Effectiveness of Rebalancing a Kafka Cluster

Bhubanesh Mishra (200495086)

School of Computing

Newcastle University

Supervisors

Dr. Paul Ezhilchelvan (Newcastle University)
Dr. Simon Woodman (Red Hat Inc)
Dr. Thomas Cooper (Red Hat Inc)

In partial fulfillment of the requirements for the degree of

MSc in Cloud Computing

August 20, 2021

DECLARATION I, Bhubanesh Mishra, hereby declare that this thesis entitled *Effectiveness of Rebalancing a Kafka Cluster* is a bonafide record of research work done by me for the award of MSc in Cloud Computing from Newcastle University, UK. It was never submitted to any university or entity before, in part or in full, for any degree, diploma, or other qualification..

Bhubanesh Mishra

Signature: _____

Abstract

Over the past couple of years, Apache Kafka has been the market leader in streaming, queueing and messaging services. As with every complex application, it also has its fair share of daily operational challenges. Past studies have been carried out to mitigate those challenges, mostly involving rebalancing load across the Kafka cluster using a tool called Cruise Control. In this study, we attempt to find a solution regarding the performance impact for various rebalance configuration. At first, we designed an example application and test infrastructure. We then carry out a series of experiments using this test environment. We perform multiple data analyses on the resulting metrics. These analyses show that doubling the default parameter values of the rebalance configuration will yield the best compromise between performance impact and rebalance length.
Keywords: Apache Kafka, Cruise Control, Data Analysis

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the modern 21st-century business environment, the amount of data being generated daily is ever-increasing. These data generally vary among user activities such as click views, logins, search view, system metrics corresponding to CPU, memory usage, I/O, Disk Usage and operational metrics such as latency, errors to name a few [1]. These large data that is gathered generally forms the critical components for various analytics, which in turn drives the modern IT landscape to perform at a higher efficiency level. One of the major requirements for these data is real-time usage and analysis, which can be very complex and challenging because of the sheer volume of the data obtained. The best example for this is Facebook that gathers data close to 6TB per day of various user activity events [2], which is used for its data analysis. Historically, there have been many enterprise messaging systems that have helped in this data gathering and analysis. However, in the last few years, Apache Kafka has been in the limelight as the go-to messaging, streaming and queueing system due to its various performance benefits over the traditional messaging and queuing system [3]. Kafka has many use cases such as activity tracking, messaging, application log gathering, metric collection on multiple sources, and integration with big data platforms such as Spark and Hadoop [4]. Kafka is a highly scalable, durable, fault-tolerant, and high performing publish-subscribe messaging system written in Scala [5]. The architecture consists of producers, consumers, brokers, zookeepers, topics and partitions [1].

Topics are particular steam of data similar to a table in a database. Topics are split into *partitions* that store the data with an incremental id called offset. These partitions are distributed over multiple brokers with high throughput [6]. Kafka servers are known as *brokers*, and a collection of these servers form a *cluster*.Only

one broker can be the leader for a given partition at any given time, and that broker will receive and handle data for that partition.The remaining brokers will just function as a replica and sync the data. *Producers* generate a stream of records and write to the topics. *Consumers* subscribe to these topics and retrieve the data using the offset. For parallel processing of data across multiple partitions in a topic, consumers can be grouped in the form of consumer groups. A *zookeeper* manages and coordinates all the brokers in a cluster [7].

As with any complex application, operating at high performance comes with its fair share of challenges. In this case, the daily operational challenges encountered on a Kafka cluster are mostly related to system maintenance. These challenges can range from a broker failure, new topic creation, balancing the workload using partition reassignment to massive operational loads [8]. Generally, these maintenance periods are time-consuming and labour intensive. Additionally, there is also a chance of frequent infrastructure downtime during maintenance.

1.1 Motivation

Since this research project is an industry-based project carried out in collaboration with Red Hat, they had a similar problem statement, as mentioned earlier, related to the daily Kafka system maintenance. Theirs is primarily related to the operational part of Kafka, involving the rebalance mechanism. For a balanced Kafka cluster, the topics should be evenly distributed amongst the brokers in an ideal scenario. However, in practicality, the cluster could become unbalanced due to faulty design of the system architecture, unpredictable workloads, rolling upgrades, application scale up/scale down, or addition/removal of brokers.

Various tools exist to rebalance a Kafka cluster by moving replicas to underutilised brokers or switching which topic replica is the master [9]. Moving replicas can be expensive and it can take hours to migrate data between brokers [8]. The cluster operates at reduced capacity during this time because the rebalance utilises the network bandwidth and disk I/O [8].

Therefore, the motivation for this work is to explore one of these rebalance tools, namely, Cruise Control [10] and try to record the performance impact of this tool on a running Kafka unbalanced cluster during the rebalance phase. Also, suggest appropriate tuning parameters that can help trade performance impact against maintenance window duration.

1.2 Research Aim and Scientific Objectives

This research aims to explore two aspects of using Cruise control to rebalance a Kafka cluster which will be running on Kubernetes. They are as follows:

1. What are the ideal values for tuning a rebalance of an unbalanced Kafka cluster to trade off speed versus performance impact?
2. Look at the effects of rebalancing on a running Kafka cluster, which would involve checking the duration it will take for a rebalance. The impact this rebalancing will have on the current workload.

In order to achieve these aims, the scientific objectives were:

1. An example application was created that was hosted on a test rig consisting of the infrastructure, libraries, installable, and monitoring tools on which the experimental study was carried out.
2. A balanced Kafka cluster was created, and essential metrics were gathered that acted as the baseline for the entire experiment. This activity was followed by purposefully unbalancing the cluster by following the real-world scenario of traffic distribution on the message production for the Kafka cluster.
3. The rebalance tool Cruise Control was applied to on this unbalanced cluster and various tuning parameters were applied in the configuration to obtain the best setting. These values helped in the calculation of trade-offs, thus fulfilling our first aim.
4. Finally, data analysis using python, pandas, and sci-kit learn (for calculating T-test) were performed on the gathered systems and application metrics like CPU, Memory Usage, Network I/O and Consumer lag, to name a few. Furthermore, estimations and conclusions were made to complete the second aim using the results of these analyses, thereby assisting in this project's critical evaluation.

1.3 Overview of Dissertation

The scope of the research primarily focuses on the rebalancing of the unbalanced Kafka cluster using the Cruise Control tool and the respective performance tuning related to its configuration and the resultant analysis.

Chapter 2 encapsulates the background on Kafka rebalancing that has been explored previously and the limitations. It also provides an insight into the current work done on Cruise Control that allows for an effective rebalancing, on top of which we are performing our research work.
Chapter 3 describes how the experimental research has been undertaken. It provides details about the experimental setup that was created. The application development work for the experiment and the corresponding metrics capture approach. Moreover, it also provides the specifics about the methodology followed to unbalance the cluster and run the experiments for the Cruise control rebalance mechanism. Furthermore, there is also a discussion about the performance tuning changes for the Cruise Control configuration that were conducted and the parameters that were explored. Additionally, it also provides a detailed overview of the data analyses undertaken during experimentation.
Chapter 4 focuses on the results obtained from the experiments. It also describes the observations of the results and data analysis undertaken as part of the research, along with the critical evaluation and limits of the research work. Consequently, it also gives an insight into the impact this research will have on the real-world scenario.
Chapter 5 provides the overall conclusion of our experiment along with the scope of future enhancements.

Chapter 2

Background

Apache Kafka is a novel distributed open-source messaging system that was developed at LinkedIn in 2010 [7]. Before the designing of Kafka by LinkedIn, they had two separate systems, one for collecting user activity data and the other for system metrics. These technologies functioned as point-to-point data pipelines, delivering data to a single destination with no integration. This single destination architecture had problems servicing real-time data access and the large variety of data processed by the system, making the overall system highly fragile and complex [11].
Therefore, after trying out other systems such as ActiveMQ [12] and RabbitMQ [13] for their infrastructure, they created Kafka, which was designed to address all the shortcomings and bottlenecks present in the infrastructure. The core functionality of Kafka is its distributed commit log [7]. Figure 2.1 depicts the high-level architecture of an Apache Kafka cluster. A cluster comprises several brokers, each of which is allocated a number and stores the message data for the topics. Message producers transmit information to a particular topic or topics in the cluster. Consumers subscribe to a topic or topics, and new messages supplied to that topic or topics are delivered to them as soon as they arrive.

