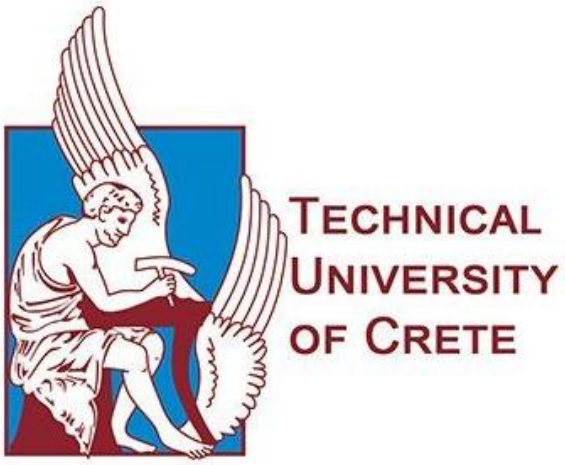
TECHNICAL UNIVERSITY OF CRETE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING



Real-time video processing in Apache Flink and the Cloud

by

Dimitrios Kastrinakis

Undergraduate Thesis

Committee:
Professor Euripides G.M. Petrakis (Supervisor)
Professor Michalis Zervakis
Associate Professor Antonios Deligiannakis

June 2022

Abstract

Σε αυτό το έργο παρουσιάζουμε ένα κατανεμημένο σύστημα επεξεργασίας ροής μέσω Apache Flink πάνω στο Kubernetes, που τροφοδοτείται από την πλατφόρμα ροής υψηλής ταχύτητας Apache Kafka. Η λειτουργία του συστήματός μας θα επεξεργάζεται ακατέργαστα βίντεο υψηλής ανάλυσης για τον εντοπισμό αλλαγών στη λήψη κάμερας. Ένας από τους κύριους στόχους αυτής της εργασίας είναι να καταστήσει αυτό το σύστημα εξαιρετικά επεκτάσιμο και ικανό να επεξεργαστεί μεγάλη ροή εισόδου βίντεο σε όσο το δυνατόν πιο κοντά στον πραγματικό χρόνο. Το Flink χρησιμοποιεί μια σειρά τελεστών για να μετατρέψει μια ροή βίντεο σε ουσιαστικά δεδομένα (δηλαδή, στην περίπτωσή μας διαφορετικές λήψης κάμερας). Αυτοί οι τελεστές μπορούν εύκολα να αντιγραφούν, δηλαδή να εργαστούν παράλληλα και να διανεμηθούν μεταξύ πολλαπλών κόμβων που διαχειρίζεται το Flink. Προκειμένου να καταστεί δυνατή η πλήρως κατανεμημένη και κλιμακούμενη επεξεργασία αρχείων βίντεο, προτείνουμε τη διαίρεση κάθε καρέ ενός βίντεο σε μικρότερα σύνολα ή ομάδες (*blocks*), τα οποία στη συνέχεια μπορούν να υποβληθούν σε χωριστή επεξεργασία από διάφορους κατανεμημένους τελεστές. Όλα τα blocks των καρέ ενός βίντεο διανέμονται πρώτα ομοιόμορφα σε topic partitions του Kafka. Στη συνέχεια, όλοι οι Flink κόμβοι διαβάζουν παράλληλα τα partitions. Για να μπορέσει να ανιχνευθεί αλλαγή λήψης μεταξύ καρέ, υπολογίζεται πρώτα το ιστόγραμμα έντασης του κάθε block από έναν τελεστή. Στην συνέχεια, μαζεύονται σε επόμενο τελεστή όλα τα blocks ενός καρέ και υπολογίζεται το συνολικό ιστόγραμμα έντασης του καρέ αυτού. Ένας τρίτος τελεστής έπειτα, υπολογίζει τις διαφορές των ιστογραμμάτων μεταξύ των διαφορετικών καρέ του βίντεο εξάγει μια συνεχόμενη ροή διαφορών ιστογραμμάτων. Ένας τέταρτος τελεστής χρησιμοποιεί την ροή αυτήν και εάν εντοπίσει μια διαφορά που υπερβαίνει ένα προκαθορισμένο όριο, ανακοινώνεται μια αλλαγή λήψης κάμερας. Διαφορετικά, ο τελεστής αναζητά αλλαγές σκηνών “fade” μεταξύ πολλαπλών συνεχόμενων καρέ. Αναπτύξαμε την Flink εφαρμογή μας σε ένα Flink cluster το οποίο με την σειρά του ήταν πάνω σε ένα Kubernetes cluster. Χρησιμοποιήθηκαν έως και 8 Flink κόμβοι, μέσω της πλατφόρμας Google Cloud και χρησιμοποιώντας την εγγενή υποστήριξη του Flink για Kubernetes clusters. Για να καθορίσουμε την επεκτασιμότητα του συστήματός μας, συγκρίνουμε την απόδοσή του με ένα μη κατανεμημένο σύστημα. Τα πειράματα παρήγαγαν εξαιρετικά αποτελέσματα επιτάχυνσης. Σημαντική βελτίωση εντοπίστηκε σε όλες τις δοκιμασμένες αναλύσεις βίντεο. Ωστόσο, παρατηρήθηκε μεγαλύτερη επιτάχυνση σε πειράματα με βίντεο υψηλότερων αναλύσεων. Επιτεύχθηκε έως και 7 φορές καλύτερη απόδοση σε σύγκριση με το μη κατανεμημένο σύστημα.

Abstract

In this work, we present a distributed stream-processing system built with Apache Flink on top of Kubernetes, fueled by the high-speed streaming platform Apache Kafka. Our system's function will be processing high-resolution raw videos in order to detect camera shot changes. This work has two mains goals. Firstly, we want to make this system highly scalable, and able to efficiently utilize multiple processing nodes. Secondly, we want our system to be able to process a high input throughput of videos in as close to real-time as possible. Flink works by applying a series of operators in a pipeline to transform a video stream into meaningful data (i.e., shots in our case). These operators can be easily duplicated. This allows them to work in parallel by being distributed inside Flink-managed nodes. To allow fully distributed and scalable processing of video files, we suggest partitioning each video frame into smaller blocks. These can then be separately processed by all the distributed operators in parallel. The blocks of each frame are first evenly distributed to multiple Kafka topic partitions. Then, all Flink nodes read in parallel from those partitions. For the purposes of shot change detection, the histogram of the intensity of each separate block is calculated by an operator. Then, a second operator assembles all the histograms of a frame's blocks into that same frame's full histogram. In the next step, a third operator receives the full histograms of adjacent frames and calculates their differences. A final operator receives these histogram differences as a stream. If a difference exceeds a predefined threshold, then a camera cut shot change is announced. Otherwise, the operator looks for gradient fades among multiple sequential frames. We deployed our Flink application on a Flink cluster on top of a Kubernetes cluster. Up to 8 Flink nodes were used on the Google Cloud Platform, using Flink's Native Kubernetes support. To determine the scalability of our system, we compare its performance against a non-distributed system. The experiments produced excellent speed-up results. An important improvement was detected in all tested video resolutions. The highest speedup however was observed in experiments with the videos of the highest resolutions. Up to 7 times better performance was reached compared to the non-distributed system.

Acknowledgements

I would like to give my sincere thanks to my supervisor Professor, Mr. Euripides Petrakis, for his all-important guidance, assistance and motivation throughout this thesis, always providing me his knowledge and suggestions to better this study. I would also like to thank Kostas Tsakos, member of Intelligent Systems Laboratory, for his great ideas and suggestions. Last but not least, special thanks go to my family, friends, and of course Fereniki Moschogiannaki, for their continuous support and love.

Contents

1. Introduction

1.1 Problem Description

Every single day data is being generated, transferred, and consumed at incredible speeds. High-resolution video data coming from video surveillance cameras, drone and car cameras, or even live media videos from streaming platforms with entertainment or news content are constantly generated. This caused the need to process video in real-time (i.e., as quickly as it is produced) to extract meaningful and useful *content* metadata. Such types of metadata can be different shot and scene detections in order to support scene classification, motion detection, object detection and tracking. The extraction and aggregation of all this metadata can lead to detailed and thorough indexing of a video based on its actual contents. In turn, this allows for a powerful content-based video organization and in-video searching.

However, video data, even in its compressed form, has a considerable file size footprint compared to image, audio, and especially text media. This results in multiple difficulties when it's required to be processed in real-time. Compressed video (e.g., MPEG-4 compression) is usually more complex to process, especially if not converted to raw format first (and converting to raw format is also a demanding task in itself). Raw video formats, however, pose their own disadvantages since they require a great amount of storage, memory space as well as significant bandwidth to be transferred. Single frames of raw video can reach sizes of over 6 Megabytes (for a typical 1920x1080 resolution colored video). Frames of this size are too big to transfer and process without splitting them up into smaller parts. A system that will be able to efficiently process multiple raw videos simultaneously in real-time requires a way of dividing each frame into even smaller parts. It will also need a way of properly aggregating the partitioned data in a distributed environment and be processed in parallel.

