

Confluent Advanced Skills for Optimizing Apache Kafka®

Version 6.0.0-v2.0.1



CONFLUENT

hitesh@datacloud.io

Table of Contents

Introduction	1
Branch 1: Monitoring - Overview	10
01: Business Needs and SLAs	12
a: What are the Pillars of Optimization and How Do You Measure?	14
02: Monitoring Basics	22
a: Monitoring Basics	24
03: Generic Monitoring	33
a: Generic Monitoring	35
Lab: Introduction	61
Lab: Monitoring via JMX	62
Lab: Monitoring librdkafka based Clients	63
Lab: Monitoring with Prometheus	64
04: Monitoring with CCC	65
a: Monitoring with Confluent Control Center	67
Lab: Monitoring with CCC - Data Streams	90
Branch 2: General Troubleshooting & Tuning - Overview	91
05: General Troubleshooting	93
a: Troubleshooting Intro	95
b: Production Down! Troubleshooting Strategies	104
c: Troubleshooting Toolkit	112
Lab: Troubleshooting Toolkit	121
06: Where are my System Log Files?	122
a: Where are my System Log Files?	124
Lab: Where are my Log Files?	145
Branch 3: Troubleshooting & Tuning Central Services - Overview	146
07: Troubleshooting & Tuning ZooKeeper	148
a: Troubleshooting ZooKeeper	150
Lab: Troubleshooting ZooKeeper	161
08: Troubleshooting & Tuning Brokers	162
a: Troubleshooting Brokers	164
b: Replication Concerns	187
Lab: Troubleshooting Brokers	194
Lab: Not Enough ISRs	195
c: Tuning Brokers: General Concepts & Best Practices	196

d: Optimization of a Message's Life Cycle on a Broker	204
e: Miscellaneous Matters	214
f: Confluent Auto Data Balancer	220
g: Message Delivery Guarantees	225
Lab: Tuning Brokers	233
09: Troubleshooting & Tuning Schema Registry	234
a: Troubleshooting the Schema Registry	236
Lab: Troubleshooting the Schema Registry	252
Branch 4: Troubleshooting & Tuning Clients - Overview	253
10: Troubleshooting & Tuning Producers	255
a: Troubleshooting Producers	257
Lab: Troubleshooting Producers	266
b: Tuning Producers	267
Lab: Tuning Producers	293
Lab: Tuning Producers - Selecting the Best Partition Strategy	294
11: Troubleshooting & Tuning Consumers	295
a: Troubleshooting Consumers	297
Lab: Troubleshooting Consumers	314
b: Tuning Consumers	315
Lab: Tuning Consumers	334
12: Troubleshooting & Tuning Streams Apps	335
a: Troubleshooting Kafka Streams & ksqlDB Apps	337
Lab: Troubleshooting Kafka Streams & ksqlDB Apps	350
b: Tuning Kafka Streams & ksqlDB Apps	351
Lab: Tuning Kafka Streams & ksqlDB Apps	367
13: Troubleshooting & Tuning Kafka Connect	368
a: Troubleshooting Kafka Connect	370
b: Tuning Kafka Connect	386
Lab: Tuning Kafka Connect	394
Conclusion	395
Appendix: Additional Problems	403
Problem A: Reviewing Message Sending and Broker Arrival	405
Problem B: Replication Review	406
Problem C: Consumer Offsets and Consumer Lag	407
Problem D: Monitoring Disk Usage	408
Problem E: Assessing Discrepancies in Settings	409
Problem F: Troubleshooting Producers and Consumers	410

Introduction



CONFLUENT
Global Education

hitesh@datacouch.io

Class Logistics and Overview

Copyright & Trademarks

Copyright © Confluent, Inc. 2014-2022. [Privacy Policy](#) | [Terms & Conditions](#).

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

All other trademarks, product names, and company names or logos cited herein
are the property of their respective owners.

hitesh@datacouch.io

Prerequisite

This course requires a working knowledge of the Apache 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 <https://training.confluent.io/learningpath/apache-kafka-fundamentals> for anyone who needs to catch up.

hitesh@datacouch.io

Additional Prerequisite

This course also requires that you have completed the course Apache Kafka® Administration By Confluent.

An understanding of the content from that course is assumed.

Some content in this course will review concepts from this prerequisite, but you will experience the greatest success in this course if you have completed this prerequisite.

hitesh@datacouch.io

Agenda



This course consists of these major parts:

- Monitoring
- General Troubleshooting & Tuning
- Troubleshooting & Tuning Central Services
 - ZooKeeper, Brokers, Schema Registry
- Troubleshooting & Tuning Clients
 - Producers, Consumers, Streams Apps, Kafka Connect

hitesh@datacouch.io

Course Objectives

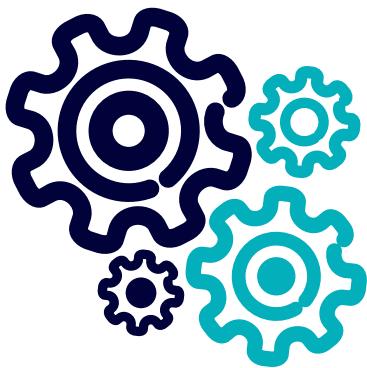
Upon completion of this course, you should be able to:

- Formulate the Apache Kafka® Confluent Platform specific needs of your company
- Monitor all essential aspects of your Confluent Platform
- Tune the Confluent Platform according to your specific needs
- Provide first level production support for your Confluent Platform

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

hitesh@datacouch.io

Class Logistics



- Timing
 - Start and end times
 - Can I come in early/stay late?
 - Breaks
 - Lunch
- Physical Class Concerns
 - Restrooms
 - Wi-Fi and other information
 - Emergency procedures
 - Don't leave belongings unattended



No recording, please!

Expanding on the rule at the bottom: You are not permitted to record via any medium, or stream via any medium any of the content from this class.

How to get the courseware?

1. Register at **training.confluent.io**
2. Verify your email
3. Log in to **training.confluent.io** and enter your **license activation key**
4. Go to the **Classes** dashboard and select your class



Your instructor may choose to have you do this now, combine it with the first lab, or do it before class begins.

Introductions



- About you:
 - What is your name, your company, and your role?
 - Where are you located (city, timezone)?
 - What is your experience with Kafka?
 - Which other Confluent courses have you attended, if any?
 - Optional talking points:
 - What are some other distributed systems you like to work with?
 - What technology most excited you early in your life?
 - Anything else you want to share?
- About your instructor

hitesh@datacouch.io

Branch 1: Monitoring - Overview



CONFLUENT
Global Education

hitesh@datacouch.io

Agenda



This is a branch of our CAO content on monitoring. It is broken down into the following modules:

1. Business Needs & SLAs
2. Monitoring Basics
3. Generic Monitoring
4. Monitoring with CCC

hitesh@datacouch.io

01: Business Needs and SLAs



CONFLUENT
Global Education

hitesh@datacouch.io

Module Overview



This module contains one lesson:

1. What are the Pillars of Optimization and How Do You Measure?

hitesh@datacouch.io

a: What are the Pillars of Optimization and How Do You Measure?

Description

Establishing what is meant by throughput, latency, durability, availability. SLIs, SLOs/SLTs, SLAs.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Identify 2 to 3 criteria that govern your streaming needs
- Quantify (qualitatively) the throughput, latency or durability needs of your streaming platform
- Define SLAs for your streaming platform

hitesh@datacouch.io

Deciding Which Service Criteria to Optimize



Throughput



Latency



Durability



Availability

The first step is to decide which service criteria you want to optimize. We'll consider four criteria which often involve tradeoffs with one another: throughput, latency, durability, availability. To figure out which criteria you want to optimize, recall the use cases your cluster is going to serve.

Think about the applications, the business requirements — the things that absolutely cannot fail for that use case to be satisfied. Think about how Kafka as a streaming platform fits into the pipeline of your business.

Measuring Aspects of Performance

There are a few different measurements people use to describe performance:

- Service Level Indicators
- Service Level Thresholds a.k.a Service Level Objectives
- Service Level Agreements

We'll look at each in turn and build up...

hitesh@datacouch.io

Service Level Indicators (SLIs)

Metric describing one aspect of a service's reliability; make objective & measurable

Example: Overall throughput

hitesh@datacouch.io

Service Level Objectives a.k.a. Thresholds (SLOs / SLTs)

SLI with target value

Example: Throughput > 200 MB/sec

hitesh@datacouch.io

Service Level Agreements (SLAs)

- Contract between service provider and client
- Usu. several SLOs, interaction agreements
- Usu. what happens if not met

Example:



- **Availability:** > 99.95%
- **Throughput:** > 200 MB/sec
- **Latency:** < 50 ms (95th percentile)
- **Durability:** Exactly Once Semantics
- **Retention:** 1 year
- **Cost:**
- **Monitoring:**
- etc.

A good SLA is important because it sets boundaries and expectations. Clearly defined promises reduce the chances of disappointing a customer. An SLA drives internal processes by setting a clear, measurable standard of performance

An SLA helps to:

1. Manage [customer] expectations
2. Establish a clear understanding of how issues will be prioritized when handling service problems

Further Reading

- [Kafka the Definitive Guide](#), v2
-

Definitions of SLI, SLO, and SLA given in this lesson were taken from p. 176 of the reference (which carefully points out and clarifies common confusion between the terms, something this lesson hopes to do as well).

hitesh@datacouch.io

O2: Monitoring Basics



CONFLUENT
Global Education

hitesh@datacouch.io

Module Overview



This module contains one lesson:

1. Monitoring Basics

hitesh@datacouch.io

a: Monitoring Basics

Description

Motivation for and basics of monitoring.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- List business relevant metrics in your Streaming Platform
- List some relevant metrics to monitor a Streaming Platform
- Determine which metrics are critical to monitor

hitesh@datacouch.io

Motivation

You cannot fix problems you cannot see!

Why is monitoring so important?

hitesh@datacouch.io

What Does Your Business Care For?

Are all of your services behaving properly and meeting SLAs?

Questions you need to be able to answer:

- Are applications **receiving all data?**
 - Are my business applications showing the **latest data?**
 - Why are the applications **running slowly?**
 - Do we need to **scale up?**
 - Can any **data get lost?**
 - Will there be **service interruptions?**
 - Are there assurances in case of a **disaster event?**
-

Monitoring help provides assurances that all your services are working properly, meeting SLAs and addressing business needs. In thinking about an operationally sound monitoring solution, think through what the business cares about. On the slide are some common business-level questions.

Notice that these questions are platform-agnostic questions; however, as the Kafka operations engineer you certainly have to be able answer them in a Kafka-specific way.

What's Important to You?

- Message retention
- Message throughput
- Producer performance
- Consumer performance
- Availability
- Durability



- Message retention - Disk size matters here most
- Message throughput - Network capacity is key
- Producer performance - Disk I/O, how fast can the broker(s) persist incoming messages?
- Consumer performance - loads of memory on brokers, to minimize disk IO when consuming
- Availability - The system and all data needs to be available even if some components are down or malfunctioning
- Durability - The data is guaranteed to be persisted even if some parts of the system malfunction

Monitoring the Foundation

- CPU load
 - Network inbound and outbound
 - File handle usage for Kafka
 - Memory utilization
 - Disk
 - Garbage collection
-

- CPU load. *It has been noted that in many cases Kafka is not very CPU intensive. Yet that doesn't mean that it is not equally important to closely monitor CPU usage. On the other hand Kafka Streams and ksqlDB applications can often be CPU bound.*
- Network inbound and outbound - how much data is flowing in or out to the Kafka cluster and/or individual brokers?
- File handle usage for Kafka
- Memory utilization - how much RAM do individual processes such as brokers consume?
- Disk
 - Free space - where you write logs (log4j), and where Kafka stores messages (commit log)
 - Free inodes
 - IO performance - at least **average wait** and **percentage utilization**
- Garbage collection - is GC blocking the system, if yes how often and how long?

The Metrics Swamp

- Hundreds of metrics per broker available
 - Cannot monitor all
 - Will concentrate on most critical ones
-
- Kafka exposes hundreds of metrics. Some of them are per broker, per client, per topic and per partition, and so the number of metrics scales up as the cluster grows. For an average-size Kafka cluster, the number of metrics can very quickly grow to the thousands.
 - Realistically one cannot monitor every single available metric
 - We will filter down the list of metrics to a short set of the most critical ones

hitesh@datacouch.io

Review



Question:

What is your company's highest priority, throughput or minimal latency?

hitesh@datacouch.io

Further Reading

- Monitoring Your Apache Kafka® Deployment End-to-End:
<https://www.confluent.io/monitoring-your-apache-kafka-deployment>
 - The Blog Post on Monitoring an Apache Kafka Deployment to End Most Blog Posts:
<https://www.confluent.io/blog/blog-post-on-monitoring-an-apache-kafka-deployment-to-end-most-blog-posts>
 - Kafka Protocol Guide:
<http://kafka.apache.org/protocol.html>
-

These are a few links to material diving into the topics of this module in more depth.

hitesh@datacouch.io

03: Generic Monitoring



CONFLUENT
Global Education

hitesh@datacouch.io

Module Overview



This module contains one lessons:

1. Generic Monitoring

hitesh@datacouch.io

a: Generic Monitoring

Description

JMX. Other relevant monitoring tools. Early considerations for monitoring various clients.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Instrument any Kafka component to collect relevant JMX metrics
- Define dashboards and alerts from important aggregated metrics
- Describe what monitoring interceptors are and how they work

hitesh@datacouch.io

Monitoring JMX Metrics

- CLI tools: jmxterm
- GUI tools: jconsole
- Use Grafana, Graphite, CloudWatch, Datadog, etc.



Kafka exposes JMX metrics, but they aren't exposed remotely by default. To expose JMX metrics remotely, the broker needs to be started with the JMX_PORT environment variable set.

Metrics can be collected and queried in a variety of different ways:

- Monitoring tools: ideally, JMX metrics are graphed in a monitoring tool such as Grafana, Graphite, CloudWatch, Datadog, others. GUI tools: such as jconsole CLI tools: such as jmxterm
IMPORTANT: jconsole and jmxterm can only connect to a remote host if the correct JVM settings are set.

Monitoring ZooKeeper - The Four Letter Words

- ZooKeeper emits operational data in response to a limited set of commands known as "the four letter words"
 - Usage: `echo "<command>" | nc <host> <port>`
 - `mntr` command response includes the following metrics:

<code>zk_outstanding_requests</code>	Number of requests queued
<code>zk_avg_latency</code>	Amount of time it takes to respond to a client request (in ms)
<code>zk_num_alive_connections</code>	Number of clients connected to ZooKeeper
<code>zk_followers</code>	Number of active followers
<code>zk_pending_syncs</code>	Number of pending syncs from followers
<code>zk_open_file_descriptor_count</code>	Number of file descriptors in use

You can issue a four letter word to ZooKeeper via `telnet` or `nc` (netcat). The most-used of these commands are: `stat`, `srvr`, `cons`, and `mntr`. The full command list can be found at the following link with a short description and availability by version:

https://zookeeper.apache.org/doc/r3.5.7/zookeeperAdmin.html#sc_zkCommands

- `zk_outstanding_requests` determines the **saturation** of ZK
- `zk_avg_latency` is an indicator for **throughput**
- `zk_num_alive_connections` & `zk_followers` are important in regards of **availability**
- `zk_open_file_descriptor_count` is a measure for the **utilization**

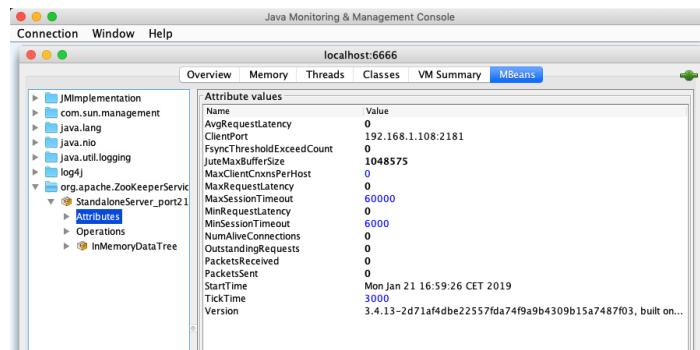
Monitoring ZooKeeper - JMX Metrics

Enable Monitoring (Docker):

```
zookeeper:
  image: confluentinc/cp-zookeeper:6.0.0-1-ubi8
  hostname: zookeeper
  networks:
    - confluent
  ports:
    - 9999:9999
  environment:
    ZOOKEEPER_CLIENT_PORT: 2181
    ZOOKEEPER_TICK_TIME: 2000
    KAFKA_JMX_PORT: 9999
    KAFKA_JMX_HOSTNAME: 127.0.0.1
```

Enable Monitoring (native):

```
export ZOOKEEPER_JMX_PORT=9999
bin/confluent start zookeeper
```



Via JMX ZooKeeper only exposes its `min`, `average`, and `max` request latency, which usually isn't very helpful because more granular statistics are required to understand performance. However, a metric in Kafka can be used as a good proxy for measuring ZooKeeper request latency performance. This Kafka metric has more statistical information than just min, average, and max. The metric is

`kafka.controller:type=ControllerStats, name=LeaderElectionRateAndTimeMs` and the attributes are the percentiles (not the rates).

When running in a container (e.g. using Docker compose), ZK can be configured as shown in the code snippet (bold parts) to enable monitoring of the JMX metrics (on port 9999).



when running natively, the definition of the environment variable for the JMX port has changed for Confluent 5.1.2. For further details see here: <https://docs.confluent.io/5.1.2/release-notes.html#confluent-cli>.

The new definition for the variable name is `${COMPONENT}_JMX_PORT`

Monitoring ZooKeeper - UI

Docker:

```
docker container run -it --rm \
--net <network name> \
-p 8080:8080 \
goodguide/zk-web
```

Native:

```
git clone git://github.com/qiuxiafei/zk-
web.git
cd zk-web
lein deps # run this if you're using
lein 1.x
lein run
```

The screenshot shows a web browser window titled "ZK-Web Make zookeeper simpler". The URL is "localhost:8080/node?path=/". The page has three main sections: "Children", "Node Stat", and "Node Data". The "Children" section lists nodes like schema_registry, cluster, controller, controller_epoch, brokers, zookeeper, admin, ier_change_notification, consumers, log_dir_event_notification, latest_producer_id_block, and config. The "Node Stat" section provides detailed statistics for each node, such as numChildren (12), ephemeralOwner (0), cversion (10), mzxid (0), czxid (0), dataLength (0), ctime (0), version (0), aversion (0), rmtime (0), and paxid (54). The "Node Data" section is currently empty, showing "0 bytes".

zk-web is a useful tool for monitoring ZooKeeper

- It includes a easy to use Web UI
- It is written in **clojure** with **noir** and **bootstrap**



To run **zk-web** natively currently one needs **leinigen** and **git**

Additional information on `leinigen` can be found at the following link.

<https://leinigen.org/>

Monitoring Brokers - System Metrics

Observe:

- CPU usage
- Memory usage
- Available disk space
- Disk IO
- Network IO
- Open file handles

Alert:

- 60% disk usage for disks
- 60% disk IO usage
- 60% network IO usage
- 60% file handle usage

On every system that we run a broker - and we should always run brokers exclusively on a system - we need to monitor the foundation. On the left side of the slide you see the list of elements to monitor for. To automate the system we further recommend to define alerts on certain metrics. Generally it is recommended to trigger an alert when the current value of the metric exceeds 60% of its maximum value. You may also want to reason whether to trigger the alert upon the first occurrence of this limit (disks) or if the value exceeds 60% over a given time period (network IO, CPU).

60% is recommended by support and is chosen to afford time and resources to add more nodes and move partitions accordingly



the Kafka broker will only use JVM heap space for meta data and replication buffers. The broker relies on the OS page cache for Kafka commit logs.

Kinds of JMX Metrics

There are two classes of JMX metrics:

- **gauge** - a measure of something *right now*
 - e.g., number of offline partitions
- **meter** - a measure of something over a time sample
 - e.g., throughput

hitesh@datacouch.io

More on Meter Metrics

"Over a time sample" really means over a set of time samples.

So how can we control that?

All meter metrics can be configured by these properties:

Name	Meaning	Default
<code>metrics.sample.window.ms</code>	Size of each sample window	30 sec.
<code>metrics.num.samples</code>	Number of samples maintained and included in reports	2

Monitoring Brokers - JMX Metrics (1)

Broker Load:

```
kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec  
kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec  
kafka.network:type=RequestMetrics,name=RequestsPerSec,request=<type>  
kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec
```

hitesh@datacouch.io

Monitoring Brokers - JMX Metrics (2)

Enable Monitoring (native):

```
# Add to  
/etc/kafka/kafka.properties:  
jmx.port=9999  
jmx.hostname=127.0.0.1  
  
# Start Kafka  
bin/kafka-server-start.sh
```

Enable Monitoring (Docker):

```
kafka:  
  image: "confluentinc/cp-enterprise-  
kafka:6.0.0-1-ubi8"  
  networks:  
    - confluent  
  ports:  
    - 9999:9999  
  environment:  
    KAFKA_BROKER_ID: 101  
    KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181  
    ...  
    KAFKA_JMX_PORT: 9999  
    KAFKA_JMX_HOSTNAME: 127.0.0.1  
    ...
```

Monitoring Kafka with Burrow

- Developed by LinkedIn
- Monitor Consumer Lag in Kafka
- HTTP Endpoints to get info about cluster

```
$ curl -s localhost:8000/v3/kafka/local
{
  "error": false,
  "message": "cluster module detail returned",
  "module": {
    "class-name": "kafka",
    "servers": [
      "kafka:9092"
    ],
    "client-profile": {
      "name": "",
      "client-id": "burrow-lagchecker",
      "kafka-version": "0.8",
      "tls": null,
      "sasl": null
    },
    "topic-refresh": 60,
    "offset-refresh": 30
  },
  "request": {
    "url": "/v3/kafka/local",
    "host": "burrow"
  }
}
```

Another way of monitoring a Kafka cluster is the use of Burrow.

Burrow is a monitoring tool for keeping track of consumer lag in Apache Kafka. It is designed to monitor every consumer group that is committing offsets to either Kafka or ZooKeeper, and to monitor every topic and partition consumed by those groups. This provides a comprehensive view of consumer status.

Burrow also provides several HTTP request endpoints for getting information about the Kafka cluster and consumers, separate from the lag status. This can be very useful for creating applications that assist with managing your Kafka clusters when it is not convenient (or possible) to run a Java Kafka client.

See: <https://github.com/linkedin/Burrow/wiki>

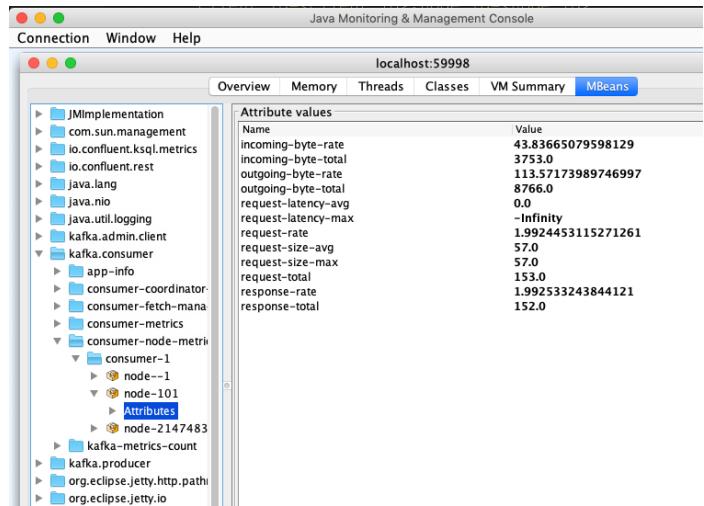
Monitoring Kafka Clients

Common per-broker Metrics for:

Consumer, Producer, Connect, Kafka Streams, ksqlDB

```
kafka.producer:type=producer-node  
-metrics,  
  client-id=<client-id>,node-id=<node-  
id>  
  
kafka.consumer:type=consumer-node  
-metrics,  
  client-id=<client-id>,node-id=<node-  
id>  
  
kafka.connect:type=connect-node-metrics,  
  client-id=<client-id>,node-id=<node-  
id>
```

JMX Metrics for ksqlDB Server:



All Kafka clients (consumer, producer, connect, streams, ksqlDB) have common metrics. They are subsumed under the MBean **[consumer|producer|connect]-node-metrics**.

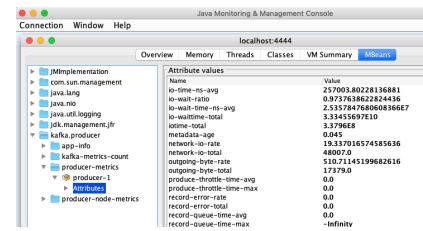
Note, a Kafka Streams instance contains all the producer and consumer metrics as well as additional metrics specific to streams.

Details: https://kafka.apache.org/documentation/#common_node_monitoring

Monitoring Producers

- Common Kafka Client metrics
- Producer metrics

```
kafka.producer:type=producer-metrics,client-id=<client-id>
```



- Producer Sender metrics
- Key Producer metrics

Response rate	Average number of responses received per second
Request rate	Average number of requests sent per second
Request latency avg	Average request latency (in ms)
Outgoing byte rate	Average number of outgoing/incoming bytes per second
IO wait time ns avg	Average length of time the I/O thread spent waiting for a socket (in ns)

Details: https://kafka.apache.org/documentation/#producer_monitoring

Monitoring Consumers

- Common Kafka Client metrics
- Consumer Group metrics
- Consumer Fetch metrics

```
kafka.consumer:type=consumer-fetch-manager-metrics,  
client-id=<client-id>
```

localhost:4445	
	Mbeans
JImplementation	
com.sun.management	
java.lang	
java.net	
java.logging	
kafka.management.jfr	
kafka.consumer	
app-info	
consumer-coordinator-metrics	
consumer-fetch-manager-metrics	
consumer1	
Metrics	
consumer-metrics	
consumer-node-metrics	
kafka-metrics-count	
Attribute values	
Name	Value
byte-consumed-rate	0.0
bytes-consumed-total	0.0
fetch-latency-avg	0.0
fetch-latency-max	-Infinity
fetch-total	0.0
fetch-size-avg	0.0
fetch-size-max	-Infinity
records-consumed-avg	0.0
records-consumed-total	0.0
records-lag-max	-Infinity
records-lead-min	1.7976931348623157E308
records-per-request-avg	0.0

- Key Consumer metrics

ConsumerLag	Number of messages consumer is behind producer
MaxLag	Maximum observed value of ConsumerLag
BytesPerSec	Bytes consumed per second
MessagesPerSec	Messages consumed per second
MinFetchRate	Minimum rate a consumer fetches requests to the broker

Details: https://kafka.apache.org/documentation/#consumer_monitoring

Monitoring Consumers - Consumer Lag

Monitor Consumer Lag for real-time apps:

- JMX: `kafka.consumer:type=consumer-fetch-manager-metrics,client-id=<client-id>`
attribute: `records-lag-max`
- `kafka-consumer-groups` tool

```
$ kafka-consumer-groups \
  --bootstrap-server kafka:9092 \
  --describe \
  --group my-group

TOPIC      PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG    CONSUMER-ID        ...
my-topic    0          2              4              2      consumer-1-029...
my-topic    1          2              3              1      consumer-1-029...
my-topic    2          2              3              1      consumer-2-42c...
```

Consumer Lag

For **real-time** consumer applications, where the consumer is meant to be processing the newest messages with as little latency as possible, consumer lag should be monitored closely. Most real-time applications will want little-to-no consumer lag, because lag introduces end-to-end latency.

Consumer lag can be monitored using the `kafka-consumer-groups` command-line tool, or using the consumer's JMX metric `kafka.consumer:type=consumer-fetch-manager-metrics,client-id={client-id}` attribute: `records-lag-max`.

The `kafka-consumer-groups` command-line tool calculates lag by comparing the last committed offset to the most recent offset, which means lag is a function of when the last commit was made and hence may not be up-to-date. On the other hand, the consumer's `records-lag-max` JMX metric calculates lag by comparing the offset most recently seen by the consumer to the most recent offset in the log, which is a more real-time measurement.

Monitoring Kafka Connect

Monitor:

- Common Kafka Client metrics
- Connect specific metrics:
see: https://kafka.apache.org/documentation/#connect_monitoring
 - Connector metrics
 - Common task metric
 - Source task metrics
 - Sink task metrics
 - Worker metrics
 - Worker rebalance metrics
- Hosts where workers run
- The source/sink system
- Workers through REST interface

Distinguish standalone vs. distributed workers

Proper monitoring of Connect clients helps facilitate sizing and scalability efforts along with exposing useful information for troubleshooting issues.

Because workers will launch consumers and/or producers, all consumer and producer metrics should be monitored for each worker.

Monitoring Workers with JMX

Workers have embedded producer and/or consumer instances that should be monitored like every other producer or consumer client. Useful JMX metrics to monitor are:

- Common metrics for all clients
- Producer metrics
- Consumer metrics

Because workers will launch consumers and/or producers, all metrics above should be monitored for each worker. ---

Monitoring Kafka Connect

Additionally we have (each metric is explained in detail in the given link):

- Connector Metrics
MBean name: `kafka.connect:type=connector-metrics,connector=([-.\w]+)`
- Common Task Metrics
MBean name: `kafka.connect:type=task-metrics,connector=([-.\w]+),task=([-.\w]+)`
- Source Task Metrics
MBean name: `kafka.connect:type=source-task-metrics,connector=([-.\w]+),task=([\d]+)`
- Sink Task Metrics
MBean name: `kafka.connect:type=sink-task-metrics,connector=([-.\w]+),task=([\d]+)`
- Worker Metrics
MBean name: `kafka.connect:type=connect-worker-metrics`
- Worker Rebalance Metrics
MBean name: `kafka.connect:type=connect-worker-rebalance-metrics`

Standalone versus Distributed Workers

Standalone workers launch either an embedded producer (source) or consumer (sink) depending on the connector configuration specified to launch in the standalone worker. Because workers do not coordinate in standalone mode, each standalone worker's monitoring setup should be tailored to the needs of that worker.

In distributed mode, workers coordinate and thus always provide metrics for the embedded producer and consumer instances. In addition to these metrics, distributed workers expose metrics for the worker group under the `kafka.connect` metrics subsection. These `kafka.connect` metrics should be monitored similar to how users monitor consumer group metrics, by watching for anomalies in group join rates, sync rates, connection counts, and bytes in/out rates. The `kafka.connect` metrics also include information on how many tasks and connectors have been assigned to the worker which should be monitored as well.

Monitoring Workers with the REST interface

The REST interface on any worker in a distributed worker cluster can be used to poll for connector and task status. Any FAILED task should be investigated further. ---

Monitoring Kafka Connect

Monitoring Hosts where Workers Run

Host and JVM level metrics are useful to understand the environment in which Connect worker(s) run. Specifically, the following metrics should be used to monitor divergence from steady-state behavior:

- CPU utilization
- Garbage collection pause duration
- Heap usage
- Physical memory usage

Monitoring the Source/Sink System

Because Connect is designed specifically to interact with systems outside of Kafka, these systems at the source or sink side should be monitored in order to identify any bottlenecks that may lie outside of Connect workers. A few examples of monitoring systems are below:

- Source database read/write operations rate
- HDFS NameNode garbage collection pause duration
- Elasticsearch host CPU and memory utilization

Monitoring Kafka Connect

The screenshot shows the Java Management Interface (JConsole) running on localhost:59997. The 'MBeans' tab is selected. On the left, a tree view lists various MBean domains, with 'kafka.connect' expanded to show its sub-MBeans: 'app-info', 'connect-coordinator-metrics', 'connect-metrics', 'connect-node-metrics', 'connect-worker-metrics' (which has 'Attributes' selected), 'connect-worker-rebalance-metrics', 'kafka-metrics-count', 'kafka.consumer', and 'kafka.producer'. On the right, a table titled 'Attribute values' displays the current values for attributes under 'connect-worker-metrics'. The table has two columns: 'Name' and 'Value'. All values listed are 0.0.

Name	Value
connector-count	0.0
connector-startup-attempts-total	0.0
connector-startup-failure-percentage	0.0
connector-startup-failure-total	0.0
connector-startup-success-percentage	0.0
connector-startup-success-total	0.0
task-count	0.0
task-startup-attempts-total	0.0
task-startup-failure-percentage	0.0
task-startup-failure-total	0.0
task-startup-success-percentage	0.0
task-startup-success-total	0.0

Monitoring Kafka Streams & ksqlDB Apps

Monitor:

- Thread metrics
- Task metrics
- Processor Node metrics
- State Store metrics
- Record Cache metrics



ksqlDB has some additional "debug"-level metrics not enabled by default!

Details: https://kafka.apache.org/documentation/#kafka_streams_monitoring

- thread metrics: `kafka.streams:type=stream-metrics,client-id=<client-id>`
- task metrics: `kafka.streams:type=stream-task-metrics,client-id=<client-id>,task-id=<task-id>`
- processor node: `kafka.streams:type=stream-processor-node-metrics,client-id=<client-id>,task-id=<task-id>,processor-node-id=<processor-node-id>`
- state store: `kafka.streams:type=stream-[store-scope]-metrics,client-id=<client-id>,task-id=<task-id>,[store-scope]-id`
- record cache: `kafka.streams:type=stream-record-cache-metrics,client-id=<client-id>,>,task-id=<task-id>,record-cache-id=<record-cache-id>`



Under the hood ksqlDB apps are Kafka Streams apps and thus the same metrics are relevant. But, ksqlDB has some additional metrics beyond what are provided by Kafka Streams:

<https://docs.confluent.io/current/ksql/docs/operations.html#monitoring-and-metrics>

These Streams metrics are "debug" level, and not enabled by default

Monitoring Confluent Schema Registry

The screenshot shows the Java Monitoring & Management Console interface. The title bar reads "Java Monitoring & Management Console" and the address bar says "localhost:59999". The top menu includes "Connection", "Window", and "Help". Below the menu is a toolbar with three colored circles (red, yellow, green). The main area has tabs: "Overview", "Memory", "Threads", "Classes", "VM Summary", and "MBeans". The "MBeans" tab is selected and highlighted in blue. On the left, there's a tree view of JMX MBeans:

- ▶ JMImplementation
- ▶ com.sun.management
- ▶ io.confluent.rest
- ▶ java.lang
- ▶ java.nio
- ▶ java.util.logging
- ▶ kafka
- ▶ kafka.consumer
- ▶ kafka.producer
- ▶ kafka.schema.registry
 - ▼ jersey-metrics
 - ▶ Attributes
 - ▼ jetty-metrics
 - ▶ **Attributes**
 - ▼ master-slave-role
 - ▶ Attributes
- ▶ org.eclipse.jetty.http.pathmap
- ▶ org.eclipse.jetty.io
- ▶ org.eclipse.jetty.jmx
- ▶ org.eclipse.jetty.server
- ▶ org.eclipse.jetty.server.handler
- ▶ org.eclipse.jetty.server.handler.gzip

The "jetty-metrics" node under "kafka.schema.registry" has its "Attributes" section expanded, with one item highlighted in blue: "Attributes". On the right, there's a table titled "Attribute values" with columns "Name" and "Value". The table contains the following data:

Name	Value
connections-accepted-rate	0.0
connections-active	0.0
connections-closed-rate	0.0
connections-opened-rate	0.0

Details: <https://docs.confluent.io/current/schema-registry/docs/monitoring.html>

The schema registry under the hood also has a pair of producer and consumer. It also has some specific metrics under the MBean **kafka.schema.registry**.

Furthermore the schema registry uses the Jetty web server to expose its REST API, thus one finds many Jetty related JMX metrics as shown on the slide.

Monitoring Confluent REST Proxy

The screenshot shows the JConsole interface for monitoring a Confluent REST Proxy. The title bar indicates the connection is to `localhost:55555`. The top navigation bar includes tabs for `Overview`, `Memory`, `Threads`, `Classes`, `VM Summary`, and `MBeans`. The `MBeans` tab is currently selected.

The left pane displays a hierarchical tree of MBeans:

- `JMImplementation`
- `com.sun.management`
- `io.confluent.rest`
- `java.lang`
- `java.nio`
- `java.util.logging`
- `kafka`
- `kafka.rest`
 - `jersey-metrics`
 - `Attributes`
 - `jetty-metrics`
 - `Attributes`
- `org.eclipse.jetty.http.pathmap`
- `org.eclipse.jetty.io`
- `org.eclipse.jetty.jmx`
- `org.eclipse.jetty.server`
- `org.eclipse.jetty.server.handler`
- `org.eclipse.jetty.server.handler.gzip`
- `org.eclipse.jetty.server.session`
- `org.eclipse.jetty.servlet`
- `org.eclipse.jetty.util.thread`
- `org.eclipse.jetty.util.thread.strategy`
- `org.eclipse.websocket.jsr356.server`
- `org.eclipse.jetty.websocket.server`
- `sun.nio.ch`

Attribute values

Name	Value
<code>brokers.list.request-byte-rate</code>	<code>NaN</code>
<code>brokers.list.request-error-rate</code>	<code>0.0</code>
<code>brokers.list.request-latency</code>	<code>NaN</code>
<code>brokers.list.request-latency-99</code>	<code>-Infinity</code>
<code>brokers.list.request-rate</code>	<code>0.0</code>
<code>brokers.list.request-size-avg</code>	<code>NaN</code>
<code>brokers.list.request-size-max</code>	<code>-Infinity</code>
<code>brokers.list.response-byte-rate</code>	<code>NaN</code>
<code>brokers.list.response-rate</code>	<code>0.0</code>
<code>brokers.list.response-size-avg</code>	<code>NaN</code>
<code>brokers.list.response-size-max</code>	<code>-Infinity</code>
<code>consumer.assign+v2.request-rate</code>	<code>NaN</code>
<code>consumer.assign+v2.request-size-avg</code>	<code>0.0</code>
<code>consumer.assign+v2.request-size-max</code>	<code>NaN</code>
<code>consumer.assign+v2.response-size-avg</code>	<code>-Infinity</code>
<code>consumer.assign+v2.response-size-max</code>	<code>0.0</code>
<code>consumer.assignment+v2.ratio</code>	<code>NaN</code>
<code>consumer.assignment+v2.rate</code>	<code>0.0</code>
<code>consumer.assignment+v2.size</code>	<code>NaN</code>
<code>consumer.assignment+v2.time</code>	<code>-Infinity</code>

`Refresh`

Details: <https://docs.confluent.io/current/kafka-rest/docs/monitoring.html>

The Confluent REST Proxy under the hood also creates producers and consumers. Thus one will find consumer and producer specific metrics there...

