FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Event Driven Publish/Subscribe Consumer Group Autoscaling

Diogo Landau

FOR JURY EVALUATION

Mestrado Integrado em Engenharia Eletrotécnica e Computadores

Supervisor: Jorge Barbosa

February 25, 2022

Abstract

Message brokers provide asynchronous communication between data producers and consumers in a distributed environment, being Kafka one of the several message broker alternatives. To scale data consumption rate, Kafka has Consumer Groups, which is an abstraction that allows to parallelize tasks between consumers in a group. This abstraction presents new concerns that depend on the broker's current load, which include: determining the number of consumers required; determining the partition assignment between the consumers in a group in order to guarantee that the consumption rate of each partition is not less than their respective production rate. Additionally considering the load varies with time, there is an increasing need to find autoscaling solutions to reduce operational cost, while guaranteeing that all data is being consumed within acceptable time after it has been produced.

As such, this problem is modeled as a new variation of the Bin Packing Problem where the bins are consumers of a consumer group, and the weights are the partitions and their respective write speeds. Due to the varying load applied to Kafka brokers, the weights change in size over time. Another variation is the fact that item assignments are not persistent, and therefore can be rebalanced between consumers in different time instants. This adds another concern related to the fact that while a partition is being rebalanced, data is not being consumed from it (rebalance cost). We propose a new metric to account for this cost (Rscore), and present four new heuristic algorithms based on the Rscore, three of which prove to be a competitive alternative when compared to existing heuristic algorithms with respect to the multi-objective optimization problem of minimizing both the number of consumers and the rebalance cost.

To deliver a fully automated solution to the autoscaling problem, using the aforementioned theoretical approach, we propose a system that is capable of responding to a wide range of load applied to the brokers. When compared to exisitng solutions, the proposed system is capable of reducing the operational cost, and guarantees that the consumption rate keeps up with the production rate, something that cannot be guaranteed by current solutions.

Resumo

Message Brokers possibilitam a comunicação assíncrona entre produtores e consumidores de informação num ambiente distribuído, sendo o Kafka uma das várias ferramentas disponíveis para este tipo de comunicação. Com o intuito de escalar a velocidade de consumo, o Kafka disponibiliza grupos de consumidores, que é uma abstração que permite paralelizar tarefas entre os consumidores de um grupo. Com esta abstração surgem as seguintes preocupações relacionadas com a carga atual de um *broker*: quantos consumidores são necessarios?; como é que as partições vão ser distribuidas entre os consumidores de modo a garantir que a velocidade de produção não excede a velocidade de consumo?. Considerando que a carga dos *Message Brokers* varia com o tempo, há uma crescente necessidade de haver um sistema capaz de escalar um grupo de consumidores de modo a garantir que não haja atrasos significativos no consumo dos dados produzidos.

Modelou-se o problem como uma nova variação do *Bin Packing Problem* (BPP), em que os *bins* são os consumidores, e os pesos são as partições e as suas respetivas velocidades de escrita. Devido à variação da carga nos *brokers*, os valores dos pesos variam com o tempo. Outra variação deste problema é que uma atribuição de uma partição a um consumidor não é persistente, e pode ser redistribuida para outros consumidores. Isto apresenta uma nova preocupação quando se procura uma solução a este problema associada ao facto de que enquanto uma partição está a ser redistribuida, os seus dados não estão a ser consumidos (custo de redistribuição). Propõe-se uma nova métrica (Rscore), que procura quantificar o custo de redistribuição de uma nova configuração. Baseados na nova metrica, apresenta-se também quatro novas heurísticas que fornecem uma solução ao BPP tendo em conta o Rscore. Três destes quatro algoritmos consideram-se soluções viáveis a este problema multi-objetivo de minimizar tanto o numero de consumidores como o custo de redistribuição, quando comparados a soluções existentes utilizando a frente de pareto.

Por forma disponibilizar uma solução autónoma capaz de escalar um grupo de consumidores, apresenta-se também um sistema que tem a capacidade de resposta a uma ampla variedade de carga dos *brokers*. Quando comparado às soluções existentes, o sistema consegue não só reduzir o custo operacional, como também garantir que a velocidade de consumo acompanha a velocidade de produção, algo que não é uma garantia nas soluções existentes.

Acknowledgments

I would first like to start off by thanking my supervisor Professor Jorge Barbosa, who made himself available to not only clarify my questions, but also for his valuable guidance in choosing between development alternatives.

I would like to thank my work colleagues, and especially Fernando Gomes. The freedom in solution approach and the tools you made available brought my work to a higher level.

I would also like to thank my friend, Dr. Xavier Andrade for having extensively reviewed this document and provided valuable insights.

To my parents, Maria João Carrapato and Jorge Landau, thank you for your patient and unconditional support, and to my sister, Beatriz Landau, thank you being my source of inspiration.

Diogo Landau

“
Genetic predispositions are only that: predispositions.
It's not a destiny written in stone. People have choices.”

Dr. Jennifer Melfi, The Sopranos

Contents

List of Figures

List of Tables

Symbols and Abbreviations

ETL	Extract, Transform, Load
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
ISR	In-Sync Replica
w.r.t	with respect to
VM	Virtual Machine
AWS	Amazon Web Services
GBQ	Google Big Query
CPU	Central Processing Unit
KEDA	Kubernetes Event-driven Autoscaling
EKS	Elastic Kubernetes Service
ECS	Elastic Container Service
QoS	Quality of Service
MOP	Multi-Objective Problem
CLI	Command Line Interface
BPP	Bin Packing Problem
SBSBPP	Single Bin Size Bin Packing Problem
WCPR	Worst Case Performance Ratio
APR	Asymptotic Performance Ratio
DE	Data Engineering
PV	Persistent Volume
PVC	Persistent Volume Claim
NP-hard	Non-deterministic polynomial-time hardness
ILP	Integer Linear Programming
NF	Next-Fit
WF	Worst-fit
FF	First-fit
AF	Any-fit
BF	Best-fit
AA	Approximation Algorithm
AAF	Almost Any-fit
AWF	Almost Worst-fit
HF	Harmonic-Fit
NFD	Next-Fit Decreasing
BFD	Best-Fit Decreasing
FFD	First-Fit Decreasing
MFFD	Modified First-Fit Decreasing
GKE	Google Kubernetes Engine

TBPP	Temporal Bin Packing Problem
TC	Transportation Company
CBS	Cardinal Bin Score
MWF	Modified Worst Fit
MWFP	Modified Worst Fit Partition
MBF	Modified Best Fit
MBFP	Modified Best Fit Partition
ID	Identifier

Chapter 1

Introduction

As a company becomes more reliant on the data it produces to drive its decisions, data warehouses become core components for providing data for analysis and reporting. To populate the data warehouse, one of the most common procedures is the ETL (Extract, Transform and Load) process which involves: extracting the data from multiple data sources; transforming the data; loading the data into the appropriate format for consultation.

It is also well known that as companies become more data driven, the challenges associated with dealing with large volumes of data becomes more apparent as digital solutions start failing to respond, or are unable to respond within appropriate time constraints.

For this reason, a distributed architecture that resorts to microservices is a common and efficient way to tackle the problem as it simplifies scaling based on the system's needs, nullifies the single point of failure, and increases the system's resilience.

On the other hand, incorrectly implemented communication between microservices can become a distributed solution's scalability bottleneck, as the microservice components can become too coupled and increase the technical cost of making any changes to the system. This is where message brokers come in. Instead of having the components communicating with one another directly, the message broker serves as an intermediary that asynchronously delivers the information between components as it is requested by the interested parts. In short, the message broker mediates the communication between data producers (entities that send data to the broker) and data consumers (entities that request data from the broker).

Throughout this work, we use Kafka as the message broker system. A Kafka cluster is made of topics, which in turn is a unit that is subdivided into several logs, commonly reffered to as partitions. When a producer sends a record to a specific topic, it will be appended into one of the topic's partitions, illustrated in the Data Production Domain in Figure 1.1.

A consumer group is a unit composed of multiple consumers, each with the common task of consuming data from the topics the group is subscribed to. To consume the data within a topic, each of its partitions have to be assigned a single consumer belonging to the same group. Assigning the partition's to different consumers in a group, is Kafka's way of providing scalability.

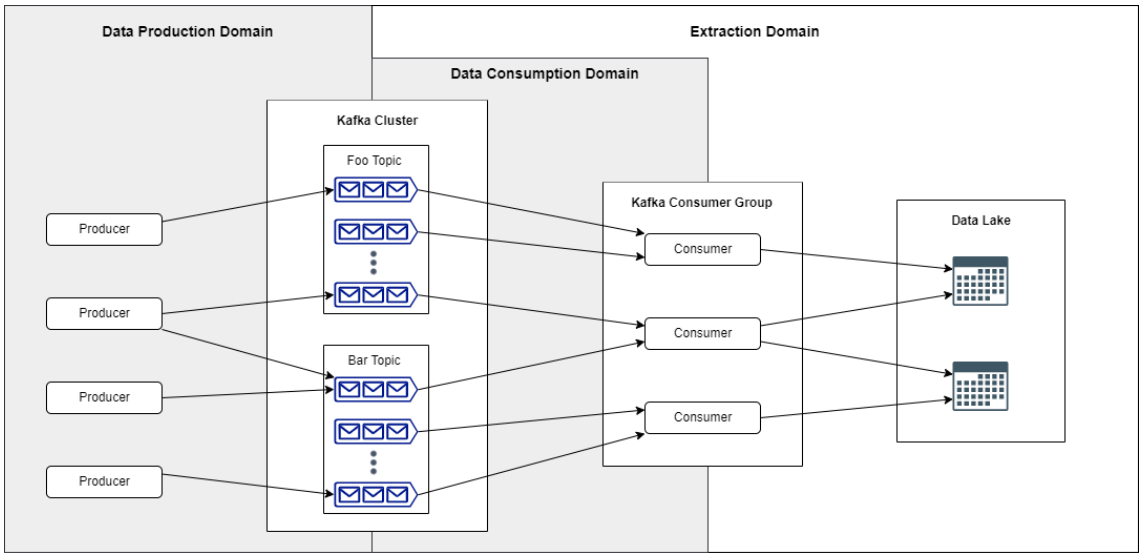


Figure 1.1: Data pipeline representing the flow of data since it is appended into one of the data sources (Topic), to when it is fetched by a consumer and inserted into a Data Lake.

This design feature leads to the fact that a group does not require more consumers than the number of partitions, as only one consumer in a group can read data from a given partition.

The problem was proposed by the data engineering team at HUUB (MAERSK), and it is framed within the scope of the Extraction Domain in Figure 1.1. It consists of being capable to dynamically manage a group of consumers, to up- and down-scale the number of instances (i.e. consumers), and also specify the tasks (partitions) that are assigned to each consumer. We need to define the required amount of parallelism as to guarantee the number of unread messages by the group does not increase with time. Hence, the rate at which the group is consuming data from each partition, must be bigger or equal to their respective write speed.

Considering consumers as bins, and partitions as items that have to be assigned a bin, we model this problem as the Bin Packing Problem (BPP), with the particularity that items vary in size with time. This occurs because the partition's size correlates to its current write speed, which fluctuates based on the current system's load and inevitably implies that a solution for a given time instant may not hold true in future instants.

On account of this BPP variation, a new solution has to be computed at each instant, which might lead to a partition (item) being assigned to a different consumer (bin) when compared to the consumer group's previous configuration. Since two consumers cannot read from the same partition concurrently, the cost associated to rebalancing a partition is related to the amount of data that is not being read while the partition is being assigned to another consumer.

Given that it is the first time the Bin Packing Problem is applied in this context, existing algorithms do not take the rebalancing cost into account. Hence, in Section 4.4.1 we propose a metric to account for a given iteration's rebalance cost (Rscore). Additionally, using the Rscore, in Section 4.4.6 we propose four new BPP heuristic algorithms that account for the rebalance costs.

The algorithms' performance is compared in Section 4.4.6.1. Since the algorithms are attempting to solve a multi-objective problem that aims to minimize both the number of bins required and the rebalance cost for a single iteration, we compute the pareto front. It shows that three of the proposed algorithms are a competitive solution to the problem at hand.

To deliver a fully automated solution to the autoscaling problem, in Chapter 4 we introduce a system comprising three components: a monitor, a consumer and a controller. Using the data from the monitor process the controller uses the aforementioned theoretical approaches (BBP heuristics) to assign tasks to consumers. Each of the components is unitarily tested throughout Sections 4.2, 4.3 and 4.4. An integration test is also presented in Chapter 5, aimed to reflect the autoscaler's response time to the message broker's current load. Lastly, in Chapter 6, we discuss results and suggest future work. The following chapters introduce this thesis' technological and theoretical background, Chapter 2 and Chapter 3 respectively.

Chapter 2

Infrastructure

2.1 Distributed Systems Architecture

When developing a digital infrastructure system, it is common to start with a monolith approach where all application concerns are contained in a single deployment, but it is hard to ignore its disadvantages as it can easily become the bottleneck of the system. With monoliths, a lot more care has to be given to how each part of the system communicates with one another, as communication is achieved through method or function calls. This leads to very tight coupling between different application components, which inevitably provides a lot of friction to updating the codebase due to the high risk of it disabling the service entirely.

Like with most applications, with every new feature, the monolith grows, also increasing the computing resources required to run a single instance. In times of traffic peak, it is common for an application to scale its service allowing for response speed to remain within acceptable times. The bigger the deployment, the longer it takes to make a new instance available and the more computing resources are used, making it more expensive to provide availability in times of high traffic.

With this type of system, technological choices have a big impact on the final product, and have to be thoroughly planned in the beginning to guarantee that the system fulfills its requirements. It also makes it harder to adopt a different technology further into the development due to the technological cost. This point alone, makes it hard to accept this design, especially considering the frequency of new advances in the field, and it is only natural that a new approach to system architecture were developed.

Although there is no formal definition for what a microservice is, “Microservices are small, autonomous services that work together” [27, Chapter 1]. Small is of course a subjective measure, but in fact there is no "theoretical" boundary on this quantity. It depends on the context of the application and the business, but considering the size of these microservices, it is only logical that each of the small services follows some kind of team structure allowing for parallel development, without any real coupling between the services.

Microservices are built around different “business capabilities” which are “independently deployable by fully automated deployment machinery” [10]. In other words, each microservice can be deployed on its own, hence scaling services and reliability is "included" with this kind of architecture. This makes updating code very low risk. As an example, in the scenario there is some flaw in a new deployment sent to production, it will only affect that microservice in particular, making it easier not only to track where the fault lies, but also guaranteeing that the remaining services are not made fully unavailable.

When scaling smaller decoupled services, only the service that requires scaling is in fact replicated. This leads to less computer resources utilization thanks to the reduced overhead of the service, resulting in a considerate cost reduction, and faster response to scaling needs.

With the possibility of running decoupled services interacting through lightweight network calls, each microservice can be programmed in whichever programming language and run the datastore that fits its needs best. The reduced size of each service, also makes technological changes and code refactoring a lot simpler, and can be done in a lot less time when compared to its counterpart.

It is also worth mentioning that the separation between each service allows several representations of the same data in different ways depending on the service that is storing it. The benefits of this approach are not only in a projection perspective (there isn’t a need to over-engineer how the data will be stored for future purposes that still are not implemented), but it also helps model the world in the way that fits each service best.

Having multiple representations for the same data, also leads to inconsistency in the data stored in different services. Employing a data warehouse in this scenario overcomes this hurdle by unifying the data models of the different services, and becoming the single source of truth for any representation of the data within the system. It is also notably hard to maintain data consistency between the distributed services, and on this note, the following section is introduced.

2.2 Distributed Messaging Systems

When interacting between two distinct distributed servers, rarely should two components communicate with one another directly or through synchronous communication. If this were not the case, the communication would become too convoluted with the increase of instances and services, becoming a source of coupling, i.e., communication routes would have to be defined at the machine level.

As stated in [30], it is clear the market needs point to a messaging system that has the following features:

- Scalable - the system provides tools with which services can process a bigger load of messages in a time of intense volume of traffic;
- Space decoupling – The receiving and sending entity do not need to “know” each other;
- Reliable – The system can guarantee that the receiver has received the message;

- Asynchronous – The sender and receiver do not have to be active at the same time.

To enable decoupling between services, most messaging systems leverage the use of a broker, providing the space and time decoupling, with the added benefit of it also being asynchronous. After the broker acknowledges the reception of the message sent by a producer, the producer does not need to worry about its consumption allowing for it to go back to the tasks it has at hand. There are two main approaches at solving this problem, event- and message-driven paradigms.

2.2.1 Message-Driven

This paradigm makes use of message queues where producers send the data for it to be consumed by a single consumer in the same order it was appended to the queue. When data is produced, in case there isn't a consumer interacting with the queue, the message is stored until read.

RabbitMQ is a well known platform commonly used for this purpose. All the previous features are provided, and on the note of scalability, a given service can connect to a queue with one or more consumers to increase consumption rate. This is done using round-robin dispatching [36] which simplifies parallelizing work between instances, always guaranteeing that each message in a queue will be read by a single consumer.

2.2.2 Event-Driven (Publish/Subscribe)

Messaging system which allows a producer to send a message to a certain category, which can be later retrieved by multiple consumers which are subscribed to the same category. This architecture not only decouples the services, it also allows many-to-many communication.