Topics are divided into partitions that can be configured during the topic creation. The brokers of a cluster can distribute partitions of a topic among themselves. However, the order of the messages across partitions cannot be ensured by Kafka. Additionally, along with the number of partitions, a replication factor can also be set, where the minimum value should be two for data redundancy. In the context of replication, Kafka establishes leaders and followers for each partition. The followers duplicate the messages committed by the leader,

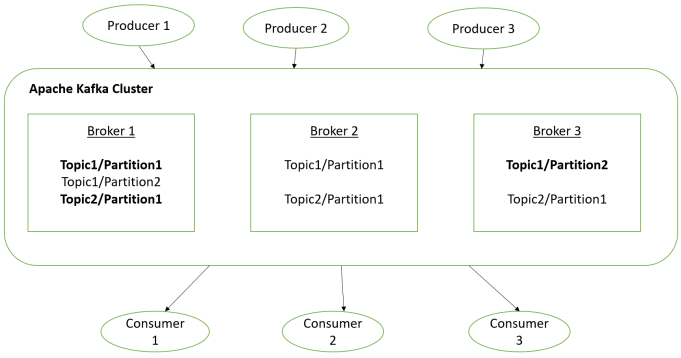

Figure 2.1: Architecture of Apache Kafka Cluster (based on [1])

while the leader is in charge of all reads and writes for the relevant topic partition. In Figure 2.1, leader partitions are highlighted in bold. topic1 has two partitions and two replication factor whereas, topic2 has one partition and replication factor value as three [1], [7].

As discussed above, the messages are stored in the partition of the topics, where the partitions act as a unit of parallelism for the producer and consumer of Kafka. Firstly, the partitions on the producer side allow for parallel message writing. For all the messages that are generated with a key, the producer will hash the key to find the destination partition. This ensures that all messages with the same key are delivered to the same partition. Furthermore, a consumer will be assured that messages will be delivered in the correct order for that partition. Secondly, on the consumer side, the maximum number of active consumers within a consumer group is limited by the number of partitions for a topic. A consumer group is a Kafka method for grouping numerous consumer clients into a single logical group to load balance partition usage. Within a group, Kafka ensures that a topic-partition is assigned to only one consumer [14].

Kafka applications may be able to scale better if they can manage with occasional imbalances. In every rebalancing round, for example, each member releases their resources while simultaneously halting all processing connected to these resources. At big scales and under certain circumstances, this scenario, also known as the stop-the-world effect, has proven rigid and costly [15].

To sum up, the main scenarios where a rebalancing occurs are when a consumer leaves or joins due to a software failure, and the partition is adjusted due to bro-

ker failure. Moreover, another case that results in a rebalance is when the cluster is adjusted, and a broker goes down, the partitions that this broker is in charge of are reallocated [16]. Additionally, there can be workload distribution problem in Kafka due to traffic patterns, bad partition distribution as well as hard or soft host failures that can also result in rebalancing [9].
Besides this rebalance scenario, Kafka also supports scaling [17] the broker infrastructure by increasing the capacity, that is, the addition of new brokers to handle the massive load that may result from a more prolonged duration of cluster operation load or any faulty architecture set-up. However, these new brokers only cater to the new partitions topics that were added after the broker's addition to the cluster. As such, if there is any existing broker with a high load, this will not help in reducing the constraint on that broker. Similarly, there may be another scenario where one of the brokers may have a disproportionate amount of partition leaders for partitions under significant stress for many reasons [18]. The partition reassignment tool [19], can be used to manually reassign the partitions to resolve one of the scenarios mentioned above. However, it is more complex job for the other scenario as we need to figure out which partitions are overburdened, find alternative brokers with less traffic that hold copies for those partitions, and then transfer partition leadership to balance the load across the cluster.

2.1 Related Work

To resolve the cluster load balancing issues and scenarios, Kafka has built-in admin utility tools such as Kafka preferred replica election, partition reassignment, and Kafka Assigner. However, they were not convenient in the long run since they were slow, had in-optimal balances in certain cases and required manual invocation and supervision [9].
Therefore, to make the Kafka cluster rebalancing more efficient and more manageable, LinkedIn created an automatic tool called Cruise Control [10], and open-sourced this project in 2016. Figure 2.2 depicts the architecture of Kafka Cruise Control. It comprises a central system with several sub-systems used for monitoring, analysing, and proposing changes to a Kafka cluster. It uses a Kafka metrics reporter to periodically gather metrics from a Kafka cluster to identify the traffic pattern of each partition. Furthermore, using these traffic attributes and distribution patterns, it calculates the load impact of each partition on the

brokers. Subsequently, Cruise control creates a workload model to predict Kafka cluster workload.

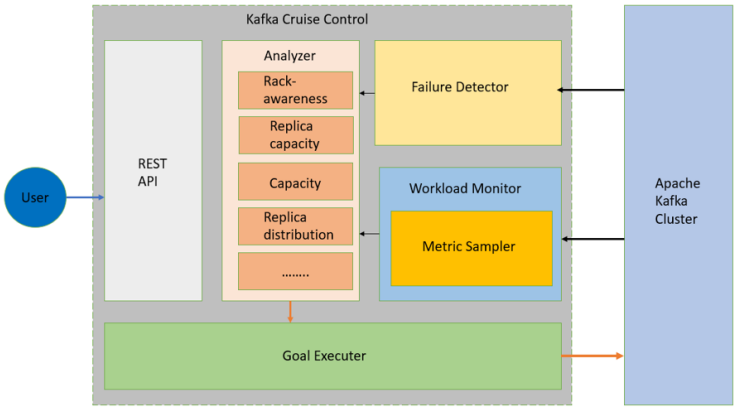

Figure 2.2: Architecture of Kafka Cruise Control (based on [8])

The user can choose from a list of optimisation goals, either hard or soft goals. Each of these goals specifies a set of parameter values that the cluster state must fulfil. These goals are set in a priority sequence, and the Cruise Control server analyser sub-system will attempt to alter the workload model to satisfy each objective if necessary. A hard goal must be met, whereas a soft goal can be violated; if the analyser offers modifications that contradict a hard goal, they will be rejected. As a result, an optimisation proposal, which is a collection of modifications results over a set of objectives, will fulfil all of the hard goals in the set but not necessarily all of the soft goals. Additionally, an important point to consider is that no optimisation proposal will be generated if none of the hard goals can be met [18].
To summarize, Cruise Control helps resolve complex dynamic workload balancing automatically. It also helps solve various challenges such as Trustworthy workload modelling, Fast Optimization Resolution, False Alarm in Failure, Controlled Balancing Execution, and cluster self-healing [8].
In the recent past, not much research has been carried out in Cruise Control by many communities except LinkedIn and Red Hat. Red Hat has a product known as Strimzi [20], that provides various deployment configurations to run an Apache Kafka cluster on Kubernetes. In 2019, Strimzi added support for Cruise Control [21].

Although there have been many performance predictions [22] and comparisons [3], [1] done between Kafka and similar tools to find out which had the best performance, none of these studies considered the performance results for scenarios during the cluster load rebalance operations. Nonetheless, LinkedIn had conducted few tests related to balancing performance on their real-time servers using Cruise Control to market the tool.A sample of that data is provided in Figure 2.3.

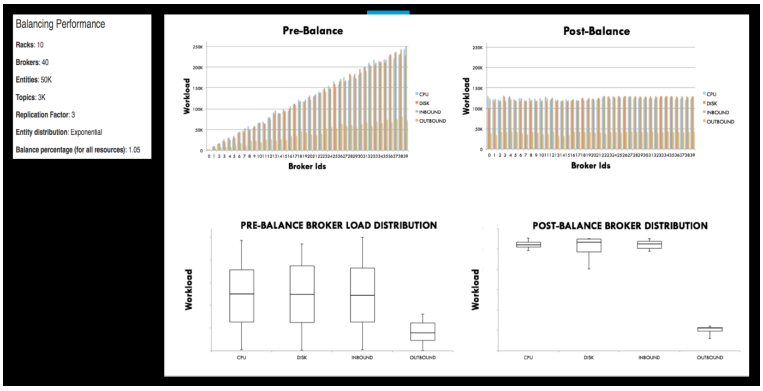

Figure 2.3: Sample Performance Balancing data for Cruise Control provided by LinkedIn [9]

Cruise control contains four major components, namely Workload Monitor, Analyzer, Failure Detector, and Goal Executor. In addition to this, it also has a REST API for client interactions, which Strimzi employed to support two features of Cruise Control. Firstly, generating optimisation proposals from various optimisation goals. Secondly, based on those optimisation proposals rebalancing a Kafka cluster [23].
Moreover, Strimzi also carried out some rebalance tuning setup and performance analysis in their corresponding releases post version (0.18.0). However, these results mainly were for a default parameter value and not for different tuning parameters [24].
Therefore, in this research, we attempt to build on top of what has been done in the Strimzi releases and try out various parameter configurations and assess their respective performance impacts as per the scope outlined in section 1.3. The research could eventually help enhance the Strimzi product offering for Cruise Control and thereby help maintain their Kafka clusters.