1.2 Proposed Solution

In this work we present an approach for detecting camera shots in raw high-resolution videos in a fully distributed and highly scalable way, using Apache Flink's distributed processing engine and Apache Kafka as the streaming platform. Each video frame is split into smaller blocks that are easier to transfer in a highly distributed system and process in parallel. The blocks are evenly distributed to multiple partitions of the Kafka input topic. Next, an equal or smaller number of parallel Flink pipelines consume that data simultaneously. They process each block by applying a specific series of transformations through multiple Flink operators. The sum of these transformations leads to detecting camera shots in these videos. The first operator in each pipeline is a Kafka Consumer, which thanks to its pull-based nature, it requests data only when the second operator in line (operator A) is ready to process the next block of data. If the received block of data is in RGB format it is transformed to grayscale by the second operator. Otherwise, it is simply forwarded to the third operator (operator B). This operator extracts from each input block of grayscale data, its histogram of intensity. The next operator (operator C) receives all the histograms of intensity of all blocks of a specific frame. Combining all the partial histograms of a frame generates the frame's full histogram of intensity. This is sent to the fifth operator (operator D), which compares the histograms of adjacent frames and calculates the difference between the two. The final operator (Operator E), receives a stream of unordered histogram differences from the previous operator. If any difference is over a predetermined threshold, it detects a camera shot. Otherwise, it looks for gradual fade camera shots in sequential frames. To achieve this operation, temporarily saving some of these histogram differences in memory is required. If any kind of shot is detected it is announced through an appropriate message that is

sent to another Kafka topic. Multiple videos of either RGB or YUV encoding can be processed simultaneously by the system, by using Flink's and Kafka's extensive key mechanisms. Scaling this system up comprises of simply increasing the number of partitions for the input data Kafka topic as well as configuring the Flink Application with higher parallelism. Our Flink application is run on a Flink Cluster which itself is run on a Kubernetes cluster on the Google Cloud Platform. By increasing our Flink Application's desired parallelism, the Flink Cluster and the Kubernetes cluster, work together to seamlessly initiate the required worker nodes. Both of our main systems (Kafka and Flink) operate completely autonomously. Our work's main contributions are:

- A highly scalable Flink pipeline for video and shot detection processing that doesn't use any external storage.
- A distributed system that can simultaneously process blocks of videos with different frame resolutions or encoding by fully utilizing Kafka's and Flink's key mechanisms.
- Demonstrating through experiments how performance speedup depends on input throughput, at what point Kafka may become a bottleneck to Flink and how to identify that critical point.

1.3 Related Work

In reference [1], the researchers describe how they transitioned from a monolithic encoding script to a custom distributed and pipelined "Streaming Video Engine" (SVE). User videos are split into smaller videos, each with a duration of 10 seconds or up to 2 minutes at most. The different size for the video segments controls the tradeoff between processing latency and compression efficiency. These segments are then processed in parallel from a pool of distributed worker machines. A set of processing tasks for each full video may include HD and SD encoding, segmentation counting, and track combination (video and audio tracks) based on its origin and metadata. Compared to the original sequential, non-parallel system, SVE managed to improve the latency due to the parallelization of the most-demanding processing tasks. Faults were an inevitability for SVE due to the scale of the systems, the incomplete control of the pipeline, and the diversity of the uploaded media. A large effort was successfully put into fault tolerance, resulting in a robust system with no impact on latency even on harsh failure tests. SVE managed to achieve a latency speedup on video processing, ranging from 2 times less latency for short video sizes (up to 1MB) up to 9 times speedup for the largest videos (1GB and larger). However, it is unsuitable for processing streams of data at real (or close-to-real) time. SVE is built around processing multiple seconds of video at a time (at least) to be able to efficiently compress its videos. Whereas a Livestream processing application has access to only a short duration of the video each time (e.g., a few frames per second or 1 second of video per second at most). Our system is designed to process data at the rate of a video's framerate (e.g., at 24 frames per second) or faster. This is accomplished by splitting the frame into smaller segments and processing them individually in a distributed environment.

SIAT [2] is a distributed video processing framework that provides multiple video analytics services such as video encoding, compression, segmentation, classification, etc. It uses Apache Kafka as its stream acquisition technology and can work with both streams and batches of data. Each different service they provide uses its own topic with multiple partitions for high throughput. The video processing layer they use is built on top of Apache Spark together with the popular image and video processing library, JavaCV (a ported library of OpenCV). They used multiple of its low-level functions such as RGB to grayscale conversion, key-frame extraction, etc. On top of these functions, they built a distributed video processing and mining API. The authors conducted multiple scalability experiments on services they provided (shot detection/key frame extraction, encoding, deep feature extraction, classification). By comparing the performance of using 1 versus 4 distributed nodes, they averaged a consistent speedup of 3.5 on their experiments. What we did differently, is that in our work we aimed to avoid the use of any

external libraries and specifically “black-box” image processing functions. Instead, by building our algorithm step-by-step, we could then assign each step to a different Flink Operator and define the operations to work at any level of detail. This allowed each one to be individually distributed and parallelized, thus providing increased efficiency.

The authors of [3] use Kafka with a cluster of distributed worker machines to process different types of video data. They use topics to differentiate between different video producers and frame resolutions (i.e., a camera with a specific video resolution uses Topic A, a remote channel is transmitted to Topic B, etc.). The RGB frames of each video are serialized, sent to Kafka, and processed individually. In other words, a frame is the smaller amount of information that their system could process in parallel. This smaller data package (i.e., single frames) used together with Kafka, allows their system to utilize their distributed workers more evenly and efficiently. During the scalability experiments they run, they saw a speedup of about 3 times faster processing when increasing the available nodes from 1 to 4. However, since all workers provide the same processing, it seems unnecessary to use multiple topics only to distinguish between different frame resolutions. The resolution of each frame is indeed necessary information for the worker nodes to be able to correctly process each frame. But in our work, this type of information is directly embedded in each data package (data blocks). This allows us to use a single topic (with multiple partitions), from which all workers request data.

Reference [4] is a system that allows the creation of user-defined execution plans (i.e., a chain of processing tasks,) mainly for visual video parsing (e.g., object classification, object tracking). This is accomplished by choosing from a set of visual recognition algorithm modules to create a final plan. These modules are predefined by the authors, but additional custom ones can be created by the users through their WebUI with notable flexibility. Each module is essentially a Spark application (i.e., an application that can contain parallelized and distributed processing tasks). These modules can be connected through Kafka and form an acyclic graph of specific processing steps, to eventually create a task. Also, by using Kafka as their inter-module communication they allow the processing capabilities of each module (i.e., each Spark Application) to greatly differ. This is enabled by Kafka’s pull system for its subscribers, which allows their Consumers to request data whenever they are ready to process the next set of data. This results in an even more flexible and reliable system. The authors mentioned that they will provide performance experiments in the future. Our work focuses on using a single Flink application with a specific task (instead of multiple customizable ones) and achieves parallel processing by using distributed Flink operators. Kafka is used for the input and output of the overall Flink task, but the inter-operator communication is entirely handled by Flink.

Lastly, in this work from 2007 [5], the framework “Parallel Horus” that is described, allows users to write sequential programs for multimedia analysis, that are automatically converted to distributed tasks, optimized, and run on Grid clusters. The framework is a continuation of an existing non-distributed multimedia analysis framework. Similar to [3], they also use uncompressed color video (of low resolution such as 352*240 or 412*318). Additionally, in specific tasks, they process each video frame individually, with good performance results. On their Grid scalability tests, they compare the performance of a single core against up to 96 cores (using 4 available clusters simultaneously). With 4 cores they gained a speedup of about 4 times the initial performance and with 8 cores they gained a speedup of about 8. At 64 cores a speedup of 32 was reached in one of their experiments. The framework however seemed that it would be difficult to integrate with any outside data sources (such as a streaming video source) since you could only use actual files as input. As a result, this significantly reduces its available use cases, such as using the system to process a video stream of surveillance or drone camera.

2. Background

2.1 Apache Flink

The content of the following section is based on Flink's documentation[1].

> "Apache Flink[2] 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."[3]

Stream Processing

There is a plethora of data available that is either created in stream form or can be represented as such. Logging events, Credit card and bank transactions, User interactions events from a website or a mobile app, or even Database CRUD operations are fundamentally generated as stream events.

