Confluent Advanced Skills for Optimizing Apache Kafka®

Instructor Guide

Version 6.0.0-v2.0.1

Table of Contents

Instructor Guide Information

CONFIDENTIAL: Only for use by Confluent staff and Confluent Authorized Training Partners. DO NOT DUPLICATE.

Suggested Pacing - Day 1

#	Title	Content	Start	End	Duration (minutes)
-	Introduction	Slides	9:00	9:15	15
-	Warm-Up	Discussion	9:15	10:05	40
Break			10:05	10:15	10
1	Business Needs and SLAs	Slides	10:00	10:15	15
2	Monitoring Basics	Slides	10:15	10:30	15
3	Generic Monitoring	Slides	10:30	10:45	15
3	Generic Monitoring	Labs	10:45	11:45	60
Break			11:45	12:15	30
Lunch			12:15	13:15	60
4	Monitoring with C3	Slides	13:15	13:30	15
4	Monitoring with C3	Lab	13:30	14:00	30
5a	Troubleshooting Intro	Slides	14:00	14:15	15
5b	Production Down & TS Strategies	Slides	14:15	15:30	15
5c	TS Toolbelt	Slides	15:30	15:40	10
5	TS Toolbelt	Lab	15:40	16:10	30
6	Where are my System Log Files	Slides	16:10	16:25	15
6	Where are my System Log Files	Lab	16:25	16:55	30

Suggested Pacing - Day 2

#	Title	Content	Start	End	Duration (minutes)
Review Day 1/Warm Up			9:00	9:45	45
Break			9:45	10:00	15
7	TS ZooKeeper	Slides	10:00	10:20	20
7	TS ZooKeeper	Lab	10:20	10:50	30
Break			10:50	11:20	30
8ab	TS Brokers	Slides	11:20	12:00	40
Lunch			12:00	13:00	60
8	TS Brokers	Lab	13:00	14:00	60
8c-8g	TU Brokers	Slides	14:00	14:30	30
8	TU Brokers	Lab	14:30	15:30	60
9	TS Schema Registry	Slides	15:30	15:45	15
9	TS Schema Registry	Lab	15:45	16:15	30

Suggested Pacing - Day 3

#	Title	Content	Start	End	Duration (minutes)
Review Days 1 and 2/Warm Up			9:00	9:55	45
Break			9:45	10:00	15
10a	TS Producers	Slides	10:00	10:15	15
10a	TS Producers	Lab	10:15	10:45	30
10b	TU Producers	Slides	10:45	11:00	15
10bc	TU Producers	Lab	11:00	11:30	30
Lunch			11:30	12:30	60
11a	TS Consumers	Slides	12:30	12:45	15
11a	TS Consumers	Lab	12:45	13:00	30
11b	TU Consumers	Slides	13:15	13:30	15
11b	TU Consumers	Lab	13:30	14:00	30
Break			14:00	14:15	15
12a	TS Streams Apps	Slides	14:15	14:30	15
12a	TS Streams Apps	Lab	14:30	14:45	30
12b	TU Streams Apps	Slides	14:45	15:00	15
12b	TU Streams Apps	Lab	15:00	15:30	30
Break			15:30	15:45	15
13a	TS Connect	Slides	15:45	16:00	15
13b	TU Connect	Slides	16:00	16:15	15
13a	TU Connect	Lab	16:15	16:45	30
	Conclusion	Slides	16:45	17:00	15

Alternative: Rather than allocating as much time for review and warm-up in the morning, allocate less time in the morning for review and warm-up and instead allocate review time after lunch to discuss Problem F. (Problem F requires having done the Producers and Consumers modules.)

Troubleshooting

Docker

- Most Docker related issues in the hands-on labs can be fixed with bringing the cluster down and then restarting the lab:

```
$ cd ~/confluent-cao
$ docker-compose down
$ docker-compose up
```

- Confluent Control Center may take some time to reflect an action taken on the broker. Please be patient
- If you encounter any catastrophic issues, remove all containers and prune the system with:

```
$ ~/docker-nuke.sh
```

- Here is a Docker troubleshooting cheatsheet: https://cnfl.io/docker-cheatsheet

Version History

Revision	Date
6.0.0-v2.0.1	2022-December
6.0.0-v2.0.0	2022-October
6.0.0-v1.0.0	2020-November
5.5.0-v1.1.0	2020-July
5.5.0-v1.0.1	2020-July-17
5.5.0-v1.0.0	2020-May-20
5.1.2-v1.1.0	2020-April-2
5.1.2-v1.0.1	2019-July-29
5.1.2-v1.0.0	2019-March-29 (GA for Kafka Summit NYC)
5.1.2-v0.7.0	2019-March-26 (pre-release)
5.1.2-v0.6.0	2019-March-21 (pre-release)
5.1.2-v0.5.0	2019-March-9 (pre-release)
5.1.1-v0.4.0	2019-Feb-28 (pre-release)
5.1.0-v0.3.0	2019-February (internal beta)

Summary of Changes

6.0.0-v2.0.1

- Adjusted Branch Overview pages for first two branches to correct moved module
- Generic Monitoring module, added slides Kinds of JMX Metrics and More on Meter Metrics
- Broker tuning module, request life cycle lesson, added slides Monitoring Requests on the Broker and Configuring Requests on the Broker and removed older, mis-titled related slide. (These graphics are the same as are added in the 6.0.0-v2.0.1 release of ADM.)
- Lab slide for Troubleshooting Producers, added fix command that may be necessary for some.
- Connect troubleshooting lesson, Error Management Options, added `errors.retry.delay.max.ms` to table.
- Changed `kafkacat` to `kcat` in command samples in a few places.
- Updated some configuration default values in various places.
- Problem F, added and tweaked Instructor Notes

6.0.0-v2.0.0

This is meant to be a "minor" edit of the course - a reorganization for streamlining, updating graphics, and including problems. Labs have not changed. The course is not expected to be delivered in this form many times; a larger-scale update coinciding with the ZooKeeper removal is anticipated.

Note that the overall technical content and objectives have not changed; the organization has. Most slide text is the same as it was.

- Reorganized contents
  - Grouped related contents on troubleshooting and tuning the same component into same module
    - Retained troubleshooting vs. tuning distinctions as lessons
    - Broke large modules up into lessons
  - Combined several prior short modules into one module on troubleshooting basics, but made those modules into lessons
  - Brought pillars of optimization tuning intro to be with general troubleshooting

    - Now four major branches — see table of contents
- Brought course into new format to match other courses that have been redone in look and feel, as well as future editing and modularization ability
- Graphics
    - Replaced assorted graphics with newer ones
    - Removed some graphics that did not add technical value
- Added warm up problems appendix
- Added slides distinguishing SLIs, SLOs, and SLAs
- Removed old modules on troubleshooting resource problems and post mortem analysis
- Removed content on TS Brokers slide "Add More Storage to Broker" about manually moving partitions
- Removed content on `suppress` is TU Streams lesson

6.0.0-v1.0.0

The primary motivation for this release was to update the course for CP 6.0 and AK 2.6.

Lab Environment:

- Updated to use the `6.0.0-1-ubi8` docker containers
- Added the following environment settings to the Kafka broker containers to prevent disk full errors:

  KAFKA_LOG_RETENTION_HOURS: 1

  KAFKA_LOG_SEGMENT_BYTES: 536870912

  KAFKA_LOG_RETENTION_BYTES: 536870912

- Added the following environment setting to the Control Center container to prevent surprise UI changes:

  CONTROL_CENTER_UI_AUTOUPDATE_ENABLE: "false"

- Updated config files to include `confluent.monitoring.interceptor.bootstrap.servers` property
- Lab image now includes the required `hosts` entries so the step that had students run `update-hosts.sh` was removed
- `docker-nuke.sh` is moved from the `training` user home folder to the path so it can now

be run as `docker-nuke.sh` from any terminal window

Labs

- General Updates
  - New content added to the beginning of the exercise guide that reminds students to use the html exercise guide located on the lab virtual machine desktop and also provides info on how to copy and paste content from the exercise guide into the terminal window
  - Updated `CREATE STREAM` and `CREATE TABLE` commands so that they use `KEY` and `PRIMARY KEY` respectively instead of the previous `key=`
- Lab 08.a. Where are my log files?
  - Added steps to install the JDBC connector since it is no longer included with the Apache Kafka distribution
- Lab 13.a. Troubleshooting Consumers
  - Replaced the datagen connector with the `ksql-datagen` tool
- Lab 14.a. Troubleshooting ksqlDB Apps
  - Replaced the datagen connector with the `ksql-datagen` tool
- Lab 22.a. Tuning Kafka Streams Apps
  - Replaced the datagen connector with the `ksql-datagen` tool

Modules

- Module 10 Troubleshooting Brokers
  - Added new broker metrics to the slide that were added by KIP-551

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

    `kafka.server:type=KafkaServer,name=TotalDiskWriteBytes`
- Module 15 Troubleshooting Kafka Connect
  - Added note to slide **Error Management (1)** about KIP-610: Error Reporting in Sink Connectors

5.5.0-v1.1.0

Switch to new Confluent built lab platform.

5.5.0-v1.0.1

- Exercise Guide
  - Corrected issues identified by first teach of CAO 5.5.0-v1.0.0
  - Removed exercise **20.a. Choosing the throttling value for replication**

5.5.0-v1.0.0

All materials

- Replaced `--zookeeper` with `--bootstrap-server` for commands as appropriate in the modules and exercises.
- Updated C3 screenshots to CP 5.5

Slides, IG, SH

- Extensive update to Module 04 Monitoring with Confluent Control Center slides and notes that cover the C3 views due to the major changes to C3 from CP 5.1.2 to CP 5.5
- Updated and relocated the Schema Registry overview slide in Module 11 Troubleshooting the Schema Registry so that it includes mention of the three currently supported schema protocols; Avro, JSON, and Protobuf.
- Added the following slides to Module 15 Troubleshooting Kafka Connect
  - Add Connector Context to Worker Logs
  - Dynamically Adjust Connect Log Levels
- Updated the recommended configuration settings slides in Module 19 Tuning Producers and Module 21 Tuning Consumers

Exercise Guide

- Added a step to run `update-hosts.sh` at the beginning of the exercise guide to add `hosts` file entries for all docker containers used in the exercises. This is needed due to the elimination of the DNS service from the lab environment.
- Updated C# apps in the monitoring labs to use the 1.4.2 .NET client
- Removed backslashes between lines in multi-line statements in the ksqlDB labs since they are no longer required
- The majority of kafka utility related commands are now run from the VM rather than the base container

- Zookeeper (12181, 22181, 32181) and Kafka (19092, 29092, 39092) instances are now assigned a unique client port to accomodate the use of `hosts` file entries versus DNS service
- The instructions to run the class in a Docker for Desktop environment were removed due to the complex nature of how exercise steps interact with CAO lab environment

Lab Image

- All 25 docker-compose.yml files were updated to be compatible with the lab environment use of hosts file entries versus DNS service

5.1.2-v1.1.0

- Updated branding
- Updated aspect ratio from 4x3 to 16x9

5.1.2-v1.0.1

- Added missing lab **Troubleshooting Consumers**
- Used spell checker to correct all typos in modules and labs
- Fixed issue-243: Added missing Dockerfile and corrected formatting of date in lab

Introduction

CONFLUENT
Global Education

Class Logistics and Overview

Copyright & Trademarks

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.

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 prerequiste.

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

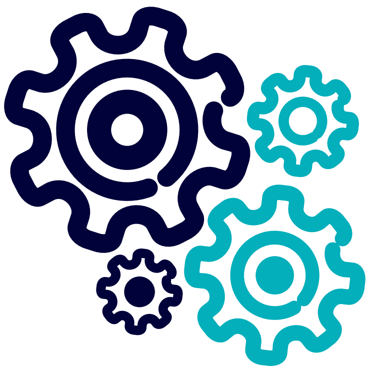
Course Objectives

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

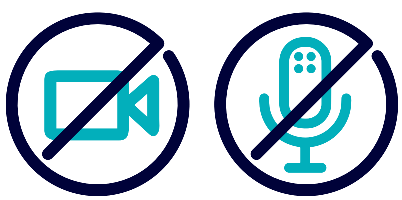
- Formulate the Apache Kafka® Confluent Platform specific needs of your company
- Monitor all essential aspects of your Confluent Platform
- Tune the Confluent Platform according to your specific needs
- Provide first level production support for your Confluent Platform

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

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

Instructor note: Balancing timing can be tricky. It might be wise to give a time "limit" (like 60 seconds) before you open the floor for introductions so this doesn't take too long.

This activity has a few purposes:

- Gather information about the experience levels and professional perspectives in the room
- Invite students to express themselves
- Encourage students to listen to each other

Consider starting by asking all students to stand (or otherwise indicate the "on" position of a binary, like raising a hand, if not able to stand). Ask students to sit if they meet the following criteria:

- You are new to Kafka
- You have worked with Kafka for less than a year
- You have worked with Kafka for between 1 and 2 years
- You have worked with Kafka for between 2 and 4 years
- You have worked with Kafka for more than 4 years

This gives you feedback about experience in the room while also involving students in a physical activity together. The last people standing can serve as assets in the classroom over the course of the class.

The next questions can be answered by individual students, giving them a chance to express themselves and understand each other.

Branch 1: Monitoring - Overview

CONFLUENT
Global Education

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

01: Business Needs and SLAs

CONFLUENT
Global Education

Module Overview

This module contains one lesson:

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

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

Description

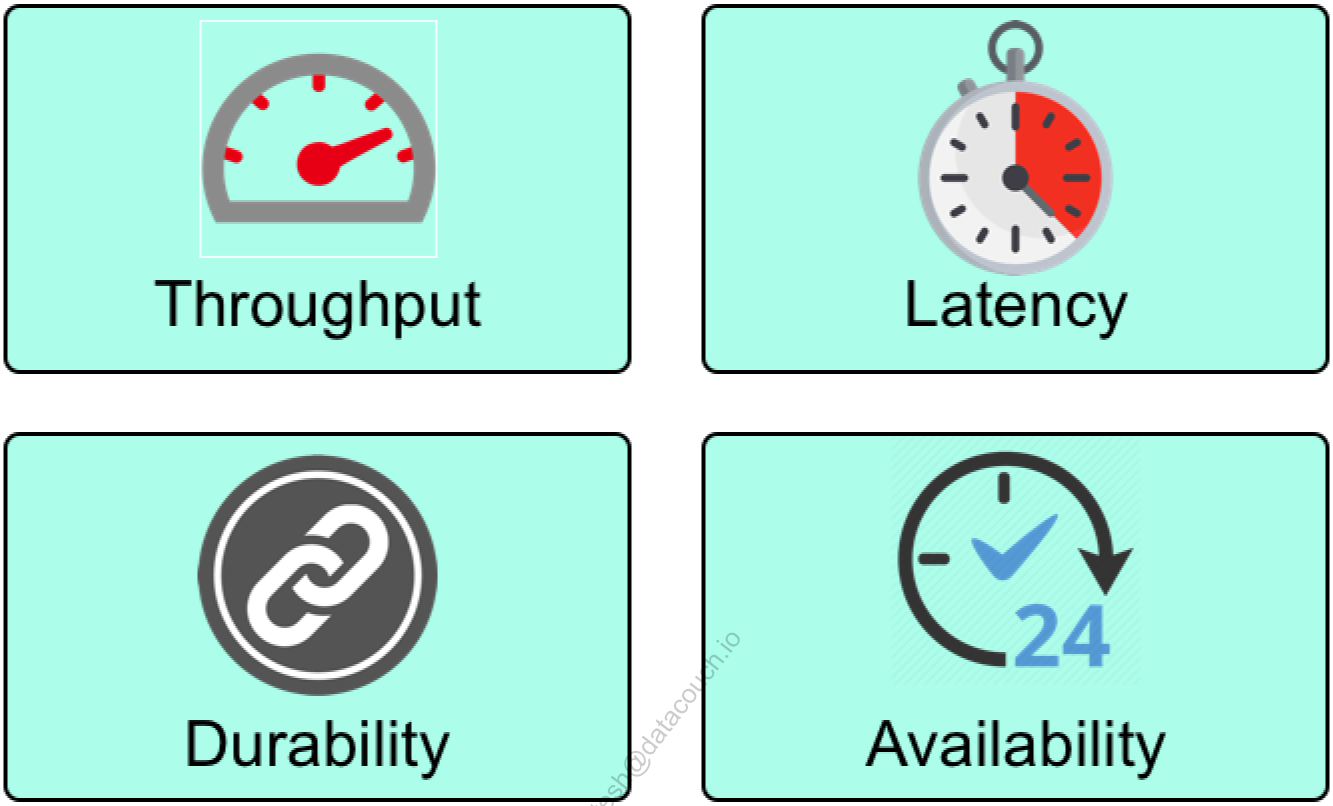
Establishing what is meant by throughput, latency, durability, availability. SLIs, SLOs/SLTs, SLAs.

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

Deciding Which Service Criteria to Optimize



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...

Instructor note: // IG only text

Service Level Indicators (SLIs)

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

Example: Overall throughput

Instructor note: // IG only text

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

SLI with target value

Example: Throughput > 200 MB/sec

Instructor note: // IG only text

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.

Instructor note: Definitions given on the last three slides are taken from *Kafka the Definitive Guide*, v2, p. 176)

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 prioritized when handling service problems

Further Reading

- *Kafka the Definitive Guide*, v2

Instructor note: // IG only text

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).

02: Monitoring Basics

CONFLUENT

Global Education

Module Overview

This module contains one lesson:

1. Monitoring Basics

a: Monitoring Basics

Description

Motivation for and basics of monitoring.

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

Motivation

You cannot fix problems you cannot see!

A real-time event streaming platform is a highly distributed system of a large quantity of interacting components, infrastructure and services. It is impossible to understand why this platform is failing to behave as expected all the time, around the clock if we do not monitor essential parameters of it.

Thus the slogan: "You cannot fix problems you cannot see!"

Let’s turn over the rug and shine light into the dark corners…

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**?

As operations engineers and/or DevOps engineers, architects, etc. we need to remember why we are operating an event streaming platform. It is to satisfy a business need. The business is our primary stakeholder. It defines the Service Level Agreements (SLAs) that we must satisfy. Questions that business stakeholders may ask, and that we need to be able to answer are those listed on the slide such as:

- Do we see all data?
- Is it the latest data, or is it stale?
- Are we sure to never lose data?
- Etc.

These questions of course drive what we will have to monitor for…

What's Important to You?

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

Not every business has the same requirements. A bank puts the focus on different aspects of event streaming than say a company dealing with IoT devices such as a car company.

Some keywords that describe the overall event streaming platform are listed on the slide. They are often somewhat orthogonal and sometimes they're exclusive, that is, we cannot optimize them at the same time. A sample of such mutual exclusiveness would be to minimize latency versus to maximize throughput.

- 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

As in within physical buildings, it is the foundation that supports the whole building. If the foundation is rotten and we do not monitor it, the whole building can collapse. For our event streaming platform the foundation is the infrastructure such as the physical and virtual hardware as well as the operating system. Thus we need to carefully monitor this infrastructure. A few things that belong into this category are listed on the slide. We need to e.g. monitor the CPU load, the network traffic, disk IO and more.

