greendotcorp.com

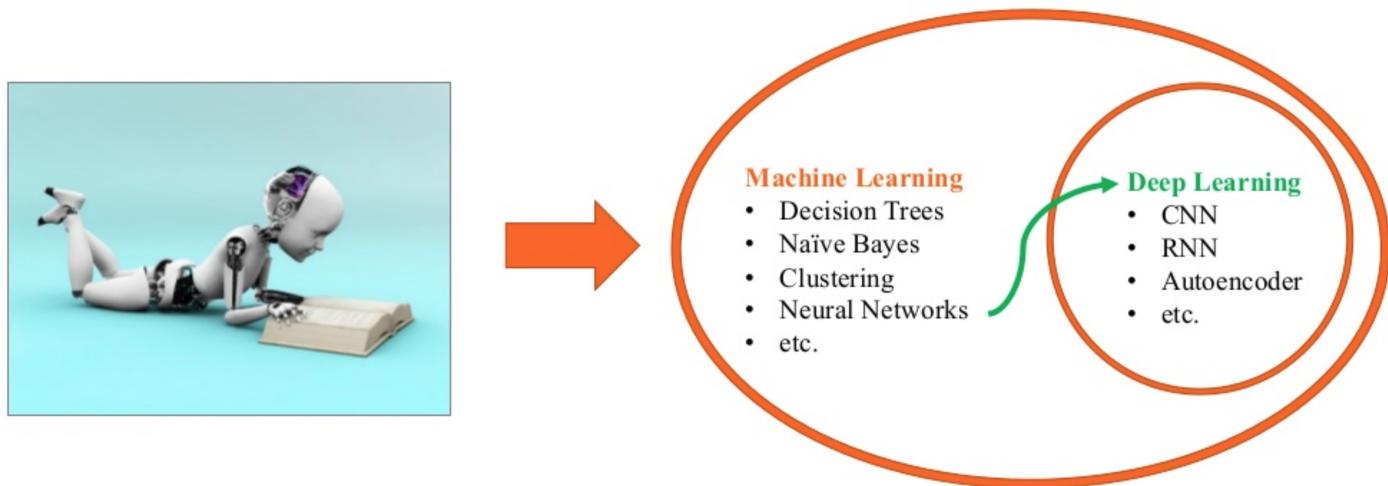
## Module Map



- Kafka and Machine Learning... ←
- KSQL and KSQL UDFs enabling ML
- Data Analysts using Jupyter & KSQL

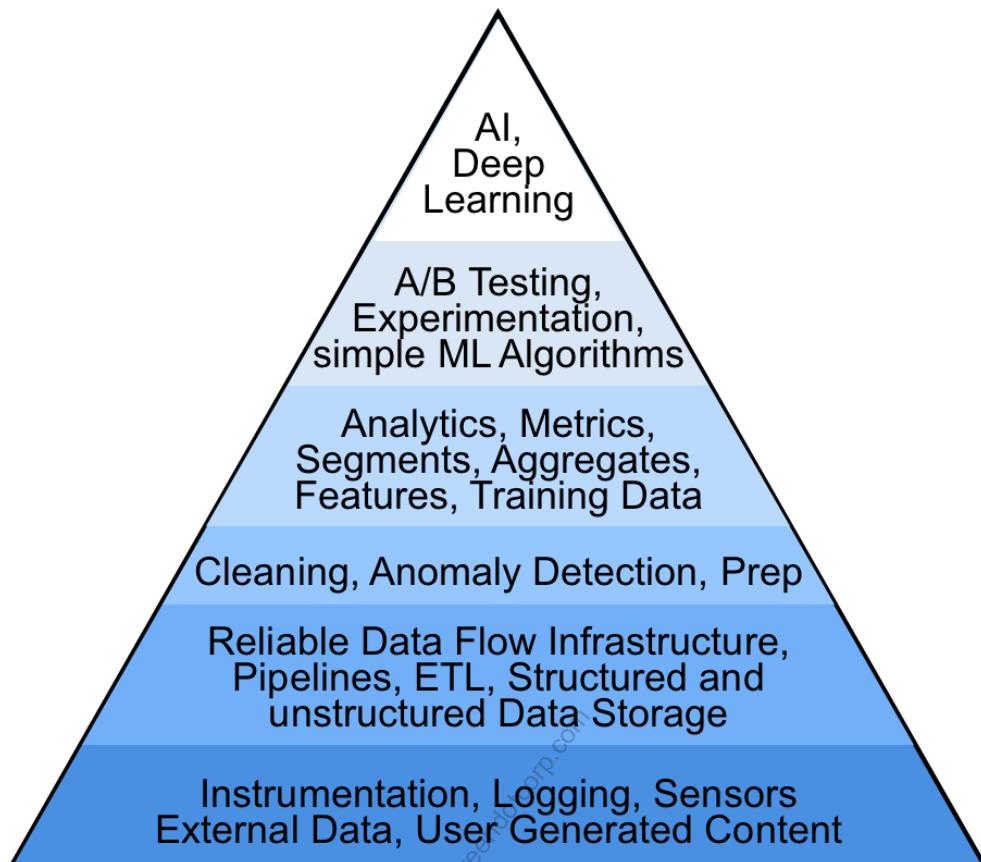
ybhandare@greendotcorp.com

... allows computers to find hidden insights without being explicitly programmed where to look.