Chapter 3

Methodology

In this chapter, a detailed description has been provided of the research undertaken. Figure 3.1 illustrates the high-level overview of the example application and test infrastructure. In this research, attempts have been made to purposefully make the Kafka cluster unbalanced over a period using the example Python application. Cruise Control is tested on this unbalanced cluster, and the monitoring tools collect the appropriate metrics. These metrics are then stored in a CSV file for data analysis.

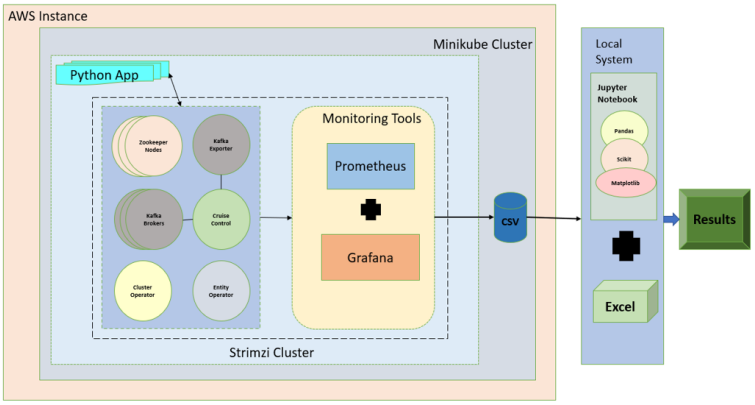

Figure 3.1: Research Design Flow

The following sections provide detailed information related to the overall methodology for the entire research project. Initially, how the example application and test infrastructure for the experiment was set up. Secondly, the core

logic for the unbalanced distributed workload and its respective performance tuning using Cruise control is discussed.

3.1 Test Infrastructure Setup

The test infrastructure setup was carried out in multiple stages which are as follows.

Stage-1:- The first step was to setup the test infrastructure on which the entire experimental study had to be carried out. The cloud provider chosen for the experiment was Amazon Web Services because of its multitude of services and ease of use. Generally, when running a Kafka cluster on AWS cloud for the real-world applications, it is advised to use an M4 or C4 ec2 instance [7]. However, since we planned to experiment with a small set of data replicating the real-world scenario, a t2xlarge ec2 instance was preferred. Moreover, T2 instances are cheaper than m4 or c4 and have sufficient computational capacity for the experiment, making this experimental study cost-effective.
The t2xlarge ec2 instance with ubuntu v20.04 image, 30 Gb storage, and 16 GiB of RAM was used. Since we have used the Strimzi project for our research study, it enables us to run a Kafka cluster on Kubernetes quickly. Therefore, a Kubernetes infrastructure using Minikube was required to be set up first on this ec2 instance. For the successful working of a Minikube cluster, specific dependencies such as java and docker were installed correctly. Following this, the Minikube cluster was set up by following the official Minikube documentation [25].
For efficiently managing a Kafka cluster within a Kubernetes cluster, Strimzi provides a functionality known as *Operators*. These Operators are categorized as Cluster Operator, Entity Operator, Topic Operator and User Operator. The role of the Cluster operator is to deploy and manage the Kafka cluster, Kafka exporter, Entity Operator, Kafka Connect, Kafka Bridge and Kafka MirrorMaker and Cruise Control. Moreover, the Entity operator is a combination of both Topic operator and User Operator, with the former managing the Kafka topics and the latter managing Kafka users [26].Figure 3.2 shows the Strimzi project architecture.

Once the Minikube cluster as created above was up and running, a primary Kafka cluster comprising a broker, zookeeper [27], entity operator, cluster operator, and respective services as per the documentation [28], was created. Finally, this cluster was accessed to send and receive messages, thus ensuring that it is working correctly. This activity completed the first stage of the experimental

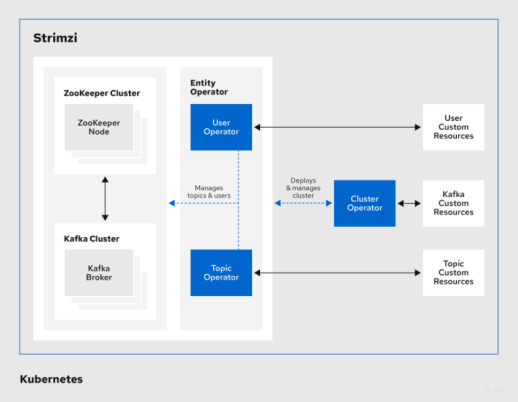

Figure 3.2: Strimzi Architecture inside a Kubernetes Cluster showing the various operators (diagram taken from the official Strimzi documentation [26])

setup.

Stage-2:- In the second stage, the monitoring tools, namely Prometheus and Grafana, were required to gather metrics. In order to achieve this, the Strimzi project has example configuration files in their official repository [29], for Prometheus and Grafana installations. The files in this repository were modified as per our experiment requirement and were used to set up the monitoring solution consisting of pods and services. Additionally, a new file, influenced by the logic presented in [30], was created for Prometheus metrics scraping. All these new and modified metrics files can be found in the principal project repository, as mentioned in Appendix A.1. Furthermore, after successfully installing these monitoring tools, their ports were exposed using port-forwarding to access them outside of the minikube cluster, using the AWS instance's public IP address.

Stage-3:- In the final stage, basic Python application was created. Due to Pythons easy readability and coding capability, it was the preferred choice for the application development work. The required python libraries, including Kafka-python [31], a Python client for Kafka was installed. The simple application had both the Kafka producer and consumer functionality. Execution of this application helped gain insight into the producer-consumer model data flow for a Kafka cluster inside the Minikube. This flow was also visualised in the monitoring tool Grafana [32], and thus it formed the baseline for developing a more complex application for future experiments. At the end of this stage, we created the primary

test infrastructure that acted as the foundation on top of which the rest of the research was carried out.

3.2 Example Python Application

One of the main aims in this research was to create an application that can be a close emulation of a real-world application, generating large amounts of data that can be used for various analyses. The core logic for our application was inspired by [35], and using this as the starting point; a dummy Pizza ordering application was created. This application could produce a fake dataset of Pizza orders continuously and push it to a Kafka topic. Subsequently, the consumer then consumes that data from the topic as per the consumer group distribution. Therefore, this forms the perfect streaming data scenario to feed Kafka much like a real-time application.
The key feature of this application was that it could produce a bulk of fake data by using the python Faker library and generating different pizza orders for a person per outlet, consisting of a random no of extra toppings and pizza numbers. Moreover, the outlet name acted as the key, and the personal information as the message and this message was then sent to the Kafka topic. The key ensured the message was sent to a particular partition in the topic. In addition, extra parameters controlled how these messages are produced using Kafka-producer. They were the topic name, number of messages and maximum waiting time. Topic name as from its name gives the details about the topic to which data needs to be produced. Number of messages is the total number of messages that can be produced per execution and maximum waiting time is the wait time between message production. Simultaneously, there is a Kafka-consumer which would consume these produced messages periodically for the respective topics. The application testing was done successfully on a local cluster by using the python-decouple library [34] and varying the message number and maximum waiting time. Thus, confirming that the application was working as expected. This application will be referred to as the Python Application throughout this dissertation.