A common application for the event driven paradigm is for event sourcing. This is the pattern of storing any event that changes a system's state via an event object, onto an event log that preserves the sequence in which the events occurred. This is presented as an alternative to storing structures that model the state of the world, into storing events that lead to the current state of the world [26, Chapter 5].

Comparing this scenario to the one of using a database, in the latter, every time the main database is updated, a second database recording the historic state has to be updated as well, storing the past states the system has been through. With event sourcing, when an event changes the state of the system, it is simply appended to the sequential event log, which in turn makes it less expensive to make operations of this type.

2.3 Kafka

Kafka functions as a distributed log running in multiple brokers that coordinate with each other to form a cluster.

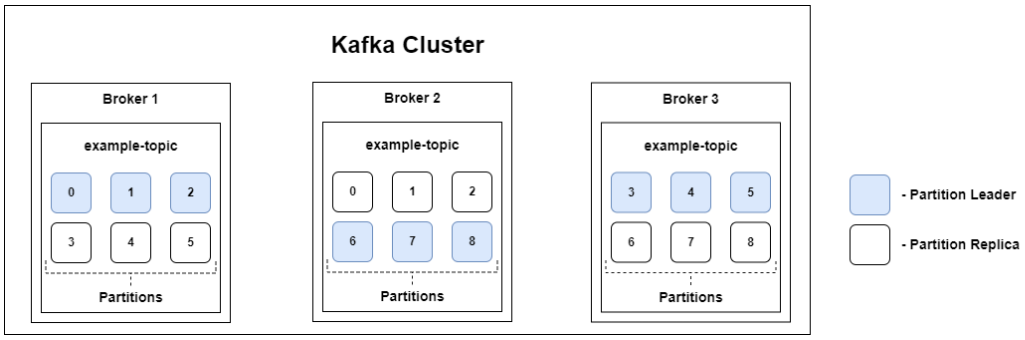


Figure 2.1: Representation of a Kafka Cluster composed of three brokers, with a single topic (Example Topic), containing 9 partitions and a replication factor of 2.

2.3.1 Broker

A Kafka cluster is made up of one or more brokers, which function as servers where the messages can be published to, and consumed from. This is the entity each client has to connect to in order to interact with Kafka. After a client connects to a single broker, it will also be connected to the remaining elements of the cluster.

2.3.1.1 Topics and Partitions

When data is published, the producer has to specify to which topic the record goes to. A topic contains several partitions, and each partition can be stored in different brokers. Message order is only guaranteed within a single partition.

At the time of topic creation, there are three parameters that have to be provided:

1. **topic** - The name of the topic to create;
2. **partitions** - The amount of partitions the topic is subdivided in;
3. **replication-factor** - The amount of replicas of each partition.

The `example-topic` presented in Figure 2.1 would have been configured with `topic=example-topic`, `partitions=9` and `replication-factor=2`.

Although the first two parameters are self-explanatory, the third is one of the most important features to guarantee a reliable service to a producer.

Assuming the scenario of choosing the replication factor of two, this means that for each partition in the created topic, there will be two partitions that represent the same log in different brokers. At any given time, there is only one partition leader (broker), which leaves the other partition trying to keep up with the main log. When a replica is up-to-date with the leader, it becomes an in-sync replica (ISR). If a partition leader fails unexpectedly, the partitions it is leading have to be reassigned a different leader. For each partition, the new partition leader will be one of the brokers that contains an in-sync replica of the same partition.

2.3.2 Producer

To publish data, producers have to connect to one of the brokers from the cluster, which will automatically inform the client as to how to connect to the remaining brokers of the same cluster. When sending a record, the producer has to specify the topic to which the record is to be added, and may optionally provide other parameters which will impact the partition the record will be added to. If no additional parameter is provided, a random partition is selected to which the record is added.

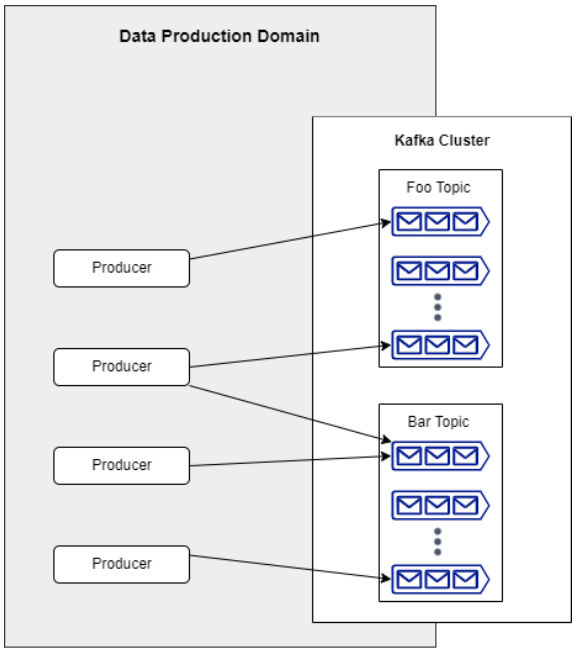


Figure 2.2: Data production representation within the Kafka Ecosystem.

For a single partition, the log that belongs to the partition leader (shaded partitions in Figure 2.1) is where the messages are appended to and read from. To guarantee a message is delivered safely to the cluster, a producer might want to wait for acknowledgement. This is one of the configurations that allows for reliability when adding a message to a topic [20]. In Figure 2.2, the partitions presented are only the leader partitions within the topic.

There are 3 possible values for this configuration:

- `acks=0` - Producer does not wait for acknowledgement, making it an unreliable configuration;
- `acks=1` - Producer waits for acknowledgement from the partition leader;
- `acks=all` - Producer expects an acknowledgement from leader and all of its replicas.

After appending data to a log, it can no longer be changed (immutability). Additionally, when producing a message, if the order of a group of messages is important, Kafka provides a feature that allows messages to be consumed in the same order as they were produced. This is done by

setting the key of a record to the same value, having a hash function run over the key, and the partition where the message goes to is consistently the same. If the number of partitions changes, this is no longer the case.

2.3.3 Consumer Group and Consumer Clients

When connecting a consuming client to the messaging system, the topic(s) it wishes to subscribe to, the consumer group id and at least one of the brokers from the cluster have to be provided (a connection to a single broker connects the consumer to all the other brokers in the cluster).

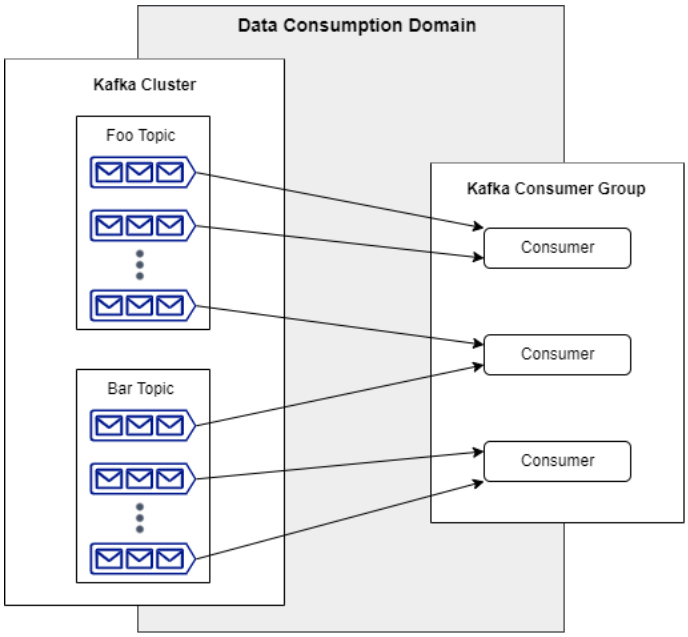


Figure 2.3: Representation of the consumption process for a Kafka Consumer Group.

Messages are then read in order w.r.t a single partition, and in parallel w.r.t. multiple partitions. Each consumer from within a consumer group, reads from exclusive partitions. This means that two consumers belonging to the same consumer group cannot be responsible for reading messages from the same partition, which in turn implies that the parallelism Kafka provides, is through the partitions in a topic. The amount of active consumers a single consumer group can have is therefore limited to the total amount of partitions in the topics the consumer group is subscribed to [19].

In order to maintain a reliable consumption service, whenever a consumer successfully reads a message and commits its offset, Kafka internally stores the offset in a topic with the name `__consumer_offset` [18]. This internal topic has another functionality which is related to the consumer group management. For a group to know when to rebalance it is important for the system to monitor the state of each consumer in the group, and this is performed by a group coordinator. To elect a group coordinator, the consumer group's ID is selected as the message key, which is then hashed into a partition of this internal topic. The broker that leads this partition becomes the group coordinator.

For a consumer to remain a member of the group, it must periodically send heartbeats with a predefined interval to the coordinator. When the maximum amount of time without receiving a heartbeat is exceeded or the coordinator is notified of a consumer leaving the group, rebalancing is triggered, reassigning the partitions the consumer was responsible for to the remaining elements of its group.

The same rebalancing is triggered when a consumer starts up and requests to join a group. Rebalancing is simply attempting to split the load between the active consumers of a group, and Kafka's group coordinator does this by assigning approximately an equal amount of partitions to each consumer in a group.

Depending on the chosen configuration for the parameter `auto.offset.reset`, when a consumer receives its assignment from the coordinator, if there are no committed offsets by the consumer group's id for the topic-partition pair, then this policy is selected to determine where to start consuming records from its partitions. If it is set to `earliest` it will start consuming from the first message in the partition, whereas if set to `latest` it will start consuming from the last message appended to that partition's log.

When data is consumed in batch, it can only be guaranteed that the data is read at least once. The reason why this is the case is due to the fact that if a consumer fails while processing the messages, before committing the offsets, the partition is reassigned and the same messages will be read again. To define the size of the batch, a combination of the following parameters, `max.partiiton.fetch.bytes` and `fetch.max.bytes`, provides more granular control. If the number of duplicate messages is relevant, the batch size is an important factor to take into consideration, since the bigger the size, the more messages can be read more than once in cases of consumer failure.

While rebalancing, it is important to note that the consumers are not capable of consuming data from the partitions being rebalanced, making it an expensive operation which is to be avoided. If a certain consumer stops unexpectedly, it is no longer consuming from its partitions, and the coordinator has to wait for as long as `session.timeout.ms` for it to trigger a rebalance. This is a configurable value, but there is of course a trade-off. The bigger the value set, the less rebalancing is performed, but it will also take the coordinator longer to determine whether a single consumer is unavailable before triggering a rebalance.

2.4 Containers

When internet services first started, it was common to have the services running on local hardware. To handle peak traffic, scaling was performed by purchasing more hardware, which would then become unused when the traffic was no longer as high [31, Chapter 1].

Virtual Machines (VMs) became the next advancement in this industry, and it allowed to use the resources of pieces of hardware more efficiently as multiple VMs could run on a single machine as long as there was space. This was when cloud service providers like Amazon, Google and Microsoft also started renting out VMs. In moments of peak traffic, a company could scale services

in minutes, and there was no need to waste hardware resources with unused instances, therefore paying only for what is used.

It became clear that Virtual Machines also came with disadvantages, as there was a considerable overhead of memory to support the operating system of this environment.

Enter containers. Running instances could be done in a matter of seconds without the overhead of an operating system, which allows applications to run consistently in different environments as long as the containers are created resorting to the same image.

An image is a bundle of code/software that contains all the dependencies and libraries a process might need to run reliably in different computing environments [35]. A container is a process that is created using an image to setup the execution environment to perform its tasks.

2.5 Kubernetes

Kubernetes works with a cluster of distributed nodes that interact with one another to work as a single unit. This service allows for an automated distribution and scheduling of containerized applications. The level of abstraction causes a deployment to have no ties to a specific machine.

A Kubernetes cluster is formed by two entities, the control plane, and nodes. The first is responsible for coordinating all activities in a cluster, such as scheduling, scaling and rolling out new updates of an application. The second, contains a process named kubelet, which manages the communication of the node with the control plane. There are additional tools like containerd or docker to allow the node to deploy containerized applications."In practice, a node is simply a VM or a physical computer that serves as a worker machine in Kubernetes" [33].

To change the Kubernetes cluster's state, the communication is performed via the **Kubernetes API** (throughout this section, bold points out the definitions that are used in the remaining Chapters) which is at the core of the control plane. This HTTP API, allows differing entities to communicate with the cluster and manage kubernetes objects [32]. Within this thesis' context, to manage a consumer group, the controller described in Section 4.4 leverages this API to manage deployment and volume resources.

As soon as the Kubernetes cluster is running, it is possible to deploy the containerized applications. This can be done using one of the many resource types, such as a Deployment. It is within this type of resource that we can specify how our applications will run, and how many instances of the application we want to make available. Within the deployments `spec.replicas` is where the number of pods for this resource is defined, whereas `spec.template` is where the Pod resource is specified for this deployment.

A **Pod** is Kubernetes smallest deployable resource, and it is an instance that runs within a single Node. It can run one or more containers that are created using their respective images, which share network and volume resources. As for a **Deployment**, it is a logical grouping of pods, which contains information about the group's state. A DeploymentController monitors the group, so as to change its actual state to reach the desired state specified in the deployment's configuration.

When up- or down-scaling the deployment, `spec.template` is used to determine the Pod template to manage. Within the deployment's spec field, which specifies its desired state, `spec.selector.matchLabels` and `spec.template.metadata.labels` allow a deployment to know which pods to manage. The pods are tagged with the label specified by `spec.template.metadata.labels` and the deployment searches for the pods tagged with the labels specified in `spec.selector.matchLabels`.

2.5.1 Autoscaling

When scaling a deployment, Kubernetes exposes the HorizontalPodAutoscaler resource that controls a workload resource such as a Deployment to match the demand. The autoscaler continuosly monitors CPU, memory or other custom metrics, to determine the amount of instances required to match the demand.

As described in [1], the algorithm works in loop, constantly evaluating the performance of the pods by measuring the selected metric's average value for the last minute. The control is performed by default every 30 seconds, although it can be modified by changing the value of `horizontal-pod-autoscaler-sync-period`.

The following equation determines the appropriate amount of pods (numPods), so as to have the average value for the metric (e.g., CPU or Memory) within a deployment below the defined target (Target).

$$numPods = ceil\left(\frac{\sum_{p \in Deployment} metric_p}{Target}\right) \tag{2.1}$$

where $metric_p$ is the average value of the metric to control in the last minute of a pod p, and *ceil* refers to the ceiling function.

To reduce noise when modifying the workload resource, the autoscaler can only scale-up if there was no rescaling within the last 3 minutes. The same applies to scale-down except that the waiting time is 5 minutes.

Within the context of the problem presented in Section 1, if a pod represents a consumer and a deployment a consumer group, the metric on which to base the autoscaling cannot be the deployment's average CPU or memory utilization, as the process shown in Figure 1.1 utilizes more network than processing resources.

Kubernetes Event-driven Autoscaling (KEDA), provides a Kafka Consumer Group scaler which allows to increase the amount of consumers based on the average lag of a consumer group [21]. Although this is definitely better than scaling based on CPU usage, it still lacks granularity on evaluating the groups performance based on other metrics, e.g., consumption and production rate of a partition. The load is also distributed between the consumer instances using the strategy mentioned in Section 2.3.3, which assigns approximately the same amount of partitions to each consumer within a group. Since the speed of each partition to be assigned is not the same, this is does not equally split the load between consumers.

2.5.2 Kubernetes Operating Modes

When running Kubernetes, there are several alternative operating modes. The first is to have the cluster run on manually provisioned hardware, which provides with the most control over which type of node is added or removed from the cluster.

As soon as a node is manually added to the cluster, the control plane can start assigning pods to run in the node. When there is no longer any more space in the cluster to run other instances, more nodes have to be added to the cluster to allow the deployments to be correctly scheduled to maintain the state described in the manifest.

With this type of configuration, the operational cost is associated with the nodes that are added to the cluster, which can represent renting out VMs or buying the actual physical hardware that runs the necessary software to be a part of the Kubernetes cluster.

Another option is to use some kind of cluster auto-management which is already included in a few cloud provider's services. Both Google and Amazon provide their own Kubernetes engine with this optional mode of operation, Google Kubernetes Engine (GKE) autopilot and Elastic Kubernetes Service (EKS) fargate respectively, which automatically manage the nodes within the cluster without having to manually interact with it. This means that if the cluster does not have enough resources to maintain the state of the objects that have been defined, a node is automatically added and the instances that require node resources can now be scheduled.

This mode of operation allows the user to pay by pod instead of node [12, 9], and provides a truly dynamic and hands-off experience to scaling a Kubernetes cluster.

Chapter 3

Bin Packing Problem

The bin packing problem (BPP), as defined in the literature, is a combinatorial optimization problem that has been extensively studied since the thirties. As specified in [34], there are several categories with which to define a BPP. The problem at hand is the one of a Single Bin Size Bin Packing Problem (SBSBPP). As described in [8], an informal definition of this BPP, is: provided there are n items, each with a given weight $w_j\,(j = 1,...,n)$ and an unlimited amount of bins with equal capacity C, the goal is to arrange the n items in such a way that the capacity of each bin is not exceeded, and to determine the minimum amount of bins required to hold the n items.