Flink can handle two types of streams. Unbounded and Bounded streams. Unbounded streams are considered to be endless, without a total aggregation to be possible and instead must be processed continuously. Most of the time, their ingestion order is also highly important as it could indicate their order in which the events occurred. Bounded streams (also known as Batch processing) on the other hand have a definite end boundary. This allows the process to first wait out all the available data before doing any kind of calculations, if designed that way. Bounded data streams can also be easily sorted, eliminating

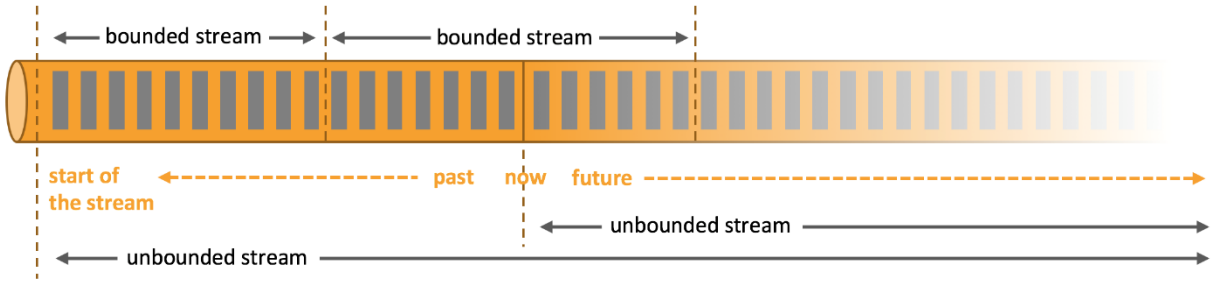


Figure 1:Difference between bounded and unbounded streams

the need for strict ordering of the data ingestion.

Both time of streams are excellently handled by Flink, by providing precise timing and state handling for unbounded streams as well as using specific algorithms and data structures for fixed sized data sets for bounded streams.

Deployment Options

Flink is easily integrated with any of the common cluster resource managers such as Hadoop YARN, Apache Mesos, and Kubernetes, but has also the ability to be setup as a stand-alone cluster or even run in single application mode. Flink automatically identifies the application's configured parallelism and requests the required resources from the resource manager, while also requesting new containers in case

[1] https://nightlies.apache.org/flink/flink-docs-release-1.14/
[2] https://flink.apache.org/
[3] https://flink.apache.org/flink-architecture.html

of a failure. Finally, for all of its job control and submission, Flink uses the REST protocol which allows for easier integration with other systems.

Flink Applications

In Flink a single application is referred to as a Job. A Flink job describes how a streaming dataflow will be transformed by a series of user-defined operators. These dataflows start with one or multiple data sources and end in one or more sinks. In the example in Figure 2, we can see a dataflow with a Source, a single map transformation, a keyed (i.e., redistributed and partitioned stream) windowed aggregation, and a Sink.

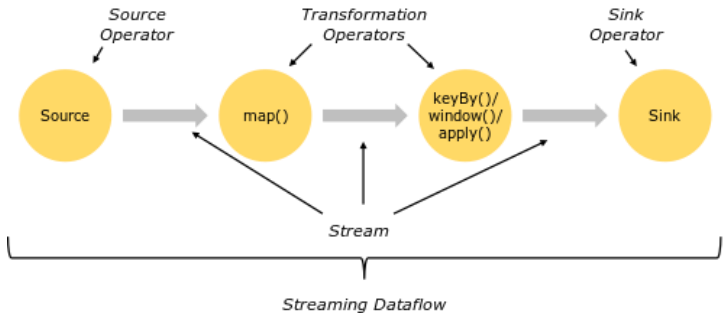


Figure 2: Simple Flink application example, with a source, a map transformation, a windowed aggregation and a sink.

Sources and Sinks

Flink has a plethora of options concerning data input and output. It may consume and output data to message queues or distributed logs like Apache Kafka or Kinesis, a device's raw data output, a database or even from much simpler forms, such as a file or a console.

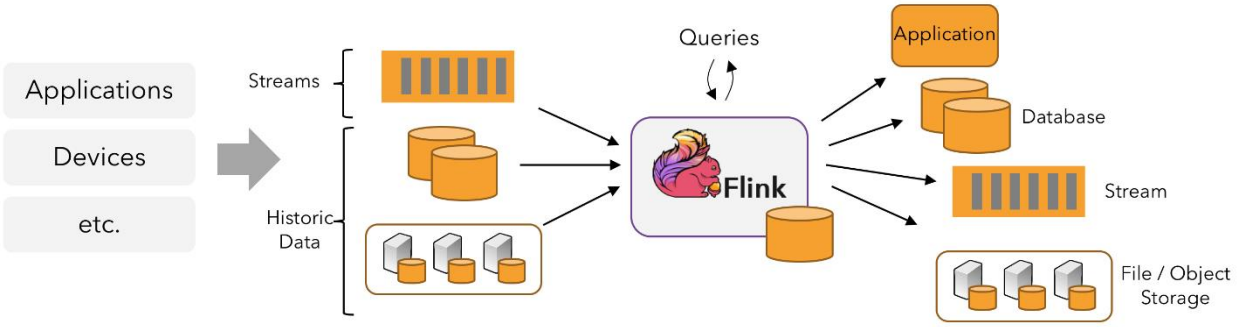


Figure 3: Various options of data sources and sinks for Flink.

Parallel Workflows

> "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 stream can transport data between two operators in a one-to-one pattern (forwarding pattern) or a redistributing pattern. The one-to-one, means that for example with two stream partitions, the elements of each partition will stay in the same one with the same exact order. On the other hand, a redistributing pattern will change the partition of the elements and place each one on a specific partition, usually based on a key.

An important note, is that there is no communication between two operator subtasks of the same operator (e.g., two map operator subtasks can not send data between them).

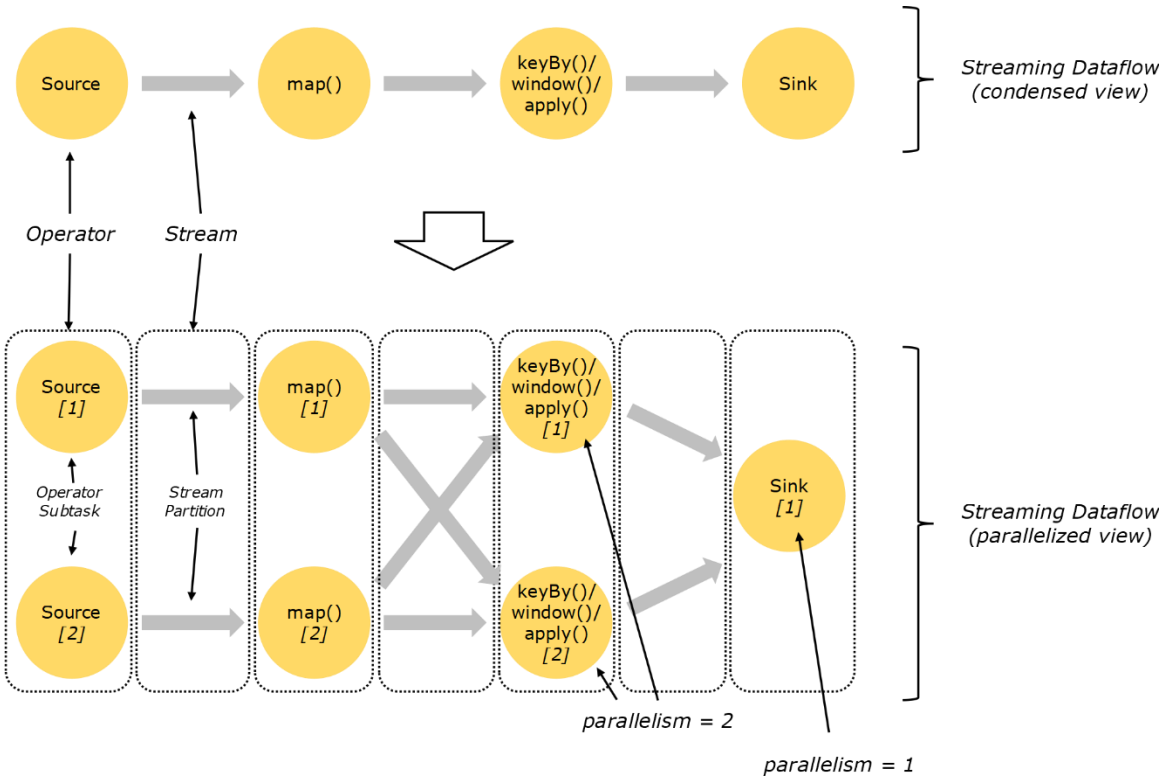


Figure 4: Parallel dataflow example

***Example** (Figure 4)*
In the diagram, we can see that for the operators with a parallelism of 2, two operator subtasks exist. The two Source operators can be consuming from an input stream different parts of it in parallel (e.g., A Kafka topic with 2 partitions). The two stream partitions are then forwarded (one-to-one) to the next map operator. The result of the map operator then gets redistributed **based on a key** to the next operator, where the elements are windowed (i.e., grouped based on a certain attribute, such as time or up to a count of elements). An aggregation function is then applied to each window (set of elements) and the result is sent to a sink of parallelism one.