- **Naïve Bayes:** In machine learning, naïve Bayes classifiers are a family of simple "probabilistic classifiers" based on applying Bayes' theorem ([https://en.wikipedia.org/wiki/Bayes'\\_theorem](https://en.wikipedia.org/wiki/Bayes'_theorem)) with strong (naïve) independence assumptions between the features.
- **CNN:** In deep learning, a convolutional neural network is a class of deep neural networks, most commonly applied to analyzing visual imagery. CNNs are regularized versions of multilayer perceptrons (see <http://deeplearning.net/tutorial/mlp.html>).
- **RNN:** A recurrent neural network is a class of artificial neural network where connections between nodes form a directed graph along a temporal sequence. This allows it to exhibit temporal dynamic behavior. Unlike feedforward neural networks, RNNs can use their internal state to process sequences of inputs.
- **Autoencoder:** An autoencoder is a type of artificial neural network used to learn efficient data codings in an unsupervised manner. The aim of an autoencoder is to learn a representation (encoding) for a set of data, typically for dimensionality reduction, by training the network to ignore signal "noise".

## The Data Science Hierarchy of Needs



Monica Rogati, one of LinkedIn's first data scientists, does an excellent job of summarizing the difficulties of building machine learning and AI systems in her recent article "The AI Hierarchy of Needs." (<https://hackernoon.com/the-ai-hierarchy-of-needs-18f111fcc007>) Just like Maslow characterizes human needs in a layered pyramid going from the most basic (food, clothing, and shelter) to the highest level (self-actualization), Monica describes a similar hierarchy for AI and machine learning projects. Her hierarchy looks like the one presented on the slide.

## Real World Examples of ML



Spam Detection



Search Results +  
Product Recommendation



Picture Detection  
(Friends, Locations, Products)



The Next Disruption:  
Google Beats Go Champion



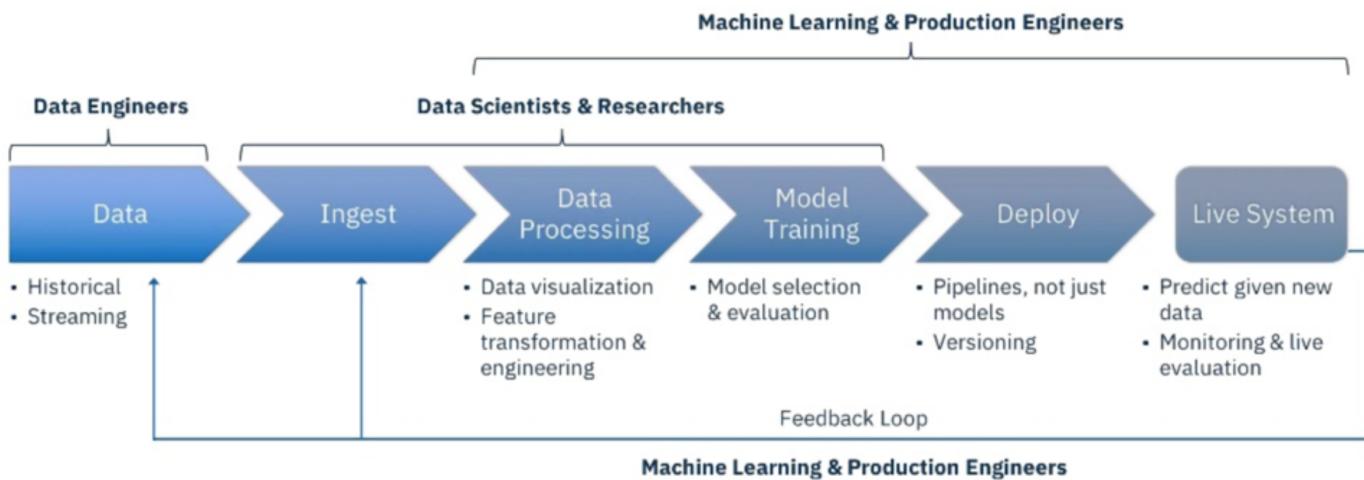
Your Company

Here we see a few use cases or "real world examples of machine learning".

ybhandare@greendotcorp.com

# Impedance Mismatch

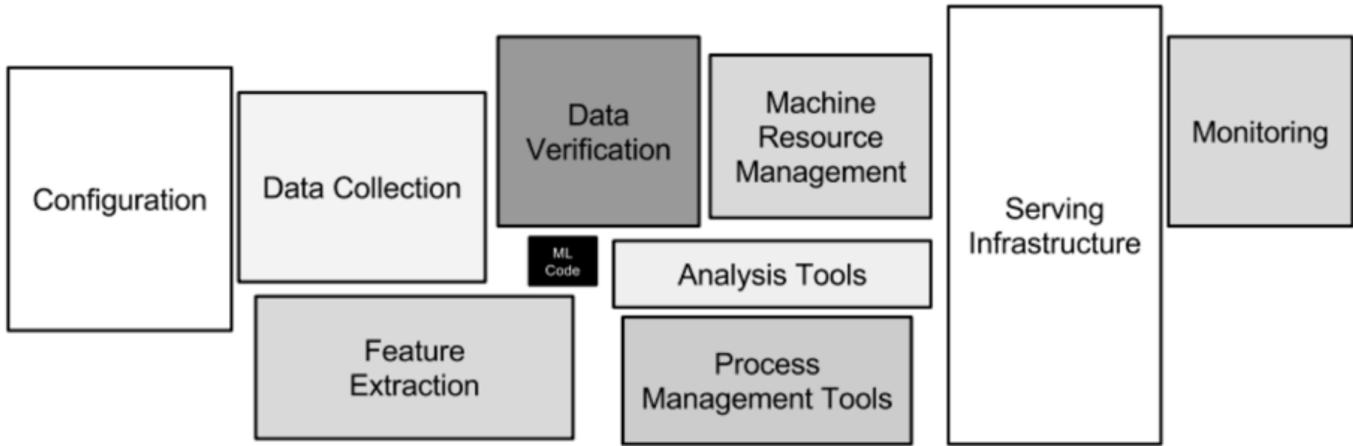
data scientists  $\leftrightarrow$  data engineers  $\leftrightarrow$  production engineers



Based on what we've seen in the field, an impedance mismatch between data scientists, data engineers and production engineers is the main reason why companies struggle to bring analytic models into production to add business value.