- 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

Assuming that we have done our homework and are monitoring the foundation, then we come to the actual event streaming platform with its plethora of components. Each component, as we will see later today, exposes dozens if not hundreds of interesting looking metrics. If we try to monitor them all then we undergo what's said "death by a thousand cuts". We will simply be overwhelmed by the sheer amount of information.
Thus we need to concentrate on the essential metrics. This course should help you to do so.

Review

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

Let's hold for a second and review what we have learned so far. And to do this I'll give you the question on the slide: "What is your company's highest priority, throughput or minimal latency?"
Or maybe it's something completely different? Think about it for a minute...

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

The last slide of every module of this course compiles a short list of material worth reading in the given context. You can just click the links in the PDF to access this additional material.

By the way, this is one of the only modules that does not have an associated hands-on lab. But don't worry, there are plenty of labs to come where you can tinker with the system.

03: Generic Monitoring

CONFLUENT
Global Education

Module Overview

This module contains one lessons:

1. Generic Monitoring

a: Generic Monitoring

Description

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

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

Monitoring JMX Metrics

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

Apache Kafka and pretty much all the other components of the platform such as Schema Registry, ksqlDB Server, Kafka Streams are written in Java. This is essential to know when trying to monitor those components. In Java the most popular way on how to expose metrics is to use Java Management Extensions (JMX). JMX is a defacto standard and wildly used in the industry. There is a significant ecosystem around JMX. We can use tools or SaaS services such as Grafana, Graphite, CloudWatch or DataDog to monitor JMX based metrics. Or we can use simple tools such as `jconsole` and `jmxterm` to do so.

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:

`zk_outstanding_requests`	Number of requests queued
`zk_avg_latency`	Amount of time it takes to respond to a client request (in ms)
`zk_num_alive_connections`	Number of clients connected to ZooKeeper
`zk_followers`	Number of active followers
`zk_pending_syncs`	Number of pending syncs from followers
`zk_open_file_descriptor_count`	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**

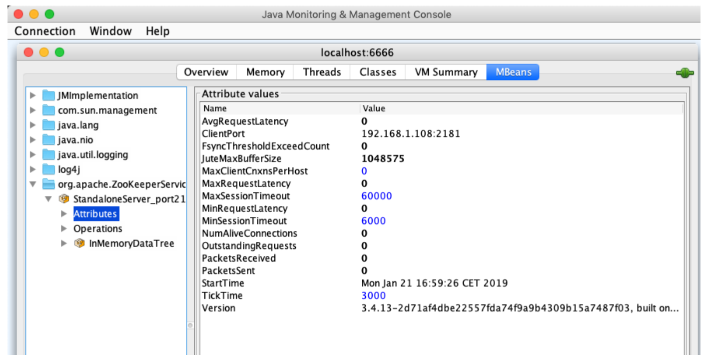
Monitoring ZooKeeper - JMX Metrics

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
```



JMX metrics are disabled by default. One needs to enable them first, to be able to monitor the relevant component. To do this for ZooKeeper we have two snippets on the slide. The top one on the left is for situations when running ZK in a container and on the right for when running natively.

Since when running inside a container we basically access its JMX metrics remotely, we have to define the environment variable `JMX_HOSTNAME`. `JMX_HOSTNAME` is the hostname of the host on which ZK runs and needs to correspond to the name used in the URL of the tool that accesses the JMX metrics…

In the container example we map the container port `9999` to the host port `9999` and thus can access the JMX metrics via `localhost:9999` or `127.0.0.1:9999`.

In one of the labs you will play with that environment variable to develop a better understanding.

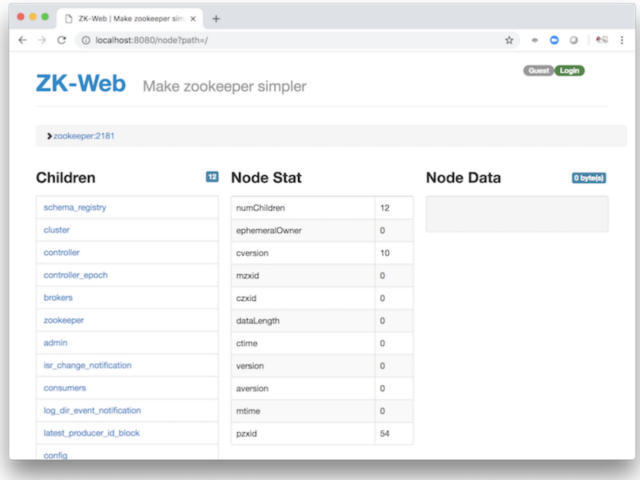
Monitoring ZooKeeper - UI

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
```



`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`

ℹ	To run `zk-web` natively currently one needs `leiningen` and `git`

Additional information on `leiningen`can be found at the following link.
https://leiningen.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

ℹ	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:

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

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
`metrics.sample.window.ms`	Size of each sample window	30 sec.
`metrics.num.samples`	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
```

On the slide you see a few important metrics exposed by a broker via JMX that are indicative for the load of the broker. These are `MessagesInPerSec`, `BytesInPerSec`, `RequestsPerSec` and `BytesOutPerSec`. Pretty much self describing, isn't it?

If the way how these metrics are listed are unfamiliar to you, don't worry, they will be much more familiar once you have done the hands-on labs at the end of the module. Let me just say that this is the standard way of describing JMX metrics.

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 Montoring (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
    ...
```

On the slide once more you can see how you enable JMX in the containerized version on the left, and the native version on the right.

As a side note, the port `9999` is by no means the only valid port. You can use any free port on the host over which JMX will expose its metrics, such as `34,444`.

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"
  }
}
```

LinkedIn, the originators of Kafka, have developed a tool called **Burrow**, they use to monitor Kafka. It is mainly used to monitor the consumer lag in 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 can also be used to get info about the cluster via well defined HTTP endpoints. On the slide you have a call to such an endpoint. The result shows some info of my simple cluster.

Monitoring Kafka Clients

Common per-broker Metrics for:

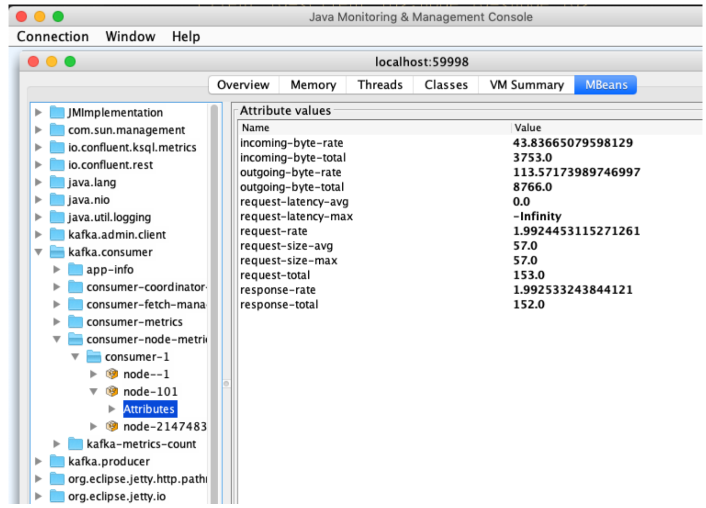
Consumer, Producer, Connect, Kafka Streams, ksqlDB

```
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:

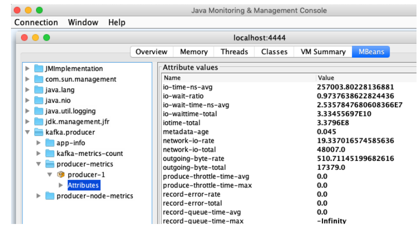


All Kafka clients (consumer, producer, connect, streams, ksqlDB) have common metrics. They are subsumed under the MBean `[consumer|producer|connect]-node-metrics`.

Part of the metric, and used for clear identification, are always the `client-id` and the `node-id` of the component under observation.

On the right hand side of the slide you see the use of jconsole when observing the metrics of a ksqlDB Server.

Monitoring Producers



- Common Kafka Client metrics
- Producer metrics

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

- Producer Sender metrics

- Key Producer metrics

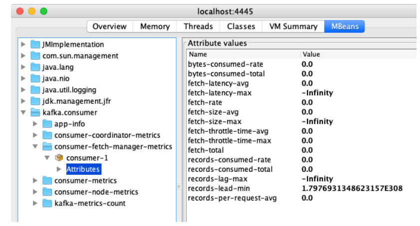
Response rate	Average number of responses received per second
Request rate	Average number of requests sent per second
Request latency avg	Average request latency (in ms)
Outgoing byte rate	Average number of outgoing/incoming bytes per second
IO wait time ns avg	Average length of time the I/O thread spent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

Let's now have a look at a producer written in Java, as one of the specific types of Kafka clients. Important metrics to observe on the producer-metrics side are listed in the bottom half of the slide. It's the response and request rates, the request latency average, as well as the outgoing byte rate.

Monitoring Consumers

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

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



- 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

Here we look at a Kafka consumer written in Java. On the bottom of the slide we have a few metrics to have an eye on, and which are good indicators whether or not the consumer behaves as expected. These are the consumer lag, current and maximum, the bytes or messages consumed per second, and the minimal fetch rate.