On the slide we have a snapshot of a REST Proxy for which no producer or consumer has yet been created. Thus there is no `kafka.producer | consumer` MBean available. ---

Monitoring Confluent REST Proxy



Currently the start script of REST Proxy is buggy and does not correctly configure JMX when running inside a container. The important `JMX_HOSTNAME` is ignored. To configure REST Proxy for JMX monitoring use this environment variable in your e.g. Docker compose file:

```
KAFKAREST_JMX_OPTS: "-Dcom.sun.management.jmxremote  
-Dcom.sun.management.jmxremote.authenticate=false  
-Dcom.sun.management.jmxremote.ssl=false  
-Dcom.sun.management.jmxremote.rmi.port=55555  
-Dcom.sun.management.jmxremote.port=55555  
-Djava.rmi.server.hostname=127.0.0.1"
```

Review



Question:

Justify why it **doesn't make sense** to monitor all possible metrics of any component of the Confluent Platform.

hitesh@datacouch.io

Further Reading

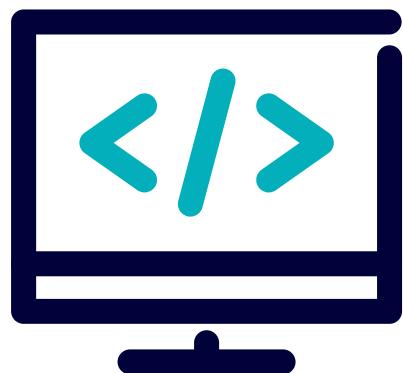
- Monitoring Kafka:
<https://docs.confluent.io/current/kafka/monitoring.html>
- JMX Reporters:
<https://cwiki.apache.org/confluence/display/KAFKA/JMX+Reporters>
- Monitoring Apache Kafka with Grafana / InfluxDB via JMX
<https://softwaremill.com/monitoring-apache-kafka-with-influxdb-grafana/>
- Monitoring Kafka Streams Metrics via JMX:
<https://www.madewithtea.com/posts/monitoring-metrics-kafka-streams>
- Monitoring Kafka in Production:
<https://logz.io/blog/monitoring-kafka-in-production/>
- ZK-Web: <https://github.com/qiuxiafei/zk-web>

hitesh@datacouch.io

Lab: Introduction

Please work on **Lab 3a: Introduction**

Refer to the Exercise Guide

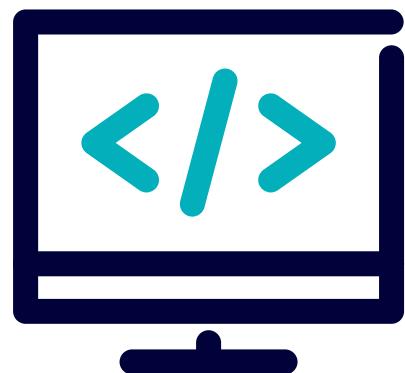


hitesh@datacouch.io

Lab: Monitoring via JMX

Please work on **Lab 3b: Monitoring via JMX**

Refer to the Exercise Guide

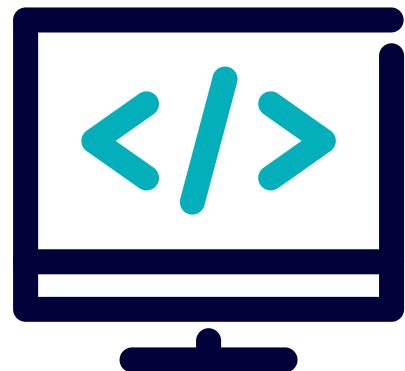


hitesh@datacouch.io

Lab: Monitoring librdkafka based Clients

Please work on **Lab 3c: Monitoring librdkafka based Clients**

Refer to the Exercise Guide

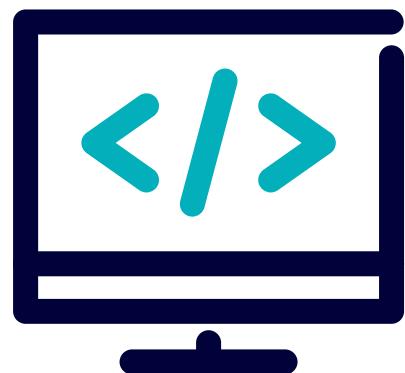


hitesh@datacouch.io

Lab: Monitoring with Prometheus

Please work on **Lab 3d: Monitoring with Prometheus**

Refer to the Exercise Guide



hitesh@datacouch.io

04: Monitoring with CCC



CONFLUENT
Global Education

hitesh@datacouch.io

Module Overview



This module contains one lesson:

1. Monitoring with CCC

hitesh@datacouch.io

a: Monitoring with Confluent Control Center

Description

Tour of CCC and its monitoring capabilities.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Discover consumer lag of any consumer group
- Monitor the cluster and broker health
- Visualize end-to-end latency in your streaming platform
- Measure the throughput of your streaming platform
- Inspect any topic
- Visualize the topic schema
- Monitor the cluster and broker health

hitesh@datacouch.io

Control Center

Expert Kafka Monitoring for the Enterprise

System Health



Are all brokers and topics **available**?

How much **data** is being processed?

What can be tuned to improve
performance?

End-to-End SLA Monitoring



Does Kafka process all events **<15 seconds**?

Is the 8am report **missing data**?

Are there **duplicate** events?

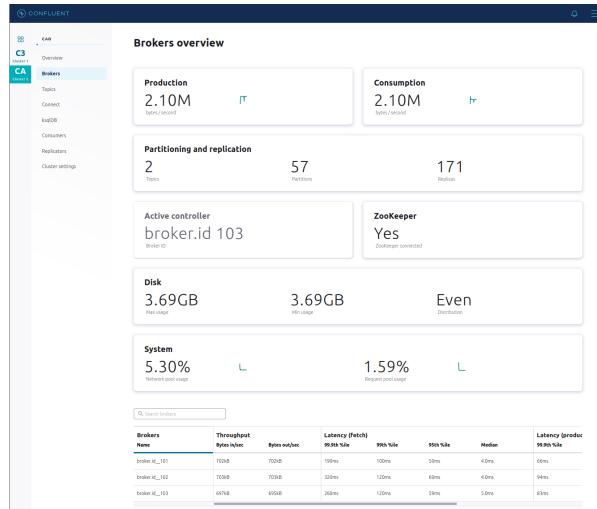
Control Center



{dev}



{ops}



Control Center is very useful for both roles, developers and operations engineers. It offers a unique insight into the Kafka powered, real-time event streaming platform with its many components.

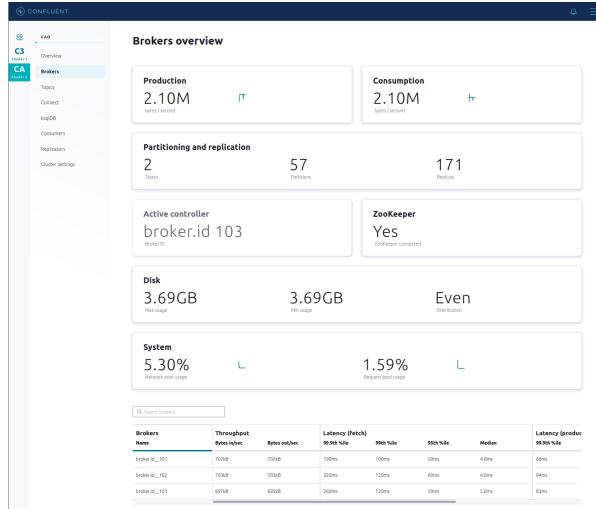
Control Center



Look inside Kafka

- Inspect messages in topics
- View changes to Schema Registry

{dev}



{ops}

Developers can use Control Center to inspect topics they're writing messages to or reading messages from. This includes reading and editing the topic properties, inspecting the messages flowing into the topic, and more. They can also inspect the schema of messages and view the schema evolution due to the tight integration of Control Center with the Confluent Schema Registry.

Control Center



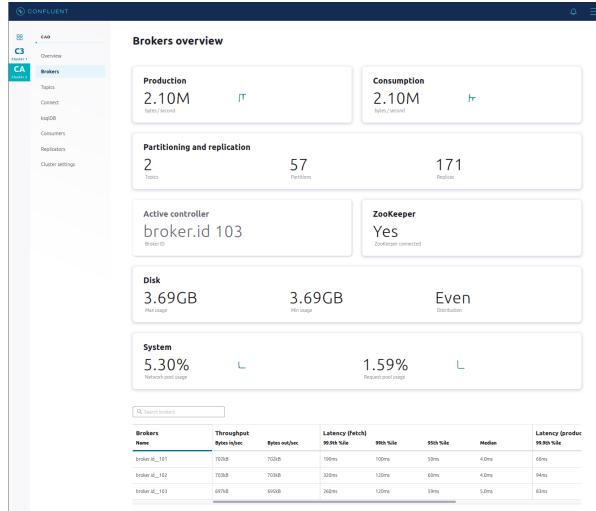
{dev}

Look inside Kafka

- Inspect messages in topics
- View changes to Schema Registry

Build pipelines and process streams

- Configure Kafka Connect and connectors
- Write ksqlDB queries



{ops}

Furthermore developers can use Control Center to also configure source and sink connectors, and defining and debugging ksqlDB queries in a friendly environment offering intellisense and auto-completion.

Control Center



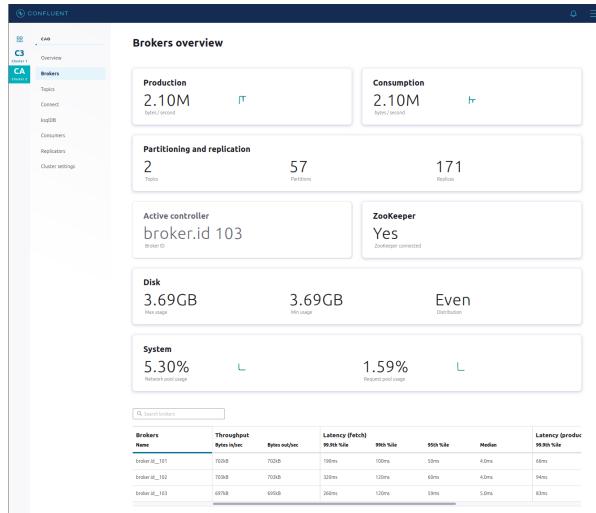
Look inside Kafka

- Inspect messages in topics
- View changes to Schema Registry

{dev}

Build pipelines and process streams

- Configure Kafka Connect and connectors
- Write ksqlDB queries



Meet event stream SLAs

- Track KPIs for event streams
- View consumer lag
- Set and receive alerts



Operations engineers on the other hand can use Control Center to help ensure that the platform meets the event stream SLAs by:

- tracking event stream KPIs (key performance indices)
- monitoring the lag of consumers and consumer groups
- defining and receiving alerts on certain events, such as exceeding a threshold value

Control Center



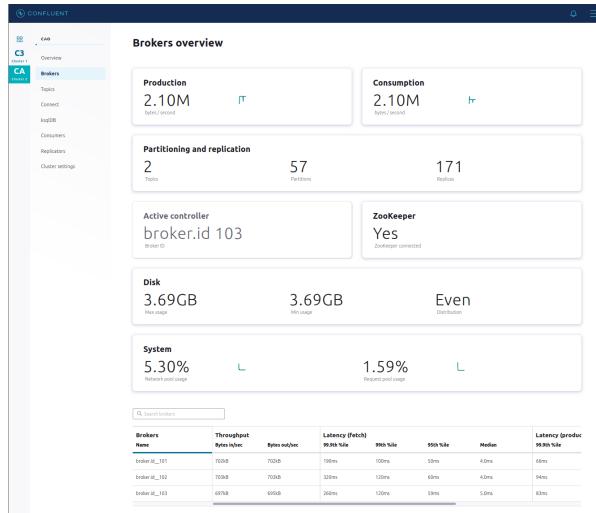
Look inside Kafka

- Inspect messages in topics
- View changes to Schema Registry

{dev}

Build pipelines and process streams

- Configure Kafka Connect and connectors
- Write ksqlDB queries



Meet event stream SLAs

- Track KPIs for event streams
- View consumer lag
- Set and receive alerts



View Kafka clusters at a glance

- Configure Kafka Connect and connectors
- Write ksqlDB queries

Finally operations engineers can use Control Center to view a whole complex Kafka cluster at a glance, including the health of individual brokers and checking on the configuration of each broker.

The Control Center Advantage

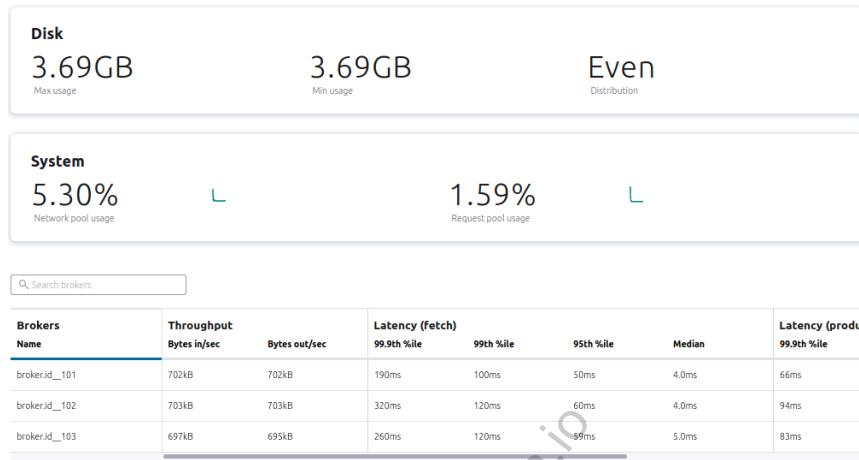
1. Monitor what's important
 2. Inherits performance and scalability improvements in Kafka
 3. 2-for-1 efficiency in operations
 4. Unified security configuration experience
 5. Confluent Support for production cluster & monitoring solution
-

Monitoring your Kafka production cluster with Confluent Control Center provides several key advantages:

1. Control Center monitors what is important as illustrated on the following slide.
2. Control Center processing is implemented as a Kafka Streams application, so all underlying Kafka improvements and Kafka streams improvements are inherited by Control Center. For example, in the Confluent Platform 4.0, Control Center leveraged memory management features of Kafka Streams to cache records optimally, thereby saving intermediate writes and optimizing network bandwidth. This resulted in huge improvements in performance and scalability.
3. Using Control Center for monitoring removes extra layers of complexity by making it easy to deploy with Confluent Platform. The deployment is easier and the operations is easier: since you already have the Kafka expertise for your production data cluster, your expertise transfers to the monitoring solution. Since you know how to scale out your production cluster, you know how to scale out your monitoring cluster. Since you already have maintenance runbooks for your production cluster, you know your maintenance runbooks for your monitoring cluster.
4. Since you know how to configure security on your production Kafka cluster, you can configure security for Control Center in a unified way.
5. With a Confluent Platform subscription, should any unexpected issues arise, you can have a single phone number to call for enterprise-level support. Confluent provides world-class Kafka support, regardless if the issue is in the production cluster or monitoring cluster.

Monitoring what's important

- Brokers added?
- Topics added?
- Network and request pool usage
- Disk utilization across cluster
- Underreplicated & offline partitions



- Control Center's monitoring capabilities were designed from the beginning with Kafka in mind, so it purposefully conveys **what is important**, helps diagnose problems, and identifies performance bottlenecks.
- Control Center also defines **sensible thresholds** so users don't have to guess: the System Health view in particular highlights if an indicator is above a pre-defined threshold. A red indicator appears for example, if:
 - network pool usage or request pool usage exceeds 70%
 - Kafka disk utilization is not evenly distributed across the cluster
 - the number of under replicated or offline partitions is greater than zero

And Control Center is just as elastic as the Kafka cluster is elastic:

- Was a new broker added to the cluster? Control Center automatically reports the new broker.
- Was a new topic added to the cluster? Control Center automatically reports data on the new topic so we can validate it has proper replication configuration and its replicas are in sync.

Production and Consumption

- These numbers can be seen in Control Center aggregated over:
 - the whole cluster
 - individual topic
 - individual consumer group
 - individual consumer instance and partition of a given consumer group

We will now examine these different Control Center views

hitesh@datacouch.io

The Cluster View

- Performance metrics (throughput and request latency)



Here on the slide we see graphs that show us overall cluster performance based upon relevant performance metrics such as throughput and request latency.

Topic Overview

The screenshot shows the Confluent Platform UI for the 'pageviews' topic. On the left, there's a sidebar with cluster navigation (C3 Cluster 1, CA Cluster 2) and various tools like ksqlDB, Consumers, Replicators, and Cluster settings. The main area displays production and consumption rates, availability, and a detailed table of partitions.

Production: 2099.666K Bytes per second

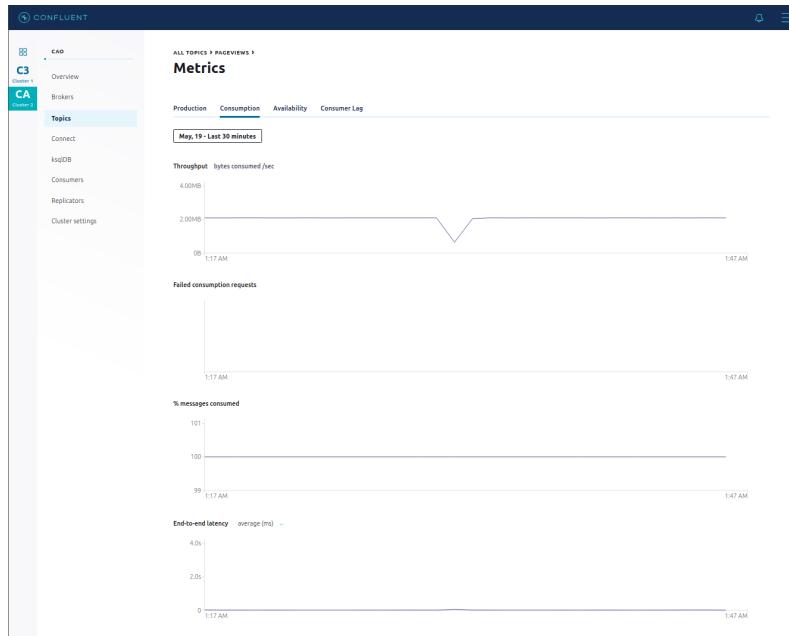
Consumption: 2100.57K Bytes per second

Availability: 0 of 6 Under replicated partitions | 0 of 18 Out of sync followers

Partitions	Partition id	Status	Replica placement	Followers (broker IDs)	Offset Start	Offset End	Size Total Size
0	0	Available	Leader (broker ID)	101, 102	0	1393809	293MB
1	1	Available	103	102, 101	0	1378676	290MB
2	2	Available	102	101, 103	0	1388725	292MB
3	3	Available	101	102, 103	0	1383275	291MB
4	4	Available	103	101, 102	0	1389593	292MB
5	5	Available	102	103, 101	0	1383543	291MB

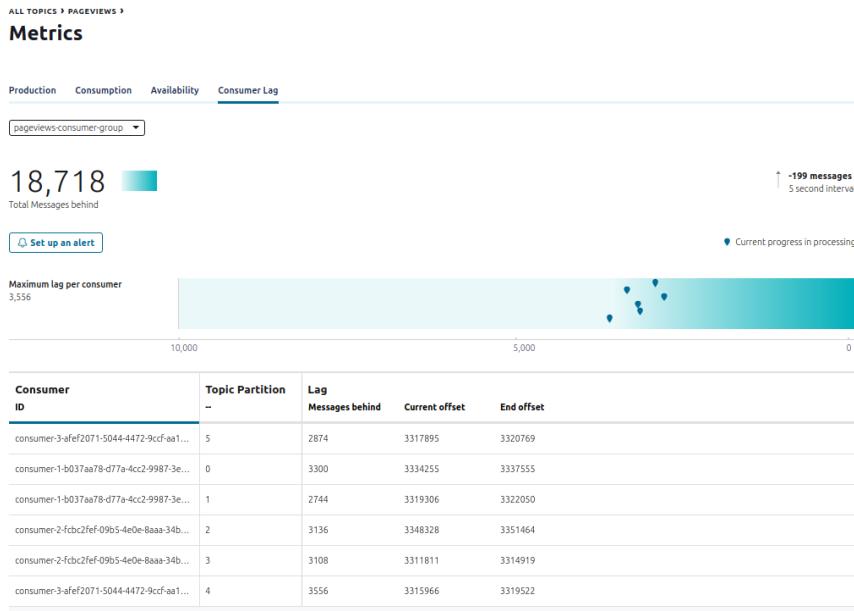
The initial Topics pane provides a list of all topics in the cluster. Selecting a topic from the list opens an overview pane that reflects current production and consumption rates, partition status and size, and replica placement.

Topic - Message Consumption



If we have enabled C3 monitoring interceptors for our producers and consumers, we can observe end-to-end consumption and latency for our event stream in the C3 topic consumption view. This makes it easy to spot any anomalies that may have occurred or be occurring in these areas for the event stream.

Topic - Consumer Lag



The Topic Consumer Lag tab reflects the overall message consumption status by all consumers. It provides a intuitive means to determine if consumer groups need to be scaled up or down.

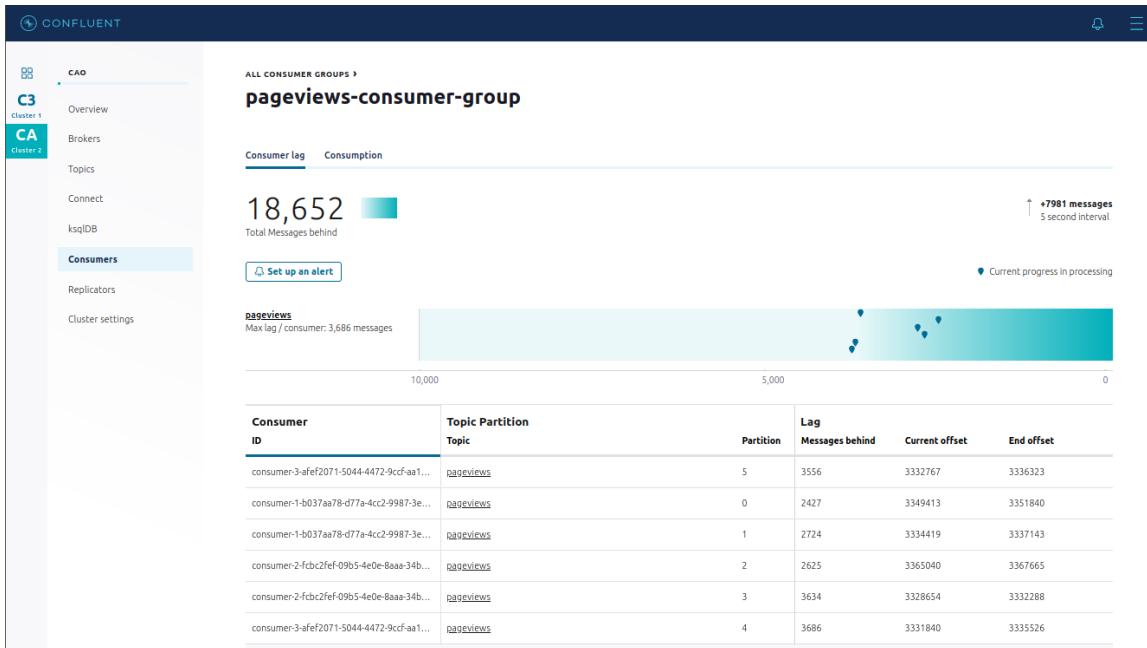
Consumers - All Consumer Groups

- The **All consumer groups** view lists all consumer groups in the cluster and the following metrics for each group:
 - consumer lag
 - number of consumers
 - number of subscribed topics

The screenshot shows the Confluent Platform interface. On the left, there's a sidebar with two clusters: Cluster 1 (C3) and Cluster 2 (CA). Under Cluster 2, the 'Consumers' tab is selected. The main content area is titled 'All consumer groups'. It features a search bar labeled 'Search consumer groups'. Below the search bar is a table with the following columns: 'Consumer group', 'ID', 'Messages behind', 'Number of consumers', and 'Number of topics'. A single row is visible in the table, corresponding to the consumer group 'pageviews-consumer-group'. The 'Messages behind' value is 41,808, and both 'Number of consumers' and 'Number of topics' are 1.

Consumer group	ID	Messages behind	Number of consumers	Number of topics
pageviews-consumer-group		41,808	1	1

Consumers - Consumer Lag

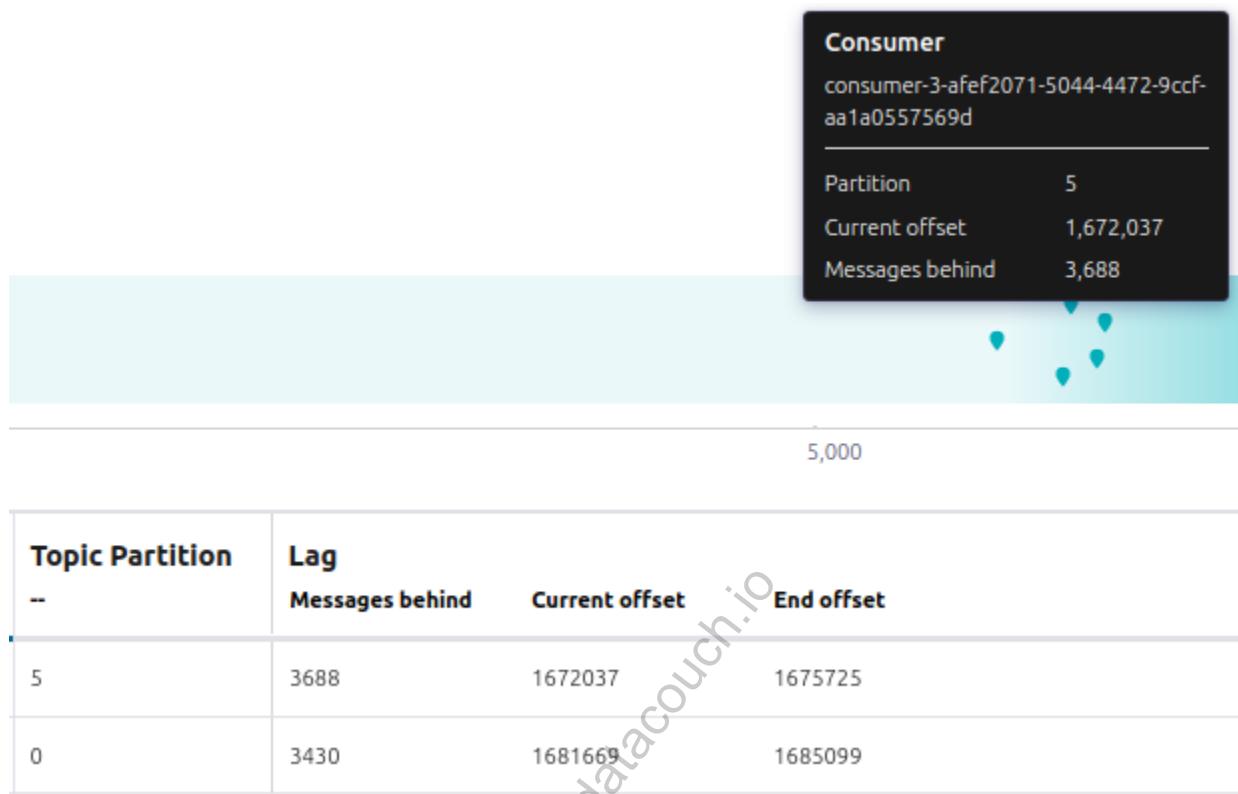


The Consumer lag tab shows a more detailed view of the current consumption progress for the selected consumer group. On the bottom we see a line item for each partition that is consumed. On top we see a graphical representation of the same.



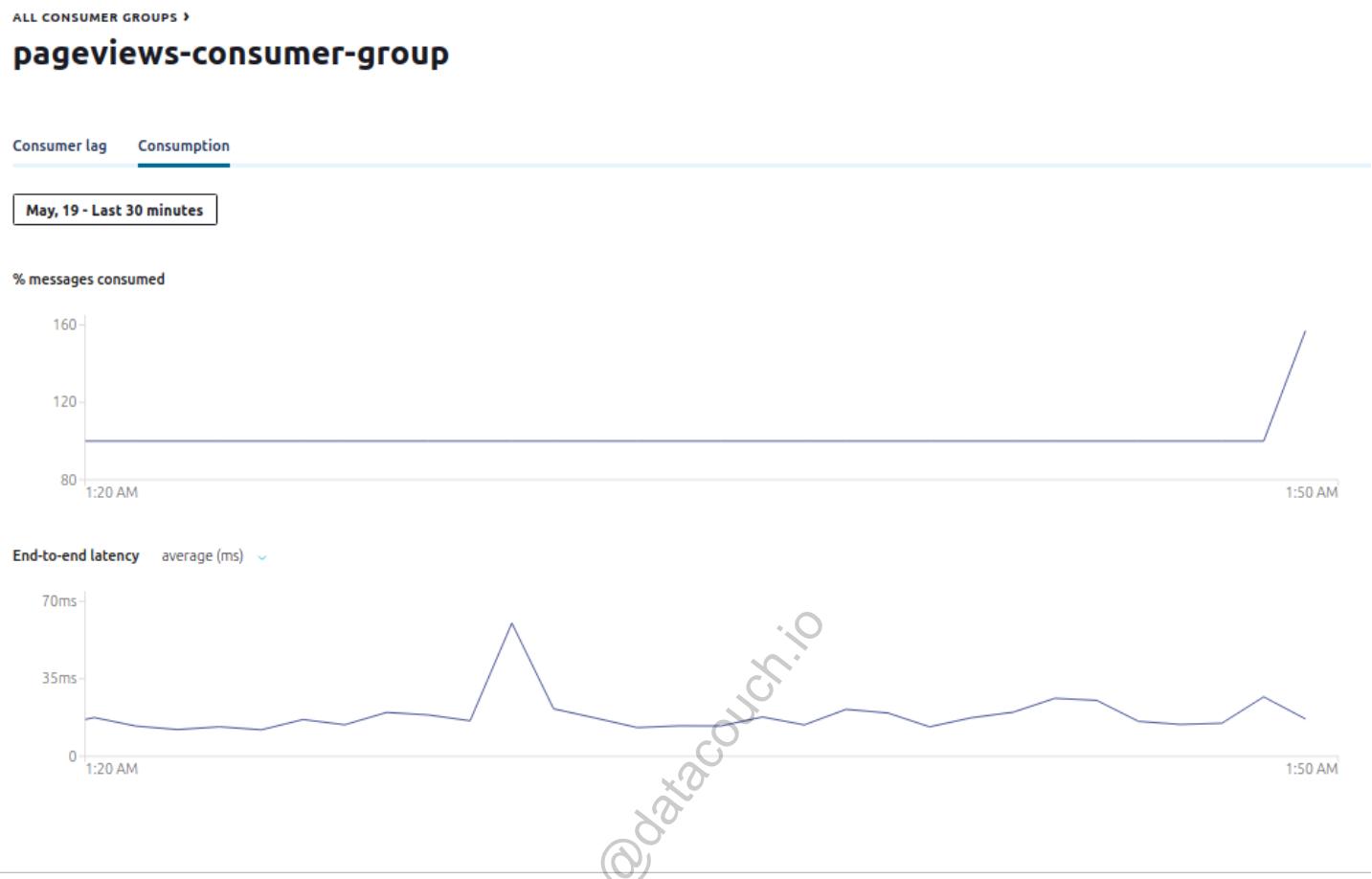
The consumer instances can be better distinguished in Control Center if you (optionally) give them a distinct `client.id`, e.g. `consumer-1,...,consumer-n`.

Consumers - Consumer Lag - Per Partition Details



If we click on one of the drop symbols in the previous chart then we can see more detailed information. We see which consumer it is, the partition number, the current offset, and the current consumption lag expressed as messages behind.

Consumers - Consumption



If we have enabled C3 monitoring interceptors for our producers and consumers, the Consumer Group Consumption tab shows performance metrics for the percentage of messages consumed and end-to-end latency.

Consumer Lag Alerts

ALERTS > OVERVIEW > TRIGGERS >
New trigger

General

Trigger name*

Components

Component type*
Consumer group

Criteria

Metric* Buffer (seconds)*
Condition* Value*

Submit **Cancel**

Knowing that the consumer lag is exceeding a certain threshold is essential. We can define alerts in C3 when the consumer lag does not meet certain criteria. In the image we see alert **High consumer lag** has been defined for the situation that the maximum latency exceeds 5000 ms (= 5 seconds). Remember, bigger lag means bigger latency.
Other possibilities are **average latency**, **consumption difference** and **consumer lag**.

Monitoring Interceptors

Producers

```
bootstrap.servers=kafka-1:9092,kafka-2:9092
key.serializer=io.confluent.kafka.serializers.KafkaAvroSerializer
value.serializer=io.confluent.kafka.serializers.KafkaAvroSerializer
schema.registry.url=http://schema-registry:8081
interceptor.classes=io.confluent.monitoring.clients.interceptor.MonitoringProducerInterceptor
```

Consumers

```
...
interceptor.classes=io.confluent.monitoring.clients.interceptor.MonitoringConsumerInterceptor
```

Many of the features that are described in this module are dependent on the use of Java interceptors in the Producers and Consumers. They need to be configured accordingly to enable Control Center to capture the relevant metrics.

- Producers are configured with the **MonitoringProducerInterceptor**
- Consumers are configured with the **MonitoringConsumerInterceptor**

Review



Question:

Control Center puts a special focus on **consumer lag**. Why is it so important to monitor those metrics?

hitesh@datacouch.io

Further Reading

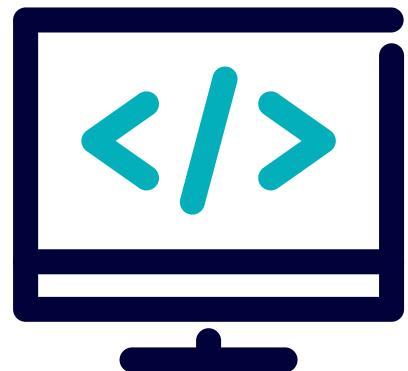
- Monitoring Your Apache Kafka® Deployment End-to-End:
<https://www.confluent.io/monitoring-your-apache-kafka-deployment>
- Manage, Monitor and Understand the Apache Kafka Cluster:+ <https://www.confluent.io/confluent-control-center/>
- 1. Intro | Monitoring Kafka in Confluent Control Center:
<https://www.youtube.com/watch?v=9myx2FtWQCI>
- Demo: Monitoring Kafka Like a Pro in Confluent Control Center:
<https://www.youtube.com/watch?v=O9LqDGSoWaU>

hitesh@datacouch.io

Lab: Monitoring with CCC - Data Streams

Please work on **Lab 4a: Monitoring with CCC - Data Streams**

Refer to the Exercise Guide



hitesh@datacouch.io

Branch 2: General Troubleshooting & Tuning - Overview



**CONFLUENT
Global Education**

Agenda



This is a branch of our CAO content on General Troubleshooting & Tuning. It is broken down into the following modules:

4. General Troubleshooting
5. Where are my System Log Files?

hitesh@datacouch.io

05: General Troubleshooting



CONFLUENT
Global Education

hitesh@datacouch.io

Module Overview



This module contains three lessons:

1. Troubleshooting Intro
2. Production Down - Troubleshooting Strategies
3. Troubleshooting Toolbelt

hitesh@datacouch.io

a: Troubleshooting Intro

Description

Troubleshooting motivation. Overview of data flow. Review of various settings.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Justify why learning how to troubleshoot is important
- Sketch a typical data flow in a Kafka based streaming platform and indicate where problems can happen on the way

hitesh@datacouch.io

Kafka is a Critical Piece of our Pipeline

- Can we be 100% sure that our data will get there?
- Can we lose messages?
- How do we verify?
- What is the root cause?



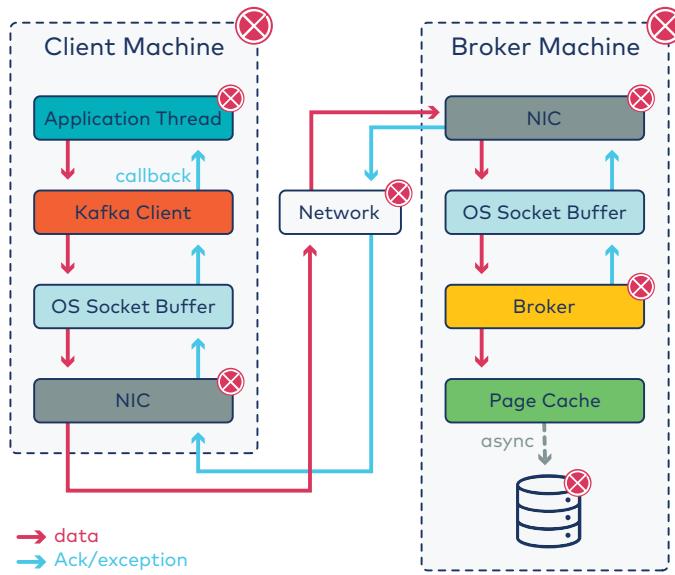
hitesh@datacouch.io

Distributed Systems

- Things fail
- Systems are designed to tolerate failures
- We must configure our system to handle them

hitesh@datacouch.io

Data Flow



There are many possible points of failure on the road of a typical request-response path.

Request:

- A data packet originating from our client application is sent via Kafka client library to the OS socket buffer
- From the socket buffer it flows via the NIC (network interface card) onto the network (intra and extranet)
- It is flowing through the broker's NIC to the OS socket buffer
- From there it's picked up by the broker and written into the page cache.
- The page cache is eventually and asynchronously flushed to the physical disk

Response:

- The response from the broker flows the same way back to the client application in inverse order.
- The response can be an ACK or an exception ---

Data Flow

Possible failures: In a distributed system things will fail. Here we have the following points of failure:

- client machine as a whole
- the client application thread crashes
- the client NIC malfunctions
- the network fails due to partition or congestion
- the broker's NIC fails
- the broker crashes or is unresponsive
- we have a disk failure on the broker
- etc.

We need to be able to troubleshoot all of these cases and more...

hitesh@datacloud.io

Be safe, not sorry

- Producer Settings
 - `max.block.ms` value is appropriate
 - `delivery.timeout.ms` is appropriate
 - `acks=all`
 - `enable.idempotence=true`
 - `producer.close()`
- Broker Settings
 - `unclean.leader.election.enable=false`
- Topic Settings
 - `replication.factor=3` (or more)
 - `min.insync.replicas=2`
- Consumer Settings
 - `auto.offset.commit=false`
- Commit **after** processing
- Monitor!

To have the highest possible guarantee of success in "... the data arrives where it should...", consider the best practices listed on the slide.

Review



Question:

What type of expertise does one need to troubleshoot a streaming platform powered by Kafka?