The diagram on the slide illustrates the different required steps and corresponding roles as part of the impedance mismatch in a machine learning lifecycle.

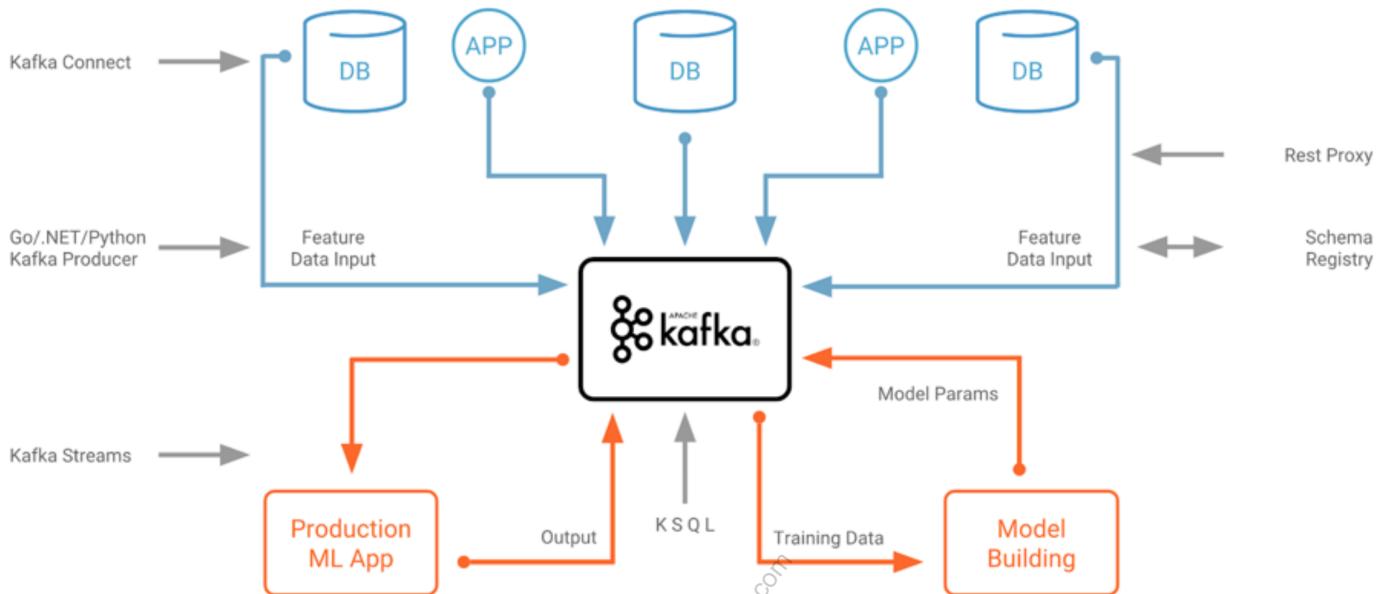
## Hidden Technical Debt



There is an impedance mismatch between model development using Python, its tool stack and a scalable, reliable data platform with low latency, high throughput, zero data loss and 24/7 availability requirements needed for data ingestion, preprocessing, model deployment and monitoring at scale. Python in practice is not the most well-known technology for these requirements. However, it is a great client for a data platform like Apache Kafka.

The problem is that writing the machine learning source code to train an analytic model with Python and the machine learning framework of your choice is just a **very small part** of a real-world machine learning infrastructure. You need to think about the whole model lifecycle. The image on the slide represents this hidden technical debt in machine learning systems - showing how small the **ML code** part is.

## Kafka helps solving Impedance Mismatch

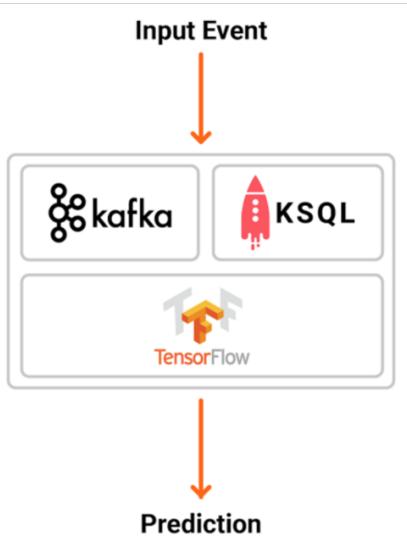


The data engineer builds a scalable integration pipeline using Kafka as infrastructure and Python for integration and preprocessing statements. The data scientist can build their model with Python or any other preferred tool. The production engineer gets the analytic models (either manually or through any automated, continuous integration setup) from the data scientist and embeds them into their Kafka application to deploy it in production. Or, the team works together and builds everything with Java and a framework like **Deeplearning4j** (<https://deeplearning4j.org/>).

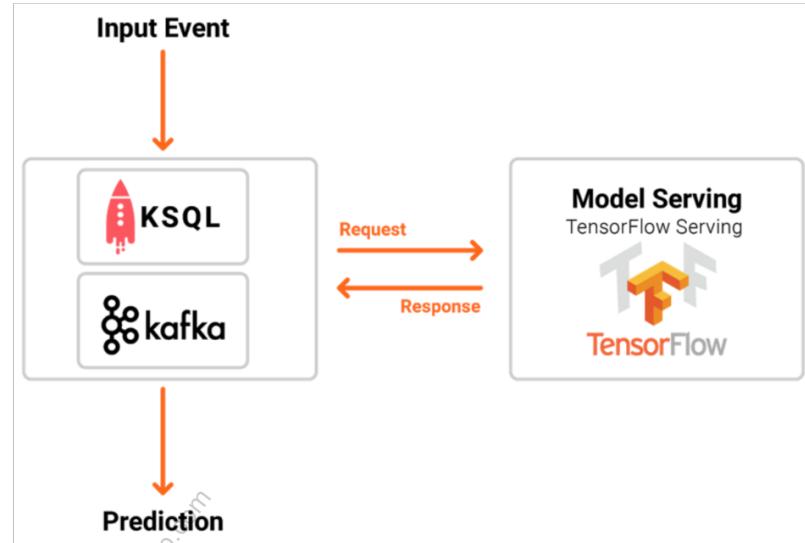
Any option can pair well with Apache Kafka. Pick the pieces you need, whether it's **Kafka core** for data transportation, **Kafka Connect** for data integration or **Kafka Streams/KSQL** for data preprocessing. Many components can be used for both model training and model inference. Write once and use in both scenarios as shown in the diagram on the slide.