Formally, the problem can be summarized as the following optimization problem:

$$\text{min} \quad \sum_{i \in B} y_i \tag{3.1}$$

$$\text{subject to} \quad \sum_{j \in L} w_j \cdot x_{ij} \leq C \cdot y_i \quad \forall\, i \in B \tag{3.2}$$

$$\sum_{i \in B} x_{ij} = 1, \quad \forall\, j \in L \tag{3.3}$$

$$y_i \in \{0,1\} \quad \forall\, i \in B \tag{3.4}$$

$$x_{ij} \in \{0,1\} \quad \forall\, i \in B, j \in L \tag{3.5}$$

where set B corresponds to the available bins, and L as the list of items to be arranged into different bins. Decision variable x_{ij} and y_i, indicate whether an item j is packed in the bin i, and whether bin i is used respectively.

As for the Constraints, 3.2 assures that the sum of the items in a bin, does not exceed it's capacity, and 3.3 makes sure that every item is assigned a bin.

It is also common to represent the bin-packing problem in its normalized version, where all the weights are down-scaled by the total capacity of a bin, and the capacity of each bin is 1. This implies modifying Constraint 3.2 into:

$$\sum_{j \in L} \hat{w}_j \cdot x_{ij} \leq y_i \quad \forall\, i \in B, \tag{3.6}$$

where $\hat{w}_j$ represents the normalized weight of an item (w_j/C). Throughout the following anaylsis, both "weight" and "size" will be used interchangeably, and the normalized BPP model will be employed.

3.1 Linear Programming

When interested in achieving the optimal solution, the textbook implementation of the BPP model presented by Equation 4.1 is not computationally efficient. A common approach is to study the Upper and Lower Bounds of the Algorithm, and to add valid constraints to restrict the search space.

In [8], there is an extensive review on the state-of-the-art algorithms that have been developed to solve the ILP problem, comparing several models and their efficiency when solving the same set of problems.

Within the context of the problem presented in chapter 1, provided two time instants t_1 (the last instant at which the item assignment was computed) and t_2 (the current time instant at which the assignment is to be re-computed), given that the items' sizes change between t_1 and t_2, it is possible that the configuration computed in t_1 no longer complies with Constraint 3.2. Considering the NP-hard nature of the problem, and the fact that the ILP problem is only interested in minimizing the amount of bins disregarding an item's previous assignment, this thesis will give more emphasis to the Approximation Algorithms that provide more control over the item reassignment problem, and provide a solution within the strict time constraints imposed by the real-time requirements presented in Chapter 1.

3.2 Approximation Algorithms and Heuristics

A method is presented to classify the BPP problem [5], which will be used throughout this section. During an algorithm's execution, a bin can find itself either *open* or *closed*. In the former, the bin can still be used to add additional items, whereas in the latter, it is no longer available and has already been used.

In the literature, it is common to present a parameter α which indicates the size limit of the weights within the list L, which varies between $]0,1]$. This thesis studies the problem where the item's size is simply limited to the size of the bin, and therefore $\alpha = 1$.

$A(L)$ denotes the amount of bins a certain algorithm makes use of for a configuration of items L. $OPT(L)$ is used to represent the amount of bins required to achieve the optimal solution. Defining Ω as the set of all possible lists, each with a different arrangement of their items, given a performance ratio defined by

$$R_A(k) = \sup_{L \in \Omega} \left\{ \frac{A(L)}{k} : k = OPT(L) \right\}, \tag{3.7}$$

the Asymptotic Performance Ratio (APR) of an algorithm A (R_A^∞) is defined as

$$R_A^\infty = \limsup_{k\to\infty} R_A(k). \tag{3.8}$$

Despite the mulititude of classes presented in [5], the only classes of problems that are of interest for this thesis are:

1. online - Algorithms that have no holistic view over any other item in the list except the one it is currently assigning a bin to. It is also a requirement that an item is assigned a bin as soon as it is analyzed.

2. bounded-space - The amount of bins open at a single instance is limited by a constant provided prior to its execution.

3. offline - The algorithm is aware of all the items in the list before assigning bins to each item, and so their ordering does not directly impact the outcome as it is not necessary to respect the list's initial order.

3.2.1 Online Algorithms

Whenever an item is analyzed it has to be assigned a bin. This also implies that the bin in which the current item will be inserted is a function of the weights of all the preceding items in the list.

This section analyzes Any fit, Almost any fit, bounded-space and reservation technique algorithms. Throughout the following sections, when *current item* is mentioned, it is referring to the next item to assign a bin to. After assignment, the item that follows it in the list, is then considered the *current item*.

Next Fit (NF) is one of the simplest algorithms developed to solve the bin packing problem heuristically, and it consists in the following procedure. The current bin is considered to be the last non-empty bin opened. If the current bin has space for the item then the item is inserted. Otherwise, the current bin is closed, a new one is opened, and the item is inserted. The next item on the list is then considered the current item.

With regards to time complexity, NF is $O(n)$. And because there is only one open bin at a time, this is a bounded-space algorithm with $k = 1$. As proven in [14],

$$R_{NF}^\infty = 2. \tag{3.9}$$

Worst Fit (WF): For each element, it looks for the open bin that has the most space where it fits, and inserts the item in that bin. If there is no open bin that can hold the item, then a new bin is opened, and the item is inserted. Having no closing procedure, this algorithm is unbounded, and it does not run in linear time.

Although it is expected that this algorithm would perform better than the NF, in the worst-case performance analysis it does not gain any benefits from not closing its bins, as stated in [17]

$$R_{WF}^{\infty} = R_{NF}^{\infty}. \tag{3.10}$$

First Fit (FF) searches each open bin starting from the lowest index, and the first bin that has the capacity to fit the item, is where the item is inserted. If there is no open bin where the item can be inserted, then a new bin is opened where the item is inserted. In [16] it is shown that

$$R_{FF}^{\infty} = \frac{17}{10}. \tag{3.11}$$

There is a more general class of algorithms presented as *Any Fit* (*AF*) in [13]. This class' constraint is that if BIN_j is empty, then no item will be inserted into this bin if there is any bin to the left of BIN_j that has the capacity to contain the item. In the same paper it is also shown that no algorithm that fits this constraint can perform worst than the WF, nor can it perform better than the FF, always with respect to the asymptotic performance ratio. Then

$$R_{FF}^{\infty} \leq R_{A}^{\infty} \leq R_{WF}^{\infty}, \forall A \in AF \tag{3.12}$$

Best fit (BF): Attempts to fit the item into any of the open bins where it fits the tightest. If it doesn't fit in any, then a new bin is created and assigned to the item. As can be seen, it is similar to the worst fit, with exception to the fitting condition, differing only in the fact that the item is inserted where it fits the tightest, and not where it leaves most slack.

As it happens, BF also belongs to the AF class, and it performs as well as the First Fit

$$R_{BF}^{\infty} = R_{FF}^{\infty}. \tag{3.13}$$

In [13], another class of algorithms presented, which is also a subclass of AF, is *Almost Any Fit* (*AAF*). This class has as constraints: *If BIN_j is the first empty bin, and BIN_k is the bin that has the most slack, where $k < j$, then the current item is not inserted into k if it fits into any bin to the left of k.* One of the properties proven in the same paper, is that

$$R_{A}^{\infty} = R_{FF}^{\infty}, \forall A \in AAF, \tag{3.14}$$

which is to say that any algorithm that fits the previous constraints, have the same APR as the FF algorithm. Focusing on the constraints, it is clear to see how BF and FF belong to this class.

Due to the constraints presented in the AAF class, an improvement for the WF algorithm arises wherein the current item is inserted in the second bin with most space, instead of the first. This algorithm is the Almost Worst Fit (AWF), and with this simple change, now belongs to the AAF class, having as APR

$$R_{AWF}^{\infty} = R_{FF}^{\infty} = \frac{17}{10}. \tag{3.15}$$

3.2.2 Bounded-space

Bounded-space algorithms have a predefined limit on the amount of bins that are allowed to be open at a given instance, which will be defined as k. It is also a subclass of the online algorithms.

An example of a bounded-space algorithm is NF, as it never has more than a single open bin at a given instant. The other presented online algorithms can also be adapted into a bounded space algorithm, simply by specifying a procedure as to which bin to close before exceeding the limit.

A bounded-space algorithm can be defined via their packing and closing rules [5]. A class that derives from rules based on the FF and BF can be defined in the following manner:

- Packing - When packing an item into one of the available open bins, the selected bin either follows a FF or BF approach.
- Closing - When choosing which bin to close, if following the FF approach, then the bin with the lowest index is closed. If following the BF it's the bin that is filled the most that is selected as the one to be closed.

The notation for this class of algorithms is AXY_k, where X represents the packing rule, and assumes the letters F or B, and Y which can either be an F or a B, refers to the closing rule. The k represents the maximum amount of open bins allowed.

Next-k-fit applies both packing and closing rules based on the first fit algorithm, and as expected when $k \to \infty$ it approximates the FF algorithm, having as APR $17/10$. As shown in [25]

$$R^{\infty}_{AFF_k} = \frac{17}{10} + \frac{3}{10k-10}, \quad \forall\, k \geq 2. \tag{3.16}$$

Best-k-fit, which is also known as the ABF_k algorithm, has been proven in [24] to have

$$R^{\infty}_{ABF_k} = \frac{17}{10} + \frac{3}{10k}, \quad \forall\, k \geq 2. \tag{3.17}$$

As for AFB_k, [37] showed that this algorithm has

$$R^{\infty}_{AFB_k} = R^{\infty}_{AFF_k}, \quad \forall\, k \geq 2. \tag{3.18}$$

For the last possible combination of this class of algorithms, ABB_k has been proven in [6] to have

$$R^{\infty}_{ABB_k} = \frac{17}{10}, \quad \forall\, k \geq 2 \tag{3.19}$$

which surprisingly indicates that the value of k (as long as it's bigger than two) has no effect on the APR metric of this algorithm, contrary to all the previous algorithms.

The next algorithms make use of a reservation technique that proved to be very useful to break the lower bound of the APR of the Any Fit class of algorithms. The first algorithm to be developed of this type was the Harmonic-Fit (HF_k) which makes use of k to split the sizes of items into k different intervals.

I_j denotes the interval of sizes with index j, and is within the range

$$\left(\frac{1}{j+1}, \frac{1}{j}\right], \quad \forall\, j \in \{1,...,k-1\}. \tag{3.20}$$

I_k is defined as the interval from $(0, 1/k]$.

B_j is used to classify the bin type which is responsible for holding items of type I_j.

It is shown in [23] that for $HF_k \forall k \geq 7$ the algorithm already performs better than the other online bin packing algorithms presented, with regards to the APR.

Posterior to this technique being presented, it was clear that the reservation technique could be a good approach to improve the performance of the Any-Fit algorithms, and as such, several other algorithms have been created around this technique, that achieve even better APR's than HF, but because these techniques all involve assigning item types, based on their size, to their respective bin type, this approach is not applicable to the problem, as the control over item rebalancing is reduced, which will further be described in the following chapter.

3.2.3 Offline Algorithms

Comparing with the previous section, offline algorithms have the added benefit that all items are known prior to its execution. As long as the set of items remains the same, items can be grouped, sorted or anything that might be convenient for the algorithm that is going to execute over the list of items.

It is important to note that as soon as an algorithm chooses to sort the list of items, it automatically implies that the algorithm no longer runs in linear-time as the sorting algorithm would have a time complexity of $O(nlog(n))$.

Most of the Any Fit algorithms, perform best if the list of items is sorted in a non-increasing order. The following three algorithms remain the same as for how the packing is done when analyzing the list of items, with the exception that before running the algorithm, the list is sorted.

Provided the list is sorted in the aforementioned manner, the **Next Fit Decreasing (NFD)** has a considerable improvement in terms of it's worst-case performance, and performs slightly better than FF and BF, as presented by [3]

$$R^{\infty}_{NFD} \approx 1.6910. \tag{3.21}$$

As can be seen in [16], **Best Fit Decreasing (BFD)** and **First Fit Decreasing (FFD)** improve the APR metric with presorting compared to their online versions of the same algorithm

$$R^{\infty}_{BFD} = R^{\infty}_{FFD} = \frac{11}{9}. \tag{3.22}$$

Another algorithm which is worth mentioning within this class of offline algorithms is the

Modified First Fit Decreasing (MFFD). As shown in [15], The APR is

$$R^{\infty}_{MFFD} = \frac{71}{60}, \tag{3.23}$$

which is achieved by initially presorting the items in a decreasing manner and grouping each item into seven distinct groups based on the item's size.

Table 3.1: Item size interval which guides the grouping for the MFFD algorithm.

Group	Item size interval
A	$(1/2, 1]$
B	$(1/3, 1/2]$
C	$(1/4, 1/3]$
D	$(1/5, 1/4]$
E	$(1/6, 1/5]$
F	$(11/71, 1/6]$
G	$(0, 11/71]$

After doing so, the algorithm then performs the following five steps:

1. For each item that belongs to group A, from biggest to smallest, assign it a bin with the same index as the item has within it's group. When this process terminates, there are as many bins as items in group A and the bins created are $B_1, ..., B_{|A|}$.

2. Iterating over the existing bins from left to right, for each bin, if any item in group B fits in the bin, insert the biggest such item in the current bin.

3. Iterating through the list of bins from right to left, for each bin, if the current bin contains an item that belongs to group B, do nothing. If the two smallest items in $C \cup D \cup E$ do not fit, do nothing. Otherwise insert the smallest unpacked items from $C \cup D \cup E$ combined with the largest item from $C \cup D \cup E$ that will fit.

4. iterate over the list of bins from left to right, and for each bin if any unpacked item fits, insert the largest such item, repeating until no unpacked item fits.

5. Lastly, the remaining items that did not fit in bins $B_1, ..., B_{|A|}$ are inserted in an FFD fashion starting with a new bin $B_{|A|+1}$.

3.3 Bin Packing Applications

There are several applications where the BPP is used to provide with an optimal or sub-optimal solution. In [4], the problem is modeled as a BPP to solve the last mile delivery problem a courier

usually faces when having to decide which transportation companies are to be used to deliver the parcels to the customers. In this case, a Transportation Company (TC) represents a type of bin, and a vehicle of a TC represents a bin instance. The problem is formulated as a generalization of the BPP, considering a profit value is incorporated into the cost function which is based on the bin-type each item ends up assigned to, which in the problem's context, this profit represents the TC's QoS. As such, the objective function aims to minimize the net cost related to the vehicle instances used to allocate the parcels and the profits associated with the parcel assignment.

Another application where the BPP proves itself useful is in the The Virtual Machine Placement (VMP) problem, which has gained more attention due to increasing use of cloud providers to support companies' technological infrastructure. Usually the problem can be generalized to a set of tasks (Virtual Machines), each having to be assigned one of the cloud provider's physical machines while attempting to minimize the operational cost. A variation of the BPP is usually presented in this case where the items are considered ephemeral and add another temporal dimension since they have associated a start time and a duration during which the task has to be ran uninterrupted.

In [7] the tasks have an equal start time and the aim is to minimize the unused capacity of a physical machine over time, and therefore the optimal solution for the BPP does not necessarily provide the optimal solution in their Temporal Variant of the BPP (TBPP). In addition to the temporal model presented by [7], in [2] the tasks may have differing start times, and the objective function presented is more in line with the traditional BPP where the number of physical machines required is minimized. Moreover, due to the increased energy costs related to machine fire-ups (switching a machine off when it is not in use and back on again when required), the authors also attempt to minimize the number of fire-ups, making their problem multi-objective.

As presented in [11], provided a set of items $N = \{1,2,...,n\}$, a weight $w_i \; \forall i \in N$, and a start and end time $[s_i,t_i[$, the set of items which have to be executing at a time instant s_j is $S_j = \{n \in N : s_n \leq s_j \; and \; t_n > s_j\}$. This allows the definition of the non-dominated set $T = \{n \in N : S_n \not\subseteq S_k \; \forall \; k \in N\}$ which discretizes the time dimension into several time instants. Due to the stateful nature of the problem that allows a task to be in more than one S_j, which implies an item may have an assignment from a previous time instant, and since an item cannot be rebalanced, an optimal solution for the $BPP(t)$ is not the same as an optimal solution for $TBPP(t)$.

3.4 Conclusion

Although the temporal aspect of the Bin Packing Problem has already been reviewed, there is a gap when it comes to the possibility of rebalancing the items between the different bin instances. Existing solutions consider a bin packing model where the items' sizes do not change with time, and the assignment is persistent and ephemeral, i.e., the items require an uninterrupted assignment for a specified duration. Modeling the BPP with item sizes and consumer assignments that vary over time, is a new variation that has not been studied, and will be one of the contributions of this work.

In view of the fact that the existing approximation algorithms were developed to solve a single iteration of the BPP, these have to be adapted to the problem at hand, which requires solving a new iteration of the BPP given an already existing assignment. Despite the fact that the adaptation presented in Section 4.4.5 improves these algorithms' performance with respect to the rebalance cost, there is still room for improvement. The Modified Algorithms presented in Section 4.4.6 are developed to solve this variation of the BPP, while taking the rebalance cost into account.

Chapter 4

Consumer Group Autoscaler

4.1 Problem Formulation

As shown in Figure 1.1, the data pipeline consists of extracting data from Kafka topics and inserting it into a data lake to be further treated in the remaining ETL processes. The work developed throughout this thesis, aims to optimize the pipeline from Kafka into BigQuery, by providing a dynamic solution that makes use of Kafka's parallelism without disregarding the cost of the number of instances deployed.