Keyed Streams and Stateful streams

One important aspect of Flink's framework is how it handles stream partitioning and states. In order to use on an operator the native local state that Flink handles, it strictly requires that the stream is Keyed. In order to create a Keyed stream, the keyBy function is used on a certain attribute of each element (e.g., event type, country of IP address) and it creates a **logical** partition for each of unique key. Then each logical partition is assigned to a certain operator subtask to be processed on.

After a stream is Keyed, Flink allows the use of an in-memory performant local state, scoped to each key, or in other words, each key has its own state, a key-value store. Of course, multiple keys with their logical stream partitions and state can reside in a single operator subtask, comprising a group of keys. An example showing, 6 different keys being redistributed in 2 parallel operator subtasks can be seen in Figure 5. The key-scoped state can be seen next to each operator.

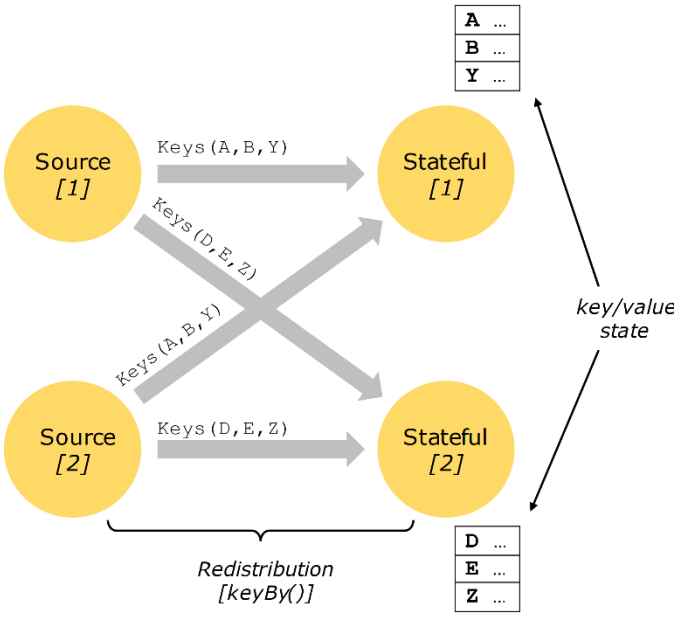


Figure 5: Stream redistribution based on key.

The partitioning of the stream into as many logical partitions as the keys and the equal redistribution of them, is one of the main reasons Flink succeeds so well in trivializing the parallelization of the applications, since the designer /developer of the system only has to think about which key to use (i.e., what attribute of each element) to split the data stream in order for it to be parallelizable and after that the rest is fully handled by Flink.

Flink Architecture

Flink's runtime includes two types of processes. A JobManager and one or more TaskManagers. A figure illustrating the architecture can be seen in Figure 6 bellow.

JobManager

The JobManager is the coordinator of the distributed Flink applications. Its responsibilities include among others, when to schedule a task (or a set of tasks), reacting to tasks that finished or failed, coordinating checkpoints and failure recovery. These responsibilities can be partitioned into three different components:

ResourceManager:

The ResourceManager is responsible for allocating and deallocating available resources in a given Flink Cluster, by managing the slots, the smallest unit of resource scheduling. By finding out the required number of slots, the ResourceManager requests an appropriate amount of TaskManagers from the current resource provider. However, if Flink is run in standalone mode, then no further TaskManagers can be started via the ResourceManager.

Dispatcher:
The Dispatcher provides a REST interface to submit a new Flink application (job) and starts a new JobMaster for each submitted job. It also runs the Flink WebUI to provide information about job executions, the available slots, TaskManagers and their status, as well as a graphical interface alternative to submitting new jobs.

JobMaster:
A JobMaster manages the execution of a single job graph. With possibly multiple concurrent jobs inside a Flink Cluster, multiple JobMasters for each one also exists.

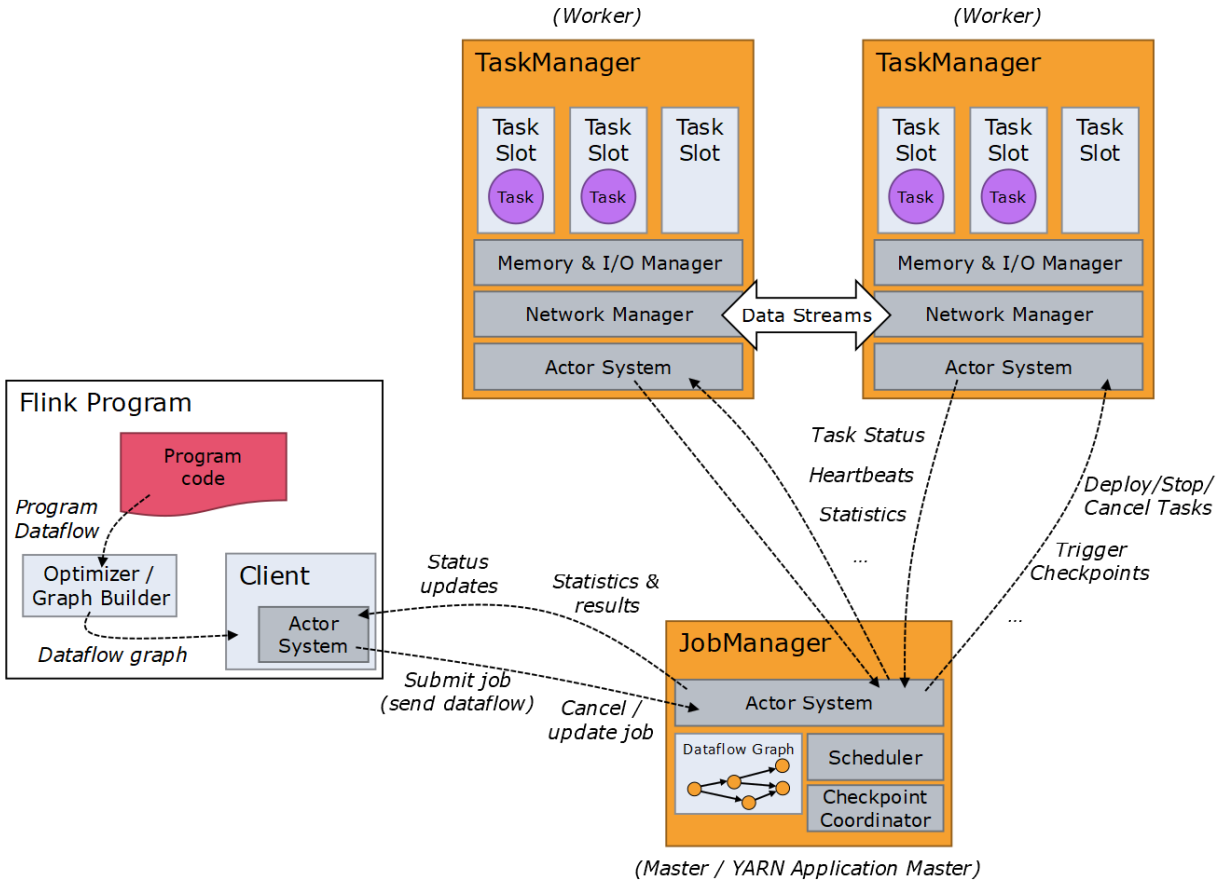


Figure 6: Flink's runtime architecture

TaskManagers
TaskManagers are the workers of the cluster, executing the tasks of a job graph (i.e., dataflow) while also buffering and exchanging the data streams. In a Flink Deployment at least a one TaskManager must exists in order for the applications to be executed. Each TaskManager contains at least one task slot, the smallest unit of resource scheduling inside a TaskManager. The number of slots inside a TaskManager indicates the maximum concurrent processing tasks

Tasks, Subtasks and Operator Chains

Each single operator (e.g., a source operator or a transformation function such as a map function) is a *task*. Each one is comprised of as many *operator subtasks* as the parallelization factor of that operator, with each subtask is a single thread. Flink can chain operator subtasks (of different operators) together into single tasks, creating an *operator chain*, essentially residing in the same thread, reducing overhead of thread-to-thread handover and buffering, increasing overall throughput and decreasing latency. Some operators are chained by default (e.g., Source operators to the next operator).

