DIPLOMA THESIS

FlinkMotionInsights:

Motion and object detection from streaming video on Apache Flink

---

AUTHOR: DIMITRIOS BANELAS

THESIS COMMITTEE: PROF. EURIPIDES G.M. PETRAKIS (SUPERVISOR)
PROF. MICHAEL ZERVAKIS
ASSISTANT PROF. NIKOLAOS GIATRAKOS

SCHOOL OF ELECTRICAL & COMPUTER ENGINEERING

TECHNICAL UNIVERSITY OF CRETE

Abstract

In this thesis, we present distributed video processing system, built with Apache Flink and Apache Kafka, and deployed on the cloud using the Kubernetes orchestration platform. The system facilitates multiple object tracking and motion pattern extraction from multiple streaming or batch video sources. The goal of implementing this system is to observe whether such a system is feasible on Apache Flink. Additionally, we aim to explore the performance, both in terms of theoretical expectations and practical benchmarks. Our focus will be on assessing the accuracy of the results and evaluating the speed and scalability of the system. Leveraging the distributed stream processing capabilities of Apache Flink and Apache Kafka as an intermediate message broker, this system utilizes a combination of computer vision algorithms, including background subtraction, connected component labeling, and centroid-based tracking. These algorithms are implemented using the dataflow model that Flink provides. Hence, each algorithm is implemented within a Flink operator. In order to achieve the full utilization of the distributed architecture, each frame is split into smaller blocks, which are distributed and processed by a number of different Flink operators. Initially, the blocks are evenly distributed to multiple partitions of an input topic in Kafka. Then equal number of Flink pipelines consume the blocks in parallel. In the first stage, each block undergoes background subtraction and component labeling. This process results in the bounding box and centroid of each object present in the blocks. Then, all the connected components from each frame are grouped, and the eligible components are merged into objects. In the last stage of the pipeline, all objects from each frame are concentrated in an operator responsible for creating the trajectory of each object present in the video. When every frame of the video has been processed, the extracted trajectories are announced to another Kafka topic. The application is deployed as a Flink cluster on top of a Kubernetes one, using the newly established Flink Kubernetes Operator. Experimenting in a 7 machine environment (1 CPU from each machine is allocated to Flink), we observed that the system both scales and outperforms a single machine monolith implementation, while providing accurate motion patterns for each video. In fact, the Flink application, configured with a parallelism setting of 7, attained a speedup of up to 6x in comparison to a single-node monolithic implementation, and up to 4.9x compared to the Flink application with a parallelism setting of 1.

Abstract

Στην παρούσα διπλωματική εργασία, παρουσιάζουμε ένα κατανεμημένο σύστημα για επεξεργασία βίντεο, με τη χρήση του Apache Flink και του Apache Kafka, εγκατεστημένο σε cloud υποδομή, με τη βοήθεια του Kubernetes ως πλατφόρμα ενορχήστρωσης. Το παρόν σύστημα πραγματοποιεί παρακολούθηση πολλαπλών αντικειμένων, καθώς και τη δημιουργία και εξαγωγή των μοτίβων κίνησης τους σε μορφή τροχιών, από πολλαπλές ροές βίντεο. Οι στόχοι αυτής της υλοποίησης είναι να συμπεράνουμε εάν ένα τέτοιο σύστημα είναι εφικτό με τη χρήση του Apache Flink. Ακόμη, στοχεύουμε να εξερευνήσουμε την απόδοση, τόσο σύμφωνα με τις θεωρητικές προσδοκίες όσο και με πρακτικά benchmarks. Η ακρίβεια των αποτελεσμάτων, η ταχύτητα και η δυνατότητα κλιμάκωσης του συστήματος, είναι σημαντικά αποτελέσματα αυτής της εργασίας. Αξιοποιώντας τις δυνατότητες κατανεμημένης επεξεργασίας ροών του Apache Flink, και το Apache Kafka ως μεσάζοντα μηνυμάτων, το σύστημα χρησιμοποιεί ένα συνδυασμό αλγορίθμων μηχανικής όρασης, συμπεριλαμβανομένης της αφαίρεσης φόντου, της αναγνώρισης συνδεδεμένων στοιχείων και της παρακολούθησης με βάση το κέντρο του αντικειμένου. Οι αλγόριθμοι αυτοί υλοποιούνται με βάση το datalflow μοντέλο το οποίο υπαγορεύει το Apache Flink. Αυτό σημαίνει ότι κάθε αλγόριθμός υπάρχει σε έναν τελεστή του Flink, ο οποίος μπορεί να κλιμακωθεί ανεξάρτητα, ανάλογα με το φόρτο του. Με στόχο να αξιοποιηθεί πλήρως η κατανεμημένη αρχιτεκτονική που προτείνουμε, κάθε καρέ χωρίζεται σε μικρότερα κομμάτια, τα οποία μπορούν να διαμοιραστούν και να επεξεργαστούν στους διάφορους τελεστές του Flink. Αρχικά, όλα τα κομμάτια ενός βίντεο διανέμονται ομοιόμορφα στα partitions του Kafka. Στη συνέχεια, όλοι οι κόμβοι του Flink διαβάζουν παράλληλα αυτά τα κομμάτια. Στο πρώτο στάδιο της επεξεργασίας, κάθε κομμάτι υφίσταται αφαίρεση φόντου και αναγνώριση συνδεδεμένων στοιχείων. Αυτή η διαδικασία παράγει το κέντρο και το ορθογώνιο οριοθέτησης του κάθε συνδεδεμένου στοιχείου που υπάρχει σε κάθε κομμάτι. Σε δεύτερο στάδιο, τα συνδεδεμένα στοιχεία από κάθε καρέ συγκεντρώνονται σε έναν τελεστή, ο οποίος συγχωνεύει αυτά τα οποία πληρούν κάποιες προϋποθέσεις συγχώνευσης. Η διαδικασία αυτή παράγει τα αντικείμενα του κάθε καρέ. Τέλος, τα αντικείμενα από όλα τα καρέ συγκεντρώνονται στον τελευταίο τελεστή, ο οποίος σχηματίζει τις τροχιές του κάθε αντικειμένου ανάλογα με τη θέση του κέντρο τους. Όταν όλα τα καρέ του βίντεο έχουν επεξεργαστεί, οι τελικές τροχιές ανακοινώνονται σε άλλο ένα Kafka topic. Η εφαρμογή εγκαταστάθηκε ως ένα Flink cluster πάνω από ένα Kubernetes cluster με τη βοήθεια του Flink Kubernetes Operator. Πραγματοποιώντας πειράματα σε ένα περιβάλλον με 7 μηχανές, παρατηρήσαμε ότι το προτεινόμενο σύστημα και έχει τη δυνατότητα κλιμάκωσης, και ξεπερνά σε απόδοση ένα μονολιθικό σύστημα, ενώ παράγει ακριβείς τροχιές για τα αντικείμενα των βίντεο. Πράγματι, η εφαρμογή στο Flink με παραλληλισμό 7 είναι έως και 6 φορές καλύτερη σε απόδοση σε σχέση με ένα μονολιθικό σύστημα και εώς 4.9 φορές πιο αποδοτική από την εφαρμογή στο Flink με παραλληλισμό 1.

Acknowledgements

It has been both a challenging and rewarding five years studying at this University. First and foremost, I would like to express my heartfelt gratitude to my thesis supervisor, Euripides Petrakis. His constant, support, guidance and ideas have been invaluable during the completion of this thesis. I would also like to thank my friends, and especially Konstantinos Ktistakis and Theodoros Kalamarakis, for helping me learn how to study and get through the numerous courses of each semester, and Ourania Ntouni, for being my study partner during the period of my thesis. Last but not least, I would like to extend my gratitude to my family -especially my brother, Panagiotis Banelas- and Alexandra Vasileiou, for their continuous support.

Contents

1 Introduction **6**
1.1 Problem definition 6
1.2 Proposed solution 7

2 Related work **9**

3 Background **12**
3.1 Apache Flink 12
3.2 Apache Kafka 19
3.3 Kubernetes 21
3.4 Flink Kubernetes Operator 23
3.5 Google Cloud Platform 24
3.6 OpenCV 24
3.7 Background Subtraction - Foreground detection 24
3.8 Connected Component Labeling 28

4 System architecture **29**
4.1 Client 30
4.1.1 Workflow 30
4.1.2 Block size 31
4.1.3 Key generation 32
4.1.4 Acknowledgments of receipt 33
4.2 Kafka broker 34
4.3 Flink pipeline design 34
4.3.1 Operators 36

5 Trajectory extraction results **46**
5.1 Methods 46
5.2 Videos 46
5.3 Impact of block size 47
5.4 Results 49
5.5 Conclusions 53

6 Performance evaluation **54**
6.1 Infrastructure 54
6.2 **Experiment 1**: Comparison to a monolith system 57
6.3 **Experiment 2**: Scaling and maximum throughput evaluation 61
6.4 **Experiment 3**: Identifying the bottlenecks of the system 68

7 Conclusion **71**

8 Future work **72**

References **74**

1 Introduction

The era of digital media, along with the proliferation of smartphones, surveillance cameras, streaming services and other sources of video, has led to an unprecedented surge in video data. On a daily basis, several exabytes of video data are produced, emphasizing the importance of harnessing and understanding the information contained in this kind of data. As videos continue to shape the landscape of the digital world, video processing and analysis is emerging as a crucial domain, providing a wide range of techniques intended to extract, analyze, and interpret valuable insights from video data.

Video processing represents a multidisciplinary field challenge that spans the domains of computer science, signal processing and multimedia technology among others. This term usually refers to low level operations where input video data is transformed into a more useful output format. Examples of such types of processing include compression/decompression, video segmentation and enhancement. Such operations have low semantic meaning, since there is limited or no understanding of the content and context. Contrary to video processing, video analysis delves into the challenge of interpreting video content in a higher level. In the context of video information, the distinction between intermediate and high semantic meaning hinges upon the depth and complexity of the conveyed insights. Intermediate semantic meaning encompasses tasks like object detection - tracking and trajectories within a video, while high semantic meaning shifts the focus to a more comprehensive understanding of video content. This involves activity recognition and scene understanding which require the recognition of broader contextual details and extraction of implicit information.

By using computer vision algorithms, machine learning and artificial intelligence, video analysis aims to extract meaningful insights from videos, determining not only the objects and actions present but also their relationships, emotions, and intentions. Through advanced methodologies like object tracking, activity recognition and sentiment analysis, video analysis transforms raw data into decision-making information.

1.1 Problem definition

Object tracking and the analysis of motion patterns are tasks of significance in the field of computer vision due to their wide range of applications, including surveillance, autonomous driving and traffic optimization. The fundamental challenge addressed in this thesis revolves around the accurate and efficient detection and tracking of objects in static camera scenes and the extraction of their trajectories. While algorithms and systems that already tackle this challenge exist, each have their set of limitations and drawbacks when low latency and/or high throughput processing becomes imperative. State-of-the-art systems like YOLO [1], while showcasing remarkable accuracy in detecting and recognizing objects in images, encounter significant obstacles when faced with videos of higher resolutions or frame rates. In order to keep up with the needed performance, such systems heavily rely on Graphics Processing Units (GPUs) for their processing tasks. However, while GPUs provide accelerated computing capabilities, they come with a significant financial burden. The acquisition and maintenance costs associated with this specialized hardware pose a

notable issue to the scalability of such systems. Furthermore, many traditional algorithms used in object or motion detection and tracking such as background subtraction or connected component labeling, have parallel implementations that rely on multithreading or CUDA architectures [2], [3], making them too, unsuitable for scaling and high throughput processing. Last but not least, while there are parallel implementations of libraries like [4], which provide a transparent interface for parallel computer vision programming, they rely on Grid computing infrastructure which is not widely accessible.

1.2 Proposed solution

In this study, we present FlinkMotionInsights, a CPU based system to facilitate multiple object tracking and trajectory extraction from multiple video sources. The system requires videos from static cameras and can support both streams and batches of videos. This is accomplished through the utilization of Apache Flink's[5] distributed stream processing capabilities and Apache Kafka as the intermediate message broker. The detection algorithm is built using a variety of computer vision algorithms such as background subtraction, connected component labeling and centroid based tracking. Said algorithms were chosen due to their ability to be parallelized efficiently. Each video is first split into frames and then each frame is split into small, fixed sized blocks, that enable rapid transfer and distribution in the partitions of the Kafka broker. A client program reads frames of a given video from the disk or an online camera, immediately creates records with said blocks and uniquely generated keys, and streams them to the broker.

On the processing side, a number of Apache Flink pipelines equal to Kafka partitions, consume the records and submit them through successive processing stages facilitated by Flink's DataStream API operators. The first operator is a Kafka consumer which consumes records when they are available, and then forwards them to the downstream operators. The Kafka consumer is responsible for transforming the records into Java objects that are eligible for processing. The next operator performs background subtraction, generating a binary mask (black and white image) for each input block. In this image, the white pixels represent the moving objects and the black pixels represent the background. Between the consumer and the background subtraction, the stream is keyed and redistributed in order for the background subtraction algorithm to function correctly. Details and further elaboration will be provided in the System architecture section. A mechanism to ensure that these blocks are processed in the correct order has been established. Following the background subtraction, each binary image block undergoes connected component labeling. This procedure yields the centroids and bounding boxes of every connected component within the block. Subsequently, the stream is structured to group all connected components from each frame together for collective processing. These grouped components are subjected to a connected component merging algorithm where the objects of each frame are detected (i.e. the centroid and bounding box of each object is determined). Concluding the pipeline, the last operator aggregates all frame objects from a single video and outputs the trajectory of each object as a string of centroid coordinates to an output Kafka topic. This is achieved by sorting the frames and implementing a centroid-based tracking algorithm[6].

Flink's extensive keying mechanisms along with correctly generated keys, allow for multiple videos to be processed simultaneously and their results to be retrieved independently. To achieve scalability, the number of input topic partitions should increase, aligning with the number of Flink pipelines. With deployment on the Kubernetes platform, scaling becomes as straightforward as modifying a configuration file. Kubernetes offers this exceptional ease of scalability because it automates many of the complex tasks associated with managing containerized applications. Experimenting with the proposed system, we observed a speedup up to almost 6x when compared to a sequential - monolith implementation of the same algorithm, and 4.9x when compared to the *naive approach*, which is the Flink application with the parallelism set to 1. At last, the system can process in real-time inputs up to $226MB/s$. This work's main contributions are:

- A highly scalable, distributed video analytics system that can track multiple objects and extract their trajectories on video streams.
- An easily reproducible system that leverages Kubernetes (K8s) for deployment. This system is versatile, capable of being deployed on various environments, including cloud infrastructure and custom hardware platforms, as all its components are compatible with Kubernetes.
- A system that achieves great speedup when compared to its sequential implementation using only CPUs. This aspect of the system results in great cost saving in comparison to systems that utilize infrastructures based on GPUs.

2 Related work

You Only Look Once (YOLO) [1] is an advanced object detection and recognition framework that has gained significant attention in the field of computer vision and deep learning. YOLO stands out for its performance and accuracy, offering a streamlined approach to object detection by simultaneously predicting object bounding boxes and class probabilities within a single forward pass of a convolutional neural network. Unlike traditional sliding window techniques, YOLO divides an image into a grid and predicts bounding boxes directly associated with specific grid cells, yielding rapid and efficient detection of multiple objects in various scales and orientations. This unique architecture enables YOLO to maintain a high level of accuracy, even in challenging scenarios. YOLO's performance heavily depends on the underlying hardware (GPU power), as well as the resolution of the input videos, making it ineligible for applications that require processing of high resolution videos but are not in possession of capable hardware. YOLO's versatility, speed, and precision have positioned it as a cornerstone technology for a wide range of applications, including autonomous driving, video surveillance, and robotics.

While YOLO focuses on detecting and recognizing objects in images, our system provides a standalone, complete solution for trajectory extraction on real-time video streams. FlinkMotionInsights can process over 220 MB/s in real-time which is the equivalent of $\approx$ 185 fps in $640x640$ frames (size on which YOLO models are evaluated), while the total cost of the system amounts to just \$1.80 per hour of usage. According to [7], when running on an Nvidia Tesla V100 GPU (AI-grade GPU), which is currently priced at \$8000, the state-of-the-art YOLO models (YOLOv7) can achieve almost 60 fps. The only models that surpass our 185 fps mark are the significant smaller ones (i.e YOLOv7-tiny), which can achieve up to 220 fps.

Antonakakis et al. [8] use YOLOv5 in order to perform object detection in ultra-high resolution images, using GPU embedded systems. In order to achieve acceptable results, they split each 8K into 6 subframes and delegate the processing of each subframe to a different grid of the GPU system. There, 6 instances of the YOLOv5 models are running, performing detection and recognition in parallel. At last, merging of all the intermediate results is performed, in order to produce the final detections for each frame. This method achieves processing times < 1 second for each frame, which is considered to be real-time given the capturing framerate of high-resolution cameras.

Regarding traditional algorithms like background subtraction and connected component labeling, which are key components of non deep-learning approaches to object tracking, their parallel implementations are tied to multithreading [2], [9], [10], [3]. It's essential to emphasize that these algorithms are not intended to operate as independent solutions for object tracking, detection, or recognition. Instead, they are designed to complement each other and work in tandem with additional algorithms. While effective on smaller datasets, said implementations either struggle when faced with maintaining their performance when applied to larger datasets or require expensive and inaccessible infrastructure such as CUDA enabled GPUs, supercomputers or computer grids. Our work focuses on parallelizing the processing of the blocks that comprise each frame, making the architecture independent of the underlying hardware.

Huang et al. [11], present the transition from a monolith encoding system to SVE, a distributed video processing engine. SVE parallelizes the uploading, storing and processing the videos by breaking the video into segments consisting of a group of pictures (i.e. smaller videos), since GOPs[1] can be encoded and decoded independently. It offers services such as thumbnail creation, video encoding for different resolutions and speech recognition among others. Furthermore, users can create application-specific executions graphs in order to implement other computer vision tasks. In the context of SVE, an execution graph refers to the logical ordering of various computational tasks and processes that constitute a complete video processing application. SVE provides fault tolerance and latency speedup up to 9x compared to the monolith system for large videos (i.e. larger than 1GB). A significant drawback of SVE as per the authors, is that it does not support the processing of real time video streams.

Uddin et al. [12], the authors present SIAT, a cloud based distributed video analytics framework. With the integration of JavaCV [2] in conjunction with Apache Spark, SIAT can process both streams and batches of video data, and provides a wide range of video analytics services including key frame extraction, background subtraction, video compression, face recognition etc. In contrast to SIAT, this study is based on breaking each frame into blocks and then process them in parallel instead of processing the whole frame. Processing the entire frame enables the use of deep learning algorithms for more advanced services.

Kim et al. [13] present a platform for distributed processing of massive video streams. RIDE provides a multilayered architecture which supports both coarse-grained and fine-grained parallelism minimizing communication between tasks on different nodes. Coarse-grained parallelism is achieved by partitioning the image stream and assigning each partition (group of frames) to a different worker while fine-grained parallelism is achieved by utilizing the GPU and/or multicore processing on each worker. The system presented by this thesis also supports both coarse and fine-grained parallelism but in a different fashion. The first operators of the Flink pipeline perform background subtraction and connected component labeling on the segments of each frame, whereas in the later stages of the pipeline the components of each frame are grouped and processed in a per frame and per video manner. In contrast to our system, which is deployed on Google Cloud Platform using Kubernetes, RIDE relies on custom implementations on both software and hardware. As a result, the system cannot be replicated or deployed with the ease that MotionFlink-Insights provides. The authors report a maximum speed up of 5x on six machines in the cloud.

Seinstra et al. [4], present Parallel-Horus, a parallel extension to the sequential multimedia library Horus [14], for a Grid execution environment. This library enables engineers to create multimedia applications as sequential programs, while fully taking advantage of Grid Computing. Parallelization is achieved by processing individual frames on different machines. Using Grid Computing shows significant potential for video processing applications when programming is made more accessible and transparent. The authors report up to 45x speedup in a grid with 64 machines, when running a feature extraction appli-

[1] https://en.wikipedia.org/wiki/Group_of_pictures

[2] A Java wrapper for commonly used computer vision libraries like OpenCV and FFmpeg

cation. Compared to Parallel-Horus, our system achieves parallelization first by splitting each frame in blocks and processing them in a parallel manner, and then doing the same in a per frame manner. Furthermore, the functionality of the two systems is fundamentally different. FlinkMotionInsights is a system targeted to address the problems of object tracking and trajectory extraction, while Parallel-Horus provides a library of algorithms implemented for Grid Computing infrastructures. At last, Parallel-Horus does not support the processing of streaming videos, unlike the system that this work presents, which is tailored to streaming video processing.

Kastrinakis & Petrakis [15] present a system that facilitates real-time shot detection, using Apache Flink. In this work, the frame is split into smaller, arbitrary sized blocks, which are processed in parallel by the Flink pipelines. They report speedup of 7x in an 8-node environment. In contrast, our work focuses on a more computationally intensive challenge: multiple object tracking and trajectory extraction. Despite the shared initial step of frame segmentation into blocks, our study diverges significantly in its objectives and computational demands.

Table 1 summarizes all the above systems and their comparison with FlinkMotionInsights. It is essential to acknowledge that these systems are designed to serve different functionalities and objectives. The main comparison points are the ways that each system achieves parallel processing, the underlying infrastructure, as well as the ability for replication and ease of deployment. FlinkMotionInsights along Video2Flink are the only systems that achieve high speedup on *streams* of videos and do not require dedicated infrastructures. Systems such as Parallel-Horus and SVE require the videos to be stored in the disk in order to process them and do not support video streams. Furthermore, our system can be easily replicated and deployed on any infrastructure, custom or not, as long as it supports the use of Kubernetes.

System	Application	Processing unit	Hardware platform	Software platform	Type of processing	Maximum speedup
SVE	Video enconding, thumbnail generation etc.	Group of Pictures	Cloud	Custom	Batch	**9:1 (N/A)**
SIAT	Video analytics	Image frames	Cloud	Kafka, Spark, Hadoop	Real-time	**3.5:1 (N/A)**
RIDE	Video analytics	Image frames	Cloud	Kafka, Hadoop	Real-time	**5:1 (6)**
Parallel Horus	Video processing library	Image frames	Grid	Custom	Batch	**45:1 (64)**
Antonakakis et al.	Object detection and recognition using YOLOv5	Image block - subframe	Embedded GPU system	-	Real-time captured images	**1 FPS at 8K**
Video2Flink	Shot detection	Image blocks	Cloud (K8s)	Kafka, Flink	Batch, real-time	**7:1 (8)**
FlinkMotionInsights	Object tracking, motion patterns	Image blocks	Cloud (K8s)	Kafka, Flink	Batch, real-time	**4.9:1 (7)**

Table 1: Video processing systems comparison

3 Background

The following section provides an overview of the key tools and technologies that were used in this thesis. It is important to note that much of the information and images presented in this chapter have been sourced from the official documentation of these tools. The source of each documentation website is given below:

1. **Apache Flink:** https://nightlies.apache.org/flink/flink-docs-stable/
2. **Apache Kafka:** https://kafka.apache.org/documentation/
3. **Kubernetes:** https://kubernetes.io/docs/home/
4. **Flink Kubernetes Operator:** https://nightlies.apache.org/flink/flink-kubernetes-operator-docs-stable/
5. **Google Cloud Platform:** https://cloud.google.com/
6. **OpenCV:** https://opencv.org/

3.1 Apache Flink

Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. Flink has been designed to run in all common cluster environments, perform computations at in-memory speed and at any scale.