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...
```

One of the most important metrics of a consumer group is consumer lag. This tells us whether or not the consumers can keep up the consumption with the inflow of messages into their source topics. On the slide we have given the relevant metric. We can use the tool `kafka-consumer-groups` to find out more details about the lag of a given consumer group. Information is given per partition and consumer instance. Note, that the tool's values are based on the last commits of offsets by the consumer group whilst JMX exposes more real-time measurements.

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

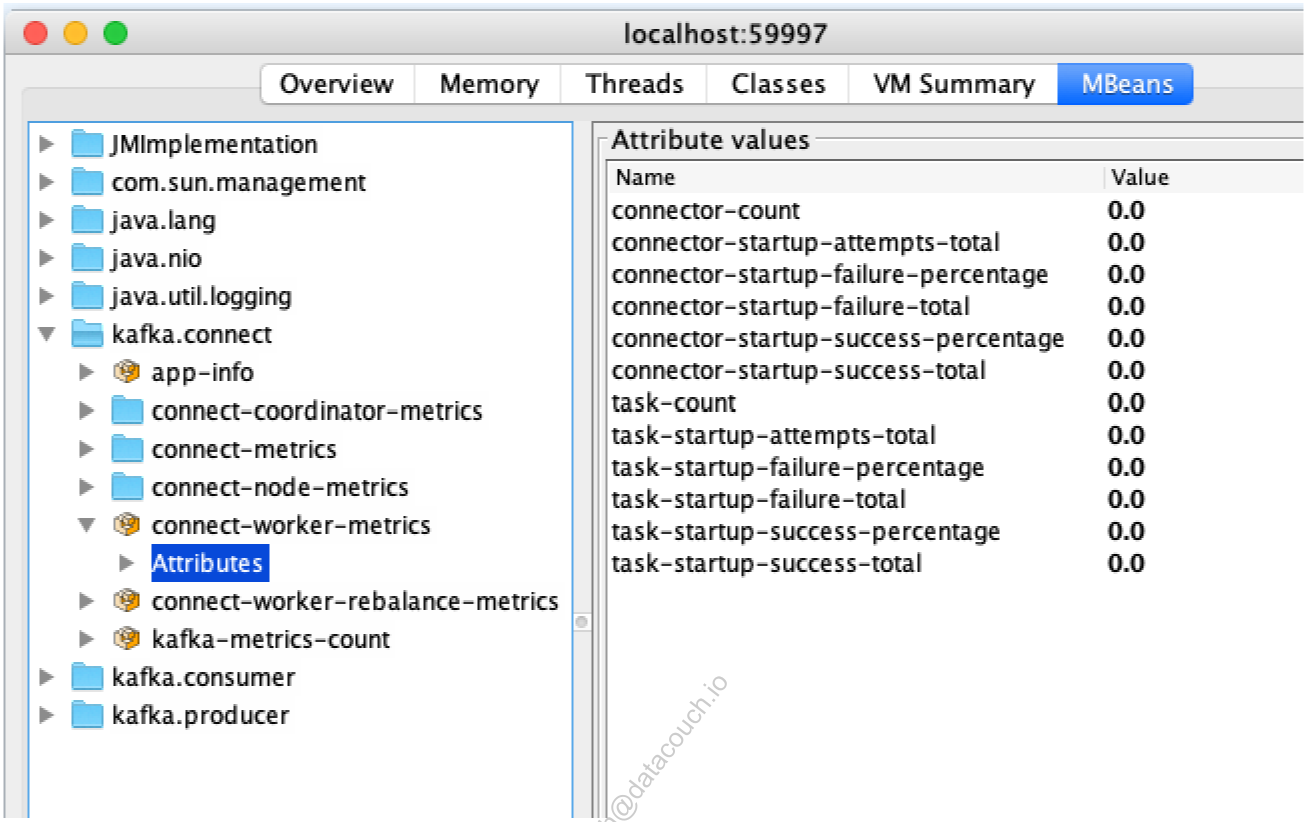
Distinguish standalone vs. distributed workers

Kafka Connect is a pretty complex beast and as such exposes many metrics. As always there are the common Kafka client metrics we discussed earlier. And then there are the Connect specific metrics, that can be sub classed further as shown on the slide. We have connector specific and common task metrics. Then we have source and sink task specific metrics. Furthermore there are worker and worker rebalance specific metrics, all nicely subsumed in the metrics tree as can be observed using a tool such as our beloved `jconsole`.

We should of course also not to forget to monitor the foundation of Kafka Connect, the host on which it runs. Then we might want to monitor the source or sink system depending on whether we have a source or sink connector. If the external systems behave bad then Connect cannot be successful, and we need to discover that.

Finally, the connect workers have a REST API through which we can monitor them

Monitoring Kafka Connect



On the slide we see jconsole used to monitor an instance of Kafka Connect. We can see that connect exposes producer and consumer metrics as well as connect specific metrics. They are further grouped into mode or worker specific metrics as we can see.

Monitoring Kafka Streams & ksqlDB Apps

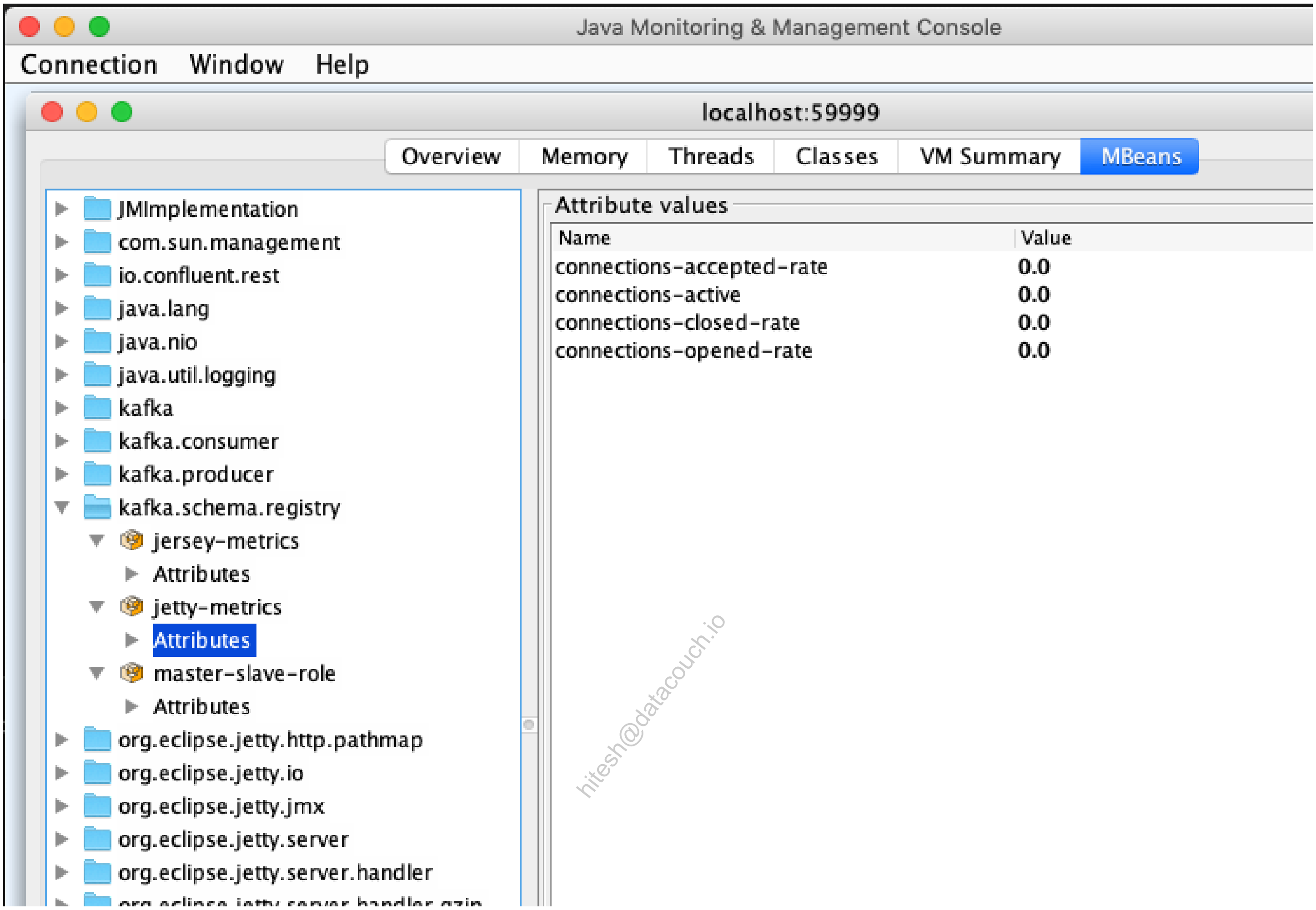
Monitor:

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

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

When dealing with Kafka Streams and/or ksqlDB applications we should monitor metrics that are related to threads, tasks, processor nodes, the state stores and the caches used by stateful applications. Note that although ksqlDB under the hood is "just" a Kafka Streams app, it exposes some additional metrics, beyond what we get from Kafka Streams.

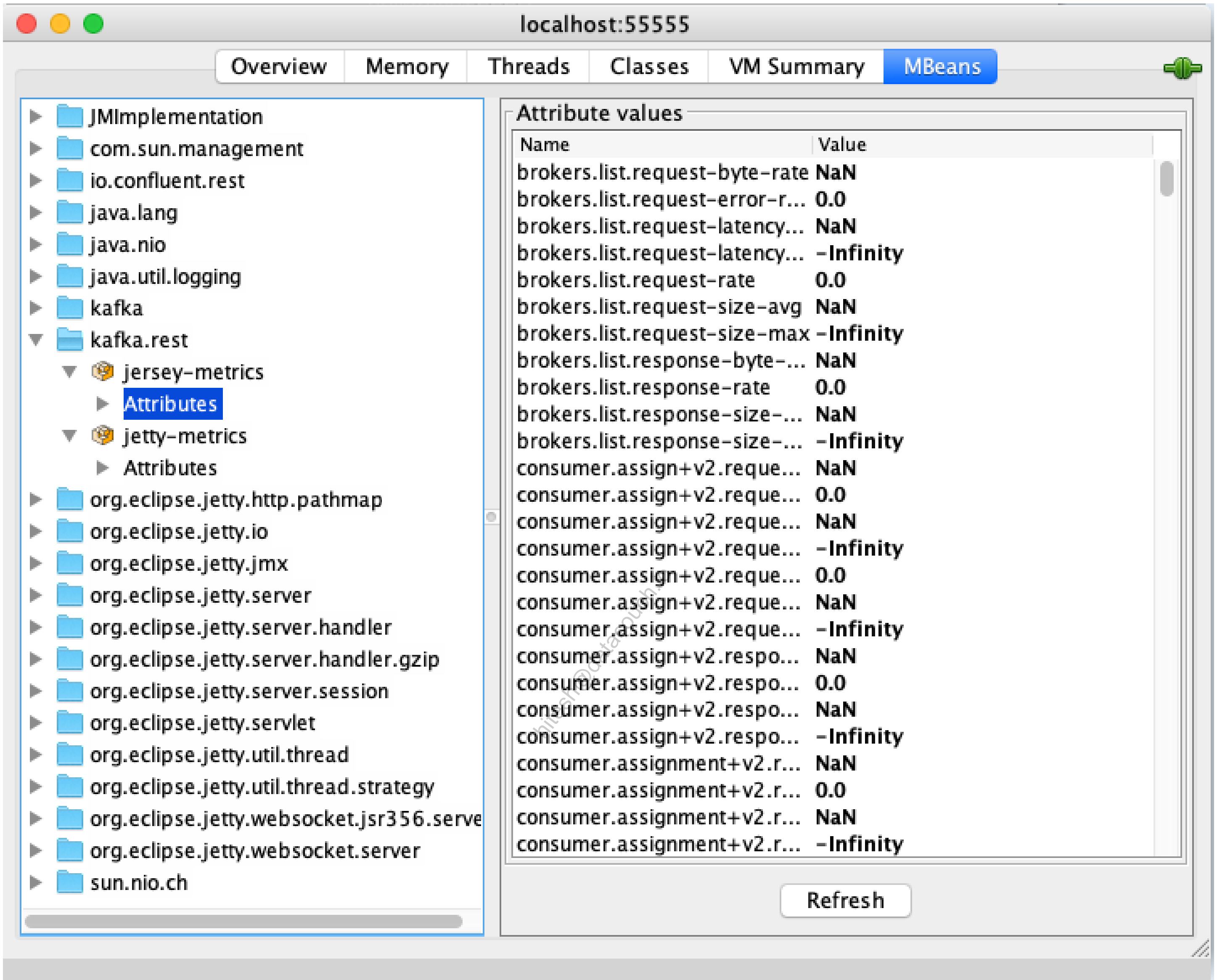
Monitoring Confluent Schema Registry



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`. You can see this clearly on the slide which contains a screen shot of `jconsole` monitoring the 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 also shown on the slide.

Monitoring Confluent REST Proxy



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 you cannot see those MBeans, simply because my REST Proxy did not have any active connections, producers or consumers.

Review

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

After having heard the theory and executed 4 exercises, please try to answer the following question: "Why it doesn't make sense to monitor all possible metrics of any component of the Confluent Platform?"

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

Lab: Introduction

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

Refer to the Exercise Guide

Lab: Monitoring via JMX

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

Refer to the Exercise Guide

Lab: Monitoring librdkafka based Clients

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

Refer to the Exercise Guide

Lab: Monitoring with Prometheus

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

Refer to the Exercise Guide

04: Monitoring with CCC

CONFLUENT
Global Education

Module Overview

This module contains one lesson:

1. Monitoring with CCC

a: Monitoring with Confluent Control Center

Description

Tour of CCC and its monitoring capabilities.

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

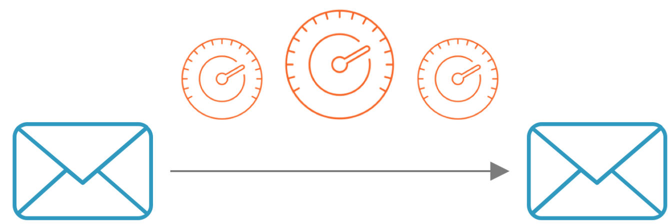
Control Center

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?

Now that we have seen how we can build our own monitoring system or setup I want to introduce Control Center to you. We call it the "Expert Kafka Monitoring for the Enterprise". Two main areas that it covers are system health and end-to-end SLA monitoring.

When it comes to system health we can monitor such things as:

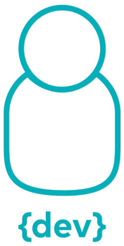
- Are all brokers and topics available?
- How much data is being processed?
- What can be tuned to improve performance?

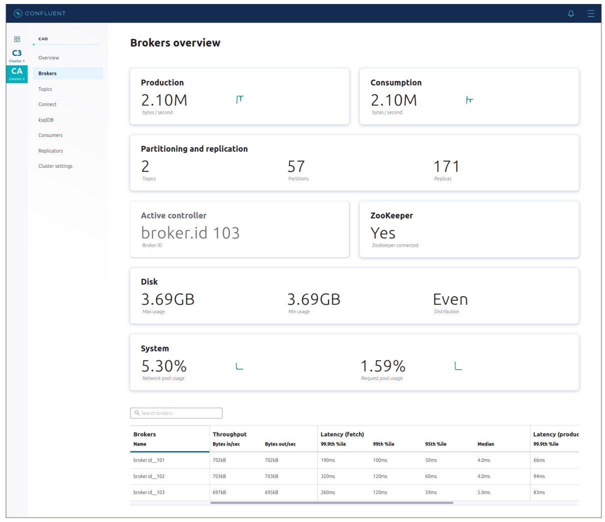
Under the monitoring of SLAs we can subsume things like:

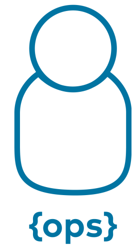
- Does Kafka process all events <15 seconds?
- Is the 8am report missing data?

Are there duplicate events?

Control Center

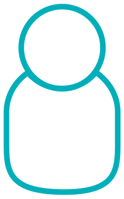






Control Center is very useful for both roles, developers and operations engineers. It offers a unique insight into the Kafka powered, real-time event streaming platform with its many components.

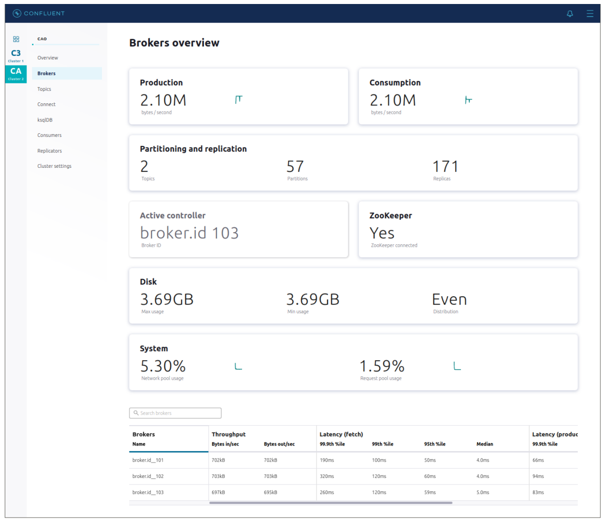
Control Center

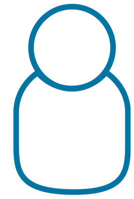


{dev}

Look inside Kafka

- Inspect messages in topics
- View changes to Schema Registry

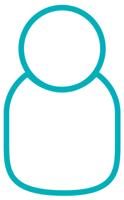




{ops}

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

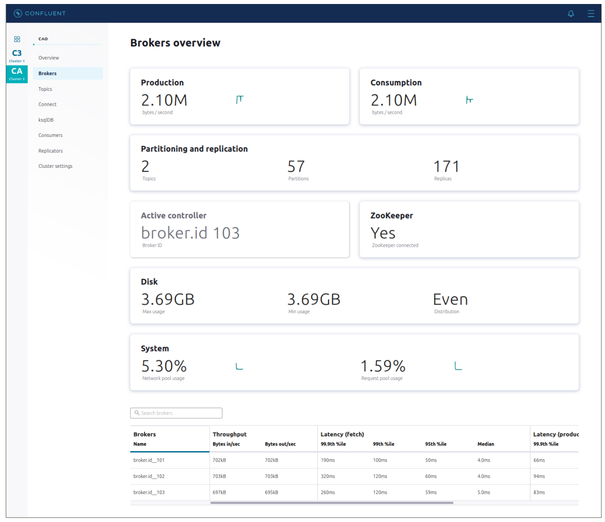

{dev}

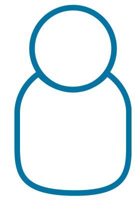
Look inside Kafka

- Inspect messages in topics
- View changes to Schema Registry

Build pipelines and process streams

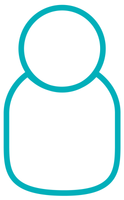
- Configure Kafka Connect and connectors
- Write ksqlDB queries




{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



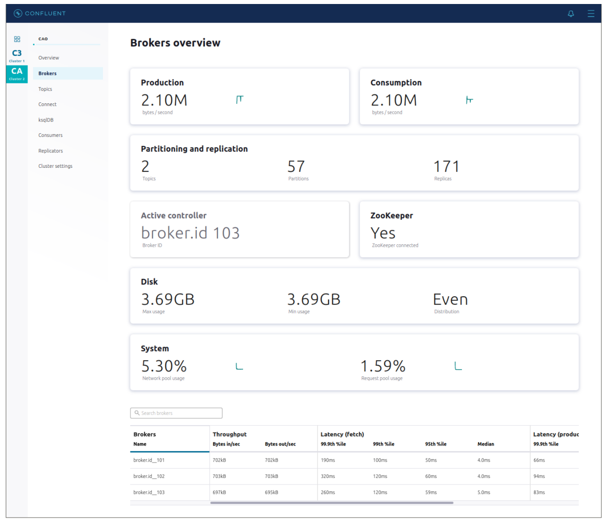
{dev}

Look inside Kafka

- Inspect messages in topics
- View changes to Schema Registry

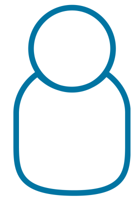
Build pipelines and process streams

- Configure Kafka Connect and connectors
- Write ksqlDB queries



Meet event stream SLAs

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



{ops}

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 perfomance indices)
- monitoring the lag of consumers and consumer groups
- defining and receiving alerts on certain event, such as exceeding a threshold value

Control Center

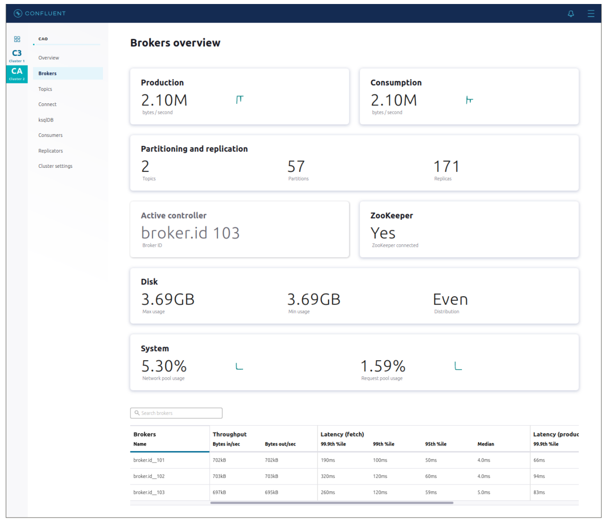
{dev}

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



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

{ops}

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

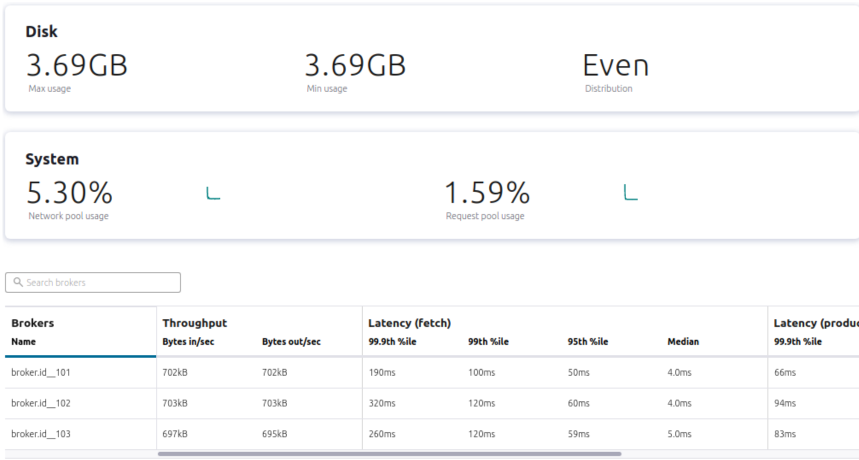
---

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

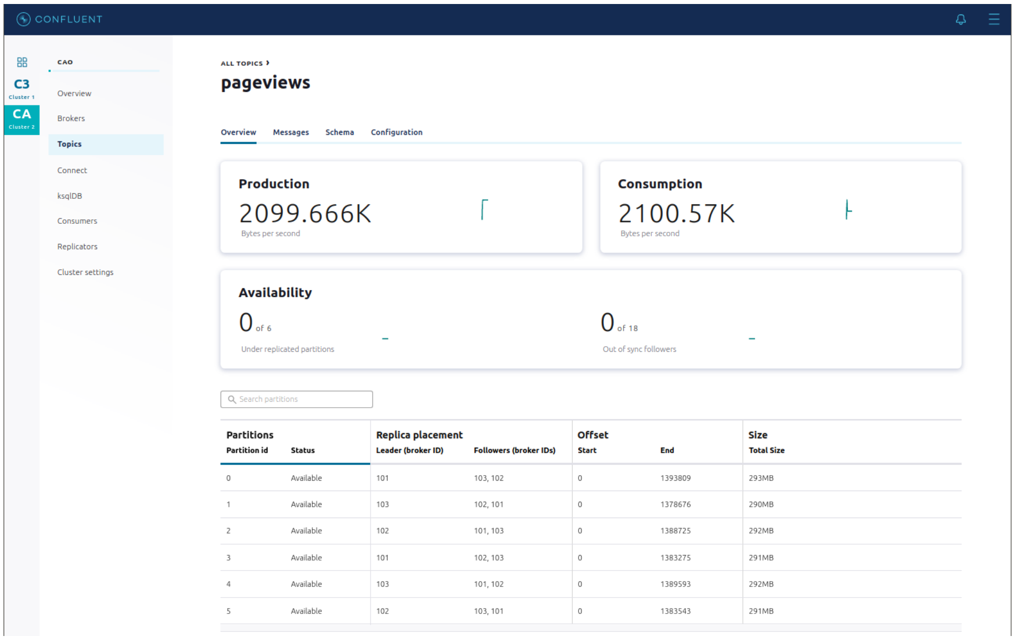
We will now examine these different Control Center views

The Cluster View

- Performance metrics (throughput and request latency)

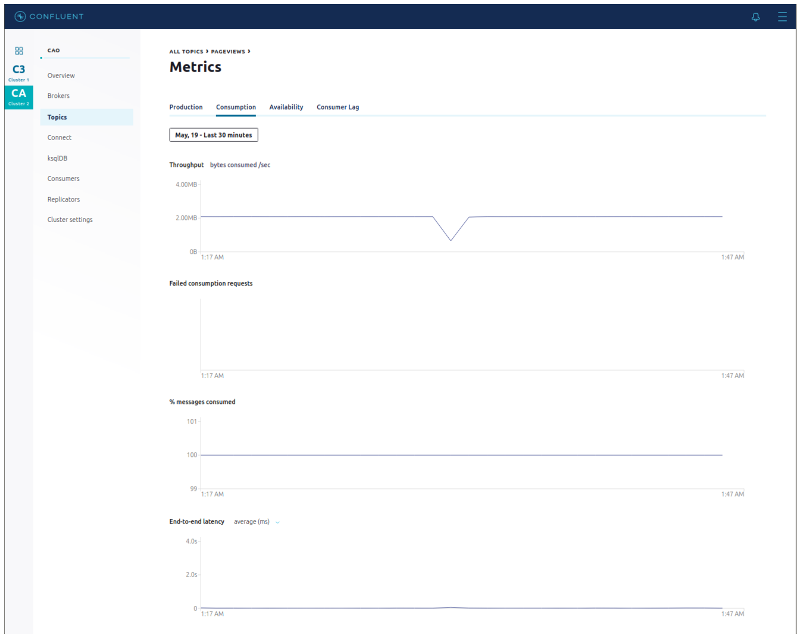
Here on the slide we see graphs that show us overall cluster performance based upon relevant performance metrics such as throughput and request latency.

Topic Overview



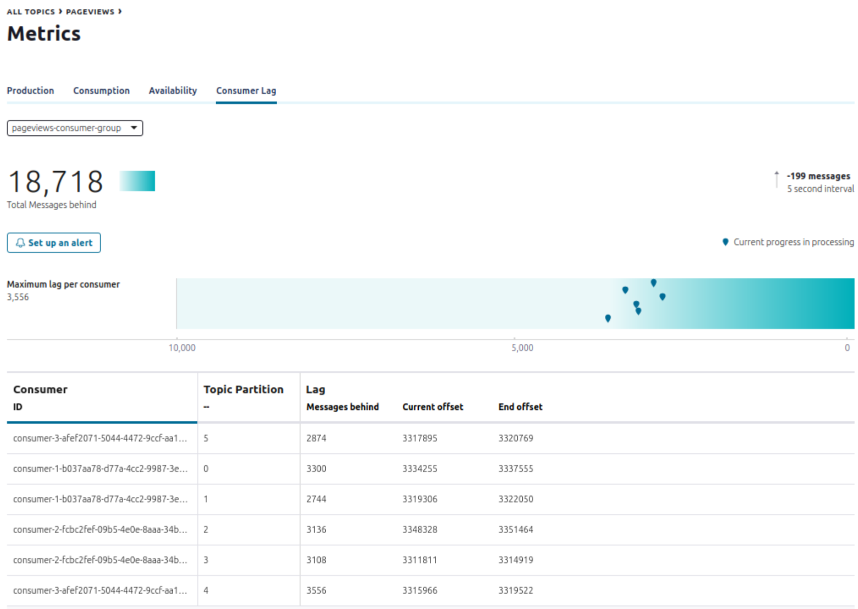
The initial Topics pane provides a list of all topics in the cluster. Selecting a topic from the list opens an overview pane that reflects current production and consumption rates, partition status and size, and replica placement.

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 anomolies that may have occurred or be occurring in these areas for the event stream.

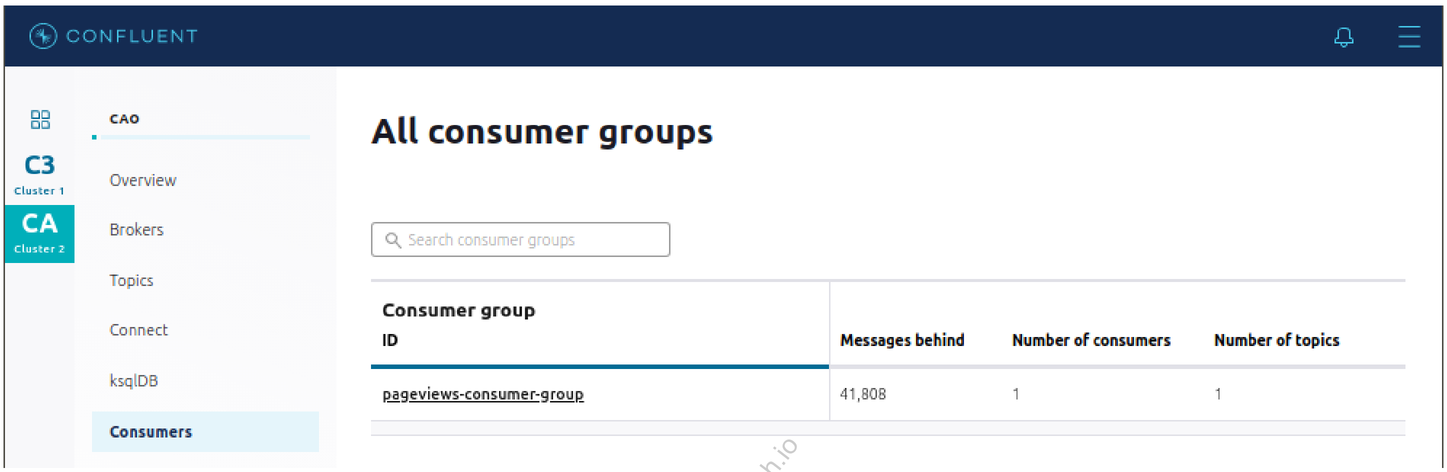
Topic - Consumer Lag



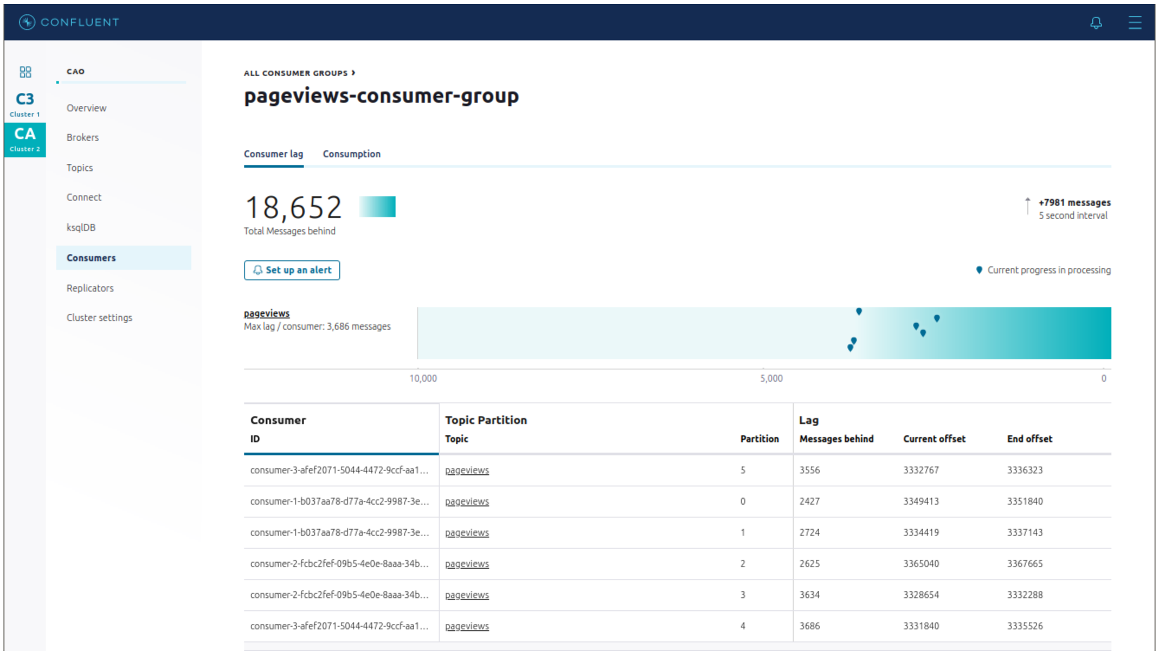
The Topic Consumer Lag tab reflects the overall message consumption status by all consumers. It provides a intuitive means to determine if consumer groups need to be scaled up or down.

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



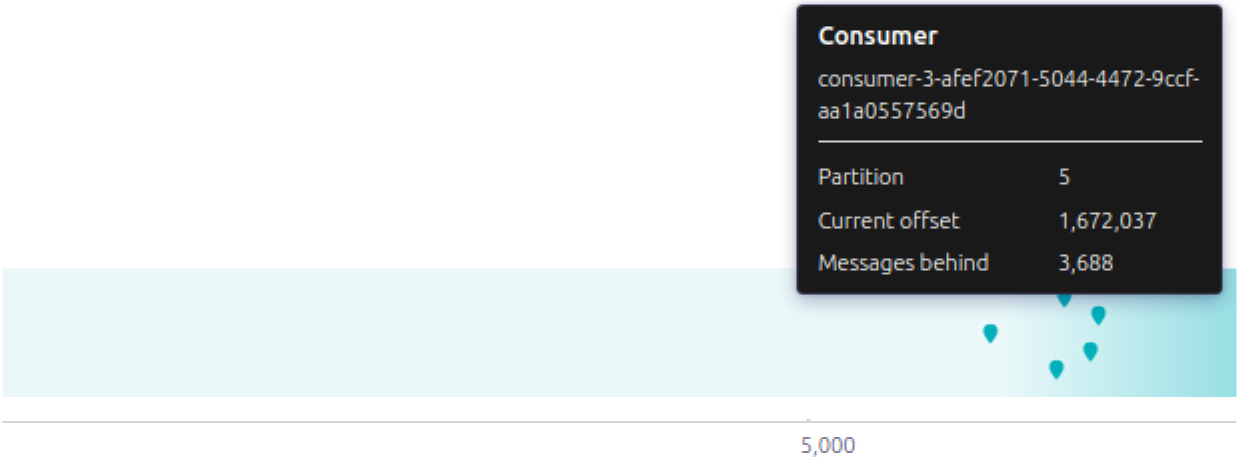
Consumers - Consumer Lag



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`.

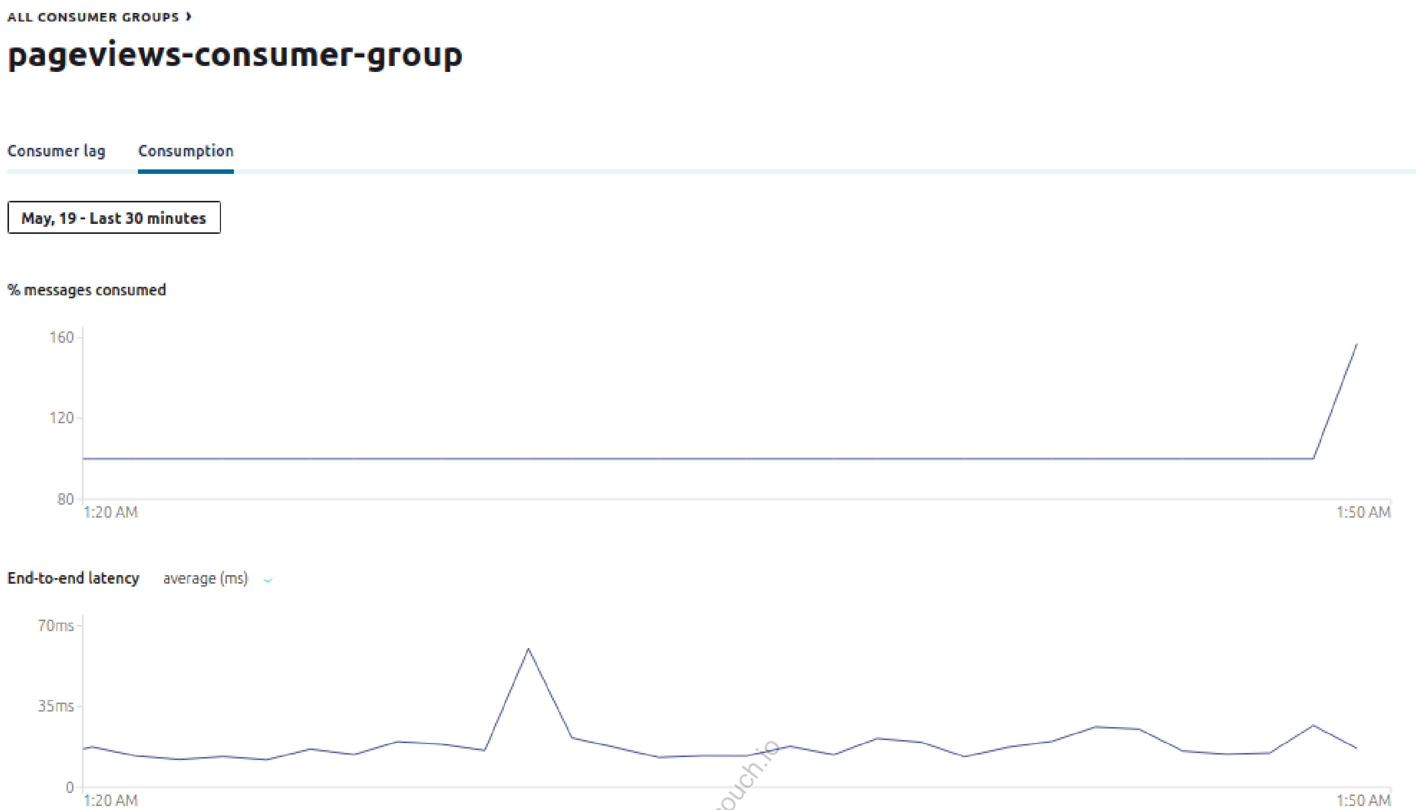
Consumers - Consumer Lag - Per Partition Details



Topic Partition	Lag		
--	Messages behind	Current offset	End offset
5	3688	1672037	1675725
0	3430	1681669	1685099

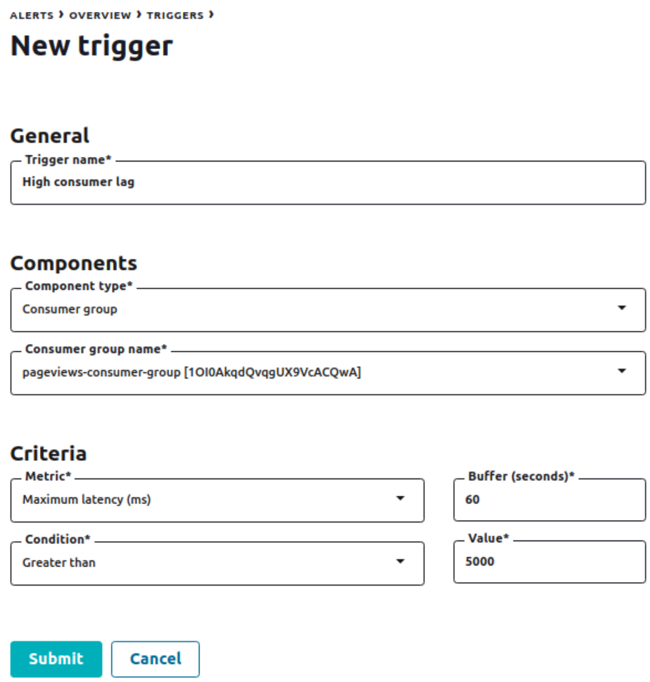
If we click on one of the drop symbols in the previous chart then we can see more detailed information. We see which consumer it is, the partition number, the current offset, and the current consumption lag expressed as messages behind.

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



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?

After having heard me raving about control center and after having played with it in the lab, please take a moment to answer the question on the slide.

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

Lab: Monitoring with CCC - Data Streams

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

Refer to the Exercise Guide

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?

05: General Troubleshooting

CONFLUENT
Global Education

Module Overview

This module contains three lessons:

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

a: Troubleshooting Intro

Description

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

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

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?

For our business stakeholder it is clear, our event streaming platform powered by Kafka is mission critical. Thus we need to ask ourselves a few questions:

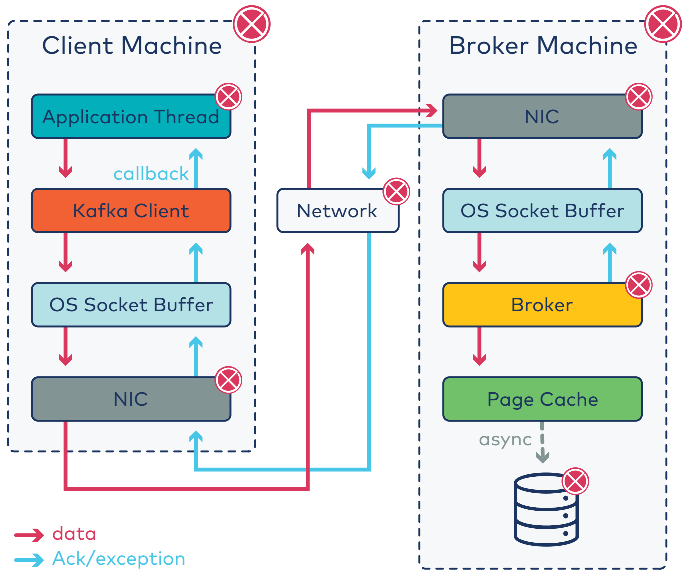
- Can we be 100% sure that our data will get there?
- Can we lose messages or is every single message essential?
- How do we verify the above two points?
- What is the root cause of the problem. Only with that knowledge can we fix the situation and prevent the problem from reoccurring.

Distributed Systems

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

First we need to be clear: in a distributed system things will fail. It is not a question IF they fail but rather WHEN. A good distributed system is designed in a way that it can tolerate failure of individual parts. And more, a good distributed system handles failures gracefully and in a deterministic way.

Data Flow



There are many possible points of failure on the road of a typical request-response path. Let's look at the request first. On the slide you can see that:

- 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

Now we can also look at the 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

What can possibly happen on the way just explained? As mentioned in the previous slide: in a distributed system things will fail. It is not a question IF they fail but rather WHEN. We can have the following points of failure:

- the client machine as a whole can fail
- the client application thread can crash
- the client NIC has a malfunction
- the network fails due to partitioning or congestion
- the broker's NIC fails
- the broker process crashes or is unresponsive
- we have a disk failure on the broker
- and many more...

We need to be prepared to troubleshoot all of the above cases and more...

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 is 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?

Now I want you to answer the question "What type of expertise does one need to troubleshoot an event streaming platform powered by Apache Kafka?"

Let the students think a bit, what the areas are that require troubleshooting, and determine what the required skillset is that is needed to be successful in it.

Further Reading

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

b: Production Down! Troubleshooting Strategies

Description

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

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

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

When an issue such as production down occurs then we need to classify the severity and the category of the issue. On the slide we have four levels of severity and the associated actions one would most probably envision.

- In case of a production down we have to implement an immediate solution. This can be a temporary hack, just to get things going again.
- If the production system is negatively affected, e.g. it is significantly slower than usual, we need a quick solution.
- If the problems creates significant friction, say for your developers to continue implementing Kafka clients, then you should report the error to Confluent via support
- Finally if it is a minor issue you should report it and in the mean time find a work around.

I also have a list of categories on the slide, under which we can classify the issue.

- Is it an infrastructure problem, e.g. physical servers, routers, firewalls, virtualization software like VMWare, etc.?
- Or is it a problem of the Confluent Event Streaming platform, such as Brokers, Kafka Connect, Schema Registry, etc.?
- Or is it a problem associated with one of the many custom applications written by your developers, such as producers, consumers or Kafka Streams applications.

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?

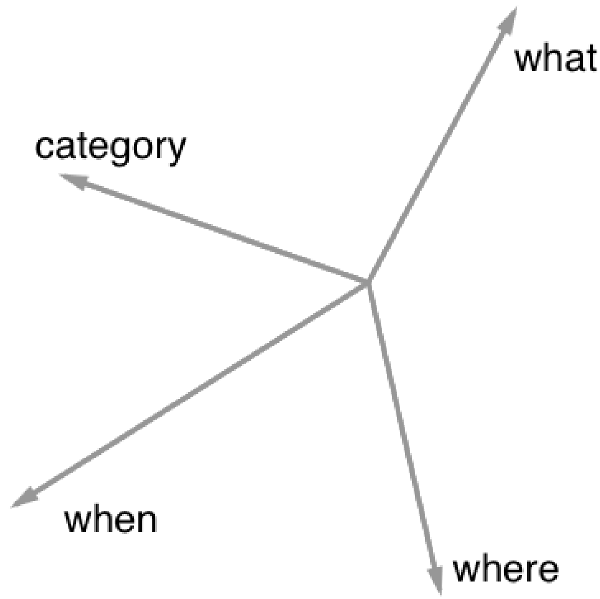
Asking the right questions in a crisis is of uttermost importance. We can loose a lot of time by asking insignificant questions, or by just trying to blame others. So, let's look into what is the right way to ask, shall we:

- 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



So, now that we have discussed what is the right way to ask questions to get a good solution strategy we look at the same from a more theoretical perspective:

- 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.

To summarize, please try to answer the following question and justifying it: "Which is the best way to troubleshoot consumer lag?"

Further Reading

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

c: Troubleshooting Toolbelt

Description

Tour of troubleshooting tools.

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

Linux Performance Observability Tools

Memory:

- free
- top, htop
- vmstat
- ...

I/O:

- iostat
- iotop
- blktrace
- ...

Network Stack:

- netstat
- netcat
- tcpdump
- iptraf
- ethtool
- ...

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:** 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
```