**Monitoring** the complete environment in real time and at scale is also a common task for Kafka. A huge benefit is that you only build a highly reliable and scalable pipeline once but use it for both parts of a machine learning infrastructure. And you can use it in any environment: in the cloud, in on-prem datacenters or at the edges, where IoT devices are.

## Embedded



## Dedicated Model Server



On a high level, a machine learning lifecycle contains of two different parts:

- **Model training:** In this step, we feed historical data into an algorithm to learn patterns from the past. The result is an analytic model.
- **Generating predictions:** In this step, we use an analytic model for making predictions on new events based on the learned pattern.

Machine learning is a continuous process, where we repeatedly improve and redeploy the analytic model over time.

Predictions can be performed in different ways within an application or microservice. One way is to embed an analytic model directly into a stream processing application, like an application that uses **Kafka Streams**. You could, for example, use the **TensorFlow for Java API** to load and apply models, as shown on the left side on the slide.

Alternatively, you could deploy the analytic models to a dedicated model server (like TensorFlow Serving), and use RPCs from the streaming application to the service (e.g., with HTTP or gRPC), as shown on the right side of the slide.

## Kafka Streams and ML

```
final KStream<String, String> inputEvents = builder.stream(inputTopic);

KStream<String, String> transformedMessages =
    inputEvents.mapValue(value -> {
        // Transform input values to expected DL4J parameters...
        String[] valuesAsArray = value.split(",");
        int val0 = Integer.parseInt(valuesAsArray[0]);
        int val1 = Integer.parseInt(valuesAsArray[1]);
        INDArray input Nd4j.create(val0, val1);
        // Apply analytic model
        output = model.output(input);
        prediction = output.toString();
        return "Prediction => " + prediction;
    });

// Send prediction result to output topic
transformedMessages.to(outputTopic);
```

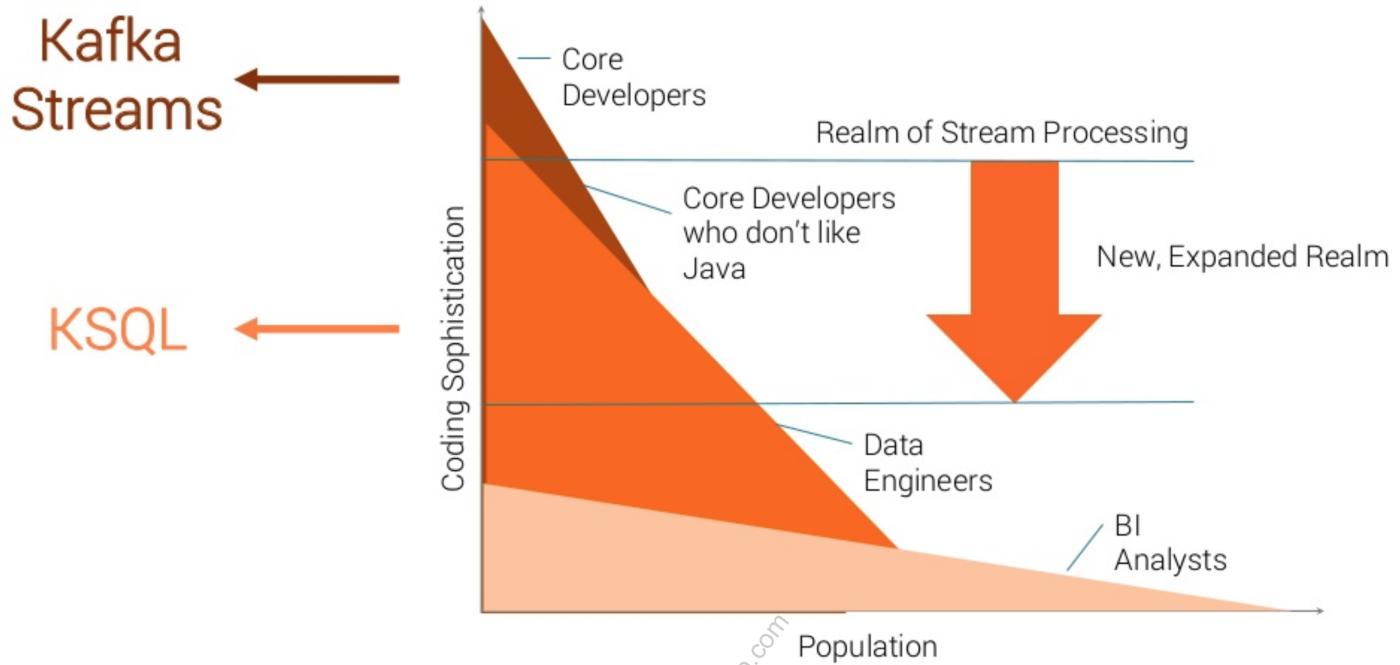
Machine Learning frameworks have different options for combining it with the Java platform (and therefore with Apache Kafka ecosystem), like native Java APIs to load models or RPC interfaces to its framework-specific model servers.