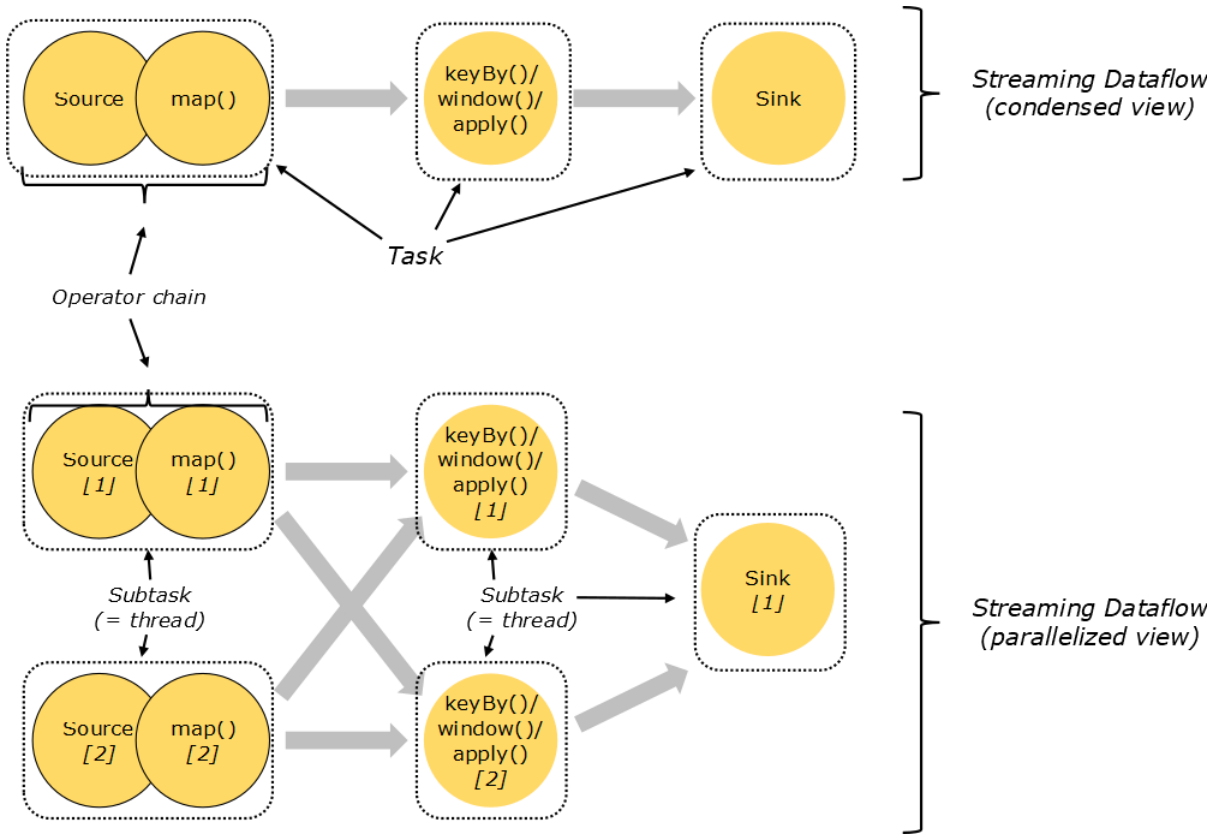


Figure 7: Example dataflow, showcasing tasks, subtasks and operator chains.

***Example** (Figure 7)*

The example dataflow includes four operators and 3 Tasks, since the Source and Map operator are chained. The first 3 operators have a parallelization of 2, so the first 2 tasks (Source/map & keyBy/window/apply) have each 2 subtasks, i.e., two threads each that can run in parallel. The final Sink operator has a parallelization factor of 1 and thus has only one subtask and is single-threaded, receiving all the elements from the previous two subtasks. In total, the final dataflow is comprised of 5 subtasks and thus 5 threads.

In the dataflow we can also see that the 2 stream partitions (outputted by the 2 map operators subtasks) are redistributed instead of forwarded to the next 2 subtasks due to the keyBy function found before the next apply operator.

Task Slots and Resources

Each worker (TaskManager) is a JVM process, and may execute one or more subtasks in separate threads. To control how many tasks a TaskManager accepts, it has so called task slots (at least one). Each task slot represents a fixed subset of the TaskManager's resources. For example, in a TaskManager with three slots, each one will have 1/3 of its total managed memory. However, no CPU isolation is happening and all slots and contained subtasks share the available CPU. Slots also share the TaskManager's JVM, which allows for TPC connection sharing (via multiplexing) and heartbeat messages, and possibly data sets and data structures, thus reducing the per-task overhead.

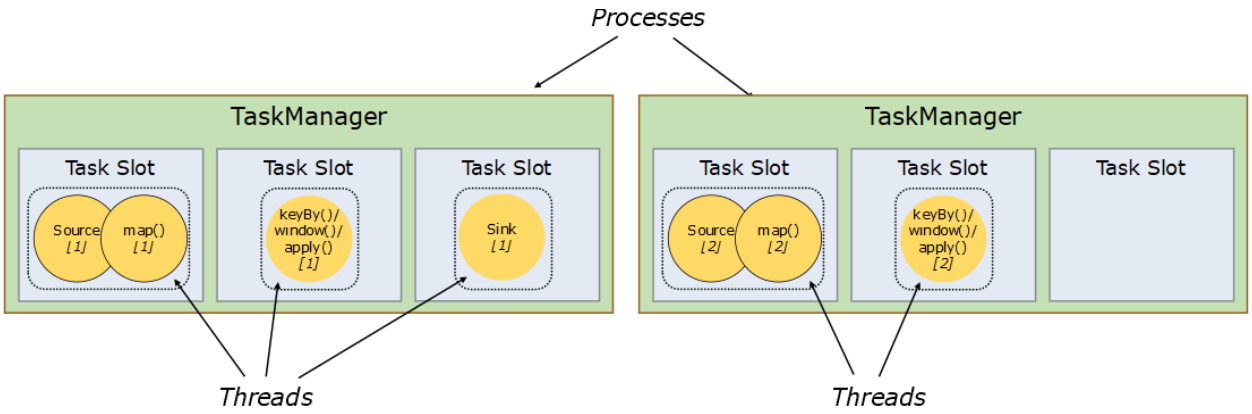


Figure 8: Two TaskManagers, with 3 slots each. Inside each slot, a single subtask is placed (except the last slot, which is empty).

Each subtask needs to be placed inside a single slot. It is possible but not necessary, to place each subtask inside a single slot, requiring the same number of slots as subtasks. However, Flink allows by default slot sharing, which means that subtasks of different tasks can share a single slot, so long as they are from the same job. The result is that one slot may hold an entire pipeline of the job (as shown in Figure 9). Allowing this slot sharing has two main benefits:

- A Flink cluster will need exactly as many task slots as the highest parallelism used in the job.
- Better resource utilization: Without slot sharing, any non-intensive tasks (e.g., source/map operator chain) would block as many resources as the more intensive tasks. By using slot sharing, we are sure that each heavy subtask will be fairly distributed among the TaskManagers.

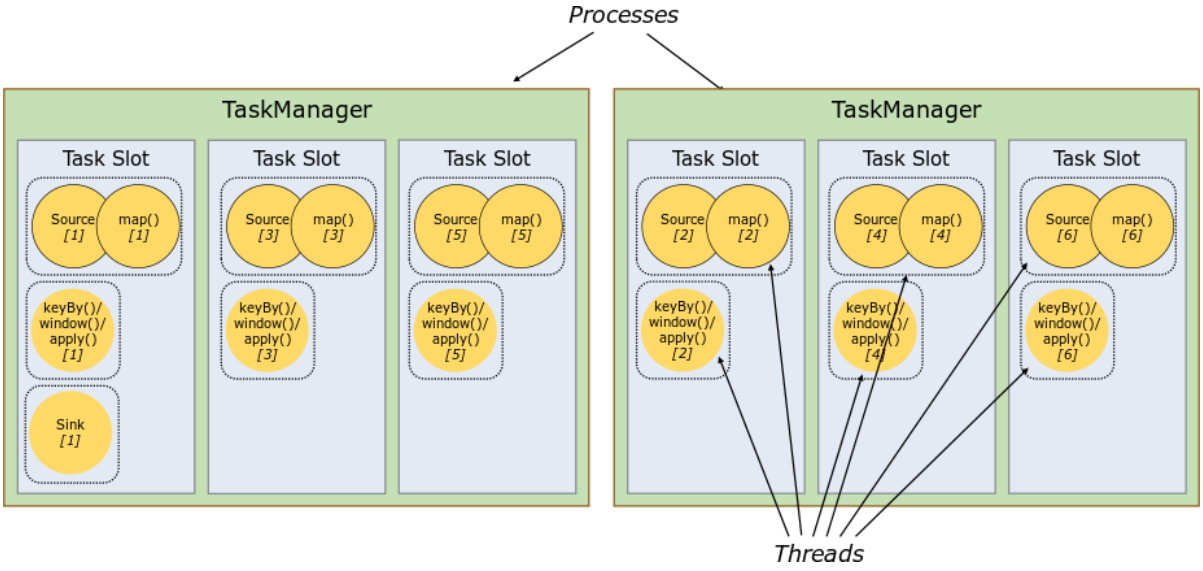


Figure 9: A Flink application with slot sharing enabled. Each slot contains a job's entire pipeline.

2.2 Kafka

The content of the following section is based on Kafka's documentation[4].