Provide a list of skills and provide reasons on why they are important.

hitesh@datacouch.io

Further Reading

- Lessons learned from Kafka in production: <https://www.youtube.com/watch?v=1vLMuWsfMcA>

hitesh@datacouch.io

b: Production Down! Troubleshooting Strategies

Description

Overview of dimensions of the problem space for troubleshooting and levels of severity.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Author a prioritized list of actions to follow in case of production down
- Categorize production issues into problem domains
- Differentiate immediate (hot-) fixes from mid- to long- term fixes for a production issue
- Assess risks of a hotfix

hitesh@datacouch.io

What am I looking for?

Severity:

Symptom	Action
Production down	immediate solution
Production negatively affected	quick solution
Significant friction	open support ticket
Minor issue	report & find work around

Category:

- Infrastructure
- Confluent Platform
- Application

First let's look at the (perceived) severity of the problem:

- If we have a production down then we need an immediate solution. Every minute of downtime in a mission critical environment cause loss in dollars and reputation. The latter is often more severe than the former.
- In this case we do not have time for a long investigation and we need to postpone this to a post mortem analysis phase which happens AFTER production is up again.
- If production is severely negatively affected then we need a quick solution. We should still carefully analyze the situation to possibly find the root cause and plan our actions. We must not start to change anything without making sure we have a very recent backup to rollback the system if something goes wrong during our troubleshooting.
- Users of Confluent Enterprise report significant friction in the usage of the product to infrastructure and operations support. Evaluate the root cause of the problem. What exactly are the users trying to do? Is the friction caused by a defect or a missing feature? In either case report it to Confluent as an issue or feature request.
- Users encounter minor issues in the usage and operation of Confluent Enterprise: the quickest solution is probably to find a work around. Also investigate the root cause. Is it a bug or a missing feature? Report it to Confluent as issue or feature request. In any case, urgent or not, we should always have a recent backup before we start making possibly irreversible changes to the environment.

Secondly we also need to look at the category of the problem:

- Is the problem related to the infrastructure of our production environment, e.g. physical servers, routers, firewalls, virtualization software like VMWare, etc.?
- Or is the problem one originating in the Confluent platform such as Brokers, Kafka Connect, Schema Registry, etc.?
- Finally could it be that the problem is one caused by the application services such as Kafka Streams and ksqlDB applications?

hitesh@datacouch.io

Asking the right questions

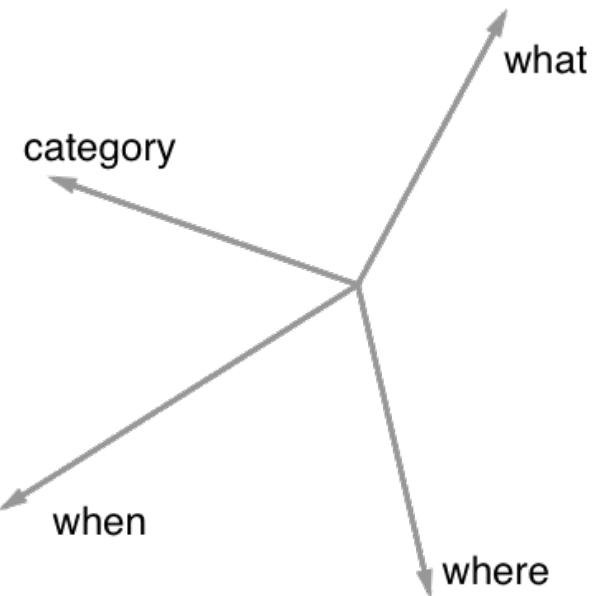
- **What** happened?
- **When** did it happen?
- **Where** did it happen?
- **Category** of the problem
- **Do** we have a support dump?
- **What** did you do?



-
- First we ask "what" happened. What were the symptoms or side effects that were observed. This might not be the root cause but we should still get as much information here as possible. It will help us reduce the problem space.
 - Next we ask "when" (date and time) that the event happened. Try to be as precise as possible. Do not forget to consider the time zone! The Confluent Platform produces loads of logging information in a somewhat busy environment and thus the closer you get (ideally minutes) the better.
 - Asking about the "when" is important since we might find plenty of helpful information in the logs generated during that time.
 - Now we also need to know about the "where" the event happened. The bigger the cluster, the harder it is to find the root cause if we have no idea about where the unexpected event happened. Was it broker related, or schema registry? Was it on a specific cluster node, etc.? The more we know about the "where" the easier it is for us to navigate through the support dump (if available).
 - The next interesting question is about the supposed category of the problem. Is it related to the ZooKeeper ensemble, the Kafka broker cluster, Schema Registry, Kafka Connect, etc.? Is it a resource problem (RAM, I/O, etc.) or maybe a networking problem (routing, throughput and latency)? Is it a permission (ACLs or SELinux, etc.) problem?
 - To get conclusive answers, it is absolutely necessary that we have data. This data we get in the form of the "support dump" that we can acquire by gathering all the logging data, host, OS and container metrics, etc..
 - We now also need to ask the relevant person "what did you do?". This is not about blaming someone but about solving a problem at hand as quickly and as efficiently as possible. Thus insist that the relevant operators describe to you as precisely as possible what action they did right before, during and after the undesired event. Once again: here it is **not about blaming** but about finding the root cause a.s.a.p.

Reducing the Problem Space

- Multi-dimensional problem space
- Dimensions: what, when, where, event category, etc.
- Pinpoint as many dimensions as possible



- On the previous slide we have discussed that asking the right questions is key to a good problem solving strategy.
- Here we look at the same topic from a more theoretical perspective. Ultimately the outcome is the same, but it is expressed in a more formalized way.
- The problem space is a multi-dimensional one. The more dimensions there are the more difficult it is to locate the root cause of a problem.
- Typical dimensions are: **what, when, where or event category**, etc.
- The primary goal is to reduce the number of dimensions by eliminating them as possible factors.
- If we e.g. know the exact time an event happened (the "when") then we can reduce the problem by this dimension since we just need to look/investigate at the given date/time. The same applies to the "where"; if we know that the problem occurred on server "abc-123" then we can again reduce the problem-space by the "where" dimension since we can reduce our search on this single node of our whole cluster.

Review



Question:

Which is the best way to troubleshoot consumer lag? Justify your answer.

hitesh@datacouch.io

Further Reading

- Troubleshooting 201: Ask the Right Questions: <http://bit.ly/2zSu5eO>
- Problem Solving & Asking The Right Questions: <http://bit.ly/2zTv3Hi>

hitesh@datacouch.io

c: Troubleshooting Toolkit

Description

Tour of troubleshooting tools.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Identify the relevant log files for a given production issue
- Locate the relevant log files for each product of the Confluent platform
- Extract relevant information from the log file at hand
- Aggregate logs of my production system in a central location

hitesh@datacouch.io

Linux Performance Observability Tools

Memory:	I/O:	Network Stack:
• <code>free</code>	• <code>iostat</code>	• <code>netstat</code>
• <code>top</code> , <code>htop</code>	• <code>iotop</code>	• <code>netcat</code>
• <code>vmstat</code>	• <code>blktrace</code>	• <code>tcpdump</code>
• ...	• ...	• <code>iptraf</code>
		• <code>ethtool</code>
		• ...

On the slide we see the most important Linux tools and utilities that help us troubleshoot memory, I/O and networking performance issues.

- **free:** Displays the amount of free memory in the system
- **top, htop:** displays Linux processes
- **vmstat:** Reports virtual memory statistics
- **iostat:** Report CPU and I/O statistics for devices and partitions
- **iotop:** Simple `top` like I/O monitor
- **blktrace:** Generate traces of the I/O traffic on block devices
- **netstat (nc):** Print network connections, routing tables, interface statistics, masquerade connections and multicast memberships
- **netcat (nc):** Arbitrary TCP and UDP connections and listens
- **tcpdump:** Dump traffic on a network
- **iptraf:** An IP traffic monitor that shows information on the IP traffic passing over the network.
- **ethtool:** Query and control network driver and hardware settings

Netshoot Container

- Contains many recommended tools
- Troubleshoot application container network:

```
$ docker run -it --net container:<container_name> nicolaka/netshoot
```

- Troubleshoot host network:

```
$ docker run -it --net host nicolaka/netshoot
```

-
- The netshoot container includes a set of powerful tools as recommended on the previous slide.
 - The advantage of this container is that we have all the tools we need at hand without a need to install them on a host. Every host of interest can run Docker containers.
 - We can debug 2 scenarios/contexts:
 - analyze a process running in the container's (Linux) namespace
 - troubleshoot the root (Linux) network namespace (i.e. the host)

Kafka Command Line Tools

- `zookeeper-shell`
- `kafka-configs`
- `kafka-topics`
- `kafka-consumer-groups`
- `kafka-acls`
- `kafka-console-consumer`,
`kafka-console-producer`
- `kafka-avro-console-consumer`,
`kafka-avro-console-producer`
- `kafka-producer-perf-test`,
`kafka-consumer-perf-test`

There exist various Kafka command line tools to troubleshoot problems in the following areas

- `zookeeper-shell`: Utility to admin a ZooKeeper instance from the command line
- `kafka-configs`: Add and remove entity config for a topic, client, user or broker
- `kafka-topics`: Create, alter, list, and describe topics
- `kafka-console-consumer`: Read data from a Kafka topic and write it to standard output
- `kafka-console-producer`: Read data from standard output and write it to a Kafka topic
- `kafka-avro-console-consumer`: Read Avro data from a Kafka topic and write it to standard output
- `kafka-avro-console-producer`: Read data from standard output and write it formatted as Avro to a Kafka topic
- `kafka-consumer-groups`: List all consumer groups, describe a consumer group, delete consumer group info, or reset consumer group offsets
- `kafka-producer-perf-test`: This tool is used to verify the Kafka producer performance
- `kafka-consumer-perf-test`: This tool is used to verify the Kafka consumer performance
- `kafka-acls`: Manage ACLs in Kafka

The **kafkacat** Tool

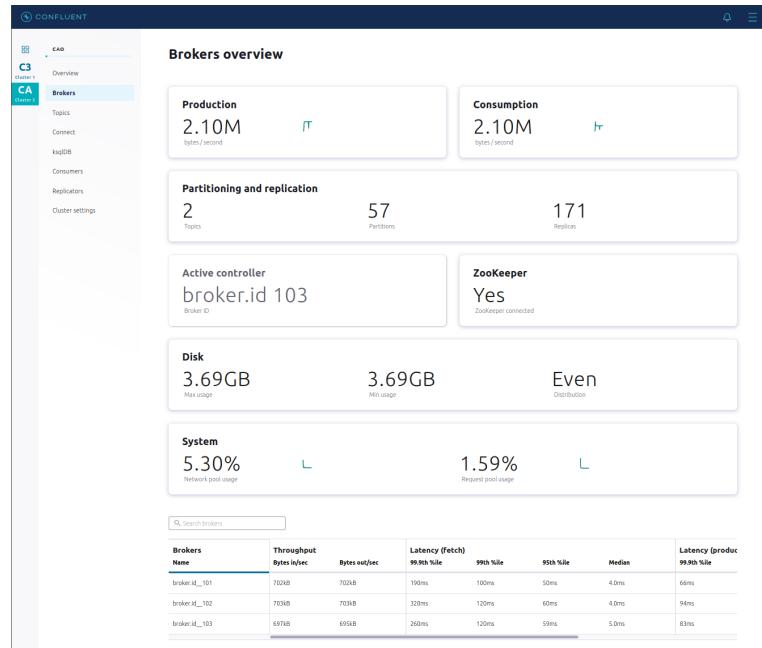
- Produce, consume and list topic and partition information
- **netcat** for Kafka

```
$ kcat \
  -b localhost:9092 \
  -t keyed_topic -C \
  -f 'Key: %k\nValue: %s\n'
Key: 1
Value: foo
Key: 2
Value: bar
```

-
- **kafkacat** is a command line utility that you can use to test and debug Apache Kafka deployments
 - You can use **kafkacat** to produce, consume, and list topic and partition information for Kafka
 - Described as "netcat for Kafka", it is a swiss-army knife of tools for inspecting and creating data in Kafka

Confluent Control Center

- Broker health and configurations
- Topics and partitions
- Data flow & end-to-end latency
- Consumer groups
- Replication



- Confluent Control Center (C3) is very helpful to troubleshoot the following scenarios
 - Broker health and configurations
 - Topics and partitions
 - Data flow & end-to-end latency
 - Consumer groups
 - Replication

Review



Question:

Which is the best way to troubleshoot consumer lag? Justify your answer.

hitesh@datacouch.io

Further Reading

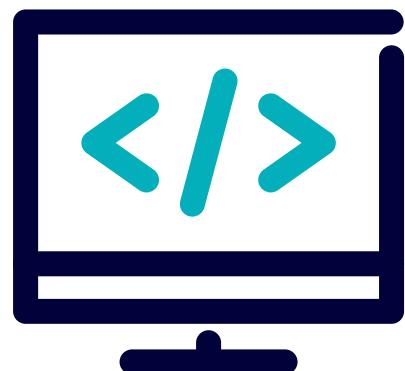
- Linux Performance: <http://www.brendangregg.com/linuxperf.html>
- **netshoot** Container: <http://bit.ly/2z2xGn4>
- **kafkacat** Utility: <https://docs.confluent.io/current/app-development/kafkacat-usage.html>
- Performance Testing: <https://cwiki.apache.org/confluence/display/KAFKA/Performance+testing>

hitesh@datacouch.io

Lab: Troubleshooting Toolkit

Please work on **Lab 5a: Troubleshooting Toolkit**

Refer to the Exercise Guide



hitesh@datacouch.io

06: Where are my System Log Files?



CONFLUENT
Global Education

hitesh@datacouch.io

Module Overview



This module contains one lesson:

1. Where are my System Log Files?

hitesh@datacouch.io

a: Where are my System Log Files?

Description

Overview of Kafka system logs and tips for analyzing them.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Identify the relevant log files for a given production issue
- Locate the relevant log files for each product of the Confluent platform
- Extract relevant information from the log file at hand
- Aggregate logs of my production system in a central location

hitesh@datacouch.io

Finding the relevant information

Questions to ask:

- What is the event?
- What is the date/time of the unexpected event?
- Which component of the platform is affected?
- Which instances/nodes of the cluster are affected?



Log files can be very large

-
- The reason why we need access to log files in the first place is that something unusual or unexpected happened.
 - The log files obtained are potentially very large and we might get lost in the sea of information if we don't have a strategy on how to narrow down the area of investigation.
 - From the operator(s) try to get the kind of event or behavior that was discovered or observed
 - Also try to get as precisely as possible the date and time the unusual or unexpected event happened or was discovered.
 - Try to get the information about which components or nodes were affected.
 - It is advised to use some **regex** based utilities to parse through the huge log files. They offer powerful means of searching for keywords using exact or pattern matching. For learners that are not familiar with such tools you might suggest the Linux **less** or **more** tools.

Analyzing large log files

- Use `less` tool:

```
$ less /var/log/kafka/kafkaServer-gc.0.current
```

or

```
$ docker-compose logs kafka | less
```

- Use `grep` tool:

```
$ docker-compose logs kafka | grep -i -E "(stopped|start(ing|ed))"
$ docker-compose logs kafka | grep -i -E "started" | wc -l
$ grep '<my pattern>' * -R
```

- To browse through potentially huge log files preferably use "less" (or "zless", or "more" [outdated])
- `less` supports powerful searching with pattern matching.
- In the code snippet we see a sample that searches for all occurrences of "stopped" or "starting" or "started" in the respective file.
- The second grep command counts the occurrences of "stopped" in the respective file
- The third one searches for <my pattern> recursively through multiple files.
- To search for a pattern in a single or multiple files use "grep"

Naming conventions

Component	Component Name	Subfolder
ZooKeeper	KAFKA	kafka
Kafka broker	KAFKA	kafka
Confluent Control Center	CONTROL_CENTER	confluent-control-center
Schema Registry	SCHEMA_REGISTRY	schema-registry
REST Proxy	KAFKA_REST	kafka-rest
Kafka Connect	CONNECT	kafka
ksqlDB Server	KSQSL	ksql

To find the relevant log files or to define the necessary environment variables correctly it is important to be aware of the naming conventions used in our event streaming platform. It is mostly logical but not always as you can see on the example of ZooKeeper and Kafka Connect.

- Many paths to e.g. configuration files contain the name of the respective component/subfolder, e.g.
`/etc/<subfolder-name>/log4j.properties`
- many **environment variables** used have to be prefixed with the `<component_name>`

The naming convention is defined in the table on the slide. Note that ZK uses "KAFKA" as component name

Logging Framework of Confluent Platform

- Kafka brokers and clients use the `slf4j` logging abstraction
- **Log4J** is the default logging framework used in all components
- **Log4J** is configured via properties files
- Default location of Log4J config file:
`/etc/<component-name>/log4j.properties`
- `log4j.properties` defines log target
 - STDOUT, STDERR
 - Files
- Also defines log level:
`TRACE, DEBUG, ..., ERROR, FATAL`

-
- `log4j` is the default logging framework, but it's pluggable. Kafka brokers and clients use the `slf4j` logging abstraction, which supports many logging backends.
 - The default location of `log4j.properties` for the Confluent Platform is
`/etc/<component>/log4j.properties`.

Log4j Configuration in Production Systems

1. For key `log4j.rootLogger`, replace `stdout` with `kafkaAppender`
2. For key `log4j.appenders.X` use `RollingFileAppender`
3. Remove key `log4j.appenders.X.DatePattern`
4. Set `MaxFileSize=100MB`, `MaxBackupIndex=10`

```
log4j.rootLogger=INFO, kafkaAppender
...
log4j.appenders.stateChangeAppender=org.apache.log4j.RollingFileAppender
log4j.appenders.stateChangeAppender.File=${kafka.logs.dir}/state-change.log
log4j.appenders.stateChangeAppender.layout=org.apache.log4j.PatternLayout
log4j.appenders.stateChangeAppender.layout.ConversionPattern=[%d] %p %m (%c)%n
log4j.appenders.stateChangeAppender.Append=true
log4j.appenders.stateChangeAppender.MaxBackupIndex=10
log4j.appenders.stateChangeAppender.MaxFileSize=100MB
...
```

Note, the following applies when **NOT** running in containers!

In production system, the default Log4j configuration must be modified slightly to ensure the log files do not fill up the disk. To do that, change the following configuration in the relevant `log4j.properties` (X is the Log4j appender name, like `kafkaAppender`):

1. For key `log4j.rootLogger`, replace `stdout` with `kafkaAppender` to ensure syslog does not redirect it to a file
2. For key `log4j.appenders.X`, replace `org.apache.log4j.DailyRollingFileAppender` with `org.apache.log4j.RollingFileAppender` to use the rolling file feature
3. Remove key `log4j.appenders.X.DatePattern`
4. Add these lines to set maximum file size as `100MB` and 10 backup log files (so the total size used for this appender is `1000MB` for backup log files PLUS up to `100MB` for the current log file):

```
log4j.appenders.X.Append=true
log4j.appenders.X.MaxBackupIndex=10
log4j.appenders.X.MaxFileSize=100MB
```



The `log4j.properties` files for other components (e.g. schema registry, connect, etc) should be modified as above in production to ensure the log files never fill up the disk.

hitesh@datacouch.io

Running in Containers

- Always log to STDOUT and STDERR
- Default location of Log4J config file:

```
/etc/<component>/log4j.properties
```

- Override with own config:

```
-Dlog4j.configuration=file:/path/to/log4j.properties
```

When running your zookeeper, broker, Kafka client, etc. in a container always log to STDOUT and STDERR and never e.g. into a file. This way the usual log aggregators can collect the logs from your producer.

The standard logging configuration can be replaced by a custom `log4j.properties` files upon start of the component using Java's command line parameter `-D`

What to collect?

- Log files of all Confluent Platform components
- System information:

```
echo $HOSTNAME  
cat /etc_host/*release  
cat /proc/version  
cat /proc/cpuinfo  
cat /proc/meminfo  
cat /proc/cgroups  
cat /proc/self/cgroup  
df -h  
mount  
vmstat 1 5  
iostat 1 5  
dmidecode
```

- When running in containers

```
$ docker version  
$ docker system info  
$ docker container stats --all --no-stream  
$ sudo cat /etc/docker/daemon.json  
$ journalctl -t dockerd --no-pager  
  
# from: cnfl.io/check-config  
$ ./check-config.sh
```

- Certificates

```
$ openssl x509 -in ca.pem -text  
$ openssl ec -in key.pem -text
```

- The most important information to identify an issue most often stems from the log files generated by the malfunctioning component of the Confluent platform such as broker logs if a broker is involved.
- But additional information about the underlying infrastructure such as the OS or the container on which the malfunctioning component is running, is very helpful too and often essential
- Thus it is advised to collect such information as well when troubleshooting a system. Here is a list of candidates:
 - Collect system information such as the one suggested in the script on the slide (Note on CentOS one might need to first install `sysstat`)
 - On a system using container use commands listed to gather valuable information relative to Docker
 - Use the `check-config.sh` script from here: <https://github.com/moby/moby/blob/master/contrib/check-config.sh>
 - On a secure system collect clear text information about the certificates used. Use `openssl` to retrieve the information using commands similar to the ones shown on the slide

Brokers - Log4J Logs

Log Name	Description
server.log	Basic broker operations (segment rolls, replica fetching, etc.)
state-change.log	Updates received from and sent to controller
kafka-request.log	All information about all requests. Verbose, not great for live debugging. Use when no other options for root cause. Rogue clients, single request slowdowns, etc.
log-cleaner.log	Cleaner thread (related to compaction)
controller.log	Written to by active controller
kafka-authorizer.log	Secure connections only (set to DEBUG if you want successful connections)

Each broker produces a lot of logging information using Log4J. If running natively with the default configuration we have the log files listed on the slide on the respective host in the configured logging folder.

- The default Log4J config can be found here: <https://github.com/apache/kafka/blob/trunk/config/log4j.properties>
- All log files are stored at `${kafka.logs.dir}`. The default is `/var/log/kafka`.

Broker - Logging Options

- Define alternative logging config file
 - running natively:

```
$ export KAFKA_LOG4J_OPTS="-Dlog4j.configuration=file:/path/to/tools-log4j.properties"
```

- running in container:

```
docker run -d \
  --name=kafka \
  --net=sample-net
  -e KAFKA_BROKER_ID=101 \
  -e KAFKA_ZOOKEEPER_CONNECT=zookeeper:2181 \
  -e KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://kafka:9092 \
  -e KAFKA_LOG4J_OPTS="-Dlog4j.configuration=file:/path/to/tools-log4j.properties"
  \
  -e KAFKA_LOG4J_LOGGERS="kafka.controller=WARN,kafka.request.logger=DEBUG" \
  -e KAFKA_LOG4J_ROOT_LOGLEVEL=WARN \
  -e KAFKA_TOOLS_LOG4J_LOGLEVEL=ERROR \
  confluentinc/cp-kafka:6.0.0-1-ubi8
```

Use the environment variable `KAFKA_LOG4J_OPTS` to define an alternative `log4j.properties`.

Or when using containers then you can configure logging specific details using either of the four environment variables shown in the sample command on the slide:

- define an alternate log config file via `KAFKA_LOG4J_OPTS`
- To override the root log level use `KAFKA_LOG4J_ROOT_LOGLEVEL`
- To override individual logger levels use `KAFKA_LOG4J_LOGGERS`
- To override the logging levels for the tools use `KAFKA_TOOLS_LOG4J_LOGLEVEL`

Set Log Levels Dynamically

- Need more data to troubleshoot!
- **Cannot restart** brokers in production
- Solution: **dynamically** set log level using **kafka-configs**
- Example: set a logger level to DEBUG for broker 101

```
$ kafka-configs \
  --bootstrap-server kafka:9092 \
  --alter \
  --add-config "kafka.server.ReplicaManager=WARN,kafka.server.KafkaApis=DEBUG" \
  --entity-type broker-logger \
  --entity-name 101
```



Be sure to reset log level after successful debugging!

- When a production issue arises it is often helpful to change the log level from say WARN to DEBUG, to get more detailed data.
- But in production one cannot simply change the log level of a broker in its **log4j.properties** file and restart the broker.
- In this case one can use JMX to dynamically change the log level.
- **IMPORTANT:** do not forget to reset the log level after successful debugging
- A good tool to use to reset the log level is the JMX CLI called **jmxterm**

Kafka Clients - Producer & Consumer

- Alternative log config file:

```
-Dlog4j.configuration=file:/path/to/log4j.properties
```

- Typical config file when running in container:

```
log4j.rootLogger=WARN, stderr
log4j.appender.stderr=org.apache.log4j.ConsoleAppender
log4j.appender.stderr.layout=org.apache.log4j.PatternLayout
log4j.appender.stderr.layout.ConversionPattern=[%d] %p %m (%c)%n
log4j.appender.stderr.Target=System.err
```



When running your producer or consumer in a container always log to STDOUT and STDERR and never e.g. into a file. This way the usual log aggregators can collect the logs from your client.

Assuming the producer is given and its code cannot be modified then you can use the following command line parameter when running the producer or consumer:

```
-Dlog4j.configuration=file:/path/to/log4j.properties
```

Typical content of such a properties file:

```
log4j.rootLogger=WARN, stderr
log4j.appender.stderr=org.apache.log4j.ConsoleAppender
log4j.appender.stderr.layout=org.apache.log4j.PatternLayout
log4j.appender.stderr.layout.ConversionPattern=[%d] %p %m (%c)%n
log4j.appender.stderr.Target=System.err
```

Change WARN to DEBUG, INFO or ERR depending on the context

Any further details about the available log appenders other than ConsoleAppender and about the ConversionPattern please find here: <https://logging.apache.org/log4j/2.x/manual/configuration.html>



When asked about REST Proxy, Kafka Connect or Schema Registry: they can be configured the same way as producers and consumers. Just make sure to use the right <component name>.

ksqldb Server

Installation environment	How to access the ksqldb logs
Docker container	<pre>\$ docker logs <container ID></pre> <p>— or —</p> <pre>\$ docker-compose logs ksqldb-server</pre>
Confluent CLI	<pre>\$ bin/confluent logs ksqldb-server</pre>
RPM/DEB	Log files in folder <code>/var/log/confluent</code>

- Examine logs and look for deserialization errors:

```
$ docker-compose logs ksqldb-server | grep -iE "deserialization error"
```

ksqldb writes most of its logs to **stdout** by default.

- When running ksqldb Server in a Docker container then run the command on the slide. If you're running in development you can use Docker compose, otherwise just use the normal `docker logs` command.
- If you have installed the Confluent Platform natively in development using the Confluent CLI, then you can run the command shown on the slide, `$ confluent log ksqldb-server`
- If you've installed Confluent Platform using RPM/DEB (e.g. in production), then you should find the logs in the folder `/var/log/confluent/`

In the context of ksqldb Server the most common errors are deserialization errors! Look for them first.

Confluent Control Center

- Sample command to start Control Center:

```
$ docker run -d \
  --name=control-center --net=sample-net \
  --ulimit nofile=16384:16384 -p 9021:9021 \
  -e CONTROL_CENTER_BOOTSTRAP_SERVERS=kafka:9092 \
  -e CONTROL_CENTER_REPLICATION_FACTOR=1 \
  -e CONTROL_CENTER_MONITORING_INTERCEPTOR_TOPIC_PARTITIONS=1 \
  -e CONTROL_CENTER_INTERNAL_TOPICS_PARTITIONS=1 \
  -e CONTROL_CENTER_STREAMS_NUM_STREAM_THREADS=2 \
  -e CONTROL_CENTER_CONNECT_CLUSTER=http://localhost:8083 \
  -e CONTROL_CENTER_LOG4J_OPTS="-Dlog4j.configuration=file:/path/to/log4j.properties" \
  -e CONTROL_CENTER_LOG4J_ROOT_LOGLEVEL=DEBUG \
  confluentinc/cp-control-center:6.0.0-1-ubi8
```

- Control Center is up and running?

```
$ docker-compose logs control-center | grep -i -E started
```

The above command is a typical way to customize the logging behavior of Confluent Control Center, via the environment variables `CONTROL_CENTER_LOG4J_OPTS` and `CONTROL_CENTER_LOG4J_ROOT_LOGLEVEL`.

The easiest way to find out IF C3 is up and running is to parse the log for "started" like shown in the command snippet on the slide. The command there is valid if Kafka was started using Docker compose. Otherwise use the command `docker logs <CC ID> | grep -i -E started`

librdkafka Clients (1)

- `librdkafka` supports logging
- Use property `debug` to enable contexts
 - Producer: `debug=broker,topic,msg`
 - Consumer: `debug=consumer,cgrp,topic,fetch`
- The `librdkafka` client library upon which most of the non-Java or Scala clients are built supports logging
- `librdkafka` has a config property `debug` which accepts a comma-separated list of debug contexts to enable
 - Possible debug contexts are: `generic, broker, topic, metadata, feature, queue, msg, protocol, cgrp, security, fetch, interceptor, plugin, consumer, admin, eos, all`
 - Details for the available contexts are provided in the GitHub repo for `librdkafka`
- Detailed Producer debugging: `debug=broker,topic,msg`
- Consumer debugging: `debug=consumer,cgrp,topic,fetch`
- To get detailed Producer debugging set: `debug=broker,topic,msg`
- To get detailed Consumer debugging set: `debug=consumer,cgrp,topic,fetch`

librdkafka Clients (2)

Python

```
logger = logging.getLogger('producer')
logger.setLevel(logging.DEBUG)

p = confluent_kafka.Producer({'debug': 'all'},
                             logger=logger)
```

-
- On the slide we have a code snippet on how to configure logging in a Python client

hitesh@datacouch.io

librdkafka Clients (3)

DOT.NET

```
var producerConfig = new ProducerConfig
{
    BootstrapServers = bootstrapServers,
    Debug = "all"
};
using (var producer =
    new ProducerBuilder<byte[], byte[]>(producerConfig)
        .SetLogHandler((_, m) => logCount += 1)
        .Build())
{ ... }
```

- On the slide we have a code snippet on how to configure logging in a .NET client
- In the case of .NET, if the log handler is not defined then the application simply logs to **STDERR**

Review



Question:

Why is it so important to collect all logging information possible from a production system?

hitesh@datacouch.io

Further Reading

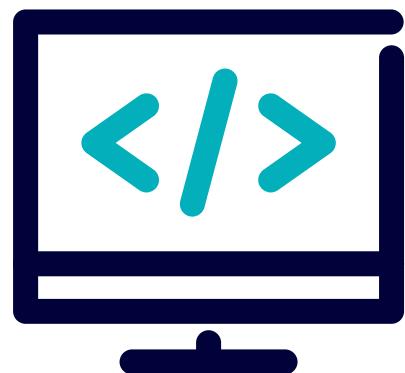
- Apache Log4j 1.x: <https://logging.apache.org/log4j/2.x/manual/compatibility.html>
- How to configure logging for Kafka producers?
<https://stackoverflow.com/questions/35773780/how-to-configure-logging-for-kafka-producers>
- Troubleshooting KSQL – Part 1: Why Isn't My KSQL Query Returning Data?:
<https://www.confluent.io/blog/troubleshooting-ksql-part-1>

hitesh@datacouch.io

Lab: Where are my Log Files?

Please work on **Lab 6a: Where are my Log Files?**

Refer to the Exercise Guide



hitesh@datacouch.io

Branch 3: Troubleshooting & Tuning Central Services - Overview



**CONFLUENT
Global Education**

Agenda



This is a branch of our CAO content on Troubleshooting & Tuning Central Services. It is broken down into the following modules:

7. Troubleshooting & Tuning ZooKeeper
8. Troubleshooting & Tuning Brokers
9. Troubleshooting & Tuning Schema Registry

hitesh@datacouch.io

07: Troubleshooting & Tuning ZooKeeper



CONFLUENT
Global Education

Module Overview



This module contains one lesson:

1. Troubleshooting Zookeeper

hitesh@datacouch.io

a: Troubleshooting ZooKeeper

Description

Overview of ZooKeeper's responsibilities and tips for managing ZooKeeper.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Sketch an architecture for my company that avoids ZooKeeper related outages
- Justify the number of ZK instances needed in my Confluent Platform
- List the root causes that can lead to a loss in ZK quorum

hitesh@datacouch.io

ZooKeeper's Responsibilities

- Topic configuration
- Controller election
- Access Control Lists (ACLs)
- Client Quotas
- Cluster membership
- Info about leaders
- Info about ISRs
- Dynamic broker configurations

Here is the list of current responsibilities of ZooKeeper. This will change over time as it is the intent to eventually remove ZK as a requirement and make Kafka self sufficient for all management purposes.

- Topic configuration—ZK is the source of truth for topic configuration
- Controller election—the broker acting as the controller is chosen by ZK
- Access Control Lists (ACLs)—authorization for resources is stored and synchronized by ZK
- Client Quotas—how much data each Kafka client is allowed to read or write is governed by what is stored in ZK
- Cluster membership—which brokers are actually active in the cluster is stored in a ZK node
- Info about leaders: The leader for each partition of each topic is stored in ZK
- Info about ISRs: information about in-sync replicas is stored in ZK,
- Dynamic broker configurations: ZK stores all dynamically configured broker values (see KIP 226: <https://cwiki.apache.org/confluence/display/KAFKA/KIP-226+-+Dynamic+Broker+Configuration>)

Tuning ZooKeeper

- Dedicated ZooKeeper cluster
- Keep ZK instances close
- Dedicated disks for TX logs
- Dedicated server for each ZK instance
- Turn **swappiness** to 1 or disable it
- Servers in different AZs
- Enough RAM
- At least 512MB Heap (1GB recommended)
- 3 to 5 instances in Quorum
- Use static IP addresses



ZK ensemble down == Kafka limited!

What is ZooKeeper sensitive to?

ZooKeeper has a narrowly scoped, well defined mission as the service that keeps all of the other services in sync. As such it needs to be in a situation that lends itself to low latency.

Hardware Considerations

ZooKeeper does not require extensive hardware resources, but it is often helpful to provide dedicated resources. Here are some examples of hardware configuration considerations to use for the ZooKeeper quorum:

- Dedicated disks for transaction log directories specified by `dataLogDir`. Provision ZooKeeper server with three dedicated drives: one for the root file system, one for the snapshot, and another for the transaction log. Store the snapshot and transaction log on SSDs
- Dedicated server for each member of the zookeeper quorum
- Deploy servers in different racks where possible to avoid correlated crashes
 - Turn **swappiness** to minimum or disable it (depends on the kernel version). In newer Linux kernels it's recommended to set `vm.swappiness` to 1.

- Enough physical memory as to not overcommit if maximum heap size is required
- Tune the JVM with the same JVM settings used for Kafka. Except: A (smaller) heap size of 1 GB is recommended for most use cases (the JVM is tuned by default in Confluent Platform). Also: Monitor the heap usage to ensure no delays are caused by garbage collection!

Service-level Considerations

As a service, ZooKeeper should have the ability to perform elections and read/write information to disk within reliably consistent timeframes. Here are some considerations at the service level to help accomplish that goal:

- At least 512Mb of heap is recommended, if any evidence of garbage collection pauses, immediately consider actions to rectify this
- 5 or 7 ZooKeeper servers in a quorum. Note an odd number of servers is required to perform reliable leader elections. A 3-node ZooKeeper cluster will have lower write latency, but can only survive a single failure

Static IP Considerations

ZooKeeper servers should not be hosted where server IP addresses may change due to the ZooKeeper clients (including Kafka) inability to re-resolve IP addresses at this time. Be sure that all clients can reach all ZooKeeper servers directly via the connection string that includes all ZooKeeper host/IPs to avoid difficult to debug issues with connection loss. In the event that a ZooKeeper server IP address does change, all clients must be restarted. This includes scenarios where a load balancer is being used to simplify client configuration as the client will continue to try connecting to the old server IP address until restarted.

Other Considerations

ZooKeeper is the backbone of the Kafka infrastructure, so it is critical that it remains stable and available at all times. A full outage in ZooKeeper means that the Kafka cluster cannot operate optimally, changes such as creating new topics or order management functionality is impossible!

Basic Check

1. Find the state of each ZK instance
 2. Verify that there is **only one** leader
 3. Verify that a quorum exists
 4. Verify all followers are in-sync
-

The ZooKeeper (ZK) sanity checks can all be executed using the ZK four letter words. You will do the same in the hands-on lab. https://zookeeper.apache.org/doc/current/zookeeperAdmin.html#sc_zkCommands

The four checks that we execute are:

1. Find the state of each ZK instance

Run the following command against a ZK instance to identify its current state

```
echo mntr | nc localhost 2181 | grep zk_server_state
```

Possible result : leader or follower

2. Verify that only one leader in the ensemble exists

Run the previous command on all instances

3. Verify that a quorum exists

Use the **srvr** 4 letter command to get detailed status of each member of the ZK cluster

For more details see: <https://stackoverflow.com/questions/34537530/how-to-validate-zookeeper-quorum>

4. Verify that all followers are in sync with the leader

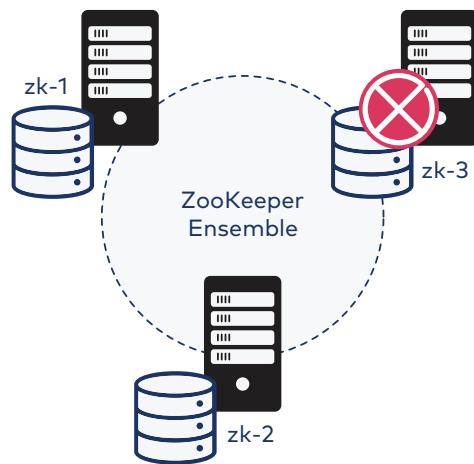
Run the following commands and confirm the responses are equal:

```
echo mntr | nc localhost 2181 | grep zk_synced_followers
```

```
echo mntr | nc localhost 2181 | grep zk_followers
```

Total Server Loss

1. Backup ZooKeeper data from the **leader**
2. Prepare a **new server** with the ZK binaries
3. Stop ZK process gracefully
4. On new server **start ZK process**
5. Optional: rolling restart of all ZK instances
6. Wait until all followers are **in-sync with leader**



This assumes a non-recoverable ZK instance, e.g. due to a fatal server error

In the event of a total loss of a server that a ZooKeeper server runs on, you can start up a new ZooKeeper process on a new server to recover. This means using the same configuration and myid file as the old server instance had. Similar to Kafka, the replication process will take over and all data required for the server will be sent to the new server. If your new server has a change in IP address, you would need to do a rolling restart of all ZooKeeper servers in the cluster in order to completely recover as connections to the new server instance would need to be resolved again. There should no service interruption.

1. Always back up ZooKeeper data from the leader it will get you back to the latest committed state in case of catastrophic failure.
2. Prepare a new server with the ZK binaries
3. If still running, stop the ZooKeeper process gracefully. By gracefully here we mean anything except `kill -9`
4. On new server start the ZooKeeper process
5. If server has new IP, rolling restart of all ZK servers is needed
6. Wait until all followers are in sync with the leader: `echo mntr | nc localhost 2181 | grep zk_synced_followers` should be equal to `echo mntr | nc localhost 2181 | grep zk_followers`

You will do this exercise in the coming lab.