In this code snippet on the slide the trained model is embedded into a Kafka Streams application for real time predictions. We can see the core Kafka Streams logic where we use the **Deeplearning4j API** (DL4J, <https://deeplearning4j.org/>) to do predictions.



Your can start here to learn about **Tensorflow**: <https://www.tensorflow.org/tutorials>

## Why KSQL?



There are very few developers that are able and willing to develop ML models in Java. The population of data engineers and BI Analysts is much bigger than the one of skilled developers. For the latter two KSQL is a much easier way to leverage a Kafka powered event streaming platform for their ML needs.

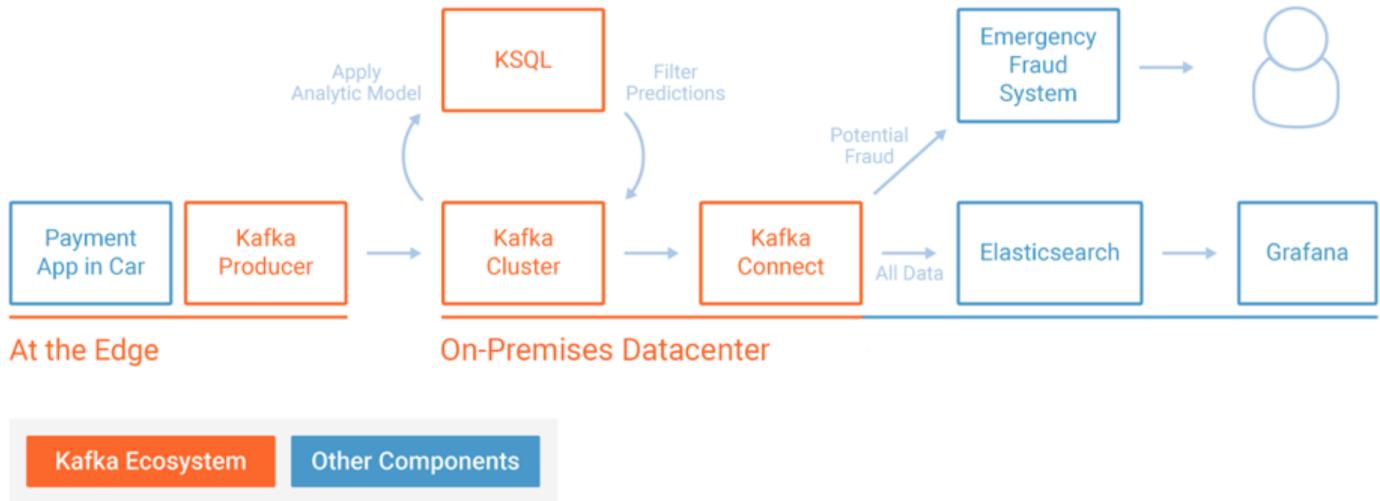
## KSQL & ML for Preprocessing and Model Deployment

```
CREATE STREAM car_sensor AS
  SELECT car_id, event_id, car_model_id, sensor_input
  FROM car_sensor c
  LEFT JOIN car_models m ON c.car_model_id = m.car_model_id
  WHERE m.car_model_type = 'Audi_A8';
```

Processing streaming data with KSQL makes data preparation for machine learning both easy and scalable. You can write SQL statements to do filtering, enrichments, transformations, feature engineering or other tasks. Here on the slide is just one example to filter sensor data from multiple types of cars for a specific car model for further processing or analytics.

ybhandare@greendotcorp.com

# Deep Learning UDF for KSQL



```
SELECT car_id, event_id, ANOMALY(sensor_input) FROM car_sensor;
```

Machine learning models are easily embedded in KSQL by building a **user-defined function** (UDF). We can e.g. create a Deep Learning UDF for KSQL for Streaming Anomaly Detection of MQTT IoT Sensor Data, where we apply a neural network—more precisely an unsupervised autoencoder—for sensor analytics to detect anomalies.

In this example, KSQL continuously processes millions of events from connected cars via **MQTT** integration to the Kafka cluster.



MQTT is a publish / subscribe messaging protocol built for constrained devices and unreliable networks. It is often used in combination with Apache Kafka to integrate IoT devices with the rest of the enterprise. The autoencoder is used for predictive maintenance.

Real-time analytics combined with this car sensor data allows us to send anomalies to a warning or emergency system to act before the engine breaks. Other use cases for intelligent connected cars worth noting include optimized routings and logistics, selling new features for a better digital car experience and loyalty services that integrate with restaurants and other shops on the streets.

While a **KSQL UDF** requires writing a bit of code, this has to be done only once by the developer. Afterwards, the end user can simply use the UDF within their KSQL statements like any built-in function. Here is the KSQL query from this example, using the ANOMALY UDF, which applies the TensorFlow model under the hood.

Both **KSQL** and **Kafka Streams**, depending on our preference and requirements, are a perfect fit for machine learning infrastructures when it comes to preprocessing streaming data and doing model inference. KSQL lowers the entry barrier and allows us to realize streaming applications with simple SQL statements instead of writing source code.

ybhandare@greendotcorp.com

## KSQL UDF using TensorFlow

```
...
import hex.genmodel.GenModel;
import hex.genmodel.easy.EasyPredictModelWrapper;
...

@UdfDescription(name = "anomaly", description = "anomaly detection using deep learning")
public class Anomaly {
    @Udf(description = "apply analytic model to sensor input")
    public String anomaly(String sensorinput) {
        GenModel rawModel;
        try {
            rawModel = (hex.genmodel.GenModel) Class.forName(modelClassName).newInstance();
            EasyPredictModelWrapper model = new EasyPredictModelWrapper(rawModel);
            ...
            AutoEncoderModelPrediction p = model.predictAutoEncoder(row);
            ...
        return ...
    }
}
```