> "Apache Kafka[5] is an open-source distributed event streaming platform and is used for high-performance data pipelines, streaming analytics, data integrations, and mission-critical applications."

Kafka's aim is to provide a high-throughput, low-latency platform to handle real-time data feeds. It is a highly scalable messaging system that can also provide permanent storage and high availability. Another of its advantages is its ability to connect to a plethora of external systems, allowing for easy integration within any project.

Event Streaming

> "Event streaming is the practice of capturing data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events; storing these event streams durably for later retrieval; manipulating, processing, and reacting to the event streams in real-time as well as retrospectively; and routing the event streams to different destination technologies as needed. Event streaming thus ensures *a continuous flow and interpretation of data* so that the right information is at the right place, at the right time."

Kafka is a "publish-subscribe" system; the produced messages are published (written) to a (single or multiple) *topics* and any client can choose to subscribe (read) any of the available topics. Kafka is also a pull-based messaging system, instead of push-based, meaning that any message consuming clients can choose when to read available messages (pull) instead of being sent messages as soon as they become available. This is an important aspect and one of the reason Kafka can easily scale with a large number of consumers, since it allows each one to read messages at their own pace, depending on their different data needs and available performance.

Architecture

Kafka consists of servers (*Brokers*) and clients (*Producers* and *Consumers*) that communicate via a high-performance binary network protocol over TCP[6]. Brokers are the middleman of this architecture and are responsible for receiving, storing, replicating and sending the messages. A Kafka system can consist of a single Broker or multiple, comprising a Kafka Cluster with better scalability and fault-tolerance. Clients are either Producers or Consumers. Producers are the client applications that publish events (i.e., messages) to a Kafka Broker and Consumers can subscribe to events through a Broker, requesting any available ones and processing them. Producers and Consumers are completely agnostic to each other.

Messages: Kafka messages (sometimes called events), are quite simple, containing only a few attributes: A specific key (in the form of a string), it's value (Any serializable type of data), a timestamp and optional metadata headers.

Topics: Messages are organized and durably stored inside *topics*. Topics are similar to a folder in a filesystem, and the messages are the files inside that folder. Any number of Producers can choose to send messages to a particular topic and similarly, any number of Consumers can choose to subscribe and read from those topics. Unlike other messaging systems, any messages read are not deleted after

[4] https://kafka.apache.org/documentation/
[5] https://kafka.apache.org/
[6] https://kafka.apache.org/protocol.html#protocol_network

consumption and a Consumer can choose to re-read any number of them. Instead, a retention policy can be defined per-topic, describing for how long a message will be kept or how large size-wise a topic can get before deleted or compacted.

Partitions: Topics are further split into smaller "buckets" called partitions, that can be located on different Kafka Brokers. This distributed placement of your data is very important for scalability because it allows client applications to both read and write the data from/to many brokers at the same time. Each message being published to a topic is actually appended to a single partition, chosen either randomly or based a key, accompanying each message. Different events (i.e., logical types of messages) can be organized in the same topic to different partitions based on a unique event type key. Messages inside each partition are guaranteed by Kafka to be read in exactly the same order as they were written. Partitions are what allow for truly parallel data consuming.

Topic Replicas: In favor of fault-tolerance and high-availability, Kafka allows topics to be replicated a configurable number of times. These topics are kept in-sync by the Brokers.

Consumer Groups: Each Consumer is always part of a consumer group which can be joined by any number of Consumers. Inside a consumer group, each topic partition (and thus each message) can only be read by at most one consumer of that group. In other words, the index of the last read message in each partition is shared among the consumer group.

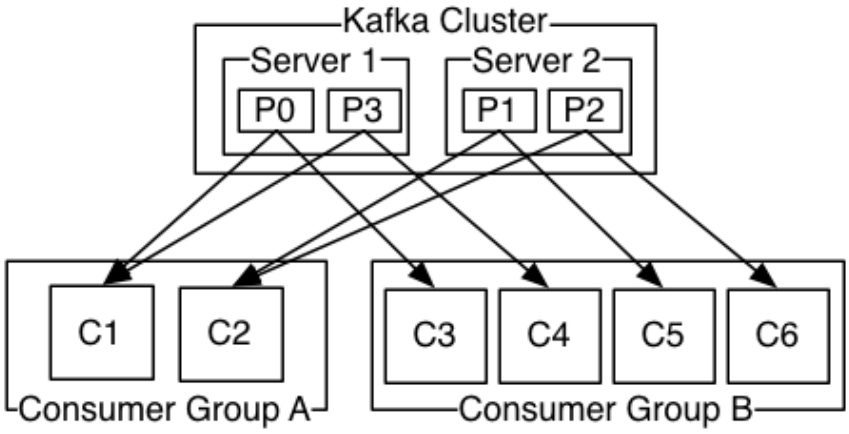


Figure 10: Example Kafka arrangement with a single topic with 1 topic and 4 (distributed) partitions. From the first consumer group, each of the 2 consumers reads 2 partitions, whereas from the second group, each one reads exactly 1 partition.

Kafka APIs

Kafka consists of five core APIs:

- The *Producer API* allows to sends streams of data to topics in the Kafka cluster.
- The *Consumer API* allows to receive streams of data from topics.
- The *Streams API* enables the transformation of streams of data between two topics.
- The *Admin API* through which topics, brokers and other Kafka objects can be managed and inspected.
- The *Connect API* allows implementing connectors to create custom Kafka Sources or Kafka Sinks.

2.3 FFmpeg

FFmpeg[7] is an open-source multimedia framework that allows any desirable actions upon multimedia files (videos, images, audio), such as decoding/encoding, streaming, filtering and or simply playing. Through its continuous development, it supports any kind of formats from considerably older ones up to the newest ones. It contains a plethora of available libraries, written in C as well as the fully featured command line programs, *ffmpeg*, *ffplay* and *ffprobe*.

Example Use Cases

Some of its most used use cases are encoding/transcoding/decoding video files into a different format. For example, with the ffmpeg tool, one can encode a raw RGB video file into a compressed H.264 format, with the following command:

```
ffmpeg -s 640x480 -i input.yuv -c:v libx264 -s 640x480 output.mp4
```

The size of the frame has to be explicitly stated (at least for the input file) for raw videos, since they do not contain that information on their own. Using the same option however, the output video can be resized to any desirable frame size (e.g., in this case we could change the second "-s 640x480" option to "-s 1280x960" to double the resolution). Relative sizes can also be used (e.g., "double or half the previous height while keeping the same aspect ratio").

FFmpeg can also very easily trim a video, change its bitrate or even its framerate. An example command that trims to the first minute and at the same times change the framerate can be seen below:

```
ffmpeg -ss 00:00:00 -i input.mp4 -to 00:01:00 -filter:v fps=30 output.mp4
```

If the original input video was 60fps, then ffmpeg would drop every other frame to get a 30fps output video. Another related available option is to speed up or slow down the input video.

An additional powerful feature ffmpeg provides is the select filter which has a great number of available parameters based on which to essentially filter the output video. In the following example, we extract from the input mpeg-4 encoded video only the I-frames, and saving each one as single image.

```
ffmpeg -i input.mp4 -vf "select=eq(pict_type\,I)" \
-vsync vfr thumbnails-%02d.jpeg
```

Conclusion

The ffmpeg command line tool has incredibly more features and options to learn and use, but at the same time, if needed to be used in its simplest form, ffmpeg has defined a great set of default values for all of its options, allowing for quick and easy use of the features it provides. On top of that, FFmpeg comes with great documentation and an extensive online community with plenty of answered forum questions and useful discussions to learn from.

[7] https://ffmpeg.org/

2.4 Kubernetes

The content of the following section is based on Kubernetes' documentation[8].

Kubernetes[9] is an open-source container-orchestration system that is able to automate the full application deployment, scaling and management and it builds upon the benefits of containerized applications. In short Kubernetes provides a handful of useful 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 to automatically mount a storage system of choice, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks:** The desired state of the deployed containers can be described and Kubernetes can change the actual state to the desired state at a controlled rate.
- **Automatic bin packing:** By providing Kubernetes a pool of nodes that it can use along with the CPU and memory requirements of each container, it autonomously places each container in order to maximize resource usage efficiency.
- **Self-healing:** Kubernetes restarts containers that fail, replaces containers and kills containers that don't respond to your user-defined health check.
- **Secret and configuration management:** Kubernetes allows storing and managing sensitive information, such as passwords, OAuth tokens, and SSH keys. Secrets and application configuration can be deployed and updated without rebuilding any container images, and without exposing secrets in the stack configuration.