Docker support and field engineers use the "netshoot" container to troubleshoot on container based environments such as Kubernetes. The netshoot container includes a set of powerful tools as recommended by 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`

Here is a list of the most useful command line tools that are part of a typical Kafka installation. The tools can be used to manage ZooKeeper, Kafka configurations, Topics and Consumer Groups, ACLs. We also have tools such as the console consumer and producer that are useful when developing for Kafka or debugging Kafka. The last two tools listed come in handy when we want to stress test our Kafka cluster.

- `zookeeper-shell`: Utility to admininister 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

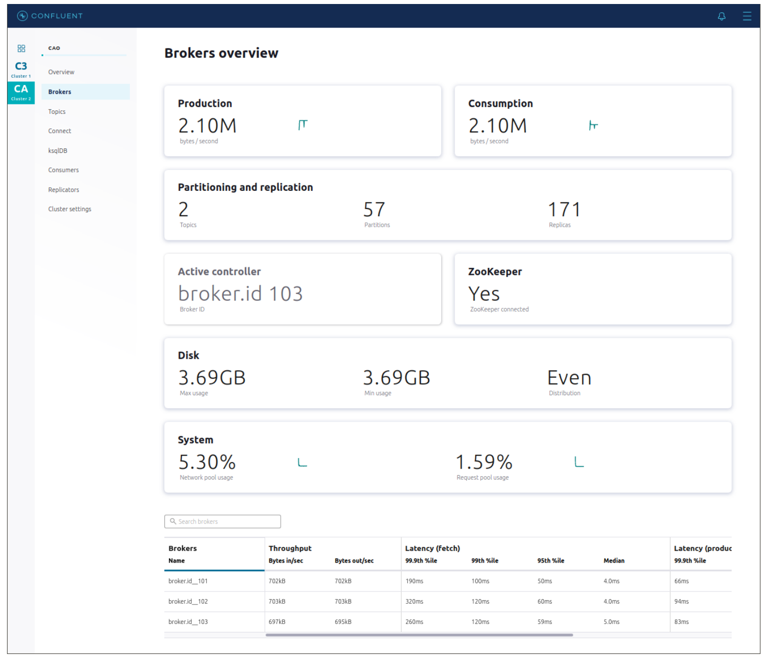
- Produce, consume and list topic and partition information
- `netcat` for Kafka

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

There is one tool that merits a dedicated slide. It is the Swiss army knife of Kafka related tools. We can use it to produce, consume and list topic and partition information as well as to test and debug Kafka deployments. No engineer working with Kafka should miss out on this tool!