3.3 Creation of a Balanced Cluster

Following the successful creation of the Python application, the next step was to create a balanced Kafka cluster and make it operational. To achieve this, an example deployment file from [29] repository was used and modified accordingly to enable external traffic to reach our cluster via Node Ports [33]. The resultant new Kafka cluster comprising three Kafka brokers, three Zookeeper nodes, Kafka exporter, entity operator and cluster operator was created along with their respective services. Additionally, we are primarily concerned with the monitoring data that is to be used in the latter half of the research. Therefore, the storage for Prometheus was made *persistent*, whereas that of Kafka was *ephemeral*. In ephemeral storage, once the system is terminated, the data vanishes, whereas in persistent storage, the data is always available irrespective of the status of the running system [36]. Since there is no actual need to store these Kafka producer data for longer durations and with the repeatability of experiments as discussed in section 3.5, this ephemeral approach saved storage space.
Since the early testing of the Python application was for a single topic use-case, we wanted to make sure our application can replicate a real-time use-case of creating multiple topics on the fly and producing data to these topics as and when we progress. Therefore, the Python application was enhanced to make our producer create data for multiple topics. Dynamic topic creation functionality was added, ensuring that if a topic does not exist, it will create a new topic and start producing to those topics. Alternately, if the topic already exists, it will continue to send the data to that topic without the need to create a new topic again. The new topic function takes partitions per topic and replicas per partition as its parameters. In our case, we have kept the default partition per topic value as ten (which can be modified as per use case) and the replicas per partition value as three since we have three brokers for even distribution. Therefore, on running the Python application with the new functionality, a balanced cluster was obtained with uniform workload distribution.
Furthermore, while testing this balanced cluster, there was another challenge that needed to be addressed. Our application was not scalable enough, and for running multiple producers and consumers, we had to manually start up different application instances. Therefore, to overcome this limitation and make the best use of the Kubernetes scaling features [37], the entire Python application was converted to run as a linux containers by writing separate Dockerfiles for the producer and consumer. Moreover, using the images for these containers, Kuber-

netes deployment files were created for the producers and consumers. Finally, by using these deployment files, we were able to achieve our application's scalability and proceed to the next part of our research. Simultaneously, the respective metrics from this working cluster were collected from Grafana, which concluded our successful creation of a balanced cluster.

3.4 Unbalanced Cluster Creation

The next critical part of the research was to find a way to deliberately unbalance a Kafka cluster. This would ensure that our experimental setting is as close as possible to the real-world problem of dynamic workload distribution. Moreover, this would set up perfectly the scenarios on which we would test the Cruise Control rebalance mechanism and try out various performance tuning options.
To achieve the unbalanced cluster the following three changes were performed on our Python application logic:

1. In the new dynamic topic creation functionality, an update was made to the replicas per partition attribute. It will randomly generate an integer value between 1 and the default value of 3. This change would guarantee that the replicas will not be the same for all the new topics created with the default partition. This means that some brokers would get more partition replicas than others, resulting in an uneven cluster.

2. A new component was also introduced to the application, which was inspired by the logic present in [38]. In this case, at first, the leader of the partition of a topic would be identified. That would be followed by identifying the broker on which this leader is present. Finally, messages for the topic would be sent to that broker according to the broker traffic distribution. A *broker traffic distribution* is a message distribution pattern where a percentage of the generated messages will be sent to a particular broker as per a predefined value. It controls the volume of messages that can be sent to a broker over a certain period.

3. Finally, the broker traffic distribution values were 80 percent for broker 1, 15 percent for broker 2, and 5 percent for broker 3, respectively. As per this distribution, whenever the leader of a partition for a particular topic was identified on broker 1, then that topic received the messages as per the

value mentioned above. Using this method, broker 1 would eventually be more loaded as compared to the other two.

The logic as mentioned above is illustrated in Figure 3.3. The topic with partition number in bold represents the leader of the partition for that topic. The coloured blocks associated with the different brokers show the amount of data the brokers have received over a period using the unbalanced logic, clearly indicating that one of the brokers is heavily loaded.

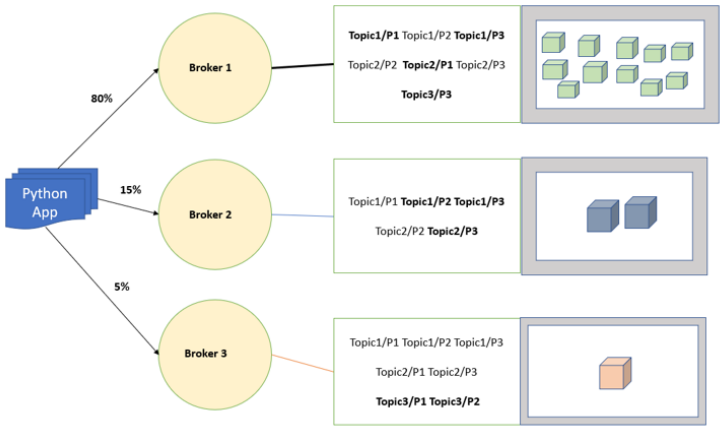

Figure 3.3: Deliberate cluster unbalancing logic representation

These new functionalities ensured that the workload distribution would become asymmetrical and thus achieve purposeful unbalancing of the Kafka cluster and making it a close replica of a real-world scenario. Since the code was updated for the application during the testing of unbalance logic, new container images were created for the producers and consumers of an unbalanced cluster, which makes the Python application scalable, when deployed using the Kubernetes deployment file.
Finally, with this new setup, the complete test infrastructure was ready. It will carry out all the experimental runs for various parameter over which the rebalance mechanism will be tested.

3.5 Experimental Run

After creating the test infrastructure, the first activity of the experimental run was to find out the approximate time for which a cluster should run to reach a stable load. This time is mentioned as burn-in period.
At first, the cluster was unbalanced for hundred topics by running the application for one producer and one consumer. There were four burn-in periods for which the cluster was run. These were 5 minutes, 15 minutes, 30 mins and 1 hour, respectively. After each of these periods, the Cruise Control was run as mentioned in section 3.6. This activity yielded an on demand balancedness score. *On demand balancedness* score compares a Kafka Cluster's overall balancedness before and after the optimization suggestion was created. Two balancedness scores are generated, before and after. The before score is determined by the Kafka cluster's current setup, whereas the after score is based on the optimization proposal generated [39]. Upon observing these scores, it was decided that 30 minutes is the most appropriate burn-in period for the cluster to run before starting the rebalancing activity because it provides a higher value of balancedness.
Following the decision to run the cluster at burn-in period of 30 mins, the final experimentations were triggered. These experiments were carried out for hundred topics, for a set of three producers and two consumers. As mentioned in section 3.7, there are three performance test scenarios for which these experiments were done. Each experiment per scenario was conducted three times. In addition, an important point to highlight is that post each experimental run, the cluster was torn down and re-created with the same attributes. Therefore, a total of 9 experimental runs were carried out for achieving robustness in our results. Figure 3.4 illustrates the graphical representation of the experimental run design.

3.6 Cruise Control Implementation

Since Strimzi provides native support for Cruise Control, it was used on our unbalanced cluster for rebalancing. In the Kafka deployment file, CruiseControl specifications were added to enable Cruise Control monitoring of the running Kafka cluster. A new rebalance configuration file was created motivated by [40], and it contained the main optimization goals that were the 11 default optimization goals inherited by Strimzi from Cruise Control in descending priority order. Out of these 11 goals, six are already preset as hard goals. Moreover, we do not specify any extra goals, either hard or soft, in the specification of rebalance

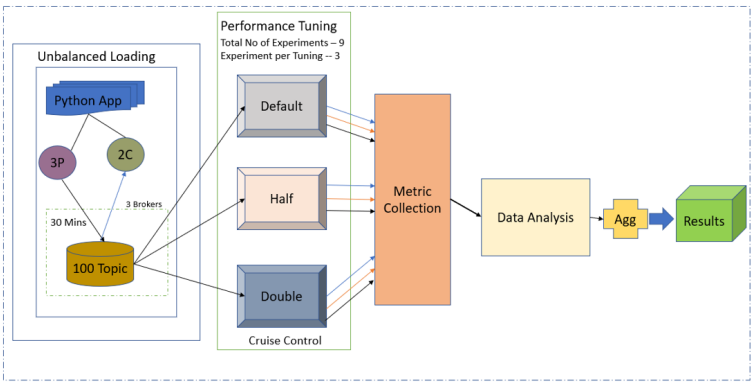

Figure 3.4: Experimental Run Design

configuration file [23].
Therefore, these configuration setting are considered as default tuning for a rebalance. The rebalance is executed after the burn-in period of the testing scenario, as described in section 3.5. Cruise Control would then generate a proposal for rebalancing, which, when approved, will start the rebalancing process. Figure 3.5 depicts a sample generated optimization proposal while running Cruise Control with default settings on an unbalanced cluster.

```
  Type:                 ProposalReady
Observed Generation:    1
Optimization Result:
  Data To Move MB:  540
  Excluded Brokers For Leadership:
  Excluded Brokers For Replica Move:
  Excluded Topics:
  Intra Broker Data To Move MB:         0
  Monitored Partitions Percentage:      100
  Num Intra Broker Replica Movements:   0
  Num Leader Movements:                 54
  Num Replica Movements:                130
  On Demand Balancedness Score After:   96.0263654042023
  On Demand Balancedness Score Before:  83.92687872002556
```

Figure 3.5: Cruise Control Optimisation proposal results

The critical properties in the generated proposals that are of interest, from a performance impact point of view while running these rebalances are, dataToMoveMB, numIntraBrokerReplicaMovements, numLeaderMovements and numReplicaMovements.