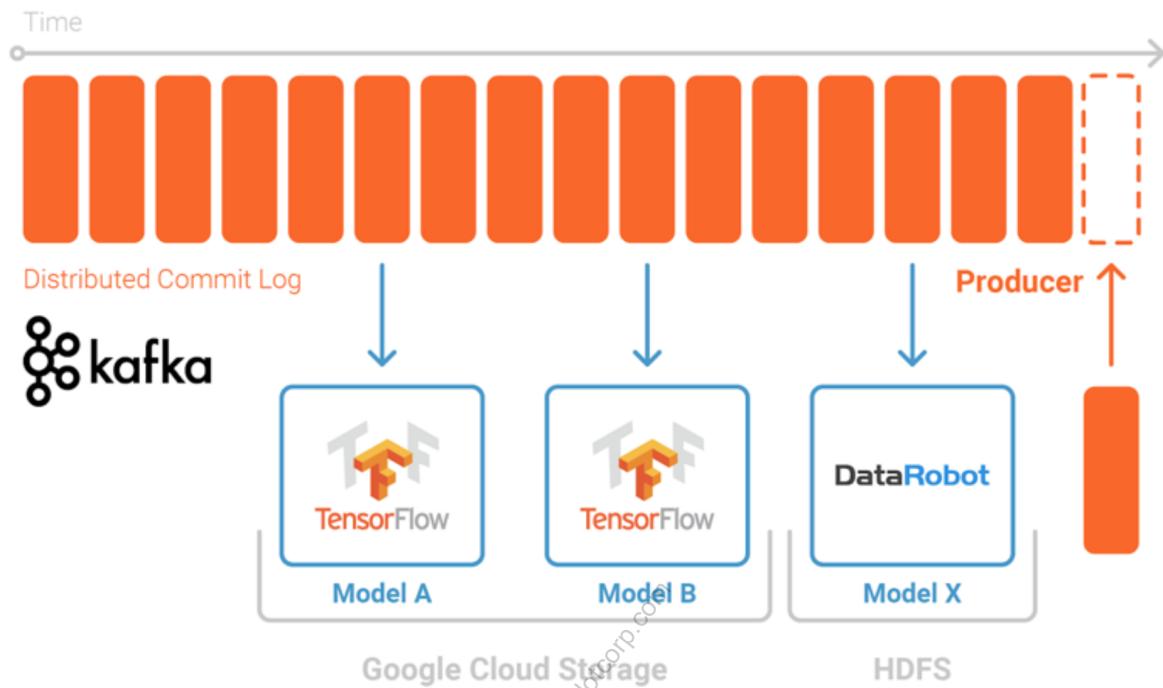
On the slide you can see a shortened code snippet which shows the essential parts of a KSQL UDF that uses TensorFlow to implement the model.

To create such an UDF you need developers with good Java skills. But it is a one time job and the function can be reused many times.

More details see here: <https://github.com/kaiwaehner/ksql-udf-deep-learning-mqtt-iot>