Confluent Control Center

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



Of course Confluent's Control Center cannot be left aside here. As we have learned in one of the previous modules, it gives us information about broker health and configurations, information about topics and partitions, insight into data flow and end-to-end latency, consumer group specific information and replication related issues.

Review

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

Please try to answer the question on how to best troubleshoot consumer lag.

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

Lab: Troubleshooting Toolbelt

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

Refer to the Exercise Guide

06: Where are my System Log Files?

CONFLUENT
Global Education

Module Overview

This module contains one lesson:

1. Where are my System Log Files?

a: Where are my System Log Files?

Description

Overview of Kafka system logs and tips for analyzing them.

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

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?

ℹ	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` [which is somewhat outdated though])
- 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	KSQL	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`. Also defined in the config file are the log targets and the log levels. Log targets for containerized components are `STDOUT` and `STDERR` whilst for native installations it is rolling files.

Log4j Configuration in Production Systems

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

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

On this slide we have some recommended settings for a production or production like environment. The specific settings are marked in bold in the snippet. The default Log4j configuration must be modified slightly to ensure the log files do not fill up the disk. Note, these settings are for native installations. 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.

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 containers, use the 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 location of the log files can be set via `kafka.logs.dir` parameter.

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
```

We can use the environment variable `KAFKA_LOG4J_OPTS` to define an alternative `log4j.properties`. If running in containers we have the options specified in the code snippet on the slide. We can e.g. override the root log level using `KAFKA_LOG4J_ROOT_LOGLEVEL`, or override individual logger levels using `KAFKA_LOG4J_LOGGERS`.

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 to `DEBUG` to get more detailed data. 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 `kafka-configs` to dynamically change the log level.

You will modify broker logging levels using `kafka-configs` in the hands on lab that is part of the **Troubleshooting Brokers** module.

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
```

On this slide you see how to configure an alternative `log4j` configuration file by using the command line parameter `-D`. When running a Kafka client in a container then the `log4j` configuration typically looks like the one given in the lower code snippet.

ksqlDB Server

Installation environment	How to access the ksqlDB logs

Docker container	`$ docker logs <container ID>`
	 — or —
	 `$ docker-compose logs ksqldb-server`

Confluent CLI	`$ bin/confluent logs ksqldb-server`
RPM/DEB	Log files in folder `/var/log/confluent`

- Examine logs and look for deserialization errors:

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


Here on the slide we have ways of getting access to log information generated by ksqlDB server, when running in containers, running natively in a development or demo environment and finally when ksqlDB has been installed using RPM/DEB. In the ksqlDB logs we should specifically look for deserialization errors. The last code snippet shows how to do that in a containerized environment such as in our labs

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 first command on the slide 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 Control Center 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 <C3 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

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?

Please try to answer the question given on the slide.

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

Lab: Where are my Log Files?

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

Refer to the Exercise Guide

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

07: Troubleshooting & Tuning ZooKeeper

CONFLUENT
Global Education

Module Overview

This module contains one lesson:

1. Troubleshooting Zookeeper

a: Troubleshooting ZooKeeper

Description

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

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

ZooKeeper's Responsibilities

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

To be able to efficiently troubleshoot ZK we need to know what are ZKs responsibilities in the first place. Well let me list them:

- First, ZK is the source of truth for topic configuration
- The broker acting as the controller is chosen by ZK
- Authorization for resources is stored and synchronized by ZK
- How much data each Kafka client is allowed to read or write is governed by what is stored in ZK
- Which brokers are actually active in the cluster is stored in a ZK node
- The leader for each partition of each topic is stored in ZK
- Information about in-sync replicas is stored in ZK,
- Finally, ZK stores all dynamically configured broker values

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 oder 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

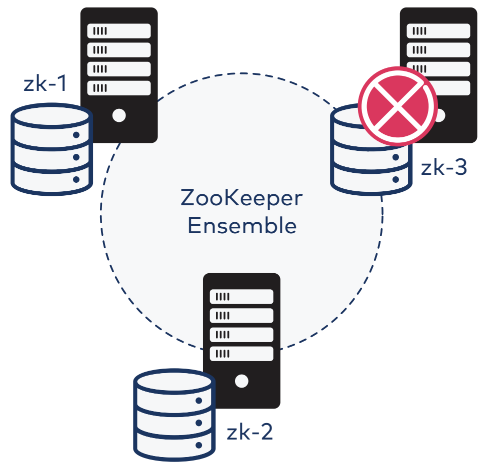
   Run the following commands and confirm the responses are equal:

   `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 by 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
```

This is useful when debugging issues with ZooKeeper connectivity or when trying to locate a unexpected ZooKeeper client. We can use commands similar to the ones given on the slide to read the transaction and/or snapshot logs.

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.

Now, 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.

Review

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

Please try to answer the question on the slide. Organize in groups of two.

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

---

Additional sources that deepen the context.

Lab: Troubleshooting ZooKeeper

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

Refer to the Exercise Guide

08: Troubleshooting & Tuning Brokers

CONFLUENT
Global Education

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

a: Troubleshooting Brokers

Description

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

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

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

This list on the slide was established with the help of our COPS engineers.

- **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.

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 rights. 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 broken pipe`
- 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...)

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`

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
`recovery-point-offset-checkpoint`	Last offset flushed to disk
`cleaner-offset-checkpoint`	Offset up to which the cleaner has cleaned (compacted topics only)
`replication-offset-checkpoint`	Last committed offset
`log-start-offset-checkpoint`	Internal broker log where Kafka tracks the starting offset of the topic partition
`leader-epoch-checkpoint`	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/`

Is Broker In Sync?

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

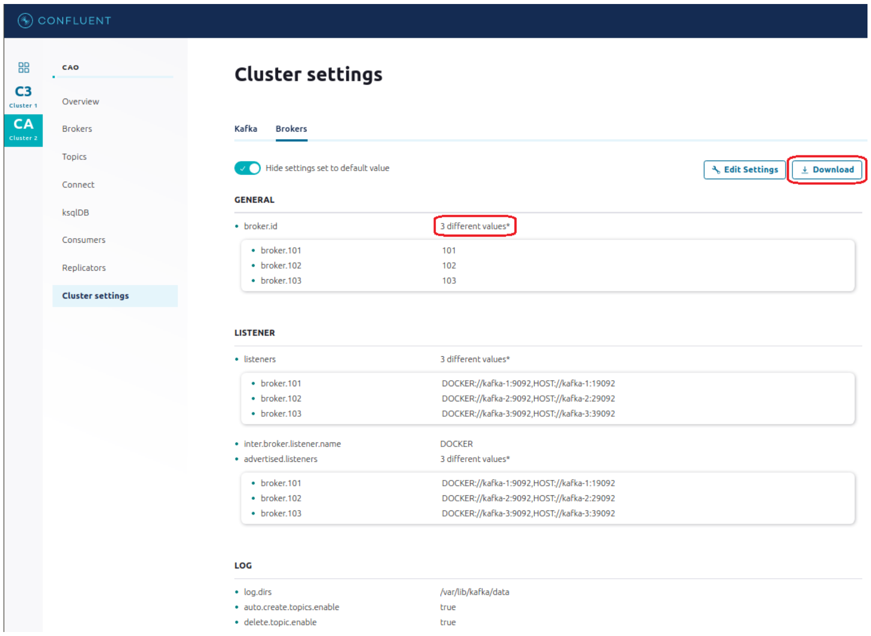
As with most distributed systems automatically handling failures requires having a precise definition of what it means for a node to be "alive". For Kafka node liveness has two conditions:

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



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 slide describes how to gracefully shutdown a set of Kafka brokers in order to guard against trouble upon restarting the cluster.

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.

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:+UseGCLogFileRotation -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:+UseGCLogFileRotation -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 replica set:

- 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/

b: Replication Concerns

Description

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

Learning Objectives

Completing this lesson and associated exercises will enable you to:

- Identify replication related issues

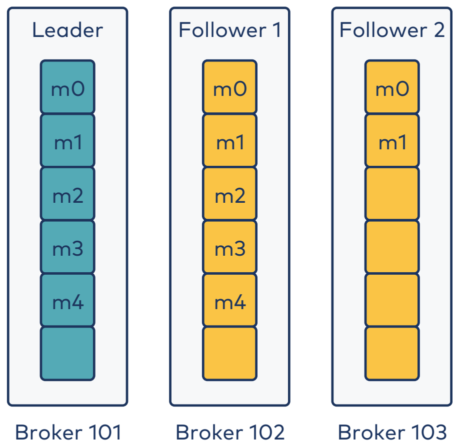
Recap

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

To recap some facts before we dive more into replication issues: 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

What are possible reasons that a replica gets out of sync?

- **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

Lab: Not Enough ISRs

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

Refer to the Exercise Guide

c: Tuning Brokers: General Concepts & Best Practices

Description

Broker monitoring, general guidelines, performance checks.

Learning Objectives

Completing this lesson and associated exercises will enable you to:

- Define a short list of broker best-practices

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

When it comes to operating a broker then there are some ground rules that are important to follow. Items that belong into the category of tuning are:

- When not explicitly justified to use a custom value then mostly stick with the defaults. Our engineers have spent a lot of effort to configure the brokers as optimal as possible right out of the box.
- Determine what your data retention policy shall be. By default it is 7 days. Every company has different requirements. Once you have determined the ideal duration for your case define it as a cluster wide retention policy
- How many brokers will you be having in your cluster? Once the number is given, set the cluster wide default partition count to a number equal or equal to the number of brokers

Monitoring is always important. We have seen that in the section about troubleshooting. You cannot expect to tune your brokers without having the feedback through monitoring.

- There is a plethora of metrics available to monitor. Choose the right ones
- Define alerts wisely. Don't alert on everything but only on critical events, otherwise no one will take alerts serious.

When it comes to problems, then try to solve them, not mask them

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

On this slide I have given a list of metrics important to monitor. Ideally you display those metrics in a dashboard such as created with Grafana, or use Confluent Control Center. The latter is optimized to monitor your Kafka cluster out of the box.

We have seen in the previous section that when deviations from trends are visible then they often indicate a problem. We can then proactively troubleshoot and tune our brokers.

The metrics listed here are on a per broker level. They represent averages or sums over all partitions that are handled by the respective broker

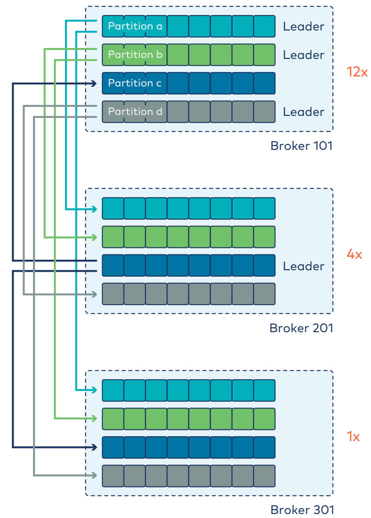
Topic Monitoring

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

Sometimes we need to drill down to specific topics and partitions on a broker. For this we have partition specific JMX metrics. I have listed the most important ones here on this slide. Ideally on our monitoring dashboard we can drill down from a broker level down to the topic level.

General Guidelines

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



There are some more guidelines to consider when looking for an optimally performing Kafka cluster. I have listed them here on the slide. The first item is very important and I have also tried to graphically represent what I am going to say about it now:

- **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?

If we experience a degradation in performance or the performance is not what we expected it to be then we can execute the following performance checks:

- First we check if all brokers in cluster are working
- Then we check if there are saturated network interfaces
  - 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)
- Next we check if the CPU utilization is high (specifically `iowait`)?
  - Are other processes competing for resources?
  - Do we have a bad disk?
- Do we have really big messages? Kafka is not optimized or architected for big messages. We should not exceed the limit of 1MB. Even that size is already a **big** message. If we absolutely need big messages then there are other architectural patterns available, e.g. to store the message payload in some blob storage and instead pass its URI to that message payload in the actual message sent to Kafka.

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

Description

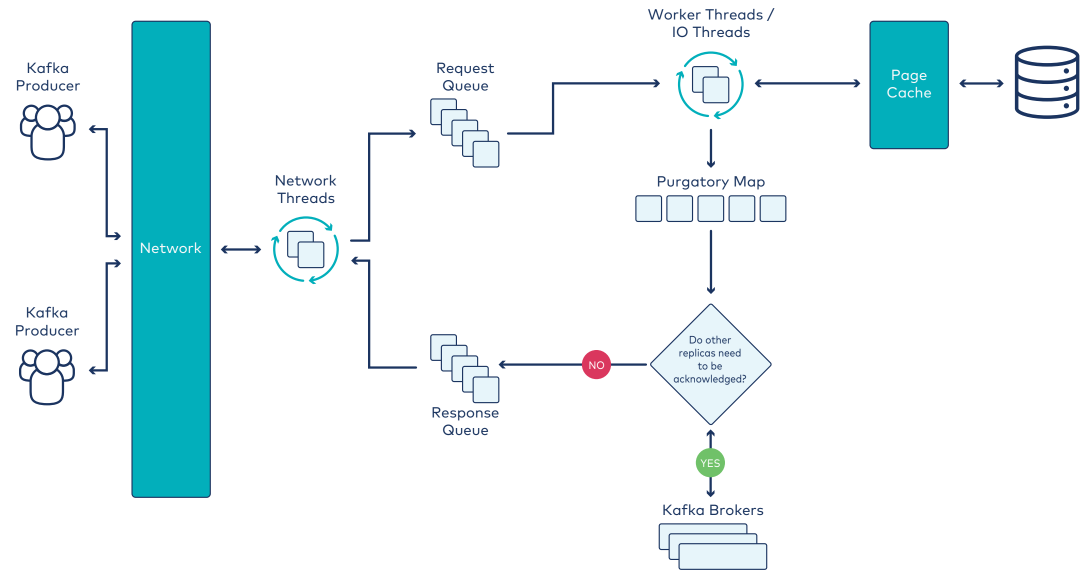
Review of the life cycle of requests on a broker. Metrics overall and at various stages. Tips for mitigating high latencies.

Learning Objectives

Completing this lesson and associated exercises will enable you to:

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

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

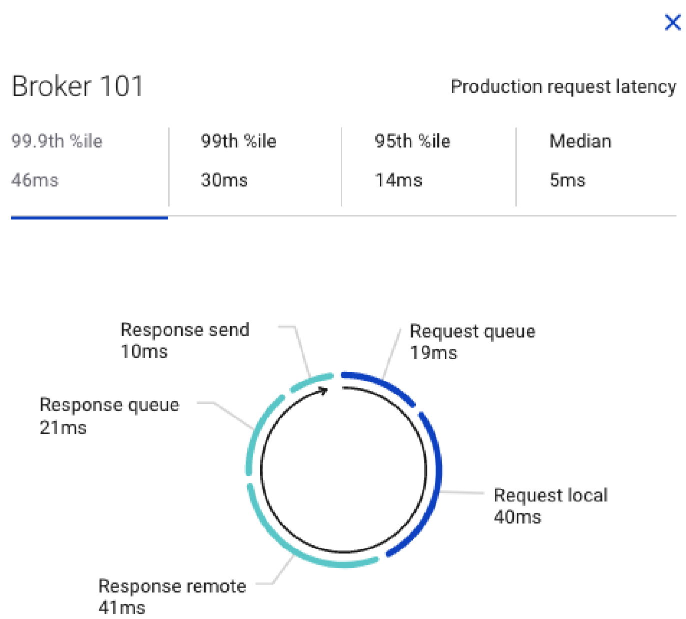
Apache Kafka has a data structure called the "request purgatory". The purgatory holds any request that hasn't yet met its criteria to succeed but also hasn't yet resulted in an error.

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:

- the criteria they requested is complete or
- some timeout occurs.

Producer Request in Control Center

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



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:

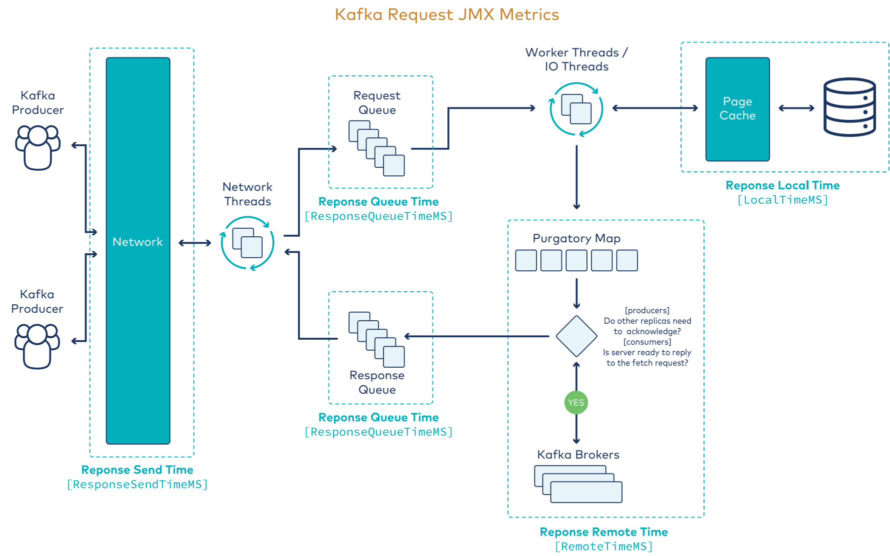
Metric	Description
`RequestQueueTimeMs`	Time the request waits in the request queue
`ResponseSendTimeMs`	Time to send the response
`ResponseQueueTimeMs`	Time the request waits in the response queue
`LocalTimeMs`	Time the request is processed at the leader
`RemoteTimeMs`	Time the request waits for the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

On this slide we see what the meaning of the previous 5 times shown in the Control Center's Producer request latency chart mean. I have numbered them from 1 to 5:

1. The Request queue is the time an individual producer request sits idle in the request queue
2. The Request local is the time used by the broker to persists the record included in the producer request
3. The Response remote is 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`
4. The Response queue is the time the response sits in the response queue until it is picked up
5. Finally the Response send is the time it takes to send the response back to the producer

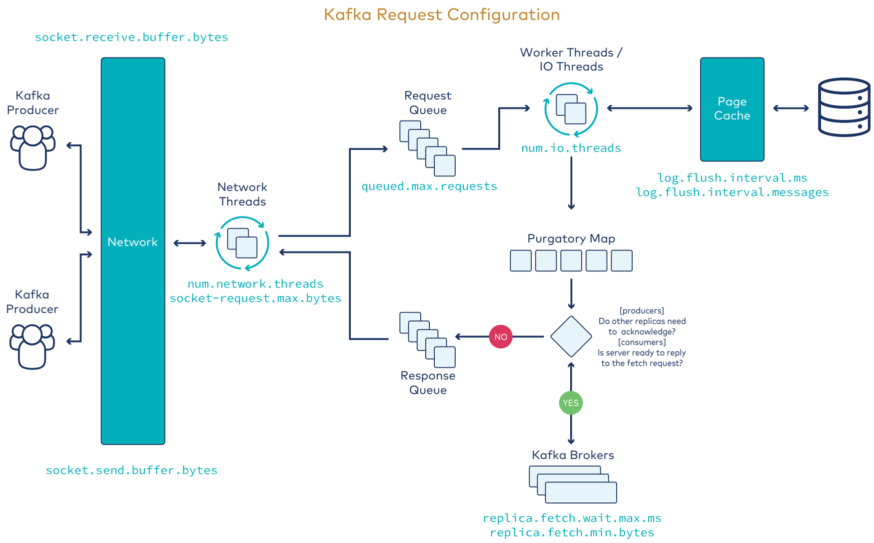
With that knowledge we can now start to think about how to optimize the broker for our particular scenario.

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
`socket.receive.buffer.bytes`	102,400	The SO_RCVBUF buffer of the socket sever sockets. If the value is -1, the OS default will be used
`num.network.threads`	3	The number of threads that the server uses for receiving requests from the network and sending responses to the network
`socket.request.max.bytes`	104,857,600	The maximum number of bytes in a socket request
`queued.max.requests`	500	The number of queued requests allowed before blocking the network threads
`num.io.threads`	8	The number of threads that the server uses for processing requests, which may include disk I/O
`log.flush.interval.messages`	2^63 – 1 (max long)	The number of messages accumulated on a log partition before messages are flushed to disk
`log.flush.interval.ms`	`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 `log.flush.scheduler.interval.ms` is used

Parameter	Default	Description
`replica.fetch.wait.max.ms`	500	max wait time for each fetcher request issued by follower replicas. This value should always be less than the `replica.lag.time.max.ms` at all times to prevent frequent shrinking of ISR for low throughput topics
`replica.fetch.min.bytes`	1	Minimum bytes expected for each fetch response. If not enough bytes, wait up to `replicaMaxWaitTimeMs`
`num.replica.fetchers`	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
`socket.send.buffer.bytes`	102,400	The SO_SNDBUF buffer of the socket sever sockets. If the value is -1, the OS default will be used.
`message.max.bytes`	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
`background.threads`	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

Mitigate High Latencies

Request queue	• increase the pool of I/O threads (`num.io.threads`)
Request local	• use dedicated, faster local disks (SSD) • tune `log.flush.interval.messages` & `log.flush.interval.ms`
Response remote	• make sure brokers are co-located • use high bandwidth network (1 GbE, 10 GbE) • use dedicated NICs for brokers • assure all brokers nodes are equal • do you really need `acks=all`?
Response queue	• use high bandwidth network (1 GbE, 10 GbE) • use dedicated NICs for brokers

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.

We should avoid clusters that span multiple data centers, even if the data centers are co-located 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.

Learning Objectives

Completing this lesson and associated exercises will enable you to:

- Name relevant broker metrics to monitor

Monitoring

Monitor the following resources:

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

We need to monitor our 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!

Optimization per Service Agreement

Optimize Latency

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

Optimize Throughput

- Nothing to do

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 Availability

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

Tuning brokers depends on which service agreement is essential for us. Remember we have identified four of them, maximum throughput, minimal latency, optimal durability and optimal availability. We cannot optimize for all at the same time. Let's look at each individually

- On the slide you see that to optimize throughput, there is really nothing much to do on the broker side.
- You can also see that to minimize latency we can increase the number of replica fetchers in followers can't keep up with the leader. The default value is `1`
- To optimize durability we have a list of settings. The most most important ones are the number of replicas and the minimum number of in-sync replicas.
- Finally to optimize for availability we have 3 parameters that need to be addressed. They are enable unclean leader election, set number of in-sync replicas to `1` and potentially define more than `1` thread per data directory for recovery.

Memory Usage

- `replica.fetch.max.bytes`: 1MB allocated per replicated partition

How much memory is used on a broker really depends on the number of partitions it handles. By default we allocate 1 MB of memory per partition. The size is defined by the parameter `replica.fetch.max.bytes`.

f: Confluent Auto Data Balancer

Description

Overview of Auto Data Balancer.

Learning Objectives

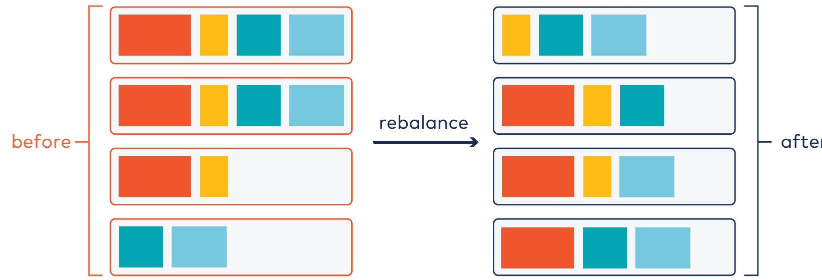
Completing this lesson and associated exercises will enable you to:

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

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



Unevenly distributed data among brokers can lead to sub-optimal overall performance of the cluster. How can this happen in the first place? One possibility is that we added additional brokers to the cluster. In this case the partitions are not redistributed automatically. Another reason could be that we have a partition algorithm on the producer side that does not spread the messages equally over all partitions of a topic.

To overcome this situation we can use the Confluent Auto Data Balancer (ADB). It can be used to:

- Shift data to create an even workload across the Kafka cluster
- Easily add and remove nodes from the Kafka cluster
- Rack aware algorithm rebalances partitions across the cluster in a way that optimizes availability

A good thing of ADB is that the traffic of the balancer can be throttled when data transfer occurs, and thus the other Kafka clients will not be negatively affected.

Confluent Auto Data Balancer (ADB)

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

On this slide we see a sample command using the ADB.

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.
```