A brief definition of these properties as per the official Strimzi documentation [39] are as follows:

1. *dataToMoveMB* - The total of each partition replica's size that will be transferred to a different broker.
2. *numIntraBrokerReplicaMovements*- The total number of partition replicas that will be moved between the cluster brokers' disks.
3. *numLeaderMovements* - The number of partitions whose leaders will be redirected to different replicas.
4. *numReplicaMovements*- The number of partition replicas that will be transferred between different brokers.

Moreover, performance impact during rebalancing for numReplicaMovements and numIntraBrokerReplicaMovements is relatively high, but the latter is lower than the former. However, for numLeaderMovements, performance impact during rebalancing is relatively low since it involves some zookeeper configuration changes. Additionally, dataToMoveMB has a variable performance impact, and a higher value might result in a longer duration for cluster rebalance completion [23]. Therefore, it can be concluded that these properties are pretty crucial while testing out various rebalance configuration settings.

3.7 Rebalance Performance Tuning

In the penultimate part of the research, the rebalance performance tuning options were adjusted for cluster rebalancing. As mentioned in section 3.6, four main properties are involved in performance impact analysis. These properties are also categorised as partition reassignment command, which involves either partition movement or leadership movement. When the optimisation proposal is approved, Cruise Control issues these partition movement commands in batches and applies them to the Kafka cluster. The performance during these rebalances is affected by two things: the number of each type of movement contained in a batch and the replica movement strategy [39].

For the partition reassignment command, we were primarily concerned with partition movement or the leadership movement. Hence, we only targeted the corresponding configuration values for tuning those rebalance parameters. As per the guidelines provided in [41], those three parameter values were concurrentPartitionMovementsPerBroker, concurrentIntraBrokerPartitionMovements, concurrentLeaderMovements and they are type integers. Two use-cases were tested in these tuning operations. In the first case, all these values were doubled, and in the next case, they were halved.
Moreover, two new rebalance configuration files, namely, double default and half default, were created in which both the use-cases as mentioned earlier were configured, respectively. Furthermore, as mentioned in section 3.5, these files were invoked for the corresponding performance test scenarios for the experimental runs. Table 3.1 illustrates the values that were used for the performance tuning operation.

Rebalance Configuration Settings			
Parameters Updated	**Half default**	**Default**	**Double default**
concurrentPartitionMovementsPerBroker	2	5	10
concurrentIntraBrokerPartitionMovements	1	2	4
concurrentLeaderMovements	500	1000	1200

Table 3.1: Performance tuning parameter values

There is one thing to notice in this Table 3.1, while running the Double default, the value for concurrentLeaderMovements is mentioned as 1200, whereas all the parameter values are doubled. The reason being, for now, Strimzi Cruise Control only supports a maximum value of 1250 for this parameter.

3.8 Data Analysis

The final activity of the research comprised capturing all the metrics and performance data observed during the experimentation run as discussed in sections 3.5-3.7 and throughout the research activities. Metrics were captured from Prometheus using the utility function [42]. These metrics were the Kafka clus-

ter metrics such as Memory, CPU and Network Input/Output that Prometheus scrapped during the experiments.
Since the data analyses were planned to be performed using utilities such pandas [43], scikit [44], matplotlib [45] and excel, it was beneficial to obtain these data from the monitoring tools in a CSV. A *CSV* is a delimited text file, where a comma separates each value, and each line of the file acts as a data record [46]. Therefore, based on this logic [47], a python script was created that would convert the Promql functional queries [42], used in Prometheus and store the results in a CSV. Furthermore, these CSV files were then transferred from the cluster to a local system for backup. Subsequently, the CSVs were then read via Jupyter Notebook, and the data analyses were carried out using the utilities mentioned earlier.
Several metrics were gathered for each experimental run. These are total rebalance time, memory usage, CPU usage, network input rate, network output rate and consumer lag. *Consumer lag* is the most significant measure on the consumer side. This metric represents the delay the consumer is experiencing from the most recent messages that are committed to the partition on the broker [7]. The metrics were gathered across the three brokers, and the primary data analyses for performance impact were carried out on it. These analyses obtained individual mean results for each metric, which was then averaged for three brokers to provide the cluster data for that rebalance operation. Two types of data were averaged out: the metrics values before rebalancing and the metrics values during the rebalancing of a running cluster. A difference was taken for these values to find out the metrics' increase or decrease in performance. These difference values were further averaged for all the individual tuning tests, and the final value was recorded. This aggregation of metrics is depicted as Agg in Figure 3.4 earlier.
Besides, a Statistical T-test [48] analysis on the rebalancing data was also performed. In this statistical analysis, the significance level (alpha) [49] was prechosen as 0.05 and the alternate hypothesis [49] was tested. In addition to this, data-visualisation was also carried out for comparative study and critical evaluation.

Finally, all the new and modified files created as part of the research activity can be found in the principal project repository, as provided in Appendix A.1.

Chapter 4

Results and Discussion

In this chapter, we present and discuss the observations of the results and the entire research activity in detail.The first part discusses the detailed analysis of the results obtained from the experiments. In the second part, a critical evaluation of the research activity is done where we try to deduce if the aims that were set out at the start of the research work have been met. In addition to this, we also explain the limitations that were there during the experimentations. Finally, in the last part, we provide the contributions this research has on the modern workforce.

4.1 Results with Discussion

The following section presents the vital results gathered during the experimentations as part of the research activity.
Firstly, Figures 4.1 and 4.2 show the observations of balanced cluster creation. In Figure 4.1, we can see that when four new topics were created, the no of partitions is 40, and replicas per partitions, represented as Replicas are 120. As mentioned in section 3.3 earlier, the default values for per topic partition and replicas per partition during topic creation for a balanced cluster was 10 and 3, respectively. Therefore, the created partitions and replicas are adequately balanced as expected. Additionally, when the cluster was run for some duration, it was also observed that the consumer lag was relatively less, with a maximum value of 50. This meant that messages were consumed quite frequently as the partitions were evenly distributed.

Moreover, as seen in Figure 4.2, the Memory and CPU usage also is quite sim-

Figure 4.1: Replicas and Consumer Lag for a balanced cluster

ilar and evenly distributed across the brokers. Hence, from these observations, we can infer that the cluster is quite balanced.

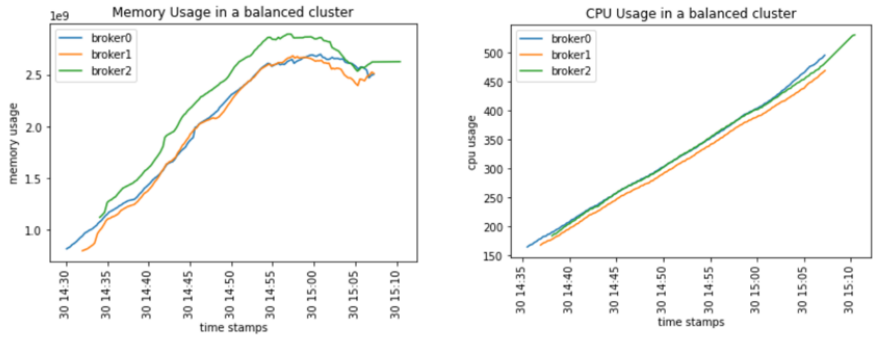

Figure 4.2: Memory and CPU usage in a balanced cluster

Secondly, Figure 4.3 and Figure 4.4 shows the observations of a cluster when it was deliberately unbalanced. In Figure 4.3, we can see that when seven new topics were created, the no of partitions was 70, and no of replicas per partition, represented as Replicas, were 190. As per section 3.4 earlier, the default values for per topic partition and replicas per partition during topic creation is still 10 and 3. However, due to the unbalance logic implemented in the Python application, the no of replicas per partition is non-uniform. Even the consumer lag observed were relatively high, nearly around 50K when the cluster was run for some burn-in period which meant there was a delay in consumption of messages since the partitions were unevenly distributed.