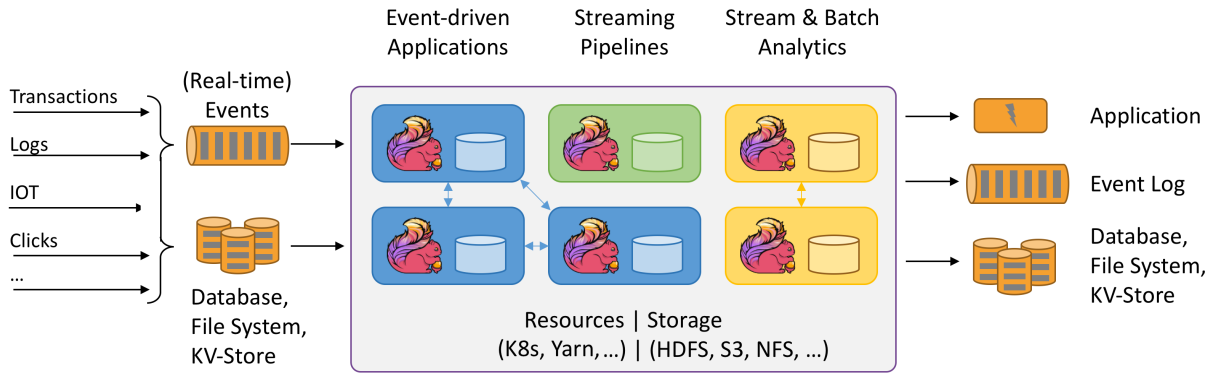


Figure 1: Apache Flink overview

Stream Processing

Apache Flink can process any kind of data that is produced as a stream. This includes credit card transactions, clicks, sensor measurements or logs. Data streams can be categorized into two categories: bounded and unbounded streams.

Unbounded streams have a defined start but no end. Continuously generated data must be processed immediately after digesting, since it is not possible to wait for all input data to be ingested due to the infinite nature of the stream. To effectively analyze unbounded data, it is often necessary to ingest events in a particular sequence, typically mirroring their chronological order, to ensure a clear understanding and result completeness.

Bounded streams have both a defined start and end. They can be processed by ingesting all data first. Ordering of events with use of time is not necessary since data can always be sorted. The processing of unbounded streams is also referred to as batch processing.

Flink's internal data structures and algorithms as well as the precise control over time and state, result in it excelling at processing both bounded and unbounded data sets.

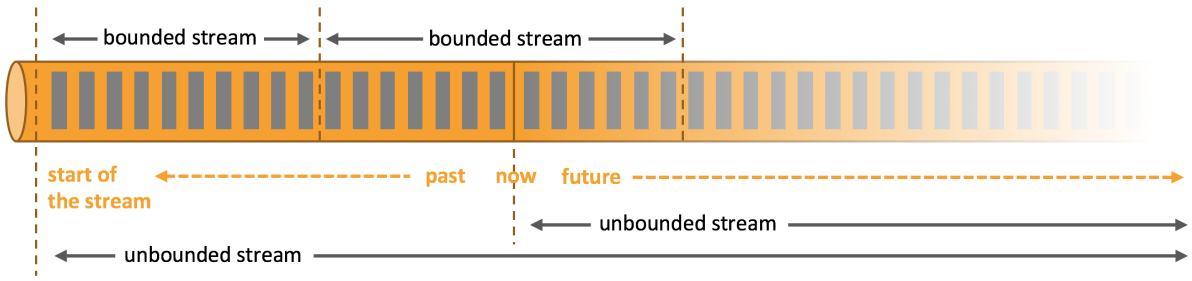


Figure 2: Representation of bounded and unbounded streams

Deployment options

Apache Flink offers various deployment options to cater to different infrastructure and scaling needs. One of the popular deployment options is Kubernetes (K8s), which allows users to orchestrate and manage Flink clusters efficiently within containerized environments. The K8s deployment can either be native (i.e. direct integration with Flink) or with the use of Flink Kubernetes Operator. The latter will be discussed in detail in the respective subsection.

Additionally, Flink can be deployed on Apache Hadoop YARN (Yet Another Resource Negotiator), making it suitable for Hadoop-based environments. YARN integration enables resource management and multi-tenancy capabilities, making it easier to coexist with other Hadoop ecosystem components.

Furthermore, Flink supports standalone deployments, where you can manually configure and manage clusters for more straightforward setups or testing purposes. This option provides full control over Flink cluster configuration.

In all cases Flink provides a RESTful API along with a Web UI in order to directly interact with Flink applications.

Flink applications

A Flink application is referred to as a job. The job represents a series of tasks or operators designed to process and transform the incoming records. These tasks can perform various data transformations, aggregations, and computations, defined by the user, to manipulate and analyze data. At its core a Flink job comprises the following key components:

- Sources and sinks
- Data Transformations
- Network operations - Keying and rebalancing mechanisms
- Windowing mechanisms

The following image describes how Flink code is represented with the above components.

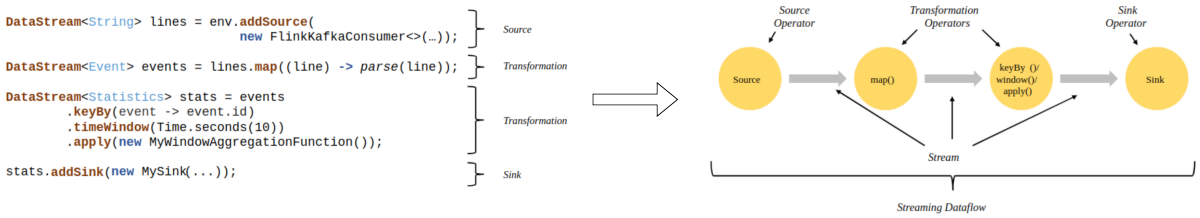


Figure 3: Example Flink application with its key components

Stateful stream processing

> Whilst many operations in a dataflow simply look at one individual event at a time (for example an event parser), some operations remember information across multiple events (for example window operators). These operations are called stateful.

In order for the operators to remember past events the stream must be **keyed**. In order for a keyed stream to be created, the ***keyBy()*** function is used on a specific property of each record. Records with the same chosen property (e.g. type of event, IP address etc.) will end up in the same stream partition. As a result, state can be used only in a per key basis.

Keyed state is stored and maintained in what can be thought of as an embedded key/value store. This state is distributed and divided alongside the data streams that are consumed by stateful operators. That is why, access to the key/value state is only possible on keyed streams, i.e. after a keyed/partitioned data exchange, and is restricted to the values associated with the current event's key, as mentioned before. Aligning the keys

of streams and state makes sure that all state updates are local operations, guaranteeing consistency without transaction overhead. This alignment also allows Flink to redistribute the state and adjust the stream partitioning transparently. Global state is not available or recommended since it would introduce complexity in the distributed and fault-tolerant design of Flink. An example of keyed state with 6 keys and 2 stream partitions can be seen in the below figure. The state is split between the 2 parallel instances of the stateful operator.

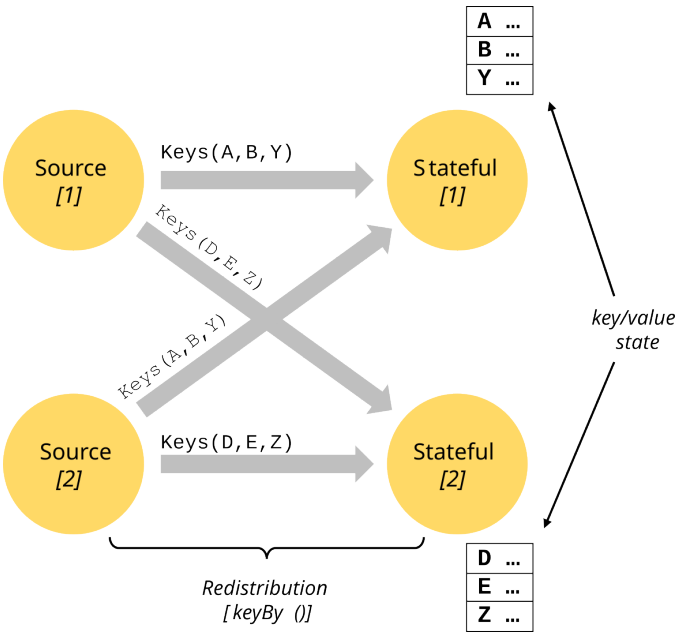


Figure 4: Flink's keying and keyed state mechanisms

State backends define the exact data structures in which the key/values indexes are stored. The heap state backend stores data in an in-memory hashmap, while the RocksDB backend uses a RocksDB instance as the key-value store. Furthermore, each state backend implements the logic for point-in-time snapshots and store these snapshots as part of the checkpointing process, ensuring fault tolerance. Flink provides a transparent way of configuring the desired state backend, without changing the application logic.

Timely stream processing

> Timely stream processing is an extension of stateful stream processing in which time plays some role in the computation. Among other things, this is the case when you do time series analysis, when doing aggregations based on certain time periods (typically called windows), or when you do event processing where the time when an event occurred is important.

When referring to time in a streaming program (for example to define windows), one can refer to different notions of time: processing time and event time.

Processing time refers to the machine time of the machine that is executing the respective operation. In processing time mode within a streaming program, time-based operations like time windows rely on the system clock of the machines executing the code of the operators. For instance, an hourly processing time window collects data between the times when the system clock indicates the full hour. So, if the application starts at 9:15 am, the first hourly window covers data from 9:15 am to 10:15 am, and subsequent windows follow suit. Processing time offers high performance and low latency but lacks determinism in distributed and asynchronous environments, due to factors like varying record arrival speeds and system outages.

Event time is the time that each individual event occurred on its producing device, and is associated with each record before the latter enters Flink. When using event time, time progression depends on the events' timestamps. When events arrive in order or all events have arrived, this timing mechanism provides consistent and deterministic results. However, when out-of-order events arrive, latency is induced. As it is only possible to wait for a finite period of time, this places a limit on how deterministic event time applications can be.

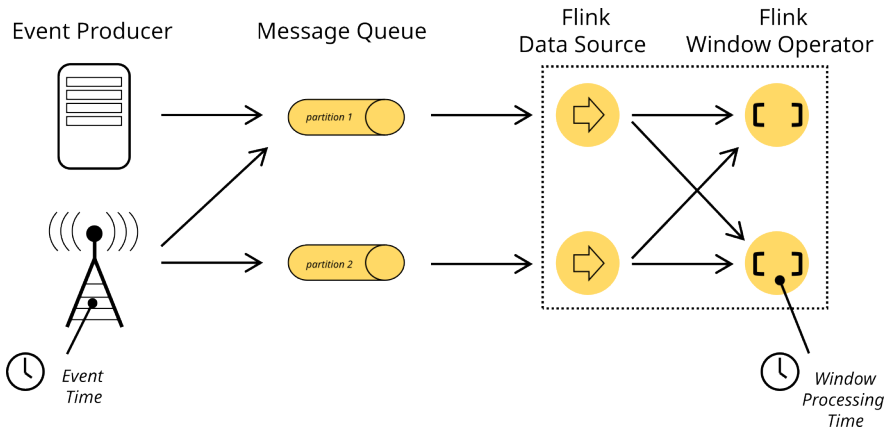


Figure 5: Event time in a Flink job

Parallel execution

Programs in Flink are inherently parallel and distributed. During execution, a stream has one or more stream partitions, and each operator has one or more operator subtasks. The operator subtasks are independent of one another, and execute in different threads and possibly on different machines or containers. The number of operator subtasks is the parallelism of that particular operator. Different operators of the same program may have different levels of parallelism.

A Flink operator can either transmit data directly into the next operator in a one-to-one pattern or distribute the data into many instances of the next operator in a distributing pattern. In a one-to-one scenario each partition will hold the initial data in its initial order until the end of the processing lifecycle. On the contrary, a redistributing pattern will

change the partition of the elements and place each one on a specific partition, usually based on a key or a rebalancing policy provided by the user with the use Flink's API.

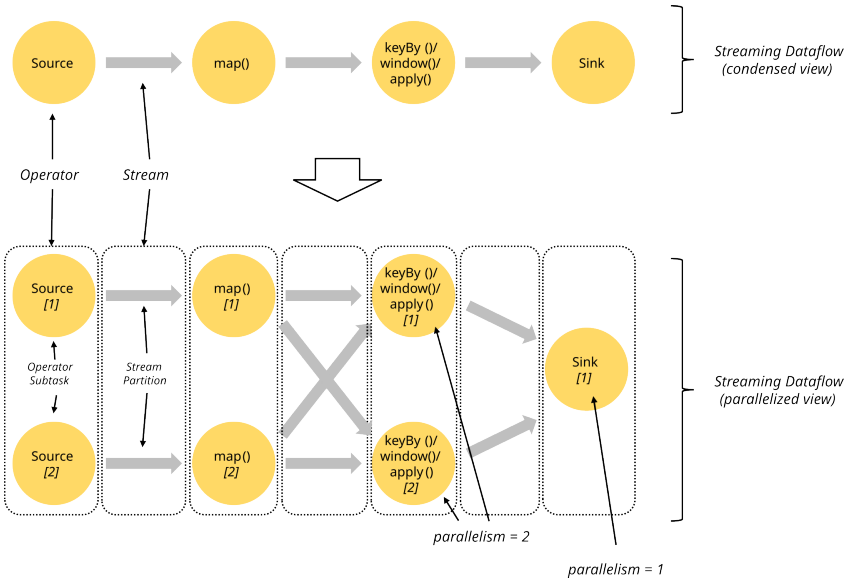


Figure 6: Parallel dataflow in a pipeline

Architecture

In order to achieve parallelism, fault tolerance etc., Flink's runtime relies on two types of processes: The JobManager and TaskManager processes.

JobManager is a crucial part of the framework. It plays an important role in managing Flink applications, coordinating and scheduling tasks across the cluster and ensuring fault tolerance. The JobManager is responsible for job submission, reacting to failed tasks and ensuring recovery when configured to do so. In order to achieve the aforementioned requisites, it uses the following components:

- ***ResourceManager:*** The process responsible for allocation and deallocation of resources inside the cluster. Manages the **task slot**, which is the unit of resource scheduling in the cluster. There are many implementations of ResourceManager to accommodate for the different cluster management systems like YARN and Kubernetes. When such systems are used, ResourceManager can also start new TaskManager processes contrary to standalone deployments where it can only distribute task slots in a single TaskManager.

- ***Dispatcher:*** The process responsible for providing a REST interface to submit a Flink job, and starting a new JobMaster for each submitted job. The Dispatcher also runs the Flink WebUI, which provides information about the JobManager and TaskManagers, different metrics and logs for each job, as well as a graphical interface for submitting and canceling jobs.

- ***JobMaster:*** The process responsible for managing the execution of a single job. Each job running in a Flink cluster has its own JobMaster.

TaskManager processes are the workers of the cluster, responsible for executing the tasks of the given job, buffer and exchange the data streams. There must be at least one TaskManager for an application to be executed. Each TaskManager contains at least one task slot which is the smallest unit of execution in a cluster. The number of task slots in the task manager denotes the number of concurrent processing tasks.

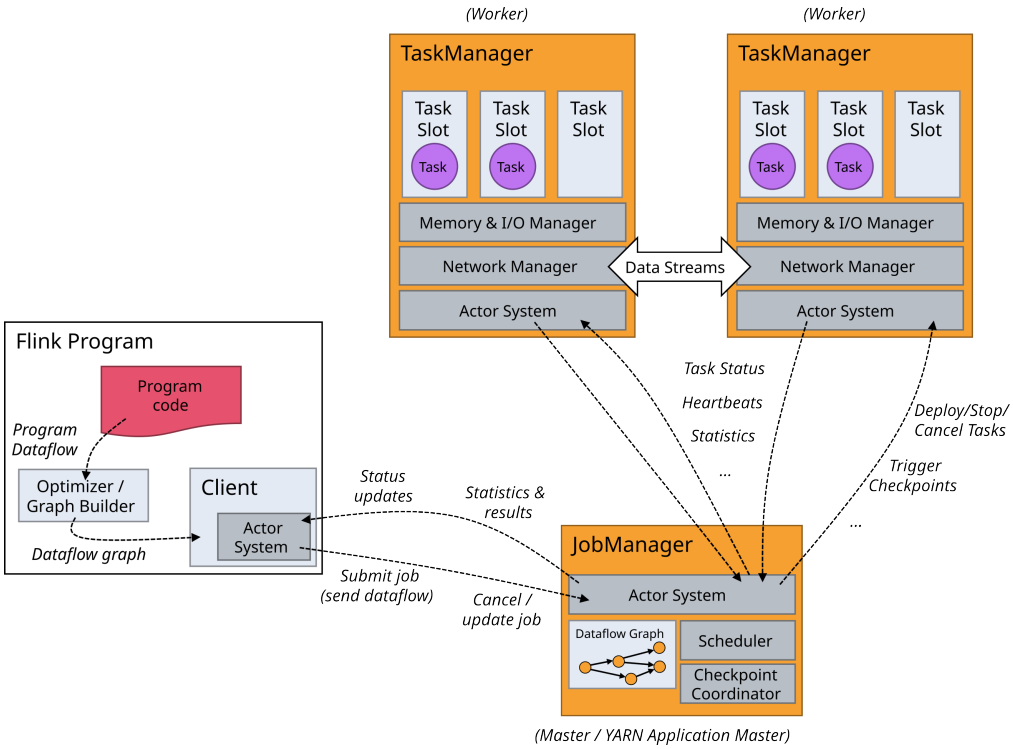


Figure 7: Flink's runtime architecture

Tasks and operator chains

The concepts of tasks and subtasks are fundamental to understanding how Flink distributes and executes processing tasks in the cluster. Each operator or chain of operators is a *task*. If the configured parallelism of an operator is greater than one, then each parallel instance of the operator is called a *subtask*. For purposes of distributed processing, Flink can chain different operators' subtasks into tasks. It is important to note that a chain can be created only if no network action (i.e. ***keyBy(), shuffle(), rebalance()***) has occurred between the chained operators. For example, if a ***keyBy()*** function is placed between a set of operators, they cannot be chained together. Each task is executed by one thread. Chaining is useful since it prevents thread-to-thread communication when unnecessary. Chaining behavior can be configured to either be enabled or disabled, depending on each user's needs. In case that it is disabled, each operator subtask is essentially a task and

will be executed by a thread. Figure 6 shows the parallel execution of five subtasks, each executed by a thread, resulting in five threads: The source-map chain operates with a parallelism of 2, utilizing 2 threads. In contrast, the window operator and the sink operator both have a parallelism of 1, each employing a single thread.

Task slots and resources

Each TaskManager is a JVM process and it executes one or more subtasks in different threads (i.e. Figure 6). The number of *task slots* in a TaskManager dictates the number of subtasks it is allowed to process.

Task slots are the smallest units of execution and represent a subset of the TaskManager's resources. These resources are evenly split between the number of task slots. For example, three task slots will use 1/3 of the total CPU and RAM each. These slots will use the CPU concurrently. They also share the JVM, network resources (i.e. TCP connections) using multiplexing, and possibly internal data structures. A Flink cluster needs exactly as many total task slots as the highest parallelism of the job. That is in order to place each subtask of the operator with the highest parallelism, in a separate slot. This enables them to be executed in parallel.

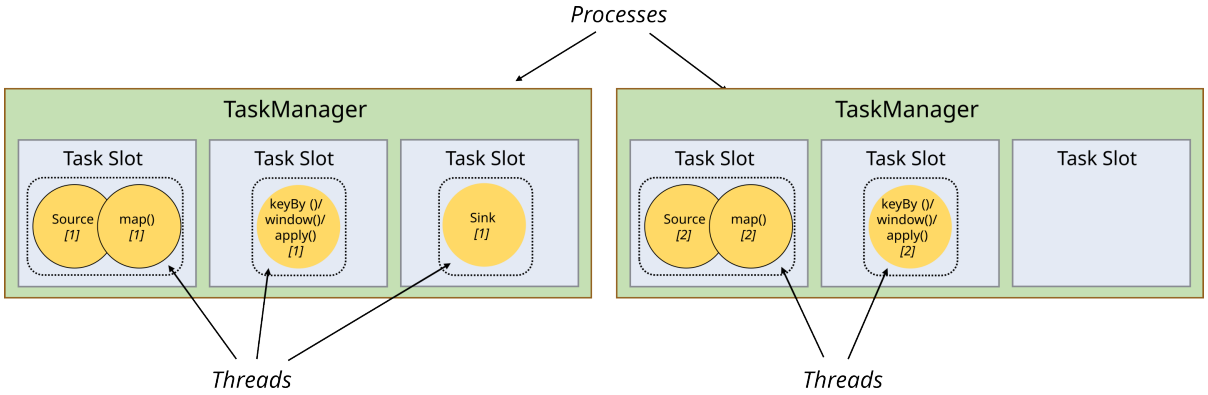


Figure 8: Subtasks in task slots

3.2 Apache Kafka

Apache Kafka is an open-source distributed streaming system used for stream processing, real-time data pipelines, and data integration at scale. Originally created to handle real-time data feeds at LinkedIn in 2011, Kafka quickly evolved from messaging queue to a full-fledged event streaming platform capable of handling over 1 million messages per second, or trillions' of messages per day.

Kafka follows the publish-subscribe model, where producers publish messages to topics, and consumers can subscribe to these topics in order to receive the sent messages. It also is a pull-based system, meaning that consumers can choose when they want to receive messages

by sending a request to the broker. Push-based architectures would deliver the messages to consumers as soon as they are available, and then notify them for this action. Its distributed architecture allows for fault-tolerant, high-throughput and low latency messaging abilities. This design enables Kafka to be incorporated to applications that require highly scalable messaging services. Kafka is consisted of servers (Brokers) and clients (Producers and Consumers). In order for all parts of the system to communicate, a binary protocol over TCP is used.

Brokers are key components of the storage and distribution of data. A broker can be thought of as a single instance of a Kafka server, responsible for managing one or more Kafka topics and their associated partitions. A Kafka cluster can be deployed with either one or more brokers, with the latter providing fault tolerance and better scalability.

Records are the basic units of data exchange in Kafka's environment. They are also referred to as events or messages. Each record contains a key, value, timestamp and optional metadata headers.

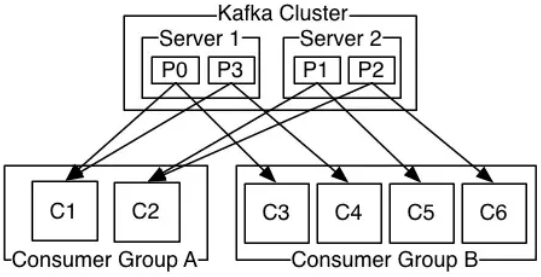
Topics, much like folders in a file system, are logical channels to which messages are published by producers, and from which they are consumed by consumers. Topics serve as a mean for categorizing data in a Kafka broker. Each topic is identified by its name. Contrast to many other messaging systems, Kafka does not delete messages upon consumption. Instead, time or size based retention policies can be configured for each topic independently. For instance, a topic whose messages are consumed slowly, can have a much greater retention period than one that has its messages consumed with a higher rate. Messages that have not been deleted can be re-consumed as many times as necessary. Topics can be replicated, even across different regions, so that there are always multiple brokers that have a copy of the data in case of an outage or broker maintenance etc.