On this slide we see a (simple) sample output.

g: Message Delivery Guarantees

Description

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

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

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**.

Kafka defines three different message delivery semantics as listed on the slide. They are:

- At most once
- At least once
- Exactly once

In the case of exactly once, even if a producer tries to resend a message, it leads to the message being written exactly once to the topic partition.

Idempotent Producer

Idempotent Producer == Exactly once per Partition

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

```
enable.idempotence=true
```

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 }
```

On this slide you see a code sample that uses Kafka transactions...

Lines 1-5 set up the producer by specifying the transactional.id configuration and registering it with the initTransactions API. After the producer.initTransactions() returns, any transactions started by another instance of a producer with the same transactional.id would have been closed and fenced off.

Line 7-10 specifies that the KafkaConsumer should only read non-transactional messages, or committed transactional messages from its input topics. Stream processing applications typically process their data in multiple read-process-write stages, with each stage using the outputs of the previous stage as its input. By specifying the read_committed mode, we can get exactly once processing across all the stages.

Lines 14-21 demonstrate the core of the read-process-write loop: we consume some records, start a transaction, process the consumed records, write the processed records to the output topic, send the consumed offsets to the offsets topic, and finally commit the transaction. With the guarantees mentioned above, we know that the offsets and the output records will be committed as an atomic unit.

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.

Of course the student is expected to mention that they're going to run Confluent ADB. They should also mention what exactly ADB does and whether or not some cluster down-time is to be expected.

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

09: Troubleshooting & Tuning Schema Registry

CONFLUENT
Global Education

Module Overview

This module contains one lesson:

1. Troubleshooting Schema Registry

a: Troubleshooting the Schema Registry

Description

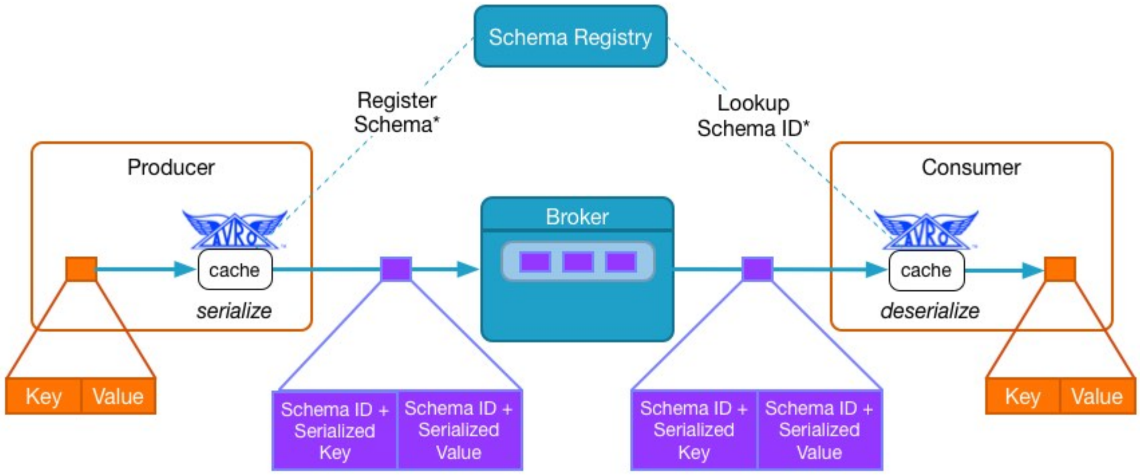
Review of schema registry and assorted issues that commonly arise with schema registry deployment.

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

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?"

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

On this and the following 2 slides we are going to look at common issues our customers have when using the Confluent Schema Registry. We have identified 3 categories of issues. They are "bad design", " inconsistent configurations" and "operational mistakes". Let's start with the category "bad design". I have listed 4 scenarios on this slide:

- Again and again customers try to co-locate their schema registry with other services of the Confluent Platform, mainly with brokers. The reason they do so because in principle SR needs very limited resources. 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.
- We also see often that customers run multiple instances of SR in their company. But 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.
- Sooner than later, customers have multi data center deployments for increased availability and globalization reasons. 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.
- For high availability, customers often run multiple SR instances in a cluster. Yet they do not deploy a virtual IP (VIP) in front of their SR cluster, which results in an extra burden on application developers and/or operators 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

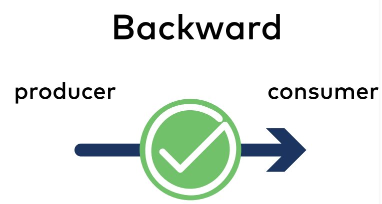
- Cannot change field data type
  - Exception - data type widening is allowed

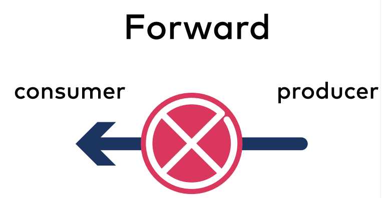
- You cannot change a field's data type. If you have decided that a field should be some data type other than what it was originally created using, then add a whole new field to your schema that uses the appropriate data type.
- schema evolution allows widening types, e.g. `int → string` or `int → long`

Example

Schema V1 fields

```
"fields":
[
    {
        "name": "firstname",
        "type": "string"
    },
    {
        "name": "lastname",
        "type": "string"
    },
    {
        "name": "age",
        "type": "int",
        "default": "-1"
    }
]
```





Schema V2 fields

```
"fields":
[
    {
        "name": "lastname",
        "type": "string"
    },
    {
        "name": "age",
        "type": "int",
        "default": "-1"
    },
    {
        "name": "hobby",
        "type": "string",
        "default": ""
    }
]
```

- 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>
```

You can test the compatibility of a new version of a given schema using the REST endpoint of the Schema Registry shown on the slide. This test considers your compatibility rule.

There is also 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
• `TopicNameStrategy` (default)	• `key.subject.name.strategy`
• `RecordNameStrategy`	• `value.subject.name.strategy`
• `TopicRecordNameStrategy`	

- `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.

To round up this module about troubleshooting the SR please try to answer the question which schema compatibility rule makes most sense for a particular use case you care.

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

Lab: Troubleshooting the Schema Registry

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

Refer to the Exercise Guide

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

10: Troubleshooting & Tuning Producers

CONFLUENT
Global Education

Module Overview

This module contains two lessons:

1. Troubleshooting Producers
2. Tuning Producers

a: Troubleshooting Producers

Description

Producer metrics, common troubleshooting scenarios, librdkafka producer specifics.

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

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

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
```

On this slide I have a few common issues found by our support engineers when troubleshooting producers. Let's look at them in some more detail:

- **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`

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 Kakfa 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?

To round up this module about troubleshooting a producer please try to answer the question shown on this slide.

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?

Further Reading

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

Lab: Troubleshooting Producers

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

Refer to the Exercise Guide

Lab Troubleshoting: 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.

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

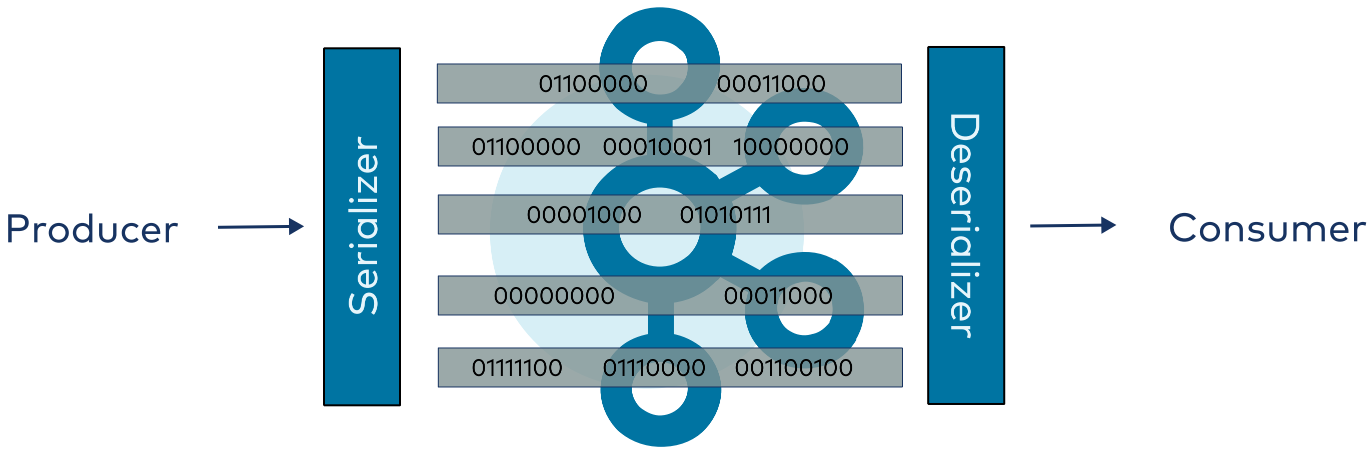
Out of the box data formats

- 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/examples/streams/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**

We have already briefly touched the reasons on why to use the AVRO data format for events sent to, or consumed from Kafka. Producers are the once generating the data and thus it makes a lot of sense to talk about the right data format in this module which is about tuning a producer.

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
- AVRO 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.
- AVRO 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, i.e., Schema

evolution

- Finally we need to add, that the usage of AVRO makes most sense in the Kafka eco system when combined with the use of the Confluent Schema Registry.

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

---

Although the effects of choosing the "right" number of partitions are mostly downstream, we still need to mention it here, since it is the producer who ultimately produces the data and distributes the events into the respective partitions.

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.

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**

Producers write to Kafka brokers. But they're not alone. Other processes compete for available resources, specifically for network bandwidth. Thus we need some kind of governance that keeps things in check.

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 separately from the outgoing quota. 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 we don't know how many unique keys there will be. However, we can configure a compacted topic to have a hard retention policy, if we can accept that some unique keys may be lost. If we do not set a hard retention policy, then the disk space used may be impacted by the users of the cluster.

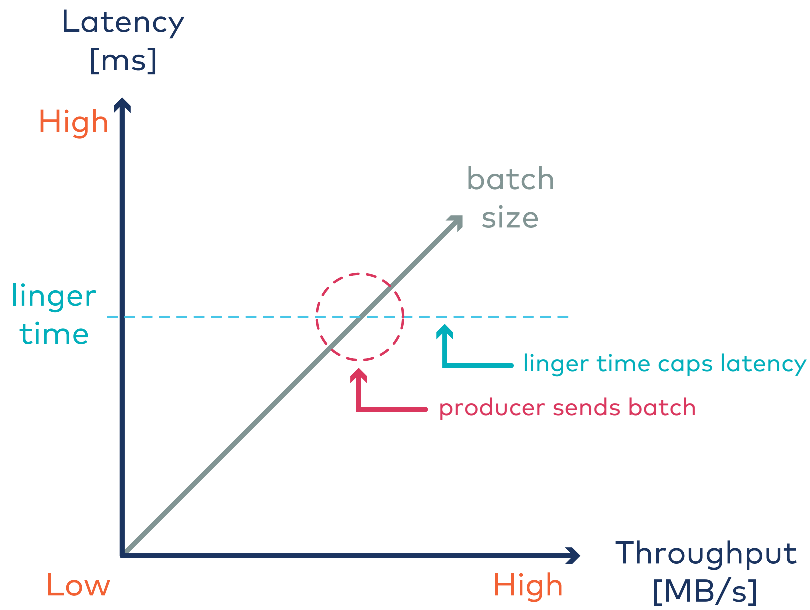
Critical Configurations

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

There are some configuration settings that are critical for the performance of a producer. I have listed them on the slide. Let's go through the list:

- The parameter `batch.size` affects how much batching can happen
- `linger.ms` is important for slow producer that is not filling up the message batch quickly enough
- `buffer.memory` 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 for a time limited by `max.block.ms` or throw an exception thereafter.
- The `compression.type` together with the batch size can massively increase the throughput
- `max.in.flight.requests.per.connection`: can affect the ordering of messages, specifically when retries in the message sending process happen.
- The `acks` setting - 0, 1 or all - affects the durability of messages sent by the producer

Batch Size



Let's talk a bit more about the 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. On the slide I have tried to visually represent the concept of batching in Kafka

- Instead of the number of messages, the batch size is measured in total bytes. That means batch size controls how many bytes of data to collect, before sending messages to the Kafka broker. So, without exceeding the available memory, set it as high as possible. Make sure the default value is 16384 (16kB).
- However, the reserved buffer might never get full, if we increase its size. On the basis of other triggers, such as linger time in milliseconds, the Producer sends the information eventually.
- In order to buffer data in asynchronous mode, the parameter `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.
- There is also a parameter buffer memory which represents how much memory to allocate overall by the 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 exposed by 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

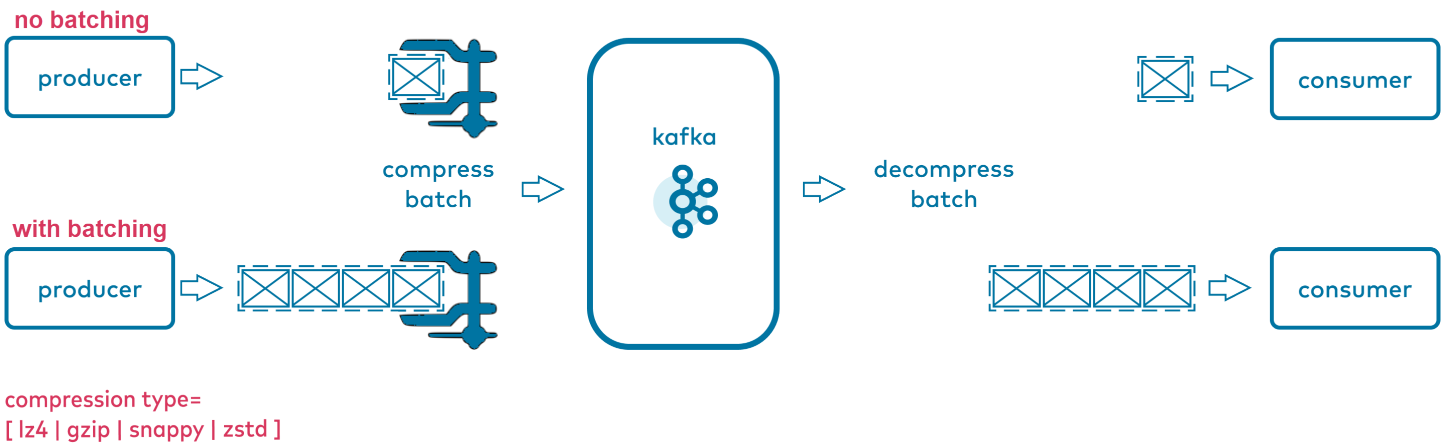
Regarding the **user processing time**, this is the time spent in user code, specifically executing user callbacks for acknowledgements.

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.

On this slide I have listed a few metrics that are on a per topic level. These are the ones you want to look at when tuning a producer.

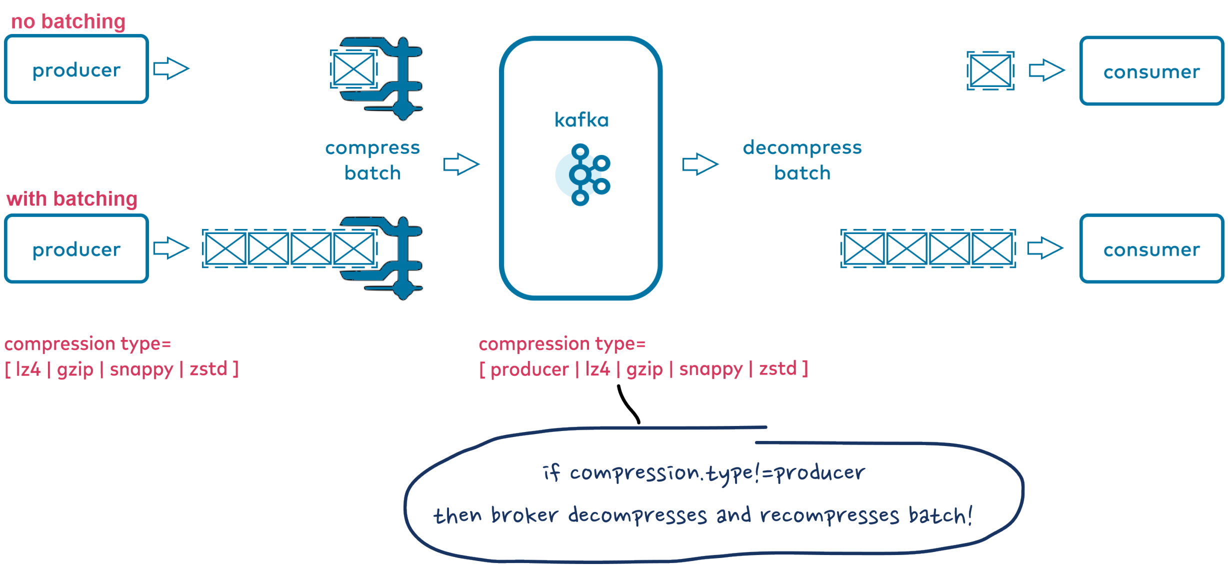
Producer Compression



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.

INFO: On September 2016, Facebook announced a new compression implementation named **ZStandard** (zstd), designed to scale with modern data processing environment. With its great performance in both speed and compression rate, lots of popular big data processing frameworks are supporting ZStandard.

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

Since the sending of messages by a producer is asynchronous we can have multiple concurrent requests in flight. Under normal circumstances this leads to better throughput, but can cause messages being written to the brokers log out of order, in the case that retries in the send process are happening. In many cases this is not an issue. It is also worth noting that when using **idempotent producers** message ordering is guaranteed on a per topic partition level.

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`)

Let's now look at the four scenarios we introduced earlier, maximum throughput, minimal latency, durability and availability, and then discuss the optimal settings used in each of them. We cannot optimize for all 4 service requirements at the same time. On this and the following slide we give recommendations for cases where one of the requirements clearly trump the others.

Let's start with the scenario of maximum 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

The next scenario is lowest Latency:

- Set `linger.ms=0` but note that this setting 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` is good in most cases, but not always. It can depend on the CPU vs I/O cost

Some feedback from the Kafka Core team regarding what I just said:

- 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

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

The next scenario is optimal durability. For this we set the replication factor to 3 or higher, acks to all and retries to 1 or more, enable idempotence, and finally max in-flight requests to 1. Note though that `max.in.flight.requests.per.connection=1` is only needed when not using an idempotent producer. Furthermore `min.insync.replicas` (a broker setting defined per topic) is commonly defined together with `acks=all`

In regards to the availability scenario the settings discussed have no meaning. Availability mostly concerns brokers.

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`

Whenever we want to test the maximum achievable performance of producers we can use the Kafka tool `kafka-producer-perf-test`, which allows us to stress test our environment with various settings. On the slide I have given a typical code snippet on how to use this tool. We can see how I can set the size of each record sent, the throughput and various other producer specific properties discussed earlier such as e.g. compression type.

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`

When tuning a non-Java producer that is based on `librdkafka` we should consider the following

- It is best to **batch multiple records** and send them in one go to achieve higher throughput.
- Generally one can say that to tune your producer use the same settings as for an equivalent Java producer
- Important settings in this context are the ones listed on the slide