The aim is to achieve a deterministic approach as for the number of consumers required working in parallel, so as to guarantee that the rate of production into a topic is not higher than the rate of consumption. In contrast, if this were not the case, messages would accumulate in a topic leading to the lag between the last message inserted and the last message read by the consumer group to increase with time.

A Kafka topic is subdivided into several partitions, distributed within the several brokers of the Kafka cluster. When sending a message into a topic, if a key is provided, then the message will consistently end up in the same partition, whereas if none is provided, then the message is inserted in a round-robin fashion in one of the partitions of the same topic. Messages going into a topic may also have different sizes ($bytes$), and for these reasons, the write speed ($bytes/s$) to the several partitions in a topic is not guaranteed to be the same.

The following model, also assumes that the maximum consumption rate of a single consumer is constant, and if there is enough data to be consumed, this is the speed the consumer functions at when working "full throttle". This is elaborated in Section 4.3.5.1 based on several tests performed to determine this capacity value for a consumer.

Based on the previous information, the model for this pipeline is considered to fit the constraints for the SBSBPP, where the consumer is the bin that has as capacity its maximum consumption rate, and the weights are the partitions and their respective write speeds. The problem then is to find the minimum amount of consumers where to fit all the partitions so as to make sure that the sum of the write speeds of the partitions assigned to a single consumer does not exceed its maximum capacity.

For a single iteration, the optimal solution finds the arrangement between the partitions and the consumers that minimizes the amount of instances, but this solution is not static, as the write speeds of the partitions changes with time. As such, there is a second factor to take into consideration which is the partition reassignment.

When a partition is reassigned, because two consumers from the same group cannot be consuming from the same partition at a time, when assigning a partition to another consumer, the one it is currently assigned to has to stop consuming in order to allow the new consumer to start. Due to this process, there is some downtime where data is not being consumed from a partition.

For this reason, making use of an optimal algorithm that also minimizes partition redistribution, is not feasible as it would not run within the necessary time requirements due to the NP hard nature of the problem. To determine the arrangement of partitions and consumers, this work is based on the studied approximation algorithms which have already been mentioned in Chapter 3. As is clear, there is no approximation algorithm which considers item redistribution. The reason for this may lay on the fact that the context where this problem lies is not very common, and there is no mention of a stateful assignment where each item is already assigned a bin prior to executing the algorithm.

Due to the dynamic nature of the problem the items change in size depending on the measurement performed at time instant t. Consequently, Constraint 3.2 is remodeled to take the time instant t at which the algorithm is to be executed into account.

$$\text{min} \quad \sum_{i \in B} y_i \tag{4.1}$$

$$\text{subject to} \quad \sum_{j \in L} \hat{w}_j(t) \cdot x_{ij} \leq y_i \quad \forall\, i \in B \tag{4.2}$$

$$\sum_{i \in B} x_{ij} = 1, \quad \forall\, j \in L \tag{4.3}$$

$$y_i \in \{0,1\} \quad \forall\, i \in B \tag{4.4}$$

$$x_{ij} \in \{0,1\} \quad \forall\, i \in B, j \in L \tag{4.5}$$

Time instant t represents an instant at which a measurement was performed of the items' sizes, leading to a new computation of the bin packing assignment, and new values for x_{ij} and y_i. Since an assignment represents a consumer reading data from the partition, an item can also be reassigned to another consumer (bin) adding a new variation to the traditional BPP.

As presented in Figure 4.1, there are three components that interact with one another in order to model the problem as a BPP, and to provide a fully dynamic pipeline capable of autoscaling based on the current load of data being produced to the partitions of interest. The monitor process is responsible for measuring the write speed of each partition the group is interested in consuming data from, which is equivalent to specifying the size of the items of the BPP. This information is then delivered to the controller, which is responsible for managing a consumer group (creating and deleting consumer instances), and mapping each partition to a consumer. The consumers are then informed of their tasks and read the data from the partitions that were assigned to them by

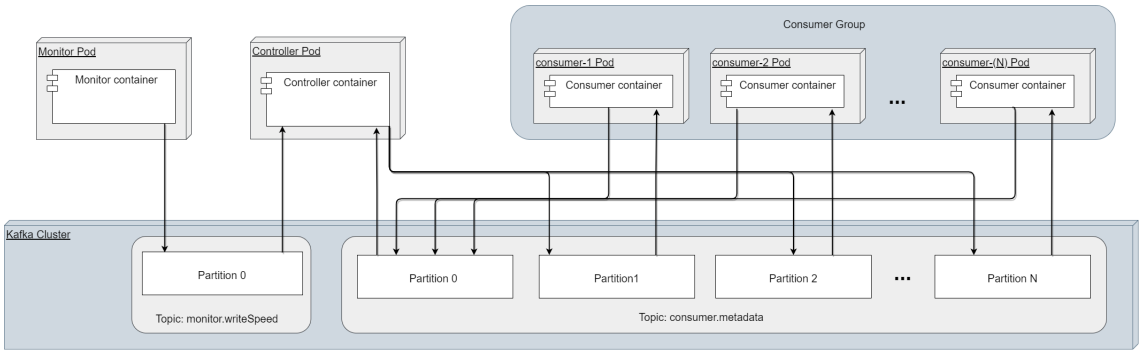


Figure 4.1: System architecture.

the controller.

4.2 Monitor

Algorithm 1: Monitor process pseudo-code.

input: admin - Kafka admin client to communicate with cluster,
producer - Kafka producer client,
topics - list of topic identifiers from which the monitor is to determine the speed of their partitions

```
1  measurementQueue ← Queue<Measurement>();
2  while true do
3      measurement ← new Measurement();
4      measurement.partitionSizes ← admin.describeLogDirs(topics);
5      measurement.timestamp ← currentTime();
6      measurementQueue.add(measurement);
7      if (measurementQueue.first.timestamp - measurement.last.timestamp) > 30s then
8          sizeDiff ← measurement.last.partitionSizes - measurement.first.partitionSizes;
9          timeDiff ← measurement.last.timestamp - measurement.first.timestamp;
10         measurementSpeed ← sizeDiff / timeDiff;
11         producer.produce("monitor.writeSpeed", measurementSpeed);
12         while (measurement.timestamp - measurementQueue.first.timestamp) > 30s do
13             measurementQueue.pop();
14         end
15     end
16 end
```

To solve the BPP, initially the controller requires as input the write speed of each partition the consumer group is interested in consuming data from. The monitor process is responsible for providing this information.

If enabled, Kafka exposes metrics to external Java processes through Java Management Extensions (JMX). Prometheus, an open source monitor process, can then be used as a wrapper to these metrics as it creates a unified model to query the data, and it also registers the data as a timeseries, allowing for a historic view of any parameter, as well as creating more elaborate metrics over the exposed cluster metrics. At the time of writing this thesis and to the best of my knowledge, there is no available metric that provides a partition's write speed. For this reason, this section aims to explain the process that is responsible for monitoring the write speed of the partitions of interest.

Kafka provides an Admin client, which can be used to administer the cluster, and also query information about it. This client/class exposes a method `describeLogDirs()` which queries the cluster for the amount of bytes each TopicPartition has. A TopicPartition is a string-integer pair, which identifies any partition (integer) within a topic (string).

Since the consumer only consumes from the partition leader, if the `replication-factor > 1`, then several partitions are excluded from the result of the previous method call, since this process is only interested in the partitions that belong to one of the topics of interest, and which are leaders.

Each time the partition size is queried by the admin client, a timestamp is appended to the measurement, and it is inserted to the back of the queue. Any query that is older than 30 seconds,

which is guaranteed to be in the front of the queue, is removed. To obtain the write speed of a single partition, the last element of the queue and the first (representing the latest and the earliest measurement of the partition size within the last 30 seconds) are used to compute the ratio between the difference in bytes and the difference in time ($bytes/s$). This is also the average write speed over the last 30 seconds.

Every topic has the parameters `retention.ms` and `retention.bytes`, which determine how long a record remains in a partition before being deleted, or how many bytes a partition retains before removing old records. When a record is deleted, the partition size reduces as it no longer reserves space for the removed record. For this process to have an accurate write speed, it is important to set both of these parameter to `-1`, which means that the record is never deleted from the partition. Ideally, the admin client would be able query for the historic size of a partition, but since this information could not be retrieved, this approach was what was implemented.

After computing the write speed for all the partitions of interest, the message has to be communicated to the controller/orchestrator that runs the algorithm to assign the partitions to the consumers. To benefit from an asynchronous approach, this monitor process communicates with the controller/orchestrator process via a Kafka topic named `monitor.writeSpeed`, as shown in Figure 4.1. The data to be inserted is the write speed of the partitions of interest, which is then consumed by the controller/orchestrator to be used as input for an algorithm's execution.

Two testing scenarios were developed to illustrate the measured partition write speeds by this method when a controlled producer is sending records at a predefined rate. The payload inserted into the partition is always 123 bytes.

The first scenario has the producer send the message at three different rates for approximately 35 seconds each, to simulate a step input. As can be seen in Figure 4.2, the monitor increases the measured write speed rate slower than the actual speed as measured by the producer. This happens because the monitor takes into consideration the first and last measurement made in the last 30 seconds, whereas the producer simply measures the produce rate since the last time it inserted a record into the partition. It takes the monitor 30 seconds to converge to the actual production speed where it then settles until the produce rate increases again.

Since a message broker receives data at variable rates from multiple producers at a time, the second scenario aims to test a noisy test case, where producers send data at different rates to a partition. This was done by having a producer wait randomly between $[0.01, 2]$ seconds before sending the payload, to increase the variability in the producer's measured speed. Provided the monitor takes into consideration the last 30 seconds from its latest measurement, it is not affected by the noisy write speed. Within these testing conditions, due to the wait time being modeled as a uniform random variable, the monitor process should stabilize at approximately the speed computed using the number of bytes produced and the expected wait time, which in this case are 123 bytes and approximately 1 second respectively. This is in fact the case as can be seen in Figures 4.3a and 4.3b, since the write speed stabilizes around $123 bytes/s$. Figure 4.3b presents a statistical description of the producer's measured write speed, that better illustrates the noise reduction in the monitor's measurement due to the computed average.

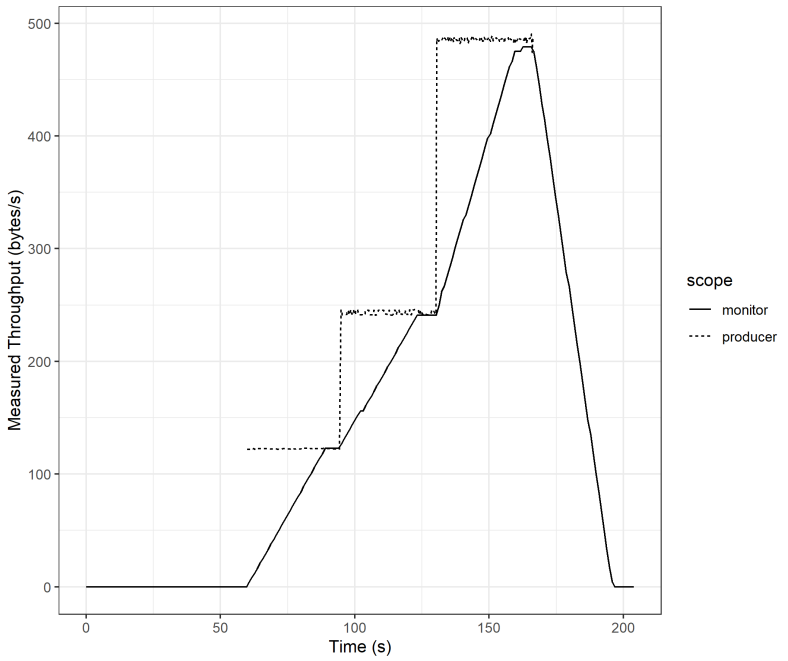
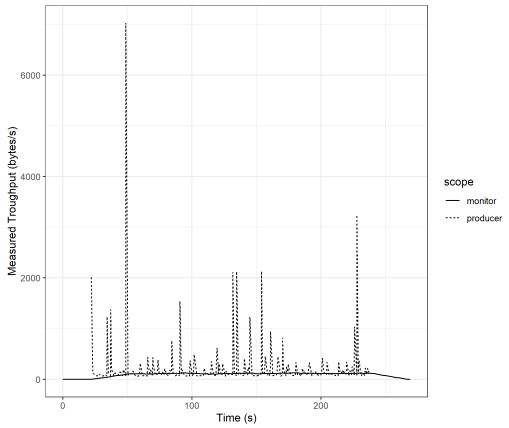
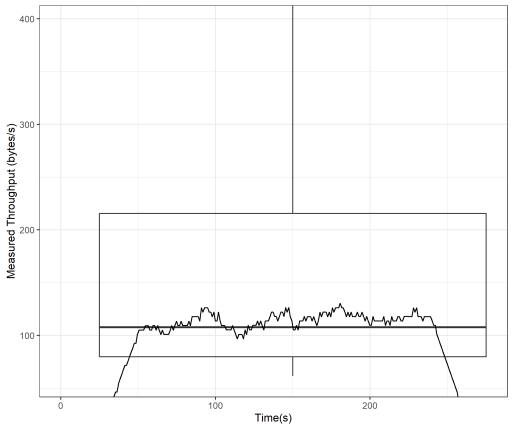


Figure 4.2: Monitor step response to three different controlled write speeds.



(a) Producer's speed measured between each production instant.



(b) Producer's speed statisctically summarized using a boxplot.

Figure 4.3: Measured write speed by the monitor and the producer, when the producer randomly waits between $[0.01, 2]s$ between 2 consecutive inserts.

4.3 Consumer

The Consumer goes through four important phases within its process, to approximate its consumption rate to a constant value when being challenged to work at its peak performance. These phases repeat cyclically until the consumer is terminated by an external termination signal.

The following Sections describe each phase in the consumer insert cycle, as illustrated in Figure 4.4.

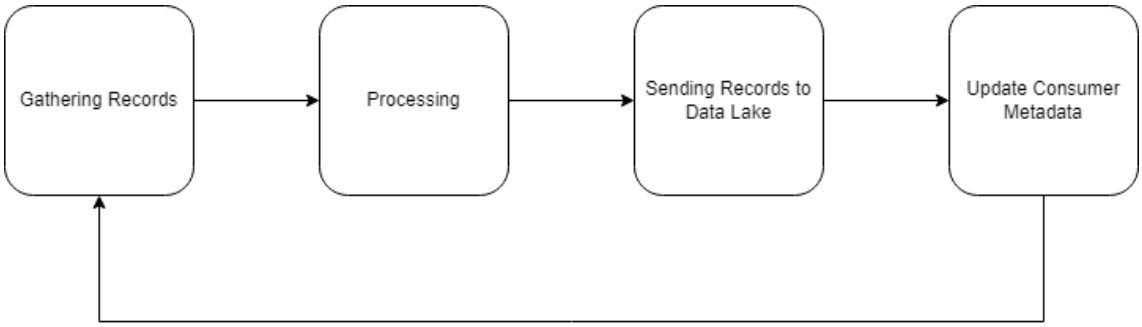


Figure 4.4: Consumer Insert Cycle.

4.3.1 Phase 1: Gathering Records

The first phase is where data is gathered from the Kafka topics before being sent to the Data Lake. The consumer is configured with two important parameters, `BATCH_BYTES` and `WAIT_TIME_SECS`, which indicate respectively, the amount of bytes the consumer waits to gather in a single iteration, and the amount of time it is allowed to wait to gather the information.

4.3.2 Phase 2: Processing

Algorithm 2: Consumer Phase 2 algorithm

input :
messages - List of Kafka messages,
mapTopicTable - Map that indicates which table a message from a topic is inserted into
output: List<BatchList>

```
1  mapTableBatchlist ← new Map[String, BatchList]();
2  for msg in messages do
3      table ← mapTopicTable.get(msg.topic());
4      batchList ← mapTableBatchlist.get(table);
5      if (batchList == None) then
6          mapTableBatchlist[table] ← new BatchList();
7          batchList ← mapTableBatchlist.get(table);
8      end
9      batchList.add(msg);
10 end
11 return mapTableBatchlist.values()
```

The second phase is where the data gathered in the previous phase is prepared to be sent into the datalake, which in this case is BigQuery. Each record fetched from a Kafka topic, represents a new row in a BigQuery table.

To insert data into BigQuery, Google's Python Bigquery Client is used. Each post request made with this client, has a batch of rows sent to a single table, and it is also limited to no more than $10 Mbytes$ per request.

Since the gathered records originate from multiple topics, each row has to be batched with data of its kind, which are rows that are intended to be sent to the same BigQuery table. A single instance of a Batch, a data structure created to agglomerate rows intended for the same BigQuery table, holds all the rows which will be sent in a single post request through Google's python BigQuery client API. Due to the imposed limit of a post request, a Batch has a payload limit of $5Mbytes$. A BatchList, is another data structure created to group instances of type Batch that refer to the same table.

To process the data, a map stores the link between a table's ID and the BatchList instance assigned to it. When analyzing a message, its topic metadata indicates which table it is directed to, which in turn points to the BatchList it must be added to. When adding a message to a BatchList instance, this class controls to which Batch it is assigned, based on the size of the row, and the size of the last created Batch within this data structure.