Read ZooKeeper Data

Why:

- Debug issues with ZK connectivity
- Locate unexpected ZK client

How:

1. Read Transaction Log:

```
java -cp zookeeper-3.4.13.jar:lib/log4j-1.2.16.jar:lib/slf4j-log4j12-  
1.6.1.jar:lib/slf4j-api-1.6.1.jar org.apache.zookeeper.server.LogFormatter version-  
2/log.xxx
```

2. Read Snapshot Logs:

```
java -cp zookeeper-3.4.13.jar:lib/log4j-1.2.16.jar:lib/slf4j-log4j12-  
1.6.1.jar:lib/slf4j-api-1.6.1.jar org.apache.zookeeper.server.SnapshotFormatter  
version-2/snapshot.xxx
```

Purpose

Provide instructions on reading ZooKeeper snapshots and transaction logs. This is useful when debugging issues with ZooKeeper connectivity or when trying to locate a unexpected ZooKeeper client.

What are ZooKeeper Transaction Logs?

Each ZooKeeper server must write out a transaction to disk before that transaction is considered to be committed. Once the quorum has committed the transaction, the transaction as a whole is considered to be complete. The location on disk where each server writes out each transaction is called the transaction log.

How to Read ZooKeeper Transaction Logs

1. Locate and change into the transaction log directory. This will be your `dataDir` by default. If you have specified `dataLogDir` then this will be the transaction log directory.
2. Use the built in tool to read the binary data in the transaction log. The files will be in your transaction log directory under the subdirectory `version-2`. ---

Read ZooKeeper Data

What are ZooKeeper Snapshots?

Each ZooKeeper server creates a sparse snapshot of the data held on the server at the time the snapshot is taken. Snapshots can be used to recover ZooKeeper data at a certain point in time if combined with the full transaction log. More details on snapshots are available in the documentation.

How to Read ZooKeeper Snapshots

1. Locate and change into the data directory. This will be your dataDir as specified in your configuration.
2. Use the built-in tooling to read the binary snapshots:

hitesh@datacouch.io

Review



Question:

Why should all ZooKeeper instances be located geographically close to each other?

hitesh@datacouch.io

Further Reading

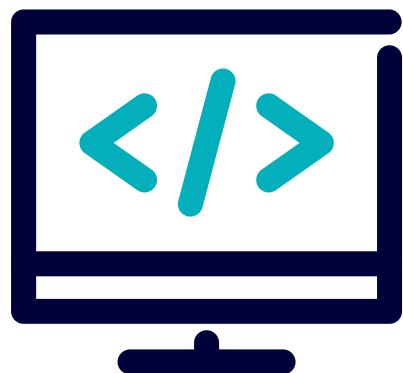
- ZK Quorums: https://zookeeper.apache.org/doc/current/zookeeperInternals.html#sc_quorum
- ZK Leader Activation: https://zookeeper.apache.org/doc/current/zookeeperInternals.html#sc_leaderElection

hitesh@datacouch.io

Lab: Troubleshooting ZooKeeper

Please work on **Lab 7a: Troubleshooting ZooKeeper**

Refer to the Exercise Guide



hitesh@datacouch.io

08: Troubleshooting & Tuning Brokers



CONFLUENT
Global Education

hitesh@datacouch.io

Module Overview



This module contains seven lessons:

1. Troubleshooting Brokers
2. Replication Concerns
3. Tuning Brokers: General Concepts & Best Practices
4. Optimization of a Message's Life Cycle on a Broker
5. Misc. Matters
6. Confluent Auto Data Balancer
7. Message Delivery Guarantees

hitesh@datacouch.io

a: Troubleshooting Brokers

Description

Application of general troubleshooting to brokers. Common broker troubleshooting scenarios. Metrics. General recommendations for brokers.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Diagnose broker performance for write and read requests
- Identify unexpected, unclean and/or frequent leader elections
- Diagnose and fix Java garbage collection related slow downs
- Recover from total server loss

hitesh@datacouch.io

Troubleshooting Checklist

- What exactly is happening?
- Make sure symptoms are well described
- Establish a time line
- Have you changed anything?
- What monitoring do you have in place?
- Can you SSH into broker host?
- Locate logs
- Parse logs for errors



Here we are assuming that the issue is clearly related to the Kafka brokers. In this case the steps to troubleshoot are as follows:

- **What's happening:** When asking about what is happening, be as precise as possible, e.g. broker 103 is down, or broker 104 is failing to catch up with the replica leaders
- **Precise description:** If answer to previous question was "broker 103 is crashing", then ask "what do you mean by crashing? Is the broker process down, or is the machine unreachable, etc."
- **Timeline:** Ask when did it happen or when did it start? Expect answer similar to this: "It all started 2 hours ago", or "it has been observed for the last 30 minutes...". Write down the start time as precisely as possible
- **Changes made:** Ask if something has been changed right prior to or during the observed issue. It's not about blaming, it's about finding the root cause as quickly as possible
- **Monitoring:** If nothing has changed, great. What monitoring do you have in place. E.g. broker is failing but its process is still in place. This might be a sign for overload
- **SSH into node:** If broker is not reachable, then next step is: can you log into the server/VM where the broker ran? If not, then the cluster node has a problem, maybe network partition or node is down/crashed
- **Locate Logs:** If we can SSH into the broker machine then we need to locate the logs. If they are not to be found at the usual/default location then find the file `log4j.properties` that the broker is using and work backwards from there.
- **Parse logs:** Having a precise idea about the symptoms of the issue should help to narrow down in which of the many log files to most likely find the error. It should also give some

indication about what type of error messages to expect in the log files.

hitesh@datacouch.io

Monitoring

- Monitoring is prerequisite to efficient troubleshooting
 - Deviation from trends indicate problems
 - Monitoring and alerts help to avoid/prevent problems
-

- Having a sound and complete monitoring in place with historical data (such as the throughput over time) is most important to successful troubleshooting
- Assuming you monitor the right metrics deviations from trends can be a warning that something is not quite right. E.g. a sharp decay of the throughput or a sharp rise in CPU usage can indicate trouble.
- Monitoring can be augmented by alerts that when triggered gives the time to react early on and prevent a failure: e.g. if the disk usage surpasses a given threshold we could add more disks prior to encountering an out of disk error.

Common Troubleshooting Scenarios

- Enabling TLS results in bad performance
- Common network problems: `java.io.brokenpipe`
- Unable to resolve hostname
- Timeout between ZK and broker
- VMs fight for shared resources
- No resource limits set for containers
- OOM on JVM
- AD integration and Kerberos
- Number of open files

This is a list of common problems in and around Kafka brokers that our support team encounters:

- **TLS:** Encryption performance (SSL; perf is bad, why?) oftentimes this is an OS level problem e.g. the entropy level too low, or the CPU load is too high. In the latter case add more cores
- A very common network problem is that `java.io.broken pipe` is reported in the broker logs. This means that there is some serious network problem, e.g. network is broken
- The error message "Unable to resolve hostname" indicates problems with DNS
- Timeout between ZK and broker can mean that on the ZK side there is not fast enough disk access or ZK ensemble is shared with other services than Kafka. Error message in the log: `fsync error (warning) took x amount of time which is...`
- Bigger companies often are on VMWare: it can happen that many VMs fight for shared resources - VM will pause until it can get resources → whole system fails (weird problems). Hard to troubleshoot. E.g. the brokers are not giving any errors. Need to look for time gaps in logs which can be an indication for the problem (say, nothing is reported in the logs for more than 30 seconds). Solution: pin resources to VMs similar to what one does when using cgroups in Linux. E.g. use at least 32GB of RAM for VM xyz.
- When running in Docker containers, e.g. on Kubernetes, customers often don't see resource limits such as min and max RAM or min and max IO. This leads to failures such as OOM where then Linux randomly kills a process (which can be a broker process or even the container daemon)

- OOM on JVM: Customers do not set enough heap for the JVM. Recommended settings for heap size for each component can be found in the docs
- Lot of security related problems (AD integration, Kerberos); root cause: mostly a lack of understanding on customer side
- Number of open files limitation (mostly broker and zookeeper related, can occur on Kafka Connect) - very common; need to specify on the service level and not just on OS level (customers sometimes run into this and then say that they have configured an insanely high number on the OS level...)

hitesh@datacouch.io

Broker Metrics for Troubleshooting (1)

- Kafka-specific metrics

```
kafka.network:type=RequestMetrics,name=TotalTimeMs,request={Produce|FetchConsumer|FetchFollower}
```

```
kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec
```

```
kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec
```

```
kafka.server:type=KafkaServer,name=TotalDiskReadBytes
```

```
kafka.server:type=KafkaServer,name=TotalDiskWriteBytes
```

```
kafka.server:type=SessionExpireListener,name=ZooKeeperExpiresPerSec
```

hitesh@datacouch.io

Broker Metrics for Troubleshooting (2)

- Host metrics
 - Disk usage
 - CPU usage
 - Page cache reads ratio
 - Network bytes sent/received
- GC metrics (JVM)

`java.lang:type=GarbageCollector,name=G1_Young|Old_Generation`

`CollectionCount`

`CollectionTime`

Broker - Important Files

- Location is set by the `log.dirs` broker config setting
 - If `log.dirs` is not specified, the `log.dir` config setting value is used
 - Default value for `log.dir` is `/tmp/kafka-logs`

File	Description
<code>recovery-point-offset-checkpoint</code>	Last offset flushed to disk
<code>cleaner-offset-checkpoint</code>	Offset up to which the cleaner has cleaned (compacted topics only)
<code>replication-offset-checkpoint</code>	Last committed offset
<code>log-start-offset-checkpoint</code>	Internal broker log where Kafka tracks the starting offset of the topic partition
<code>leader-epoch-checkpoint</code>	Per partition. Contains rows with epoch and offset . Each row is a checkpoint for the latest recorded leader epoch and the leader's latest offset upon becoming leader.

- For debugging purposes we have a few files worth analyzing.
- In Kafka, a **leader epoch** refers to the number of leaders previously assigned by the controller. Every time a leader fails, the controller selects the new leader, increments the current "leader epoch" by 1, and shares the leader epoch with all replicas. The replicas use the leader epoch as a means of verifying the current leader. If a leader fails and returns, when it tries to contact other replicas, it will send what it believes is the current leader epoch. The replicas will ignore the messages sent with outdated leader epochs.
- The `leader-epoch-checkpoint` file contains two columns: **epochs** and **offsets**. Each row is a checkpoint for the latest recorded leader epoch and the leader's latest offset upon becoming leader. Both replicas and leaders contain this file. Its role is for checking what range of offsets pertain to which epoch.
- The `recovery-point-offset-checkpoint` file is updated by the broker after a segment rolls and upon controlled shutdown. By default, this is the only time the broker knows for sure that records have been flushed to disk. This could be different depending on how the `log.flush.interval.messages` and `log.flush.interval.ms` properties are configured. For additional information, review the documentation.

<https://kafka.apache.org/documentation/#log.flush.interval.messages>

<https://kafka.apache.org/documentation/#log.flush.interval.ms>



Two of the files (`recovery-point-offset-checkpoint` and `replication-offset-checkpoint`) will live in the location `/var/log/kafka` (by default) and the rest of the files are maintained per partition and also live at `/var/log/kafka/`

hitesh@datacouch.io

Is Broker In Sync?

- Broker maintains session with ZooKeeper
 - If broker hosts a **Follower**, don't lag behind "too much"
-

As with most distributed systems automatically handling failures requires having a precise definition of what it means for a node to be "alive". For Kafka node liveness has two conditions:

1. A node must be able to maintain its session with ZooKeeper (via ZooKeeper's heartbeat mechanism)
2. If it is a follower it must replicate the writes happening on the leader and not fall "too far" behind

We refer to nodes satisfying these two conditions as being "in sync" to avoid the vagueness of "alive" or "failed". The leader keeps track of the set of "in sync" nodes. If a follower dies, gets stuck, or falls behind, the leader will remove it from the list of in sync replicas. The determination of stuck and lagging replicas is controlled by the `replica.lag.time.max.ms` configuration.

image: <http://wny-obgyn.com/wp-content/uploads/2015/02/HeartHealth.jpg> (via Bing,
license: free to share and use commercially)

Synchronize Broker Configs

The screenshot shows the Confluent Control Center interface. On the left, there's a sidebar with clusters C3 (Cluster 1) and CA (Cluster 2). The main area is titled 'Cluster settings' and has tabs for 'Kafka' and 'Brokers'. Under 'GENERAL', it shows 'broker.id' with three values: broker.101, broker.102, and broker.103. A red box highlights the '3 different values' note. Below that, under 'LISTENER', it shows 'listeners' with three values: broker.101, broker.102, and broker.103, each pointing to DOCKER://[IP]:9092. It also shows 'inter.broker.listener.name' and 'advertised.listeners' with the same three values. Under 'LOG', it shows 'log.dirs' as /var/lib/kafka/data and 'auto.create.topics.enable' and 'delete.topic.enable' both set to true. At the top right, there are 'Edit Settings' and 'Download' buttons, with 'Download' being the one highlighted by a red box.

Many troubles in a Kafka broker stem from the fact that not all brokers are using the same configuration settings. It is a recommended best practice to use the same configuration settings for all brokers in a cluster. The easiest way to find out discrepancies in the configurations is to use Confluent Control Center for this task.

1. In Confluent Control Center navigate to the **Cluster** and then in the list of available views, click **Cluster settings**
2. In the **Cluster settings** view, click the **Brokers** tab
3. Click the **Download** button to download all configuration settings of all brokers as a JSON formatted file

Use automation tools such as Chef or Puppet to automate provisioning of brokers and thus guarantee consistent configuration files!



The screenshot on this slide highlights a config setting (`broker.id`) on the brokers which actually must be different, and can't be synchronized (unless auto broker id generation is used).

Controller Issues

The Controller is the "brain" of the cluster

- Symptoms of Controller issues:
 - ISR updates are not happening
 - Replication stopped for no reason
- Causes of Controller issues:
 - Broker failure
 - Admin removed the `/controller` path in ZK

A Controller is the "brain" of the cluster

- One broker in a cluster runs the controller
- Monitor the liveness of brokers
- Elect new leaders on broker failure
- Communicate new leaders to brokers

Controller election is based on ZK. Who wins the creation of the `/controller` path in ZK becomes the controller! State of all partitions is cached on controller. For failover reasons this state is stored in ZK

Most common task of controller: act on broker failure

- Controlled shutdown of a broker
 1. Controller updates state of affected partitions locally and in ZK (new leader)
 2. Communicate state change to remaining brokers
- Recover from own failure
 1. `/controller` path in ZK is ephemeral, thus it goes away with failed controller (linked to ZK session)
 2. New controller election happens among remaining brokers
- New controller needs to load state from ZK

Add more Storage to Broker

- Using a partition reassignment tool:
 - Confluent Auto Data Balancer
 - `kafka-reassign-partitions`
 1. Using one of the tools, move all partitions off the broker
 2. Add new disks or SSDs
 3. Stop broker daemon gracefully
 4. Modify `log.dir` in `server.properties` with new data folder(s)
 5. Start broker daemon
 6. Using one of the tools, add the original partitions back to broker
- Using the Confluent Auto Data Balancer or `kafka-reassign-partitions` tool
 1. Run the tool to move all partitions off the broker that will have more disks added to it
 2. Add the new disks or SSDs to the broker and mount them
 3. Stop the broker daemon gracefully (`SIGTERM`, not `SIGKILL`)
 4. Modify `server.properties` by adding the new data folders to `log.dir` (or `log.dirs`)
 5. Start the broker daemon
 6. Run the Auto Data Balancer or partition reassignment tool again to add the original partitions back to this broker
- Example: Using `kafka-reassign-partitions`

For details please consult KIP-113 (<https://cwiki.apache.org/confluence/display/KAFKA/KIP-113%3A+Support+replicas+movement+between+log+directories>)

 1. Start with the file `sample.json`:

```
{"partitions":  
  [  
    {  
      "topic": "foo",  
      "partition": 1,  
      "replicas": [101],  
      "log_dirs": ["/var/lib/kafka/data2"]  
    }  
  ],  
  "version":1  
}
```

2. Update `log.dirs` to the value `/var/lib/kafka/data,/var/lib/kafka/data2`:

```
$ kafka-reassign-partitions --bootstrap-server kafka:9092 \  
  --bootstrap-server kafka:9092 \  
  --reassignment-json-file sample.json \  
  --execute
```

3. Restart the broker

This will move partition 1 to the new log directory `/var/lib/kafka/data2`

Gracefully Shut Down Kafka Cluster

Prerequisite configuration settings:

- All brokers `controlled.shutdown.enable=true`
- Use defaults for `controlled.shutdown.max.retries` and `controlled.shutdown.retry.backoff.ms`

Process:

1. Stop all clients
2. Shutdown one broker at a time
3. Wait for completion; observe log:
 - `...Starting controlled shutdown...`
 - `...Controlled shutdown succeeded...`
 - `...shut down completed...`

For maintenance of machines, it may be useful to completely shutdown a Kafka cluster in some circumstances. This article describes how to gracefully shutdown a set of Kafka brokers in order to guard against trouble upon restarting the cluster.

Procedure

Prerequisites

- `controlled.shutdown.enable` should be set to true on all brokers
- `controlled.shutdown.max.retries` and `controlled.shutdown.retry.backoff.ms` are parameters to be aware of for how robust the process is of a controlled shutdown. Defaults are generally appropriate for most environments

Steps to Perform

1. Stop all producer and consumer processes using the cluster
2. Shutdown each broker one at a time, waiting for controlled shutdown to complete. The controlled shutdown log messages to be aware of are:
 - `INFO [Kafka Server 0], Starting controlled shutdown (kafka.server.KafkaServer)`—indicating the controlled shutdown request is received
 - `INFO [Kafka Server 0], Controlled shutdown succeeded`

(`kafka.server.KafkaServer`)—indicating the controlled shutdown has been processed and the broker has been dropped from the cluster but the process is still running

- `INFO [Kafka Server 0], shut down completed`

(`kafka.server.KafkaServer`)—indicating the broker is now down completely

Conclusion

By shutting down each broker one at a time, each other live broker in the cluster is able to cleanly stop communication with the other brokers and you can avoid potential trouble with out of sync replicas and brokers hanging during shutdown while they await communication from brokers they perceive to be alive. Use this procedure when you have a known maintenance outage.

image: <http://i.stack.imgur.com/kmfPT.png> (via Bing, license: free to use and share commercially)

Java Heap Dump for Support

- Automatically upon **OutOfMemoryError**

```
-XX:+HeapDumpOnOutOfMemoryError -XXHeapDumpPath=<file_name>
```

- Manually using:

- **jmap** tool

```
jmap -dump:format=b,file=<file_name> <pid>
```

- **jconsole**
 - **jvisualvm**

Sometimes Confluent support might ask for a Java heap dump to help troubleshoot your problem. There are 4 ways to get a Java Heap dump:

- automatically upon **OutOfMemoryError**
- manually using JDK's **jmap** tool
- With the tool **jconsole** we can connect to any local or remote java process. By using MBeans we can get the heap dump.
- We can use **jvisualvm** GUI tool to connect to any local or remote JAVA processes. Through **jvisualvm** also we can generate the heap dump.

JVM GC Logging

1. Location for log dumps
2. Number of logs
3. Size of log files (best ~5MB)

```
export KAFKA_OPTS="-Xloggc:<location>`date +%F_%H-%M-%S`-gc.log -XX:+PrintGCDetails  
-XX:+PrintGCDateStamps -XX:+PrintTenuringDistribution -XX:+PrintGCCause  
-XX:+UseGLogFileRotation -XX:NumberOfGCLogFiles=<num> -XX:GCLogFileSize=<size>M"
```

4. Restart brokers

Depending on the disk configuration of a Kafka Broker, it can become necessary to configure the JVM GC logs to be written in a different fashion.

- Steps to configure JVM GC logging
 1. Determine the location where you would like your GC logs to be written to.
 2. Determine the number of GC logs you would like to maintain at any give period of time.
 3. Determine the size you would like each log file to be, we generally recommend around 5MB.
 4. We can then export the following parameters as follows (Please note, you will need to replace anything between <> to reflect our environment):

```
export KAFKA_OPTS="-Xloggc:<location>`date +%F_%H-%M-%S`-gc.log  
-XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintTenuringDistribution  
-XX:+PrintGCCause  
-XX:+UseGLogFileRotation -XX:NumberOfGCLogFiles=<num> -XX:GCLogFileSize=<size>M"
```

5. We will then need to restart your Broker for the changes to take affect.



Additional parameters that we can submit for GC logging on the JVM are documented here:

<http://www.oracle.com/technetwork/articles/java/vmoptions-jsp-140102.html>

Recommendations

- Add more brokers
 - Make sure all brokers use (similar) SSDs
 - All brokers have the same configuration
 - Brokers are located in same region
 - Fast network connections and dedicated NICs
 - Limit rogue clients with quotas
-

What actions do we recommend?

The following recommendations concern all brokers of a given replicaset

- add more brokers to better distribute the workload among the individual members of the cluster
- all brokers should have the fastest persistent local storage possible. In most cases these are SSDs. Also make sure all brokers use SSDs with similar performance characteristics
- all brokers have same configuration, specifically network and IO threads, buffer sizes and the like
- Due to latency reasons all brokers should be located in the same geographical region
- If there is a slow network connection or a NIC that is shared with other processes then a slowdown is unavoidable
- Rogue clients can flood Kafka with an unexpected amount of data and thus starve other processes such as replication. Use quotas to avoid this.

Review



Question:

- Assuming you do not have access to Confluent Control Center. How would you make sure all brokers of your cluster are using the same configuration values?
- How many in-sync replicas should you minimally have in production to guarantee fault tolerance?

To summarize this module please try to answer the 2 questions on the slide. You may organize in teams of 2 to 3 persons.

Possible answers:

- Use automation tools such as Chef or Puppet to automate provisioning of brokers and thus guarantee consistent configuration files!
- Usually Confluent recommends to define the replication factor to be 3 for each topic as a compromise between durability and storage cost
- To be able to guarantee durability the minimum number of ISRs should be 2 (the leader and 1 follower)

Further Reading

- Kafka Replication:
<https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Replication>
- Hands-free Kafka Replication: A lesson in operational simplicity
<https://www.confluent.io/blog/hands-free-kafka-replication-a-lesson-in-operational-simplicity/>

hitesh@datacouch.io

b: Replication Concerns

Description

Review of replication, metrics related to replication, and tips for troubleshooting replication-related issues.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Identify replication related issues

hitesh@datacouch.io

Recap

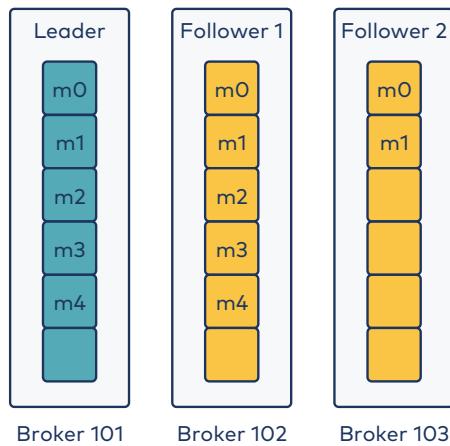
- Messages are **only** committed if all ISR^s have persisted it
- Consumers **only** see committed messages
- Producers can choose between:
 - latency: `acks=0 | 1`
 - durability: `acks=ALL`



Only committed messages are ever given out to the consumer. This means that the consumer need not worry about potentially seeing a message that could be lost if the leader fails. Producers, on the other hand, have the option of either waiting for the message to be committed or not, depending on their preference for tradeoff between **latency** and **durability**.

What is an out-of-sync replica?

- Follower can fall behind or crash
- Follower is "sufficiently" behind leader



We use replica and follower interchangeably. In this context they are the same.

Followers may fall behind or crash so we must ensure we choose an up-to-date follower. A replica/follower is considered to be out-of-sync or lagging when it falls "sufficiently" behind the leader of the partition. Where sufficiently behind is defined in the property `replica.lag.time.max.ms` (30 s. default).

What to monitor for?

- ISR shrinks

`IsrShrinksPerSec`

`IsrExpandsPerSec`

`kafka-topics --describe ...`

- Under-replicated partitions

`UnderReplicatedPartitions`

`OfflinePartitionsCount`

- Unclean leader elections

`LeaderElectionRateAndTimeMs`

`UncleanLeaderElectionsPerSec`

We should monitor each broker for the metrics listed on the slide. If the values are greater than zero then these are signs of potential problems in the cluster. For example, frequent ISR shrinks for a single partition can indicate that the data rate for that partition exceeds the leader's ability to service the consumer and replica threads.

What happens if too few ISRs?

- Assumptions:
 - Producer uses `acks=all`
 - `min.insync.replicas` set (typically =2)
 - Producer receives **error** when too few ISRs
 - `NOT_ENOUGH_REPLICAS`
 - `NOT_ENOUGH_REPLICAS_AFTER_APPEND`
-

A message is committed only after it has been successfully copied to all the in-sync replicas.

Specify a minimum ISR size - the partition will only accept writes if the size of the ISR is above a certain minimum, in order to prevent the loss of messages that were written to just a single replica, which subsequently becomes unavailable. This setting only takes effect if the producer uses `acks=all` and guarantees that the message will be acknowledged by at least this many in-sync replicas. This setting offers a trade-off between **consistency** and **availability**. A higher setting for minimum ISR size guarantees better consistency since the message is guaranteed to be written to more replicas which reduces the probability that it will be lost. However, it reduces availability since the partition will be unavailable for writes if the number of in-sync replicas drops below the minimum threshold.

Possible Error messages to producer:

- `NOT_ENOUGH_REPLICAS`, Messages are rejected since there are fewer in-sync replicas than required. Retriable
- `NOT_ENOUGH_REPLICAS_AFTER_APPEND`, Messages are written to the log, but to fewer in-sync replicas than required. Retriable

Why is my replica out of sync

- Slow replica
- Stuck replica
- Bootstrapping replica

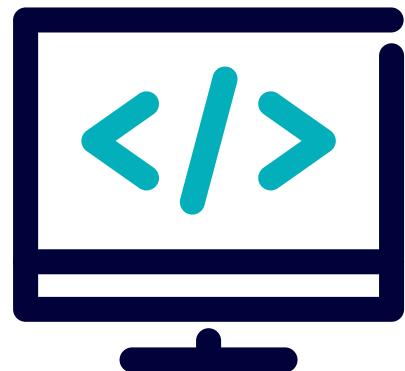


-
- **Slow replica:** A follower replica that is consistently not able to catch up with the writes on the leader for a certain period of time. One of the most common reasons for this is an I/O bottleneck on the follower replica causing it to append the copied messages at a rate slower than it can consumer from the leader.
 - **Stuck replica:** A follower replica that has stopped fetching from the leader for a certain period of time. A replica could be stuck either due to a GC pause or because it has failed or died.
 - **Bootstrapping replica:** When the user increases the replication factor of the topic, the new follower replicas are out-of-sync until they are fully caught up to the leader's log.

Lab: Troubleshooting Brokers

Please work on **Lab 8a: Troubleshooting Brokers**

Refer to the Exercise Guide

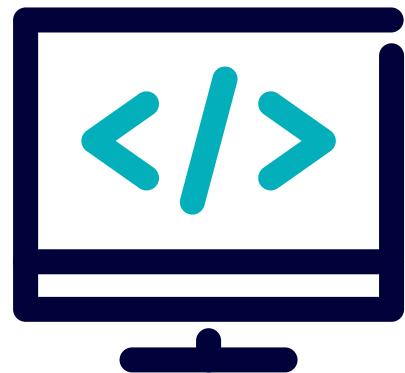


hitesh@datacouch.io

Lab: Not Enough ISRs

Please work on **Lab 8b: Not Enough ISRs**

Refer to the Exercise Guide



hitesh@datacouch.io

c: Tuning Brokers: General Concepts & Best Practices

Description

Broker monitoring, general guidelines, performance checks.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Define a short list of broker best-practices

hitesh@datacouch.io

Broker Best-Practices

- Tuning
 - Stick (mostly) with the defaults
 - Set default cluster retention as appropriate
 - Default partition count should be at least the number of brokers
 - Monitoring
 - Watch the right things
 - Don't try to alert on everything
 - Triage and Resolution
 - Solve problems, don't mask them
-

hitesh@datacouch.io

Broker Monitoring

- Bytes in/out; Messages in
 - Partitions
 - Count and leader count
 - Under replicated and offline
 - Threads
 - Network pool, request pool
 - Max. dirty percent
 - Requests
 - Rates and times - total, queue, local, and send
-

- Bytes in/out; Messages in (why not messages out?)
- Partitions
 - Count and leader count: Usually they should be the same, but if e.g. a broker goes down then there can be a discrepancy for a short time.
 - Under replicated and offline
- Threads
 - Network pool, request pool
 - Max. dirty percent
- Requests

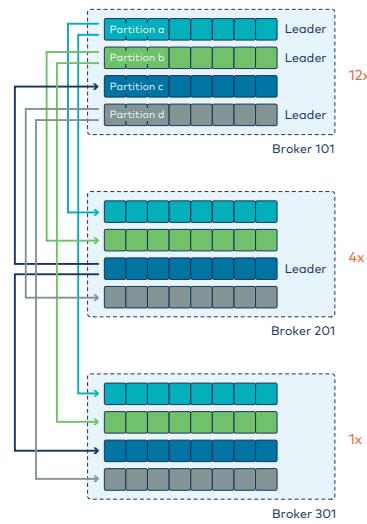
Topic Monitoring

- Bytes in/out
 - Messages in, Produce rate, Produce failure rate
 - Fetch rate, fetch failure rate
 - Partition bytes
 - Quota throttling
-

hitesh@datacouch.io

General Guidelines

- Evenly distribute partition leadership
- Provision sufficient memory
- Avoid version mismatches
- Mind your logging (Log4j)
- Compacted topics require extra resources
- Set User Limits



- **Distribute partition leadership among brokers in the cluster.** Leadership requires a lot of network I/O resources. For example, when running with replication factor 3, a leader must receive the partition data, transmit two copies to replicas, plus transmit to however many consumers want to consume that data. So, in this example, being a leader is at least **four times** as expensive as being a follower in terms of network I/O used. Leaders may also have to read from disk; followers only write. In the graphic on the slide we can see that leadership is not spread equally among the brokers. Thus the top broker has about 12 times the amount of work to do than the bottom worker.
- **For sustained, high-throughput brokers, provision sufficient memory to avoid reading from the disk subsystem.** Partition data should be served directly from the operating system's file system cache whenever possible. However, this means you'll have to ensure your consumers can keep up; a lagging consumer will force the broker to read from disk. Also: do not forget to configure enough Java heap as recommended in the documentation.
- **Using older clients with newer topic message formats, and vice versa, places extra load on the brokers** as they convert the formats on behalf of the client. Avoid this whenever possible.
- **Modify the Apache Log4j properties as needed;** Kafka broker logging can use an excessive amount of disk space. However, don't forego logging completely — broker logs can be the best, and sometimes only — way to reconstruct the sequence of events after an incident
- **Compacted topics require memory and CPU resources on your brokers.** Log compaction needs both heap (memory) and CPU cycles on the brokers to complete

successfully, and failed log compaction puts brokers at risk from a partition that grows unbounded. You can tune `log.cleaner.dedupe.buffer.size` and `log.cleaner.threads` on your brokers, but keep in mind that these values affect heap usage on the brokers. If a broker throws an `OutOfMemoryError` exception, it will shut down and potentially lose data. The buffer size and thread count will depend on both the number of topic partitions to be cleaned and the data rate and key size of the messages in those partitions. Monitoring the log-cleaner log file for `ERROR` entries is the surest way to detect issues with log cleaner threads.

- **Setting User Limits for Kafka:** Kafka opens many files at the same time. The default setting of 1024 for the maximum number of open files on most Unix-like systems is insufficient. Any significant load can result in failures and cause error messages such as `java.io.IOException... (Too many open files)` to be logged in the Kafka log files. You might also notice errors such as this:

```
ERROR Error in acceptor (kafka.network.Acceptor)
java.io.IOException: Too many open files
```

We recommend a relatively high starting point, such as 100,000.

Broker Performance Checks

- Are all brokers in cluster working?
 - Are network interfaces saturated?
 - Is CPU utilization high?
 - Do you have really big messages?
-
- Are all brokers in cluster working?
 - Are network interfaces saturated?
 - Reelect partition leaders (remember a leader does at least 4x the work of a follower; make sure partition leadership is distributed homogeneously across all brokers)
 - Rebalance partitions in cluster (again, make sure partitions are equally distributed across all brokers)
 - Spread out traffic (more brokers and/or partitions)
 - Is CPU utilization high (specifically `iowait`)?
 - Are other processes competing for resources?
 - Do you have a bad disk?
 - Do you have really big messages? Kafka is not optimized or architected for big messages. Do not exceed the limit of 1MB. Even that size is already a **big** message. If you absolutely need big messages then there are other architectural patterns available, e.g. store message payload in some blob storage and pass URI to that message payload in the actual record/message sent to Kafka.

d: Optimization of a Message's Life Cycle on a Broker

Description

Review of the life cycle of requests on a broker. Metrics overall and at various stages. Tips for mitigating high latencies.

hitesh@datacouch.io

Learning Objectives

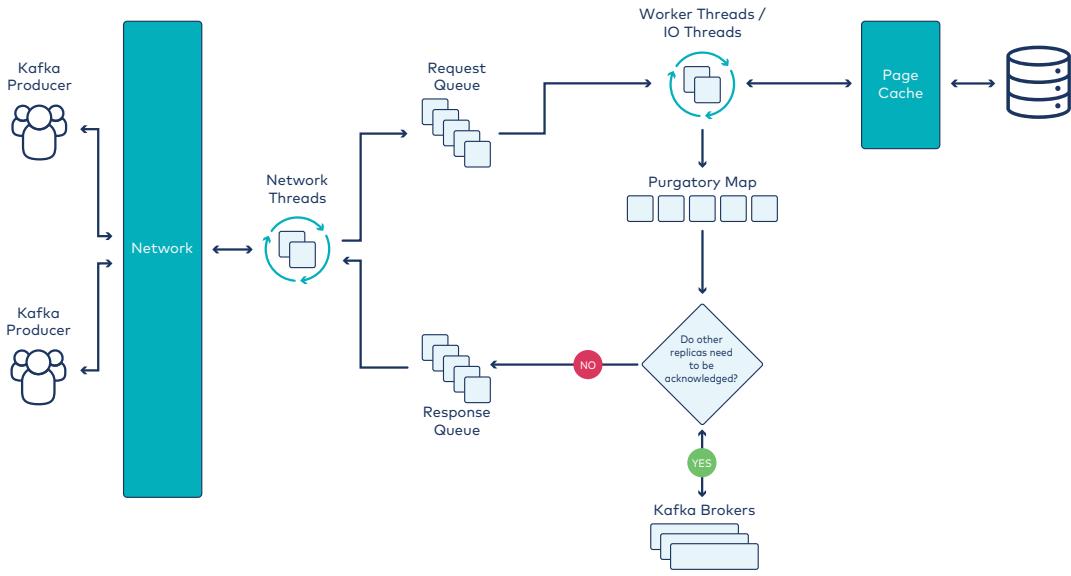


Completing this lesson and associated exercises will enable you to:

- Sketch the broker **producer request** in detail

hitesh@datacouch.io

Producer Request



On this slide we see how a producer request, coming from a producer via the network to the broker, is handled:

1. Request is picked up from the network by a free thread from the network thread pool
2. Network thread puts request into request queue
3. A free thread from the IO thread pool is picking up next available request from request queue
4. IO thread writes record to local page cache of broker - from where it eventually is persisted on local disk of broker
5. IO thread puts request into **request purgatory**
6. If `acks=all` then broker waits for write confirmation (ACKs) of in-sync replicas
7. Completed producer request is put into response queue
8. Any free thread from the network thread pool takes next available (handled producer) request from response queue
9. Network thread sends response back to network

Apache Kafka has a data structure called the "request purgatory". The purgatory holds any request that hasn't yet met its criteria to succeed but also hasn't yet resulted in an error.



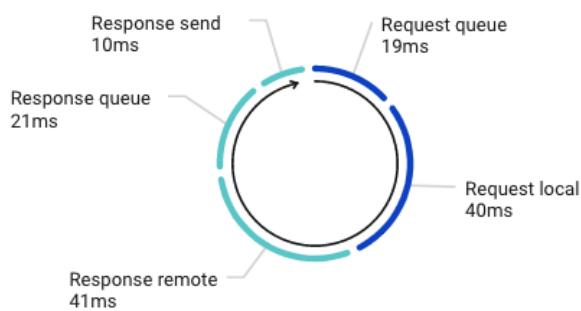
A produce request with `acks=all` cannot be considered complete until all in-sync replicas have acknowledged the write and we can guarantee it will not be lost if the leader fails. These requests are considered complete when either (a) the criteria they requested is complete or (b) some timeout occurs.

hitesh@datacouch.io

Producer Request in Control Center

JMX Metrics: `kafka.network:type=RequestMetrics,request=Produce,name=<name>`

Broker 101				Production request latency
99.9th %ile	99th %ile	95th %ile	Median	
46ms	30ms	14ms	5ms	



✖ where `<name>` is:

- `RequestQueueTimeMs`
- `LocalTimeMs`
- `RemoteTimeMs`
- `ResponseQueueTimeMs`
- `ResponseSendTimeMs`
- `TotalTimeMs`

The Confluent Control Center is the ideal tool to analyze the internals of producer requests. One can display the total response time of a producer request and see its split into the five partial times:

- Request queue time
- Request local time
- Response remote time
- Response queue time
- Response send time

In this graph it is evident that the `response remote time` takes the lion's share of the overall **producer request latency**. Please identify the one time that is above the expected value (or percentage) in your particular case.

In the next slide we'll see how those times are defined. We will use again the previous graphic for illustration.