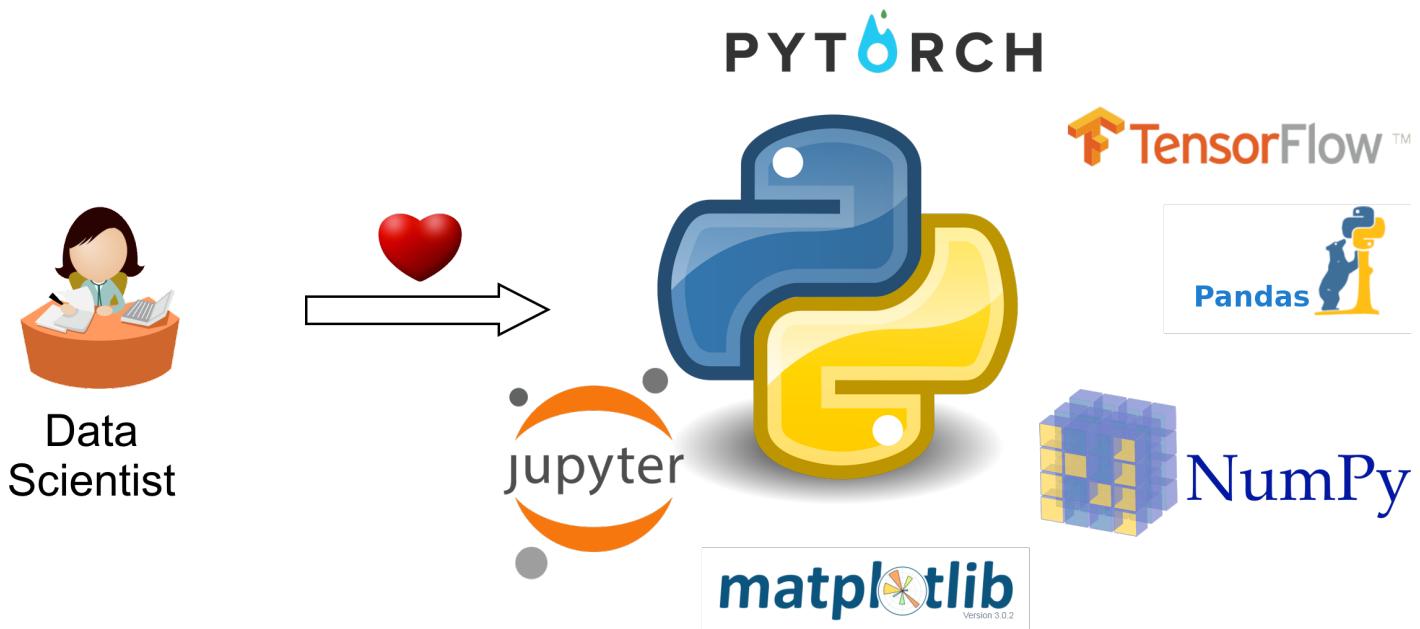
# Replayability

**Replayability**—a log never forgets!



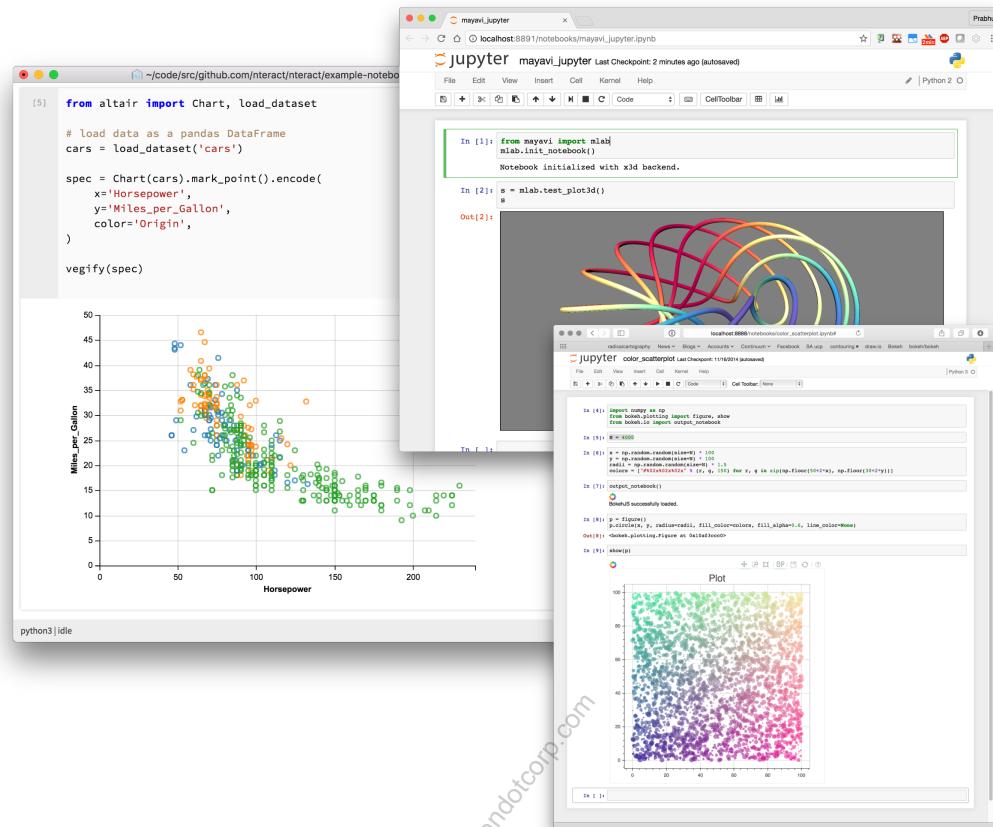
One of the great design concepts of Kafka is that we can re-process data again and again from its distributed commit log.

# Data Scientists love Python



Data scientists love Python, period. Therefore, the majority of machine learning/deep learning frameworks focus on Python APIs. Both the stablest and most cutting edge APIs, as well as the majority of examples and tutorials use Python APIs. In addition to Python support, there is typically support for other programming languages, including JavaScript for web integration and Java for platform integration—though oftentimes with fewer features and less maturity. No matter what other platforms are supported, chances are very high that your data scientists will build and train their analytic models with Python.

# Data Scientists & Python Jupyter

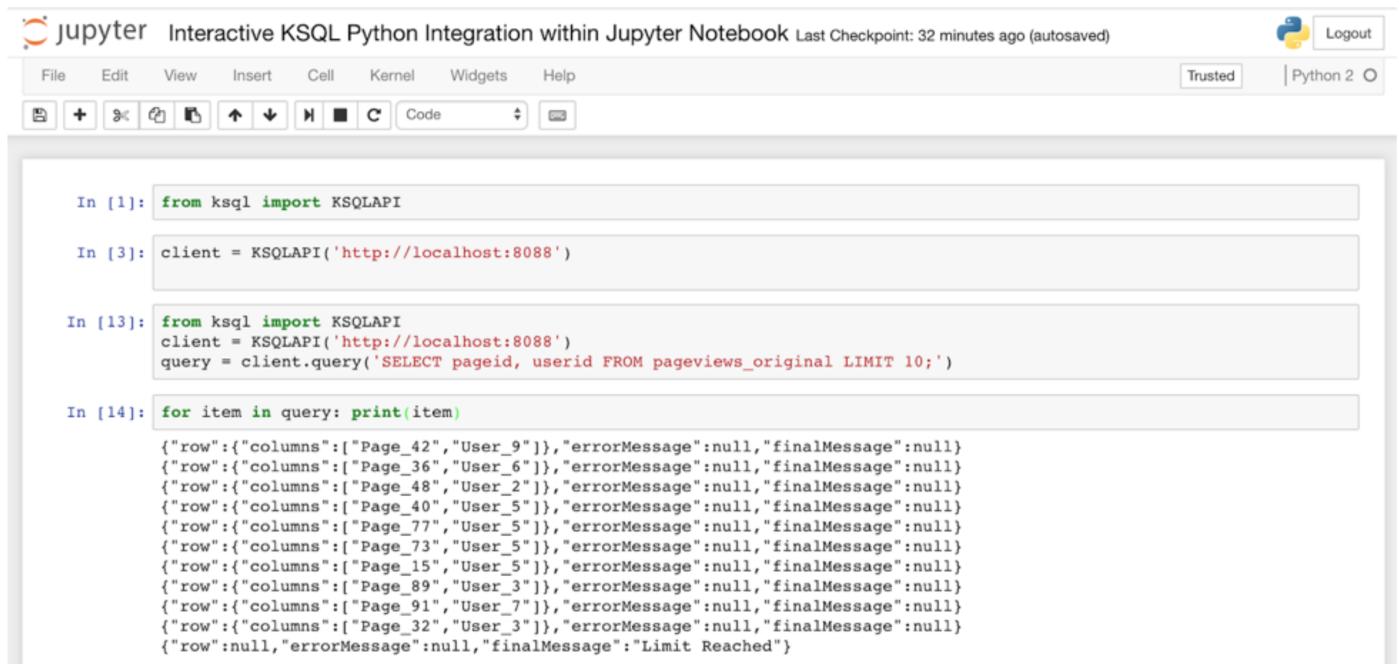


Data scientists use tools like **Jupyter Notebooks** to analyze, transform, enrich, filter and process data. The preprocessed data is then used to train analytic models with machine learning/deep learning frameworks like **TensorFlow**.