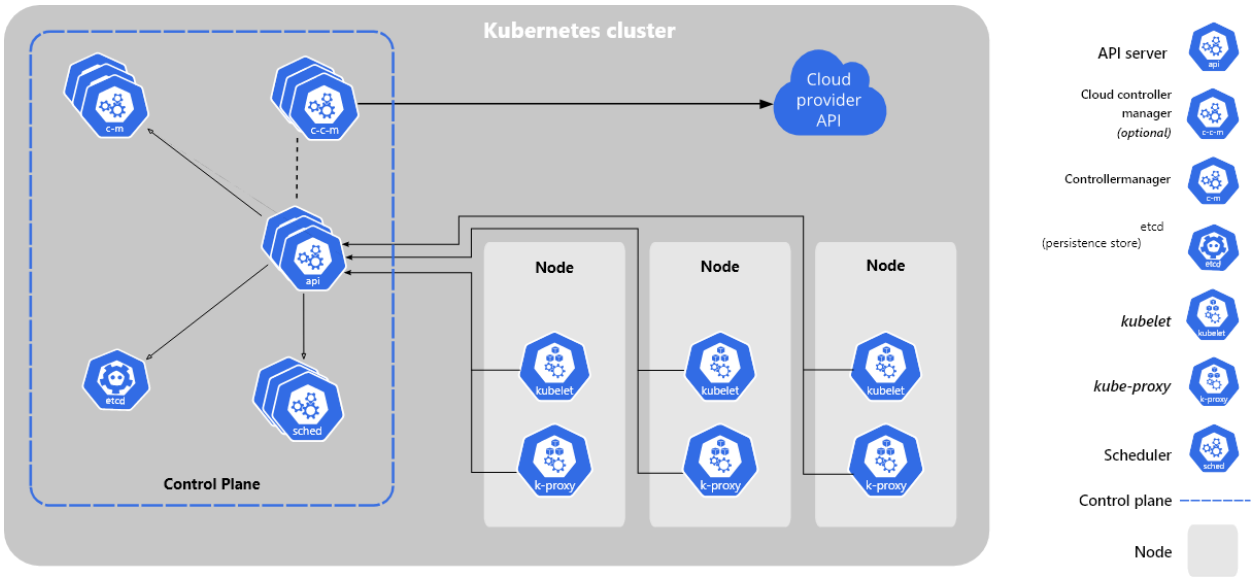


Figure 11: Kubernetes Component map

[8] https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
[9] https://kubernetes.io/

Kubernetes Components

A Kubernetes cluster consists of a set of worker machines, called *Nodes*. Each cluster has at least one. Inside each Node can be placed one ore more *Pods* which are the components of the application workload. Next, the *Control plane* components which are responsible for all the global decisions of the cluster, i.e., managing the Nodes and Pods, as well as detecting and responding to cluster events (for example, starting up a new pod when a deployment's replicas field is unsatisfied). In a production environment the Control plane may run across multiple computers to provide increased fault-tolerance and high availability.

Inside each Node there is a kubelet and kube-proxy Pod. The kubelet is an agent that makes sure that the containers are running inside the Pods and makes sure that each of them is healthy based on a set of PodSpecs. The kube-proxy is a network proxy that is part of the Kubernetes service concept. It maintains network rules on nodes which allow network communication to Pods from network sessions inside or outside of the cluster.

2.5 Flink Native Kubernetes

Flink provides native support for Kubernetes[10] and highly suggests new users to use this option versus deploying a standalone Flink cluster which requires creating all the required Kubernetes configurations on their own from scratch. The native option isn't actually constraining a developer's options to finely tune a Kubernetes cluster, rather it hides much of the initial complexity as well as saving a lot of time by automating a big part of the process such as starting and stopping a cluster, controlling its scale and submitting Flink jobs to it. It is excellent for speeding up the development time, by allowing more time to be invested on creating and testing designs rather than their actual time-consuming implementation and configuration details.

Any configuration changes can be set inside the flink-conf.yaml file. Briefly, among other things this includes CPU and Memory requests for each JobManager and TaskManager, slots per TaskManager, default parallelism for operators, addresses and ports for various network configurations etc. Additionally, any overrides on the default templates files for any of the pods or services can be defined here, which docker image to be used for a job's TaskManager containers and any environment variables our app may require.

After the Flink cluster has been deployed to the Kubernetes cluster, at least one JobManager will be running, hosting the WebUI and waiting for job submissions. After submitting a Flink job to it and the job's graph has been successfully extracted, the required task slots are calculated and the appropriate amount of TaskManager pods is requested from the Kubernetes Cluster. If the amount of vCPU and Memory required by the pods is available in the cluster, the TaskManager pods will be shortly available, in communication with the JobManager and ready execute any operations required.

By default, since a single job's subtasks can share the same slot, the number of slots requires is exactly the maximum parallelization among the operators of that job. For example, with a max parallelization of 6 for a job's operators and three slots per TaskManager, the end result would look like Figure 9.

[10] https://nightlies.apache.org/flink/flink-docs-release-1.13/docs/deployment/resource-providers/native_kubernetes/

2.6 Video Segmentation

The content of the following section is based on Kubernetes' documentation[11].

In this work, the aim of our processing was video segmentation. In order to partition a video or any other kind of media, we have to find it elemental index units. In text for example, that would be its words, phrases and sentences. In a video these are the distinct uninterrupted camera shots.

In the simplest form, a camera break indicates the simplest form of boundary between two different shots. A more complex form of shot change can be a gradual transition such as a fade-in or fade-out, dissolve or wipe. Finally, actual camera movements themselves can be considered a boundary between two shots, like pan and zoom. Such effects can take up to 5 or even 10 seconds to complete which can complicate and hinder their detection.

There are two main ways to detect a change in shot between two individual images. Pair-wise pixel comparison and Histogram comparison. As the name denotes, in the first method (for a grayscale image) we simply observe the difference in intensity of each pixel between the two frames. If the count of the pixels that did change surpassed a certain percentage of the total number of pixels, then a shot is declared with these two frames as its boundaries. A slightly more resistant modification of the same idea against camera movements is comparing whole regions of the image altogether.

The second main method of shot detection and the one we will be using ourselves, is the comparison of intensity levels histogram. Such a comparison is largely unaffected by slow camera movements and general in-shot object motion, since it ignores spatial changes inside a frame.

Let $H_i(j)$ denote the histogram value of the i^{th} frame with j being one of the possible G gray levels, which in turn is the number of histogram "bins", a value that we can increase or decrease depending on the video's bit depth and our desired computational time. The difference between two consecutive frames (i^{th} and $i^{th}+1$) can then be calculated equation (1):

$$SD_i = \sum_{j=1}^{G} |H_i(j) - H_{i+1}(j)| \qquad (\mathbf{2.1})$$

If this sum of differences between two frames is higher than a given threshold T_b a camera shot is declared. For a suitable T_b threshold, we can normalize SD_i by dividing it by the product of G and $M * N$, the total number of pixels in the frame. In *Figure 12* we can see a sequence of normalized frame-to-frame differences, taken from a regular tv show scene in which two camera cuts can be observed (the two large spikes) and an acute and long gradual transition between two shots.

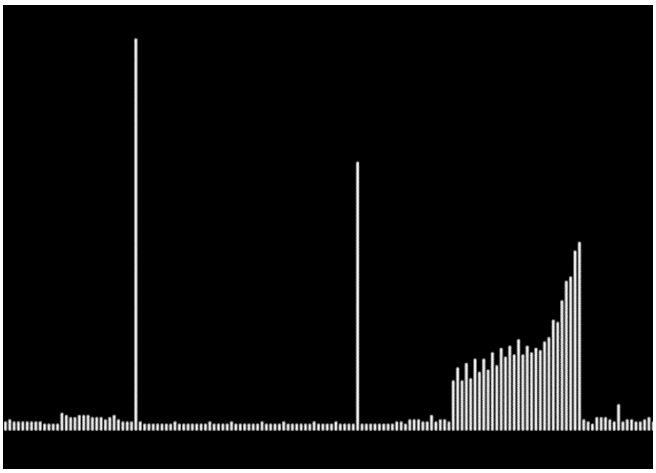


Figure 12: Sequence of frame-to-frame histogram differences, taken from a short tv-show scene. In the image there can be seen two camera cut-offs and one rather acute gradual transition.

[11] https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

A necessary step in applying any of the two techniques described above **is finding a threshold T_b** for distinguishing the shot boundaries. Selecting a good threshold will allow for higher accuracy in finding those shot, with low or "loose" thresholds allowing many false positives but catching all of the actual camera shots while a tighter threshold would be prone to missing some true transitions. Its value is possible to be determined a priori experimentally, i.e., by calculating the same video's full length histogram differences and obtaining it via formula (2):

$$T_b = \mu + \alpha\sigma \qquad (\mathbf{2.2})$$

where μ and σ are the median and standard deviation respectively, of the frame-to-frame differences and α is a constant. Experimental evidence [6] indicates that a good value for α is between five and six. Under a Gaussian distribution the probability that a frame that does not belong in a transition will be over this threshold is close to zero.