Producer Request - Latency Explained

Break down **TotalTimeMs** further to see the entire request lifecycle:

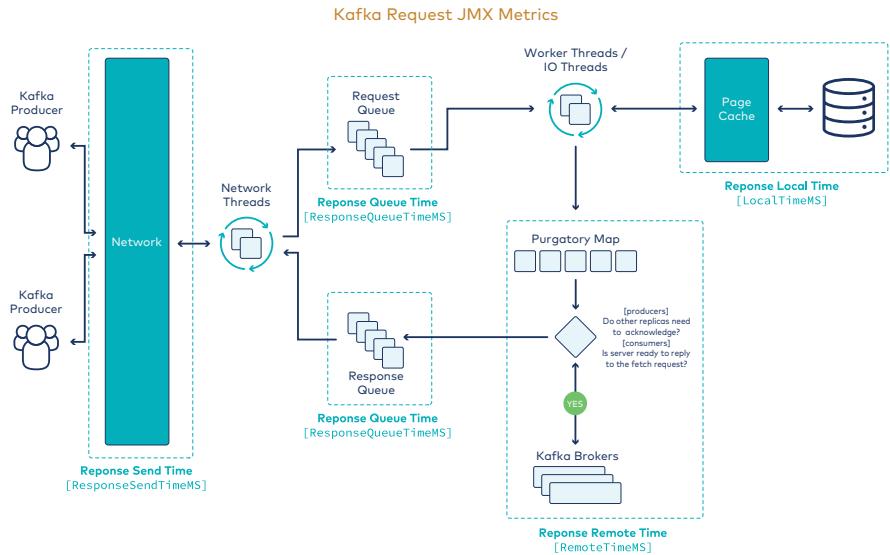
Metric	Description
RequestQueueTimeMs	Time the request waits in the request queue
ResponseSendTimeMs	Time to send the response
ResponseQueueTimeMs	Time the request waits in the response queue
LocalTimeMs	Time the request is processed at the leader
RemoteTimeMs	Time the request waits for the follower

Here we see what the meaning of the previous 5 times shown in the Control Center **Producer request latency** chart mean:

Request queue	the time an individual producer request sits idle in the request queue
Request local	the time used by the broker to persist the record included in the producer request
Response remote	the time the response sits in the purgatory and waits for the other brokers (that is the ISRs) to acknowledge that the record has been persisted in their respective local commit log. This is mostly relevant if acks>1
Response queue	This is the time the response sits in the response queue until it is picked up
Response send	The time it takes to send the response back to the producer

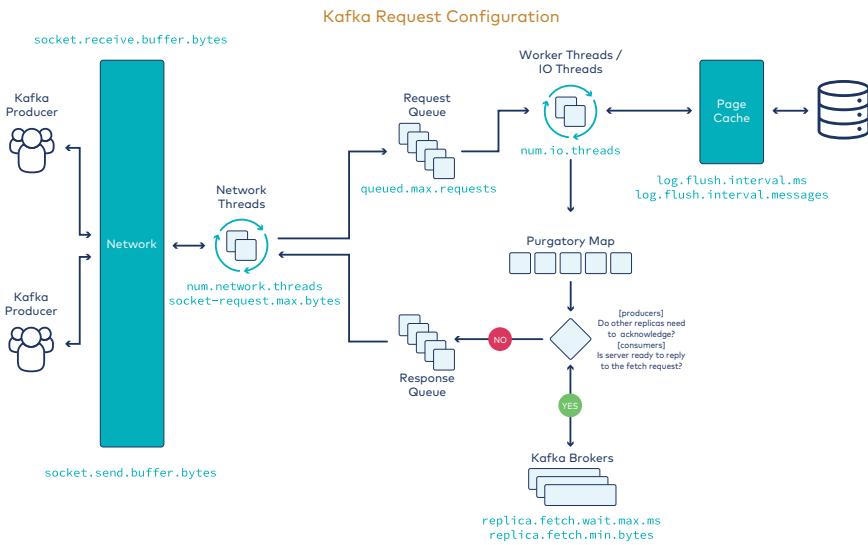
With that knowledge we can now start to think about how to optimize the broker for our particular scenario.

Monitoring Requests on the Broker



Because the **ProduceRequest** is handled by multiple components within the broker, a slow down at any of them will increase the overall latency for the request.

Configuring Requests on the Broker



Once you determine where the bottleneck is, consider making changes to the broker to fix the issue. This slide lists some of the properties that can be changed for each of the components in the request flow.



Kafka level log flushing is disabled by default because modern Linux operating systems are more optimized for page cache flushing. If IO is a bottleneck, see again the previous slide about request queue metrics.

Parameter	Default	Description
<code>socket.receive.buffer.bytes</code>	102,400	The SO_RCVBUF buffer of the socket sever sockets. If the value is -1, the OS default will be used
<code>num.network.threads</code>	3	The number of threads that the server uses for receiving requests from the network and sending responses to the network
<code>socket.request.max.bytes</code>	104,857,600	The maximum number of bytes in a socket request
<code>queued.max.requests</code>	500	The number of queued requests allowed before blocking the network threads
<code>num.io.threads</code>	8	The number of threads that the server uses for processing requests, which may include disk I/O
<code>log.flush.interval.messages</code>	$2^{63} - 1$ (max long)	The number of messages accumulated on a log partition before messages are flushed to disk
<code>log.flush.interval.ms</code>	null	The maximum time in ms that a message in any topic is kept in memory before flushed to disk. If not set, the value in <code>log.flush.scheduler.interval.ms</code> is used

Parameter	Default	Description
<code>replica.fetch.wait.ms</code>	500	max wait time for each fetcher request issued by follower replicas. This value should always be less than the <code>replica.lag.time.max.ms</code> at all times to prevent frequent shrinking of ISR for low throughput topics
<code>replica.fetch.min.bytes</code>	1	Minimum bytes expected for each fetch response. If not enough bytes, wait up to <code>replicaMaxWaitTimeMs</code>
<code>num.replica.fetchers</code>	1	Number of fetcher threads used to replicate messages from a source broker. Increasing this value can increase the degree of I/O parallelism in the follower broker
<code>socket.send.buffer.bytes</code>	102,400	The SO_SNDBUF buffer of the socket sever sockets. If the value is -1, the OS default will be used.
<code>message.max.bytes</code>	1,000,000	This sets the maximum size of the message that the server can receive. This should be set to prevent any producer from inadvertently sending extra large messages and swamping the consumers
<code>background.threads</code>	10	This sets the number of threads that will be running and doing various background jobs. These include deleting old log files. Its default value is 10 and you might not need to change it

hitesh@datacouch.io

Mitigate High Latencies

Request queue	<ul style="list-style-type: none">increase the pool of I/O threads (<code>num.io.threads</code>)
Request local	<ul style="list-style-type: none">use dedicated, faster local disks (SSD)tune <code>log.flush.interval.messages</code> & <code>log.flush.interval.ms</code>
Response remote	<ul style="list-style-type: none">make sure brokers are co-locateduse high bandwidth network (1 GbE, 10 GbE)use dedicated NICs for brokersassure all brokers nodes are equaldo you really need <code>acks=all</code>?
Response queue	<ul style="list-style-type: none">use high bandwidth network (1 GbE, 10 GbE)use dedicated NICs for brokers

Network

A fast and reliable network is an essential performance component in a distributed system. Low latency ensures that nodes can communicate easily, while high bandwidth helps shard movement and recovery. Modern data-center networking (1 GbE, 10 GbE) is sufficient for the vast majority of clusters.

You should avoid clusters that span multiple data centers, even if the data centers are colocated in close proximity; and avoid clusters that span large geographic distances.

Kafka clusters assume that all nodes are equal. Larger latencies can exacerbate problems in distributed systems and make debugging and resolution more difficult.

From the experience of Confluent, the hassle and cost of managing cross-data center clusters is simply not worth the benefits.

e: Miscellaneous Matters

Description

Broker monitoring, memory usage, optimization.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Name relevant broker metrics to monitor

hitesh@datacouch.io

Monitoring

Monitor the following resources:

- Network throughput
- Disk I/O
- Disk space
- CPU usage

-
- **Monitor your brokers for network throughput** — both transmit (TX) and receive (RX) — as well as disk I/O, disk space, and CPU usage. Capacity planning is a key part of maintaining cluster performance

hitesh@datacouch.io

Optimization per Service Agreement

Optimize Latency

- `num.replica.fetchers=<value>`

Optimize Durability

- `default.replication.factor=3`
- `auto.create.topics.enable=false`
- `min.insync.replicas=2`
- `unclean.leader.election.enable=false`
- `broker.rack=<rack ID>`
- `log.flush.interval.messages`
- `log.flush.interval.ms`

Optimize Throughput

- Nothing to do

Optimize Availability

- `unclean.leader.election.enable=true`
- `min.insync.replicas=1`
- `num.recovery.threads.per.data.dir`

Optimize Latency

- `num.replica.fetchers`: increase if followers can't keep up with the leader (default 1)

Optimize Throughput

- Nothing to do

Optimize Durability

- `default.replication.factor=3` (default 1)
- `auto.create.topics.enable=false` (default true)
- `min.insync.replicas=2` (default 1); topic override available
- `unclean.leader.election.enable=false` (default true); topic override available
- `broker.rack`: rack of the broker (default null)
- `log.flush.interval.messages`, `log.flush.interval.ms`: for topics with very low throughput, set message interval or time interval low as needed (default allows the OS to control flushing); topic override available

Optimize Availability

- `unclean.leader.election.enable=true` (default true); topic override available
- `min.insync.replicas=1` (default 1); topic override available
- `num.recovery.threads.per.data.dir`: number of directories in log.dirs (default 1)

hitesh@datacouch.io

Memory Usage

- `replica.fetch.max.bytes`: 1MB allocated per replicated partition
- Brokers allocate `replica.fetch.max.bytes` (default 1MB) for each replicated partition

hitesh@datacouch.io

f: Confluent Auto Data Balancer

Description

Overview of Auto Data Balancer.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

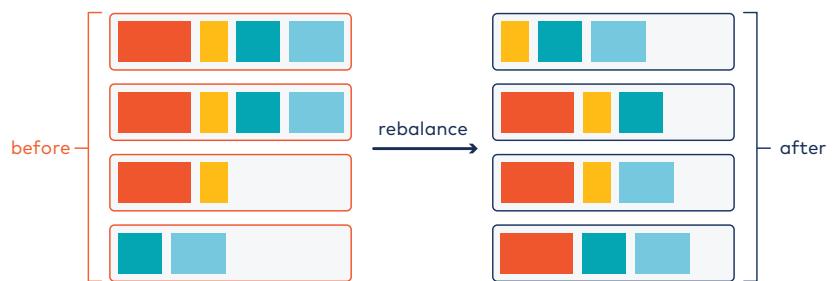
- Explain how Auto Data Balancer allows partitions to be moved between brokers easily.

hitesh@datacouch.io

Confluent Auto Data Balancer (ADB)

Dynamically move partitions to optimize resource utilization and reliability

- Shift data to create an even workload across your cluster
- Easily add and remove nodes from your Kafka cluster
- Rack aware algorithm rebalances partitions across a cluster
- Traffic from balancer is throttled when data transfer occurs



Confluent Auto Data Balancer (ADB)

```
$ confluent-rebalancer execute \
--bootstrap-server kafka:9092 \
--metrics-bootstrap-server kafka:9092 \
--throttle 10000000 \
--verbose
```

Here is a sample command using the ADB.

hitesh@datacouch.io

Confluent Auto Data Balancer (ADB)

```
Computing the rebalance plan (this may take a while) ...
You are about to move 17 replica(s) for 14 partitions to 4 broker(s) with total size 827.2 MB.
The preferred leader for 14 partition(s) will be changed.
In total, the assignment for 15 partitions will be changed.
The minimum free volume space is set to 20.0%.

The following brokers will have less than 40% of free volume space during the rebalance:
Broker    Current Size (MB)  Size During Rebalance (MB)  Free % During Rebalance  Size After Rebalance (MB)  Free % After Rebalance
 0        413.6              620.4                  30.1                519.6                  30.5
 2        620.4              723.8                  30.1                520.8                  30.5
 3        0                  517                   30.1                520.8                  30.5
 1        1,034              1,034                 30.1                519.6                  30.5

Min/max stats for brokers (before -> after):
Type   Leader Count          Replica Count          Size (MB)
Min   12 (id: 3) -> 17 (id: 0)    37 (id: 3) -> 43 (id: 3)    0 (id: 3) -> 517 (id: 1)
Max   21 (id: 0) -> 17 (id: 0)    51 (id: 1) -> 45 (id: 0)    1,034 (id: 1) -> 517 (id: 3)
No racks are defined.

Broker stats (before -> after):
Broker   Leader Count      Replica Count      Size (MB)      Free Space (%)
 0        21 -> 17       48 -> 45       413.6 -> 517     30.5 -> 30.5
 1        20 -> 17       51 -> 44       1,034 -> 517     30.5 -> 30.5
 2        15 -> 17       40 -> 44       620.4 -> 517     30.5 -> 30.5
 3        12 -> 17       37 -> 43       0 -> 517       30.5 -> 30.5

Would you like to continue? (y/n):
Rebalance started, its status can be checked via the status command.

Warning: You must run the status or finish command periodically, until the rebalance completes, to ensure the throttle is removed. You can also alter the throttle by re-running the execute command passing a new value.
```

Here is a sample output.

g: Message Delivery Guarantees

Description

Message delivery guarantees. High-level view of idempotence and transactions.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Explain what an **idempotent** producer is
- Quantify the impact of EOS on throughput and latency in your apps

hitesh@datacouch.io

Message Delivery Semantics

At most once - messages may be lost but are never redelivered.

At least once - messages are never lost but may be redelivered.

Exactly once - each message is delivered **once and only once**.

-
- **At most once:** messages are either delivered exactly once, or not at all → Some message loss, but there will never be any duplicates.
 - **At least once:** messages are delivered one or more times, but no message is ever lost; some duplicates are possible.
 - **Exactly once:** In this case, even if a producer tries to resend a message, it leads to the message being written exactly once to the topic partition.

Some performance measurements can be found here:

- KIP-98 perf testing: https://docs.google.com/spreadsheets/d/1dHY6M7qCiX-NFvsgvaEOYoVdNq26uA8608XIh_DUpI4/edit#gid=61107630

Idempotent Producer

Idempotent Producer == Exactly once per Partition

- No data loss
- No duplicates
- In-order semantics

```
enable.idempotence=true
```

hitesh@datacouch.io

Atomic Transactions

Simplified sample code...

```
1 producer.initTransactions();
2
3 try
4 {
5     producer.beginTransaction();
6     producer.send(...);
7     producer.send(...);
8     producer.send(...);
9     producer.commitTransaction();
10 }
11 catch(ProducerFencedException pfe)
12 {
13     producer.close();
14 }
15 catch(KafkaException ke)
16 {
17     producer.abortTransaction();
18 }
```

hitesh@datacouch.io

Consumers and EOS

- Use `isolation.level=read_committed`
 - Commit offsets with computed results in transaction
-
- To use transactions, you need to configure the Consumer to use the right `isolation.level` and use the new Producer APIs. There are two new isolation levels in Kafka consumer:
 - `read_committed`: Read both kinds of messages (those that are not part of a transaction and that are) after the transaction is committed.
 - `read_uncommitted`: Read all messages in offset order without waiting for transactions to be committed. This option is similar to the current semantics of a Kafka consumer.
 - If the consumer produces some results and stores them say in an external DB, then to be truly EOS end-to-end the consumer must also make sure that the offsets of the source topic are stored transactionally with the computed values. If using a Kafka Sink Connector then there exist connectors that support EOS out of the box.

Review



Question:

- You are going to add new brokers to your Kafka cluster. What are actions you're going to execute to balance the load among all brokers evenly?
- Do you think EOS should be the default for Kafka? If yes, why? Why not?

To round up this module please answer the two questions on the slide.

hitesh@datacouch.io

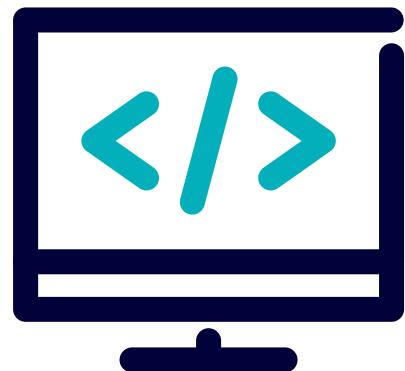
Further Reading

- Apache Kafka, Purgatory, and Hierarchical Timing Wheels:
<https://www.confluent.io/blog/apache-kafka-purgatory-hierarchical-timing-wheels/>
- Kafka Protocol Guide: <http://kafka.apache.org/protocol.html>
- Confluent Auto Data Balancer:
<https://docs.confluent.io/current/kafka/rebalancer/rebalancer.html>
- Transactions in Apache Kafka: <https://www.confluent.io/blog/transactions-apache-kafka/>
- Exactly-once Semantics are Possible: Here's How Kafka Does it:
<https://cnfl.io/kafka-eos>
- Introducing Exactly Once Semantics in Apache Kafka: <https://cnfl.io/intro-eos>
- Exactly-once Support in Apache Kafka:
<https://medium.com/@jaykreps/exactly-once-support-in-apache-kafka-55e1fdd0a35f>

Lab: Tuning Brokers

Please work on **Lab 8c: Tuning Brokers**

Refer to the Exercise Guide



hitesh@datacouch.io

09: Troubleshooting & Tuning Schema Registry



CONFLUENT
Global Education

hitesh@datacouche.com

Module Overview



This module contains one lesson:

1. Troubleshooting Schema Registry

hitesh@datacouch.io

a: Troubleshooting the Schema Registry

Description

Review of schema registry and assorted issues that commonly arise with schema registry deployment.

hitesh@datacouch.io

Learning Objectives

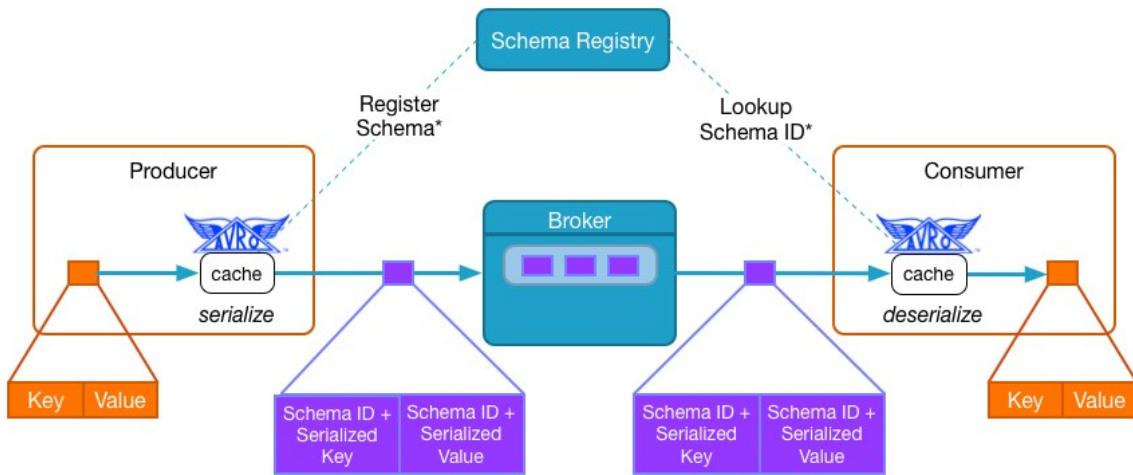


Completing this lesson and associated exercises will enable you to:

- Optimally configure, deploy and operate Schema Registry
- Identify issues caused by breaking schema changes
- Explain the differences between transitive and non-transitive schema compatibility

hitesh@datacouch.io

Confluent Schema Registry



Avro, JSON, and Protobuf Supported Formats and Extensibility

Avro was the original choice for the default supported schema format in Confluent Platform, with Kafka serializers and deserializers provided for the Avro format.

Confluent Platform 5.5 adds support for Protocol Buffers and JSON Schema along with Avro, the original default format for Confluent Platform. Support for these new serialization formats is not limited to Schema Registry, but provided throughout Confluent Platform. Additionally, as of Confluent Platform 5.5, Schema Registry is extensible to support adding custom schema formats as schema plugins.

New Kafka serializers and deserializers are available for Protobuf and JSON Schema, along with Avro. The serializers can automatically register schemas when serializing a Protobuf message or a JSON-serializable object. The Protobuf serializer can recursively register all imported schemas, .

The serializers and deserializers are available in multiple languages, including Java, .NET and Python.

Schema Registry supports multiple formats at the same time. For example, you can have Avro schemas in one subject and Protobuf schemas in another. Furthermore, both Protobuf and JSON Schema have their own compatibility rules, so you can have your Protobuf schemas evolve in a backward or forward compatible manner, just as with Avro.

Schema Registry in Confluent Platform 5.5 also adds the support for schema references in Protobuf by modeling the import statement.



Although this slide is not necessarily directly related to troubleshooting, it still is important information to consider when asking the question:

- "Why is my SR slow?", or
- "Why at all bothering about SR?"

hitesh@datacouch.io

Common Issues - Bad Design

- Co-locating SR with Kafka Broker
 - Multiple SRs in Company
 - Wrong setup of SR in multi-DC
 - No use of VIP
-

- **Co-locating SR with Kafka Broker:** Confluent Schema Registry application requires very few resources—about 1 GB for heap—but other than that, it does not need a lot of CPU, memory or disk. Given its relatively low footprint, operators may be tempted to co-locate Schema Registry with other services, like a Kafka broker. However, co-locating a Schema Registry instance on a host with any other application means that its uptime is entirely dependent on the co-located services behaving properly on the host. It is best practice to deploy Schema Registry on its own.
- **Multiple SRs in Company:** You should have unique schemas and schema IDs across an entire company, across geographical areas or clusters in a multi-datacenter design. Over time, organizations restructure, project scopes change and an end system that was used by one application may now be used by multiple applications. If schema IDs are not globally unique, there may be collisions between schema IDs. For consistency in schema definitions and operational simplicity, stick with one global Schema Registry.
- **Wrong setup of SR in multi-DC:** In a multi-datacenter design, the same schema and schema IDs must be available in both datacenters. Whether the design is active-active or active-passive, designate one Kafka cluster as the primary for Schema Registry.
- **No use of VIP:** Not deploying a virtual IP (VIP) in front of Schema Registry instances, which results in an extra burden on application developers to update connection information if IP addresses change

Common Issues - Inconsistent Configurations

- Different names for schemas topic
 - Mixing election modes
 - Different settings between instances of SR
 - Using same `host.name`
-

Let's now look at some issues that arise from inconsistent configuration of the Schema Registry.

- **Different names for schemas topic:** There is a commit log with all the schema information, which gets written to a Kafka topic. All Schema Registry instances should be configured to use the same schemas topic, whose name is set by the configuration parameter `kafkastore.topic`. This topic is the schemas source of truth, and the primary instances read from this topic. The name defaults to `_schemas`, but sometimes customers choose to rename it. This has to be the same for all Schema Registry instances; otherwise it may result in different schemas with the same ID.
- **Mixing election modes:** The Confluent Schema Registry can operate in one of two election modes. Mixing the election modes among the Schema Registry instances in the same cluster, such that some are configured to use the Kafka group protocol and others are configured to use ZooKeeper leads to issues
- **Different settings between instances of SR:** In addition to the election mode needing to be the same between the Schema Registry instances, there are a few other configuration parameters that must match in order to avoid unintended side effects. You can typically leave those other parameters as default (e.g., the group ID for the consumer used to read the Kafka store topic). If you override any default settings, they must be consistently changed in all instances.
- **Use of the same `host.name`:** There is one configuration parameter that should differ between Schema Registry instances and that is `host.name`. This should be unique per instance to prevent potential problems and variance from which instance is primary.

Common Issues - Operational Mistakes

- SR is not secured
- Misconfigured credentials
- Manually creating schemas topic
- Deleting the schemas topic
- Not backing up the schemas topic
- Restarting SR before restoring schemas topic
- Not monitoring SR
- Wrong Java version for SR
- Poorly managed Kafka cluster

Finally let's look into some issues caused by operational mistakes.

- **SR is not secured:** Securing Schema Registry is just as critical as securing your Kafka cluster, because the schema is how different applications and organizations talk to each other through Kafka. Therefore, not limiting access to the schemas in Schema Registry might allow an unauthorized user to mess with the schemas in such a way that client applications can no longer deserialize their data.
- **Misconfigured credentials:** Configuring Schema Registry for SSL encryption and SSL or SASL authentication to the Kafka brokers is important for securing their communication. This requires working with the security team in your company to get the right keys and certificates, and configuring the proper keystores and truststores to ensure that Schema Registry can securely communicate with the brokers. We have observed many customers spending time troubleshooting wrong keys or certificates, which slows down their ability to spin up new services.
- **Manually create schemas topic:** The primary Schema Registry instance registers all new schemas and backs it up to a schemas topic in Kafka. This Kafka topic is the source of truth for all schema information and schema-to-schema ID mapping. By default, Schema Registry automatically creates this topic if it does not already exist, and it creates it with the right configuration settings: replication factor of three and retention policy set to **compact** (versus **delete**).
- **Deleting the schemas topic:** Once the schemas topic is created, it is important to ensure that it is always available and never to delete it. If someone were to delete this topic, producers would not be able to produce data with new schemas, because Schema

Registry would be unable to register new schemas. We hope it doesn't happen to you, but we have to mention it because this has happened before.

- **Not backing up the schemas topic:** Should the schema topic be accidentally deleted, operators must be prepared to restore it. Therefore it is a best practice to backup the schemas topic on a regular basis. If you already have a multi-datacenter Kafka deployment, you can backup this topic to another Kafka cluster using Confluent Replicator. You can also use a Kafka sink connector to copy the topic data from Kafka to a separate storage (e.g., AWS S3). These will continuously update as the schema topic updates.
- **Restarting SR before restoring schemas topic:** Restoring the schemas topic requires a series of steps to be followed. Kafka operators often do not have control over pausing the client applications, which may try to register new schemas at random intervals, particularly if the configuration parameter `auto.register.schemas` is left at its default of `true`. Therefore, the schemas topic should be fully restored before Schema Registry instances are restarted, so that when Schema Registry does restart and read the topic, it reads them in order and schemas IDs maintain their proper sequence.
- **Not monitoring SR:** Like any other component, you have to monitor the Schema Registry instances to know that they are able to service clients. The last thing you want is for a Kafka client to be the first to alert the operations team that the Schema Registry service is unreachable. In fact, the operations team should be the first to know through good monitoring practices!
- **Wrong Java version for SR:** Schema Registry is a Java application. It is compatible with Java 8 and Java 11. Trying to run Schema Registry with an incompatible Java version will not succeed.
- **Poorly managed Kafka cluster:** The source of truth for schemas is stored in a Kafka topic, so the primary Schema Registry instance needs access to that Kafka topic to register new schemas. Schema Registry communicates with the Kafka cluster to write to the schemas topic. That cluster needs to be highly available to ensure that new schemas can be properly registered and written to that topic.

Using Topic Schemas

The importance of shared Schema Registry:

- Tackling organizational challenges of data management
- Resilient data pipelines
- Safe schema evolution
- Storage and computation efficiency
- Data discovery
- Cost-efficient ecosystem
- Data policy enforcement

By applying automatic schema governance to your topics, you are able to avoid problems with schema drift. This may not immediately seem like an issue but many software systems end up suffering from this eventually as teams grow and people come and go. As time goes on, it becomes more and more difficult to reason about the type of data that exists in your topics if you have no schema guarantees. This is analogous to a schema-less K/V store versus a relational database that has a well structured schema.

You can also mix strategies where some topics have well managed schemas, and some topics are free form.

See Gwen's post on the Schema Registry, and the importance of having one, on the Confluent blog: <https://www.confluent.io/blog/schema-registry-kafka-stream-processing-yes-virginia-you-really-need-one/>

Schema Compatibility Rules

- BACKWARD
 - BACKWARD_TRANSITIVE
 - FORWARD
 - FORWARD_TRANSITIVE
 - FULL
 - FULL_TRANSITIVE
 - NONE
-

The Schema Registry server can enforce certain compatibility rules when new schemas are registered in a subject. These are the compatibility types:

- **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

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

Schema Evolution

- Cannot change field data type
 - Exception - data type widening is allowed
 - You cannot change a field's data type. If you have decided that a field should be some data type other than what it was originally created using, then add a whole new field to your schema that uses the appropriate data type.
 - schema evolution allows widening types, e.g. `int → string` or `int → long`
-

hitesh@datacouch.io

Example

Schema V1 fields		Schema V2 fields
<pre>"fields": [{ "name": "firstname", "type": "string" }, { "name": "lastname", "type": "string" }, { "name": "age", "type": "int", "default": "-1" }]</pre>	<p>Backward</p> <p>producer → consumer</p>  <p>Forward</p> <p>consumer ← producer</p> 	<pre>"fields": [{ "name": "lastname", "type": "string" }, { "name": "age", "type": "int", "default": "-1" }, { "name": "hobby", "type": "string", "default": "" }]</pre>

- Default value for `hobby` allows consumer using V2 to process messages produced with V1, i.e. schema V2 is **BACKWARD** compatible with V1
- No default value for `firstname` means consumer using V1 cannot process messages produced with V2, i.e. schema V2 is **not FORWARD** compatible with V1

Test Compatibility of Schema

REST Endpoint:

```
POST /compatibility/subjects/<subject>/versions/<version>
```

Maven Plugin for Schema Registry:

Goal:

`schema-registry:test-compatibility`

```
<plugin>
  <groupId>io.confluent</groupId>
  <artifactId>kafka-schema-registry-maven-plugin</artifactId>
  <version>5.1.2</version>
  <configuration>
    <schemaRegistryUrls>
      <param>http://schema-registry:8081</param>
    </schemaRegistryUrls>
    <subjects>
      <product-key>src/main/avro/product-key.avsc</product-key>
      <product-value>src/main/avro/product-value.avsc</product-value>
    </subjects>
  </configuration>
  <goals>
    <goal>test-compatibility</goal>
  </goals>
</plugin>
```

- If `<version>=latest`, then checks compatibility with the last registered schema
- There is a Schema Registry **Maven plugin**, which contains goals for things like registering and downloading schemas and **testing compatibility**. For more info see: <https://docs.confluent.io/current/schema-registry/docs/develop/maven-plugin.html#schema-registry-test-compatibility>. You will be using this plugin in the upcoming hands-on lab.

Subject Name Strategies

- **Subject:** scope where schemas can evolve in Schema Registry

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

- TopicNameStrategy example

Topic: `driver-positions`

Subjects: `driver-positions-key`
`driver-positions-value`

3 possible settings are available for each of the above config property, where `<topic>` is the topic name and `<type>` is the fully qualified Avro record type name:

- TopicNameStrategy (default): `<subject-name> = <topic>-key | <topic>-value`
- TopicRecordNameStrategy: `<subject-name> = <topic>-<type>-key | <topic>-<type>-value`
This is used in a topic with many event types to allow each type to evolve separately
- RecordNameStrategy: `<subject-name> = <type>-key | <type>-value`
This allows evolution of an event type that is used across many topics



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

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 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.

Review



Question:

Which schema compatibility rule makes most sense for your particular use case(s)? Justify your answer.

hitesh@datacouch.io

Further Reading

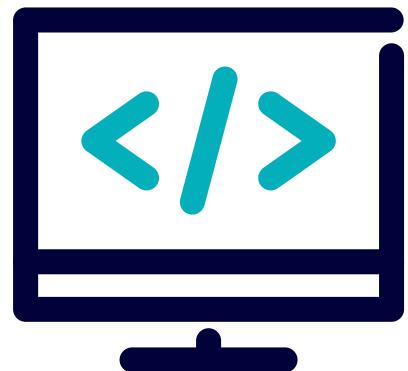
- Schema Registry: <https://docs.confluent.io/current/schema-registry/docs/index.html>
- Decoupling Systems with Apache Kafka, Schema Registry and Avro:
<https://www.confluent.io/blog/decoupling-systems-with-apache-kafka-schema-registry-and-avro/>
- Yes, Virginia, You Really Do Need a Schema Registry:
<https://www.confluent.io/blog/schema-registry-kafka-stream-processing-yes-virginia-you-really-need-one/>
- Should You Put Several Event Types in the Same Kafka Topic?
<https://www.confluent.io/blog/put-several-event-types-kafka-topic/>
- Schema Registry Maven Plugin:
<https://docs.confluent.io/current/schema-registry/docs/develop/maven-plugin.html>

hitesh@datacouch.io

Lab: Troubleshooting the Schema Registry

Please work on **Lab 9a: Troubleshooting the Schema Registry**

Refer to the Exercise Guide



hitesh@datacouch.io

Branch 4: Troubleshooting & Tuning

Clients - Overview



**CONFLUENT
Global Education**

Agenda



This is a branch of our CAO content on Troubleshooting & Tuning Clients. It is broken down into the following modules:

10. Troubleshooting & Tuning Producers
11. Troubleshooting & Tuning Consumers
12. Troubleshooting & Tuning Streams Apps
13. Troubleshooting & Tuning Kafka Connect

hitesh@datacouch.io

10: Troubleshooting & Tuning Producers



CONFLUENT
Global Education

hitesh@datacouch.io

Module Overview



This module contains two lessons:

1. Troubleshooting Producers
2. Tuning Producers

hitesh@datacouch.io

a: Troubleshooting Producers

Description

Producer metrics, common troubleshooting scenarios, librdkafka producer specifics.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- discover producer performance issue
- monitor a producer for data loss
- troubleshoot a **librdkafka** based producer

hitesh@datacouch.io

Producer Metrics for Troubleshooting

- Observe: `kafka.producer:type=producer-metrics,client-id=<client_id>`
 - Response Rate
 - Request Rate
 - Request latency avg
 - Outgoing byte rate
 - IO ratio & IO wait ratio
 - Record retry & error rate

hitesh@datacouch.io

Common Producer Issues

Issues:

- Cannot connect to Kafka
- Cannot write to topic
- Producer is very slow

Troubleshooting

Use "kafkacat" to troubleshoot producer issues:

```
$ kcat -L -b kafka:9092  
$ cat iot_data.csv | kcat -P -p -1 -b kafka:9092 -t iot-data
```

-
- **Cannot connect to Kafka:** reasons could be:
 - you might have defined only a single bootstrap server and this server is not reachable
 - misconfigured `listeners` and `advertised.listeners` on broker;
 - are you using the correct endpoint on the producer (including port)?
 - the Kafka cluster is secured and you try to access it with wrong credentials or no credentials at all
 - **Cannot write to topic:** reasons could be;
 - The topic does not exist and auto topic creation is turned off
 - The topic is protected by ACLs and the producer does not have the necessary authorizations
 - **Producer is very slow:** reasons include:
 - you are recreating the producer each time you send a record (reuse same producer instance!)
 - suboptimal producer configuration, specifically `batch.size`, `linger.ms`, `compression.type` and `acks`
 - are there quotas set on the Kafka cluster? They might limit your throughput
 - are you experiencing a lot of retries and errors when sending records? Observe `record-retry-rate` and `record-error-rate`

- To troubleshoot common producer related issues the tool **kafkacat** is very helpful.
 - The first command shown on the slide retrieves the metadata (about all topics) from Kafka
 - the second command writes the content of a CSV file to the topic **iot-data**

hitesh@datacouch.io

Troubleshoot librdkafka Clients

Context	Description	Verbosity
generic	anything generic enough not to fit the other contexts	sparse
broker	broker handling (protocol requests, queues)	medium
topic	topic and partition state changes and events	medium
queue	internal request and message queue events	low
msg	message transmission and parsing	high
protocol	Kafka protocol requests and responses	medium/high
cgrp	consumer group state machine	medium
security	SSL and SASL handshakes - on connect only	low
fetch	Consumer's fetcher state machine and fetch decisions	high
feature	Broker feature discovery - on connect only	medium
interceptor	interceptor handling and callbacks	low

librdkafka is a C/C++ client library for Kafka and is the basis for the non-Java Kafka clients that are packaged with Confluent Platform. The goal of this slide is to describe the various debug contexts that can be enabled to help facilitate troubleshooting a problematic client.

- The list on the slide describes each context and its expected verbosity.
- Each desired context can be specified in a comma separated list using the `debug` client property.

Troubleshoot librdkafka Clients

Context	Description	Verbosity
plugin	dynamic plugin loading	sparse
metadata	Topic and broker metadata updates	medium
all	enable all of the above contexts	very high

Common usages of the debug contexts:

- Troubleshooting common producer issues

Set `debug=broker,topic,msg`

- Troubleshooting security related issues

Set `debug=broker,security`

On the slide are a couple of reference examples for common usages of the debug contexts.

- **Common issues:** This will provide information on handling the broker connection, topic metadata, and message transmission
- **Security issues:** This will provide information on handling the broker connection and security related (SSL and SASL handshakes) issues

Review



Question:

How do you make sure that all messages produced are actually received by Kafka?

In a mission critical application it is most often essential that every single message generated by a source system is guaranteed to be persisted in Kafka. The question is, what means does an operator have to make an informed claim about the truth of this statement in their production system?

hitesh@datacouch.io

Further Reading

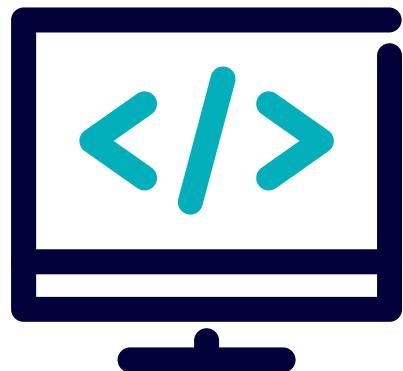
- The (Kafka) Producer: <https://kafka.apache.org/documentation/#theproducer>
- librdkafka Wiki: <https://github.com/edenhill/librdkafka/wiki>

hitesh@datacouch.io

Lab: Troubleshooting Producers

Please work on **Lab 10a: Troubleshooting Producers**

Refer to the Exercise Guide



Lab Troubleshooting: If you run into errors when asked to run `/config/install-libmnl.sh`, running this command beforehand as a quick fix should help:

```
mv /etc/yum.repos.d/confluent.repo /tmp/
```

This may apply to other places `libmnl` is used.

b: Tuning Producers

Description

Tuning producer serialization, partitioning, and replication. Producer configs. Producer metrics. Compression. Producer tuning summary.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Select the data format appropriate for your specific applications
- Decide how many partitions are reasonable for a given topic
- Configure optimal buffering related values for your producers
- Name a few pros and cons of end-to-end compression

hitesh@datacouch.io

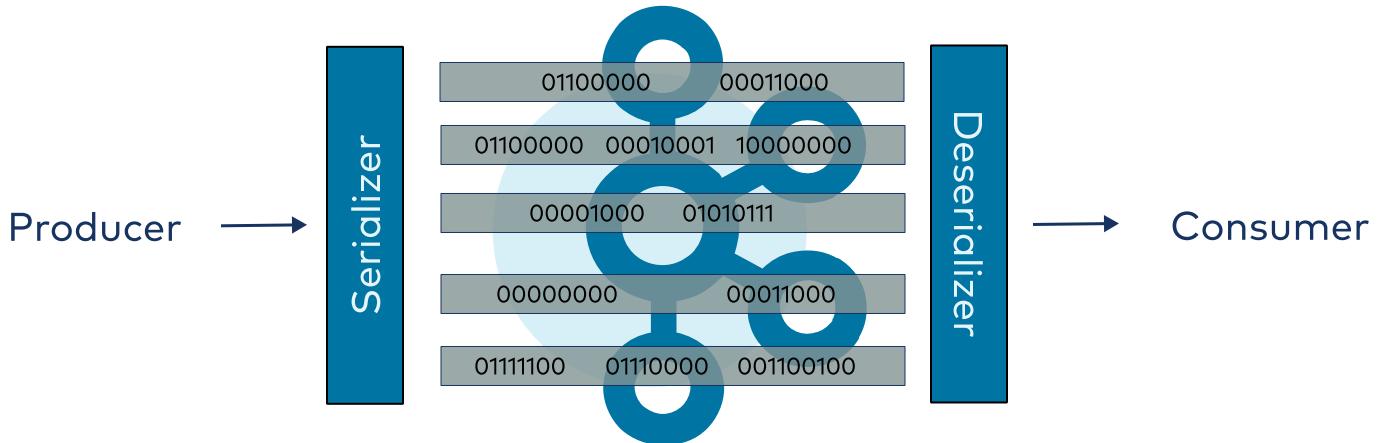
Out of the box data formats

- String
 - Short, Integer, Long
 - Float, Double
 - UUID
 - JSON
 - AVRO
 - Protobuf
 - Binary (`byte[]`, `ByteBuffer`)
-

- **String:** Only for simple string values
- **Short/Integer/Long:** Only for simple integer values
- **Float/Double:** Only for simple floating point values
- **UUID:** Values of type Universal Unique Identifier
- **JSON:** If plain JSON is used as the serialization format, things like field names are redundantly stored per message and it is also a plain text format and so further inefficient. JSON is supported by the Confluent Schema Registry.
- **Avro:** Apache Avro is a data serialization standard for compact binary format widely used for storing persistent data on HDFS as well as for communication protocols such as used with Kafka.
- **Protobuf:** Protocol Buffers (a.k.a., protobuf) are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data. Protobuf is supported by the Confluent Schema Registry.
- **Binary:** Useful for when data should not be serialized/deserialized, e.g. when replicating data.