Furthermore, in Figure 4.4, it is observed that the Memory and CPU usage is

Figure 4.3: Replicas and Consumer Lag for an unbalanced cluster

unevenly distributed across the brokers as per the traffic distribution set during the testing. Thus, making one of the brokers more loaded than others. Therefore, we can confirm from these observations that the cluster is in an unbalanced state.

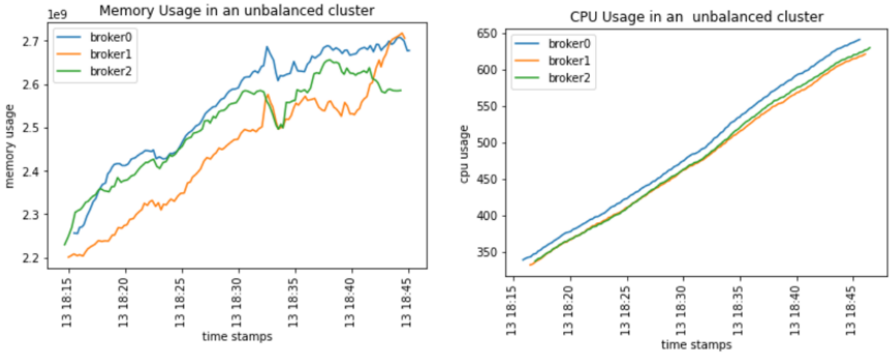

Figure 4.4: Memory and CPU usage in an unbalanced cluster

Besides, the following results were gathered during the Performance Tuning and Data Analyses phase.
During the primary experimentation on the test infrastructure described in section 3.5, an optimisation proposal was generated for each tuning parameter runs via Cruise Control. These optimisation proposals were approved and applied to the Kafka cluster for the cluster rebalancing operation. Table 4.1 shows the average values for each property over the three experiment runs per tuning settings while rebalancing. From these values, we can infer that the number of replica movements in Half and Default tuning is more than the number of leader movements. However, in Double tuning, the opposite holds, and the number of leader movements is more than that of num of replica movements. Also, the values

for numIntraBrokerReplicaMovements is observed as zero across the three tuning settings because this property is currently not supported by Strimzi. Since the leadership movements are less expensive than replica movements and, along with the data to move property, governs the performance impact. Therefore, different performance observations are noticed while running the tunings, which are based on these movements.

Avg Rebalance Property in Different PT	**Half**	**Default**	**Double**
numReplicaMovements	84	113	38
numLeaderMovements	75	63	133
dataToMoveMB	198	278	207
numIntraBrokerReplicaMovements	0	0	0

Table 4.1: Average rebalance configuration via Cruise Control Optimisation proposals

Moreover, as per section 3.7 and 3.8, nine experiments consisting of three each for every type of performance tuning were conducted. The difference between the average value before rebalancing and the average value during the rebalancing for the metrics was calculated. Finally, these difference values were averaged cumulatively to provide the final increase or decrease of performance data, as observed in Table 4.2.

Performance Tuning			
Metrics	**Half**	**Default**	**Double**
Total Rebalance Time	9 Mins 20 sec	5 Min 11 sec	2 Min 2 Sec
Memory Usage	+0.20Gib	+0.61Gib	+0.46Gib
CPU Usage	+0.144	+0.155	+0.149
Input Rate	+376.2Mb/s	+328Mb/s	+461.7Mb/s
Output Rate	+175Mb/s	+263.5Mb/s	+316.8Mb/s
Consumer Lag	+19.5K	+17.6K	+14.6 K

Table 4.2: Performance comparison data gathered during the experimentations

From Table 4.2, we can see that the total Rebalance time was relatively less in the Double tuning compared to Default and Half tuning. Also, all the metrics

have an overall increase in their respective values during the rebalance phase and it is represented by a "+" sign in the table. However, when we compare the Input/Output rate in Double tuning, we notice that it is much more than that of Half tuning. Thereby clearly indicating that the cluster is performing more activities during the movement of replicas and leaders, hence higher values.

Additionally, when comparing the memory and CPU usage between Double and Default tuning, we observe that the usage is comparatively low. Similarly, even the consumer lag is less in comparison, indicating more and more data is being sent in and out during the rebalance.
From all these above observations, it can be inferred that when the rebalancing time reduces, there is a high-performance impact on input and output rate. Likewise, with the increase in rebalance time, there is a lower performance impact on input and output rate. Moreover, as less activity is performed in Half tuning, the consumer lag is higher than other tuning options.
Furthermore, Table 4.3-4.5 gives the statistical analysis for the average rebalance values across the three brokers, with the significance level (alpha) predefined as 0.05. From the p values obtained, it can be seen that they are very less than the significance level defined and thus null hypothesis [49] can be rejected. Thus, confirming the alternate hypothesis of significant difference between the values obtained before rebalancing and after rebalancing for the metrics involved. Therefore, confirming that the experimental study was statistically significant.
Table 4.3, 4.4, and 4.5 shows the Statistical analysis results that were obtained from the data gathered during the default, double and half tuning tests across the three brokers.

T-Test (Broker0 + Broker1 + Broker2)						
Metrics	**statistic**	**pvalue**	**statistic**	**pvalue**	**statistic**	**pavalue**
Memory Usage	4.859	2.987E-06	4.859	2.9866E-06	3.056	0.00272
CPU Usage	9.751	4.739E-17	9.684	1.5101E-16	9.751	4.74E-17
Input Rate	7.003	4.451E-09	6.022	1.8911E-07	6.553	3.27E-08
Output Rate	5.194	5.338E-06	5.052	9.4738E-06	5.065	6.74E-06

Table 4.3: T-test results for default tuning

Furthermore, the performance comparison among the tuning tests were visualised and illustrated in Figure 4.5-4.8.

T-Test (Broker0 + Broker1 + Broker2)						
Metrics	**statistic**	**pvalue**	**statistic**	**pvalue**	**statistic**	**pavalue**
Memory Usage	4.869	3.102E-06	4.869	3.102E-06	3.149	0.002016
CPU Usage	6.992	1.403E-10	7.531	1.032E-11	6.992	1.4E-10
Input Rate	5.356	1.086E-06	5.142	2.418E-06	4.672	1.45E-05
Output Rate	8.532	7.275E-13	8.325	3.442E-12	58.07	1.24E-11

Table 4.4: T-test results for double tuning

T-Test (Broker0 + Broker1 + Broker2)						
Metrics	**statistic**	**pvalue**	**statistic**	**pvalue**	**statistic**	**pavalue**
Memory Usage	14.124	8.461E-27	14.124	8.464E-27	17.831	1.69E-34
CPU Usage	9.751	1.448E-36	18.824	2.557E-34	19.524	1.45E-36
Input Rate	12.762	6.986E-17	12.329	2.452E-16	13.166	3.27E-17
Output Rate	8.466	8.708E-11	8.005	3.957E-10	7.918	6.19E-10

Table 4.5: T-test results for half tuning

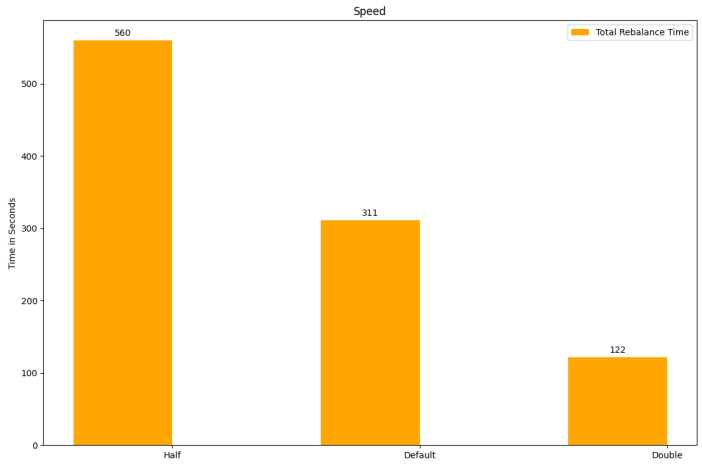

Figure 4.5: Data visualisation of the speed during different rebalance tuning

Figure 4.5 visualises the average total rebalance time in seconds for all three tuning tests. We can see that the time for the test run in double rebalance

configuration is the lowest. This confirms that the time taken for the rebalance operation to complete in the double tunning is faster than the others.