Partitions are a way of breaking down a topic into smaller, independently manageable segments. Each partition holds part of the total data published on the topic it belongs. Every message that is published to a topic is placed on a single partition either based on the key of the record, or a partitioning policy (e.g. round-robin manner). Messages within different partitions can be processed independently by Kafka consumers and can be spread across multiple Kafka broker nodes. This parallelism enhances the overall throughput, scalability and fault tolerance of Kafka. The messages residing in a partition are guaranteed to be in the order they arrived from their respective producers.

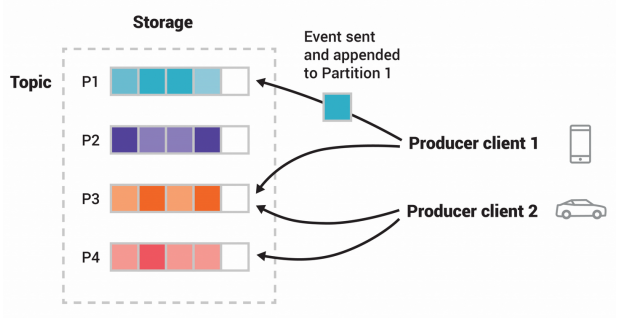
Consumers are client applications that consume (read) records from a Kafka broker. They are fully decoupled and agnostic from producers, which is a crucial design element in order to achieve the scalability that Kafka is known for. Consumers can be grouped into consumer groups and leverage the topic partitioning, by assigning different partitions to different consumers within the group. Each consumer within the group reads from its assigned partitions independently, allowing for concurrent message processing.

Producers are client applications that publish (write) records to a Kafka broker. They are not required to await the consumption of messages by consumers before dispatching subsequent ones. Furthermore, producers offer fine-grained control over message delivery reliability through acknowledgment settings (acks) and retry configurations. By proper configuration, producers can choose the level of acknowledgment they require, ranging

from no acknowledgment, to acknowledgment from all replicas. This mechanism allows for adaptation for different scenarios, balancing performance and reliability.



(a) Kafka arrangement with 2 brokers and 4 distributed partitions



(b) Distribution of messages from different producers

3.3 Kubernetes

Kubernetes is an open source container orchestration engine for automating deployment, scaling, and management of containerized applications.

Kubernetes is a framework to run distributed systems resiliently providing the following features:

- ***Service discovery and load balancing:*** Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
- ***Storage orchestration:*** Kubernetes allows for automatic mounting of storage systems, such as local storages, public cloud providers, and more.
- ***Automated rollouts and rollbacks:*** Developers can describe the desired state of the application and Kubernetes can change the actual state to the desired state at a controlled rate. For example, Kubernetes can be configured to create new containers for specified deployments, remove existing containers and adopt all their resources to the new container.
- ***Automatic bin packing:*** Kubernetes is provided with a cluster of nodes that it can use to run containerized tasks. The developer can configure the CPU and memory (RAM) for each container according to its specific needs. Then K8s can fit containers onto nodes to make the best use of the cluster's resources.
- ***Self-healing:*** Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to the user-defined health check, and doesn't advertise them to clients until they are ready to serve.

- ***Secret and configuration management:*** Kubernetes allows for storage and management of sensitive information such as passwords, OAuth tokens, and SSH keys. Application configurations and secrets can be deployed and updated without rebuilding the container images and without exposing secrets in configuration files.

Components

When an app is deployed on Kubernetes, a cluster is created. A cluster consists of a set of machines, virtual or physical, called *nodes*. At least one node must be present in each cluster. The worker nodes host *Pods*, which constitute the workload of the containerized applications. A high level architecture of a Kubernetes' cluster can be seen in the Figure 10.

Control plane components manage the worker nodes and the Pods in the cluster. Furthermore, they are responsible for responding to cluster events. For example, a cluster event may be the loss of a pod for a specific deployment. In this case, the *replicas* field number is not satisfied and action is taken by K8s control plane. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

Node components run on every node of the cluster. *Kubelet* is an agent that makes sure that containers are running inside the Pods, and that each of them is healthy according to their respective configurations. *Kube-proxy* is a network proxy that maintains node network rules. These rules allow network communication to Pods, either from networks within or outside the cluster.

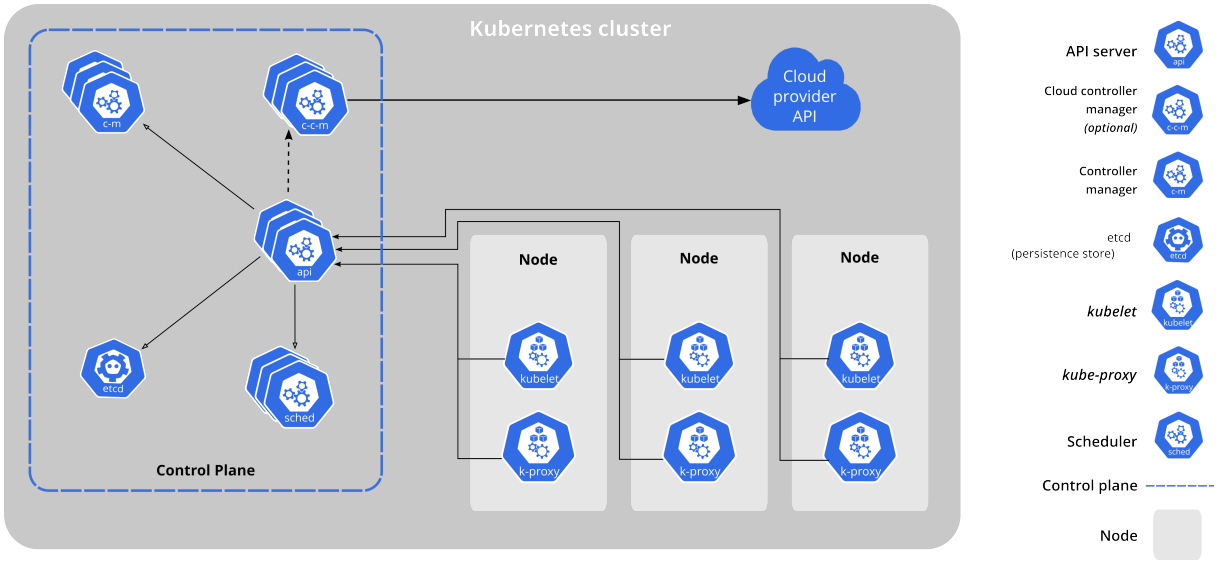


Figure 10: Kubernetes' cluster architecture

3.4 Flink Kubernetes Operator

A Kubernetes operator is a method for application packaging and deployment that extends Kubernetes' capabilities to manage more complex, stateful workloads. Examples of these workloads include databases, distributed systems that use quorum mechanisms and applications that cannot easily be reconfigured in a single step.

The Flink Kubernetes Operator extends the Kubernetes API with the ability to manage and operate Flink Deployments. The operator acts as a control plane to manage the complete lifecycle of a Flink deployment, as well as provides logging and metrics integration with tools such as Prometheus. The objective of this operator is to automate activities which would otherwise require the expertise of a human operator. Said automations cannot be achieved through the Flink native integration alone. An example deployment using the operator can be seen in Figure 11.

The operator is installed with the help of Helm directly on the Kubernetes cluster. Once installed, the user creates YAML files with the required Flink deployments. By configurations contained in these files, the user can not only manage the specs of the TaskManagers and JobManagers, but can also select the nodes where they will be placed, according to their needs. Last but not least, the user specifies the required JAR file containing the Flink application along with the desired parallelism and checkpointing behavior.

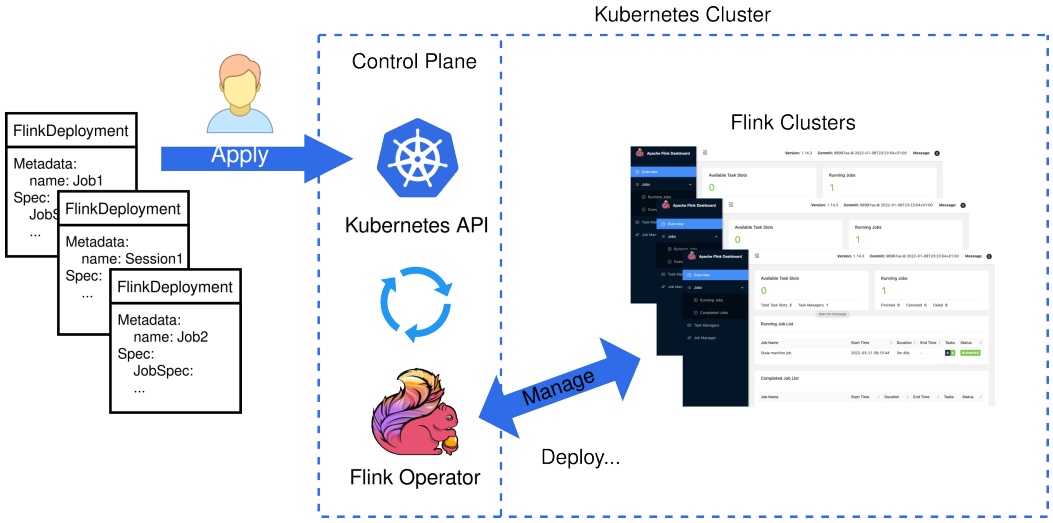


Figure 11: Kubernetes operator deployment

3.5 Google Cloud Platform

Google Cloud Platform is comprehensive cloud computing platform offered by Google that has had a profound impact on the field of cloud technology and services. GCP includes a broad range of cloud-based solutions, including computing, storage, databases, machine learning, and data analytics, among others. It is renowned for its large data centers around the world, providing high-performance services. Key GCP services include Google Compute Engine for virtual machines, Google Kubernetes Engine for container orchestration, BigQuery for data analytics, and TensorFlow for machine learning.

3.6 OpenCV

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. The library has more than 2500 optimized algorithms, which include a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. In this Thesis OpenCV is used to facilitate both background subtraction and connected component labeling.

In order to use this library in Java projects, the OpenCV JAR file must be created. This is accomplished by building the source code with specific build directives and the use of the Java Native Interface (JNI) framework. Said framework enables Java code running in the Java Virtual Machine (JVM) to call and be called by native applications and libraries written in other languages like C, C++ and assembly. This allows the Flink code to access OpenCV's functions at runtime.

3.7 Background Subtraction - Foreground detection

Background subtraction is a computer vision technique used in image and video processing, to separate the foreground objects or regions of interest from the background in a given image or video sequence. For video, the goal is to extract the relevant information from an image or video by removing the stationary or unchanging parts of the scene, which are considered the background. It is used in a variety of computer vision tasks, such as gesture recognition and object detection and tracking, where the isolation of foreground objects simplify further processing and understanding of video data. Furthermore, background subtraction is employed in image and video editing tasks, where foreground extraction is necessary for creating special effects or background replacement.

While being a fundamental technique, background subtraction presents some challenges that must be addressed. Variation in lighting or illumination changes can result in false positives or true negatives in the detection process. Moreover, non-static backgrounds such as moving foliage or image movement as a result of camera jitter, can distort and cause noise in the resulting foreground. This can significantly undermine the performance, accuracy and reliability of subsequent algorithms (e.g. connected component labeling), since most of them heavily rely on an accurate foreground segmentation.

Numerous approaches for background subtraction have been developed by researchers and engineers over the years in order to address the aforementioned challenges. Among the diverse range of methods, we will discuss five different notable approaches: Frame Differencing with and without running average [16], [17], Gaussian Mixture Models (GMM) [18], the KNN background subtraction method [19], and Deep Learning approaches [20].

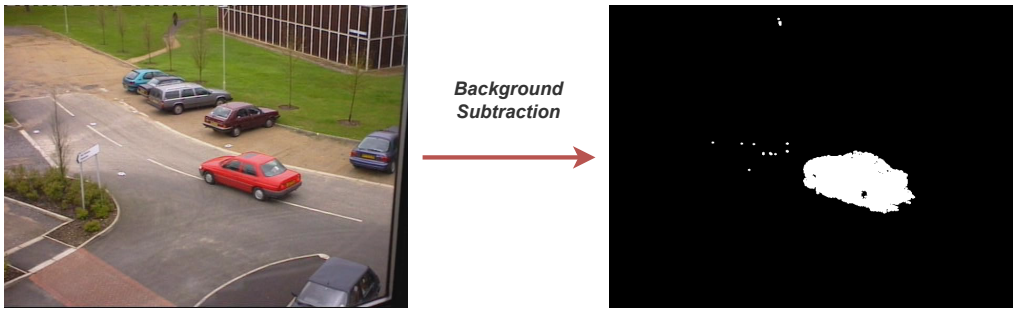


Figure 12: Background subtraction visualization

Frame Differencing is a straightforward background subtraction technique that calculates the pixel-wise difference between the current frame and a reference frame (typically the first frame in the sequence). Pixels with significant differences are considered part of the foreground. There are two variations of frame differencing, regarding the modeling of the background image. Static frame differencing selects the first frame of the video sequence as the background model and never updates it, while the dynamic version updates the background model over time, considering the previous model and the current image. The second version is referred to as the running average approach. The equations of frame differencing are presented below:

$$FG(x,y,t) = \begin{cases} 1, & |I(x,y,t) - BG| > Th \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

$$BG_{static}(x,y,t) = I(x,y,0) \tag{2}$$

$$BG_{dynamic}(x,y,t) = (1-\alpha) \cdot BG_{dynamic}(x,y,t-1) + \alpha \cdot I(x,y,t) \tag{3}$$

Where $I(x,y,t)$ denotes the pixel at position (x,y) in frame t, $BG_{static}(x,y,t)$ denotes the pixel at position (x,y) of the background image in frame t using the static background model, $BG_{dynamic}(x,y,t)$ denotes the pixel position in the background image using the dynamic background model, α denotes the learning factor of the dynamic model (i.e. how much impact does the current frame have over the background model) and Th denotes the foreground decision threshold. When the frame difference $|I(x,y,t) - BG|$ is greater than Th, the pixel is classified as foreground.

This method is effective for relatively stable backgrounds and gradual illumination changes. Moreover, due to the simple nature of the necessary computations, this method is quite efficient and suitable for applications that require fast background subtraction. On

the contrary, it cannot handle sudden background variations well and may require careful parameter tuning such as α and Th. Currently, the literature does not provide methods for adaptive selection of α and Th. These parameters are user defined and heavily depend upon the video in question.

Deep learning-based approaches for background subtraction utilize convolutional neural networks (CNNs) to learn complex features and patterns in the data. These models are trained on labeled data to distinguish foreground from background. They are highly adaptable and capable of handling complex scenes with dynamic backgrounds and illumination changes. The pursuit of high accuracy and adaptability is often accompanied by the requirement for large amounts of training data and training time, followed by reduced performance when significant computational resources (e.g. GPUs) are not available.

Gaussian Mixture Models is a technique for background subtraction that treats foreground segmentation as a data clustering problem. It was first introduced in by Stauffer et al. in [18], producing stable results and reliability with lighting changes. This approach is based on modeling pixel intensities as a mixture of Gaussian distributions $\mathcal{N}_i$.

Each pixel is tracked over time (i.e. frame progression) and fitted with K Gaussian distributions. K is usually chosen between 3 and 5, in order to balance the accurate depiction of the pixel's color intensities' range and the amount of needed computations. Each Gaussian distribution characterizes a specific range of pixel intensities observed in the historical frames and is represented by its mean and variance. When working in the RGB color space, the needed Gaussian distributions are three-dimensional with means $\hat{m_i} = (m_r, m_g, m_b)$ and covariance matrices $\Sigma_i = \sigma_i \mathbb{I}$ For instance, let $X_t = I(x, y, t)$ be a pixel of frame t. Suppose that this pixel is modeled with 3 Gaussians, with mean values:

$$\hat{m_1} = (255, 0, 0) \rightarrow \blacksquare \rightsquigarrow w_1$$
$$\hat{m_2} = (0, 255, 0) \rightarrow \blacksquare \rightsquigarrow w_2$$
$$\hat{m_3} = (0, 0, 255) \rightarrow \blacksquare \rightsquigarrow w_3$$

This indicates that during the past frames, this pixel has been red, green and blue. To determine the impact of each distribution in the modeling of the pixel, each one is associated with a weight w_i, denoting the number of times that this pixel was matched to a specific Gaussian. Usually the weights are normalized. The Gaussian components with higher weights typically correspond to the background model (indicating that the pixel has predominantly exhibited the color range represented by this Gaussian). The number of distributions that comprise the background model is given by the following equation, where T is the minimum portion of data that should be accounted for by the background. T is a user defined parameter and may need to be different for different videos. In this work, T is set to 0.9, which is the default value in the OpenCV implementation of the algorithm. This value indicates that a pixel should be part of the background $\approx 90\%$ of the time. For this formula, the distributions are supposed to be sorted with respect to

their weights.

$$B = argmin_b \left(\sum_{i=1}^{b} w_i > T \right) \tag{4}$$

The first B distributions represent the background model. As a result, if a pixel matches either of these Gaussians, it is classified as a background pixel.

If at frame $t + 1$, the pixel does not match either of the 3 Gaussians, it is classified as foreground. In this case, the Gaussian with the least weight is replaced with a new Gaussian based on the input pixel. The formula for matching a pixel X_t to the closest Gaussian N_i can be expressed as:

$$i = argmin_i \left(\frac{|X_t - \hat{m}_i|}{\sigma_i} \right) \tag{5}$$

This formula calculates the index i corresponding to the Gaussian distribution that minimizes the distance of the pixel value X_t from the mean $\hat{m}_i$ in terms of standard deviations σ_i. If $|X_t - \hat{m}_i| \leq 2\sigma_i$ for this selected Gaussian distribution N_i, then the pixel X_t is considered to be "close" to the Gaussian distribution.

When a pixel matches a distribution, the latter has to be updated in order to accommodate for the matched pixel. The equations for updating the matched Gaussian are given below.

$$\hat{m'} = (1 - \alpha) \cdot m + a \cdot X_t \tag{6}$$

$$\sigma'^2 = (1 - \alpha) \cdot \sigma + a \cdot (X_t - m)^2 \tag{7}$$

$$w' = (1 - \alpha) \cdot w + \alpha \tag{8}$$

The α value denotes the learning factor of the model. This determines the level of significance for each update. A smaller learning factor leads to gradual adaptation, making it suitable for stable background environments where noise and small fluctuations need to be filtered out. This minimizes false positives by maintaining a stable background model. Conversely, a larger learning factor facilitates rapid adaptation to sudden changes in the scene, ideal for dynamic environments with frequent variations. This prevents false negatives by quickly accommodating changes. Striking the right balance between adaptability and stability depends on the specific application's requirements, and the choice of the learning factor significantly impacts the accuracy and robustness of background subtraction algorithms. In this work, α is set to 0.01 which is the default value of the OpenCV implementation. As a result, the background model updates slowly, adapting to gradual changes in the scene. The Gaussian mixture models approach is available in OpenCV using `cv2.createBackgroundSubtractorMOG2()` and `cv2.createBackgroundSubtractorMOG()` functions. This method is going to be the main background subtraction method in the proposed system.

The KNN method for background subtraction is a non-parametric method relying on Kernel Density Estimation, along with the KNN algorithm for the voting pro-

cess. This method does not assume any kind of distribution for the data. The probability density function of each pixel is estimated using Kernel Density Estimation, with a Normal Gaussian kernel. This method is going to be used in the performed experiments only to see the differences compared to the Gaussian Mixture Models method. This method is proposed by Zivkovic et al. in [19] and is available in OpenCV using `cv2.createBackgroundSubtractorKNN()`.

3.8 Connected Component Labeling

Connected component labeling is a fundamental video processing and analysis technique, which plays a significant role in the comprehension of video context. This method involves identifying and labeling distinct regions of foreground pixels in binary images. By systematically grouping pixels or regions with shared spatial connectivity, connected component labeling enables the separation of objects of interest from the background and aids in subsequent tasks such as object tracking, shape analysis, and pattern recognition. In this work,

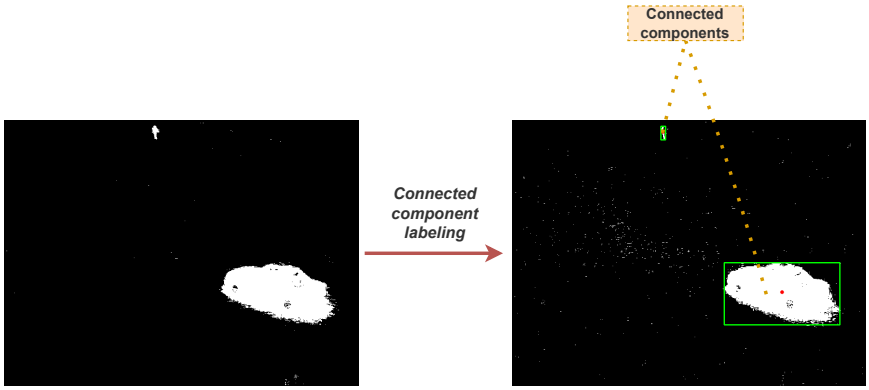


Figure 13: Connected component labeling visualization

the CCL algorithm provides the centroid, area and bounding box for each object, laying the groundwork for tracking and trajectory detection. The entire frame is also not needed in the case of the CCL algorithm too. Given a block of pixels (part of the frame), the algorithm will identify the groups of foreground pixels inside the blocks. Since the whole frame is not available, it is necessary to aggregate and combine the results for each block in a later stage. This process will be elaborated upon in the architecture section. The CCL algorithm is provided by OpenCV with the use of `cv2.connectedComponentsWithStats()` function.

4 System architecture

The system in its entirety consists of three main parts. The clients, the Kafka broker and the Flink cluster. Each client splits the video into smaller parts called blocks and generates a unique key for each block. Then, each key-block pair is bundled into a Kafka record and streamed to the partitions of the designated input topic. Flink consumes the records from Kafka, distributing and then re-grouping them using their keys, ultimately extracting the trajectories of every input video.

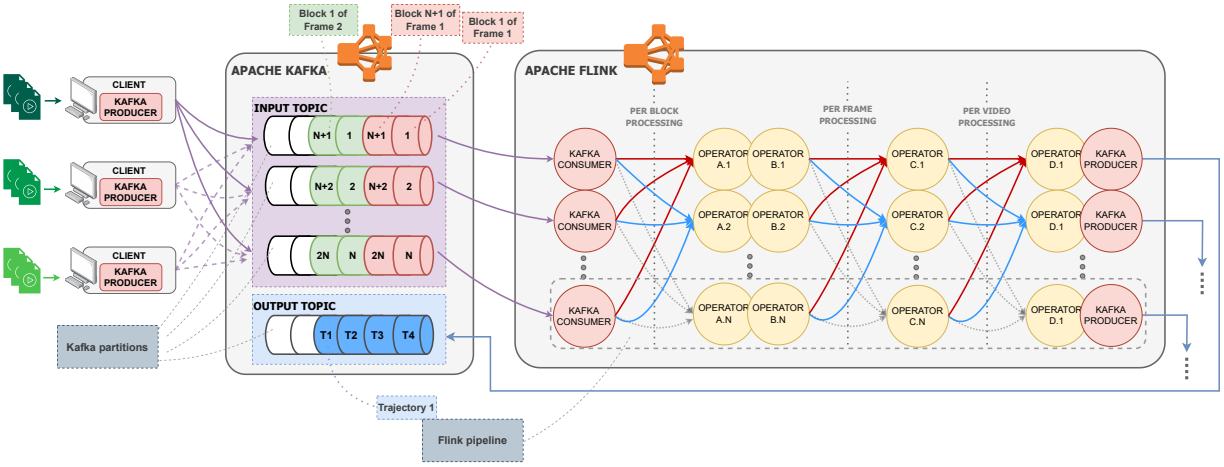


Figure 14: High level system architecture. On the left side, clients are streaming the videos to Kafka. Kafka organizes each topic to partitions. Flink handles the processing and sends the extracted trajectories back to Kafka

Figure 14 shows the overall architecture of the system. In the leftmost section of the diagrams there are the clients. Each client program uses a Kafka producer in order to stream the blocks and their keys to the Kafka broker. The Kafka broker manages two topics, one to hold the blocks sent from clients and one to hold the trajectories extracted from Flink. The input topic has N partitions while the output topic only needs one. Moving along the diagram, our attention now shifts to the right side, where N Flink pipelines handle the processing of each block. A series of 5 different operators - excluding the Kafka consumer and producer - transform and process the input data in order to extract the trajectories of the existing objects. All extracted trajectories are sent to the output topic of the Kafka broker. Each resulting trajectory has a key that corresponds the video that it was extracted from.