Confluent recommends using a more efficient serialization mechanism such as Avro. Avro is a binary format which is a much more efficient use of bytes. When also used in combination with the Schema Registry which efficiently externalizes store of the schema of the data formats, Avro saves a lot of space over plain JSON. The schema registry also has many other benefits such as enforcing compatibility levels of schemas during schema evolution and providing a convenient centralized repository for agents to access schemas without embedding dependencies.

Serialization & Deserialization



It is important to remember that all events that flow through Kafka are just streams of bytes from the broker's perspective. In many situations where Kafka is used serialization and deserialization play an important part in the overall time it takes for an event to flow through the data pipeline. Thus let's look at it in more details.

- Serialization is the process of converting an object into a stream of bytes
- Deserialization is the opposite process of the serialization
- Serialization is a recursive algorithm. Starting from a single object, all the objects that can be reached from that object by following instance variables, are also serialized
- Both serializing and deserializing require the serialization mechanism to discover information about the instance it is (de)serializing
- The process of serialization/deserialization is usually CPU bound
- Keep your record payload as simple as possible to increase throughput

Custom SerDes for any data format

Can define custom SerDes...

1. Write **serializer**
 2. Write **deserializer**
 3. Write **serde**
-

If you have a data format that is not supported out of the box by Kafka, you can implement what's called custom **SerDes**. **SerDes** stands for a pair of Serializer and Deserializer. To do so you need to implement the interfaces **Serializer** and **Deserializer** as well as implement the interface **Serde**.

The Confluent examples repository demonstrates how to implement templated serdes:
[https://github.com/confluentinc/kafka-streams-examples/tree/5.4.1-post/src/main/java/
io/confluent/examplesstreams/utils](https://github.com/confluentinc/kafka-streams-examples/tree/5.4.1-post/src/main/java/io/confluent/examplesstreams/utils)

Process:

1. Write a serializer for your data type by implementing `org.apache.kafka.common.serialization.Serializer`
2. Write a deserializer for your data type by implementing `org.apache.kafka.common.serialization.Deserializer`
3. Write a serde for your data type by implementing `org.apache.kafka.common.serialization.Serde`, which you either do manually or by leveraging helper functions in Serdes such as `Serdes.serdeFrom(Serializer<T>, Deserializer<T>)`.

Why AVRO?

- Many tools to support it
- Direct mapping from/to JSON
- Much more compact than JSON
- Very fast
- Many language bindings
- Rich, extensible schema language
- Best in supporting schema evolution



Works with **Confluent Schema Registry**

Apache Avro is a data serialization standard for compact binary format widely used for storing persistent data on HDFS as well as for communication protocols such as used with Kafka. One of the advantages of using Avro is lightweight and fast data serialization and deserialization, which can deliver very good ingestion performance.

Why is AVRO a good choice?

- Avro is popular not just in the Kafka eco system but also on Hadoop, etc. Thus many tools exist that support this data format
- It has a direct mapping to and from JSON
- It has a very compact format. The bulk of JSON, repeating every field name with every single record, is what makes JSON inefficient for high-volume usage.
- It is very fast.
- It has great bindings for a wide variety of programming languages so you can generate Java objects that make working with event data easier, but it does not require code generation so tools can be written generically for any data stream.
- It has a rich, extensible schema language defined in pure JSON
- It has the best notion of compatibility for evolving your data over time.

For more details see: <https://www.confluent.io/blog/avro-kafka-data/>

Number of Partitions

- Number of Partitions == directly proportional to scalability
 - More consumer instances than partitions → waste
 - Unless we use extra consumers as standby instances for fast failover
-

The number of partitions in a topic is directly proportional to the scalability of the downstream consumers. If you have the necessary computing resources and need maximum scalability then select a high number of partitions for the respective topic.

If you have a consumer group with more consumers than there are available partitions to consume, then the extra consumers will be idle. In many cases this is a waste

There are scenarios where extra consumers make sense though, this is the case when the consumers are stateful such as consumers of a Kafka Streams application. Then extra consumers can be designated as standby and will replicate the local state such that if the source consumer fails, they can take over quickly where the failed consumer left off.

Partitioning Strategy

- Default: Compute Hash of Record **Key**
 - Select best partition strategy for given key
 - Strategy should **evenly distribute** data among partitions
-

By default Kafka producers use a partitioning strategy based on the hash value computed of the record key. That is a good strategy in many cases. But there might be scenarios where this leads to unbalanced partitions. The goal is to select the best partition strategy for a given key, that evenly distributes the data among partitions.

hitesh@datacouch.io

Replication Quotas

- Use replication quotas to protect network bandwidth
- To be considered when using Confluent Rebalancer
- Replication quota is per broker:
 - Incoming quota: `leader.replication.throttled.rate`
 - Outgoing quota: `follower.replication.throttled.rate`



Usually **incoming = outgoing**

Replication quotas are very useful to protect the network bandwidth, when existing data needs to be moved across brokers. This is typically required and should be considered for the Confluent Rebalancer, Kafka reassign-partition tool, replacing a broker with a new machine, etc.

The replication quota is per broker. You can control the incoming quota, `leader.replication.throttled.rate` separately from the outgoing quota, `follower.replication.throttled.rate`. However, in the common case, one will just set both to the same value based on the constraints for the ongoing network bandwidth, which typically determines a lower bound of the replication quota.

Compacting Topics

- Kafka guarantees presence of last value for every key
 - Total number of keys not known in advance
 - Define hard retention policy to limit disk space usage
-

In the case of a compacted topic, Kafka guarantees the presence of a value for every unique key. A priori you don't know how many unique keys there will be. However, you can configure a compacted topic to have a hard retention policy, if you can accept that some unique keys may be lost. If you do not set a hard retention policy, then the disk space used may be impacted by the users of the cluster.

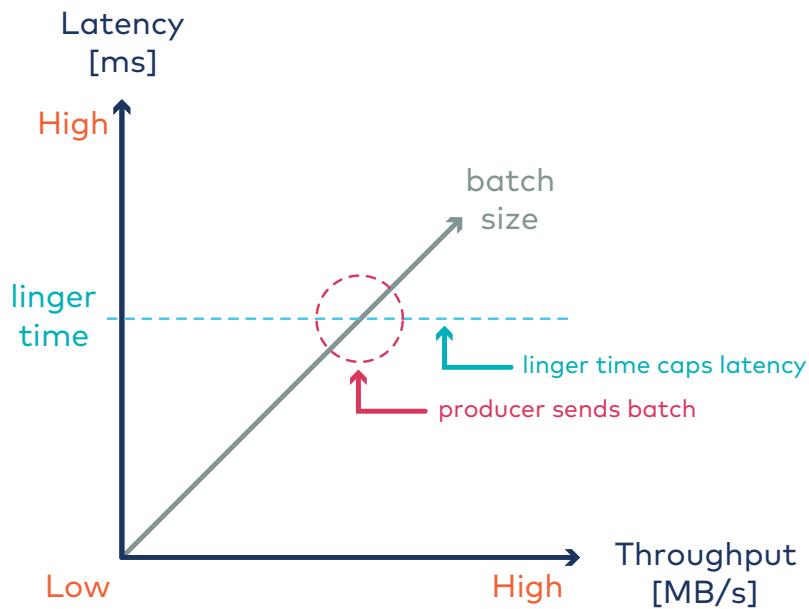
hitesh@datacouch.io

Critical Configurations

- `batch.size`
 - `linger.ms`
 - `buffer.memory`
 - `compression.type`
 - `max.in.flight.requests.per.connection`
 - `acks`
-

- `batch.size`: affects how much batching can happen
- `linger.ms`: this is important for slow producer not filling up the batch quickly enough
- `buffer.memory`: this is the total bytes of memory the producer can use to buffer records waiting to be sent to the server. It affects how well batching works. If there is not enough memory then the producer will either block (`max.block.ms`) or throw an exception.
- `compression.type`: the efficiency depends on the batch size and the compression type
- `max.in.flight.requests.per.connection`: this affects ordering
- `acks`: this affects durability

Batch Size



Sometimes one wants to increase the throughput by trading in on **latency per message**. Message batches offer higher throughput due to the fact that they require less RPC calls from the producer and batches of messages usually provider better compression ratios.

- **Batch Size:** Instead of the number of messages, `batch.size` measures batch size in total bytes. That means it controls how many bytes of data to collect, before sending messages to the Kafka broker. So, without exceeding available memory, set this as high as possible. Make sure the default value is 16384 (16kB). However, it might never get full, if we increase the size of our buffer. On the basis of other triggers, such as linger time in milliseconds, the Producer sends the information eventually.
- **Linger Time:** In order to buffer data in asynchronous mode, `linger.ms` sets the maximum time. Let's understand it with an example, a setting of `linger.ms=100` batches 100ms of messages to send at once. Here, the buffering adds message delivery latency but this improves throughput.
- **Buffer Memory:** Default 32MB; overall buffer allocated for batching by producer

Producer Metrics

Metric	Description
io-ratio	Fraction of time I/O thread spent doing I/O
io-wait-ratio	Fraction of time I/O thread spent waiting
1 - io-ratio - io-wait-ratio	User processing time
batch-size-avg	Average batch size
compression-rate-avg	Average compression rate

When tuning the producers one needs to establish a feedback loop. the best one is to monitor the behavior of key metrics of the producer. There are two main categories of metrics:

- metrics for the producer aggregated over all topics handled by it
- metrics per topic handled by the producer

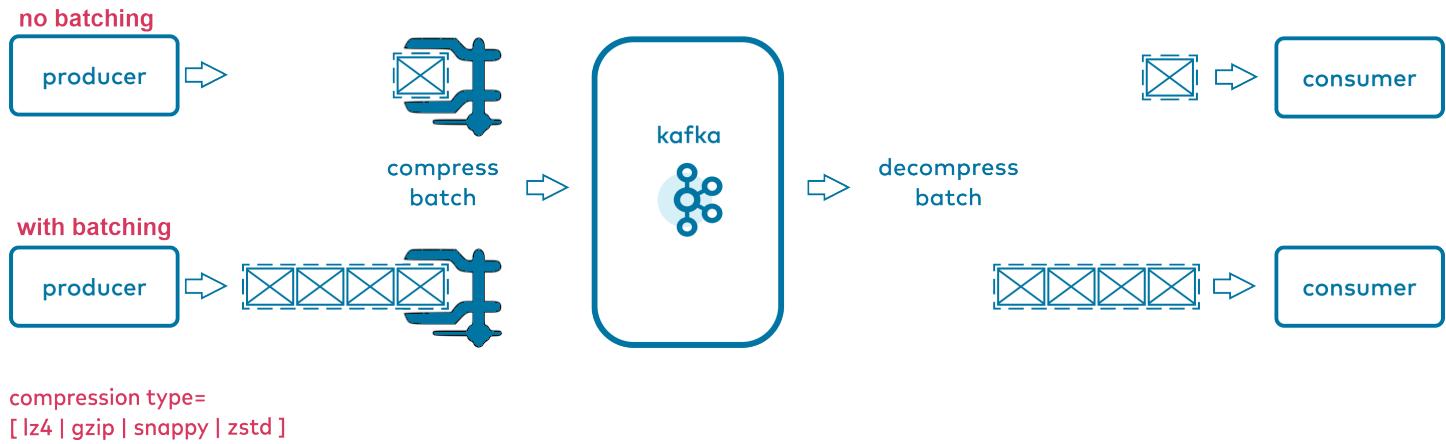
User processing time: This is the time spent in user code, specifically executing user callbacks for acknowledgements

Producer Metrics - Per Topic

Metric	Description
record-send-rate	The average number of records sent per second for a topic.
byte-rate	The average number of bytes sent per second for a topic.
record-error-rate	The average per-second number of record sends that resulted in errors for a topic.
record-retry-rate	The average per-second number of retried record sends for a topic.
compression-rate	The average compression rate of record batches for a topic.

hitesh@datacouch.io

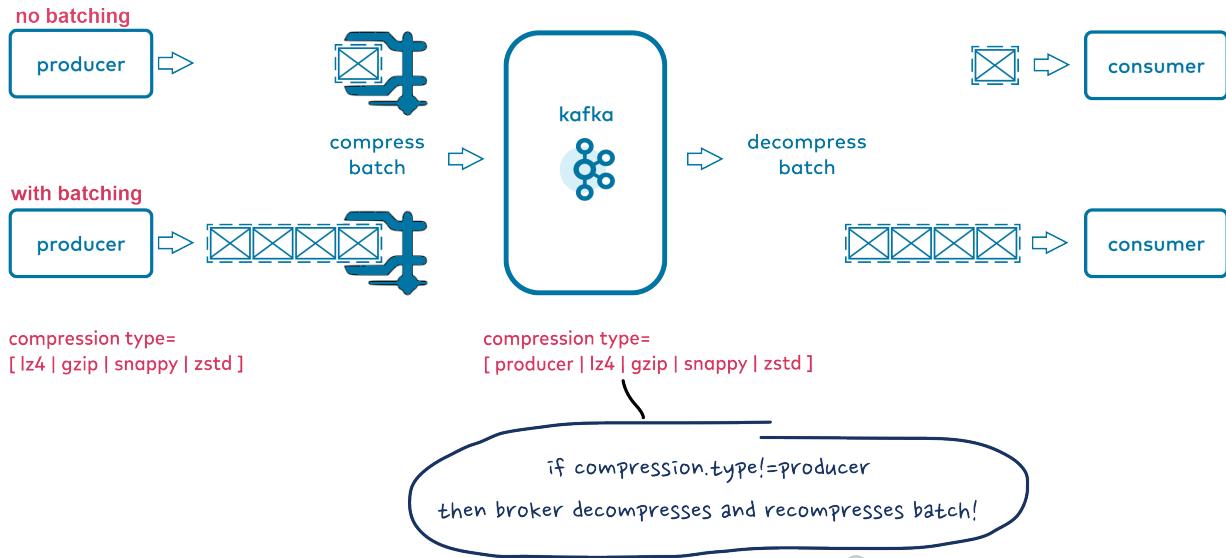
Producer Compression



Another important factor when tuning a producer is compression.

- Producer compression is activated by setting the `compression.type` property. Possible values are `gzip`, `lz4`, `snappy`, or `none` (the default).
- If `buffer.size > 0` then batches of messages that are present in the send buffer before sending will be compressed together. If `buffer.size = 0` then messages will be compressed individually when sent.
- Where batches of more than one message are compressed, the entire batch is passed as a single unit to a consumer. As a consumer iterates over the messages in the batch it commits consumed offsets back to the broker. If the offset corresponds to a message that is mid-batch and the consumer shuts down, then the broker will send the **entire batch** again the next time the consumer group polls - this may lead to duplicate message processing.

Broker Compression



- Broker compression is set via the `compression.type` property. This may be overridden at the topic level via a config override of the same name. Possible values are `zstd`, `gzip`, `lz4`, `snappy`, `uncompressed`, or `producer` (the default).
- When the default value of `producer` is used, the broker will take messages and batches as-is from the producer without modifying/recompressing the payload.
- If the broker has a different compression setting, it will decompress the messages, and compress them in the stated format (if using something other than uncompressed) before writing them to disk.
- If you do not want to incur this recompression overhead, you need to ensure that your broker compression is set to `producer`, or that it is set to the value typically used by the producer.

Compression Summary

- Compression is usually the **dominant** part of `producer.send()`
 - Speed of different compression types differs **A LOT**
 - Compression is in user thread → add more user threads if compression is slow
-

This slide presents a representative example of a test in which data transfer metrics were monitored and collected using several compression types. Another set of test results that can be found in KIP-110 are provided below. It is important to note that compression is usually the dominant part of the `producer.send()` call, and that the speed varies widely among the various compression types.

As illustrated in the two sets of test results, compression not only affects the throughput but also the bandwidth and the over the wire traffic.

From KIP-110

name	ratio	compression	decompression
zstd 1.3.4-1	2.877	470 MB/s	2060 MB/s
lz4 1.8.1	2.101	750 MB/s	3700 MB/s
snappy 1.1.4	2.091	530 MB/s	1820 MB/s

Concurrent in-flight requests

- `max.in.flight.requests.per.connection > 1` means pipelining
- In general, pipelining:
 - gives **better throughput**
 - may cause **out of order** delivery when retry occurs
 - **Excessive pipelining** → drop of throughput
 - lock contention
 - worse batching



Idempotent producer prevents out of order messages

- Benchmarks on request pipelining can be found here: <https://cwiki.apache.org/confluence/display/KAFKA/An+analysis+of+the+impact+of+max.in.flight.requests.per.connection+and+acks+on+Producer+performance>.
- Also note that **pipelining** will not cause out of order messages on retry with the idempotent producer!
- excessive pipelining can lead to a degradation in throughput due to the fact that it may lead to lock contention and/or worse batching behavior in the producer request.

Tuning Scenarios (1/2)

Optimize Throughput

- `batch.size`: increase to 100,000 - 200,000 (default 16,384)
- `linger.ms`: increase to 10 - 100 (default 0)
- `compression.type=lz4` (default none)
- `acks=1` (default 1)
- `buffer.memory`: increase if there are a lot of partitions (default 33,554,432)

Optimize Latency

- `linger.ms=0` (default 0)
- `compression.type=None` (default none, i.e., no compression)
- `acks=1` (default 1)

Notes:

Optimize Throughput:

- We should increase the batch size to a higher number than the default
- We should set the `linger.ms` variable to 10-100 ms
- Select `lz4` as compression type
- Use `acks=1`
- And finally increase the buffer memory from its default if there are a lot of partitions

Optimize Latency:

- `linger.ms=0` is not always optimal for low latency. Latency can often be improved with at least a small amount of batching, as it's less likely to bottleneck the request pipeline and create backpressure in the Producer
- Choosing `compression.type=None` not always means lower latency - it can depend on CPU vs I/O cost

From Kafka Core team:

- if you don't have a lot of requests, `linger.ms=0` will give you the best latency however, batching improves efficiency and can improve latency given enough load. In such cases, a small but non-zero `linger.ms` helps
- same for compression, if you use `lz4` it can reduce latency by reducing network traffic,

but it depends on various factors

hitesh@datacouch.io

Tuning Scenarios (2/2)

Optimize Durability

- `replication.factor=3` (topic override available)
- `acks=all` (default 1)
- `enable.idempotence=true` (default false), to handle message duplication and ordering
- `max.in.flight.requests.per.connection=1` (default 5), to prevent out of order messages when not using an idempotent producer

Optimize Availability

- Not relevant

We cannot optimize for all 4 service requirements at the same time. On this slide we give recommendations for cases where one of the requirements clearly trump the others.

Note:

- `max.in.flight.requests.per.connection=1` only needed when not using idempotent producer
- `min.insync.replicas` (broker setting per topic) is commonly defined together with `acks=all`

Producer Performance Tool

Sample Command:

```
$ kafka-producer-perf-test \
--num-records 1000000 \
--record-size 1000 \
--topic sample-topic \
--throughput 1000000 \
--print-metrics \
--producer-props bootstrap.servers=kafka:9092 \
    max.in.flight.requests.per.connection=1 \
    batch.size=100000 \
    compression.type=lz4
```

Above shell is wrapper around:

```
org.apache.kafka.tools.ProducerPerformance
```

Before running the perf test, create the topic with:

```
$ kafka-topics --bootstrap-server kafka-1:19092,kafka-2:29092 \
--create \
--topic sample-topic \
--partitions=6 \
--replication-factor=3
```

Tuning librdkafka based Clients

- Batch write requests
 - Same optimizations as Java producer
 - Important settings:
 - `batch.num.messages`
 - `queue.buffering.max.ms`
 - `compression.codec`
 - `request.required.acks`
-

- Batch write requests: send multiple records at once to Kafka
- In general to tune librdkafka based clients use the same settings as for the Java producer
- Configurations settings can be set like this for e.g. .NET:

```
// pseudo code...
var properties = GetFromConfigurationFile();
var config = new ProducerConfig(properties);
var producer = new Producer< TKey, TValue>(config);
...

```

Tuning the Confluent REST Proxy

- Tune standard Java producer settings via REST Proxy's config file
 - Batch write requests
 - Reuse a session to push data
 - Avoid large messages
-
- All the standard Java producer settings are supported via the REST Proxy's properties file. Clients can batch messages to send to the REST Proxy, but internally, the normal Java producer is being used to actually send messages to Kafka, and it needs to be tuned appropriately.
 - It is recommended to batch write requests, that is, send multiple records at once in a single **POST** to the endpoint `/topics/<topic-name>`
 - Don't create a new session to the REST proxy each time you write a batch of data but rather reuse a single session. Below is an example with Python and the **requests** library:

Bad:

```
def produce(payload):
    headers = {'Content-Type': 'application/vnd.kafka.binary.v1+json'}
    response = requests.post('http://rest-proxy:8082/topics/test', headers=headers,
    json=payload)
    ...
```

Good:

```
def send_messages(url, payload):
    session = requests.Session()
    headers = {'Content-Type': 'application/vnd.kafka.binary.v1+json'}
    response = session.post(url, headers=headers, data=payload)
    ...
```

- The performance characteristics of the Confluent REST Proxy (and Kafka in general) change depending on the message size. Very small messages will be handled faster than very large messages. The design should take this difference into consideration and favor smaller messages.

Review



Question:

- What data formats are you envisioning to use in your Kafka powered streaming platform? What reasons did influence your choice?
- Envision a typical topic you are going to produce or use. How many partitions will you create? Justify your choice.
- Assuming you want to write a producer for highest possible throughput, where do you start to tune?

To conclude this module please answer the questions listed on the slide.

hitesh@datacouch.io

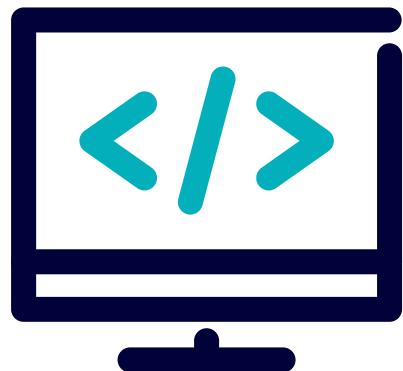
Further Reading

- Decoupling Systems with Apache Kafka, Schema Registry and Avro: <https://cnfl.io/decoupling-systems>
- Optimizing Your Kafka Deployment: <http://cnfl.io/optimize-kafka-deployment>
- Introduction to Schemas in Apache Kafka with the Confluent Schema Registry: <https://bit.ly/2RvNkVr>
- Data Types and Serialization: <https://cnfl.io/data-types>
- The problem of managing schemas: <https://www.oreilly.com/ideas/the-problem-of-managing-schemas>
- How to choose the number of topics/partitions in a Kafka cluster? <https://cnfl.io/nbr-of-partitions>
- Reliability Guarantees in Kafka <https://cnfl.io/summit-reliability>
- Producer Perf. Tuning for Apache Kafka: <https://cnfl.io/slides-kafka-perf-tuning>
- Compression in Kafka now 34% faster: <https://cnfl.io/kafka-compression-faster>

Lab: Tuning Producers

Please work on **Lab 10b: Tuning Producers**

Refer to the Exercise Guide

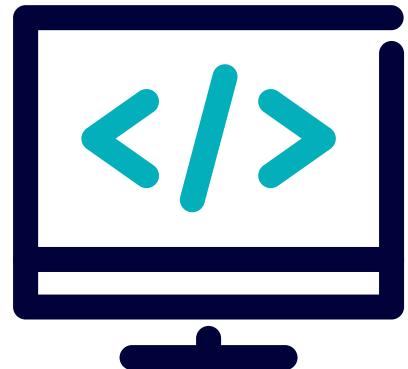


hitesh@datacouch.io

Lab: Tuning Producers - Selecting the Best Partition Strategy

Please work on **Lab 10c: Tuning Producers - Selecting the Best Partition Strategy**

Refer to the Exercise Guide



hitesh@datacouch.io

11: Troubleshooting & Tuning Consumers



CONFLUENT
Global Education

hitesh@datacouch.com

Module Overview



This module contains two lessons:

1. Troubleshooting Consumers
2. Tuning Consumers

hitesh@datacouch.io

a: Troubleshooting Consumers

Description

Consumer lag troubleshooting. Reading and resetting offsets. Rebalancing.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Find the root cause of consumer lag
- List the options available to reset a consumer group
- Use the tool `kafka-consumer-groups` to reset a whole consumer group
- List the (downstream) implications of resetting a consumer group

hitesh@datacouch.io

Troubleshoot Consumer Lag

Three phases:

1. Understand structure of environment
 2. Determine cluster health
 3. Test consumption
-

The next three pages will go into the details of each step. Slides will have high-level steps, and there are some specific commands given in the handbook after the third slide.

hitesh@datacouch.io

Troubleshoot Consumer Lag: (1) Understand structure of environment

1. How many **brokers**?
2. Which **topic** is consumed?
3. Understand **details** of topic

hitesh@datacouch.io

Troubleshoot Consumer Lag: (2) Determine cluster health

1. Are there under-replicated topics?
2. What is the name of the consumer group?
3. How many consumers?
4. What's their consumption status?
5. Note partitions where lag is increasing
6. Confirm current broker configuration
7. Check CPU load on brokers
8. Are JMX metrics enabled?

hitesh@datacouch.io

Troubleshoot Consumer Lag: (3) Test consumption

1. Use `kafka-console-consumer` to consume from each affected partition
-

From time to time, you will experience **Consumer Lag** and even **timeouts** when Consuming from Kafka. Here we outline how to troubleshoot and determine the root cause for such issues.

A) Understand the structure of the environment

We first need to understand the structure of the Kafka environment. It's critical that we identify the following:

1. How many brokers are in the cluster?

We should get a list of the hostnames and BrokerID associated with each broker, e.g.,
`host1.foo.com:BrokerID=1`

2. We need to confirm the Topic being consumed from the customer and note it.
3. We should then confirm the details of the Topic:

```
kafka-topics \
  --bootstrap-server <hostname>:9092 \
  --describe \
  --topic <topicName>
```

B) Determine Cluster health

Once we have retrieved the general structure of the cluster, we should review it for general health issues which could affect Consumption.

1. Check for Under-replicated partitions on the cluster

```
kafka-topics \
  --bootstrap-server <hostname>:9092 \
  --describe --under-replicated-partitions
```

Should we see under replicated partitions, we should investigate the cause of this before continuing.

2. We should confirm the affected Consumer group name:

```
kafka-consumer-groups \
--list \
--bootstrap-server <hostName>:9092
```

3. We need to confirm how many Consumers are in the Consumer group and the current consumption status:

```
kafka-consumer-groups \
--bootstrap-server <hostName>:9092 \
--describe --group <my-group>

TOPIC           PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG
CONSUMER-ID          HOST
CLIENT-ID
my-topic          0          2              4
consumer-1-029af89c-873c-4751-a720-cefd41a669d6 /127.0.0.1   2
consumer-1
my-topic          1          2              3
consumer-1-029af89c-873c-4751-a720-cefd41a669d6 /127.0.0.1   1
consumer-1
my-topic          2          2              3
consumer-2-42c1abd4-e3b2-425d-a8bb-e1ea49b29bb2 /127.0.0.1   1
```

From the above example output, we can see there are three Consumers with the first having a higher level of LAG.

4. Confirm if the LAG is increasing on one partition or on several partitions, note which are affected.
5. Confirm the current Broker configuration, specifically around thread counts. Defaults are as follows:

```
num.network.threads = 3
num.io.threads = 8
num.replica.fetchers = 1
```

6. Check the CPU load on the Brokers to confirm there is enough CPU for the number of threads which have been specified, using eg. `top`
7. Confirm if the customer has JMX metrics enabled or not. If they do, confirm that the network threads and io threads are not overloaded.

Provided no issues are found, you can start Testing Consumption.

C) Test Consumption

Once we have determined which partition(s) are lagging, we can test Consumption using the Console Consumer.

1. Using the Console consumer, Consume from each affected partition to confirm their isn't a problem with the partition itself:

```
kaka-console-consumer \
--bootstrap-server localhost:9092 \
--from-beginning \
--topic test1 \
--new-consumer \
--partition <Integer: partition> \
> partitionTest.txt
```

The above will pipe out the contents of the given partition to a file. Should this complete successfully, then the partition is healthy.

Conclusion

After analyzing the above and **not** finding any issues, you can safely state the issue is on the **Consuming Application side**.

Reset Consumer Group Offset

Motivation:

- Reprocess old records
 - Does not require a code change

WHY?

- Bug in consumer code is fixed
- New and better consumer algorithms/models

Example:

- Reset to first offset since 01 January 2019, 00:00:00 hrs UTC

```
$ kafka-consumer-groups --reset-offsets \
--group <Group ID> \
--bootstrap-server kafka:9092 \
--to-datetime 2019-01-01T00:00:00.000
```

Reason:

- the option to reset offsets from outside of the application will enable a cleaner and consistent way to achieve reprocessing and move along the topic.
- No changes to the consumer code is needed (e.g. `KafkaConsumer#seek()`)

There is many reasons why one may want to reprocess records:

- the team discovers a bug in the consumer's code, fixes it and wants to reprocess the data with the correct computational logic
- a new and better consumer logic has been developed that calculates results more precisely
- other reasons?

Code:

- With the parameter `--reset-offsets` one can reset all topics (with all partitions) consumed by a given consumer group to the earliest offset represented by a given time stamp.
- Other more sophisticated reset scenarios are possible, e.g. to only reset some explicitly

listed partitions of a topic to a chosen offset.

hitesh@datacouch.io

Read Consumer Group Offsets

```
$ kafka-consumer-groups --bootstrap-server kafka:9092 --describe --group my-group
```

GROUP HOST	TOPIC CLIENT-ID	PARTITION	CURRENT-OFFSET	LOG-END-OFFSET	LAG	CONSUMER-ID
my-group /172.28.0.9	my-topic consumer-1	0	2536818	2542828	6010	consumer-1...
my-group /172.28.0.9	my-topic consumer-1	1	2519420	2525328	5908	consumer-1...
my-group /172.28.0.11	my-topic consumer-3	4	2528207	2533934	5727	consumer-3...
my-group /172.28.0.11	my-topic consumer-3	5	2510668	2515775	5107	consumer-3...
my-group /172.28.0.10	my-topic consumer-2	2	2541226	2546270	5044	consumer-2...
my-group /172.28.0.10	my-topic consumer-2	3	2509518	2515056	5538	consumer-2...

It is very easy to get the current offsets for all topics and partitions consumed by a given consumer group. Just use the tool `kafka-consumer-groups` with the parameter `--describe`

Read __consumer_offsets Topic (1)

If broker is online:

```
$ kafka-console-consumer \
--consumer.config consumer.properties \
--from-beginning \
--topic __consumer_offsets \
--bootstrap-server kafka:9092 \
--formatter \
"kafka.coordinator.group.GroupMetadataManager\$OffsetsMessageFormatter"
```

or

```
...
--formatter \
"kafka.coordinator.group.GroupMetadataManager\$GroupMetadataMessageFormatter"
```

- The __consumer_offsets topic contains consumer offset data and other consumer group metadata.

Reasons you may want to read this topic:

- See when a consumer last committed an offset, and what the offset is
- See which broker is the group coordinator for a consumer group

There are two ways to read this topic:

- when the broker is online
- when the broker is offline

Both options will print a lot of data.



- `kafka.coordinator.group.GroupMetadataManager\$GroupMetadataMessageFormatter` provides additional information about consumer group partition assignments and status

Read __consumer_offsets Topic (2)

If broker is offline:

```
$ kafka-dump-log \
--files /var/lib/kafka/data/__consumer_offsets-<X>.log \
--offsets-decoder \
--print-data-log
```

hitesh@datacouch.io

Consumer Group Rebalancing Issues

- Frequent rebalancing:
Metric: `join-rate`, `sync-rate`

- Long rebalancing times:

Metrics:

- `join-time-avg`
- `join-time-max`
- `sync-time-avg`
- `sync-time-max`

-
- **Frequent rebalancing:** the common cause of frequent rebalances is that your consumer takes too long to process batches. Consumers use a separate thread to send an alive signal to the broker which is their group coordinator. If the group coordinator does not receive an alive signal for longer than defined in the `session.timeout.ms` property it considers the according consumer to be dead and removes it from the consumer group. This of course triggers a rebalance. But to avoid that a consumer is considered to be healthy although its poll thread is stuck, there is also the need to poll occasionally. The maximum time interval allowed between polls is defined in the property `max.poll.interval.ms`, whose default is set to 5 min. Now if your batch handling exceeds the value defined in `max.poll.interval.ms`, then the group coordinator will also remove the corresponding consumer from the consumer group and trigger a rebalance. So the solution is to either make the batches smaller or the max poll interval longer.
 - `join-rate`: The number of group joins per second. Group joining is the first phase of the rebalance protocol. A large value indicates that the consumer group is unstable and will likely be coupled with increased lag.
 - `sync-rate`: The number of group syncs per second. Group synchronization is the second and last phase of the rebalance protocol. Similar to `join-rate`, a large value indicates group instability.
 - **Long rebalancing times:** Rebalancing can take a long time for (stateful) consumers as found in e.g. a Kafka Streams or a ksqlDB application. In older versions of streams the state store recovery was included as part of the rebalance, but state store recovery is now done on the main loop so it shouldn't slow down the rebalance.
 - `join-time-avg` and `join-time-max`: The average and maximum time taken for a group rejoin. This value can get as high as the configured `max.poll.interval.ms` for the consumer, but should usually be lower.

Other Important Metrics for Troubleshooting

- Consumer lag: `records-lag-max`
- Consumer throughput:
 - `fetch-rate`
 - `fetch-latency-avg`
 - `fetch-latency-max`
 - `records-per-request-avg`
 - `bytes-consumed-rate`

There are a few other JMX metrics that are useful to troubleshoot issues with a consumer group.

- To monitor consumer lag use:
 - `records-lag-max`: The maximum lag in terms of number of records for any partition in this window.
 - An increasing value over time is your best indication that the consumer group is not keeping up with the producers.
- To monitor consumer throughput use:
 - `fetch-rate`: The number of fetch requests per second.
 - `fetch-latency-avg|max`: The average/max time taken for a fetch request.
 - `records-per-request-avg`: The average number of records in each request.
 - `bytes-consumed-rate`: The average number of records consumed per second.

Review



Question:

You realized that one of your consumer instances in consumer group `Foo` processed the data from partitions 1, 4 and 7 of topic `topic-1` incorrectly. You decide to start the processing of this consumer instance over from scratch. How can you do that?