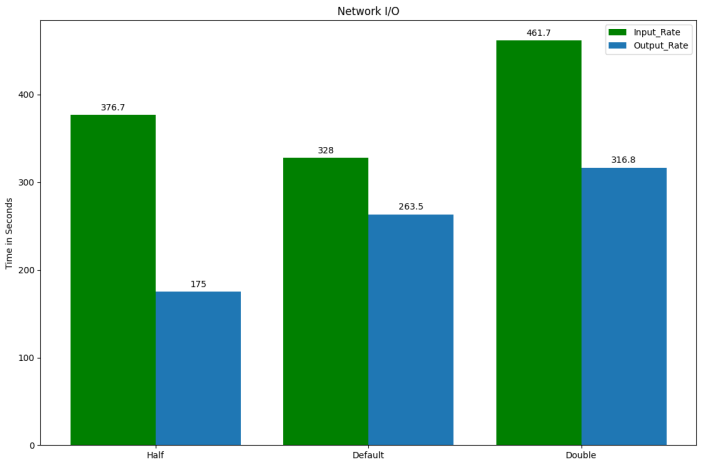

Figure 4.6: Data visualisation of the Network I/O during different rebalance tuning

From Figure 4.6, we observe that the Network Input and Output rate(in MB/s) is higher in double tuning. This highlights that more performance impact is happening on the network since many leaders and replicas movements are happening fast. However, as shown in Figure 4.7, the memory and cpu usage is relatively lower and provide good performance while carrying out the rebalance.

In addition to this, Figure 4.8 also shows the graphs observed for the consumer lag. It is lower in the case of double tuning, again emphasising that since more movement operations are happening, more data is being consumed as the partitions get balanced. Hence, the reduction in consumer lag.

Therefore, we can conclude from the above observations that double tuning is the better method for cluster rebalance.

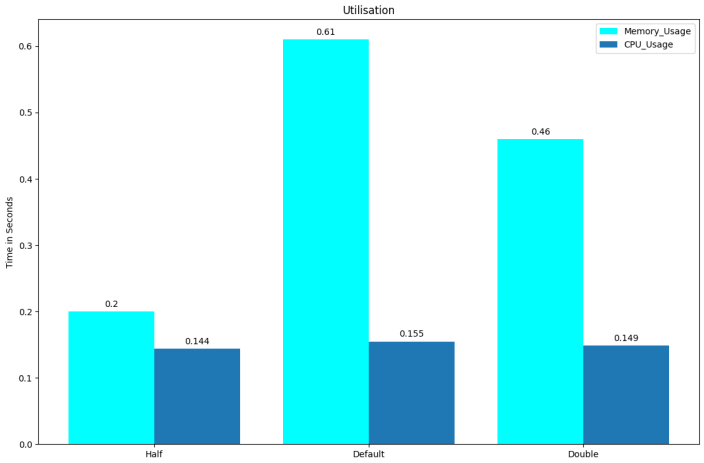

Figure 4.7: Data visualisation of the Memory and CPU Usage during different rebalance tuning

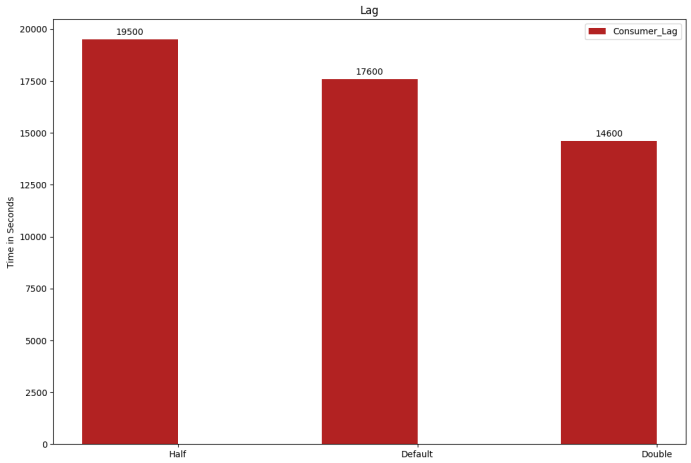

Figure 4.8: Data visualisation of the consumer lag observed during the different rebalance tuning

4.2 Critical Evaluation

Although the scope of Cruise Control is quite broad, and many parameters can be altered to affect the performance impact. However, our research activity is primarily limited to partition assignments commands issued in batches and consists

of replicas and leader movements. We are more concerned about the configuration settings related to these parameters and their respective performance impacts. Based on these limits, when the experimentation was carried out for the research, the following observations were made:

For the performance tuning of the rebalance, when the cluster was run on the double tuning mode, configuration values were double than the default values as mentioned in section 3.7. The resultant rebalance time was significantly less, albeit there was a higher performance impact on the network. Hence, this is quite good for the daily maintenance conducted by the Site Reliability Engineers since they will be able to achieve an evenly balanced cluster in a short duration and with less overall infrastructure downtime.

However, one of the most significant performance impacts from this double tuning would be on the users of the cluster. Shorter rebalance time is good, but it slows down the cluster due to high network usage. In that case, this could affect the client workload (their daily task and activities). The client then needs to decide as per their use cases as to how much of these disruptions can be sustained, as per the service level agreements.An example scenario could be, during the night when the client workload is usually low, the site reliability engineers can run the rebalance operation in a double tuning setting and get the activity done quickly. While, during the daytime with a higher client workload, the rebalance operation can be performed using the half tuning. It will take time to finish the operation. However, it will have a lower performance impact on the users. Therefore, this fulfils the first aim of the research and provides an understanding of the trade-off between rebalance speed and performance impacts.

Similarly, the research also provided us with the durations required to rebalance a running cluster with different tuning parameters. Moreover, it also offered insight into how the current workload will vary in terms of consumer lag which can have more extensive financial ramifications if the lag corresponds to chargeable data. Since a higher level of delay can cause organisations costs to increase. It motivates the site reliability engineers to always use the optimal solutions for rebalancing and managing a Kafka cluster's operational load. Subsequently, the second aim of the research was fulfilled by this.

In addition to this, another important highlight was the way the research

project was carried out. The application development was attempted to be as close as possible to a real-world scenario. Also, the experimentation was highly scalable and robust. Therefore, it provided a good learning curve and exposure to the working of Kafka and Strimzi.

4.3 Impact

Finally, this research project will have a significant contribution to the modern business operation of a Kafka cluster. It provides the efficient values that can be used for the daily system maintenance work relating to the Apache Kafka cluster.

Moreover, this research work was performed in collaboration with the Red Hat team as part of the industry project. The results and observations from this work will help them provide the best settings for their Kafka services with its native Cruise Control support. Therefore, enhancing the product.

Chapter 5

Conclusion

5.1 Closing Statement

In this research, we have attempted to address the problem related to the daily operational issues of Kafka by studying the effectiveness of cluster load rebalancing. Since maintaining a Kafka cluster regularly is quite a cumbersome task and various complex dynamic workload issues arise frequently. One of the methods to resolve this is by making use of the partition assignment mechanism. Although several tools exist to achieve this, we were predominantly focused on Cruise control and its performance tuning on a running Kafka cluster.

In the initial stages of the research, we studied the related work carried out in this field by LinkedIn and Red Hat. At the start of the research activity, we set up a testing infrastructure where we conducted our research experimentations. In the next stage, we developed an example python application that was scalable and used it to intentionally unbalance the Kafka cluster. On this test infrastructure, a highly robust series of experiments were carried out. Subsequently, data analysis was conducted on the metrics generated from these experimentations, and the corresponding results were assessed.

Finally, an evaluation was carried out on these results. It was concluded (from Table 4.2) that a decrease in rebalancing time comes at the cost of a higher performance impact both at the network level and user level. Thus, showing the trade-off between performance impact and rebalance length. Furthermore, it was also summarised that during daily system maintenance by site reliability engineers, it would be preferable to finish the rebalance operation quickly and avoid

more extended downtime. For achieving this, the best performance tuning setting that was decided was double tuning, as mentioned in section 3.7. This parameter would generally be the most optimal for Kafka system maintenance.

5.2 Future Work

The research undertaken was able to answer most of the project requirements and problems as per the expectations. However, there are several areas that can be looked at as part of future work.

Firstly, a variety of permutations and combinations for the performance tuning parameters can be tried out. It could provide us with an even better performance analysis result.