The main idea of the system is that each frame is split into small blocks, so as to make fast streaming and easier processing by Flink possible. Each one of the Flink pipelines is equipped with a Kafka consumer operator, in order to consume the blocks from Kafka. The first operator after the Kafka consumer is responsible for performing background subtraction on the input pixel block. In a nutshell, this operator transforms the RGB image of the input block, to a binary image, where the white pixels represent the foreground and black pixels represent the background of the image. Next, each binary image created in the previous step undergoes connected component labeling. This process outputs

every connected component of the input block. A connected component consists of its centroid, bounding box and area, and represents a group of connected foreground (white) pixels. Subsequently, all connected components of each frame are grouped, and a merging algorithm merges components that are part of the same object. This stage outputs the detected objects for each frame in a list format. An object contains the same information as a connected component (i.e. centroid, bounding box etc.). The final stage of the Flink pipeline groups the object lists, arranging them to align with the video's frame order, and implements a centroid matching algorithm in order to generate trajectories for all identified objects.

4.1 Client

The client is a Java program which reads a video file from the disk, splits each frame into blocks, generates a unique key for each block, and streams it into Apache Kafka.

4.1.1 Workflow

The client program begins by parsing user input, extracting essential parameters such as block size, the number of video transmissions, and the path to the video file. Leveraging an FFmpeg process, the program converts the video into raw RGB format. It then proceeds to enter a process that iteratively segments the video into block-size chunks, generating unique keys for each segment, and dispatches them to the designated Kafka topic. Figure 15 displays the process of creating the necessary block stream out of the video frames. It is important to note that each record carries an array with 8-bit value integers that represent R, B and G colors. These values are going to be manipulated and transformed by Flink. Upon successfully transmitting the entire video, the client provides informative feedback, presenting essential metrics such as the total time taken for the operation and the number of bytes transferred.

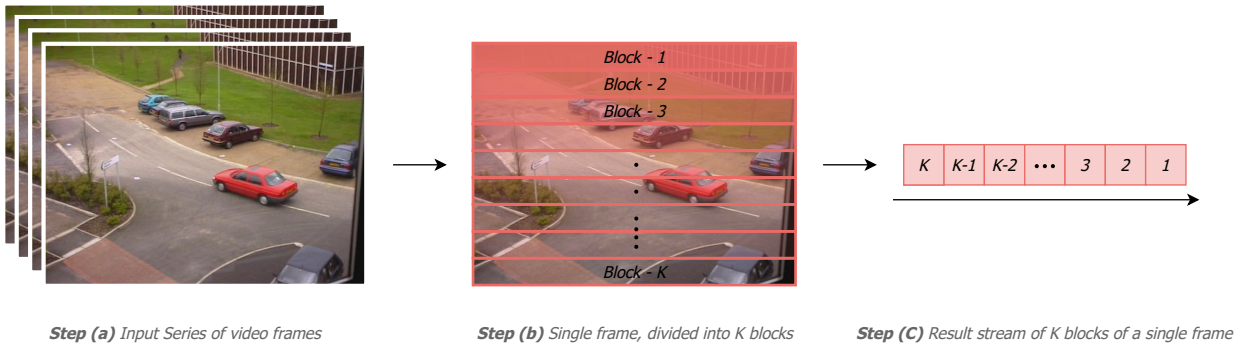


Figure 15: Process for creating a stream of blocks. Each block contains a number of fully populated rows

The messages are distributed to N partitions that belong to the designated input topic in Kafka. Clients have been configured to distribute the messages in a round-robin manner.

In the context of Apache Kafka, this is referred too as round-robin partitioning strategy. Figures below are visual representations of the block distribution for each frame, having one or two clients.

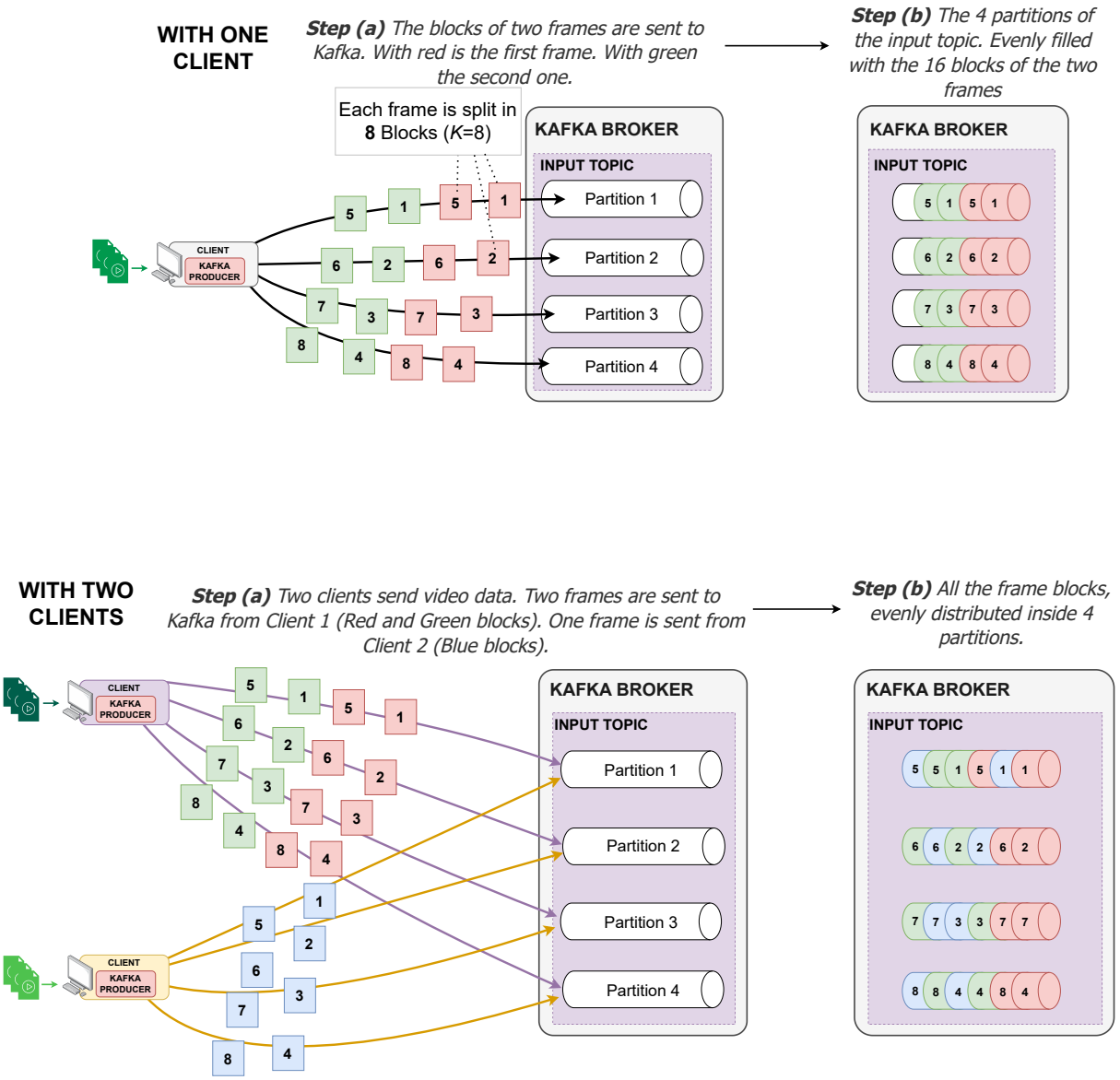


Figure 16: Example message distribution among 4 partitions with one and two clients

4.1.2 Block size

Block size is an important parameter of the client and the system in general. A frame of a high-resolution video in raw color has a significant size. For an RGB888 frame (24 bits

per pixel, 8 bits per color channel) of $2560x1440$ resolution, its total size is:

$$(2560 \times 1440)pixels \times \frac{24bits}{pixel} = 110592000\, bytes \approxeq \textbf{10.54 MBytes}$$

As mentioned before, each frame can be split into smaller blocks. However, there is a caveat: The block size cannot be expressed purely as a memory chunk such as **200KB** or **1MB**. That is because the algorithms for background subtraction and connected component analysis require blocks with orthogonal dimensions, ensuring that each row of the frame is fully populated. In other words, frames must not have partial rows, ensuring that all rows contain a consistent and complete set of pixels. Therefore, in this work the block size is defined as follows. Let H be the frame's height, W be the frame's width and k be the number of rows in the block. Then the following 2 equations must hold:

$$block\_size = k \times W \tag{9}$$

$$H \mod k = 0 \tag{10}$$

Equation 9 gives the block size in pixels, while equation 10 describes the conditions for the possible values of k. For instance, the values of H and W for a 320×240 (240p) video, are 240 and 320 respectively. In order to abide to equation 10, the possible values of k are given below:

$$k \in \{1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 16, 20, 24, 30, 40, 48, 60, 80, 120, 240\}$$

As a result, the block size cannot be completely arbitrary since both the above equations must hold. Splitting a video frame into multiple smaller parts has various advantages. It allows to parallelize the processing of a single frame against multiple worker nodes. Processing smaller blocks requires significantly less CPU utilization than processing large ones. This allows the use of multiple weaker worker nodes instead of fewer and larger ones. The configurability of the block size is an advantage on its own. If the Flink Cluster uses more powerful nodes, a larger size for each block can be set, since the time to process a large block will amount to the time to process a smaller block on a less powerful machine. Alternatively, with weaker nodes, a much smaller size can be configured. This aspect of the client, and the system in general, allows for optimal use of Flink's resources.

4.1.3 Key generation

Key generation is a crucial functionality of the client. Appropriate keys allow Flink pipelines to distinguish between blocks read from different partitions. In addition, using the correct keys enables Flink to correctly distribute and group the initial blocks and the subsequent objects. Keys must not only be unique per video but also per frame and per block, since Flink will need to process data with this granularity.

Each block is assigned a key that contains the *id* of their origin video. Furthermore, it contains the *id* of the frame which the block belongs to, along with its block *id*. Each key is generated automatically by the client just before each block is sent to Kafka and

requires no user intervention. Other information necessary for control sequences in Flink can also be found embedded in the keys: the number of blocks per frame, the width of the video resolution, the number of rows contained in each block and a boolean value denoting whether the referred block belongs to the last frame of the video. All this extra information is necessary for determining when to conclude the algorithms running in Flink's operators. For instance, when merging the connected components of each frame, the result can be forwarded only if *all* the components of the blocks belonging to the frame have been processed. The algorithm uses the `blocksPerFrame` value to determine whether everything has been processed.

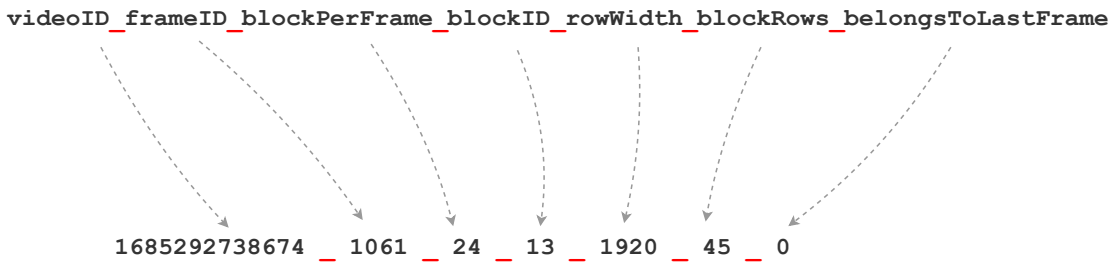


Figure 17: Example key with its various components

Figure 17 shows an example key. The *videoID* component is generated using the timestamp in milliseconds (time elapsed since Unix epoch[3]), at the moment the program started streaming the specific video. The last two digits are randomized to accommodate for clients that start streaming a video at the exact same time. This key pertains to the 14^{th} (using a 0-indexed numbering system) block within the 1061^{th} (using a 1-indexed numbering system) frame of the video identified by the unique *id* 1685292738674. The width of the video's resolution is 1920, each frame contains 24 blocks with each block containing 45 rows, and the specific block does not belong in the last frame.

If more functionalities are added to the system in the future, the key can be extended as needed. For example, a value which will determine the background subtraction algorithm for each video can be added, if the videos require processing with different background subtraction algorithms.

4.1.4 Acknowledgments of receipt

Clients contain Kafka producers in order to send the records to the Kafka broker. Producers can be configured to require one of three acknowledgment levels from the broker. In this system, the highest acknowledgment level among the available three is mandated (`acks=all`). This decision is made to mitigate the risks associated with missing block data. If not all blocks are processed by the Flink pipelines, a result cannot be produced. The potential omission of a block can significantly impact the connected component merging and centroid matching algorithms, potentially causing indefinite waiting as they await the missing block.

[3]https://en.wikipedia.org/wiki/Unix_time

4.2 Kafka broker

The Kafka Broker serves as the central hub for receiving messages from the clients and storing them within partitions designated for the input topic. Additionally, the broker supplies Kafka consumers within the Flink pipelines with messages when they request for more data. This process employs two distinct topics, as illustrated in Figure 14. One for the input data and another for the output data.

Clients utilize the input topic to transmit video data blocks, which are subsequently consumed by the initial Flink operator. This topic is configured with a number of partitions equal to the parallel Flink pipelines. This allocation is essential to ensure that each pipeline consumes data from a different partition. Furthermore, employing multiple Kafka consumers in conjunction with multiple partitions significantly enhances Kafka's overall throughput when compared to a setup involving a single partition and a single consumer.

Once Flink has completed the processing of a video, it forwards all extracted trajectories to the specified output topic. Notably, the output topic operates with a single partition.

4.3 Flink pipeline design

Each Flink pipeline consumes records from a specific partition of the input Kafka topic. These records contain an array of raw RGB values, that represent the frame block. Moreover, these records contain the keys that define the origin of each block ($blockID$, $frameID$, $videoID$). Records can originate from either real-time streaming data sources or scheduled batch processes, since Flink can handle both types. Multiple videos may coexist concurrently in the pipeline. The results of multiple videos are tagged with appropriate keys in order to be retrieved independently.

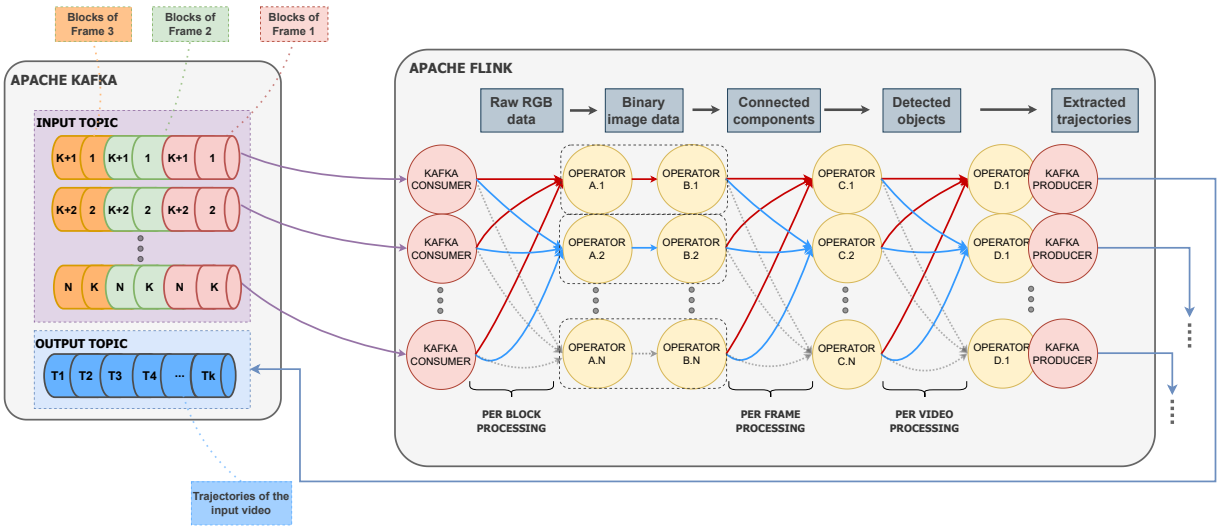


Figure 18: A more detailed Flink diagram, along with the results of each operator.

Figure 18 shows a more detailed Flink pipeline, displaying both the granularity level of the processing (per block, per frame and per video) and the intermediate results of the

operators. Four operators, A, B, C and D (excluding the source and sink operators) can be seen along with the intertwined data streaming flows. Parallelism set to N for the entire pipeline means that each operator will have N parallel instances. The source operator is not coupled with operator A since a ***keyBy()*** stream redistribution process is required in between. In contrast, operator D is coupled with the Kafka sink operator in order to send (produce) the extracted trajectories to the output topic.

The output of each operator is located in the gray blocks at the top of the Flink pipeline, after each operator. The arrows between each pair of operators show the re-distribution of the stream according to the needs of the second operator of the pair. For instance, operator C requires all the connected components of a specific frame in order to output the detected objects. These connected components previously existed in multiple instances of operator B, but where grouped and transmitted to operator C. Opposed to operator C, operator B operates on the stream without the need for any specific grouping criteria. As a result, operator A can just forward its results directly. Since no network action takes place between operators A and B, Flink can chain them together in order to be processed by the same thread.

Keys and Keyed State

In Flink, a regular data stream (such as the combined output from all parallel instances of Operator B) can be transformed into a keyed stream based on a specific identifier (for example, the unique ID of each frame), by using the ***keyBy()*** function. This key-based approach is employed to gather together the outcomes of all prior operator instances, much like the connected components that were mentioned before. Each set of grouped records is then processed on an instance of the subsequent operator.

For every distinct key, a corresponding state is established. This state, associated with each key, is entirely managed and stored within a single operator instance. Consequently, any output from the preceding operator sharing the same key (with the key containing a subset of the key components of Figure 17) is directed exclusively to that corresponding operator instance.

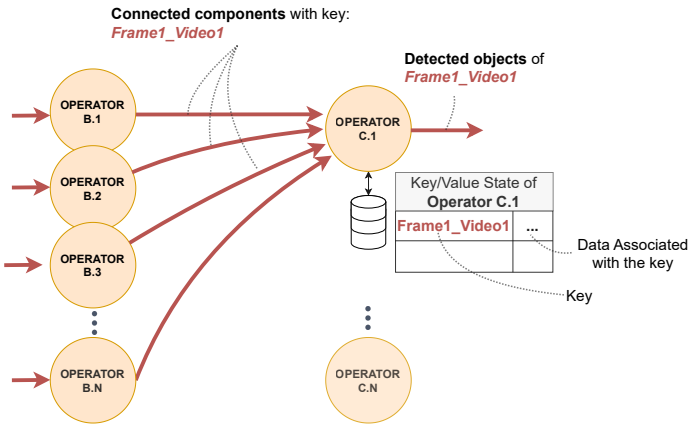


Figure 19: Keys and keyed state illustration

4.3.1 Operators

Source operator – Kafka consumer

The source operator consists of a Kafka consumer that requests records from the input topic of the Kafka broker. Messages coming from Kafka have been serialized in order to be transmitted by the custom TCP protocol of Kafka. The source operator acts as a deserializer, providing an interface for transforming the Kafka record into a Java object. In this case the Kafka record is transformed into a *ByteBlock* object, which contains all the necessary fields of the key (i.e. videoID, blockID, frameID etc.), plus the array of RGB values of the record. An illustration of the source operator functionality can be seen below.

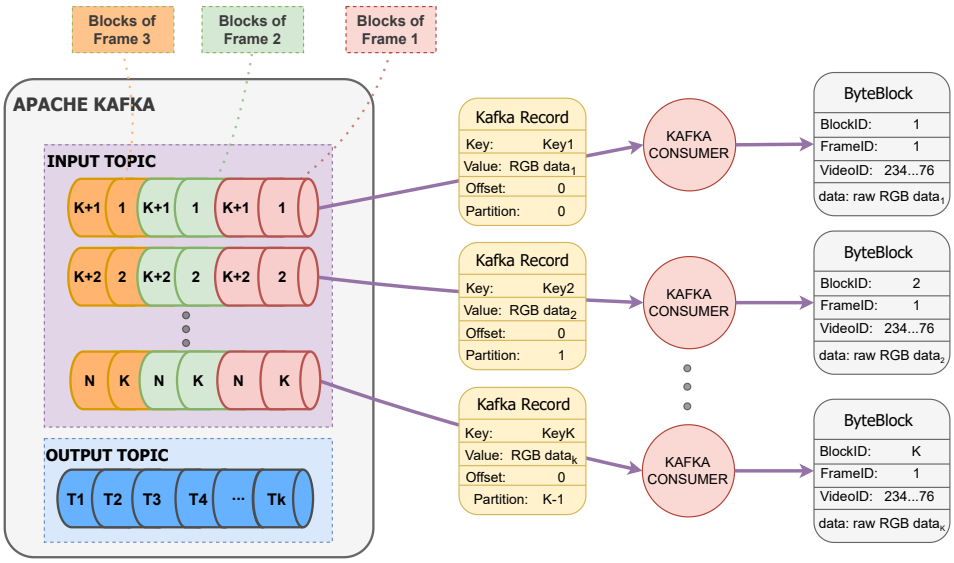


Figure 20: Transformation of Kafka record to ByteBlock objects. The source operator deserializes Kafka records, and assembles and forwards ByteBlock objects.

Operator A – Background subtraction

Operator A performs background subtraction on an input *ByteBlock* object. The array of RGB data is passed as an argument into the background subtraction methods from OpenCV, and an array representing a binary image is returned. The result of the background subtraction operator undergoes noise reduction using OpenCV's morphological operations[4]. Background subtraction is a stateful process. It relies on learning and updating the statistical model of the background over time. This model includes information about the distribution of pixel intensities in the background. This introduces two constraints, related to which blocks can be forwarded to each parallel instance, and the order in which these blocks are processed by each parallel instance.

[4]https://docs.opencv.org/4.x/d9/d61/tutorial_py_morphological_ops.html

Addressing the constraints: To achieve accurate background subtraction, the relevant algorithms take advantage of temporal consistency. Each pixel is modeled according to its counterparts in the previous frames. In order to achieve this functionality in the Flink pipelines, three conditions must be met:

- Blocks of the same video, with the same $blockID$ must be processed by the same parallel instance of operator A
- Blocks that arrive in each parallel instance must be processed in the order of the frames they originate from
- The background subtraction object must be stored in the state of each parallel instance (of operator A) in order to have information about the same block from previous frames available.