You can reset the offset of each partition of topic `Foo` using the CLI tool `kafka-consumer-groups`:

```
$ kafka-consumer-groups --reset-offset \
--group <Group ID> \
--bootstrap-server kafka:9092 \
--to-datetime 2019-01-01T00:00:00.000
--topic topic-1:1,4,7
...
```

Further Reading

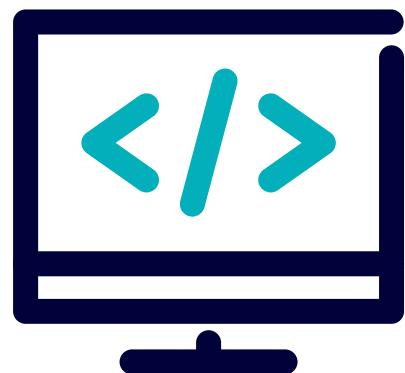
- KIP-122: Add Reset Consumer Group Offset tooling:
<https://cwiki.apache.org/confluence/display/KAFKA/KIP-122%3A+Add+Reset+Consumer+Group+Offsets+tooling>
- Managing Consumer Groups:
https://kafka.apache.org/documentation/#basic_ops_consumer_group

hitesh@datacouch.io

Lab: Troubleshooting Consumers

Please work on **Lab 11a: Troubleshooting Consumers**

Refer to the Exercise Guide



hitesh@datacouch.io

b: Tuning Consumers

Description

Consumer/partition parallelism. Fault detection and rebalancing. Fetch requests. Details specific to librdkafka consumers.

hitesh@datacouch.io

Learning Objectives

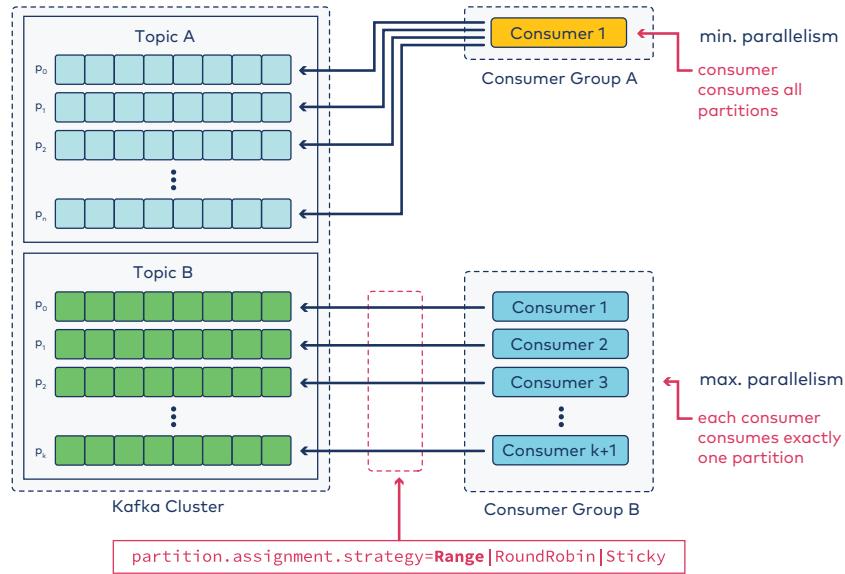


Completing this lesson and associated exercises will enable you to:

- Optimally load balance the load in a consumer group
- Select the best partition assignment strategy for your application
- List 3 to 4 factors that most impact your consumer's performance
- Describe the impact when number of partitions >> number of consumers
- Identify ways to make REST-based and non-Java consumers more efficient

hitesh@datacouch.io

Maximize Parallelism



- Partitions of a topic are assigned to a consumer **as a whole**
- Each partition is only consumed by a single consumer of a given consumer group
- A consumer can consume multiple partitions
- The **partition assignment strategy** determines which partition is assigned to which consumer
- We have 3 strategies: **Range**, **RoundRobin** and **Sticky**. In the image we have the default **Range**
- The work of consuming can be parallelized by adding more than one consumer to a consumer group
- The minimum parallelism is given for a consumer group with a single consumer
- The maximum parallelism can be achieved if each consumer has to consume exactly one partition
- If the consumer group has more consumers than the topic has partitions then the extra consumers will be idle

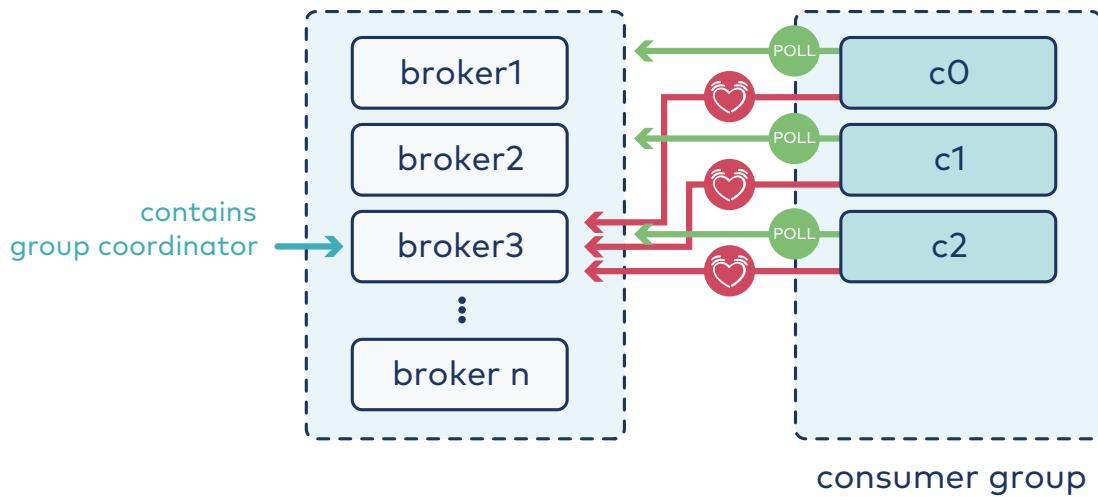
About the number of partitions, important considerations:

- How many brokers in the cluster?
- How many consumers (in the consumer group)?
- Do you have specific partition requirements?

- Keep partition sizes manageable
- Do not over-allocate partitions!

hitesh@datacouch.io

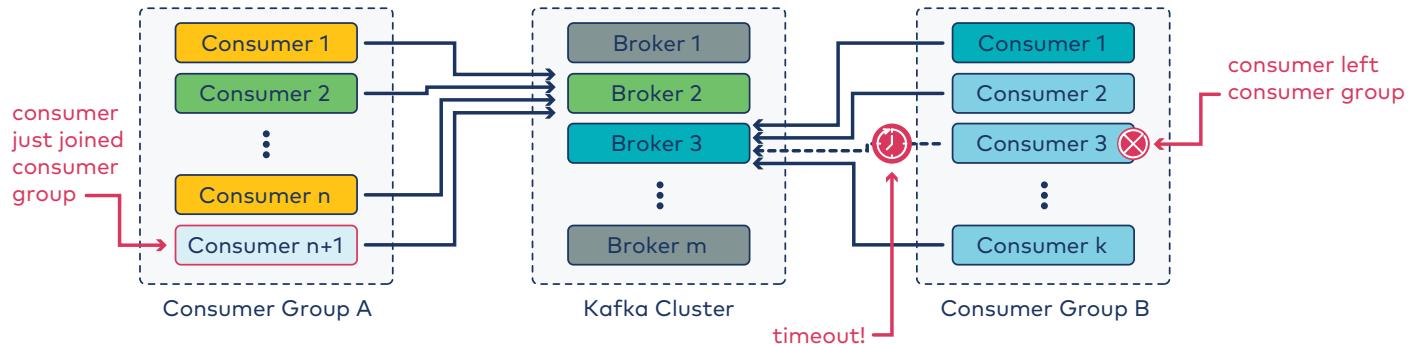
Consumer Liveness



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 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` after exceeded the corresponding consumer is considered dead and the group coordinator triggers a partition reassignment. The parameter `heartbeat.interval.ms` in turn defines how long the interval between two successive heartbeat signals are.

Joining or Leaving a Consumer Group



The partition reassignment (or "Rebalance") is also triggered when either a consumer joins the group or an existing consumer leaves the group. The latter can happen if for example the consumer crashes or is shutdown.

On the graphic we see a **consumer group A** that just got a new member. The group coordinator (broker 2) will trigger a reassignment/rebalance. We also have a **consumer group B** which just lost a member and its group coordinator (broker 3) will notice this due to the fact that there is no more liveness signal from that consumer. Thus it triggers a partition reassignment.



It is not necessary that each consumer group has its own group coordinator. A broker can be the coordinator of multiple consumer groups.

How does a Consumer Group identifies which broker will be its coordinator.

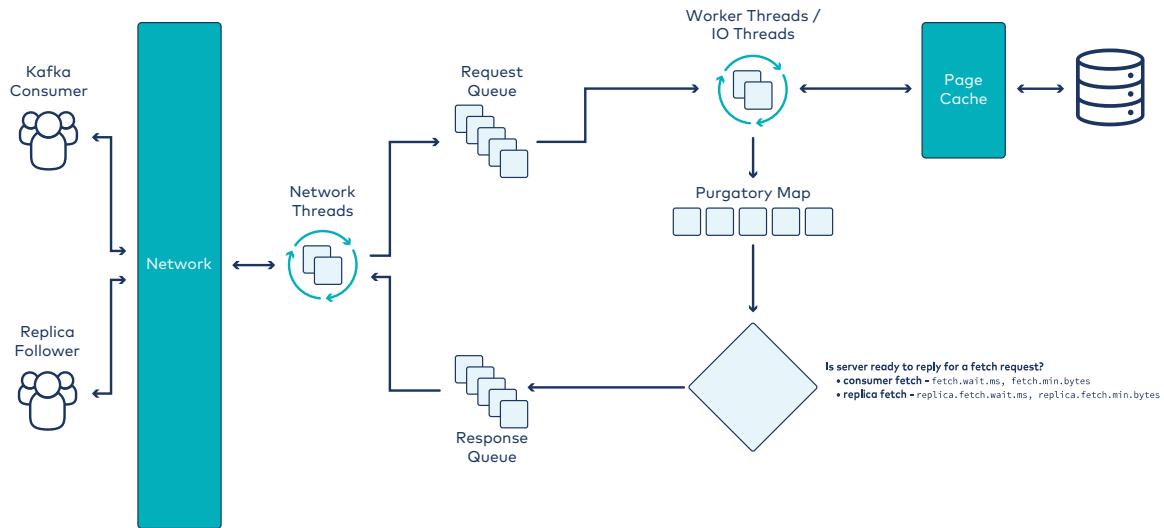
hash(group.id) % number-of-partitions in `_consumer_offsets` - leader for that partition is coordinator, and of course if that broker fails, the coordinator will failover automatically to the new leader for that partition. This strategy also allows the coordinator to update offsets for the group locally without any remote communication with other brokers.



With the example of a consumer leaving the group, it's only necessary to wait for the `session.timeout.ms` in the event of an unexpected failure. If the consumer shuts down gracefully, it will proactively notify the coordinator that it's leaving, and a rebalance will happen immediately. Also, these are not the only changes will cause a rebalance - any changes to the topic subscription or number of partitions in the subscribed topic(s) will also cause this to happen - the coordinator monitors topic metadata.

hitesh@datacouch.io

Fetch Request on a Broker



The anatomy of a consumer request on the broker looks very much like a producer request. The only exception is the **purgatory** where this time the request does not sit and wait for the ISRs to reply but the request waits until either of the two consumer properties:

- `fetch.max.wait.ms`
- `fetch.min.bytes`

is exceeded. By default former is `500ms` and the latter is equal to `1`.

Apache Kafka has a data structure called the "request purgatory". The purgatory holds any request that hasn't yet met its criteria to succeed but also hasn't yet resulted in an error.



A fetch request with `fetch.min.bytes=1` won't be answered until there is at least one new byte of data for the consumer to consume. This allows a "long poll" so that the consumer need not busy wait checking for new data to arrive. Note that this can be overridden by an expiring `fetch.wait.max.ms`!

Tuning Consumer Fetch Requests

`fetch.min.bytes` vs. `fetch.max.wait.ms`

- What if topic does not have a lot of data?
 - Reduce load on broker by letting fetch requests wait a bit for data
 - Add latency to increase throughput
 - Caution: do not fetch more than you can process!
-
- What if topic doesn't have a lot of data? Then it is possible to minimize latency by leaving the `min.fetch.bytes` on its default of 1 (and also leave the `fetch.wait.max.ms` on its default)
 - Reduce load on broker by letting fetch requests wait a bit for data
 - Add latency to increase throughput: set `min.fetch.bytes` to a high value and select a reasonable `fetch.wait.max.ms` time, e.g. 500ms.
 - Careful: don't fetch more than you can process! Otherwise we have the same problem as mentioned on the previous slide, where the consumer is considered as dead and thus a rebalance happens in the group. Use `fetch.max.bytes` and `max.partition.fetch.bytes` as a way to configure upper limits on how much data is fetched at a time.

Commits Take Time

- Commit less frequently
 - Commit asynchronously
-

Committing the consumer offset takes a non negligible amount of time. If your use case and the respective consumers allow for it, then commit less frequently. Note, the consequence of this is that, if the consumer crashes some messages are reprocessed. With EOS transactional semantics though, this is not an issue.

Automatic Commit

The easiest way to commit offsets is to allow the consumer to do it for you. If you configure `enable.auto.commit=true`, then every five seconds the consumer will commit the largest offset your client received from `poll()`. The five-second interval is the default and is controlled by setting `auto.commit.interval.ms`. Just like everything else in the consumer, the automatic commits are driven by the poll loop. Whenever you poll, the consumer checks if it is time to commit, and if it is, it will commit **the offsets it returned in the last poll**.

Before using this convenient option, however, it is important to understand the consequences.

Consider that, by default, automatic commits occur every five seconds. Suppose that we are three seconds after the most recent commit and a rebalance is triggered. After the rebalancing, all consumers will start consuming from the last offset committed. In this case, the offset is three seconds old, so all the events that arrived in those three seconds will be processed twice. It is possible to configure the commit interval to commit more frequently and reduce the window in which records will be duplicated, but it is impossible to completely eliminate them. But note, with EOS transactions, there's no adverse effect to duplicate message processing.

Commit Current Offset

Most developers exercise more control over the time at which offsets are committed—both to eliminate the possibility of missing messages and to reduce the number of messages duplicated during rebalancing. The consumer API has the option of committing the current offset at a point that makes sense to the application developer rather than based on a timer.

By setting `enable.auto.commit=false`, offsets will only be committed when the application explicitly chooses to do so. The simplest and most reliable of the commit APIs is `commitSync()`. This API will commit the latest offset returned by `poll()` and return once the offset is committed, throwing an exception if commit fails for some reason.

Asynchronous Commit

One drawback of manual commit is that the application is blocked until the broker responds to the commit request. This will limit the throughput of the application. Throughput can be improved by committing less frequently, but then we are increasing the number of potential duplicates that a rebalance will create.

Another option is the asynchronous commit API, `commitAsync()`. Instead of waiting for the broker to respond to a commit, we just send the request and continue on. The drawback is that while `commitSync()` will retry the commit until it either succeeds or encounters a non-retrievable failure, `commitAsync()` will not retry.



Consumers can choose to manage offsets manually outside of the normal `--consumer_offsets` topic. Sink connectors often do this to provide an exactly-once guarantee, by storing the offsets atomically in the same external system where the data is being written.

Monitoring

When tuning, observe:

- records-lag-max
 - fetch-rate
 - fetch-latency
 - records-per-request, bytes-per-request
-

To help you tuning the consumer group(s) you absolutely need a feedback loop. The best way to do so is to monitor essential parameters of the consumers. The ones listed on the slide are the most important ones.

hitesh@datacouch.io

Tuning via Partition Assignment Strategy

Options:

- Range (default)
 - might not distribute workload evenly
- RoundRobin
 - uses all consumer resources evenly
- Sticky
 - preserves assignments if possible and reasonable

Consumer property:

```
partition.assignment.strategy=Range|RoundRobin|Sticky
```

The property `partition.assignment.strategy` allows us to choose a partition-assignment strategy. The default is **Range**. One can implement their own strategy

The overall importance of this setting is "medium"

Range

Assigns to each consumer a consecutive subset of partitions from each topic it subscribes to. So if consumers C1 and C2 are subscribed to two topics, T1 and T2, and each of the topics has three partitions, then C1 will be assigned partitions 0 and 1 from topics T1 and T2, while C2 will be assigned partition 2 from those topics. Because each topic has an uneven number of partitions and the assignment is done for each topic independently, the first consumer ends up with more partitions than the second. This happens whenever Range assignment is used and the number of consumers does not divide the number of partitions in each topic neatly.

Note that **Range** assignment is useful for joins between co-partitioned topics.

RoundRobin

Takes all the partitions from all subscribed topics and assigns them to consumers sequentially, one by one. If C1 and C2 described previously used RoundRobin assignment, C1 would have partitions 0 and 2 from topic T1 and partition 1 from topic T2. C2 would have partition 1 from topic T1 and partitions 0 and 2 from topic T2. In general, if all consumers are subscribed to the same topics (a very common scenario), RoundRobin assignment will end up with all consumers having the same number of partitions (or at most 1 partition

difference).

Sticky see: <https://cwiki.apache.org/confluence/display/KAFKA/KIP-54+-+Sticky+Partition+Assignment+Strategy>

Neither the **Range** nor the **RoundRobin** strategies consider what topic partition assignments were before reassignment, as if they are about to perform a fresh assignment. Preserving the existing assignments could reduce some of the overheads of a reassignment.

The Sticky Assignor serves two purposes. First, it guarantees an assignment that is as balanced as possible, meaning either:

- the numbers of topic partitions assigned to consumers differ by at most one; or
- if a consumer A has 2+ fewer topic partitions assigned to it compared to another consumer B, none of the topic partitions assigned to A can be assigned to B.

When starting a fresh assignment, the Sticky Assignor would distribute the partitions over consumers as evenly as possible. Even though this may sound similar to how round robin assignor works, the second example below shows that it results in a more balanced assignment.

Second, during a reassignment the Sticky Assignor would perform the reassignment in such a way that in the new assignment,

- topic partitions are still distributed as evenly as possible, and
- topic partitions stay with their previously assigned consumers as much as possible.



The sticky partition assignment algorithm favors fairness over stickiness. Therefore, some partitions may change their consumer towards a fair assignment.

Tuning librdkafka based Clients

- Same optimizations as Java consumer
- Important Properties:

`fetch.wait.max.ms`

`fetch.min.bytes`

- Commit less frequently
- Commit asynchronously
- Avoid large messages

-
- In general to tune librdkafka based clients use the same settings as for the Java consumer
 - The two settings about max wait time and min bytes influence the performance characteristic of a librdkafka based consumer similar to a Java based consumer (throughput vs. minimal latency)
 - Committing the consumer offset takes a non negligible amount of time. If your use case and the respective consumers allow for it, then commit less frequently. Note, the consequence of this is that, if the consumer crashes some messages are reprocessed.
 - The performance characteristics of the Kafka clients based on `librdkafka` (and Kafka in general) change depending on the message size. Very small messages will be handled faster than very large messages. The design should take this difference into consideration and favor smaller messages.

Tuning Clients that use REST Proxy

- Reuse a session to pull data
- Avoid large messages
- Tune `fetch.min.bytes` & `consumer.request.timeout.ms`

-
- Regarding the reuse of a session: below is an example with Python and the `requests` library:

Bad:

```
def produce(url):  
    headers = {'Accept': 'application/vnd.kafka.binary.v1+json'}  
    response = requests.get(url, headers=headers)  
    ...
```

Good:

```
def get_messages(url):  
    session = requests.Session()  
    headers = {'Accept': 'application/vnd.kafka.binary.v1+json'}  
    response = session.get(url, headers=headers)  
    ...
```

- The performance characteristics of the Kafka REST Proxy (and Kafka in general) change depending on the message size. Very small messages will be handled faster than very large messages. The design should take this difference into consideration and favor smaller messages.
- The following two settings are relevant when max throughput or min latency is required:
 - `fetch.min.bytes`: The minimum number of bytes in message keys and values returned by a single request before the timeout of `consumer.request.timeout.ms` passes.
 - `consumer.request.timeout.ms`: The maximum total time to wait for messages for a request if the maximum request size has not yet been reached. The consumer uses a timeout to enable batching. A larger value will allow the consumer to wait longer, possibly including more messages in the response. However, this value is also a lower bound on the latency of consuming a message from Kafka. If consumers need low latency message delivery, this setting should be reduced.

Tuning Scenarios

- Optimize Latency

`fetch.min.bytes=1` (default 1)

- Optimize Durability

`enable.auto.commit=false` (default true)

`isolation.level=read_committed` (when using EOS transactions)

- Optimize Availability

`session.timeout.ms` (set as low as feasible, default 45 s.)

hitesh@datacouch.io

Review



Question:

Assuming your consumers need minimal latency, what are the first things you address? Where do you tune?

To keep the latency as low as possible:

- keep the parameter `fetch.min.bytes=1` (default), so that upon arrival of the first record the fetch request returns
- use "small" record payloads
- avoid highly structured schemas for the record key and value (deserialization takes time)
- tune your compression type and batching to the record size and bandwidth available (monitoring while tuning is essential)

hitesh@datacc.io

Further Reading

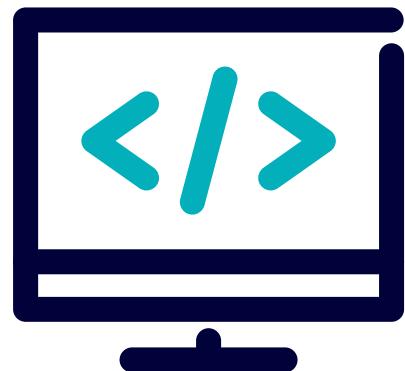
- Running Kafka at Scale:
<https://www.slideshare.net/gwenshap/kafka-at-scale-facebook-israel>
- Apache Kafka, Purgatory, and Hierarchical Timing Wheels:
<https://www.confluent.io/blog/apache-kafka-purgatory-hierarchical-timing-wheels/>
- Kafka Protocol Guide:
<http://kafka.apache.org/protocol.html>
- REST Proxy, Important Configuration Options:
<https://docs.confluent.io/current/kafka-rest/docs/deployment.html#important-configuration-options>
- **librdkafka** Repository on GitHub: <https://github.com/edenhill/librdkafka>

hitesh@datacouch.io

Lab: Tuning Consumers

Please work on **Lab 11b: Tuning Consumers**

Refer to the Exercise Guide



hitesh@datacouch.io

12: Troubleshooting & Tuning Streams Apps



CONFLUENT
Global Education

hitesh@datacouchce

Module Overview



This module contains two lessons:

1. Troubleshooting Streams Apps
2. Tuning Streams Apps

hitesh@datacouch.io

a: Troubleshooting Kafka Streams & ksqlDB Apps

Description

Common streams apps problems. Metrics. Health checks.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- List 3 to 5 reasons that can lead to a slower than expected Kafka Streams application
- Determine the health of a ksqlDB cluster
- Identify and fix sub-optimal parallelism in a Kafka Streams application

hitesh@datacouch.io

Low Throughput - Kafka Streams

Potential factors:

- low frequency input topic(s)
- available network bandwidth too low
- number of stream threads too low, or
- not enough consumer instances
- low number of partitions

Stateful Apps:

- not enough memory
- slow state store IO
- slow connection to Kafka

Potential Factors:

If you observe that your Kafka Streams application is slower than expected, the following items could be the reason:

- low frequency input topic(s): If the flow of input events is low, then of course the throughput of the streams app is low too. Check production rate of input topic(s)
- num stream threads too low: If you run a streams app instance on a machine with 4 cores/8 threads but use fewer than 8 streams threads, some of the scalability potential is not being leveraged, e.g., increase num threads to 8. Be aware that configuring too many stream threads (exceeding the number of CPU cores or threads) can slow down the app.
- available network bandwidth too low: In a stateless streams application the network is one of the primary limiting factors of throughput. Consider using at least 1Gbit network
- not enough consumer instances: If you have less app instances or threads than partitions in the input topic(s) then you are not leveraging parallelism to its fullest. Either scale out (more app instances) or up (more stream threads). The maximum of instances and/or stream threads is equal to the number of partitions of the input topic.
- low number of partitions: With a high volume input topic having a low number of partitions, parallelism is limited. Consider increasing the number of partitions

Stateful Apps:

When dealing with stateful applications, consider the following:

- not enough memory: Kafka Streams tries to keep as much state cached in memory as possible. Ideally the whole state should fit into RAM for optimal performance. Otherwise a lot of paging is needed.
- slow state store IO: If you are using RocksDB consider to work with SSDs or similar high speed storage. Spindle drives are not optimal here since the DB uses random access.
- slow connection to Kafka: for high availability reasons the local statestore writes their changelog to Kafka. If this connection is slow then this can affect the performance of the streams app. The connection can be slow due to the network between the app and Kafka or due to the fact that the relevant Kafka brokers are overloaded.

hitesh@datacouch.io

Low Throughput - ksqlDB Apps

Potential factors:

- Same as for Kafka Streams apps, plus:
- Too many queries
- Growing consumer lag
- Slow network
- Kafka cluster underpowered

The concerns mentioned on the previous slide for a Kafka Streams application also apply to a ksqlDB application since under the hood, the work is being done by one or more Kafka Streams applications.

ksqlDB applications can also have the following concerns:

- Too many queries: each query competes for resources with all other running queries on the ksqlDB Server cluster
- Growing consumer lag: Simple, fast running queries can starve slower, complex queries. The latter falls behind more and more
- Each query reads all records from the input topic from Kafka and writes all resulting records back to Kafka. If it is a stateful query then its changelog also has to be written to Kafka. This leads to a lot of additional network traffic.
- As mentioned above, query results and intermediate state are written to Kafka for high availability and durability purposes. This takes a toll on the brokers. One has to provision the Kafka cluster accordingly.

Troubleshooting ksqlDB Queries - No data

- No data in the source topic
- No new data arriving in the topic
- ksqlDB consuming from a later offset than for which there is data
- Wrong filter/predicate
- Deserialization errors



Common issues that our customers stumble upon when authoring ksqlDB queries are:

- No data in the source topic: the source topic of the query does not contain any data. Test with `print '<topic-name>' from beginning;`
- No new data arriving in the topic: the input topic is not getting any new data written to it. Thus the query has nothing to process. Test with `print '<topic-name>';` to see if some data is coming in or not.
- ksqlDB consuming from a later offset than for which there is data: This statement is similar to the previous statement, but to be more explicit we still state it here! By default a query starts at the current offset of the input topic. If you want to start from the beginning of the topic use `auto.offset.reset=earliest`.
- Deserialization errors: The definition of the query stream or table does not match with the underlying topic. This can lead to deserialization errors. ksqlDB silently skips records that are not serializable. Error can be found in the logs.
- Wrong filter: There is no input record that matches the filter defined in the ksqlDB query

What's happening under the covers? (1)

```
ksql> DESCRIBE EXTENDED GOOD_RATINGS;
[...]
Local runtime statistics
-----
messages-per-sec:      1.10 total-messages:      2898 last-message: 9/17/18 1:48:47 PM
UTC
failed-messages:      0 failed-messages-per-sec:      0 last-failed: n/a
(Statistics of the local ksqlDB server interaction with the Kafka topic GOOD_RATINGS)
ksql>
```

```
ksql> SHOW QUERIES;
```

Query ID	Kafka Topic	Query String
CSAS_GOOD_IOS_RATINGS_0	GOOD_IOS_RATINGS	CREATE STREAM GOOD_IOS_RATINGS AS SELECT * FROM...

To troubleshoot a query we want to peek under the covers a bit more. Some helpful commands that can be used e.g. from within the ksqlDB CLI are:

- **DESCRIBE EXTENDED:** We need to know how many messages have been processed, when the last message was processed and so on. The simplest option for gathering these metrics comes from within ksqlDB itself. Note that if the metric **failed-messages** is increasing, then this is not a good sign for the health of the query. It could be caused by serialization errors, as discussed before.
- **SHOW QUERIES:** To dig deeper into the execution of queries, we should start by listing the queries that are running

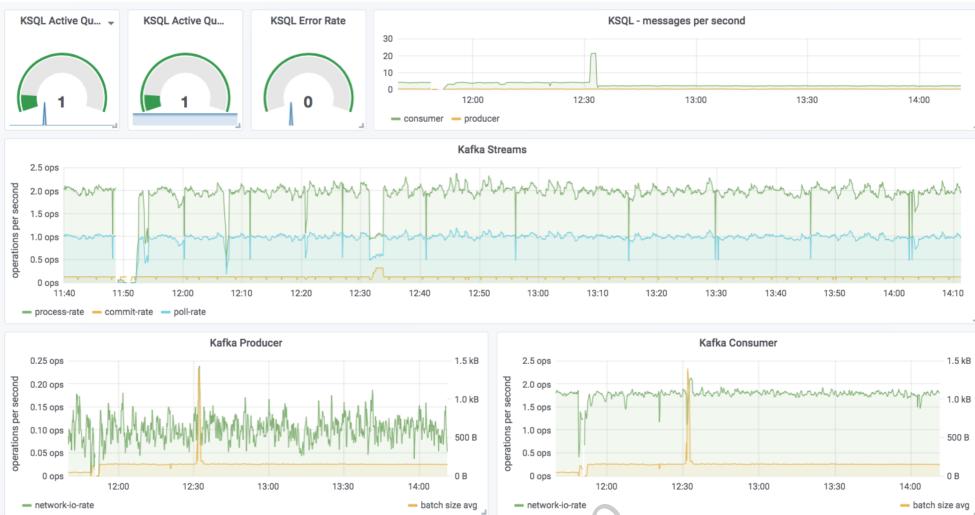
What's happening under the covers? (2)

```
ksql> EXPLAIN CSAS_GOOD_IOS_RATINGS_0;  
[...]  
Execution plan  
-----  
> [ SINK ] Schema: [ROWTIME : BIGINT, ROWKEY : VARCHAR, RATING_ID : BIGINT, USER_ID :  
BIGINT, STARS : INT, ROUTE_ID : BIGINT, RATING_TIME : BIGINT, CHANNEL : VARCHAR,  
MESSAGE : VARCHAR].  
    > [ PROJECT ] Schema: [ROWTIME : BIGINT, ROWKEY : VARCHAR, RATING_ID :  
BIGINT, USER_ID : BIGINT, STARS : INT, ROUTE_ID : BIGINT, RATING_TIME : BIGINT, CHANNEL  
: VARCHAR, MESSAGE : VARCHAR].  
        > [ FILTER ] Schema: [RATINGS.ROWTIME : BIGINT,  
RATINGS.ROWKEY : VARCHAR, RATINGS.RATING_ID : BIGINT, RATINGS.USER_ID : BIGINT,  
RATINGS.STARS : INT, RATINGS.ROUTE_ID : BIGINT, RATINGS.RATING_TIME : BIGINT,  
RATINGS.CHANNEL : VARCHAR, RATINGS.MESSAGE : VARCHAR].  
            > [ SOURCE ] Schema: [RATINGS.ROWTIME  
: BIGINT, RATINGS.ROWKEY : VARCHAR, RATINGS.RATING_ID : BIGINT, RATINGS.USER_ID :  
BIGINT, RATINGS.STARS : INT, RATINGS.ROUTE_ID : BIGINT, RATINGS.RATING_TIME : BIGINT,  
RATINGS.CHANNEL : VARCHAR, RATINGS.MESSAGE : VARCHAR].
```

- **EXPLAIN:** We can examine a query itself and how ksqlDB is going to perform the transformation itself through the execution plan — the same thing as one gets in a RDBMS. We can access it by using the `EXPLAIN <query ID>` command. Note that this command also shows the **topology** the ksqlDB query will use. For brevity we haven't shown this on the slide.

ksqldb Troubleshooting with JMX

- Use **jconsole** or **jmxterm**
- Use **Prometheus** and **Grafana**



- We can look at the JMX metrics with a tool such as JConsole or JMXTerm. Even more useful is persisting these values to a data store, such as InfluxDB, for subsequent analysis. This analysis can be done through Grafana, as shown in the example on the slide.
- Prometheus is a widely used mechanism to collect and aggregate metrics. We can use JMX-to-Prometheus exporters alongside our ksqldb Servers, that provide the JMX metrics to the Prometheus server. The Prometheus server in turn can then be used as a data source for a Grafana based dashboard.

ksqldb Health Checks - REST Endpoint

```
$ http://ksqldb-server:8088/info
* Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8088 (#0)
> GET /info HTTP/1.1
> Host: localhost:8088
> User-Agent: curl/7.54.0
> Accept: /
>
< HTTP/1.1 200 OK
< Date: Fri, 23 Nov 2018 11:44:00 GMT
< Content-Type: application/vnd.ksql.v1+json
< Transfer-Encoding: chunked
< Server: Jetty(9.4.11.v20180605)
<
* Connection #0 to host localhost left intact
{"KsqlServerInfo":{"version":"5.1.2","kafkaClusterId":"8MvZ8QmPRYeluBTY0iZk_Q","ksqlServiceId":
```

- The ksqldb REST API supports a "server info" request at <http://<server>:8088/info>.

Troubleshooting RocksDB

Issue	Solution options
RocksDB performance appears low	<ul style="list-style-type: none">• Use SSDs• Add Cores
RocksDB's file sizes appear larger than expected	<ul style="list-style-type: none">• Use Linux du tool• Check for Write Amplification
Application memory utilization seems high	<ul style="list-style-type: none">• Mind number of stores in topology• Lower RocksDB memory usage

RocksDB is a high performance key value store that is used as default storage by Kafka Streams and by ksqlDB applications that are stateful. These are applications or queries that join streams and/or tables and group and aggregate values. The following issues related to RocksDB may occur.

- **The store/RocksDB performance appears low:** The workload might be IO bound. This happens especially when using a hard disk drive, instead of an **SSD**. However, if you already use SSDs, check your **client-side CPU utilization**. If it is very high, it is likely you may need more cores for higher performance.
- **RocksDB's file sizes appear larger than expected:** RocksDB tends to allocate sparse files, hence although the file size might appear large, the actual amount of storage consumed might be low. Check the real storage consumed (in Linux with the **du** command). If the amount of storage consumed is indeed higher than the amount of data written to RocksDB, then **write amplification** might be happening.
- **The app's memory utilization seems high:** If you have many stores in your topology, there is a **fixed per-store memory cost**. E.g., if RocksDB is your default store, it uses some off-heap memory per store. Either consider spreading your app instances on multiple machines or consider lowering RocksDb's memory usage using the **RocksDBConfigSetter** class.

	Write amplification is an undesirable phenomenon associated with flash memory and solid-state drives (SSDs) where the actual amount of information physically written to the storage media is a multiple of the logical amount intended to be written. More details see here: https://en.wikipedia.org/wiki/Write_amplification
---	---

Review



Question:

Why does a stateful Kafka Streams application need much more memory than a stateless application working on the same input topic?

A stateful application needs to store state from operations such as JOINs, windowing and aggregations locally. For this Rocks DB is used by default. For each such operation a Rocks DB instance is generated. The memory overhead is 50 MB per instance, plus the actual data produced by the stateful operation. For performance reason Kafka Streams tries to keep all the local state in memory.

Further Reading

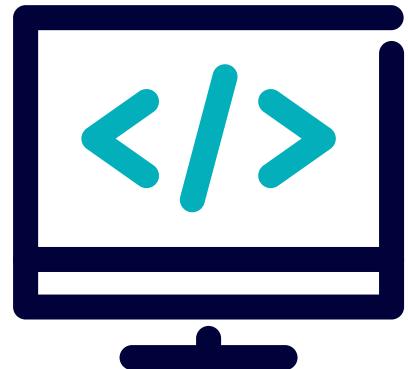
- Elastic Scaling of Your Application: <https://bit.ly/2l1hOVA>
- Memory Management: <https://bit.ly/2M1phXa>
- KSQL capacity planning: <https://bit.ly/2x6KnAC>
- Troubleshooting KSQL – Part 1: Why Isn't My KSQL Query Returning Data?:
<https://www.confluent.io/blog/troubleshooting-ksql-part-1>
- Troubleshooting KSQL – Part 2: What's Happening Under the Covers?:
<https://www.confluent.io/blog/troubleshooting-ksql-part-2>
- Data Wrangling with Apache Kafka and KSQL:
<https://www.confluent.io/blog/data-wrangling-apache-kafka-ksql>

hitesh@datacouch.io

Lab: Troubleshooting Kafka Streams & ksqlDB Apps

Please work on **Lab 12a: Troubleshooting Kafka Streams & ksqlDB Apps**

Refer to the Exercise Guide



hitesh@datacouch.io

b: Tuning Kafka Streams & ksqlDB Apps

Description

Streams apps scaling, memory management, fault tolerance. Best practices.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Configure standby ksqlDB and Kafka Streams instances for fast failover
- Only generate one single output per key, per window in Kafka Streams aggregate queries
- Identify bottlenecks in your ksqlDB queries and/or Kafka Streams apps
- Configure your ksqlDB cluster for production use

hitesh@datacouch.io

How many Application Instances?

- Number of instances <= number of topic partitions
 - Distribute & balance data (topics)
 - Distribute processing workload
-

For a Kafka Streams application the following rules should be considered for maximum parallelism and throughput:

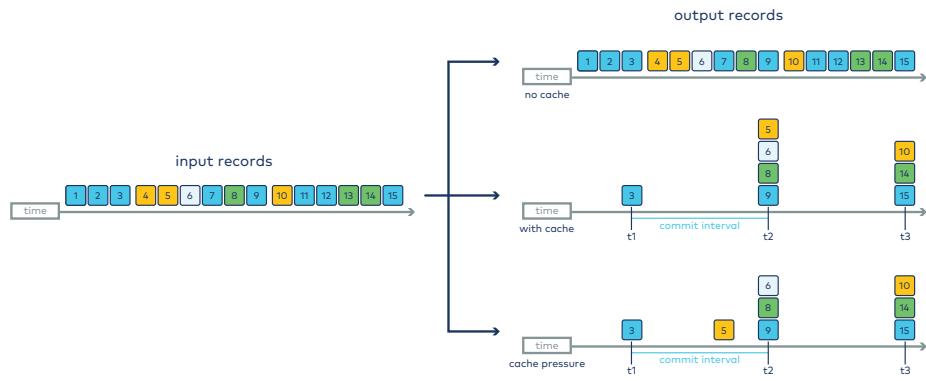
- The parallelism of a Kafka Streams application is primarily determined by how many partitions the input topics have. For example, if your application reads from a single topic that has ten partitions, then you can run up to ten instances of your applications. You can run further instances, but these will be idle.

The number of topic partitions is the upper limit for the parallelism of your Kafka Streams application and for the number of running instances of your application.

To achieve balanced workload processing across application instances and to prevent processing hotspots, you should distribute data and processing workloads:

- Data should be equally distributed across topic partitions. For example, if two topic partitions each have 1 million messages, this is better than a single partition with 2 million messages and none in the other.
- Processing workload should be equally distributed across topic partitions. For example, if the time to process messages varies widely, then it is better to spread the processing-intensive messages across partitions rather than storing these messages within the same partition.

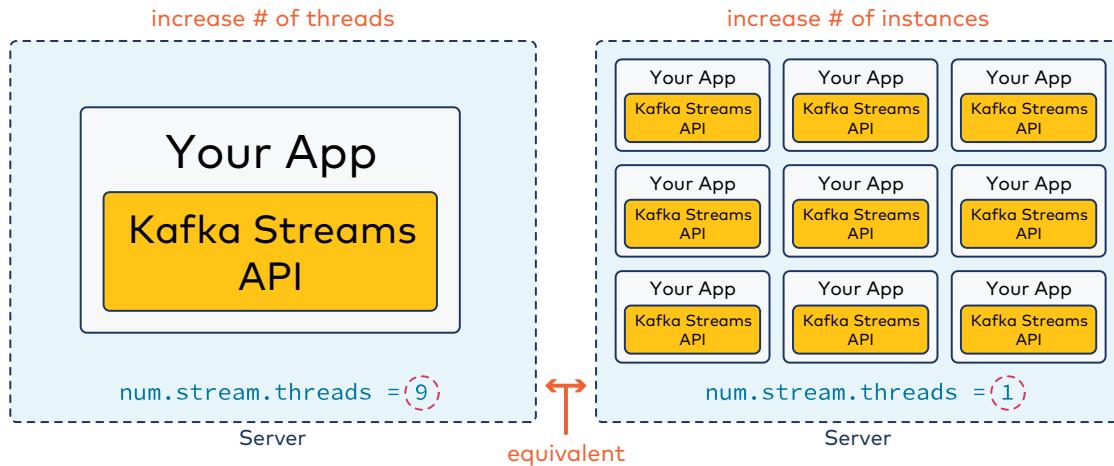
Memory Management



For **KTable** instances (as well as for the Processor API) we can specify the total memory (RAM) size of the record cache for an instance of the processing topology. For further details consult: <https://docs.confluent.io/currentstreams/developer-guide/memory-mgmt.html>

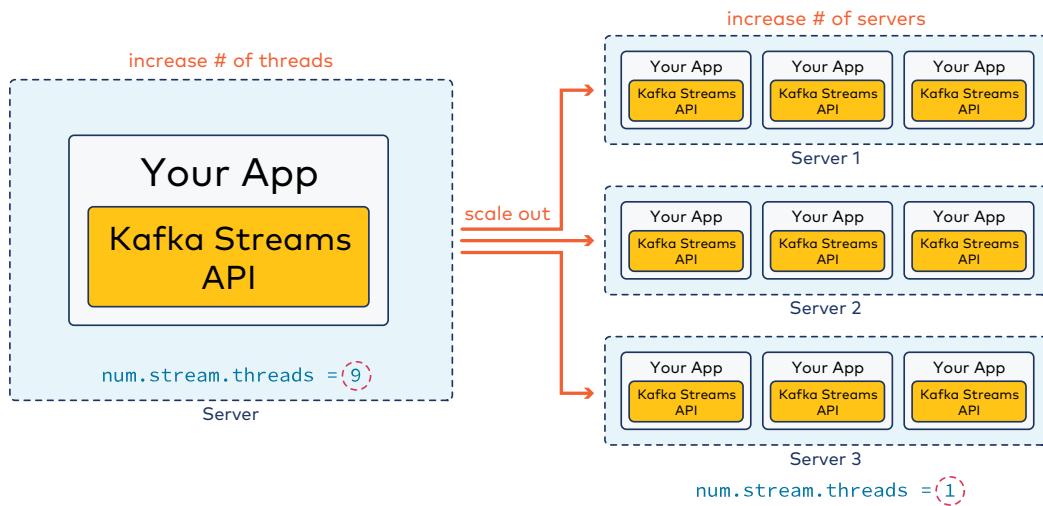
- When the cache is disabled, all of the input records will be output.
- When the cache is enabled:
 - Most records are output at the end of commit intervals (e.g., at t1 a single blue record is output, which is the final over-write of the blue key up to that time).
 - Some records are output because of cache pressure (i.e. before the end of a commit interval). For example, see the red record before t2. With smaller cache sizes we expect cache pressure to be the primary factor that dictates when records are output. With large cache sizes, the commit interval will be the primary factor. The total number of records output has been reduced from 15 to 8.