```
// 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. THere are three questions that help the learners to assess their understanding of the material taught in this module and the associated hands-on lab.

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

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

11: Troubleshooting & Tuning Consumers

CONFLUENT
Global Education

Module Overview

This module contains two lessons:

1. Troubleshooting Consumers
2. Tuning Consumers

a: Troubleshooting Consumers

Description

Consumer lag troubleshooting. Reading and resetting offsets. Rebalancing.

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

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.

Troubleshoot Consumer Lag: (1) Understand structure of environment

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

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?

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               2
consumer-1-029af89c-873c-4751-a720-cefd41a669d6   /127.0.0.1
consumer-1
my-topic                       1          2               3               1
consumer-1-029af89c-873c-4751-a720-cefd41a669d6   /127.0.0.1
consumer-1
my-topic                       2          2               3               1
consumer-2-42c1abd4-e3b2-425d-a8bb-e1ea49b29bb2   /127.0.0.1
```

From the above example output, we can 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 their 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
```

Some times it is necessary to reset the offsets of a consumer group. There is a tool in the Kafka toolset specifically for this purpose. It is called `kafka-consumer-groups`. Using this tool to reset offsets has the benefit that it requires no code or configuration changes on the consumer side. Why would we need to reset the offsets in the first place? I have given two possible reasons on the slide. You might have others that come to mind. We can reset the offset to the beginning or any valid offset. We can also using a timestamp and reset the offset to the first event equal or younger than that point in time. The method of doing so is shown in the code snippet on the slide

Read Consumer Group Offsets

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

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

The same tool can also be used to get the current offset for all topics and partitions consumed by a given consumer group. A sample command and its result is shown on the slide.

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 related metadata.

There are reasons we may want to read this topic. Two of them are:

- We want to see when a consumer last committed an offset, and what the offset is
- We want to know which broker is running the group coordinator for a consumer group

There are 2 ways on how we can read the content of this topic.

- The first one requires that a broker is online. We can then use the Kafka console consumer to consume from the topic like we would do for any other topic. The important detail to notice though is that we are using a special formatter to decode the messages in this topic.
- The second way we can use when brokers are offline. In this case we can use the `kafka-dump-log` tool to get access to the data. Note that the topic by default has 50 partitions and we potentially need to parse all 50 corresponding files for what we are looking for.

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
```

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 pools 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?

To round up this module try to answer the question on the slide.

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

Lab: Troubleshooting Consumers

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

Refer to the Exercise Guide

b: Tuning Consumers

Description

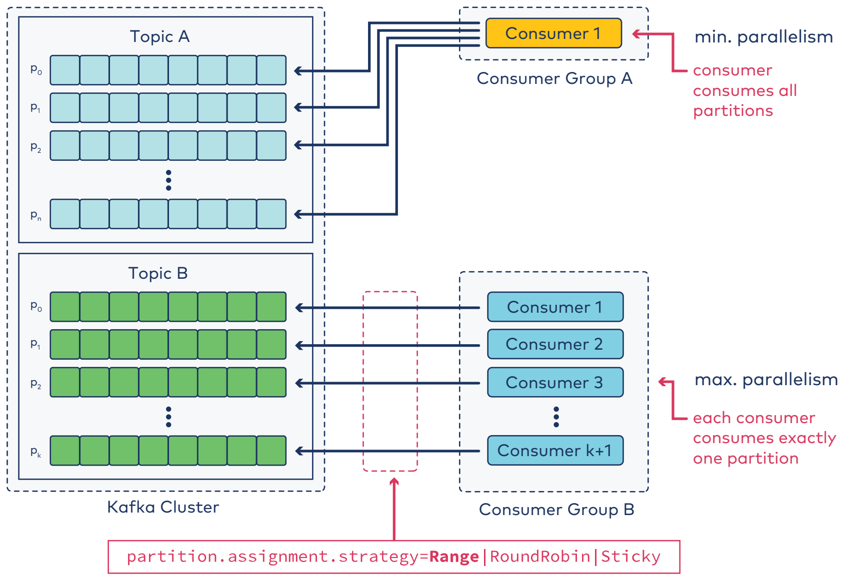
Consumer/partition paralellism. Fault detection and rebalancing. Fetch requests. Details specific to librdkafka consumers.

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

Maximize Parallelism



On this slide I have a graphic depicting how consumer groups consume data from Kafka:

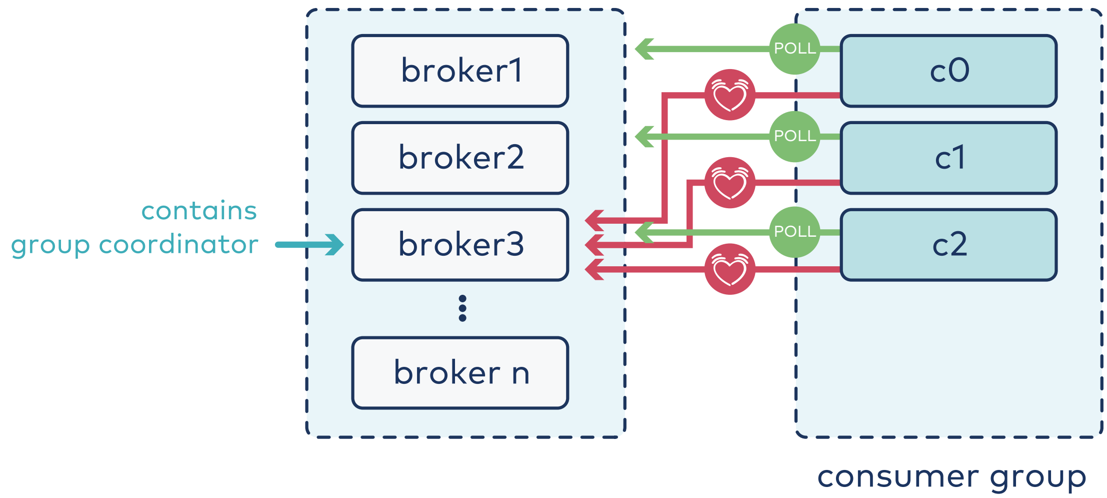
- Partitions of a topic are assigned to a consumer **as a whole**
- Each partition is only consumed by a single consumer of a given consumer group
- A consumer can consume multiple partitions
- The **partition assignment strategy** determines which partition is assigned to which consumer
- We have 3 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!

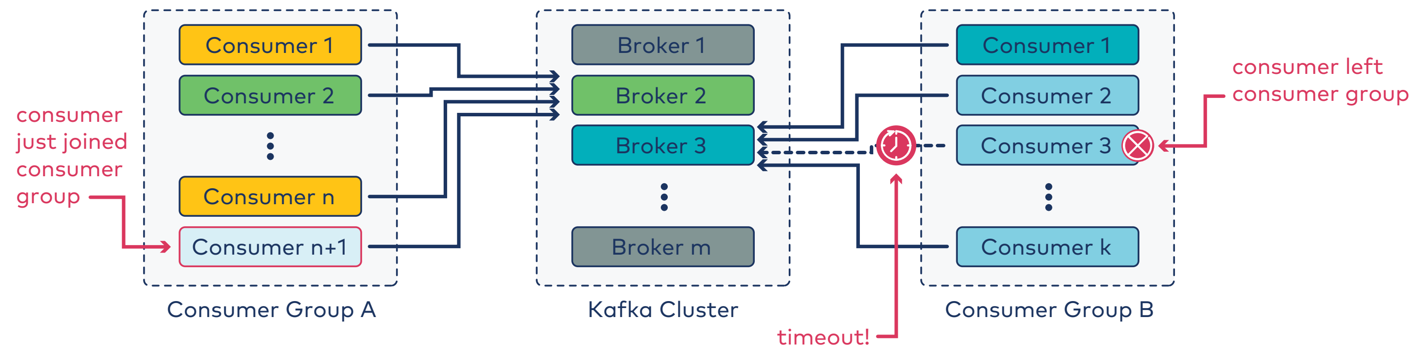
Consumer Liveness



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 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 gracefully shutdown.

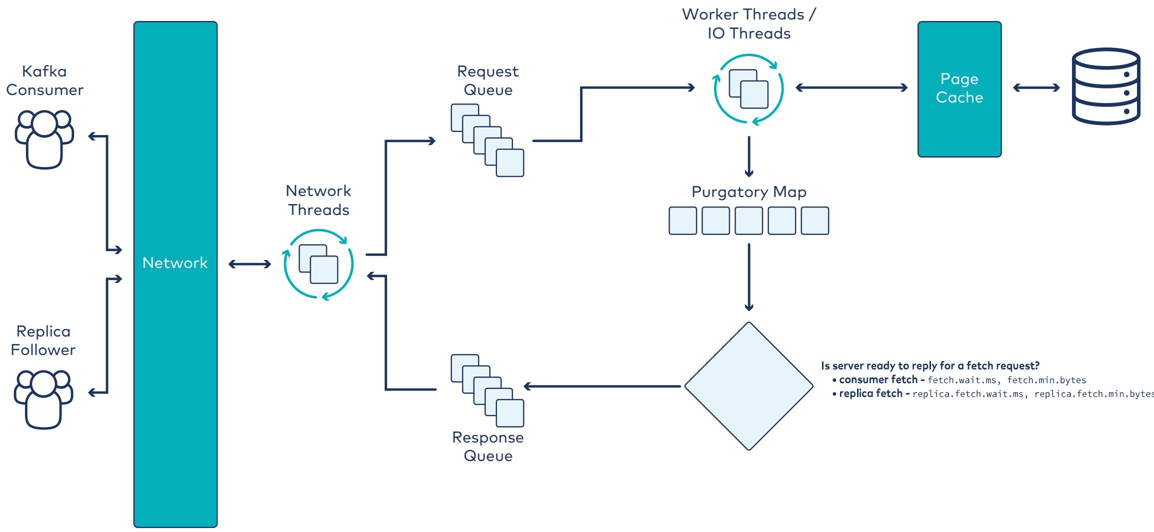
On the graphic we see a **consumer group A** that just got a new member. The group coordinator (broker 2 in this case) will trigger a reassignment. We also have a **consumer group B** which just lost a member and its group coordinator (broker 3) will notice this due to the fact that there is no more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

Please note that 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 identify 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.

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-retriable 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

On this slide I have listed a few metrics that are important to monitor when tuning our consumers. These metrics indicate the lag of the consumer, the frequency of fetching, the fetch duration and the throughput.

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.

ℹ	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`

When tuning consumers that use the REST Proxy the 3 hints on the slide apply

- As already mentioned in the module about tuning the producer, we should reuse the session to the REST proxy and not create a new one for every pull
- 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` defines 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` is 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.)

Review

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

To conclude this module please try to answer the question on the slide.

Sample Answer:
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)

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

Lab: Tuning Consumers

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

Refer to the Exercise Guide

12: Troubleshooting & Tuning Streams Apps

CONFLUENT
Global Education

Module Overview

This module contains two lessons:

1. Troubleshooting Streams Apps
2. Tuning Streams Apps

a: Troubleshooting Kafka Streams & ksqlDB Apps

Description

Common streams apps problems. Metrics. Health checks.

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

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.

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 appliy 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

Now we're going to concentrate a bit on ksqlDB. First we troubleshoot situations where there is no data produced by our ksqlDB queries. What can possibly be the root cause? Let's see:

- First, the source topic of the query does not contain any data. We can double check that with the `print '<topic-name>' from beginning;` command
- Second: no new data is arriving into the source topic. We can test this with the same print command, just leaving away the `from beginning` part
- Third, ksqlDB tries to consume data from a later offset than 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. We can reset the offset by setting the variable `auto.offset.reset` to the value `earliest`.
- Fourth, you might have defined a where clause whose predicate does not match any data
- Lastly, there may be some issues with the deserialization of messages from Kafka. We need to check the logs of ksqlDB to find out more about this.

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 show 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":"8MvZ8QmPRYeluBTYOiZk_Q","ksqlSer
viceId":"
```

The ksqlDB REST API supports a “server info” request at http://<server>:8088/info. The result of such a call you can see on the slide.

Troubleshooting RocksDB

Issue	Solution options
RocksDB performance appears low	• Use **SSDs** • Add **Cores**
RocksDB's file sizes appear larger than expected	• Use Linux `du` tool • Check for **Write Amplification**
Application memory utilization seems high	• 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?

To conclude this module please try to answer the question why a stateful Kafka Streams applications needs much more memory than a stateless app even though they're working on the same input topic?