The first condition is met by keying the stream with respect to the $blockID$ value (per block processing). This is the first stream redistribution in the Flink pipeline, and can be seen in Figure 18. This mechanism forwards blocks that have the same $videoID$ and $blockID$ to the same parallel instance of operator A. ***The second condition*** is met with the implementation of an online sorting algorithm using the block's $frameID$ as the sorting criterion. The sorting of the blocks is a key functionality operator A and allows for correct background subtraction. Simply, we instruct operator A to process only the blocks that are in the correct order regarding the frames. Any blocks that are not in sequential order are temporarily stored in the instance's state until it's time for them to be processed. ***The third condition*** is met by simply storing the background subtraction object in state. That way information from previous blocks can be preserved and accessed at any time.

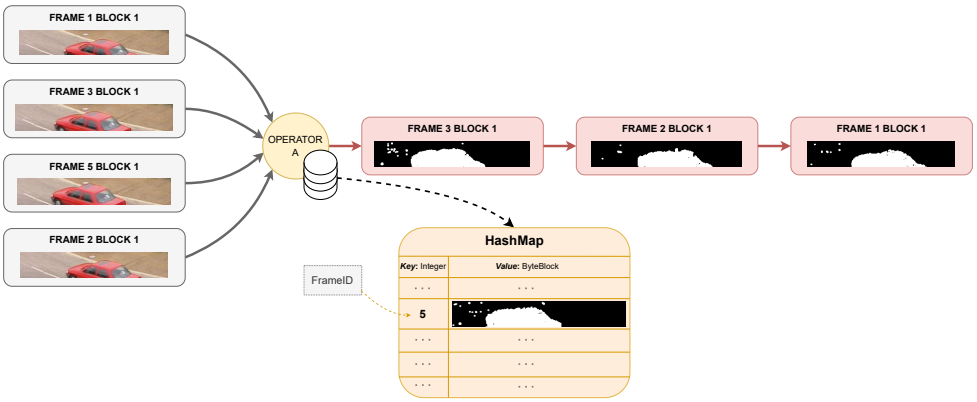


Figure 21: Illustration of operator A functionality

The processing of the blocks in the correct order relies on a HashMap and a counter variable. Both are stored in state. The counter simply indicates the $frameID$ of the block that needs to be processed next. For instance, in Figure 21, the counter variable is set to 4, since the next block that should be processed is that of frame with $frameID = 4$. When the incoming block's $frameID$ value is different from that of the counter, it is stored

in the mentioned HashMap. The HashMap's key is the $frameID$ of every out-of-order block and its value is the ByteBlock object. When block with $frameID = 4$ is eventually processed, all eligible subsequent blocks that were stored in the HashMap (such as block with $frameID = 5$ in Figure 21) are processed too. The pseudocode of the algorithm can be seen below.

Algorithm 1 Process the incoming blocks in the correct order

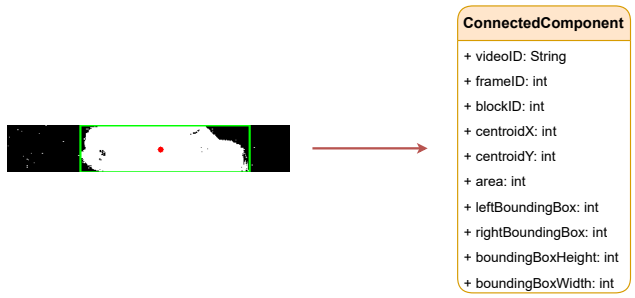
```
1: procedure OnlineSort(inputBlock, expectedFrameID, outOfOrderMap)
2:     currentFrameID ← inputBlock.getFrameID()
3:                                                                    ▷
4:     outOfOrderMap.add(currentFrameID, inputBlock)
5:                                                                    ▷
6:     while outOfOrderMap.contains(expectedFrameID) do
7:         blockToBeProcessed ← outOfOrderMap.get(expectedFrameID)
8:         outOfOrderMap.remove(expectedFrameID)
9:         performBackgroundSubtraction(blockToBeProcessed)
10:        expectedFrameID ← expectedFrameID + 1
11:    end while
12: end procedure
```

The outputs of operator A are also *ByteBlock* objects, with the original RGB array replaced by the binary image array.

Operator B – Connected component labeling

Operator B performs connected component labeling in the input ByteBlock object which contains the binary image. The array containing the binary image data is passed as an argument to OpenCV's `cv2.connectedComponentsWithStats()` method. This method returns the number of connected components in the given block, along with their centroids, bounding boxes and areas. The operator generates an output for every connected component it detects in the given block. Each detected component contains all the necessary information for identification. The area of each resulting connected component is compared against the noise threshold which is set as the 0.01% of the total block area. This value corresponds to single digit areas for most of the selected block sizes. Connected components that have areas smaller than the threshold are discarded, since they are highly likely to be noise. Operator B is the last operator in the pipeline that processes pixels. From this point on, every algorithm operates with information from the connected components (centroid, bounding box, area).

An illustration of how a connected component is represented inside the Flink pipeline can be found on Figure 22. The actual functionality of operator B is quite simple, as shown on Figure 23. In fact, this is operator with the least complexity of the entire pipeline, since it performs a one-to-many transformation without the use of state.



(1) An actual connected component (2) UML representation of the ConnectedComponent class

Figure 22: Class representation of a connected component

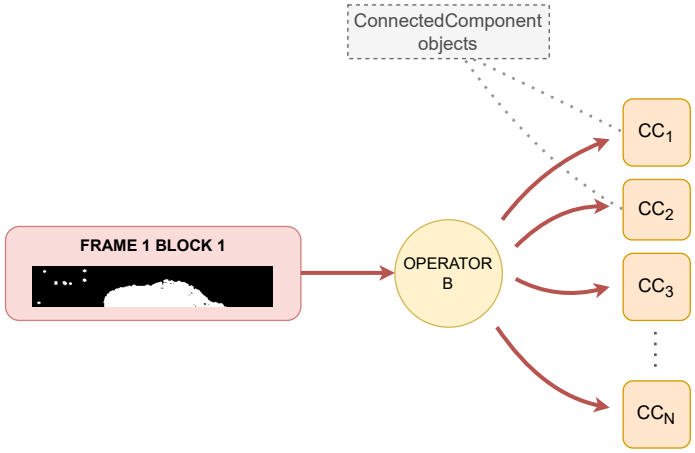


Figure 23: Illustration of operator B functionality

Operator C – Connected component merging

Operator C is responsible for running the process to perform connected component merging. This operator gathers all the blocks within a particular frame and combines those components that meet specific merging criteria, specifically, adjacency and overlapping between components. A merge is defined as the process of combining two or more connected components. The merging process typically involves updating the representation of the connected components, such as modifying their bounding boxes, centroids and areas. A semantic illustration of the described procedure can be seen in Figure 24. The result of the merging algorithm and thus the operator's is a list of connected components, that now represent the objects present in the frame. It should be strongly emphasized that this algorithm operates exclusively with the information of connected components (i.e. centroids, bounding boxes etc.), not with individual pixels or pixel arrays.

As evident from the preceding figure, the objects of a frame may be split among many blocks. In order to merge the necessary connected components, an algorithm has to iterate over every component of every block, and check if it can be merged with other components. Any specific connected component can be merged with any of the following:

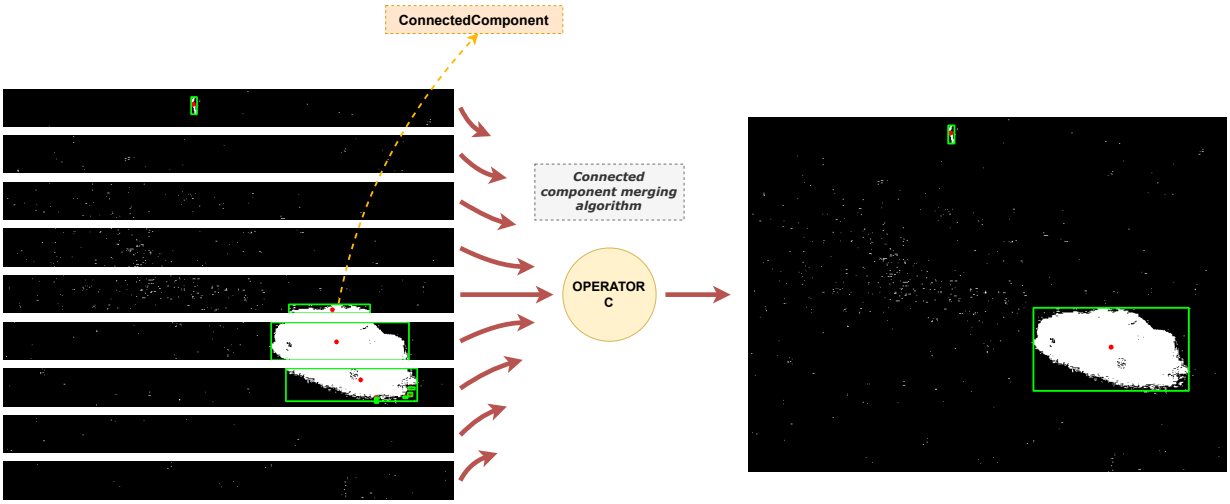


Figure 24: Semantic illustration of the connected component merging algorithm

1. Components from the same block
2. Components from the above block
3. Components from the below block
4. Already merged components

In order to implement the algorithm, the state must be used to hold the necessary data structures. There are two main data structures that are important for the correct implementation of the algorithm. The initial data structure is a HashMap where the $blockID$ serves as the key, and the associated values are lists of connected components. The second data structure is a list that holds all the connected components that have been merged before. In this list, there cannot be any pair of components that satisfy the merging criteria.

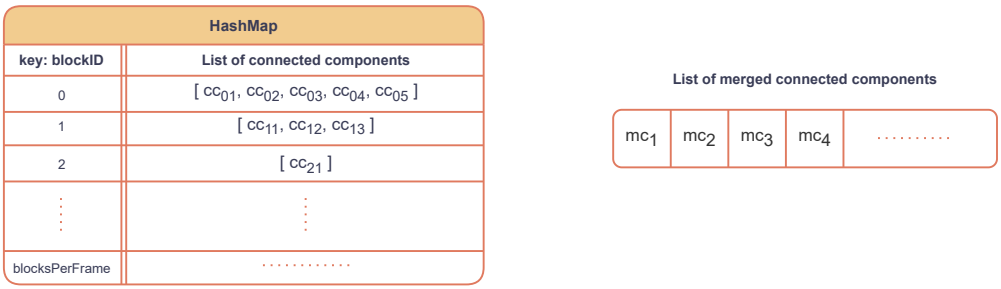


Figure 25: The necessary data structures for the implementation of the connected component algorithm

As previously mentioned, the criteria for merging involve examining adjacency and overlap between components. Adjacency typically pertains to components that are associated with consecutive blocks. To assess adjacency, each side of a block's bounding box is examined

to determine if it directly touches or adjoins the corresponding side of another block. In our case, the adjacency criterion usually applies to components that belong to consecutive blocks. To assess overlapping between connected components, we simply check if their bounding boxes overlap. This criterion usually applies to same block component merging. Both criteria apply to merging with already merged connected components. In both cases, the emphasis is on the existence of overlap or adjacency, with no specific concern on its extent or magnitude.

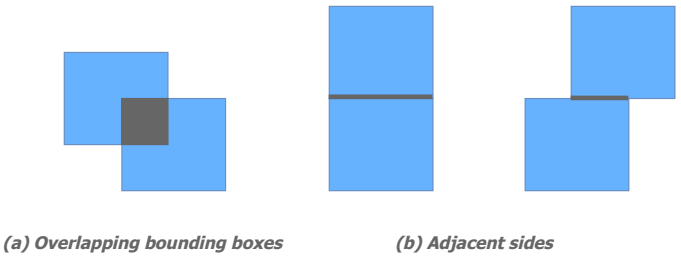
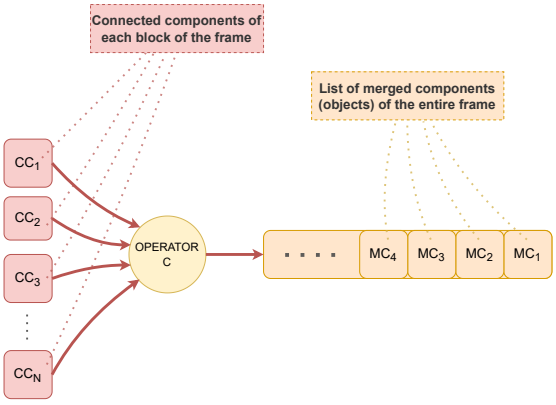
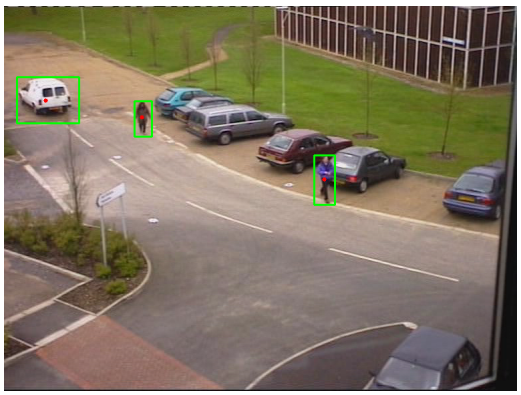


Figure 26: Visual representation of the merging conditions

Algorithm 2 shows a high level pseudocode implementation of the merging algorithm. Lines 8 - 29 show the merging of the input connected component with components from the same, above and below block respectively. Lines 32 - 37 show the merging of the input component with the already merged components. The output of the operator is a list containing the final merged components. These components correspond to the detected objects of each frame. Since all blocks are needed for the object detection process, the output is generated after the last block has been processed. At this point of the pipeline, a side output (another Kafka sink) may be added in order to output the bounding boxes and centroids of the objects of each frame. The result of the side output can be seen at Figure 27b The high level functionality of the operator can be seen at Figure 27a.



(a) Illustration of Operator C functionality



(b) Merged components drawn on a video frame

Algorithm 2 Merge ConnectedComponent with Neighboring Components

```
    procedure MergeWithNeighbors(inputCC, blockMap, mergedComponents)
2:      blockID ← inputCC.getBlockID()
        sameBlockCCs ← blockMap[blockID]
4:      aboveBlockCCs ← blockMap[blockID − 1]
        belowBlockCCs ← blockMap[blockID + 1]
6:                                                                  ▷
            ▷ Check the input connected component against the same block components
8:      for neighborCC in sameBlockCCs do
            if inputCC.canMergeWith(neighborCC) then
10:             inputCC.merge(neighborCC)
                sameBlockCCs.remove(neighborCC)
12:         end if
        end for
14:                                                                 ▷
          ▷ Check the input connected component against components of the above block
16:     for neighborCC in aboveBlockCCs do
            if inputCC.canMergeWith(neighborCC) then
18:             inputCC.merge(neighborCC)
                sameBlockCCs.remove(neighborCC)
20:         end if
        end for
22:                                                                 ▷
          ▷ Check the input connected component against components of the below block
24:     for neighborCC in belowBlockCCs do
            if inputCC.canMergeWith(neighborCC) then
26:             inputCC.merge(neighborCC)
                sameBlockCCs.remove(neighborCC)
28:         end if
        end for
30:                                                                 ▷
                    ▷ Check the input component against the already merged components
32:     for alreadyMergedCC in mergedComponents do
            if inputCC.canMergeWith(alreadyMergedCC) then
34:             inputCC.merge(alreadyMergedCC)
                mergedComponents.remove(alreadyMergedCC)
36:         end if
        end for
38:                                                                 ▷
        if a merge has occurred then
40:         mergedComponents.append(inputCC)
        end if
42: end procedure
```

Operator D – Centroid matching/Trajectory extraction

The last operator of the pipeline, operator D, is responsible for creating the trajectory for each object that is present in the video. The inputs are lists containing the object information (i.e. each element of the list represents an object with its centroid and bounding box) for each frame of a specific video. In order to group these lists, the stream is keyed with respect to the *videoID* right before this operator. The algorithm running on operator D iterates over every object in the input list and matches every centroid to the closest existing trajectory, as seen on Figure 29. A trajectory is represented as a string of coordinate pairs, delimited by semicolons in order to be easily retrieved.

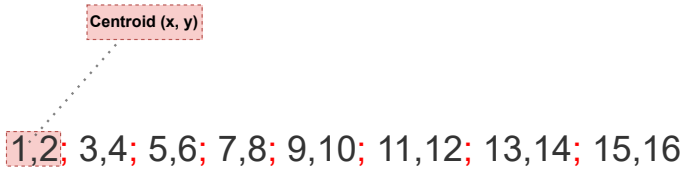


Figure 28: String representation of a trajectory

When the centroids of the first frame that contains objects arrive (some frames may not contain objects), each one is stored in a separate list. Centroids of the following frames try to match an existing trajectory, which consist of centroid from previous frames. If a match has not been found, the centroid is stored in another list, denoting that it belongs to an object not seen until that time.

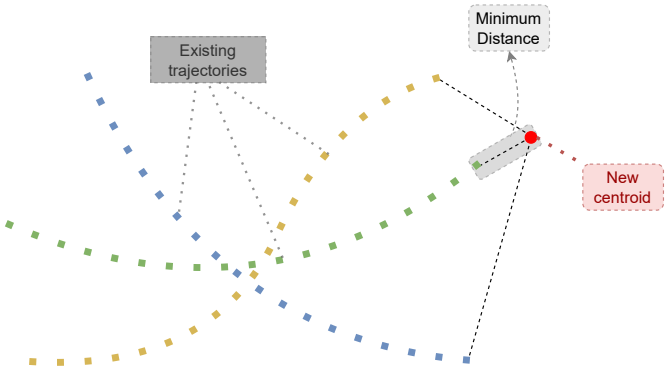


Figure 29: Matching of centroids to existing trajectories

Matching criterion

To establish a match between a centroid and its closest trajectory, it is essential for the Euclidean distance to fall below a specified threshold. In this work, this threshold is set to 50. In general, this threshold is a function of each object's speed. As a result, objects with high speeds may introduce irregularities in the extracted trajectories (e.g. a single trajectory split into two, due to the object's high speed). Ideally, speed, distance between

previous centroids, angle relative to the trajectory and object identity should be taken into consideration.

Frame ordering

In order to build the trajectories, the algorithm needs to take into consideration the order of the frames. For example, assuming that objects are present in all frames of the video, the objects detected in the first frame should initiate the trajectories, while those detected in the last frame should conclude the trajectories. A mechanism for sorting the input object lists according to their frame (much like the sorting mechanism of Operator A, see Algorithm 1), has been established in order to achieve the aforementioned functionality.

Unmatched trajectories

When no suitable match is found for a specific centroid, a new trajectory is created with this centroid as its starting point. Furthermore, in the event that a trajectory remains unaltered for a defined duration and is subsequently matched with a centroid, the centroid is not added to that existing trajectory. Instead, the algorithm initiates a new trajectory for this centroid, treating it as a distinct object. The duration is set to $2s * fps$, which corresponds to 2 seconds regardless of the video's fps. For example, if a video's frame rate is $60\frac{frames}{secs}$, the expression $2s * fps$ is evaluated to 120 frames, which corresponds to 2 seconds of footage. This approach is employed to account for object identity changes (i.e. different objects in the same location at different timestamps) or discontinuities in the video, ensuring that each trajectory corresponds to a distinct object or a continuous movement pattern. This value is chosen arbitrarily and may not produce accurate results for every video. For example, a car stopping at a traffic light will eventually become part of the background due to the background subtraction method. When it eventually starts moving again, our approach will not match its centroid to its previous trajectory, if the duration spent stationary was greater than 2 seconds. If there is prior knowledge of the type of the video to be processed, the user can set the duration accordingly.

Trajectory filtering – Noise reduction

At last, in an effort to further remove any remaining noise, trajectories with less than $2s * fps$ points are not considered part of the operator's result. These kinds of trajectories are considered to have been created out of noise and not actual objects. The selection of the threshold of $2s * fps$ data points is based on the consideration that it corresponds to 2 seconds of on-screen time in a video recorded at any frame rate. This choice ensures that trajectories with sufficient duration and significance are retained for analysis, while shorter trajectories, which might represent brief or inconsequential movements, are filtered out.

Sink operator – Kafka producer

The sink operator consists of a Kafka producer that is responsible for writing the trajectory strings to the output topic of the Kafka broker. Messages arriving to Kafka need

to be serialized and as a result, a serialization schema needs to be provided. This schema is an implementation of a Java interface and dictates how a trajectory string will be transformed into a Kafka record. Every Kafka record representing a trajectory includes a key that identifies the video source from which the trajectory originates. An illustration of the sink operator functionality can be found on the below Figure.

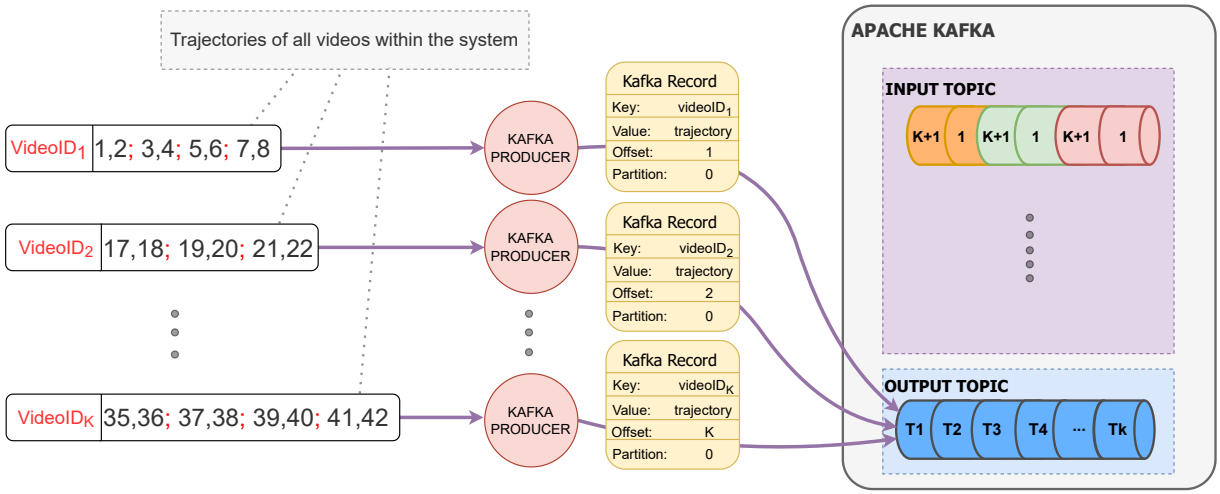


Figure 30: Illustration of the sink's functionality

5 Trajectory extraction results

In this section, we delve into the presentation and discussion of the trajectory extraction results obtained from the execution of several videos. The analysis will not only showcase the results, but also the differences between the background subtraction algorithms, along with the use of less color components.