4.3.3 Phase 3: Sending Records to the Data Lake

This is the final stage of the consumer's insert cycle. There is a list of BatchList instances, each having to be sent to a single BigQuery table. To optimize the insert cycle, this phase is done asynchronously with respect to all the batches within the list of BatchLists contained in the map specified in Algorithm 4.3.2. In other words, each batch corresponds to an asynchronous request to insert rows into a table. After successful insert of the rows into the BigQuery table, the messages are then committed to Kafka.

4.3.4 Phase 4: Update Consumer Metadata

Algorithm 3: Consumer Phase 4 algorithm

```
   input: consumer - consumer instance used to fetch information from the Kafka topics.
 1 Set<Partition> current ← consumer.getCurrentState();
 2 Set<Partition> future ← current.copy();
 3 Queue< Set<Partition> > changeStateQueue ← consumer.consumeMetadata();
 4 while changeState ← changeStateQueue.pop() do
 5 |   if changeState.type == "StopConsumingCommand" then
 6 |   |   future ← future - changeState.partitions;
 7 |   else
 8 |   |   future ← future ∪ changeState.partitions;
 9 |   end
10 end
11 toAssign ← future - current;
12 toStop ← current - future;
13 consumer.incrementalAssign(toAssign);
14 consumer.incrementalUnassign(toStop);
15 consumer.persist();
```

The consumer has to be informed by the Controller, as to which partitions it gets data from. For this purpose, to allow both the controller and the consumers to work asynchronously, a message queue is the ideal structure for this purpose.

Following event sourcing patterns, the current state of a single consumer should be attained by consuming every message that was directed to it, which enforced a change in state. With a slight modification, Kafka is used for this communication between the controller and the consumers by creating a single topic named `consumer.metadata`.

Maximum efficiency in data conveyed between each process, occurs when a single process only reads messages which are relevant for its functioning, without having to ignore messages or data that it receives. As such, each consumer has to be assigned a separate queue in the `consumer.metadata` topic, which in Kafka represents a distinct partition per consumer.

As we further describe in Section 4.4, the consumer knows which partition to consume from through its deployment's name. When the controller desires to communicate with this consumer, it simply has to send a record to its partition.

Each record published into this topic has to have the same AVRO schema as specified by the schema in Appendix A.1. A command can be either a:

- StartConsumingCommand - The consumer is to start consuming from each of the partitions within the record.
- StopConsumingCommand - The consumer stops consuming from the partitions specified in the record.

The fourth phase starts with the consumer determining whether there are any messages in its metadata queue (the partition it was assigned from the `consumer.metadata` topic). If there are none, the consumer's state isn't changed, and the phase is finalized. Otherwise, if there are any messages in the queue, the consumer goes through the process of consuming all the messages in the metadata partition, and adds them to a Queue instance `changeStateQueue`.

The next step is to process the `changeStateQueue`. Initially, a set is created where each element represents a partition from a topic the consumer is currently consuming from. This set is stored in the variable `current`. A copy of this set is made and assigned to a variable `future`. While the queue is not empty, the front-most element is removed and assigned to a temporary variable `changeState`. Depending on the command, if the record requests the consumer to start consuming from its set of partitions then a union operation is performed between the `future` and the `changeState`. In turn, if the record is of type "StopConsumingCommand", the difference between `future` and `changeState` is computed.

Having processed the whole `changeStateQueue`, future holds the new state the consumer has to change into. Lines 11 to 14 change the consumer's assignment into future's state.

4.3.4.1 Persisting Metadata

The consumer's final stage has it persisting its metadata for the case it unexpectedly fails and has to pick up the work it was last performing. This avoids having the consumers reprocess the

whole metadata queue to reach its last state. The data to be persisted is the set of partitions it is currently consuming from. A successful change in consumer state would only occur after the consumer successfully persists the data to its persistent volume.

When dealing with Kubernetes pods, by default the pod's storage is ephemeral, which means that the group of containers starts with a clean slate, and when terminated the data written to disk is cleaned, making it inacessible for future reads outside of a single pods lifetime.

Kubernetes provides volumes which can be of type ephemeral or persistent (its lifetime is independent of the pod's). A persistent volume (PV) can be created static or dynamically, and is a resource within the cluster. A persistent volume claim (PVC), is a method of abstraction which allows a user to request for storage. This request, then tries to match the claim to one of the available resources (PVs), and if there are none available, then a new persistent volume can be created dynamically if the Storage Class is defined. The mapping between PV and PVC is one-to-one.

This consumer uses both types of volumes, since the downwardAPI is of type ephemeral and it provides the consumer with its context, giving it access to its deployment's name (data the consumer requires for it to know which partition to consume from in the `consumer.metadata` topic). As for the persistent volume, the consumer will use this type of volume to persist the data for its current consumption state. If the consumer fails unexpectedly, then on startup, it just has to verify its state on the volume, and in case it was performing any tasks, it picks up where it left off.

When stopping a consumer, to safely terminate its tasks, the controller sends a StopConsumingCommand for all the partitions the consumer is currently assigned to. After the consumer acknowledges it acted to the command (it sends a StopConsumingEvent back to the controller), the controller then terminates the pod. This communication between the controller and the consumer is better described in Section 4.4.7.3.

The fact termination only occurs after the consumer updates its metadata, implies the persistent volume is also updated to an empty set, which allows a new consumer of the same deployment to start off with a clean slate as should be the case, since the consumer was gracefully terminated.

4.3.5 Consumer Maximum Capacity

The way the problem is formulated assumes the consumer is capable, when required, to achieve a maximum data consumption rate (still to be defined), analogous to the capacity of a bin in a BPP.

With the goal of attaining a value for this maximum bin capacity, the consumer was tested in 3 different scenarios, each requiring the consumer to be working at peak performance. Peak performance is defined as the case when the sum of the bytes still to be consumed from all the partitions the consumer is assigned to is bigger than `BATCH_BYTES`. The three testing scenarios aim to test the consumer's throughput while varying the number of tables it has to insert data into, the average amount of bytes available in each of the assigned partitions, and the number of partitions assigned to it.

The consequence of having more tables where data has to be inserted directly affects the third phase by increasing the amount of asynchronous requests that have to be made. Reducing the average amount of bytes available in the partitions assigned, but still being able to make the consumer work at full capacity, aims to test the first phase of the algorithm, where the data has to be consumed from the partitions assigned from Kafka. This is the case since this high-level consumer is based on the Kafka client provided by the `confluent_kafka` package. When the client begins, it starts a low-level consumer implemented in C that runs in the background. As the low-level consumer runs, it buffers messages into a queue until the high level python consumer requests for what it has consumed with either `poll()` or the `consume()` methods. The difference between these two methods is that the first returns a single message whereas the second returns a batch of messages defined by `num_messages`, from the low level client's buffer.

To improve throughput, when a consumer requests for data from the broker, the broker attempts to batch data together before sending it back to the consumer. Configuration parameters `fetch.max.bytes` and `fetch.max.wait.ms` define how a single request is handled by the broker, wherein the first determines the maximum amount of bytes returned by the broker, and the second, the amount of time the broker can wait before returning the data if `fetch.max.bytes` is not satisfied.

Reducing the average amount of bytes in each of the partitions assigned, leads to the consumer having to perform more requests to fetch the same amount of data defined by `BATCH_BYTES`.

The number of partitions assigned to the consumer is expected to influence the time it takes for the consumer to gather `BATCH_BYTES`, by increasing the amount of requests required to fetch the data, as assigning partitions increases the probability of the consumer having to communicate with more brokers.

Another important definition is that the consumer is said to have reached its steady state, when it is capable of fetching `BATCH_BYTES` prior to `WAIT_TIME_SECS` being triggered in its first phase. This reflects that the low-level consumer is capable of gathering the necessary amount of bytes into its buffer, while the high-level consumer is running all the remaining phases except the first of the insert cycle.

For the following test cases, `BATCH_BYTES = 5000000` and `WAIT_TIME_SECS = 1`. These values inherently reflect the time it takes for a consumer to go through each of its phases, and the rate at which it can consume data.

Table 4.1: Testing conditions to obtain consumer maximum throughput measure.

Test ID	Total Bytes	Average Bytes	Number of Partitions	Number of Tables
Test 1	648 *Mbytes*	20 *Mbytes*	32	1
Test 2	100 *Mbytes*	0.86 *Mbytes*	116	5
Test 3	678 *Mbytes*	4 *Mbytes*	144	5

From Table 4.1, test 1 has the consumer fetching records which are all directed to the same table, and every partition it visits has enough bytes to satisfy the `max.fetch.bytes` condition, optimizing the throughput between the low-level consumer and the brokers, as the data is batched

together. Since there is only one table to send the data to, the third phase is also optimized as there will only be a single BatchList instance.

As for Test 2, the increased number of partitions and the reduced average amount of bytes in each partition has the consumer polling more brokers to gather `BATCH_BYTES`, increasing the time the consumer takes in the first phase. Although the time taken in this phase increased, the average time it takes the consumer to fetch `BATCH_BYTES` from the kafka cluster, indicates that the consumer still manages to reach its steady state.

Lastly, Test 3 combines both of the previous scenarios, where there is more data per partition to be sent back to the consumer, although some iterations require the consumer to send data to five different tables since the data originates from partitions that belong to different topics.

The metrics that were deemed important for the analysis of the consumer's behaviour, were: Time of Phase 1 (Δt_{P1}); Time of Phase 2 (Δt_{P2}); Time of Phase 3 (Δt_{P3}); Total cycle time (Δt); Measured cycle throughput. Each of these parameters is statistically summarized in Table 4.2.

Table 4.2: Statistical summary of the metrics.

Test ID	**Summary Measure**	Δt_{P1}	Δt_{P2}	Δt_{P3}	Δt	**Throughput**
Test 1	Average	0.394	0.000775	1.69	2.08	2418855
	Standard Deviation	0.0338	0.00620	0.0941	0.101	97853
Test 2	Average	0.462	0.00684	1.72	2.18	2307501
	Standard Deviation	0.125	0.001600	0.0946	0.169	171231
Test 3	Average	0.397	0.00896	1.68	2.08	2418220
	Standard Deviation	0.0466	0.00619	0.105	0.121	132017

Figure 4.5 demonstrates how the consumer is capable of maintaining a stable speed above a threshold of $2Mbytes/s$, when it finds itself in its steady state (capable of consuming $5Mbytes$ before `WAIT_TIME_SECS` is triggered), sharing a mode between the three different testing scenarios around $2.3Mbytes/s$.

4.3.5.1 Bin Capacity

Two measures are defined in order to run the heuristics described in Section 4.4. The `ALGORITHM_CAPACITY`, is the constant bin size that will be used when running all the heuristic algorithms when solving the dynamic BPP, whereas the `CONSUMER_CAPACITY` is the consumer's maximum capacity, which if exceeded, must trigger a new configuration as for which partition should be assigned to which consumers.

The reason for having defined these two configuration parameters are due to the problem's dynamic nature. Since the measured write speed for each partition varies between each measurement, after executing the BPP algorithm for a single measurement, it is common to have bin's filled close to their capacity, which would lead to excessive rebalancing computations if the algorithm and consumer capacity were defined using the same value. Defining the Consumer's capacity to a value higher than the algorithm's perceived capacity, reduces the number of rebalances triggered, at the cost of a reduced bin capacity.

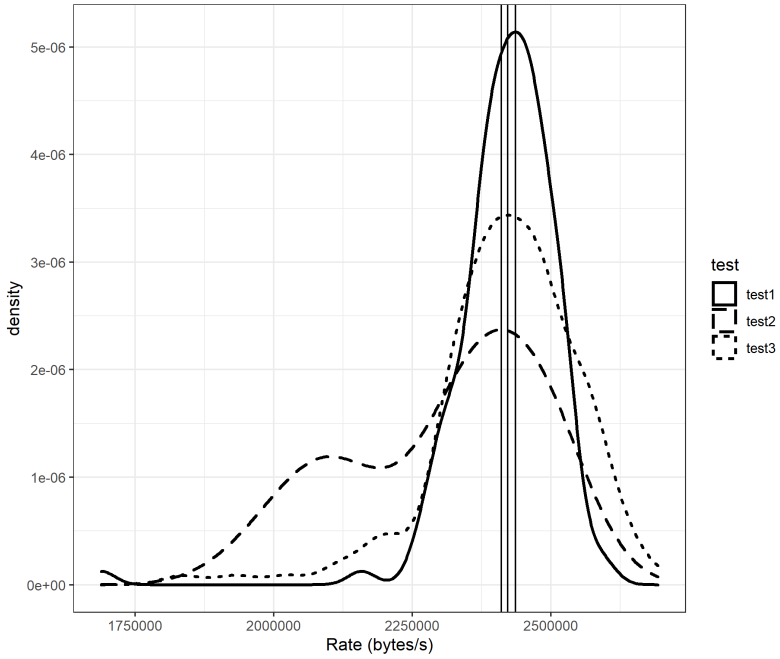


Figure 4.5: Density Plot for the consumer's measured throughput in the three testing conditions.

From Figure 4.5, the Consumer's capacity is set to $2Mbytes/s$, whereas the Algorithm's capacity is defined as $1.5Mbytes/s$. These are the values used for the remaining part of the work. It is important to note that these values are only valid when the consumers are deployed in the environment where they were tested. Having executed the test in multiple environments, it can be said that the consumer's maximum capacity does not change between consumer instances as long as the environment is similar for all consumers.

4.4 Controller/Orchestrator

The controller is the component of the system which is responsible for orchestrating and managing the consumer group that is intended to consume the data from the partitions of interest. The problem is modeled as a dynamic bin packing problem, where the size of each item is equivalent to the write speed of each partition, and the size of the container represents the maximum achievable speed of a consumer which has been provided by the data presented in Section 4.3.5.1.

At any given time, no consumer within the consumer group can have its capacity exceeded by the cumulative write speed of the partitions assigned to it, and if that is the case, then a rebalance has to be triggered. The same applies in case a partition has not yet been assigned a consumer.

After the controller determines the state in which it wants its consumer group, it then creates the consumers that don't yet exist, communicates the change in assignment to each consumer in the group, followed by deleting the consumers that are not required in the new computed group's state.

4.4.1 Rscore

Within this problem's context, rebalancing inevitably implies having the consumer group stop consuming from a partition while the controller is reassigning the partition from one consumer to another (as there cannot be a concurrent read from the same partition by members of the same group).

A new measure is proposed to compute the total rebalance cost (Rscore) of a new group's configuration, which aims to reflect the impact of rebalancing a set of partitions in an iteration due to temporarily stopping data consumption from all rebalanced partitions during the rebalance phase (described in Section 4.4.7.3). As such, the Rscore computes the rate at which information is accumulating, in consumer iterations per second.

Table 4.3: Data to compute the Rscore for an iteration.

Symbol	Description
P_i	set of partitions that were rebalanced in iteration i
$s(p)$	The write speed of partition p
C	Constant that represents the maximum consumer capacity from Section 4.3.5.1

Provided the data presented in Table 4.3, the following equation presents the rebalance cost (Rscore) for a single iteration i (R_i).

$$R_i = \frac{1}{C} \sum_{p \in P_i} s(p) \tag{4.6}$$

4.4.2 System Design

As has been described, there are three different components in the system that work together to get the data from Kafka into BigQuery. To do so, each component has to be able to communicate with one another to inform any change in state.

As shown in Figure 4.1, page 27, there are two main topics that will be responsible for providing an asynchronous communication between the 3 different entities of the system. The `monitor.writeSpeed` topic, is where the monitor process sends the speed measurements for the controller to consult, whereas the `consumer.metadata` topic is where the controller sends messages to each consumer to inform the change in their state. Partition 0 of the `consumer.metadata` topic is reserved for communication which is intended to reach the controller.

When using a Kafka topic to communicate between the consumers and the controller there are certain requirements that need to be complied with. The first, is that when the controller desires to communicate with a consumer from the consumer group, it has to be guaranteed that the message reaches the intended consumer.

Secondly, as will be common with this system, the controller will create and delete a single consumer multiple times in its lifecycle. The message offset a consumer starts on after restarting, has to be precisely the one where it left off before being shut down by the controller. This can be

done leveraging kafka's `group-id` property defined in each consumer client, so the system just has to guarantee that the consumer gets the same id as before, and that no other client with the same id consumed messages from this consumer's queue.

Thirdly, each consumer only reads messages which are intended to it. This would represent maximum efficiency in the information transmitted between both controller and consumer, since no process is reading messages or data that has to be ignored. To better understand this design requirement, the following scenario is described: It might happen that a certain number of consumers satisfies the controller's conditions, without any need to scale up or down the group. This would imply that the control messages sent by the controller are only read by the current active consumers. When a new consumer is started by the controller, if the messages are shared, then there is a big queue of messages that have not yet been read by the new consumer since it was last up. For the new consumer to be able to read new control messages sent by the controller, it would first have to read all messages that were not intended to it prior to it starting up.

Lastly, to guarantee temporal consistency in the control messages sent to a single consumer, it is clear that a control message can only be sent to a single partition, as kafka only guarantees message read order when the messages belong to the same partition. Kafka already does this by allowing a message to have a key attribute which is then hashed to determine the partition in which the message is to be inserted. The issue with this approach is if the number of partitions in the `consumer.metadata` topic increases, the key might not send the messages to the same partition as before. As such, each consumer is given an incremental id $(1, 2, ...)$, which represents the partition where change in state information has to be sent for the consumer to read (both controller and consumer are aware of this id).

4.4.3 State Machine

The controller can be defined by a state machine, intended to continuously manage a group of consumers and their assignments. The following sections will further describe each of the states, which are illustrated in the state machine in Figure 4.6.