However, some data scientists do not even know “bread-and-butter” concepts of software engineers, such as version control systems like GitHub or continuous integration tools like Jenkins.

This raises the question of how to combine the Python experience of data scientists with the benefits of Apache Kafka as a battle-tested, highly scalable data processing and event streaming platform.

# KSQL for Data Scientists & Data Engineers



The screenshot shows a Jupyter Notebook interface with the following details:

- Header:** jupyter Interactive KSQL Python Integration within Jupyter Notebook Last Checkpoint: 32 minutes ago (autosaved)
- Toolbar:** File, Edit, View, Insert, Cell, Kernel, Widgets, Help, Trusted, Python 2
- Code Cells:**
  - In [1]: `from ksql import KSQLAPI`
  - In [3]: `client = KSQLAPI('http://localhost:8088')`
  - In [13]: `from ksql import KSQLAPI`  
`client = KSQLAPI('http://localhost:8088')`  
`query = client.query('SELECT pageid, userid FROM pageviews_original LIMIT 10;')`
  - In [14]: `for item in query: print(item)`  

```
{"row":{"columns":["Page_42","User_9"]}, "errorMessage":null, "finalMessage":null}
{"row":{"columns":["Page_36","User_6"]}, "errorMessage":null, "finalMessage":null}
{"row":{"columns":["Page_48","User_2"]}, "errorMessage":null, "finalMessage":null}
{"row":{"columns":["Page_40","User_5"]}, "errorMessage":null, "finalMessage":null}
{"row":{"columns":["Page_77","User_5"]}, "errorMessage":null, "finalMessage":null}
{"row":{"columns":["Page_73","User_5"]}, "errorMessage":null, "finalMessage":null}
{"row":{"columns":["Page_15","User_5"]}, "errorMessage":null, "finalMessage":null}
{"row":{"columns":["Page_89","User_3"]}, "errorMessage":null, "finalMessage":null}
{"row":{"columns":["Page_91","User_7"]}, "errorMessage":null, "finalMessage":null}
{"row":{"columns":["Page_32","User_3"]}, "errorMessage":null, "finalMessage":null}
{"row":null, "errorMessage":null, "finalMessage": "Limit Reached"}
```

Kafka offers integration options that can be used with Python, like the Confluent's Python Client for Apache Kafka or the Confluent REST Proxy for HTTP integration. But this is not really a convenient way for data scientists who are used to quickly and interactively analyzing and preprocessing data before model training and evaluation. Rapid prototyping is typically used here.

KSQL enables data scientists to take a look at Kafka event streams and implement continuous stream processing from their well-known and loved Python environments like **Jupyter** by writing simple SQL-like statements for interactive analysis and data preprocessing.

The Python example shown on the slide executes an interactive query from a Kafka stream leveraging the open source framework **ksql-python**, which adds a Python layer on top of KSQL's REST interface. You can see a few lines of the Python code using KSQL from a **Jupyter Notebook**.

## From Prototype to Production

---

- Fast prototyping with Python & Jupyter
- Use `ksql-python` to integrate with KSQL
- Deploy KSQL queries to Production
- Leverage scalability, HA and EO of KSQL

KSQL can feel Python native with the `ksql-python` library, but why use KSQL instead of or in addition to your well-known and favorite Python libraries for analyzing and processing data?

The key difference is that these KSQL queries can also be deployed in production afterwards. KSQL offers you all the features from Kafka under the hood like high scalability, reliability and failover handling. The same KSQL statement which you use in your Jupyter Notebook for interactive analysis and preprocessing can scale to millions of messages per second. Fault tolerant. With zero data loss and exactly once semantics. This is very important and valuable for bringing together the Python-loving data scientist with the highly scalable and reliable production infrastructure.

ybhandare@greendotcorp.com



Reflect for a minute on what application involving machine and/or deep learning could you use your Kafka based event streaming platform in your own company.

Would you rather use Kafka Streams with ML libraries for Java to implement the models or would you consider KSQL and maybe Python and Jupyter for the job? Please specify why.

ybhandare@greendotcorp.com

## Hands-On Lab

---

Please refer to the lab **KSQL for Data Scientists & Data Engineers** in the Exercise Book



ybhandare@greendotcorp.com

## Further Reading

---

- Using Apache Kafka to Drive Cutting-Edge Machine Learning: <https://cnfl.io/ml>
- How to Build and Deploy Scalable Machine Learning in Production with Apache Kafka: <https://cnfl.io/kafka-and-ml>
- Machine Learning with Python, Jupyter, KSQL and TensorFlow: <https://cnfl.io/ml-ksql-jupyter>
- Predicting Flight Arrivals with the Apache Kafka Streams API: <https://cnfl.io/flight-arrivals>
- Big Data, Machine Learning, Integration, Messaging, Microservices, Cloud, Internet of Things, Blockchain: <https://cnfl.io/ml-trends>
- 5 Machine Learning Trends for 2018 Combined With Apache Kafka Ecosystem: <https://cnfl.io/ml-trends-2018>
- Apache Kafka and the four challenges of production machine learning systems: <https://cnfl.io/ml-kafka-challenges>

ybhandare@greendotcorp.com