5.1 Methods

For each video, we will present the results of 4 different approaches. The initial two of approaches differ in the background subtraction method they use. The first uses the Mixture of Gaussians algorithm, and the second uses the KNN method, as elaborated in the Background section. In the second pair of methods, only two out of the three color components for each pixel are utilized, again with the said background subtraction methods. Specifically, the chosen color components are the R (Red) and G (Green) components, normalized by the sum of all three components. The equations for the used color components are given below.

$$R \leftarrow \frac{R}{R+G+B}$$
$$G \leftarrow \frac{G}{R+G+B}$$

The selection of the used components is arbitrary. Experiments with these methods were facilitated in order to observe if there are any differences compared to the full color component range algorithms.

5.2 Videos

In order to test the algorithms, both real-world and synthetic videos from static cameras were used. The synthetic videos have been sourced from the 2012 Background Models Challenge website[5]. Their primary purpose was to showcase the accuracy and efficiency of background subtraction algorithms, yet they also offer valuable footage for trajectory extraction. The real-world videos were sourced from YouTube object tracking footage and the PETS2000[6] dataset (outdoor people and vehicle tracking). The entire video dataset can be found in the following Dropbox folder: Video dataset. This section will present the results of 3 of the videos of the dataset. Screenshots of said videos are presented below.

	Link	Resolution	Description
Video A	https://youtu.be/iqXr5fUN5Sw	480p	Synthetic
Video B	https://youtu.be/w-UpDWlmdE8	576p	Real-world surveillance camera
Video C	https://youtu.be/VeIu1oaBmzE	720p	Real-world highway camera

(a) Video A - Background models challenge

(b) Video B - Sourced from PETS2000 Dataset

(c) Video C - YouTube object tracking footage

5.3 Impact of block size

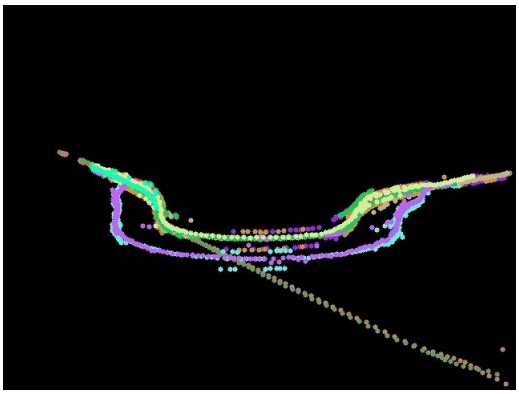
Before we can review the final outcomes for each video, it's essential to assess the influence of the block size on the trajectory results. Splitting the frame into smaller blocks is essential for efficient parallel processing, but may cause distortion and noise in the resulting trajectories. This is due to the nature of the algorithms used in the model. The connected component labeling algorithm utilizes spatial consistency, meaning that information from groups of neighboring pixels are used in order to determine the existence of a connected component. A larger block size results in more accurate connected component labeling, and subsequently more accurate connected component merging. When the block size is too small, many detected connected components in each block may be discarded as noise, due to their negligible areas. Furthermore, very small connected components can cause

[5]http://backgroundmodelschallenge.eu/
[6]https://ftp.cs.reading.ac.uk/pub/PETS2000/

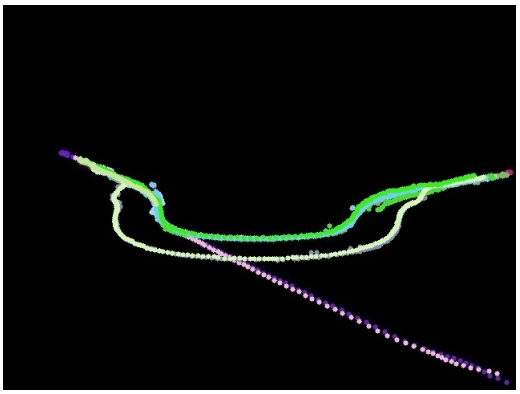
inaccurate connected component merging, since this algorithm relies on simple criteria in order to merge the components into objects. Below are presented the results for Video A, at different block sizes. It should be noted that the specific video contains 11 trajectories, belonging to 11 cars and the background subtraction method that was used, is the Mixture of Gaussians one.

	Block size (rows)	**Block size (KB)**	**Blocks / frame**	**Extracted trajectories**
Video A	1	1.92	480	14
	20	38.4	24	12
	40	76.8	12	11
	96	184.3	5	11

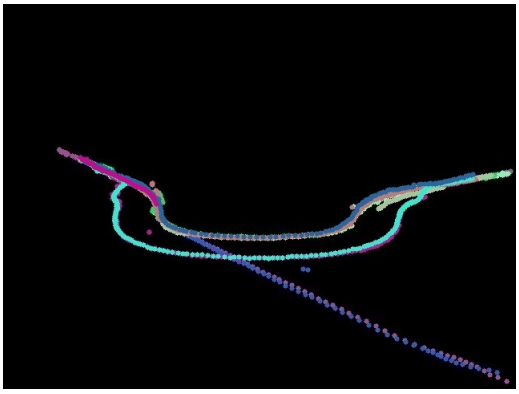
Table 2: Block size information for each video



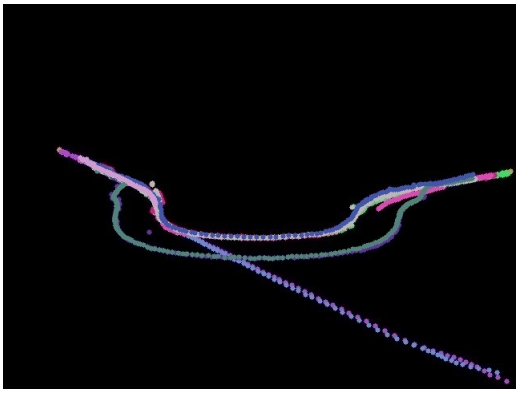
(a) Block size = 1 row



(b) Block size = 20 rows



(c) Block size = 40 rows



(d) Block size = 96 rows

It is evident from the above figures that indeed the results from the execution with Block size = 1 are significantly worse than the executions with the other block sizes. The other executions are quite similar, with the ones successfully extracting the correct number of trajectories being those with block sizes 40 and 96 rows respectively. Increasing the block size more than 96 rows per frame would not further improve the accuracy of the algorithm, but would have a negative impact in the efficient parallelization of the specific video. In

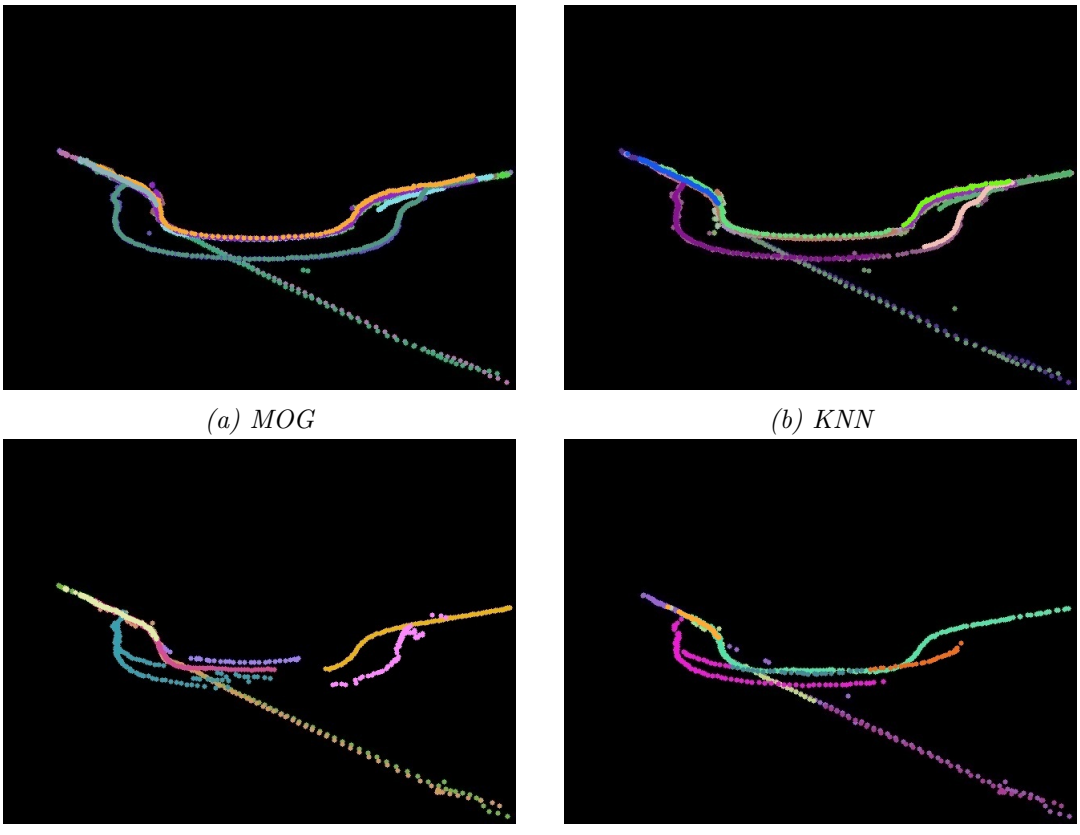
fact, since the block size of 40 rows produces the correct number of trajectories, it is the optimal block size for each video.

It can be safely concluded that, in order to balance between efficient parallelization strategies and accurate results, the optimal block size for each video should be the minimum of the sizes that produce accurate results. Unfortunately, there is no way of knowing the optimal block size before some test executions of each video with different block sizes. One can only deduce that the block size should align with the video resolution. Higher resolutions require larger block sizes to achieve precise results.

5.4 Results

Video A

The results regarding each method for video A are presented at Figure 33 below. The block size was chosen as 40 rows of pixels as mentioned previously.



(a) MOG *(b) KNN*

(c) 2 / 3 MOG *(d) 2 / 3 KNN*

Figure 33: Results for Video A

Analyzing the figures, it becomes apparent that the Mixture of Gaussians method yields the more accurate results of all the tested methods. The exact number of extracted trajectories for each method can be found in the below table.

Method	Correct trajectories	Extracted trajectories
MOG	11	11
KNN		16
2/3 MOG		8
2/3 KNN		9

Table 3: Number of extracted trajectories for each method

The KNN method struggles to maintain a constant accurate segmentation and as a result, many trajectories are split in places, yielding an incorrect number of extracted trajectories. Visually, since in this particular videos many trajectories overlap, the result of KNN is quite similar with the result of MOG. Regarding the 2/3 methods, the missing color component results in a totally inaccurate trajectory extraction. The missing data causes both background subtraction methods to underperform, and subsequently ruin the performance of the entire model. It should be noted that the objects that are not accurately segmented by the background subtraction methods exhibit hues that closely resemble the color of the missing component. The arbitrary choice of the excluded component generates uncertainty as to which objects are going to be incorrectly segmented by the algorithms. For example, eliminating the R (Red) component will result in objects of red hue not being correctly segmented from the background.

Video B

The results regarding each method for video B are presented at Figure 34 below. The block size was chosen as 48 rows of pixels as this is the minimum block size that produces the most accurate trajectory extraction. Once more, the MOG method seems to be the superior one, yielding the most accurate results. This video contains 6 trajectories that should have been detected. The exact number of trajectories that each method extracted can be seen at Table 4.

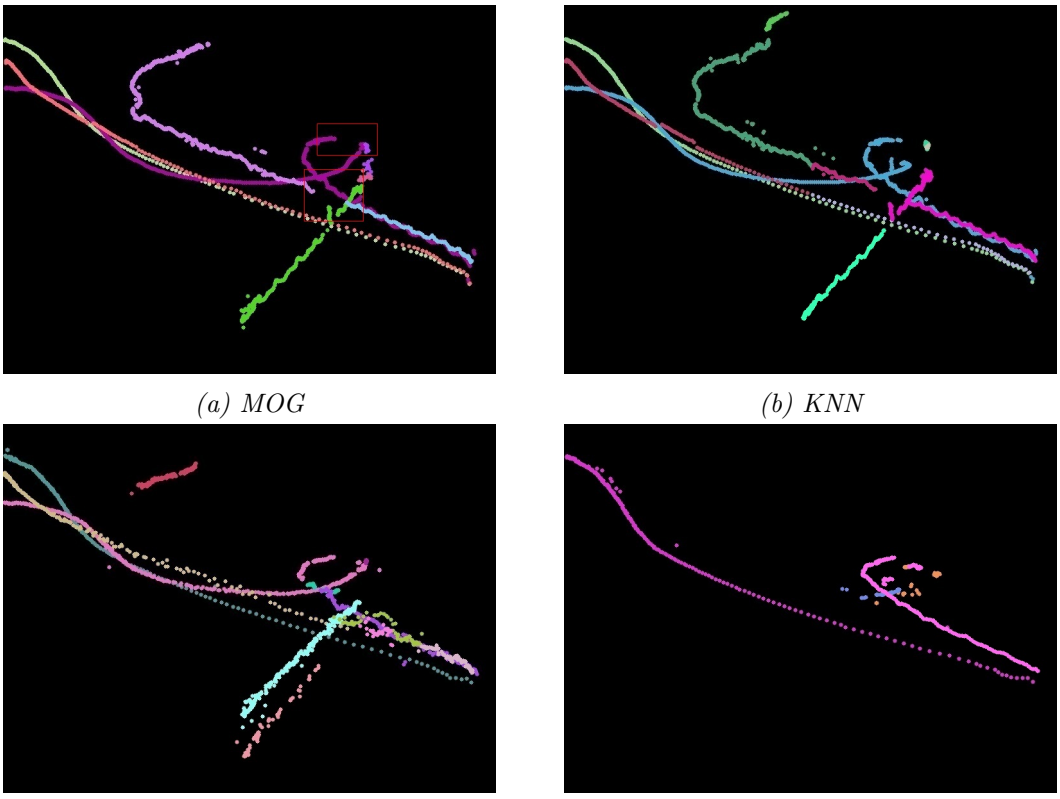
Method	Correct trajectories	Extracted trajectories
MOG	6	8
KNN		11
2/3 MOG		12
2/3 KNN		4

Table 4: Number of extracted trajectories for each method

Directing our focus to the MOG method, which has a superior performance compared to the other methods, we notice several interesting details. Foremost among these, is a significant gap in the *purple* trajectory, primarily attributed to the thresholds established by the centroid matching algorithm. In Video B, the purple trajectory includes both a car and a human. The car approaches a parking spot and eventually parks. After a brief

period, the driver exits the car and begins walking away from it. It's important to note that once the car is parked, it merges into the background, causing its centroid to vanish.

Normally, two distinct trajectories should have been extracted. However, the second object (the person leaving their car) matches the existing trajectory of the car, due to the Euclidean distance falling below the predefined threshold and the duration being less than 2 seconds (criteria specified in operator D). Additionally, it becomes apparent that certain gaps exist between two trajectories that ideally should have formed a singular trajectory, as seen in the pink and light blue trajectories. In this segment of the video, the two pedestrians' paths overlap (pink and green paths). Due to the merging criteria, the connected component merging algorithm merged the two pedestrians into one object and matched it to the green trajectory. This results in the separation of what would have been a continuous trajectory.



(a) MOG *(b) KNN*

(c) 2/3 MOG *(d) 2/3 KNN*

Figure 34: Results for Video B

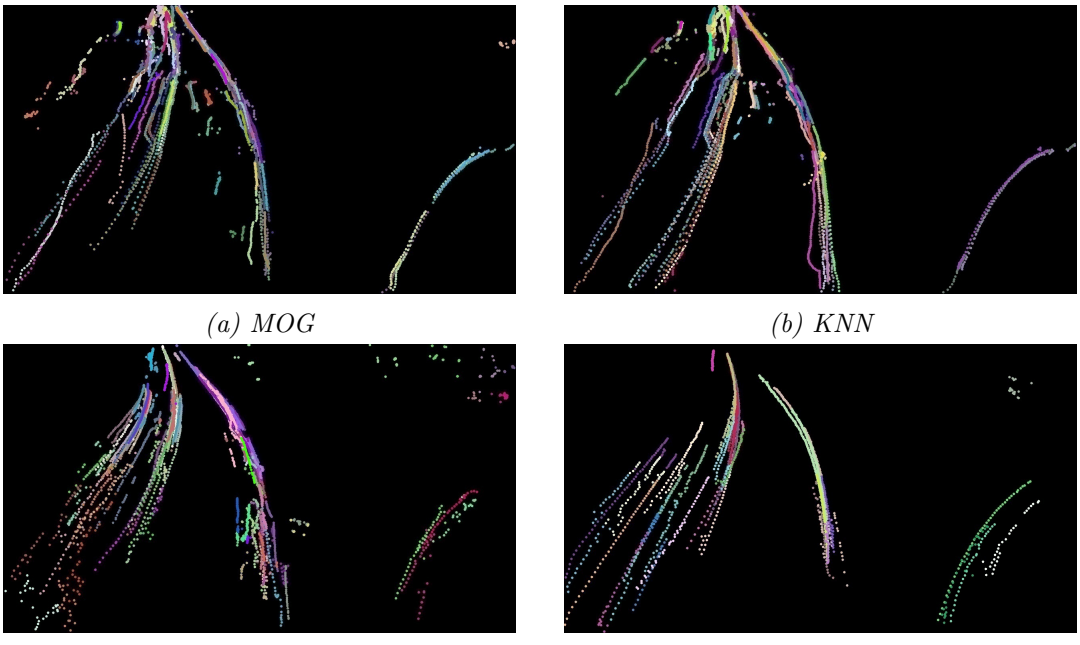
Much like the previous video, the KNN method provides visually similar results with MOG, except for splitting some trajectories due to not so accurate background-foreground segmentation. The 2/3 methods fail to provide accurate results once more, due to the missing color component, with the 2/3 KNN method detecting only two complete trajectories, along with noise.

Video C

The results regarding each method for video C are presented at Figure 35 below. The block size was chosen as 45 rows. This video is quite different that the previous two. There are a lot of vehicles, running at high speeds in the highway. In fact, there are 53 cars, motorcycles and trucks appearing in this 1-minute video. There are also many trajectory and object overlaps. It is important to note that, due to these aspects of the specific video, a larger block size did not make much of a difference in the final results. Hence, a relatively small block size is chosen. As we can see, all methods provide an adequate visual result, but lack in accuracy. Table 5 below presents the exact number of trajectories extracted by each method.

Method	Correct trajectories	Extracted trajectories
MOG	53	101
KNN		86
2/3 MOG		77
2/3 KNN		29

Table 5: Number of extracted trajectories for each method



(a) MOG *(b) KNN*

(c) 2/3 MOG *(d) 2/3 KNN*

Figure 35: Results for Video C

In this video, the 2/3 methods do not underperform as much as in the previous ones. This is mainly due to the colors of the vehicles appearing in the video, which are mostly gray

tones. This particular video is a great example to test the computational ability of the system, because of the amount of foreground it contains.

5.5 Conclusions

In light of the results of the trajectory extraction experiments, three conclusions have been drawn. Firstly, the MOG (Mixture of Gaussians) method, which has demonstrated superior performance compared to alternative methods, will be employed as the primary methodology for background subtraction. We choose the MOG method for its swift adaptability, and being more accurate on average than the KNN method. Secondly, the block size for each video employed in these experiments, will be adjusted to the minimum value that produces accurate results. As noted previously, there may be videos for which, the block size does not make a difference in the final outcome, since the algorithms cannot provide better accuracy when using a larger block of pixels. Last but not least, the video that will be used in the following performance evaluation experiments will be Video C, in many resolutions. These strategic choices lay the foundation for our forthcoming experiments, where we aim to delve deeper into the system's performance and capabilities.

6 Performance evaluation

In this chapter, we will delve into the performance of the system in question. This includes identification the bottlenecks of the system, a comparison to an identical but monolith application and an overall performance review.

6.1 Infrastructure

The experimental setup that hosts the entire system (i.e. clients, Kafka, Flink) consists of a Kubernetes cluster with two node pools. A node pool is a group of nodes within a cluster that all have the same configuration. The initial node pool is allocated for running the clients (applications that stream the video), the Kafka broker and the Flink Kubernetes operator. The second node pool is dedicated to Flink's cluster. Opting for this two node pool setup, as opposed to a single virtual machine outside K8s for Kafka, and a K8s cluster for Flink, offers several advantages. This approach provides easier installation of all the necessary components using Helm[7] charts (community-made charts for Kafka and K8s operator), simplified network configurations, and enhanced portability across different cloud providers.

Kafka node pool

The node pool dedicated to the Kafka broker, the clients and the Flink K8s operator consists of a single, powerful virtual machine. This virtual machine (VM) has 8 virtual CPUs (vCPUs) with a base frequency of 3.1GHz and a turbo frequency of 3.8 GHz, with 32GB of RAM. In order to achieve high performance I/O operations, it is equipped with a 200GB persistent SSD disk, which provides up to 1200MB/s read and write throughput per instance. The operating system is the Container-Optimized OS[8] that Google Cloud Platform provides. This instance hosts multiple client application and the Kafka broker with two topics and multiple partitions, configured based on the desired parallelism, as well as the Flink K8s operator. Both client and Kafka processes require great I/O performance and multiple cores/threads, to accommodate for the parallel needs of the application. Hence the deployment of 8 CPUs and the dedicated SSD attached to this instance.

Flink node pool

The node pool responsible for hosting and running the Flink cluster contains up to 7 nodes. Experiments 1 and 3 use nodes with 2 vCPUs (2.0/2.8GHz Base/Turbo Frequency), 8GB of RAM and 30GB of storage[9]. Much like the instance of the Kafka node pool, each node runs the Container-Optimized OS, which is provided by Google. Experiment 2, which requires greater processing capabilities than the others, uses nodes with 2 vCPUs (3.1/3.9GHz Base/Turbo Frequency), 16GB of RAM and 30GB of storage[10]. In all cases, each TaskManager's resource needs are configured in a manner where each node is limited

[7] https://helm.sh/
[8] https://cloud.google.com/container-optimized-os/docs

to hosting only one TaskManager instance. TaskManagers are allowed to have one slot each. The spawning of necessary TaskManagers - slot allocation, is handled entirely by the Flink Kubernetes Operator, according to the configuration of the YAML file provided by the user. Due to the TaskManagers only having one slot, each slot holds an entire pipeline. An overview of the Flink cluster on top of the K8s cluster can be seen at Figure 36.

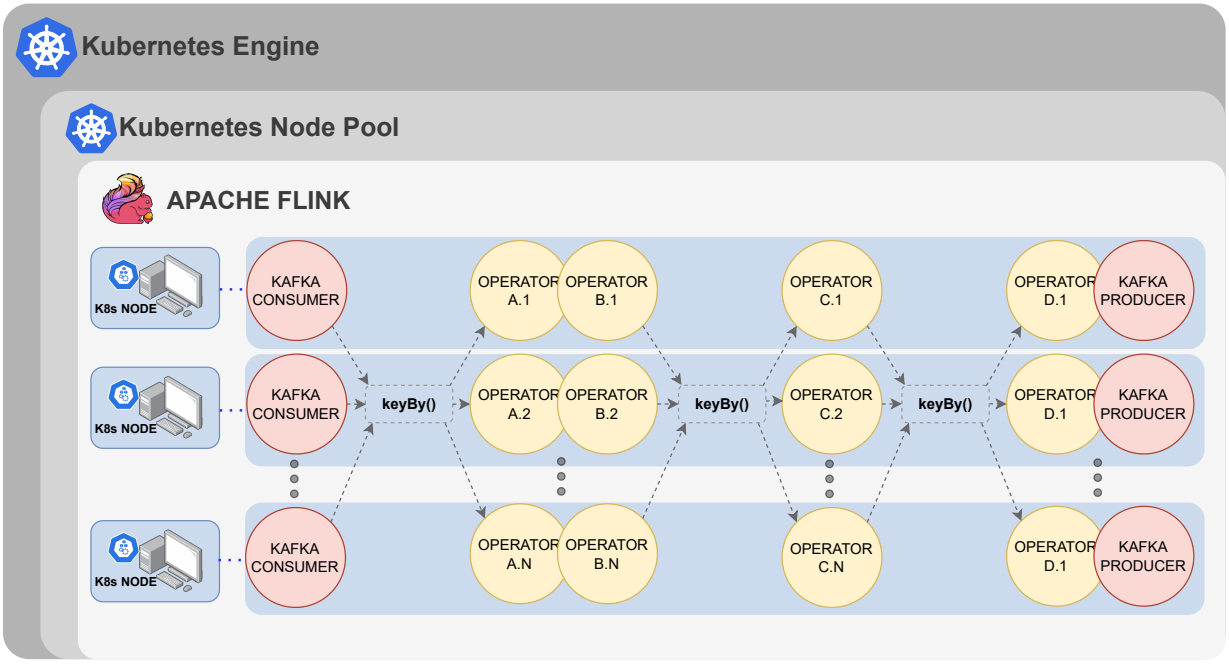


Figure 36: An illustration of the Flink cluster on top of the K8s cluster. Each node hosts a single task manager, which executes an entire Flink pipeline.