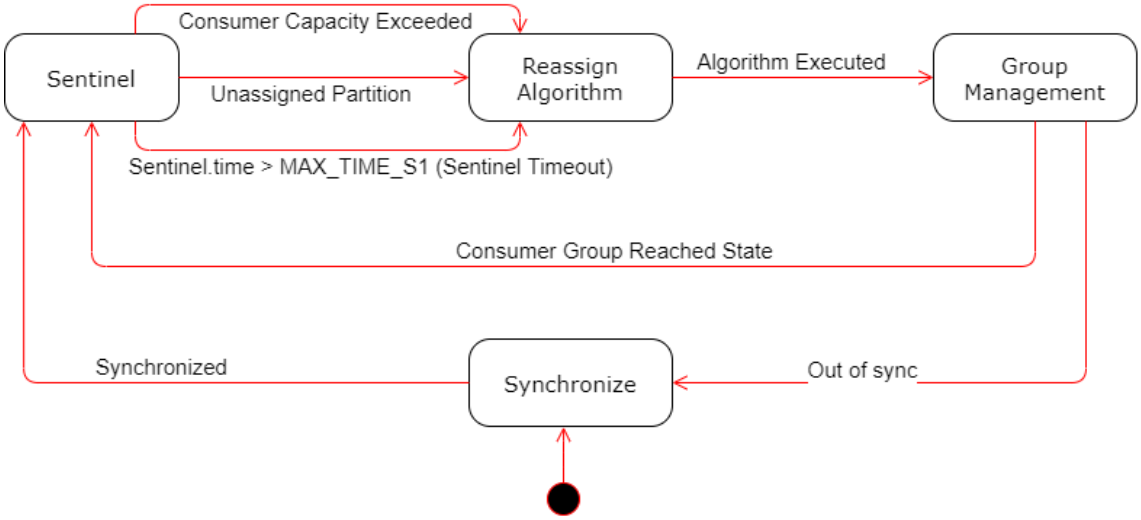


Figure 4.6: Controller State Machine.

4.4.4 State Sentinel

This state is where the controller uses the information provided by the monitor component described in Section 4.2, to determine whether it has to recompute the consumer group's assignment. The first step the controller goes through in this state is to read the last speed measurement the monitor component added to the `monitor.writeSpeed` topic.

The controller then updates each of the partition's speeds, which in turn updates the cumulative speed of each consumer. As can be seen in Figure 4.6, there are three transitions that can lead to the controller re-computing the current consumer group's assignment:

1. **Consumer Capacity Exceeded** - This transition is triggered if the cumulative speed of all the partitions assigned to any consumer, exceeds `CONSUMER_CAPACITY` obtained in Section 4.3.5.1.

2. **Unassigned Partition** - If there is any partition within the last speed measurement that is not currently assigned to a consumer, then this transition is triggered.

3. **Sentinel Timeout** - One of the controller's configurations, is a parameter `MAX_S1_TIME` which indicates the maximum amount of time the controller can spend in the sentinel state without triggering a rebalance. This trigger is to have a condition that runs the algorithm to verify if downscaling is viable, as the other triggers are directed to upscaling conditions.

4.4.5 State Reassign Algorithm

This state receives as input the current consumer group's assignment and the remaining unassigned partitions, and produces as output a new consumer group assignment, which is to be defined as the group's future state.

This state uses only approximation algorithms to determine the group's future assignment. The algorithms implemented are the already existing Next Fit, First Fit, Worst Fit, Best Fit, and each of the previous algorithms' decreasing versions, and modified versions of the Best and Worst Fit algorithms, which are to be further described in Section 4.4.6.

In the implementation of the existing approximation algorithms (AA), another step was included in the bin creation process which does not affect the outcome in terms of number of consumers of the AA. If the consumer that is currently assigned to the partition has not yet been created in the future assignment, this is the bin that is created, otherwise, the lowest index bin that does not yet exist is the one created.

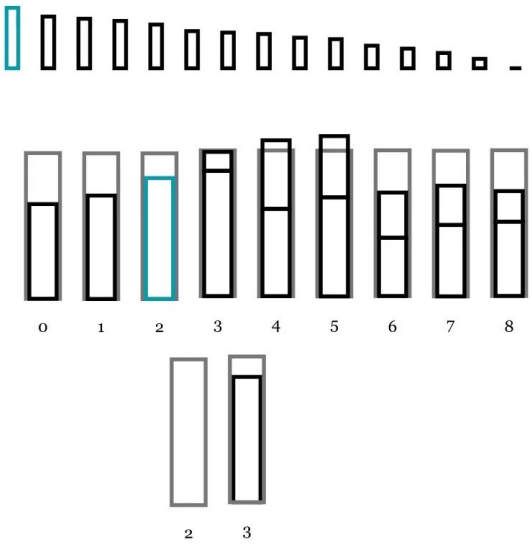


Figure 4.7: New bin creation procedure when executing an existing approximation algorithm.

Figure 4.7 illustrates this process, wherein the first row of consumer's (grey containers) represents the current consumer group's state, and the second the new consumer group assignment that is being computed by the controller. Partitions are represented in blue and black, blue being the partition currently being analyzed by the controller. In this example, the blue partition currently finds itself assigned to consumer 2. When analysing the partition, the controller verifies if the partition fits in any of the existing consumers. Since it does not fit, and another consumer has to be created to assign this partition, the controller verifies if the consumer that is currently assigned already exists in the new group's state. Since this is not the case, the controller proceeds to creating the same consumer and assigning the current partition to the new bin.

In view of the fact that a group of consumers (bins) already exists at the moment the approximation algorithm runs to provide a solution to the BPP, this approach is preferred to simply creating the lowest index bin available as it reduces the amount of partitions rebalanced. This is not presented as a modification to the existing algorithms, it simply adapts these to the added rebalance concern.

4.4.6 Modified Any Fit Algorithms

The motivation to modifying the existing Any Fit algorithms, is that the existing algorithms only focus on reducing the amount of bins used to pack a set of items, disregarding the cost associated with reassigning items.

Since the controller has access to the speed measurements of all partitions and it does not require to read and assign each partition in any particular order, this bin packing problem can be categorized as offline.

Given the current consumer group's state (the partitions assigned to each consumer), and a set of unassigned partitions, the modified algorithms differ from the previous decreasing versions of the Any fit algorithms described in Section 3.2, wherein this type of algorithm does not sort the set of items, but in turn the consumers of a consumer group based on their assignment.

There will be 2 main approaches to sorting the consumers:

- Sorting each consumer based on the cumulative speed of all partitions assigned to it (cumulative sort);

- Sorting each consumer based on the partition assigned to it that has the biggest measured write speed (max partition sort).

After sorting the current consumer group using one of the above strategies, for each consumer in the sorted consumer group, the partitions assigned to the consumer are sorted based on their write speed. From smallest to biggest, each partition is inserted into one of the bins that have already been created in the future assignment, based on one of the any fit strategies, which can either be the Best or Worst Fit strategy. If the insert is successful, then the partition is removed from the sorted list of partitions, otherwise, if there is no existing bin that can hold the partition, then the current consumer assigned to it is created.

The remaining partitions in the sorted list are now inserted into the newly created bin, from biggest to smallest, and removed from the sorted list if successful. If a partition does not fit into the newly created consumer, then the remaining partitions in the list of sorted partitions are added to the set of unassigned partitions.

After performing the same procedure over all consumers, there is now a set of partitions which have not been assigned to any of the consumers in the future assignment. The final stage involves first sorting the unassigned partitions in decreasing order (based on their measured write speed), and each partition is assigned a consumer from the new consumer group using their respective any fit strategy.

Algorithm 4: Modified Any Fit Pseudo Code

```
input  : current consumer group C and set of unassigned partitions U
output: consumer group N which is the next state for the consumer group
N ← new ConsumerList(assign_strategy=("BF" | "WF"));
sorted_group ← sort(C, sort_strategy=("cumulative" | "max partition"));
for consumer ∈ sorted_group do
    Set<Partition> partitions ← consumer.assignedPartitions();
    List<Partition> sorted_partitions ← sort(partitions, reverse=True);
    for i ← length(sorted_partitions)-1 to 0 do
        partition ← sorted_partitions[i];
        result ← assignExisting(N, partition);
        if result == False then
            break;
        end
        remove(sorted_partitions, partition);
    end
    if sorted_partitions.length() == 0 then
        continue;
    end
    createConsumer(N, consumer);
    for partition ∈ sorted_partitions do
        result ← assignCurrent(N, partition, consumer);
        if result == False then
            break;
        end
        remove(sorted_partitions, partition);
    end
    extend(U, sorted_partitions);
end
sorted_unassigned ← sort(U, reverse=True);
for partition ∈ sorted_unassigned do
    assign(N, partition);
end
return N
```

Table 4.4: Modified implementations of the any fit algorithms.

Algorithm	Assign Strategy	Consumer Sorting Strategy
Modified Worst Fit	Worst Fit	cumulative write speed
Modified Best Fit	Best Fit	cumulative write speed
Modified Worst Fit Partition	Worst Fit	max partition write speed
Modified Best Fit Partition	Best Fit	max partition write speed

4.4.6.1 Testing Procedure

To compare the performance between the implemented algorithms, a randomized stream of speed measurements (which represents the partition size in the BPP) was generated, and each algorithms was compared with respect to the same streams of data.

Table 4.5: Data to generate the measurement streams.

Symbol	Description
P	Set of partitions of interest for the consumer group.
$s_i(p)$	Speed for a partition $p \in P$ at an iteration i
$\phi(\delta)$	Uniform random function that selects a value between $[-\delta, \delta]$
C	Equivalent to `ALGORITHM_CAPACITY`

To generate a stream of measurements, given N (the number of measurements desired) and δ (maximum relative speed variation between two sequential iterations), at first the initial speed $s_0(p)$, $\forall p \in P$ has to be defined. Four different approaches were tested:

1. Choosing a random value for the initial speed of each partition between $[0,100]\% \cdot C$;
2. Setting all partition's initial speed to $0\% \cdot C$;
3. Setting all partition's initial speed to $50\% \cdot C$;
4. Setting all partition's initial speed to $100\% \cdot C$.

Given that there was no significant difference on the outcome when the initial parition speed varied, for the remaining part of this section we will consider the case where the initial speed was randomly selected between $[0,100]\% \cdot C$ for all partitions $p \in P$.

Therefore, given $s_0(p)$ $\forall p \in P$, the remaining measurements were obtained using:

$$s_i(p) = s_{i-1}(p) + \frac{\phi(\delta)}{100} \cdot C, \quad \forall\, p \in P \,\wedge\, i \in \{1,2,...,N-1\} \tag{4.7}$$

Using the aformentioned procedure, 6 different streams of data were generated by setting $N = 500$ and setting δ to a value belonging to the set $\{0,5,10,15,20,25\}$ for each stream of data (δ does not change within a stream).

4.4.6.2 Testing Metrics

The metrics used to compare the performance between the algorithms are the Cardinal Bin Score ($CBS_\delta(a)$) and the Average RScore ($E^a_\delta(R)$) over all iterations of each stream.

Table 4.6: Data to compute the cardinal bin score for a stream of measurements.

Symbol	Description
A	Set of algorithms implemented.
$z_i^{\delta}(a)$	number of bins used in iteration $i \in \{0,1,...,N-1\}$ for a stream defined by δ by an algorithm $a \in A$.
R_i^a	Rscore in an iteration $i \in \{0,1,...,N-1\}$ for an algorithm's ($a \in A$) assignment.

The cardinal bin score is calculated using the following expression:

$$CBS_{\delta}(a) = \frac{1}{N}\sum_{i=0}^{N-1}\frac{z_i^{\delta}(a) - min_{b\in A}\{z_i^{\delta}(b)\}}{min_{b\in A}\{z_i^{\delta}(b)\}},\ \forall\, a \in A\ \wedge\ \delta \in \{0,5,10,15,20,25\}. \tag{4.8}$$

The expected value of the Rscore, is used to compare the rebalance cost for a single stream of data, and is computed as follows:

$$E_{\delta}^{a}(R) = \frac{1}{N}\sum_{i=0}^{N-1} R_i^a, \quad \forall\, a \in A\ \wedge\ \delta \in \{0,5,10,15,20,25\}. \tag{4.9}$$

4.4.6.3 Test Results

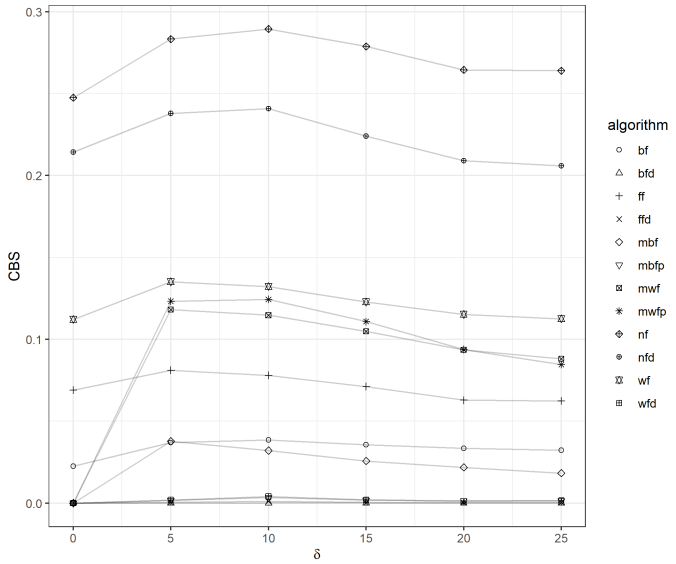


Figure 4.8: Cardinal Bin Score (CBS) for all implemented algorithms.

In Figure 4.8, the worst performing algorithm is the next fit followed by its decreasing version, which is due to every time a partition has to be assigned, the algorithm only verifies if it fits in the last created bin, as opposed to considering all existing bins. The remaining any fit decreasing algorithms, are the ones that perform the best, with the best fit decreasing consistently presenting the best results.

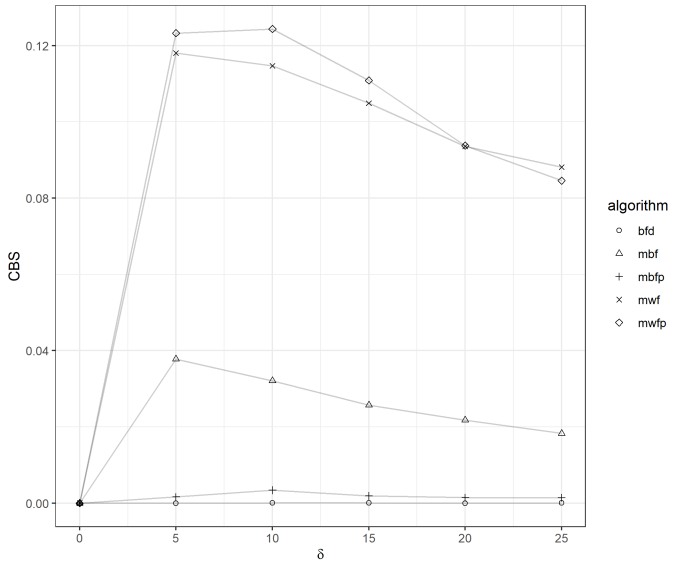


Figure 4.9: Cardinal Bin Score (CBS) filtered to present the modified and the BFD algorithms.

As for the modified versions of these algorithms, due to its sorting strategy, MBFP shows the best results. It is also worth noting that for smaller variabilities, the modified algorithms behave similarly to the online versions of their any fit strategy with respect to the CBS, since the partitions aren't necessarily assigned from biggest to smallest. On the other hand, the higher the delta, the bigger the variability, which also leads to more rebalancing, having the modified algorithms behave more like the decreasing versions of their fit strategy.

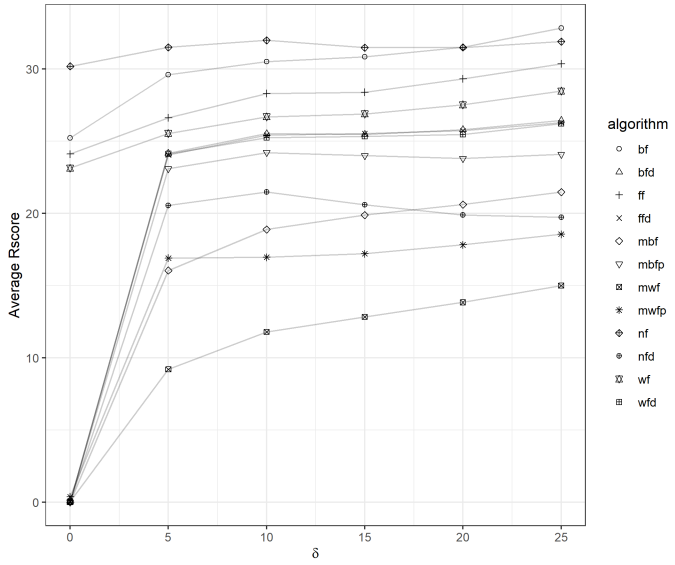


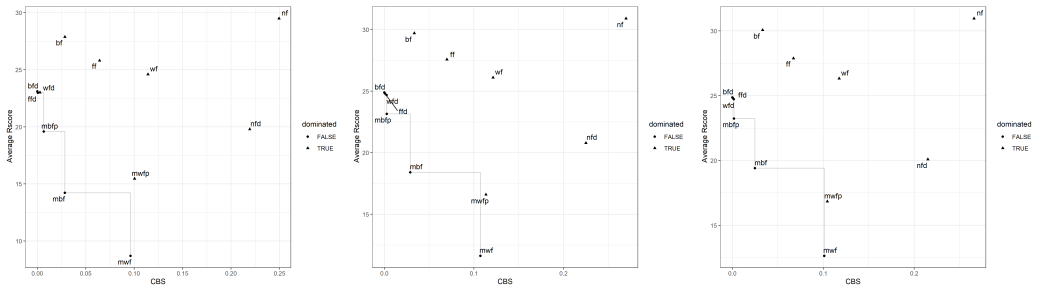
Figure 4.10: Impact on Rscore for different Deltas (random initial partition speed).