Task Placement - Scale Up



- **Task placement matters:** Increasing the number of partitions/tasks increases the potential for parallelism, but we must still decide where to place those tasks physically. There are two options: scale up, by putting all the tasks on a single server. This is useful when the app is CPU bound and one server has a lot of CPUs. You can do this by having an app with lots of threads (`num.stream.threads` config option, with a default of 1) or equivalently have clones of the app running on the same machine, each with 1 thread. There should not be any performance difference between the two.

Task Placement - Scale Out



- **Task placement matters:** The second option is to scale out, by spreading the tasks across more than one machine. This is useful when the app is network, memory or disk bound, or if a single server has a limited number of CPU cores.

Stateless versus Stateful

Stateless Applications:

- CPU & Network are key

Stateful Applications:

- **Memory:** High Performance
 - **Local Disks:** Fast Queries
 - **Kafka:** Fault Tolerance
 - **Standby Replicas:** High Availability
-

Stateless Kafka Streams Applications:

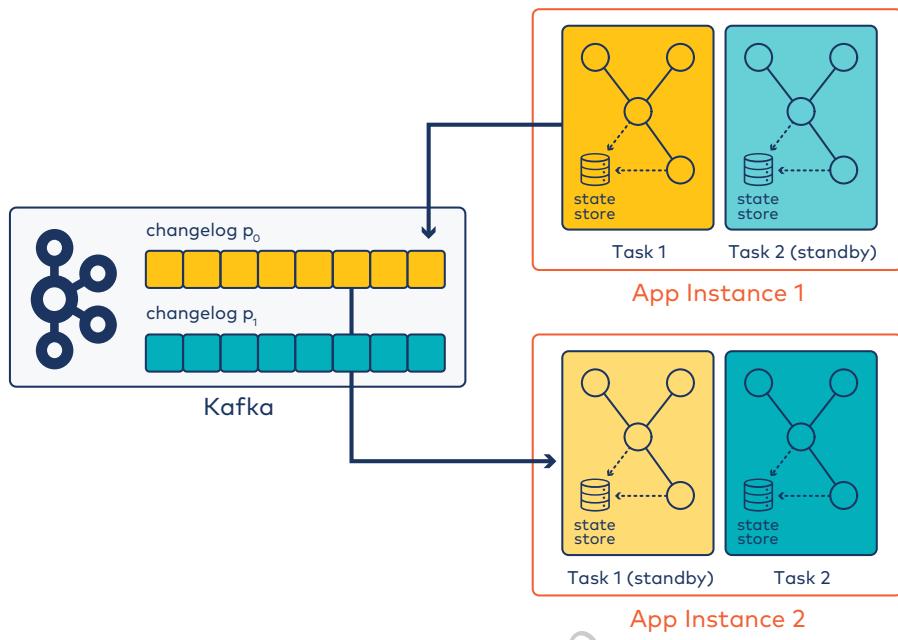
Here the CPU and network resources are key. These are applications that don't need to keep any state around. For example, they could be filtering, or performing some logic on the streaming data as it flows through the processor nodes, such as data conversion. They also might write their final output back to a Kafka topic.

Stateful Kafka Streams Applications:

Here we need to monitor another two resources, local disks and memory. These are applications that perform aggregates and joins.

- **Local storage space (for fast queries):** Kafka Streams uses embedded databases to store local data.
- **Global storage space (for fault tolerance):** In addition to writing locally, the data is also backed up to a changelog topics in Kafka for which log compaction is enabled, so that if the local state fails, the data is recoverable.
- **Global storage space (for internal operations):** Kafka Streams might store internal topics that are critical for its operations.
- **Memory (for performance):** each state store has some memory overhead for buffering.
- **Standby replicas (for high availability):** To improve availability during failure, Kafka Streams tasks can be replicated by setting `num.standby.replicas`

Standby Replicas



Standby Replicas

- Kafka Streams: `num.standby.replicas=1`
- ksqlDB Server: `ksql.streams.num.standby.replicas=1`



n **standby replicas** require **n+1** Kafka Streams instances or ksqlDB servers

-
- **Kafka Streams:** When a task is migrated, the task processing state is fully restored before the application instance resumes processing. This guarantees the correct processing results. In Kafka Streams, state restoration is usually done by replaying the corresponding changelog topic to reconstruct the state store. To minimize changelog-based restoration latency by using replicated local state stores, you can specify `num.standby.replicas`. When a stream task is initialized or re-initialized on the application instance, its state store is restored like this:
 - If no local state store exists, the changelog is replayed from the earliest to the current offset. This reconstructs the local state store to the most recent snapshot.
 - If a local state store exists, the changelog is replayed from the previously checkpointed offset. The changes are applied and the state is restored to the most recent snapshot. This method takes less time because it is applying a smaller portion of the changelog.
 - **ksqldb:** Bump the number of replicas for state storage for stateful operations like aggregations and joins. By having two replicas (one main and one standby) recovery from node failures is quicker since the state doesn't have to be rebuilt from scratch.

ksqlDB Best Practices & Patterns

- Test Queries before moving to Prod
 - Select the ksqlDB server mode based upon your requirements
 - Interactive mode
 - Headless mode
 - Avoid big multi-purpose cluster
 - Create ksqlDB cluster per App or per Team
-

These are a few best practices & patterns for ksqlDB Server that make operating and tuning easier:

- Always before moving queries to production we have to test them in a test environment. For this we can create a test or integration ksqlDB cluster in a test Kafka environment.
- ksqlDB server modes
 - Interactive mode - The REST interface is referred to as ksqlDB's "interactive" deployment mode. This mode would be appropriate where REST API requests like pull queries are needed.
 - Headless mode - With this mode, the REST interface is disabled and applications are submitted to the cluster with SQL files. This allows for more tight control over the behavior of the ksqlDB server cluster. This is often used in production for stream processing use cases where REST API requests like pull queries are not needed. A common example for a headless deployment would be for a streaming ETL application, where the query is known ahead of time and there is no need to access ksqlDB interactively.
- Once our ksqlDB queries are ready for production we can create a "*.sql" file containing the queries and then run the ksqlDB cluster in headless mode by instructing each node to load said file. This will automatically disable the REST API of the ksqlDB server such as that it cannot be remote accessed anymore.
- It is highly recommended to avoid creating one big ksqlDB cluster on which multiple ksqlDB applications (queries) are run. Why that? Well,
 - First, a big cluster is harder to monitor and it is more difficult to trace performance bottlenecks of each ksqlDB application
 - Secondly it is also harder to tune a big complex cluster
 - Lastly, one rogue ksqlDB application might starve all other apps

- Consider instead creating a ksqlDB cluster per ksqlDB application or per group of closely related applications

hitesh@datacouch.io

ksqldb Tuning

- Monitor **Consumer lag**
 - Decrease threads for cheap queries
 - Add resources to increase throughput
 - Add standby instances for faster fail-over
-

How to Know When to Scale?

- If ksqldb cannot keep up with the production rate of your Kafka topics, it will start to fall behind in processing the incoming data.
- Consumer lag is the Kafka terminology for describing how much a Kafka consumer including ksqldb has fallen behind.
- It's important to monitor consumer lag on your topics and add resources if you observe that the lag is growing.
- Confluent Control Center is the recommended tool for monitoring.

Mixed Workloads

- Your workload may involve multiple queries, perhaps with some feeding data into others in a streaming pipeline.
- Monitoring consumer lag of each query's input topic is especially important for such workloads.
- ksqldb currently does not have a mechanism to guarantee resource utilization fairness between queries.
- So a faster query like a project/filter may "starve" a more expensive query like a windowed aggregate if the production rate into the source topics is high.
- If this happens you will observe growing lag on the source topic for the more expensive queries and very low throughput to their sink topics.
- You can fix this situation by using either of these methods:
 - Tune the cheaper queries to consume less CPU by decreasing `kafka.streams.num.threads` for that query. In interactive mode you can use this command
`set kafka.streams.num.thread=<desired number of threads>` before each query. In headless mode you would add the same command before each query in the `*.sql` file (though note that according to our engineers this will only be

supported in a future version, and not 5.1.x)

- Add resources to reduce the per-CPU usage of the cheaper queries, which in turn will increase the throughput for the more expensive queries.
- ksqlDB server nodes can fail and if the failover time must be small then consider adding standby instances to the cluster. These standby instances always sync the state of the leader node and thus can take over immediately if the leader fails.

hitesh@datacouch.io

Recommended ksqlDB Production Settings

```
ksql.streams.producer.delivery.timeout.ms=2147483647      # Integer.MAX_VALUE
ksql.streams.producer.max.block.ms=9223372036854775807    # Long.MAX_VALUE
ksql.streams.replication.factor=3
ksql.streams.producer.acks=all
ksql.streams.topic.min.insync.replicas=2
ksql.streams.state.dir=/some/non-temporary-storage-path/
ksql.streams.num.standby.replicas=1
```

On this slide we have a list of recommended settings for ksqlDB when running in production:

- **Batch Expiry:** Set the batch expiry to Integer.MAX_VALUE to ensure that queries will not terminate if the underlying Kafka cluster is unavailable for a period of time.
- **Producer Blocking:** Set the maximum allowable time for the producer to block to Long.MAX_VALUE. This allows ksqlDB to pause processing if the underlying Kafka cluster is unavailable.
- **Kafka Streams Internal Topics:** Configure underlying Kafka Streams internal topics to achieve better fault tolerance and durability, even in the face of Kafka broker failures.
 - Highly recommended for mission critical applications.
 - Note that a value of 3 requires at least 3 brokers in your Kafka cluster.
- **Storage Directory:** Set the storage directory for stateful operations like aggregations and joins to be at a durable location. By default, they are stored in /tmp.
- **State Store Replicas:** Bump the number of replicas for state storage for stateful operations like aggregations and joins. By having two replicas (one main and one standby) recovery from node failures is quicker since the state doesn't need to be rebuilt from scratch. This configuration is also essential for pull queries to be highly available during node failures.

Review



Question:

What are the pros and cons of scaling up versus scaling out a Kafka Streams application?

Some possible answers...

Scaling up:

- Mostly beneficial on-premise where the procurement of a new server is costly and time consuming
- Beneficial when one has beefy servers/VMs available with loads of cores and RAM
- If scaling up too much network and disk IO quickly become a bottleneck

Scaling out:

- Best in the cloud where provisioning new instances takes seconds to low minutes
- more lightweight VMs are cheaper than equivalent massive VMs with many cores and loads of RAM
- Many instances can be better distributed (different racks and/or DCs) for higher availability

Further Reading

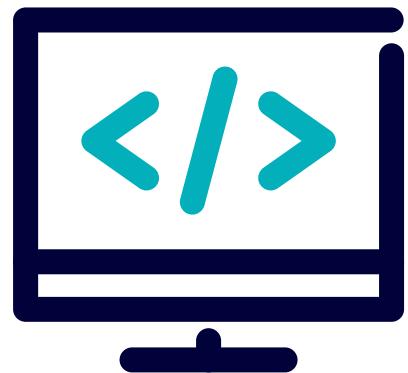
- Kafka Streams Architecture: <https://kafka.apache.org/21/documentationstreams/architecture>
- Achieving High Availability With Stateful Kafka Streams Applications: <https://tech.transferwise.com/achieving-high-availability-with-kafka-streams/>
- Troubleshooting KSQL – Part 1: Why Isn't My KSQL Query Returning Data?: <https://www.confluent.io/blog/troubleshooting-ksql-part-1>
- Capacity Planning & Sizing: <https://docs.confluent.io/currentstreams/sizing.html>
- KSQL Capacity Planning: <https://docs.confluent.io/currentksql/docs/capacity-planning.html>

hitesh@datacouch.io

Lab: Tuning Kafka Streams & ksqlDB Apps

Please work on **Lab 12b: Tuning Kafka Streams & ksqlDB Apps**

Refer to the Exercise Guide



hitesh@datacouch.io

13: Troubleshooting & Tuning Kafka Connect



CONFLUENT
Global Education

hitesh@datacouch

Module Overview



This module contains two lessons:

1. Troubleshooting Connect
2. Tuning Connect

hitesh@datacouch.io

a: Troubleshooting Kafka Connect

Description

Connect error handling framework. Other Connect issues.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Validate connector configuration
- Configure production ready error management
- Monitor your connect cluster for errors

hitesh@datacouch.io

Validating Connector Configuration

```
PUT /connector-plugins/hdfs-sink-connector/config/validate HTTP/1.1
Host: connect.example.com
Accept: application/json

{
  "connector.class": "io.confluent.connect.hdfs.HdfsSinkConnector",
  "tasks.max": "10",
  "topics": "test-topic",
  "hdfs.url": "hdfs://fakehost:9000",
  "hadoop.conf.dir": "/opt/hadoop/conf",
  "hadoop.home": "/opt/hadoop",
  "flush.size": "100",
  "rotate.interval.ms": "1000"
}
```

Look out for errors:

```
jq '.configs[]|select (.value.errors[]!=null) | .value'
```

This describes how to **validate connector configurations** using the Connect worker REST API in an efficient manner.

Background

Each Connect worker is capable of validating a configuration using the REST API by submitting a proposed JSON formatted configuration. The response back from the worker will be a full listing of the available configurations for the connector in addition to any errors that were found while validating the configuration. This is a useful operation to perform to prevent connector submission errors when trying to update or create a connector.

Performing validation

We can validate a connector configuration by providing the configuration as a **JSON** formatted file and issuing a **PUT** request to the worker endpoint **/connector-plugins/<name>/config/validate** where **<name>** is the fully qualified name of the connector plugin.

There is an example in the documentation: [https://docs.confluent.io/current/connect/references/restapi.html#put--connector-plugins-\(string-name\)-config-validate](https://docs.confluent.io/current/connect/references/restapi.html#put--connector-plugins-(string-name)-config-validate)



The configuration being submitted is the content of the **config** JSON map we would eventually use to create the connector, not the same content we would use to create the connector in full.

The response can be quite long and difficult to read, so we can use `jq` to parse the response for errors like:

```
jq '.configs[]|select (.value.errors[]!=null) | .value'
```

A video example of configuration validation is found here: <https://asciinema.org/a/MU2fdyQHJ3nKfYW6v1DLgkz1I>

hitesh@datacouch.io

Fail Fast Scenario

Why?	How?
<ul style="list-style-type: none">• Poisoned messages i.e., cannot be processed• Source/target system unavailable	<p>Configuration settings:</p> <pre># disable retries on failure (default 0) errors.retry.timeout=0 # do not log the error and their # contexts errors.log.enable=false # do not record errors in a # dead letter queue topic errors.deadletterqueue.topic.name="" # Fail on first error errors.tolerance=none</pre>

The fail-fast (i.e. non-managed errors) behavior can be achieved by disabling the error management framework, this can be done with the configuration shown on the right side of the slide. This is certainly not a good practice; if we are running an Apache Kafka version newer or equal than 2.0.0, it is always recommended to enable error management.

A situation that we may find is if a connector shows a status of **FAILED**, but we are not able to bring it back to a **RUNNING** state, even after multiple restart attempts.

We should work to remove the impediment that is causing this fast failure. Usual problems that could cause this situation are poisoned messages as well when the source or target systems are not available, or underperforming.

What are **Poisoned messages**?

These are messages that can not be processed and so are rejected. As the message was not successfully consumed, the connector will retry again and again.

The second situation is **source or target systems are unavailable**:

A situation that could arise from time to time in our deployment is when one of the systems the connectors are pulling data from or pushing data to becomes unavailable. This could be for multiple reasons, e.g., this target system is being taken offline for maintenance, or an upgrade, or simply because the target system is currently overloaded.

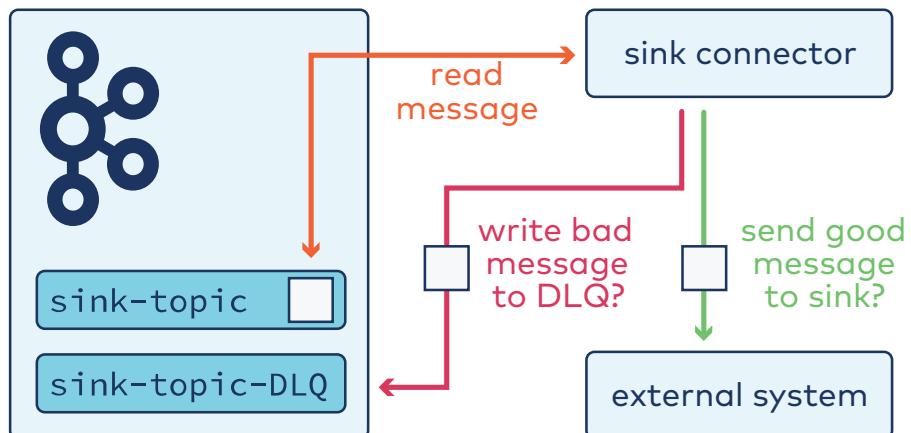
Each connector handles this slightly different. The JDBC connector handles connection attempts or operation retries, both with exponential back off for easy recovery. Other popular connectors such as the HDFS sink have an option to retry delivery of messages with exponential back off.

To plan for such scenarios, it is very important to check the different options present in the configuration of the connector and search for how each one is handling timeouts and reconnection attempts.

hitesh@datacouch.io

Dead Letter Queue

- **Problem:** Writing message to external system fails
- **Solution:**
 - Rather than giving up, produce this message to a special Kafka topic
→ Called the **dead letter queue (DLQ)**
 - Can inspect those messages separately and decide how to handle



 The DLQ is for sink connectors only.

More on DLQ:

- [Deep Dive on Connect Error Handling](#)
- [DLQ in Confluent Cloud](#)
- Retries, configurable via the settings on the slides to come, happen before a message that could not be written to the sink system is written to the DLQ.

Error Management Options

To configure error management, configure the **connector** settings:

Name	Default	Source Connectors ?	Sink Connectors ?
errors.retry.timeout	0	yes	yes
errors.retry.delay.max.ms	1 min	yes	yes
errors.tolerance	-	yes	yes
errors.deadletterqueue.topic.name	""	no	yes
errors.log.enable	false	yes	yes
errors.log.include.messages	false	yes	yes

The Connect error framework handles:

- **Retry on failure:** Which handles how an operation is retried after failing.
- **Task error tolerance:** How many errors to tolerate per task.
- **Dead letter queue:** For sink connectors, the original record (from the Kafka topic the sink connector is consuming from) which caused the failure will be written to another topic used as queue.

See next slide for an example configuration.

Recommended Error Management Config

Here's an example of configuring error management:

```
# retry for at most 10 minutes waiting up
# to 30 seconds between consecutive failures
errors.retry.timeout=600000
errors.retry.delay.max.ms=30000

# log error context along with application logs
# but do not include configs and messages
errors.log.enable=true
errors.log.include.messages=false

# produce error context into the Kafka topic
errors.deadletterqueue.topic.name=my-connector-errors

# Tolerate all errors.
errors.tolerance=all
```

hitesh@datacouch.io

Monitoring for Errors

Metric/Attribute Name	Description
<code>total-record-failures</code>	Total number of failures seen by this task.
<code>total-record-errors</code>	Total number of errors seen by this task.
<code>total-records-skipped</code>	Total number of records skipped by this task.
<code>total-retries</code>	Total number of retries made by this task.
<code>total-errors-logged</code>	The number of messages that was logged into either the dead letter queue or with Log4j.
<code>deadletterqueue-produce-requests</code>	Number of produce requests to the dead letter queue.
<code>deadletterqueue-produce-failures</code>	Number of records which failed to produce correctly to the dead letter queue.
<code>last-error-timestamp</code>	The timestamp when the last error occurred in this task.

The MBean to look for is:

```
kafka.connect:type=task-error-metrics,connector=([-.\w]+),task=(-.\w)+
```

A number of additional metrics are also available to inspect the error management, and since Apache Kafka 2.0 we can check for the list shown on the slide.

Monitoring Workers and Hosts

- Monitor Workers with REST interface
- Monitor Hosts where Workers run
 - CPU utilization
 - Garbage collection pause duration
 - Heap usage
 - Physical memory usage

Monitoring Workers with the REST interface

The REST interface on any worker in a distributed worker cluster can be used to poll for connector and task status. Any **FAILED** task should be investigated further.

Monitoring Hosts where Workers Run

Host and JVM level metrics are useful to understand the environment in which Connect worker(s) run. Specifically, the following metrics should be used to monitor divergence from steady-state behavior:

- CPU utilization
- Garbage collection pause duration
- Heap usage
- Physical memory usage



failed tasks are not automatically recovered by Connect, and require manual intervention. Automatic fault tolerance is only at the worker level.

Add Connector Context to Worker Logs

- Include `%X{connector.context}` in the log4j layout
 - Adds connector-specific and task-specific information to the log message
 - Makes it easier to identify log messages that apply to a specific connector
- To add this parameter, update the log layout configuration as follows:

```
#log4j.appender.stdout.layout.ConversionPattern=[%d] %p %X{connector.context}%m  
(%c:%L)%n
```

Before:

```
[2020-05-18 12:23:47,987] INFO Started JDBC source task  
(io.confluent.connect.jdbc.source.JdbcSourceTask:257)
```

After:

```
[2020-05-18 12:23:47,987] INFO [Credits-and-Grants-Connector|task-0] Started JDBC  
source task (io.confluent.connect.jdbc.source.JdbcSourceTask:257)
```

For more info on this option, see KIP-449: <https://cwiki.apache.org/confluence/display/KAFKA/KIP-449%3A+Add+connector+contexts+to+Connect+worker+logs>

Dynamically Adjust Connect Worker Log Levels

The Connect Worker `/admin/logger`s endpoint supports the following operations:

- Get a list of all named loggers
- Get the log level of a specific logger
- Set the log level of a specific logger

Log level modifications:

- will not be persisted across worker restarts
- will only affect the worker whose endpoint received this REST request

For more info on this option, see KIP-495: <https://cwiki.apache.org/confluence/display/KAFKA/KIP-495%3A+Dynamically+Adjust+Log+Levels+in+Connect>

Dynamically Adjust Connect Log Levels - Examples

- Get a list of all named loggers

```
curl -s http://connect:8083/admin/loggers/ | jq
{
  "root": {
    "level": "INFO"
  }
}
```

- Set the log level of a specific logger

```
curl -s -X PUT -H "Content-Type:application/json" \
> http://connect:8083/admin/loggers/org.apache.kafka.connect.runtime.WorkerSourceTask
\ -d '{"level": "TRACE"}' | jq '.'
[ "org.apache.kafka.connect.runtime.WorkerSourceTask"
]
```

Review



Question:

How can you best discover that your Kafka Connect connectors are having an issue?

Possible answer: by monitoring the relevant Connect specific JMX metrics such as `total-record-errors` or infrastructure metrics such as CPU and IO bandwidth utilization. Also do not forget to parse the logs of Kafka Connect for errors.

hitesh@datacouch.io

Further Reading

- KIP-298: Error handling in Connect:
<https://cnfl.io/kip-298>
- Connect Concepts:
<https://docs.confluent.io/current/connect/concepts.html>
- Connect Worker Configs:
<https://docs.confluent.io/current/connect/references/allconfigs.html>
- Install Connector Manually:
<https://docs.confluent.io/current/connect/managing/install.html#install-connector-manually>

hitesh@datacouch.io

b: Tuning Kafka Connect

Description

Connect best practices.

hitesh@datacouch.io

Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Reduce the amount of imported data to the absolute minimum required
- Filter and transform with the source and/or sink connector
- Optimally configure the number of connect workers and tasks

hitesh@datacouch.io

Monitoring Connect

- Monitoring helps in sizing & troubleshooting
- Workers have embedded producer and/or consumer

Metrics	MBean Name
Connector Metrics	<code>kafka.connect:type=connector-metrics,connector=<connector-ID></code>
Task Metrics (Common, Source & Sink)	<code>kafka.connect:type=task-metrics,connector=<connector-ID>,task=<task-ID></code> <code>kafka.connect:type=source-task-metrics,connector=<connector-ID>,task=<task-ID></code> <code>kafka.connect:type=sink-task-metrics,connector=<connector-ID>,task=<task-ID></code>
Worker Metrics	<code>kafka.connect:type=connect-worker-metrics</code>
Worker Rebalance Metrics	<code>kafka.connect:type=connect-worker-rebalance-metrics</code>

Proper monitoring of Connect clients helps facilitate sizing and scalability efforts along with exposing useful information for troubleshooting issues.

Workers have embedded producer and/or consumer instances that should be monitored like every other producer or consumer client. Useful JMX metrics to monitor are:

- Common metrics for all clients
- Producer metrics
- Consumer metrics

Because workers will launch consumers and/or producers, all metrics above should be monitored for each worker.

Filter and Transform Data

- Create/extract key from data field
- Add metadata to Kafka message
- Field Masking & whitelist/blacklist fields
- Route messages with
 - Regular expressions
 - Timestamp



Can chain many transformations!

The connect framework offers plenty of possibilities to filter, transform and route messages without having to write a single line of code and by simply adding some configuration items when defining a sink or source connector.

Sample:

A) Create a message Key

To go from this situation (output of AVRO console consumer showing imported data):

```
null
{"c1":{"int":101}, "c2":{"string":"foo"}, "create_ts":1501796305000, "update_ts":150179630
5000}
null
{"c1":{"int":102}, "c2":{"string":"foo"}, "create_ts":1501796665000, "update_ts":150179666
5000}
```

to this:

```
101
{"c1":{"int":101}, "c2":{"string":"foo"}, "create_ts":1501796305000, "update_ts":150179630
5000}
...
```

Filter and Transform Data

we can use this:

```
{  
  "name": "jdbc_source_mysql_foobar_01",  
  "config": {  
    ...  
    "_comment": "---- Single Message Transforms ----",  
    "transforms": "createKey,extractInt",  
    "transforms.createKey.type": "org.apache.kafka.connect.transforms.ValueToKey",  
    "transforms.createKey.fields": "c1",  
  
    "transforms.extractInt.type": "org.apache.kafka.connect.transforms.ExtractField$Key",  
      "transforms.extractInt.field": "c1"  
  }  
}
```

B) Reduce amount of data

Use the whitelist/blacklist feature to remove unwanted data from the source. E.g. to remove field "c2" from source use:

```
"transforms": "dropFieldC2",  
"transforms.dropFieldC2.type": "org.apache.kafka.connect.transforms.ReplaceField$Value",  
"transforms.dropFieldC2.blacklist": "c2"
```

C) Route messages

The following could be used e.g. for an Elasticsearch sink connector to

1. remove the prefix "mysq-" from the topic name and then
2. append a timestamp to the remaining topic name

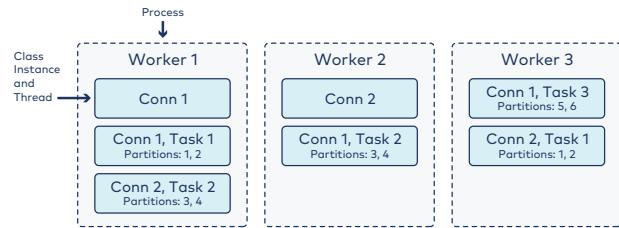
The resulting topic name will be used to define the index name in Elasticsearch. Thus in this sample all messages of the same month go into a dedicated index.

Example: from "mysql-foo" to "foo-201901"

```
"transforms": "dropPrefix,routeTS",  
"transforms.dropPrefix.type": "org.apache.kafka.connect.transforms.RegexRouter",  
"transforms.dropPrefix.regex": "mysql-(.*)",  
"transforms.dropPrefix.replacement": "$1",  
"transforms.routeTS.type": "org.apache.kafka.connect.transforms.TimestampRouter",  
"transforms.routeTS.topic.format": "kafka-${topic}-${timestamp}",  
"transforms.routeTS.timestamp.format": "YYYYMM"
```

Scale your Connect cluster - Distributed Workers

- Scalability & fault tolerance
- All workers have same `group.id`
- Workers coordinate to distribute connectors and tasks across all worker instances
- Workload is auto-rebalanced upon failure



Distributed mode provides scalability and automatic fault tolerance for Kafka Connect. In distributed mode, you start many worker processes using the same `group.id` and they automatically coordinate to schedule execution of connectors and tasks across all available workers. If you add a worker, shut down a worker, or a worker fails unexpectedly, the rest of the workers detect this and automatically coordinate to redistribute connectors and tasks across the updated set of available workers. Note the similarity to consumer group rebalance. Under the covers, connect workers are using consumer groups to coordinate and rebalance.

The graphic shows a three-node Kafka Connect distributed mode cluster. Connectors (monitoring the source or sink system for changes that require reconfiguring tasks) and tasks (copying a subset of a connector's data) are automatically balanced across the active workers. The division of work between tasks is shown by the partitions that each task is assigned.



Fault tolerance and auto-rebalancing only happens with **worker** failure. Task failure must be handled manually. We can use the REST APIs to retrieve the task status: <https://cnfl.io/connect-rest-api>

We can also use the metrics instead, which report task status:

`kafka.connect:type=task-metrics,connector=<connector-ID>,task=<task-ID>`

Review



Question:

You want to import a huge table from your MySQL DB to Kafka. The table has about 2 dozen fields and you only really need about 5 of them. How do you proceed?

There are at least two ways of doing so:

- In the DB define a view that only queries the said 5 fields
- Or define a query in the configuration of the source connector

Further Reading

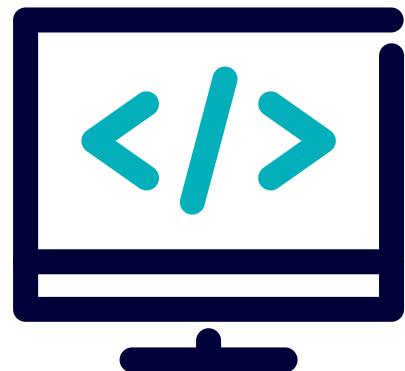
- The Simplest Useful Kafka Connect Data Pipeline In The World ... or thereabouts—Part 1-3:
<https://cnfl.io/connect-pipeline-1>
<https://cnfl.io/connect-pipeline-2>
<https://cnfl.io/connect-pipeline-3>
- Connect transformations:
<https://docs.confluent.io/current/connect/transforms/index.html>

hitesh@datacouch.io

Lab: Tuning Kafka Connect

Please work on **Lab 23a: Tuning Kafka Connect**

Refer to the Exercise Guide



hitesh@datacouch.io

Conclusion



**CONFLUENT
Global Education**

hitesh@datacouch.io

Course Contents



Now that you have completed this course, you should have the skills to:

- Formulate the Apache Kafka® Confluent Platform specific needs of your company
- Monitor all essential aspects of your Confluent Platform
- Tune the Confluent Platform according to your specific needs
- Provide first level production support for your Confluent Platform

hitesh@datacouch.io

Other Confluent Training Courses

- Confluent Developer Skills for Building Apache Kafka®
- Confluent Stream Processing Using Apache Kafka® Streams & ksqlDB
- Apache Kafka® Administration By Confluent
- Managing Data in Motion with Confluent Cloud



For more details, see <https://confluent.io/training>

-
- **Confluent Stream Processing Using Apache Kafka® Streams & ksqlDB** covers:
 - Identify common patterns and use cases for real-time stream processing
 - Understand the high level architecture of Kafka Streams
 - Write real-time applications with the Kafka Streams API to filter, transform, enrich, aggregate, and join data streams
 - Describe how ksqlDB combines the elastic, fault-tolerant, high-performance stream processing capabilities of Kafka Streams with the simplicity of a SQL-like syntax
 - Author ksqlDB queries that showcase its balance of power and simplicity
 - Test, secure, deploy, and monitor Kafka Streams applications and ksqlDB queries
 - **Apache Kafka® Administration by Confluent** 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 Advanced Skills for Optimizing Apache Kafka**

- Formulate the Apache Kafka® Confluent Platform specific needs of your organization
- Monitor all essential aspects of your Confluent Platform
- Tune the Confluent Platform according to your specific needs
- Provide first level production support for your Confluent Platform

hitesh@datacouch.io

Confluent Certified Developer for Apache Kafka

Duration: 90 minutes

Qualifications: Solid understanding of Apache Kafka and 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



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
 - Multiple choice questions
 - 90 minutes
 - Designed to validate professionals with a minimum of 6-to-9 months hands-on experience
 - Remotely proctored on your computer
 - Available globally in English

Confluent Certified Administrator 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 per day!

Cost: \$150

Register online: www.confluent.io/certification



This course prepares you to manage a production-level Kafka environment, but does not guarantee success on the Confluent Certified Administrator Certification exam. We recommend running Kafka in Production for a few months and studying these materials thoroughly before attempting the exam.

Benefits:

- Recognition for your Confluent skills with an official credential
- Digital certificate and use of the official Confluent Certified Administrator Associate logo

Exam Details:

- The exam is linked to the current Confluent Platform version
- Multiple choice and multiple select questions
- 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

We Appreciate Your Feedback!



Please complete the course survey now.

Your instructor will give you details on how to access the survey

hitesh@datacouch.io

Thank You!

hitesh@datacouch.io

Appendix: Additional Problems



CONFLUENT
Global Education

hitesh@datacouch.io

Overview

This section contains a few additional problems to reinforce concepts in this course. These problems were originally written as warm-up problems for instructor-led training for this course. Your instructor may or may not choose to incorporate some or all of these problems in class; you may find them to provide additional enrichment in any case. Some other problems originally created as warm-up problems have been adapted into activities in the content of this version of this course.

hitesh@datacouch.io

Problem A: Reviewing Message Sending and Broker Arrival

Suppose you have a new producer. Suppose, further, `linger.ms` has been increased from its default to 5 minutes.

1. You send a message and it gets partitioned to partition p_5 . What all happens from the moment that assignment is determined until the message makes it to the page cache of the broker containing p_5 ?
2. Suppose, now, an overall total of 31.9 MB of messages has been produced by this producer and assigned to various partitions. No broker yet knows anything about any of these messages. You send a message of size 0.5 MB. Does it fail? Explain why or why not.

Prerequisite Modules:

- Admin class

Problem B: Replication Review

Suppose you have 5 brokers. We will concern ourselves with a partition p_1 .

1. Suppose replication factor for p_1 is 3. Illustrate a possible scenario.
2. Suppose replication factor for p_1 is instead 7. Illustrate a possible scenario.
3. Finally, regardless of anything in the previous parts, we have the following situation for our replicas:

$p_{1,L}$ contains messages m_0 , m_1 , and m_2 and the last write went to this replica first

$p_{1,F0}$ contains messages m_0 and m_1

$p_{1,F1}$ contains messages m_0 , m_1 , and m_2

Suppose the broker containing $p_{1,L}$ fails. What will happen? Explain.

Prerequisite Modules:

- Admin class

Problem C: Consumer Offsets and Consumer Lag

Suppose partition p_0 has messages at offsets 0 through 20 and consumer c_3 is assigned to p_0 . Say you are monitoring this consumer using CCC or the CLI.

- a. What are valid values of c_3 's consumer offset in p_0 ?
- b. What would be reported as the "log end offset" for this consumer and partition?
- c. Suppose the offset is 12. What is the consumer lag?
- d. Where would you go in CCC to find lag? What would you look for as evidence that lag is increasing?
- e. Suppose your consumer application is reading messages containing mobile food orders and printing order slips for a restaurant's kitchen. How would you interpret your answer to (c)? What would we ideally want lag to be in this case?
- f. Give a different use case where we wouldn't care about lag so much?

Prerequisite Modules:

- M2 (Generic Monitoring)
- M3 (Monitoring with Confluent Control Center)

Problem D: Monitoring Disk Usage

Say you're doing the right thing and monitoring broker disk usage.

- a. You notice a disk on broker b_{101} is at 95% capacity. You decide that might be dangerous and decide to do something about that. Is this good? What could be problematic?
 - b. How can you prevent such issues? What's Confluent's recommended best practice?
-

Prerequisite Modules:

- M2 (Generic Monitoring)

hitesh@datacouch.io

Problem E: Assessing Discrepancies in Settings

- A colleague insists he changed how long a log segment could be the active log segment before it rolls to a max of 2 hours.
- But another colleague is reporting that she has seen some log segments for topic t_7 , partition p_{12} on broker 103 have been the active log segment with timestamps 4 and 5 hours in the past.

Another colleague wants answers and explanations. What do you tell them?

Prerequisite Modules:

- Admin class

Problem F: Troubleshooting Producers and Consumers

Question 1

How do you make sure that all messages produced are actually received by Kafka? (Hint: think about some important settings to inspect/set.)

Question 2

You realized that one of your consumer instances in consumer group **Foo** processed the data from partitions 1, 4, and 7 of **topic-1** incorrectly. You realize this is unacceptable...

- Conceptually, what should you do and why?
- Can you tell how to do this with a command or code, either at a high level or specifically?

Question 3

After noticing problems with consumption, you find out that some consumers get significantly more messages than others. This behavior is not temporary; there is a consistent imbalance, even after rebalancing. Some consumers go long periods of time consuming zero messages.

- What could be some causes of this problem?
- What are some possible solutions?

Prerequisite Modules:

- 3a (Monitoring with Confluent Control Center)
- 4a (Troubleshooting Intro)
- 10a (Troubleshooting Producers)
- 11a (Troubleshooting Consumers)
- 11b (Tuning Consumers)