Note: *The compute-optimized infrastructure costs approximately* ***$1300*** *per month when using GCP. That is about* ***$1.80*** *per hour of system usage. When using the general-purpose infrastructure, the cost is reduced to* ***$805*** *per month, or* ***$1.10*** *per hour.*

Videos

In order to test the performance of the system, we will use only one video, in multiple resolutions. That way, the amount of foreground information remains constant across the videos. The amount of foreground in a video is a key factor of the speed at which the video is going to be processed. The video that will be used is Video C, as mentioned in the previous chapter. The block size for each resolution will be the minimum so that the results are accurate, according to the findings of the previous chapter. As mentioned, the minimum block size is determined empirically, by testing different block sizes for each video, and selecting the smallest one that produces accurate results.

[9]Machine type: e2-standard-2 - "Cost Optimized"
[10]Machine type: c2-standard-4 - "Compute Optimized"

	Link	Resolution	Raw RGB size (GB)	Block size (rows)	Block size (KB)	Blocks / Frame
Video C	https://youtu.be/VeIu1oaBmzE	480p	1.38	40	76.8	12
		576p	1.92	48	110.6	12
		720p	4.14	45	172.8	16
		1080p	9.33	54	311.0	20
		1440p	16.58	72	553.0	20

Table 6: Info of the video that is going to be used in the experiments

The background subtraction algorithm is chosen to be the Mixture of Gaussians method. Its parameters (e.g α, T) are the OpenCV's defaults. All videos are used in raw RGB888 (Reg, Green, Blue colors, 1 byte each) format. The below table presents all the necessary information about the said videos.

6.2 Experiment 1: Comparison to a monolith system

Purpose

The focus of the following experimental process is to determine whether a Flink cluster can outperform a monolith implementation of the system, with the same resources. Both Flink and the VM instance running the monolith system, will be given the same number of CPUs and GBs of RAM. This approach will enable a direct comparison, which will allow us to observe the efficiency that a distributed system offers, in contrast to the monolith implementation, when operating under similar resource constraints.

Procedure

The monolith implementation of the system is a single Python script running OpenCV. It contains the entire algorithm which was implemented on Flink (i.e Background subtraction → Connected component labeling → Centroid matching), but in a sequential manner. The script processes entire frames, one after the other. It is deployed as a Docker container, in a single VM instance with 8 CPUs, and 32GBs of RAM. The resources of this VM instance are equal to the total resources of the Flink cluster. ***The Flink cluster*** is configured with a parallelism value of 7, since this is the maximum parallelism allowed by the resource constraints. Table 7 below shows the exact resource allocation of the Flink cluster's components.

Flink component	Number of units	CPUs/unit	RAM/unit (GB)
TaskManager	7	1	4
JobManager	1	1	4
Total Resources	8 units	8 CPUs	32GB RAM

Table 7: Flink components resource allocation

Since the monolith implementation can only process one video at a time, the metric that is going to be studied in this experiment is `single video latency`, or else referred to as `single video response time`. For the experimental process, each video will run through both system **5** times in a row and the average response time for each system will be calculated. The results will present the average response time/single video latency for each video. For the monolith implementation the response time value pertains to the duration from the instance when the user starts the program until the point at which the results are generated. For the Flink application, the response time value refers to the time elapsed from when the user initiates the client program until the moment when the results are written to the output Kafka topic. An illustration of the response time metric in the Flink application is given in the below Figure.

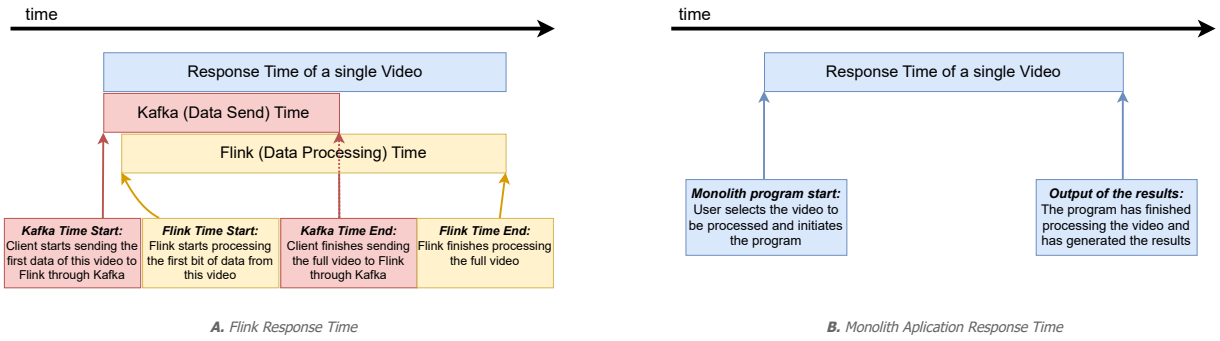


Figure 37: Comparison of response time values between the two systems in question

Results

The results of the experiment are presented in the table and charts below. The first column of the table is the resolution of each video, the second column of the table is the monolith system's response time and the third column is Flink's response time. As mentioned before, the response times of each implementation are the average of 5 video executions.

Resolution	Monolith response time (s)	Flink response time (s)
480p	8.66	46.39
576p	9.93	50.63
720p	161.78	59.58
1080p	475.10	116.74
1440p	1.413.13	245.35

Table 8: Response times of the two implementations

The two gray rows of the table denote the resolutions in which the monolith implementation outperforms the Flink application. The below chart representation of the results provides a more informative and direct comparison. The video resolution results are split between two charts in order to better visualize the differences in each order of magnitude.

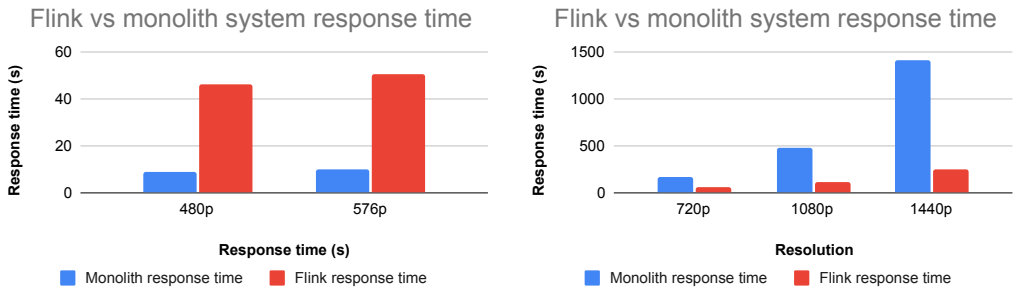


Figure 38: Response time comparison between the two systems

Analysis

Both the tabular data and the charts, notably show, that for resolutions below 720p, the monolith system exhibits quite superior performance than the Flink application. Video files with greater size, such as the ones originating from resolutions above 720p, are processed much more rapidly in the Flink application than the monolith system. These results can be attributed to several factors.

Reduced Overhead: When dealing with video files of a smaller scale (i.e. videos of lower resolutions → lower input throughput), the inherent overhead within Flink's processing environment has a significant impact on the performance of the processing. Network intensive operations such as **keyBy()** functions, along with the necessity for serialization and deserialization of the data at each operator, introduce an overhead that is absent in a monolithic environment.

Parallelization efficiency: Frames of lower resolutions along with the necessary block size for accurate results, generate a limited set of distinct keys. It is important to recall that the distinct keys of each frame correspond to the number of the blocks within that frame. For instance, in a 480p frame with a block size of 40 rows, there are 12 blocks, translating to 12 unique keys per frame. When Flink executes the **keyBy()** functions, it organizes these keys into groups, and then distributes each group to a different parallel instance (subtask) of an operator. A low number of keys leads to a low number of groups. As a result, there may be parallel instances of operators that remain completely idle, as no key group has been assigned for processing. In contrast, higher resolution videos yield a greater number of keys and groups, resulting in improved resource utilization and operational efficiency. A skewd KeyGroup distribution can be seen at Figure 39. This is the case for low resolution videos.

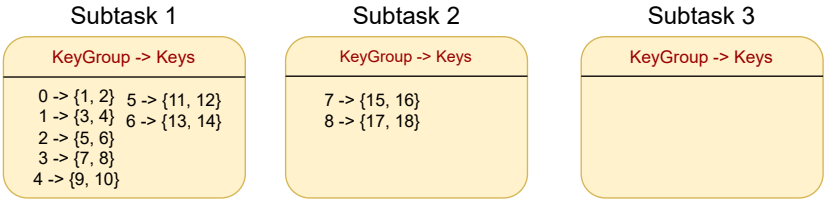


Figure 39: Illustration of a skewed KeyGroup distribution (18 keys)

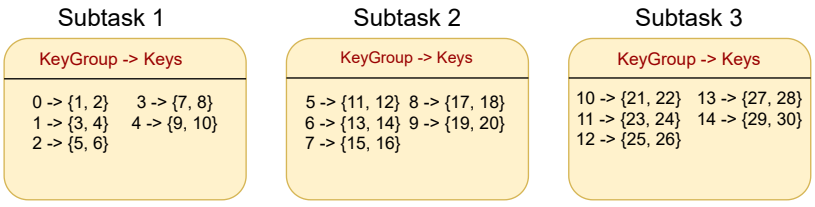


Figure 40: Illustration of a uniform KeyGroup distribution (26 keys)

In contrast, a uniform KeyGroup distribution, such as the one seen at Figure 40 can utilize the underlying resources much more efficiently. The grouping of keys into key groups and

the distribution of the latter are handled internally by Flink and depend on many values such as the hash value of the key, parallelism and maxParallelism (number of total key groups). The addressing of the issues caused by the skewed key distributions as well as Flink's internal mechanism will not be further investigated as part of this Thesis and will be deferred as future work.

Data locality and caching mechanisms: The monolith system, running in a single VM and processing one video at a time, can efficiently utilize data locality and CPU caching mechanisms. Having all the data in the same place (same machine, in memory) can provide substantial reduction in data transfer overhead, by storing and retrieving frequently used data from the local memory. Moreover, caching a subset of the data will result in an even further reduction in data transfer times, since it eliminates the need for RAM accesses, given that the CPUs fetch the data from caches with far superior I/O performances. When dealing with lower-resolution videos, which typically involve smaller data sizes (and therefore a higher percentage of the dataset is cached), these mechanisms prove particularly advantageous.

When processing videos of high resolutions, one can safely conclude that the distributed system is a much more suitable solution. It provides better response times along with the ability to process multiple videos at the same time. Figure 41 shows that Flink can be up to almost 6x faster than the monolith approach. The speedup is defined as the ratio of the `Monolith response time` to `Flink response time` values, as seen at Table 8.

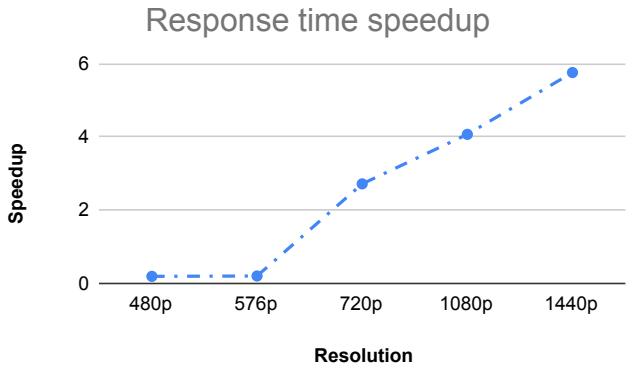


Figure 41: Response time speedup when compared to the monolith system

Should someone need to process multiple videos using the monolith application, they would need to deploy multiple instances of it, along with a custom load balancer to correctly distribute the video traffic. Furthermore, Flink provides the necessary mechanisms for scaling depending on the workload. Flink's ability to scale will be further investigated in the next experiment.

6.3 Experiment 2: Scaling and maximum throughput evaluation

Purpose

The goal of the second experiment is twofold: Firstly, it seeks to determine whether the video processing application running on Flink can scale, as Flink's parallelism increases. This scalability aspect holds significant importance, given that Flink applications are predominantly intended for deployment in large-scale scenarios. Secondly, this experiment aims to explore the boundaries of the application within a designated infrastructure. That is, the upper bounds of its throughput capabilities.

Procedure

The procedure that is going to be repeated for every video resolution is the following: Setup the number of Kafka input topic partitions according to the parallelism value of Flink. For instance, if the parallelism is set to 3, there are going to be 3 input topic partitions. Setup a number of clients that are going to send data to Kafka concurrently. For this experiment, two parallel clients are needed, each sending two videos. The number of clients can scale according to needs and each client can send an arbitrary number of videos. An illustration of the process is given at Figure 42. The data is processed by Flink, and the total processing time is calculated. For each resolution there will be four iterations of the above procedure, one for each parallelism value. The parallelism values will be 1, 3, 5, 7. All the previously mentioned videos will be used for this experiment too. Each video's characteristics, along with some useful values can be seen at Table 9 below.

Resolution	Width x Height	Bitrate (MB/s) (1)	Parallel clients(2)	Times sent per client (3)	Total throughput sent (MB/s) (4)
480p	640 × 480	27.6	2	2	55.2
576p	768 × 576	39.8	2	2	79.6
720p	1280 × 720	82.9	2	2	165.8
1080p	1920 × 1080	186.6	2	2	373.2
1440p	2560 × 1440	331.5	2	2	663.0

Table 9: Video characteristics

Bitrate(1) represents the rate at which bits are transmitted, for a specific video resolution and frame rate. The Bitrate formula for RGB888 (3 color channels, 1 byte each) videos can be expressed in bytes per second as follows: Let W be the video's width, H be the video's height and fps the video's frame rate. The fps value for each resolution is 30.

$$Bitrate = H \times W \times 3 \times fps$$

Parallel clients(2), shows how many clients were used concurrently. Each client reads the same video file and sends it to Kafka. They attach different keys to each video (video IDs), so they can later be differentiated from Flink. The reason we use multiple parallel clients is to increase the total workload the Flink will have to handle. This is because with each additional client, the total required bitrate (Total MB/s sent) increases linearly.

Times sent per client$^{(3)}$, is the total number of times the video is sent (sequentially) by each client in every iteration. Each time a different video ID is generated by the client. When a client finishes sending a video to Kafka, it will start sending it again with a different ID. The only reason this is done to simulate a more realistic scenario for Flink where data keeps coming in through Kafka (i.e., higher workload) and where Clients send multiple videos to be processed.

Total throughput sent$^{(4)}$, is the sum of the bitrates of all the videos being sent in parallel. For example, if two clients use as input a 1920x1080 resolution video, each one sends data to Kafka at a bit rate of 186.6MB/s (single video). Therefore, the total input throughput for 2 videos in parallel will be $2 \times 186.6MB/s = 373.2MB/s$

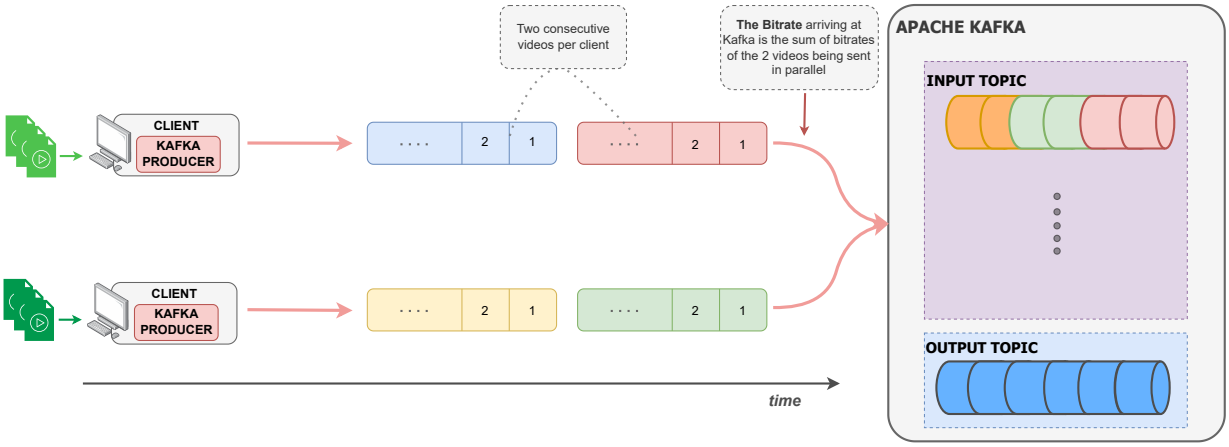


Figure 42: Illustration of the experimental process. Two clients send two consecutive videos each. The Bitrate arriving at Kafka is the sum of the bitrates of the two parallel videos.

Results

The results of the experimental process are presented at Table 10. As previously described, for each video resolution, four tests were performed, each for a value of parallelism (i.e. 1, 3, 5, 7). For each value of parallelism, there are three columns of result data.

Total GB processed$^{(1)}$ is the total size of the videos in each iteration. Since there are 4 videos, this value is 4 times the size of each video. This number denotes the amount of data that Flink processed.

Flink processing time$^{(2)}$ is the time Flink needed to process all 4 videos and produce the results. This duration is calculated by logging the exact time Flink started processing the first video and the exact time it output the trajectories of the last video. In this duration, Flink processed *Total GB processed* of data.

Output throughput$^{(3)}$ is the amount of data Flink processed in a 1-second duration. It is the division of the previous two values. It represents the throughput of Flink processing.

Resolution	Parallel clients	Input throughput (MB/s)	Parallelism	Total GB processed	Processing time (s)	Output throughput (MB/s)
480p	2	55.2	1	5.52	380.48	**14.50**
			3	5.52	202.36	**27.28**
			5	5.52	166.28	**33.19**
			7	5.52	145.56	**37.92**
576p	2	79.6	1	7.68	459.67	**16.71**
			3	7.68	234.57	**32.74**
			5	7.68	183.65	**41.81**
			7	7.68	164.93	**46.56**
720p	2	165.8	1	16.56	615.51	**26.90**
			3	16.56	392.72	**42.16**
			5	16.56	270.6	**61.20**
			7	16.56	210.92	**78.51**
1080p	2	373.2	1	37.32	1344.56	**27.96**
			3	37.32	768.48	**48.56**
			5	37.32	643.56	**57.99**
			7	37.32	420.36	**88.78**
1440p	2	663.0	1	66.32	2886.08	**22.97**
			3	66.32	2343.48	**28.30**
			5	66.32	1313.32	**50.49**
			7	66.32	937,4	**70.75**

Table 10: Results of the current experiment. Output throughput is calculated for every resolution and every parallelism value.

Directing our focus to the *Output throughput* metric of Table 10, we observe that it cannot match the *Input throughput* of the system in any case. This observation underscores a critical concern: *The current configuration of the system is not operationally sustainable.* The inability of the Flink application to match the input throughput will result in a vast accumulation of data in the Kafka partitions. Consequently, this buildup poses a significant risk, since the storage capacity of the Kafka machines is finite, and is only a matter of time before it reaches its limits.

In order to address this issue, we will replicate the experiment, but this time, the more powerful machines, mentioned in the Infrastructure subsection [6.1], will be used. We anticipate that the additional 1Ghz of processing power provided by these machines will have a significant impact in the overall performance of the system, and will render the system not only viable, but capable of meeting the demands of real-time processing.

The results of the replicated experiment can be seen at Table 11 below. This time the results are much more optimistic. The first thing that we notice is that the system can match the *Input throughput* for 2 concurrent videos up to 720p resolutions. Furthermore, we observe that, as the parallelism increases, the *Output throughput* metric also exhibits an increase, in all the cases.

Resolution	Parallel clients	Input throughput (MB/s)	Parallelism	Total GB processed	Processing time (s)	Output throughput (MB/s)
480p	2	46.0	1	5.52	174.77	**31.58**
			3	5.52	70.59	**78.19**
			5	5.52	58.82	**93.84**
			7	5.52	50.73	**108.81**
576p	2	66.0	1	7.68	208.92	**36.76**
			3	7.68	91.91	**83.56**
			5	7.68	64.72	**118.66**
			7	7.68	55.60	**138.12**
720p	2	165.8	1	16.56	386.46	**42.85**
			3	16.56	148.27	**111.68**
			5	16.56	125.78	**131.65**
			7	16.56	85.80	**193.00**
1080p	2	373.2	1	37.32	800.34	**46.63**
			3	37.32	373.68	**99.87**
			5	37.32	267.62	**139.37**
			7	37.32	172.10	**216.85**
1440p	2	663.0	1	66.32	1279.81	**51.82**
			3	66.32	633.36	**104.71**
			5	66.32	460.30	**144.07**
			7	66.32	262.38	**252.76**

Table 11: Results of the current experiment. Output throughput is calculated for every resolution and every parallelism value.

This increase is more evident in the higher resolution videos (i.e. 720p and greater). In order to further investigate this behavior, we will calculate the percentage increase of the *Output throughput* as the parallelism increases from 1 to 7 and from 5 to 7. The video suite is divided into two categories: Low resolutions contain 480p and 576p videos, while high resolutions contain the 720p, 1080p and 1440p videos.

Category	Resolution	Percentage increase from 1 to 7 (%)	Avg. percentage increase from 1 to 7 / category (%)	Percentage increase from 5 to 7 (%)	Avg. percentage increase from 5 to 7 / category (%)
Low resolution	480p	244.55	**260.14**	15.95	**16.18**
	576p	275.73		16.40	
High resolution	720p	350.40	**367.74**	46.60	**59.21**
	1080p	365.04		55.60	
	1440p	387.76		75.44	

Table 12: Percentage increase in output throughput per parallelism increase

Looking at the output throughput gain due to the increase of the parallelism from 5 to 7, it can be deduced that higher values of parallelism will mostly affect the high resolution videos in out experiment. Increasing the parallelism from 5 to 7 results in a 16.18% average output throughput gain in the low resolution videos as opposed to the 59.21% gain in the high resolution videos. This can be attributed to the number of keys generated from 4 consecutive videos of each resolution. The number of total keys for low resolution videos will be much lower than the respective number for high resolution videos. As mentioned in the previous experiment, the key distributions generated by each video are heavily skewed. Data skew refers to an unbalanced distribution of data in the pipeline, and subsequently, non-optimal utilization of the parallelism of the system. This is the case for the 480p

and 576p resolutions, when the parallelism increases from 5 to 7. This increase has little effect in the performance for these resolutions, since the extra 2 parallel instances for each operator remain almost idle.