As can be seen in Figure 4.10, the modified algorithms present the best average Rscore for each stream of measurements along with the NFD. The reason why the NFD presents such a result, is due to the increased amount of consumers it creates to assign new partitions, which due to the

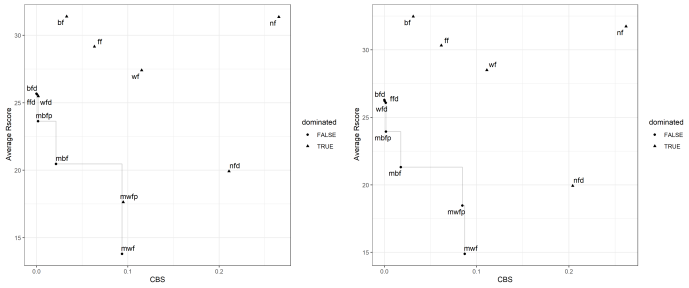
procedure exemplified in Figure 4.7, will assign the partition to the same consumer it is currently assigned to, as long as it has not yet been created in the future assignment.

For a similar reason, the modified algorithms that perform the best with regards to the Rscore, are also the ones that perform worst (compared to the remaining modified algorithms) when evaluating the CBS.

To select the algorithm to use within the controller, there is a trade-off between the aforementioned testing metrics. On account of the added rebalance concern within the modified algorithms, these present an improvement with regards to the rebalance cost when compared to the existing approximation algorithms.



(a) Pareto front for $\delta = 5$. (b) Pareto front for $\delta = 10$. (c) Pareto front for $\delta = 15$.



(d) Pareto front for $\delta = 20$. (e) Pareto front for $\delta = 25$.

Figure 4.11: Pareto front for different deltas comparing the Cardinal Bin Score and the Average Rscore.

The pareto front is a way of finding the set of solutions that are most efficient, provided there are trade-offs within a multi-objective optimization problem.

Excluding MWFP, the modified algorithms are consistently a part of the pareto front as shown in Figure 4.11, which implies these are a competitive option as to which algorithm to pick for the reassign algorithm to be executed in the controller. The algorithm that shows the best Rscore for the different variabilities is the MWF, whereas the modified algorithm that performs the best relative to the Cardinal Bin Score is the MBFP.

4.4.7 State Group Management

This state is where the controller informs each consumer of their change in state. Since there cannot be any concurrent read of a partition by two consumers of the same consumer group, when rebalancing a partition, the controller has to: inform the consumer currently assigned to the

partition to stop consuming from it; wait for confirmation that the consumer stopped consuming the data; inform the new consumer of its new assignment.

This state not only handles this message exchange but it also creates and deletes consumer and persistent volume resources from the Kubernetes cluster.

4.4.7.1 Difference between two Group States

The current consumer group's state is represented by a list of consumers, each having their own assignment (set of partitions). The new computed group's state is defined as the next state, and is also a list of consumer's each having an assignment, but representing the state the controller wants its consumer group to achieve.

Similar to computing the difference between two sets, when computing the difference between two consumer lists the procedure involves iterating over each consumer (in both states), and calculating the difference between the two consumer assignments (set of assigned partitions). There are three scenarios that result in different actions the controller has to perform, for a given position *i* of each consumer list:

- Next state has a consumer at *i* but the current state does not - This means that the consumer has to be created by the controller, and the partitions assigned to the consumer in position *i* of the next state, have to be associated with a StartConsumingCommand for this same consumer.

- Next state does not have a consumer at *i* but the current state does - The partitions associated with the consumer at position *i* of the current state have to be included in a StopConsumingCommand directed to this consumer, and the consumer has to be removed from the consumer group.

- Both next state and current state have a consumer at position *i* - This means that no creation or deletion operation has to be performed for this bin, and the operations to perform in this case are only communicating to the already existing consumer the difference in its assignment.

  The same consumer has two different sets of partitions assigned to it represented in the two different group states. `next_assignment` will denote the set of partitions attributed to the consumer's state in the next group's context, and `current_assignment` is defined as the consumer's set of partitions in its current state.

```
1 partitions_stop = current_assignment - next_assignment
2 partitions_start = next_assignment - current_assignment
```

  The resulting set of partitions within `partitions_stop` have to be included in a StopConsumingCommand, whereas the partitions in `partitions_start` in a StartConsumingCommand, directed to consumer *i*.

4.4.7.2 Managing the Consumer Group in the Kubernetes Cluster

Each active consumer in the current group's state represents a deployment with a single replica (pod), since each consumer requires a volume to persist its data, which implies having a different volume for each pod. Assigning different volumes to a set of pods is possible with stateful sets, but when removing an instance from the set, only the highest index pod can be removed. In this context, this does not provide with the granular control required when deleting a consumer.

Consequently, each consumer is given an individual ID through its deployment's `metadata.name` property, a value that can be obtained within the pod using the downwardAPI, that provides a pod with its context. This individual ID assigned to the deployment's name, is the one used to inform the pod of its metadata partition to consume from in the `consumer.metadata` topic.

To allow the controller to create, list and delete the resources within the cluster, a Kubernetes service account is used to authenticate the controller, which is then given permissions for the aforementioned operations through a Kubernetes Role. In this scenario, the controller's service account is linked with a Role object that has permissions to create, list and delete deployment and persistent volume claim resources.

Since all consumers have to be able to persist data, each consumer has to be mapped to a persistent volume using a persistent volume claim. If it is the first time a consumer with a given ID is being spawned, the controller has to dynamically create and map a persistent volume claim to the consumer's deployment.

To simplify the process, two template yaml files (Annexes A.2, A.3) are used, one for creating persistent volume claims (PVC), and another used to create deployments. The controller is only responsible for changing the template PVC ID when creating it. When the controller has to create the deployment, it has to reference the created PVC in the template deployment, and change the deployment's ID to the incremental ID attributed by the controller.

As an example, if the controller has to create a consumer whose ID is 5, it would go through the following steps:

1. If the PVC with name `de-consumer-5-volume` does not yet exist, change the PVC metadata.name parameter to `de-consumer-5-volume` in the template yaml file (Annex A.2), and send the create request with the body containing the modified yaml file;

2. Change the template deployment's `metadata.name` parameter in the template yaml file (Annex A.3) to `de-consumer-5` , and add a reference to the PVC created in the previous step.

4.4.7.3 Communication between Controller and Consumer Group

Using the computed difference between the next and current group's state, each consumer has a set of start and stop messages that have to be sent out by the controller, for the group to reach the intended state. Each partition can have associated to it at most two actions, which correspond to a start and/or a stop command.

Firstly, the controller prepares a batch of StartConsumingCommand messages, each directed to a single consumer. For each partition in the set of unassigned partitions, the partition ID is added to the StartConsumingCommand message that is intended to the consumer that was assigned the partition.

Another batch of StopConsumingCommand messages is prepared, and for each partition that has to be either rebalanced or removed, that partition's ID is added to the message which is intended to the consumer that is currently assigned to that partition.

Each message the controller sends out to a consumer, has to be acknowledged by the consumer with a corresponding event which can be one of two types, StartConsumingEvent, StopConsumingEvent. Each of these messages contain a set of partitions from which a consumer acted upon. For each partition contained within one of the events, the controller removes its corresponding actions from the set of actions that have to be deployed by the controller.

If the received event is of type StopConsumingEvent, a new batch of StartConsumingCommand records are prepared for the consumers that have to start consuming from the partitions that are being rebalanced. This means that for each partition referred in the StopConsumingEvent, if it has a corresponding start action, the partition is added to the StartConsumingCommand directed to the new consumer.

When the controller sends a message to a consumer, it sends it to the same partition as the consumer's ID in the `consumer.metadata` topic. As for the consumer, to communicate the event of having reacted to one of the controller's commands, the record has to be sent to the partition with ID 0 of the same topic, as depicted in Figure 4.1, Page 27.

This process terminates when there are no more actions to perform over any partition, since the controller removes a start or stop action from a partition as soon as it receives the acknowledgement by the consumer that it has performed the corresponding command.

4.4.8 State Synchronize

This state exists to mitigate desynchronization between the consumer group's state as perceived by the controller and the real state the group finds itself in. The procedure involves having the controller query the Kubernetes cluster to verify which are its active consumers, and with this information, it then queries each consumer through their respective partitions in the `consumer.metadata` topic for their current state. Only after all active consumers have responded with their state does the controller proceed to its normal behaviour triggering the transition to the Sentinel state.

Chapter 5

Integration Tests

This chapter aims to provide with an overview of the events the system goes through to automatically scale a group of consumers, so as to achieve the required parallelism to have the group consuming data from each partition at a speed which is bigger or equal to their respective write speed. This involves communication between all three components presented in Chapter 4, each playing a role in allowing the problem to be modeled as a Bin Packing Problem.

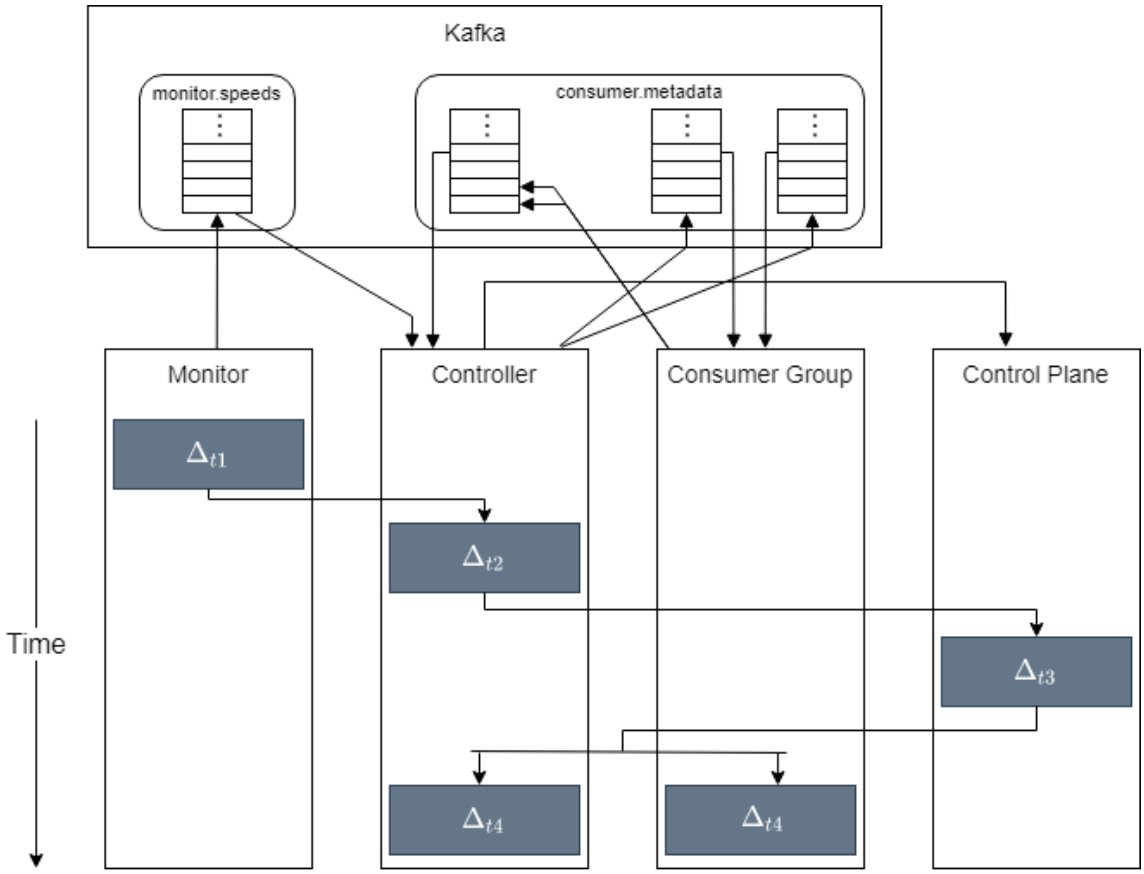


Figure 5.1: System sequence of events.

Initially, a measurement has to be provided by the monitor process to then be consulted by the controller (Δ_{t1}). The controller then updates the group's current state based on the new measurement, and proceeds to computing a new group configuration using one of the heuristic algorithms presented in Section 4.4.6. Having calculated the new configuration, the controller calculates the difference between the new and current configuration, to determine the consumers it has to create, the ones it has to delete, and the messages that have to be communicated to each consumer to reach the intended state (Section 4.4.7). This then leads to the controller communicating with the Kubernetes control plane, to create the new consumer resources (Δ_{t2}).

The controller then waits for the newly created deployments to be ready (Δ_{t3}), followed by communicating with the consumer group to inform the consumers of their change in state, which only terminates as soon as all messages have been sent out to the respective consumers, and when every message has been acknowledged back to the controller (Δ_{t4}). This process is illustrated by Figure 5.1.

5.1 Monitor Measurement Convergence Time (Δ_{t1})

To evaluate the monitor's response time, the data points were obtained by feeding the system a step input, similar to what was done in Section 4.2, which is obtained by starting a producer that sends data to one of the partitions the consumer group is consuming data from.

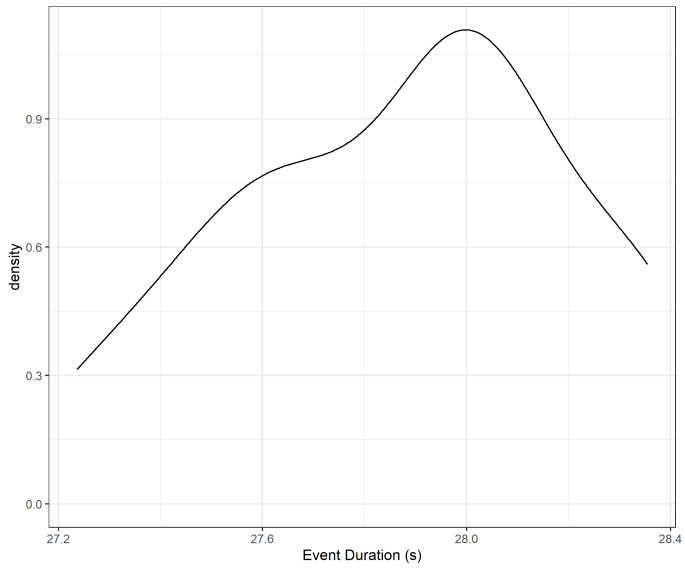


Figure 5.2: Distribution of Δ_{t1} for 31 different step inputs.

As such, this measure is the time it takes the monitor process to converge to the real production rate, which, as can be verified in Figure 5.2, takes no more than $30s$, similar to what had been obtained in Figure 4.2.

5.2 Time to Trigger Scale-up (Δ_{t2})

Measured as the time it takes the controller to compute the consumer group's new assignment, computing the difference between the new and current states, and to send an asynchronous request to the GKE cluster for every new consumer instance to be created. To obtain the distribution for this metric, the system was tested with the aforementioned step inputs and the randomly generated measurement sequences used in Section 4.4.6.1.

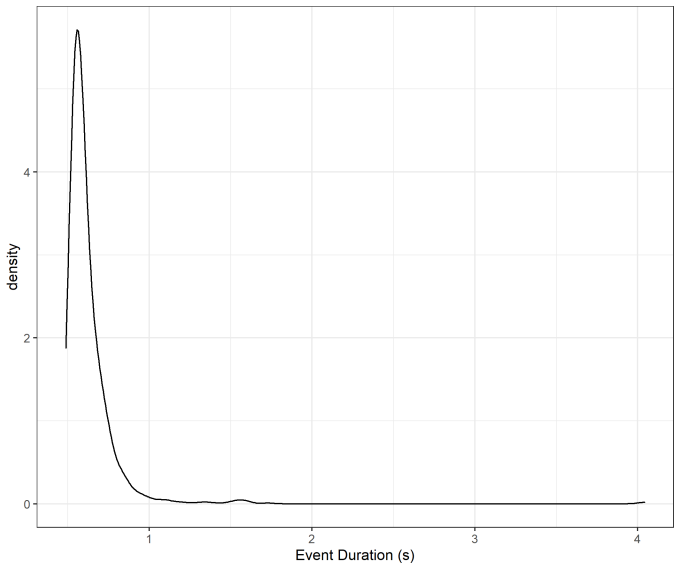


Figure 5.3: Distribution of Δ_{t2} for 1345 observations.

The time the controller takes with this procedure depends on the number of partitions to distribute between the consumer group, the algorithm the controller is executing to figure a new consumer group assignment, and the number of new consumer instances it has to create in the GKE cluster.