Secondly, the data set used for the experiment can be made more heterogeneous. In other words, more than one application data can be sent to the Kafka cluster. This will make the scenarios for use cases even closer to a real-time, production level cluster.

Thirdly, the Cruise Control work load model could be enhanced. Various methods could be explored to make this model more intelligent and non-linear.

Lastly, for one of the rebalance properties *numIntraBrokerReplicaMovements*, as provided by Cruise Control, a study could be undertaken to enable Strimzi support for future releases.

Bibliography

[1] Kreps, J., Narkhede, N. and Rao, J., 2011, June. Kafka: A distributed messaging system for log processing. In Proceedings of the NetDB (Vol. 11, pp. 1-7).

[2] Facebook Usage: `https://www.slideshare.net/prasadc/hive-percona-2009` [Accessed 19 Aug. 2021].

[3] Dobbelaere, P. and Esmaili, K.S., 2017, June. Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper. In Proceedings of the 11th ACM international conference on ditributed and event-based systems (pp. 227-238).

[4] Hiraman, B.R., 2018, August. A study of Apache Kafka in big data stream processing. In 2018 International Conference on Information, Communication, Engineering and Technology (ICICET) (pp. 1-3). IEEE

[5] Sharvari, T. and Sowmya Nag, K., 2019. A study on Modern Messaging Systems-Kafka, RabbitMQ and NATS Streaming. CoRR abs/1912.03715.

[6] Wang, Z., Dai, W., Wang, F., Deng, H., Wei, S., Zhang, X. and Liang, B., 2015, November. Kafka and its using in high-throughput and reliable message distribution. In 2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS) (pp. 117-120). IEEE.

[7] Narkhede, N., Shapira, G. and Palino, T., 2017. Kafka: the definitive guide: real-time data and stream processing at scale. " O'Reilly Media, Inc.".

[8] Qin, 2016, 'Introduction to Kafka Cruise Control', Stream Processing Meetup, LinkedIn, Mountain View, CA, 2 November, 2016: `https://www.slideshare.net/JiangjieQin/introduction-to-kafka-cruise-control-68180931` [Accessed 10 Aug. 2021].

[9] Kumar, I., 2018. Autonomous Workload Rebalancing in Kafka.

[10] LinkdedIn Cruise Control Repo: https://github.com/linkedin/cruise-control [Accessed 12 Aug. 2021].

[11] Goodhope, K., Koshy, J., Kreps, J., Narkhede, N., Park, R., Rao, J. and Ye, V.Y., 2012. Building LinkedIn's Real-time Activity Data Pipeline. IEEE Data Eng. Bull., 35(2), pp.33-45.

[12] ActiveMQ: https://activemq.apache.org/ [Accessed 5 Aug. 2021].

[13] RabbitMq: https://activemq.apache.org/ [Accessed 5 Aug. 2021].

[14] Kafka Rebalance Magic(Add -e94baf68e4f2 at the end of the URL): https://medium.com/streamthoughts/apache-kafka-rebalance-protocol-or-the-magic-behind-your-streams-applications- [Accessed 19 Aug. 2021].

[15] Incremental Cooperative Rebalancing- Select(Support and Policies): https://cwiki.apache.org/confluence/display/KAFKA/Incremental+Cooperative+Rebalancing [Accessed 19 Aug. 2021].

[16] Rebalancing Partition: https://docs.cloudera.com/runtime/7.2.10/kafka-developing-applications/topics/kafka-develop-rebalance.html [Accessed 17 Aug. 2021].

[17] Thein, K.M.M., 2014. Apache kafka: Next generation distributed messaging system. International Journal of Scientific Engineering and Technology Research, 3(47), pp.9478-9483.

[18] Kafka Scaling Issues: https://strimzi.io/blog/2020/06/15/cruise-control [Accessed 8 Aug. 2021].

[19] Reassigned Partition Tool: https://cwiki.apache.org/confluence/display/KAFKA/Replication+tools#Replicationtools-4.ReassignPartitionsTool [Accessed 19 Aug. 2021].

[20] Strimzi Documentation v0.22: https://strimzi.io/docs/operators/0.22.0/overview.html [Accessed 6 Aug. 2021].

[21] Manual Cruise Control Deploy: https://strimzi.io/blog/2019/09/03/hacking-for-cruise-control/ [Accessed 7 Aug. 2021].

[22] Wu, H., Shang, Z. and Wolter, K., 2019, August. Performance prediction for the apache kafka messaging system. In 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS) (pp. 154-161). IEEE.

[23] Cruise Control Concept: `https://strimzi.io/docs/operators/latest/using.html#cruise-control-concepts-str` [Accessed 11 Aug. 2021].

[24] Cruise Control Rebalance-Config: `https://strimzi.io/docs/operators/latest/using.html#con-rebalance-str` [Accessed 12 Aug. 2021].

[25] Minikube: `https://minikube.sigs.k8s.io/docs/start/` [Accessed 8 Aug. 2021].

[26] Strimzi Operator: `https://strimzi.io/docs/operators/latest/quickstart.html` [Accessed 9 Aug. 2021].

[27] Hunt, P., Konar, M., Junqueira, F.P. and Reed, B., 2010, June. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In USENIX annual technical conference (Vol. 8, No. 9).

[28] Strimzi Quickstart: `https://strimzi.io/quickstarts/` [Accessed 8 Aug. 2021].

[29] Strimzi Metrics Repo: `https://github.com/strimzi/strimzi-kafka-operator/tree/main/examples/` [Accessed 12 Aug. 2021].

[30] Prometheus Operator Metric Repo: `https://github.com/1984shekhar/AMQ_STREAMS_KAFKA_POC/tree/master/strimzi-grafana-prometheus` [Accessed 12 Aug. 2021].

[31] Kafka Python Library: `https://kafka-python.readthedocs.io/en/master/index.html` [Accessed 8 Aug. 2021].

[32] Grafana Labs - The open platform for beautiful analytics and monitoring: `https://grafana.com` [Accessed 9 Aug. 2021].

[33] Node Port Expose: `https://strimzi.io/blog/2019/04/23/accessing-kafka-part-2` [Accessed 12 Aug. 2021].

[34] Python Decouple Library: https://pypi.org/project/python-decouple/ [Accessed 17 Aug. 2021].

[35] Application Base Streaming Logic Repo: https://github.com/aiven/kafka-python-fake-data-producer [Accessed 17 Aug. 2021].

[36] Driscoll, J.R., Sarnak, N., Sleator, D.D. and Tarjan, R.E., 1989. Making data structures persistent. Journal of computer and system sciences, 38(1), pp.86-124.

[37] Moilanen, M., 2018. Deploying an application using Docker and Kubernetes

[38] Unbalance Cluster Base Logic Repo: https://github.com/tomncooper/kafka-topic-loader/blob/master/producers.py [Accessed 16 Aug. 2021].

[39] Rebalance Schema Config: https://strimzi.io/docs/operators/latest/using.html#con-rebalance-str [Accessed 12 Aug. 2021].

[40] Strimzi Rebalance Config File Repo: https://github.com/strimzi/strimzi-kafka-operator/tree/main/examples/cruise-control [Accessed 9 Aug. 2021].

[41] Rebalance config specificatios: https://strimzi.io/docs/operators/latest/using.html#type-KafkaRebalanceSpec-reference [Accessed 9 Aug. 2021].

[42] Prometeus Function: https://prometheus.io/docs/prometheus/latest/querying/functions [Accessed 5 Aug. 2021].

[43] Python Pandas library: https://pandas.pydata.org/ [Accessed 5 Aug. 2021].

[44] Scikit Learn library: https://scikit-learn.org/stable/ [Accessed 5 Aug. 2021].

[45] Matplotlib library: https://matplotlib.org/ [Accessed 5 Aug. 2021].

[46] Shafranovich, Y., 2005. Common format and MIME type for comma-separated values (CSV) files.

[47] Promql queries to CSV conversion logic Repo: https://github.com/RobustPerception/python_examples/blob/master/csv/query_csv.py [Accessed 7 Aug. 2021].

[48] Statistical Analysis library for python: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_ind.html [Accessed 12 Aug. 2021].

[49] P-Value interpretation: https://www.statsdirect.co.uk/help/basics/p_values.htm [Accessed 14 Aug. 2021].

v, 1, 6, 9

Appendix A

Code

A.1 Main Repository

1

The code developed and the changes made during the research activity : `https://github.com/bhuvigithub/Kafka-Project` 1, 9

1