Should we define as *naive approach* the parallelism value of 1 in the Flink application, the speedup gained from increased parallelism can be seen at Figure 43 below. The speedup is defined as:

$$speedup = \frac{Output\ \ throughput\ \ at\ \ paralellism = 1}{Output\ \ throughput\ \ at\ \ paralellism = 7}$$

Here, we can distinctly visualize the lesser impact of the increasing parallelism on resolutions 480p and 576p. Further increase in these resolutions will have little or no impact at all, since new instances will receive only a small amount of data to process. On the other hand, further increase of parallelism in higher resolutions will most likely result in a similar trend in speedup as the below chart. This behavior will not continue indefinitely. When the parallelism value becomes high enough that keys are not distributed to new instances, the trendline will become sublinear, as seen on the diagram for the low resolutions.

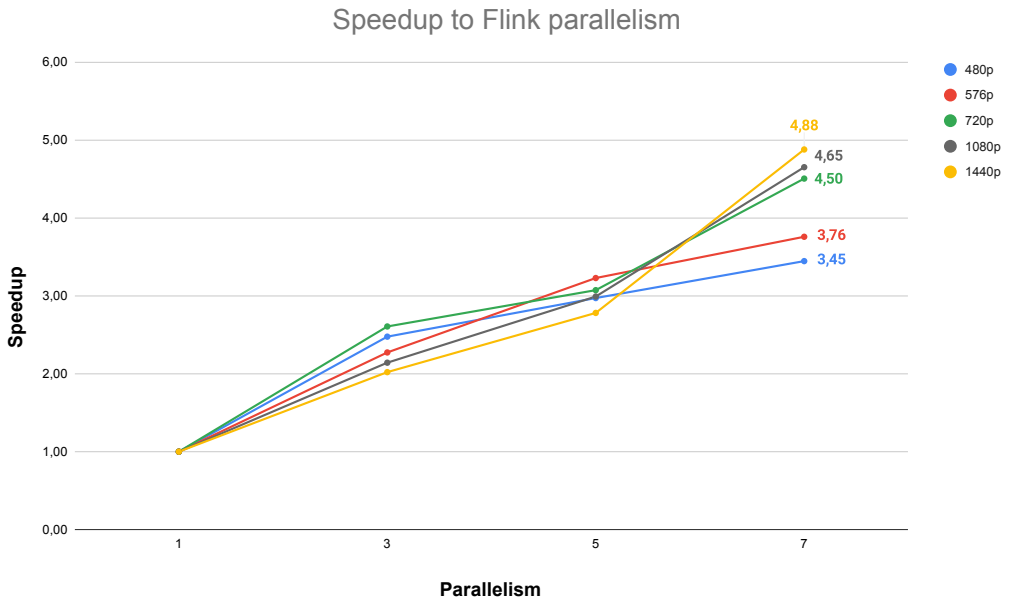


Figure 43: Speedup gained as the parallelism increases

Overall, it can be observed that the speedup increases with the increase of parallelism, reaching a maximum of 4.9x for 1440p resolution at a parallelism of 7. It should be noted that, if Kafka was facing difficulties providing Flink with data, increasing the parallelism would yield no increase in the *Output throughput* of the system.

All the above results, and especially those of Table 11, yield the following questions: What is the maximum throughput of the Flink application? What is the maximum *Input throughput* that the system can match? In order to address these questions, we will conduct a supplementary experiment. The same infrastructure will be used (the more powerful, "Compute Optimized" machines) and the system will operate with the **parallelism** value set to **7**. First, a list of increasing input throughput values will be created as a combination of input videos. Each video is streamed by a separate client. Then, the response for each video combination will be measured, and the output throughput of the system will be calculated. The client setup is the same as the previous procedure, with the difference that there can be more than 2 clients in order to accommodate for more concurrent videos. For instance, an *Input Throughput* of 309.77 MB/s requires 3 clients, as shown in the table below. Each client streams the video twice, in order to simulate a longer data flow into the system. The video combinations that create each input throughput value can be found at Table 13.

Resolution	480p	576p	720p	1080p	1440p	**Input Throughput (MB/s)**
Bitrate	27.64	39.81	82.94	186.62	331.77	
		✓				39.81
	✓	✓				67.45
			✓			82.94
	✓		✓			110.58
				✓		186.62
		✓		✓		226.43
			✓	✓		269.56
		✓	✓	✓		309.37
					✓	331.77

Table 13: Video combination for each input throughput value

The resulting response time and output throughput of the system for each video combination can be seen at Table 14. Each row of the table shows the GB processed, response time and *Output throughput* for each case of *Input throughput* A more intuitive and visual interpretation of the results can be found at Figure 44. In this illustration, in cases where the output throughput curve surpasses the input throughput curve, the system can handle the data in real time. Conversely, in cases where the opposite occurs, the system is unable to do so.

Case	Input Throughtput (MB/s)	Total GB Processed (GB)	Response Time (s)	Output throughput (MB/s)
1	**39.81**	3.84	35.4	**108.47**
2	**67.45**	6.60	50.25	**131.34**
3	**82.94**	8.28	55.32	**149.67**
4	**110.98**	11.04	69.34	**159.21**
5	**186.62**	18.66	91.26	**204.47**
6	**226.43**	22.5	99.15	**226.92**
7	**269.56**	26.94	115.65	**232.94**
8	**309.37**	30.78	130.24	**236.77**
9	**331.77**	33.16	140.14	**236.62**

Table 14: Resulting throughput for each video combination

The data shows that the Flink application in the current infrastructure, can keep process up to $226MB/s$ in real time. Every input throughput up to $226.43MB/s$ is surpassed by the system's output throughput. For greater values of input throughput, we can see that the output throughput increases, but not quite enough in order to process the incoming data in real time. In this case, both increasing the parallelism and the computational

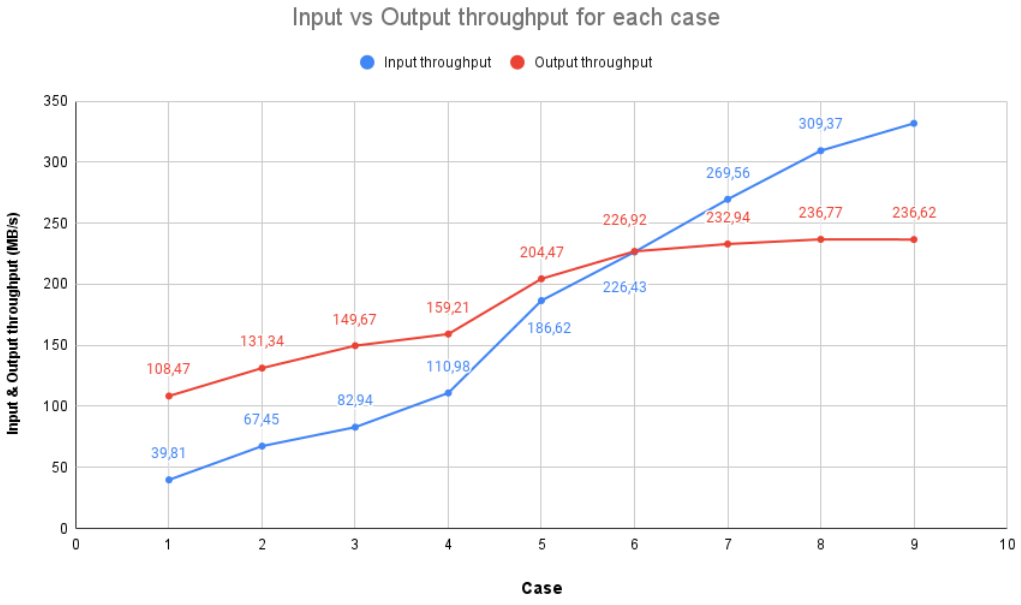


Figure 44: Input and output throughput curves. Parallelism = 7

power of the machines will help with matching higher values of *Input Throughput*. It is important to note that increasing parallelism will enhance the system's performance up to the point where no key groups or only a minimal number of key groups are assigned to each new parallel instance. Beyond that point, performance will plateau.

6.4 Experiment 3: Identifying the bottlenecks of the system

Purpose

The primary objective of the current experiment is to identify potential bottlenecks of the system. To elaborate further, this experiment will determine whether Kafka negatively affects the total performance of the system, or if Flink struggles to process the video, impeding the entire system's performance. Due to the computational complexity in Flink's video processing pipeline, we anticipate that it will not be able to keep up with Kafka. In this experiment, the less powerful infrastructure was used. Similar results were observed when using the more powerful infrastructure too.

Procedure

The procedure for this experiment closely resembles the procedure for the first one. Each video of Table 6 will be given as an input to the system **5** times. Two timestamps are important: The first timestamp marks the completion of video transmission from the client to Kafka. This will be referred to as the `Kafka send time`. The second timestamp denotes the duration between when the client starts sending the video and when Flink outputs the results. This is the `Response time` for a video. `Flink-to-Kafka Lag` is defined as the difference between `Response time` and `Kafka send time` values. `Flink-to-Kafka Lag` approaching 0, is an indication of Kafka being the bottleneck of the system, since Flink immediately consumes everything that is sent to Kafka. An illustration of the sending and processing timeline can be found at Figure 37.

Results

In Table 15, the results of the experiment are presented. The first column shows the resolution for each input video, second column shows the `Kafka send time` value, third column shows the `Response time` value, and the last column shows their difference, the `Flink-to-Kafka Lag` value.

Resolution	Kafka Send Time	Response Time	Flink-to-Kafka Lag
480p	12.58	46.39	33.81
576p	14.33	50.63	36.3
720p	22.88	59.58	36.7
1080p	47.65	116.74	69.09
1440p	59.2	245.35	186.15

Table 15: Numeric results for the bottleneck identification experiment

Figure 45 visually depicts the relationship between video resolutions and their corresponding `Flink-to-Kafka lag` value. Each data point represents a specific resolution, and the associated value is plotted on the y-axis.

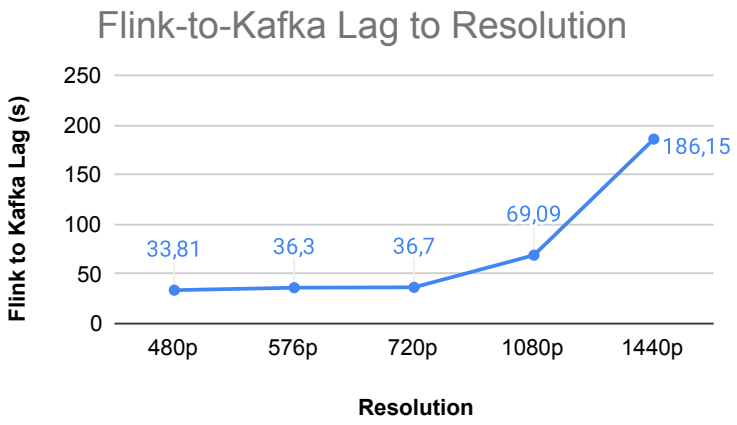


Figure 45: Flink-to-Kafka lag vs resolution chart

Discussion

The above chart provides valuable insights in order to determine whether Kafka causes a bottleneck in the system. The `Flink-to-Kafka lag` value appears to be monotonically increasing as the video resolution gets progressively higher. This is a clear indication that Kafka is not the bottleneck. It is obvious that in every resolution, the video is sent to Kafka well before it is processed by Flink. Greater `Flink-to-Kafka lag` values for higher resolutions, show that Flink encounters difficulties in keeping up with the rapid data transmission rates of the clients to Kafka. At this point, it is safe to conclude that the bottleneck of the system resides in Flink's video processing pipeline

Further investigation on each individual task of the pipeline shows that the majority of processing is concentrated on the background subtraction and connected component labeling operators. These are the only algorithms/operators that use the raw pixels of the video in order to produce results. They require multiple passes over the blocks of pixels, which introduce a significant delay in the pipeline. As a result, a significant amount of backpressure is built up in the previous operators, the Kafka consumers. Backpressure refers to the following phenomenon: When a downstream operator becomes overwhelmed and cannot keep up with the rate of incoming data, it can send a backpressure signal to its upstream operators. This signal essentially requests the upstream operators to slow down the rate at which they produce data. The backpressured operator controls the flow rate of the entire pipeline. Figure 46 shows the tasks of the pipeline along with their Busy and Backpressure metrics, as shown in Flink's WebUI.

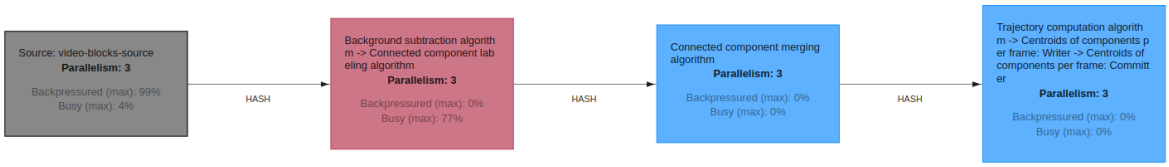


Figure 46: Tasks of the pipeline along with their metrics. The background subtraction/connected component labeling chain is at 77% busy, causing extreme backpressure in the consumer operator.

We can see that the last two tasks spend most of their time in an idle state, since their workload is considerably smaller compared to the second task.

In certain cases, usually when the input video was of the greatest quality, the busy percentage of the second task would surge to 99% for an extended period of time. This led to occasional failures in TaskManagers running the specific task, due to missed heartbeats. Heartbeats are a mechanism for JobManager to check if the TaskManagers of the job are still alive. JobManager periodically sends heartbeats to TaskManagers. If a TaskManager does not answer to a heartbeat, it is marked as dead, resulting in the job shutting down.

7 Conclusion

In this thesis, we presented the design and implementation of a distributed video processing system, that facilitates multiple object tracking and trajectory extraction. Doing so involved acquiring a comprehensive understanding of both Apache Kafka and Apache Flink, along with deep analysis of the proposed architecture. The biggest challenge in the design of the system was the mapping of the needed computer vision algorithms, such as background subtraction and connected component labeling and merging, into the dataflow model of Apache Flink. Thanks to the outstanding capabilities, well documented APIs and enormous community support of both Apache Flink and Kafka, we were able to build a system that scales as needed and outperforms its monolith implementation.

In terms of the architecture, the system consists of three main parts. The clients, the Kafka broker, and the Flink cluster. The clients are responsible for streaming the video to Kafka. In order to do so, they split each frame read from the disk into smaller blocks, generate the appropriate keys, build Kafka records and produce them to Kafka. Client are completely transparent about the parallelism value of the system. The Kafka broker is responsible for providing Flink with data to process. It contains two topics, an input and an output topic. The input topic is configured to have as many partitions as number of Flink pipelines. These partitions hold the records with the raw video data sent from the clients. The output topic contains only one partition and holds the trajectories generated by Flink for each video. The Flink cluster contains the pipeline that implement the chosen algorithms. The initial operator reads data from Kafka and creates the data objects that flow through Flink. The next two operators are responsible for performing background subtraction and connected component labeling, generating the bounding boxes and centroids for each connected component found in the input blocks. Then another operator groups all these components from each frame, and merges those that satisfy the defined merging criteria. This process generates the bounding boxes and centroids of the objects of each frame. At last, an operator receives the objects of all frames in the video, creates trajectories from each object's centroid, and then outputs the results to the output topic of Kafka.

In our experiments with the proposed system, we showed that it produces accurate trajectories for several synthetic and real world videos, and observed the different results for different block sizes. We also performed various comparisons which show that the performance of the distributed system can be up to 6 times better than the monolithic system. Moreover, in the second experiment, we showed that the system can scale as the parallelism increases, achieving a maximum speedup of 4.9x compared to the naive approach. Finally, the system's performance limits were examined, revealing its ability to process data in real-time at speeds of up to 226MB/s, considering the specific infrastructure in use.

8 Future work

In this section, we are going to present aspects of the system that have yet to be implemented and integrated. These features are expected to enhance the overall performance, while increasing its functionality and usability. At last, some of the proposed enhancements target the system's lack of fault tolerance and reliability.

Client extensions

When using the application, users may need the option to choose the background subtraction algorithm, or the color components for each input video. To facilitate this feature, it is necessary to extend the client's functionalities enabling it to generate the appropriate keys for each block. Moreover, adjustments to the Flink application are required, involving the addition of control sequences that parse the new keys and select the appropriate algorithms/methods at each step of the pipeline.

Improve key distributions and resource utilization

As of now, an important downside of the system are the skewed key distributions, which result in suboptimal resource utilization. Looking into Flink's internal mechanisms, as well as modifying the keys of the system may result in a significant improvement in the overall performance.

Enhance system resilience and reliability

As of now, the implemented system does not take advantage of Flink's fault tolerance guarantees, and is prone to TaskManager failures due to heavy workloads. Should a TaskManager encounter an exception or miss a heartbeat, JobManager immediately cancels the job. This results in the loss of all progress made in the videos flowing through the system at that time. The incorporation of Flink's fault tolerance mechanisms[11] will eliminate this problem. JobManager will be able to gracefully handle TaskManager failures without shutting down the entire application, and without losing information that was being processed in the TaskManager which encountered the failure.

Minimize the data used in the pipeline

An important aspect of enhancing Flink's video processing pipeline speed and efficiency is minimizing the data flow. This can either be achieved through adjustments on the client side (e.g. transmit each second frame), or by conducting a thorough inspection on the Flink's side, in order to determine the relevance of the information of the input and output objects of each operator. Even a small decrease in the volume of the data flowing through the pipeline can have a significant impact both in latency and throughput performance.

[11] https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/fault_tolerance/

Query support

Queries are an important feature for long-running streams. Users might not necessarily be interested in the trajectory information of all objects in a large video file. Instead, their interest might lie in detecting movement or trajectories in specific areas of the frame, defined by their rectangular coordinates within a 2D plane. The query mechanisms can either be implemented inside the Flink pipeline, or in a different module, using Flink's resulting trajectories for each video. If queries were to be implemented within Flink's pipeline, they would typically be processed immediately following the connected component merging stage.

Integration of the YOLO model

The integration of the YOLO [1] model would have a substantial impact on the accuracy of the resulting trajectories. It would also provide recognition capabilities, which the current system lacks. Most notably, it would eliminate the need for background subtraction and connected component labeling. However, it is worth noting that the task of adapting the YOLO model to work with blocks of the frame instead of the whole frame in lower resolution videos, may prove to be a quite challenging endeavor. At last, the integration of YOLO might come with extensive resource requirements.

Autoscaling support

Another important feature to be integrated in the system is an autoscaler, such as the one provided by the Flink Kubernetes Operator or an algorithm such as the one described in [21]. Autoscalers offer the advantage of fine- or coarse-grained autoscaling for Apache Flink applications in Kubernetes clusters, enabling the dynamic allocation of resources based on real-time workloads. This feature will enable the system to better utilize the underlying infrastructure and adjust its performance when needed.

References

[1] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 779–788, 2016.

[2] O. Kalentev, A. Rai, S. Kemnitz, and R. Schneider, "Connected component labeling on a 2d grid using cuda," *Journal of Parallel and Distributed Computing*, vol. 71, no. 4, pp. 615–620, 2011.

[3] D. Pawar, "Gpu based background subtraction using cuda: State of the art," in *2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, pp. 1201–1204, 2017.

[4] F. J. Seinstra, J.-M. Geusebroek, D. Koelma, C. G. Snoek, M. Worring, and A. W. Smeulders, "High-performance distributed video content analysis with parallel-horus," *IEEE MultiMedia*, vol. 14, no. 4, pp. 64–75, 2007.

[5] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink™: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, pp. 28–38, 2015.

[6] H. Wang, Z.-M. Wang, Z.-H. Miao, and X.-T. Cui, "The application of centroid tracking algorithm in video action recognition," in *2021 40th Chinese Control Conference (CCC)*, pp. 8570–8575, 2021.

[7] S. Rath and V. Gupta, "Performance comparison of yolo object detection models – an intensive study." https://learnopencv.com/performance-comparison-of-yolo-models/, 2022.

[8] M. Antonakakis, A. Tzavaras, K. Tsakos, E. G. Spanakis, V. Sakkalis, M. Zervakis, and E. G. Petrakis, "Real-time object detection using an ultra-high-resolution camera on embedded systems," in *2022 IEEE International Conference on Imaging Systems and Techniques (IST)*, pp. 1–6, 2022.

[9] L. He, X. Ren, Q. Gao, X. Zhao, B. Yao, and Y. Chao, "The connected-component labeling problem: A review of state-of-the-art algorithms," *Pattern Recognition*, vol. 70, pp. 25–43, 2017.

[10] G. Szwoch, D. Ellwart, and A. Czyżewski, "Parallel implementation of background subtraction algorithms for real-time video processing on a supercomputer platform," *Journal of Real-Time Image Processing*, vol. 11, pp. 111–125, Jan 2016.

[11] Q. Huang, P. Ang, P. Knowles, T. Nykiel, I. Tverdokhlib, A. Yajurvedi, P. Dapolito, X. Yan, M. Bykov, C. Liang, M. Talwar, A. Mathur, S. Kulkarni, M. Burke, and W. Lloyd, "Sve: Distributed video processing at facebook scale," in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, (New York, NY, USA), p. 87–103, Association for Computing Machinery, 2017.

[12] M. A. Uddin, A. Alam, N. A. Tu, M. S. Islam, and Y.-K. Lee, "Siat: A distributed video analytics framework for intelligent video surveillance," *Symmetry*, vol. 11, p. 911, Jul 2019.

[13] Y.-K. Kim, Y. Kim, and C.-S. Jeong, "Ride: real-time massive image processing platform on distributed environment," *EURASIP Journal on Image and Video Processing*, vol. 2018, 06 2018.

[14] W. Eckstein and C. Steger, "Architecture for computer vision application development within the horus system," *J. Electronic Imaging*, vol. 6, pp. 244–261, 04 1997.

[15] D. Kastrinakis and E. G. M. Petrakis, "Video2flink: Real-time video partitioning in apache flink and the cloud," *Mach. Vision Appl.*, vol. 34, apr 2023.

[16] R. Jain and H.-H. Nagel, "On the analysis of accumulative difference pictures from image sequences of real world scenes," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-1, no. 2, pp. 206–214, 1979.

[17] I. Haritaoglu, D. Harwood, and L. Davis, "W/sup 4/: real-time surveillance of people and their activities," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 8, pp. 809–830, 2000.

[18] C. Stauffer and W. Grimson, "Adaptive background mixture models for real-time tracking," in *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*, vol. 2, pp. 246–252 Vol. 2, 1999.

[19] Z. Zivkovic and F. van der Heijden, "Efficient adaptive density estimation per image pixel for the task of background subtraction," *Pattern Recognition Letters*, vol. 27, no. 7, pp. 773–780, 2006.

[20] T. Bouwmans, S. Javed, M. Sultana, and S. K. Jung, "Deep neural network concepts for background subtraction:a systematic review and comparative evaluation," *Neural Networks*, vol. 117, pp. 8–66, 2019.

[21] A. N. Zafeirakopoulos and E. G. M. Petrakis, "Hyas: Hybrid autoscaler agent for apache flink," in *Web Engineering* (I. Garrigós, J. M. Murillo Rodríguez, and M. Wimmer, eds.), (Cham), pp. 34–48, Springer Nature Switzerland, 2023.