For the tested input data, where there were at most 32 partitions to rebalance and no more than 20 consumers to be created in a single interation, the event consistently takes less than 1 second to be executed as shown in Figure 5.3.

5.3 Newly Created Deployments Ready (Δ_{t3})

After making the asynchronous request to the GKE cluster, the control plane schedules the pods to a node, and within the node starts the containers that are to be executed.

After discarding the data points that didn't have the controller creating any new consumer, there remain 273 data points, which show two clusters of data points that are justified by two different scenarios when creating a new pod.

The first, and most frequent (88%), has the GKE cluster taking between $[10, 50]$ seconds to have a new consumer instance ready. This is usually the case when the controller requests for the creation of new consumer instances and the Kubernetes cluster has resources available to

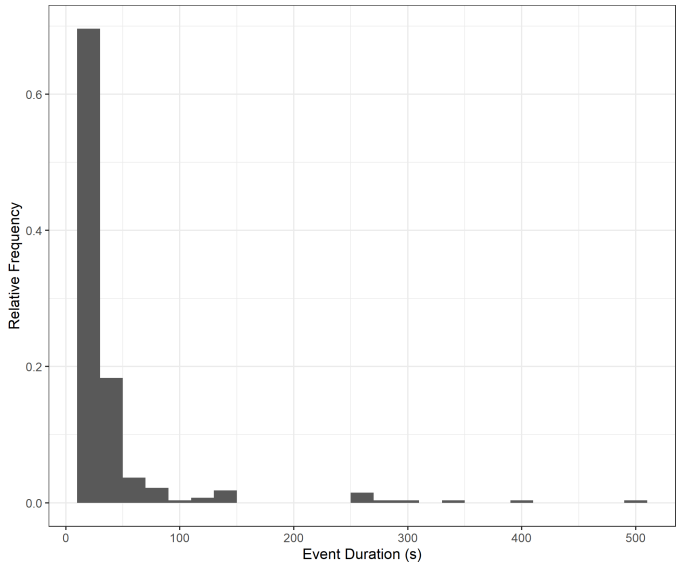


Figure 5.4: Histogram of Δ_{t3} for 273 observations.

schedule the new consumer instance. As such, the event's duration is related to the time it takes the scheduled node(s) to download the image from the container registry, and to start the containers.

The second group of data points, any time span greater than $50s$ (represents 12% of the data points), occurs when the Kubernetes cluster does not have any available resources. Here the actions the cluster undergoes are adding a new node to the cluster, and only then scheduling the consumer instance into the new available node. Due to the autoscaling feature of the GKE cluster, this is done automatically but it is also more inconsistent, having data points taking up to $500s$, although very sporadically.

Although there isn't much control over the time it takes the Kubernetes cluster to schedule and start the pods, one variable that can be controlled is the size of the image which has to be downloaded by the nodes that were assigned the newly created consumer instances.

5.4 Communicating Change in State (Δ_{t4})

Each partition that the controller is monitoring can either be associated with a start, stop, rebalance or a do nothing action. A start action refers to the case where there is no consumer currently consuming data from the partition, and the controller has assigned that partition to a consumer in its newly computed consumer group state. The stop action occurs when the controller no longer wants a consumer assigned to this partition therefore having no consumer fetching data from this partition in the group's future state. Rebalancing happens when the controller is changing the partition from being assigned from one consumer to another. Lastly, as the name suggests, do nothing is when there are no actions to be communicated for a partition.

Since there is no concurrent read from the partition when the controller is analyzing a start or a stop action, the controller can send this message as soon as it enters this event, having to wait at

most for one consumer cycle to receive the acknowledgement by the consumer.

When rebalancing, the controller has to first send out the stop command, wait for a response, send out the start command, and wait for the response. Without taking into account any processing and network delays, at worst, the controller might have to wait for 2 consumer cycles to be able to rebalance a partition.

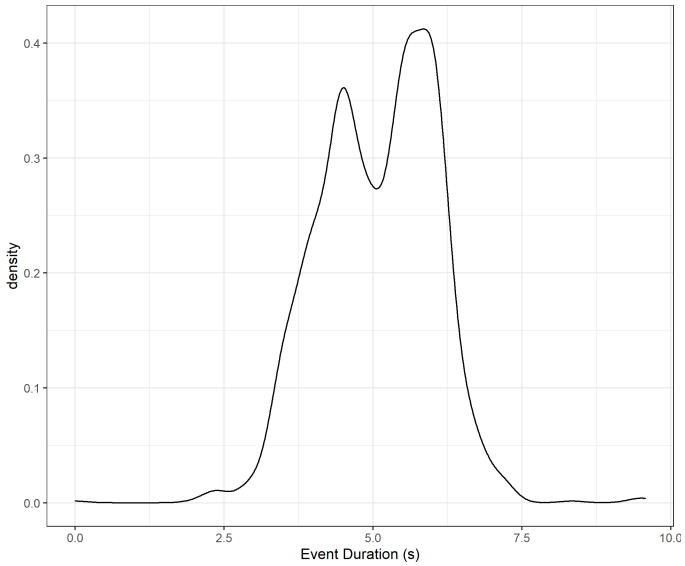


Figure 5.5: Distribution of Δ_{t4} for 1331 observations.

A consumer's insert cycle goes through the different phases mentioned in Section 4.3, and only after performing it's tasks does it verify the metadata queue to check if it has received any change in state command. For this reason, between two metadata reads from the `consumer.metadata` topic, it takes the consumer one whole insert cycle. Having defined in Section 4.3.5 that the consumer gathers at most $5Mbytes$ per cycle, and provided the results from Figure 4.5, which indicate the consumer has a data rate of approximately $2Mbytes/s$, each consumer cycle can take approximately $2.5s$. Since the controller has to wait for two consumer cycles, this would imply that changing the group's state could take around 5 seconds, as can be seen in Figure 5.5

It is also worth noting that the more consumers there are in the group, the higher the probability that the controller has to wait 1 whole cycle after sending out the stop command, and another whole cycle after a start command, as the communication is performed with more consumers.

Although this analysis was performed for a single partition, without any loss of generality and due to the fact that the controller sends out the change state commands in batch, the time it takes the controller to change a consumer group's state remains at 2 consumer cycles.

To vary the duration of this event, the consumer's cycle has to be altered using the `BATCH_BYTES` and `WAIT_TIME_SECS` parameters, which in turn have an effect on the time the consumer spends in a whole insert cycle.

Chapter 6

Conclusion and Future Work

6.1 Summary and Discussion of Thesis Results

The work developed in this thesis, aims to achieve a deterministic approach as for the number of consumers required working in parallel, so as to guarantee that the rate of production into a topic is not higher than the rate of consumption, while minimizing the operational cost.

The goal was to deterministically solve the autoscaling problem related to a group of consumers, which we model as a Bin Packing Problem where the items' sizes and bin assignments can change over time. In light of these variations, items' assignments can be rebalanced (bin assignment can change), and therefore we propose the Rscore to account for the rebalance cost, Section 4.4.1.

Given the fact that, to the best of our knowledge, it is the first time the BPP is applied in these conditions, there was lack of heuristic algorithms that solve the Bin Packing assignment while considering the Rscore. As such, in Section 4.4.6 we propose four new heuristic algorithms based on the Rscore, three of which are proven to be competitive solutions in Section 4.4.6.3, when solving the multi-objective optimization problem.

Furthermore, in Chapter 4, a system was developed to automatically scale a group of consumers and manage the partitions' assignments based on the aformentioned theoretical approach. This system was integrated in the company's infrastructure, and as expected is capable of providing a solution that scales a group of consumers based on the system's current load.

6.2 Future Work

Given the multiple parts developed within this system, additional improvements could be achieved. Most notably, due to the close relationship between the system and the Kubernetes cluster, the autoscaler could be wrapped as a Kubernetes Operator [22] as is common with systems of this kind ([28, 29, 21]).

6.2.1 Monitor

To the best of our knowledge, for lack of a better metric being provided by the Kafka cluster, the value that was used to track the write speed of each partition was the number of bytes in each partition. For this reason, a process (Section 4.2) was used to evaluate the speed of each partition by historically storing the amount of bytes in a partition at a certain timestamp.

A clear disadvantage of using this component is the fact that if `retention.ms != -1`, a record is deleted from its partition after `retention.ms` which leads to a reduction in the amount of bytes in the partition, consequently having the monitor process erroneously evaluating a negative partition write speed. To circumvent this behaviour, either one of the following metrics provided by Kafka would suffice: The current write speed (bytes/s) per partition; A historic amount of bytes that have been written to a partition. Since the monitor process presents the write speed as the average data rate over the last 30 seconds, using a different strategy to evaluate the speed could lead to more robust measures and improved convergence times.

6.2.2 Consumer

If the network is experiencing increased latency, the current implementation of the autoscaler does not account for changes in the size of the consumer capacity, therefore the controller assumes a maximum capacity of a consumer which is not accurate within this scenario. Metrics related to a consumer's performance provided by Kafka JMX metrics could be leveraged to obtain a more accurate dynamic representation of the consumer's current capacity. This could also lead to a variable bin size BPP.

6.2.3 Controller

Throughout this thesis, the controller process was presented with strict execution rules and actions to manage the Kafka consumer group. Evolving this component into a more abstract concept, could not only maintain its current functionalities, but it could also be used to more accurately solve load balancing within a consumer group without necessarily modeling the problem as a BPP.

In fact, the heuristic algorithms that make use of the worst fit rule when assigning partitions to consumers, are at the same time applying a load balancer rule between the available consumers.

6.3 Final Remarks

In spite of the fact that, as a whole, the thesis presents quite a specific use case, individually several of Kafka's functionalities are challenged and tested alternatives are also provided which can be used as a foundation so as to improve the way in which a consumer's load is modeled, and the manner in which the load (partitions) is distributed between the elements of a consumer group.

Appendix A

Appendix

```
{
    "name": "DEControllerSchema",
    "type": "array",
    "items": {
        "name": "TopicPartition",
        "type": "record",
        "fields": [
            {"name": "topic_name", "type": "string"},
            {"name": "bq_table", "type": "string"},
            {
                "name": "topic_class",
                "type": {
                    "name": "topic_class",
                    "type": "record",
                    "fields": [
                        {"name": "module_path", "type": "string"},
                        {"name": "class_name", "type": "string"},
                    ]
                }
            },
            {
                "name": "partitions",
                "type": {
                    "type": "array",
                    "items": {
                        "name": "partition",
                        "type": "int"
                    }
                }
            },
            {
                "name": "ignore_events",
                "type": {
                    "type": "array",
                    "items": {
```

```
                        "name": "event_type",
                        "type": "string"
                    }
                }
            },
        ]
    }
}
```

Listing A.1: Avro Schema for records sent to the data-engineering-controller topic

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: de-consumer-1-volume
  namespace: data-engineering-dev
  labels:
    consumerGroup: de-consumer-group
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: de-consumer-volume
  resources:
    requests:
      storage: 500Mi
```

Listing A.2: Template Persistent Volume claim

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: de-consumer-1
  namespace: data-engineering-dev
  labels:
    consumerType: consumer-autoscaler
    consumerGroup: de-consumer-group
spec:
  replicas: 1
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: de-consumer
  template:
    metadata:
      labels:
        app: de-consumer
    spec:
      containers:
        - name: consumer-capacity
```

```
        image: ...
        imagePullPolicy: Always
        resources:
          requests:
            memory: "0.4Gi"
            cpu: "500m"
          limits:
            memory: "0.6Gi"
            cpu: "500m"
        env:
          - name: CONSUME_ENV
            value: "uat"
          - name: WRITE_ENV
            value: "uat"
          - name: GROUP_ID
            value: "data-engineering-autoscaler"
          - name: BATCH_BYTES
            value: "5000000"
          - name: WAIT_TIME_SECS
            value: "1"
        volumeMounts:
          - name: podinfo
            mountPath: /etc/podinfo
          - name: podpvc
            mountPath: /usr/src/data
    volumes:
      - name: podinfo
        downwardAPI:
          items:
            - path: "pod_name"
              fieldRef:
                fieldPath: metadata.name
      - name: podpvc
        persistentVolumeClaim:
          claimName: de-consumer-1-volume
```

Listing A.3: Template consumer deployment

Bibliography

[1] *Autoscaling Algorithm*. https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/horizontal-pod-autoscaler.md.

[2] Nurşen Aydın, İbrahim Muter, and Ş İlker Birbil. "Multi-objective temporal bin packing problem: An application in cloud computing". In: *Computers & Operations Research* 121 (2020), p. 104959.

[3] Brenda S Baker and Edward G Coffman Jr. "A tight asymptotic bound for next-fit-decreasing bin-packing". In: *SIAM Journal on Algebraic Discrete Methods* 2.2 (1981), pp. 147–152.

[4] Mauro Maria Baldi et al. "A generalized bin packing problem for parcel delivery in last-mile logistics". In: *European Journal of Operational Research* 274.3 (2019), pp. 990–999.

[5] Edward G Coffman et al. "Bin packing approximation algorithms: survey and classification". In: *Handbook of combinatorial optimization*. 2013, pp. 455–531.

[6] János Csirik and David S Johnson. "Bounded space on-line bin packing: Best is better than first". In: *Algorithmica* 31.2 (2001), pp. 115–138.

[7] Milan De Cauwer, Deepak Mehta, and Barry O'Sullivan. "The temporal bin packing problem: an application to workload management in data centres". In: *2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2016, pp. 157–164.

[8] Maxence Delorme, Manuel Iori, and Silvano Martello. "Bin packing and cutting stock problems: Mathematical models and exact algorithms". In: *European Journal of Operational Research* 255.1 (2016), pp. 1–20.

[9] *Elastic Kubernetes Service Fargate*. https://aws.amazon.com/fargate/pricing/.

[10] Martin Fowler. *Microservices*. https://martinfowler.com/articles/microservices.html. Mar. 2014.

[11] Fabio Furini and Xueying Shen. "Matheuristics for the temporal bin packing problem". In: *Recent Developments in Metaheuristics*. Springer, 2018, pp. 333–345.

[12] *Google Kubernetes Engine Autopilot overview*. https://cloud.google.com/kubernetes-engine/docs/concepts/autopilot-overview.

[13] David S Johnson. "Fast algorithms for bin packing". In: *Journal of Computer and System Sciences* 8.3 (1974), pp. 272–314.

[14] David S Johnson. “Near-optimal bin packing algorithms”. PhD thesis. Massachusetts Institute of Technology, 1973.

[15] David S Johnson and Michael R Garey. “A 7160 theorem for bin packing”. In: *Journal of complexity* 1.1 (1985), pp. 65–106.

[16] David S. Johnson et al. “Worst-case performance bounds for simple one-dimensional packing algorithms”. In: *SIAM Journal on computing* 3.4 (1974), pp. 299–325.

[17] EG Co man Jr, MR Garey, and DS Johnson. “Approximation algorithms for bin packing: A survey”. In: *Approximation algorithms for NP-hard problems* (1996), pp. 46–93.

[18] *Kafka Consumer*. https://docs.confluent.io/5.5.2/clients/consumer.html.

[19] *Kafka Consumers: Reading Data from Kafka*. https://www.oreilly.com/library/view/kafka-the-definitive/9781491936153/ch04.html.

[20] *Kafka Producer*. https://docs.confluent.io/5.5.2/clients/producer.html.

[21] *Kafka scaler provided by Kubernetes Event Driven Autoscaling*. https://keda.sh/docs/1.4/scalers/apache-kafka/.

[22] *Kubernetes Operator Pattern*. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/.

[23] Chan C Lee and Der-Tsai Lee. “A simple on-line bin-packing algorithm”. In: *Journal of the ACM (JACM)* 32.3 (1985), pp. 562–572.

[24] W Mao. “Besk-k-Fit bin packing”. In: *Computing* 50.3 (1993), pp. 265–270.

[25] Weizhen Mao. “Tight worst-case performance bounds for next-k-fit bin packing”. In: *SIAM Journal on Computing* 22.1 (1993), pp. 46–56.

[26] Irakli Nadareishvili et al. *Microservice architecture: aligning principles, practices, and culture*. " O’Reilly Media, Inc.", 2016.

[27] Sam Newman. *Building microservices: designing fine-grained systems*. " O’Reilly Media, Inc.", 2015.

[28] *Postgres Kubernetes Operator*. https://www.kubegres.io/.

[29] *Pulumi Kubernetes Operator*. https://github.com/pulumi/pulumi-kubernetes-operator.

[30] T Sharvari and K Sowmya Nag. “A study on Modern Messaging Systems-Kafka, RabbitMQ and NATS Streaming”. In: *CoRR abs/1912.03715* (2019).

[31] Randall Smith. *Docker Orchestration*. Packt Publishing Ltd, 2017.

[32] *The Kubernetes API*. https://kubernetes.io/docs/concepts/overview/kubernetes-api/.

[33] *Using Minikube to Create a Cluster*. https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/.

[34] Gerhard Wäscher, Heike Haußner, and Holger Schumann. "An improved typology of cutting and packing problems". In: *European journal of operational research* 183.3 (2007), pp. 1109–1130.

[35] *What is a Container?* https://www.docker.com/resources/what-container.

[36] *Work Queues*. https://www.rabbitmq.com/tutorials/tutorial-two-python.html.

[37] G Zhang. *Tight worst-case performance bound for AFBk*. Tech. rep. Technical Report 015, Institute of Applied Mathematics, Academia Sinica ..., 1994.