Sample answer:
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/2I1hOVA
- 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

Lab: Troubleshooting Kafka Streams & ksqlDB Apps

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

Refer to the Exercise Guide

b: Tuning Kafka Streams & ksqlDB Apps

Description

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

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

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:

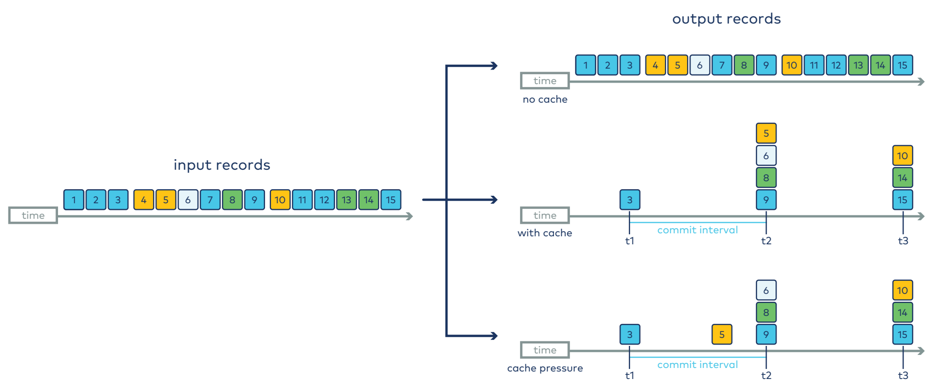
- The parallelism of a Kafka Streams application is primarily determined by how many partitions the input topics have. For example, if your application reads from a single topic that has ten partitions, then you can run up to ten instances of your applications. You can run further instances, but these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

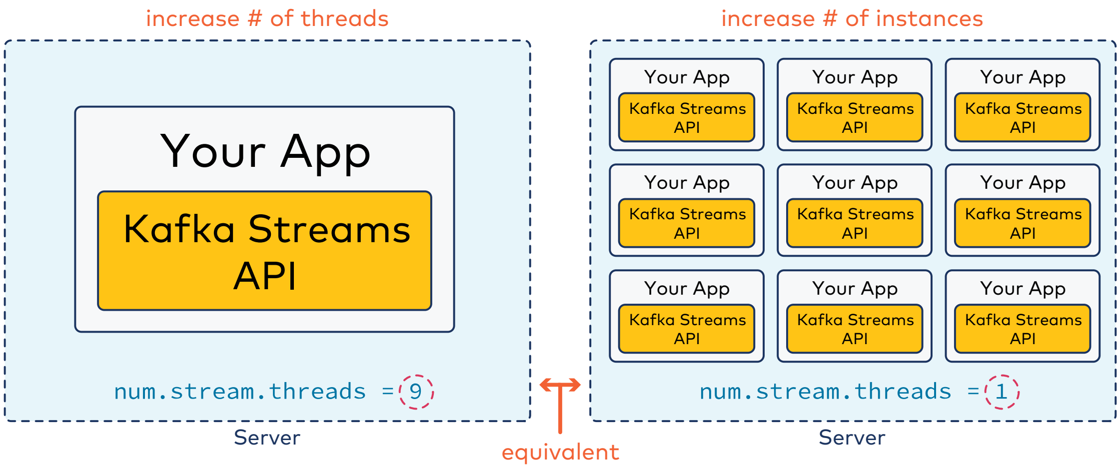
Memory Management



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/current/streams/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



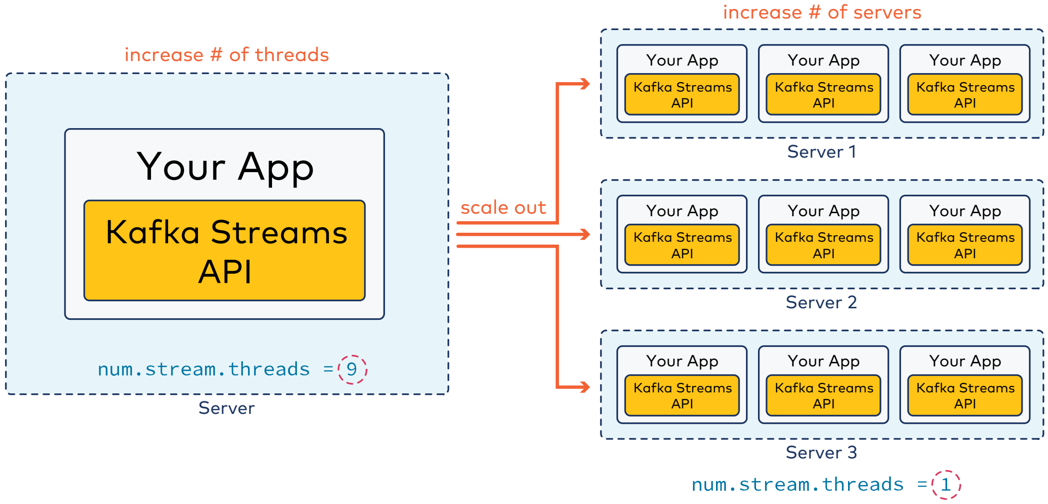
In a Kafka Streams or ksqlDB application task placement really matters. Increasing the number of partitions, and with it tasks, increases the potential for parallelism, but we must still decide where to place those tasks physically. There are two options, scale up or scale out.

We can 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. We 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.

Generally though, I would prefer running multiple instances in parallel than only a single instance with many threads. The former is more robust. In case one task crashes only a single instance goes down and all the others continue to work, whilst in the latter case all tasks are affected.

Scaling up has its limits though. In the end all stream. Tasks have to share the available CPUs, memory, IO and NICs. Also the more CPUs and cores as well as memory a server has the more expensive it is. And the cost does not scale linearly but rather exponentially.

Task Placement - Scale Out



The second option is to scale out, by spreading the tasks across more than a single machine or VM. This is useful when the application is network, memory or disk bound, or if a single server has a limited number of CPU cores. This variant is most useful in the cloud where it is easy and fast to provision additional VMs.

Stateless versus Stateful

Stateless Applications:

- CPU & Network are key

Stateful Applications:

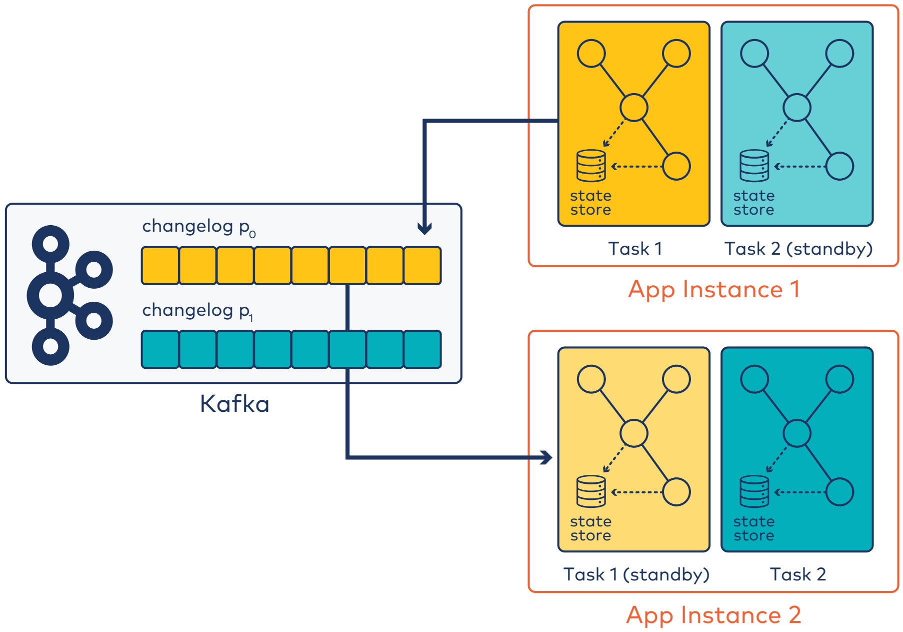
- **Memory**: High Performance
- **Local Disks**: Fast Queries
- **Kafka**: Fault Tolerance
- **Standby Replicas**: High Availability

When dealing with stateless Kafka Streams or ksqlDB applications then 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.

When dealing with stateful Kafka Streams & ksqlDB applications on the other hand, then we need to monitor another two resources, local disks and memory. These are applications that perform aggregates and joins.

- Kafka Streams & ksqlDB keep their local state in memory for highest possible performance. Thus we need plenty of RAM on those instances to avoid to much disk IO.
- Each Kafka Streams or ksqlDB instance uses local storage to store the intermediate results resulting from joins and aggregation functions. Fast local disks such as SSDs are important for fast queries.
- Local state from each Kafka Streams or ksqlDB instance is save to Kafka to provide fault tolerance
- To improve availability during failure, Kafka Streams tasks can be replicated to standby replicas by setting the parameter `num.standby.replicas` to a value greater than zero. This helps to decrease the time it takes to rebalance the work of a failed instance, since it can be simply transferred to the standby replica which already has (most) of the local data needed to take over

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

---

As mentioned on the previous slide, Kafka Streams and ksqlDB allow us to define standby replicas. They are useful to improve the availability of our streaming applications.

- Let's first look at what happens in 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 as follows:
  - 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.
- Let's look at ksqlDB: Since under the hood a ksqlDB application is really a Kafka Streams application the same applies in regards to standby replicas. In case of a ksqlDB node failure the failover to remaining ksqlDB instances is much quicker since the local state is already available on the target nodes where the tasks are going to be reassigned to.

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

ksqlDB Tuning

- Monitor **Consumer lag**
- Decrease threads for cheap queries
- Add resources to increase throughput
- Add standby instances for faster fail-over

How do we know when to scale a ksqlDB app? First and foremost we should monitor the consumer lag:

- If ksqlDB cannot keep up with the production rate of our 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 our topics and add resources if we observe that the lag is growing.
- Confluent Control Center is the recommended tool for monitoring the consumer lag.

Now if we have mixed workloads on our ksqlDB cluster, that is apps consisting of multiple queries, of which some may feed data into others in a streaming pipeline, then

- We need to monitor the consumer lag of each query's input topic
- Unfortunately ksqlDB currently does not have a mechanism to guarantee resource utilization fairness between queries.
- Thus a faster query like a project or filter query may "starve" a more expensive query like a windowed aggregate if the production rate into the source topics is high.
- If this happens we will observe a growing lag on the source topic for the more expensive queries and very low throughput to their sink topics.

We can fix this situation by using either of the following methods:

- We tune the cheaper queries to consume less CPU by decreasing the number of Kafka Stream threads for that query. Unfortunately at this time we can only do this In interactive mode
- Or we can add resources to reduce the per-CPU usage of the cheaper queries, which in turn will increase the throughput for the more expensive queries.

Finally, ksqlDB server nodes can fail and if the failover time must be small then we should consider adding standby nodes to the cluster. These standby nodes always sync the state of the leader node and thus can take over immediately if the leader fails.

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?

To conclude this module please try to answer the question on the slide. 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/documentation/streams/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/current/streams/sizing.html
- KSQL Capacity Planning: https://docs.confluent.io/current/ksql/docs/capacity-planning.html

Lab: Tuning Kafka Streams & ksqlDB Apps

Please work on **Lab 12b: Tuning Kafka Streams & ksqlDB Apps**

Refer to the Exercise Guide

13: Troubleshooting & Tuning Kafka Connect

CONFLUENT
Global Education

Module Overview

This module contains two lessons:

1. Troubleshooting Connect
2. Tuning Connect

a: Troubleshooting Kafka Connect

Description

Connect error handling framework. Other Connect issues.

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

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

> 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/MU2fdyQHJ3nKfYW6v1DLgkz1l

Fail Fast Scenario

Why?	How?
• **Poisoned messages** i.e., cannot be processed • Source/target system unavailable	Confirguration settings:

```
# 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
```

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.

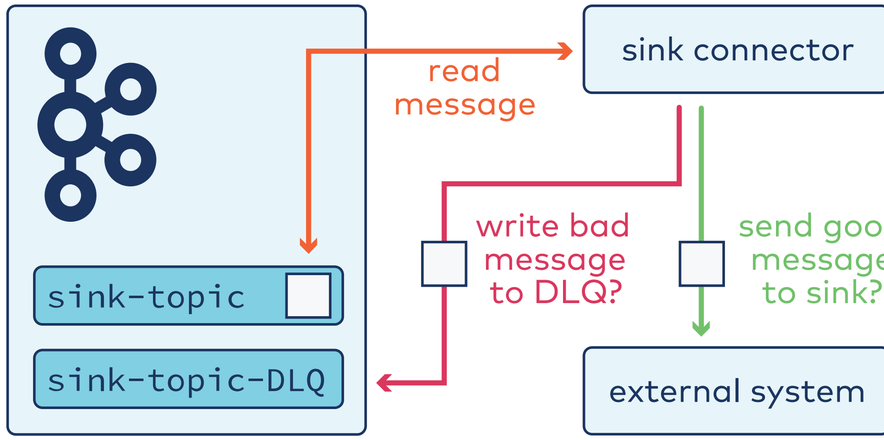
The second situation is **source or target systems are unavailable**:
A situation that could arise from time to time in our deployment is when one of the systems the connectors are pulling data from or pushing data to becomes unavailable. This could be for multiple reasons, e.g., this target system is 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.

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
```

Monitoring for Errors

Metric/Attribute Name	Description
`total-record-failures`	Total number of failures seen by this task.
`total-record-errors`	Total number of errors seen by this task.
`total-records-skipped`	Total number of records skipped by this task.
`total-retries`	Total number of retries made by this task.
`total-errors-logged`	The number of messages that was logged into either the dead letter queue or with Log4j.
`deadletterqueue-produce-requests`	Number of produce requests to the dead letter queue.
`deadletterqueue-produce-failures`	Number of records which failed to produce correctly to the dead letter queue.
`last-error-timestamp`	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 available to inspect the error management, and since Apache Kafka 2.0 we can check for the list show on the slide. The **MBean** to look for these values is under `kafka.connect` → `task-error-metrics`.

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

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.

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

⚠	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/loggers` 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?

We do not have a lab for this module. To conclude the module try to answer the question show here on the slide.

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.

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

b: Tuning Kafka Connect

Description

Connect best practices.

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

Monitoring Connect

- Monitoring helps in sizing & troubleshooting
- Workers have embedded producer and/or consumer

Metrics	MBean Name
Connector Metrics	`kafka.connect:type=connector-metrics,connector=<connector-ID>`
Task Metrics (Common, Source & Sink)	`kafka.connect:type=task-metrics,connector=<connector-ID>,task=<task-ID>` `kafka.connect:type=source-task-metrics,connector=<connector-ID>, task=<task-ID>` `kafka.connect:type=sink-task-metrics,connector=<connector-ID>, task=<task-ID>`
Worker Metrics	`kafka.connect:type=connect-worker-metrics`
Worker Rebalance Metrics	`kafka.connect:type=connect-worker-rebalance-metrics`

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

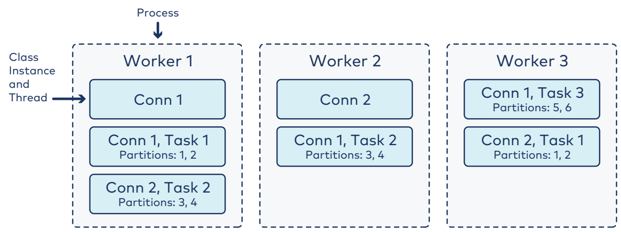
Can chain many transformations!

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. If you want to apply more than one transformation then you can chain them. We can e.g. define transformations to:

- Create a message key
- Reduce the amount of data by dropping fields
- Route messages
- And more

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?

To conclude this module please try to answer the question shown on the slide. It is about importing a reduced set of data.

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

Lab: Tuning Kafka Connect

Please work on **Lab 23a: Tuning Kafka Connect**

Refer to the Exercise Guide

Conclusion

CONFLUENT
Global Education

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

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

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

Thank You!

Appendix: Additional Problems

CONFLUENT
Global Education

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.

Instructor note: These problems are provided as additional enrichment. All have been used successfully in virtual deliveries of this course.

It works well to share a problem like this as students arrive on days other than the first as review. It gets the students thinking about content right away and reviewing thoughts from before and revisiting them. In virtual deliveries, it can be very effective to put students in randomly-chosen breakout rooms to discuss with a few classmates before a whole-class discussion. Pointing students to the student handbook during their discussions can be helpful in nudging them without "giving away answers."

See the notes on each for prerequisites / suggested placement.

Some have virtual whiteboarding aids available. Internal instructors: see the Google Drive. External instructors: contact Doug, for now.

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

Instructor notes:

Recommended Placement: Day 1 Arrival Warm-Up

Specific Lessons to Point Students To: ADM course

Additional Notes:

- On producer: message goes to buffer on producer, wait for buffer to be flushed for this partition. On broker, produce request received by network threads, put into request queue, received by worker/IO threads
- The reason for "`linger.ms` has been increased from its default to 5 minutes" is to make batching happen.
- The default for `buffer.memory` is 32 MB, hence the 31.9. This means the buffer will fill up…
- …and `max.block.ms` kicks in to allow for some time for space to become available in the buffer.

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

Instructor notes:

Recommended Placement: Day 1 Arrival Warm-Up

Specific Lessons to Point Students To: ADM course

Additional Notes:

1. One broker will contain leader replica. Two other brokers will contain follower replicas.
2. Cannot have more replicas than broker. Kafka will give an error. It does not make sense to have a backup copy on the same machine as the one with the main copy; the follower could never become leader, so we would be wasting space.
3. A leader election would be triggered. As $p_{1,F1}$ is an in-sync replica and $p_{1,F0}$ is not, the Controller would select $p_{1,F1}$ as the new leader.

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)

Instructor notes:

Recommended Placement: Day 2

Specific Lessons to Point Students To: M2 (Generic Monitoring), M3 (Monitoring with Confluent Control Center)

Additional Notes:

a. Offsets can range from 0 to 21

b. 21

c. 21 - 12 = 9

d. Could go via Consumers or Topics view

e. How many orders have landed on this kitchen, but whose order slips have not yet been printed

f. Inventory tracking

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)

Instructor notes:

Recommended Placement: Day 2 Warm-Up

Specific Lessons to Point Students To: M2 (Generic Monitoring)

Additional Notes: This *could* be an issue, because, probably, to do anything about it, e.g., ADB, we'd need additional temp space. The recommendation is to alert at 60% capacity.

Problem E: Assessing Disprecancies 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} are 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?

Instructor notes:

Possibilities:

- The setting of 2 hours could have been a static broker setting on broker 102 but it had been overridden with a dynamic broker setting.
- The setting of 2 hours could have been a static broker setting on broker 102 but it had been overridden with a dynamic cluster-wide default setting.
- The setting of 2 hours could have been a broker setting on broker 102, static or dynamic. Maybe the active segments on broker 103 that have been around longer than 2 hours are partitions of topics where the topics have segment rolling settings that differ from the 2 hours.
- The setting of 2 hours could have been a cluster-wide default but broker 102 has a broker-specific dynamic override that is greater.

Sidenote: It's intentional that the gender pronouns for the three referenced colleagues are different—because illustrating diversity is the right thing to do!

Recommended Placement:

Specific Lessons to Point Students To:

Additional Notes:

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…

a. Conceptually, what should you do and why?

b. 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.

a. What could be some causes of this problem?

b. 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)

Instructor notes:

Recommended Placement: Day 3 Warm-Up

Specific Lessons to Point Students To:

1. 4a (Troubleshooting Intro)
   10a (Troubleshooting Producers)
2. 11a (Troubleshooting Consumers)
3. 3a (Monitoring with Confluent Control Center)
   11a (Troubleshooting Consumers)

Additional Notes:

Possible things to consider for 1:

- `acks`
- `min.in.sync.replicas`
- `retries`
- having a callback
- listeners on brokers
- ACLs for producer
- log files

Command for 2b:

```
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
```

Ideas for 3a:

- more consumers than partitions
- range
- partitioning strategy
- skewed input key distribution with default partitioner, quotas