Gradual Transitions

For detecting gradual transitions, there is a method called the "Twin-Comparison", where two thresholds are used instead of one. In addition to the higher threshold T_b, a second, much lower one, T_S is used too. If a frame-to-frame difference, calculated with the equation (2.1), goes over the T_S threshold, then the frame is marked (F_s) and the next differences are continuously summed up. The accumulation of the differences continues until a single frame-to-frame difference drops below the previous T_S threshold and is marked as F_E.

If up to that point, the accumulated sum of differences has gone over the higher T_b threshold, then a gradual transition is said to be found. Otherwise, if a set number of too many frame differences has been summed up or the sum of differences has not reached T_b, the starting search point is dropped, and the search continues as normal. An example visualization of this process can be seen in Figure 13.

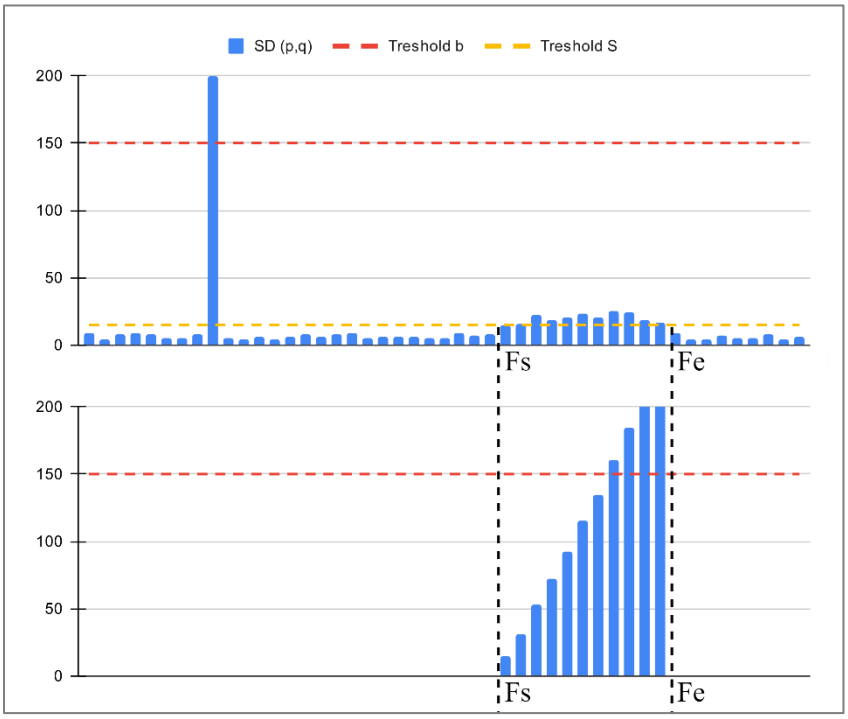


Figure 13: In the top diagram, a sequence of histogram differences can be seen with a camera cut and a gradual transition (Frames Fs to Fe). In the bottom diagram, the accumulated sum of differences that surpass the T_S threshold can be seen, where the sum surpassed the T_b threshold. The red dashed line is the T_b threshold and the yellow dashed line the T_S

3. Architecture

The system consists of three main parts. The clients, the Kafka Broker and our Flink Cluster. The clients read a video and extract its frames. Each frame is split into smaller parts, called blocks. Each block along with a key used to identify it, are sent to Kafka as a message. The messages are evenly distributed into multiple topic partitions. The Flink Cluster consumes these messages from Kafka and processes them. It combines them based on their keys and calculates the histogram of each frame. Based on these histograms, potential camera shots are detected.

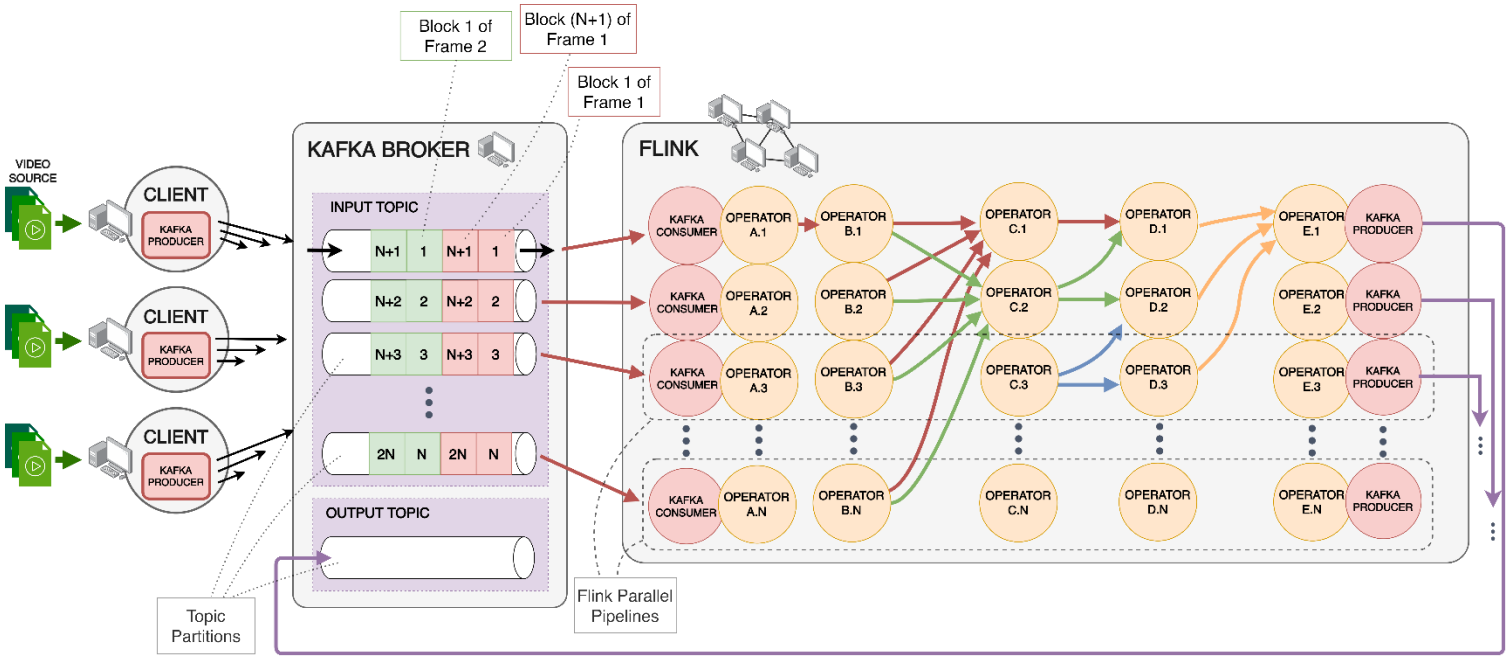


Figure 14: Overall Architecture of our system. On the left are the Clients sending data to the Kafka input topic. In the middle is the Kafka Broker who manages the topics and their partitions. On the right is our Flink Application with N parallel pipelines of 5 operators.

In Figure 14 is the overall architecture of the system. Three Clients can be seen on the left. They send video data to the input topic. The Kafka Broker in the middle manages the two topics (input and output) and their partitions. The input topic has N partitions. Input data from Clients is evenly distributed among the N partitions. On the right, N parallel Flink pipelines request data from the Kafka Broker. Each pipeline has one Kafka Consumer. Each Consumer reads from one partition of the input topic. 5 different Operators transform the input data to detect different shots. If any shot is detected, it is announced to the output topic through each pipelines Kafka Producer. The output topic has one partition.

The main idea of our work is that we split our original video data into smaller parts that can be more easily processed by the Flink Cluster. All the parallel Flink pipelines receive these blocks that contain pixel data. In the first operator they transform the pixel data to grayscale if encoded in RGB. Then the next operator generates the histogram of intensity of each grayscale block it receives. Next, all the histograms generated from all the blocks of a single frame, are gathered to a single operator. The histograms are all summed together to produce the total histogram of that single frame. In the next operator, the difference between the histograms of two adjacent frames is calculated. The final operator receives all the result differences and checks if they pass a predefined threshold T_b. If they do, a shot transition is detected. This operator also searches for potential gradual fade transitions. Keeping in memory some of these differences is required for this process to work. If any kind of shot is detected, it is announced by sending a descriptive message to the output Kafka topic.

3.1 Clients

The clients are a Java application that supply video data to Kafka from video sources. In our work, we use local files as our video source. Multiple clients can exist and send video data to Kafka at the same time.

3.1.1 Client Workflow

First, the video's resolution is determined through the FFmpeg tool. At this point we set the size of the blocks. This will determine the final size of the messages sent to Kafka. Each block can be a fixed size (e.g., 200KB) or equal a number of the frame's row. The optimal size of blocks is determined through experiments (see Chapter 4). The video is read through the Ffmpeg tool as a stream, each time reading as many bytes as the size of a single block. A unique identification key is generated for each video and block. The raw bytes of the block with the key are sent to Kafka as a single message.

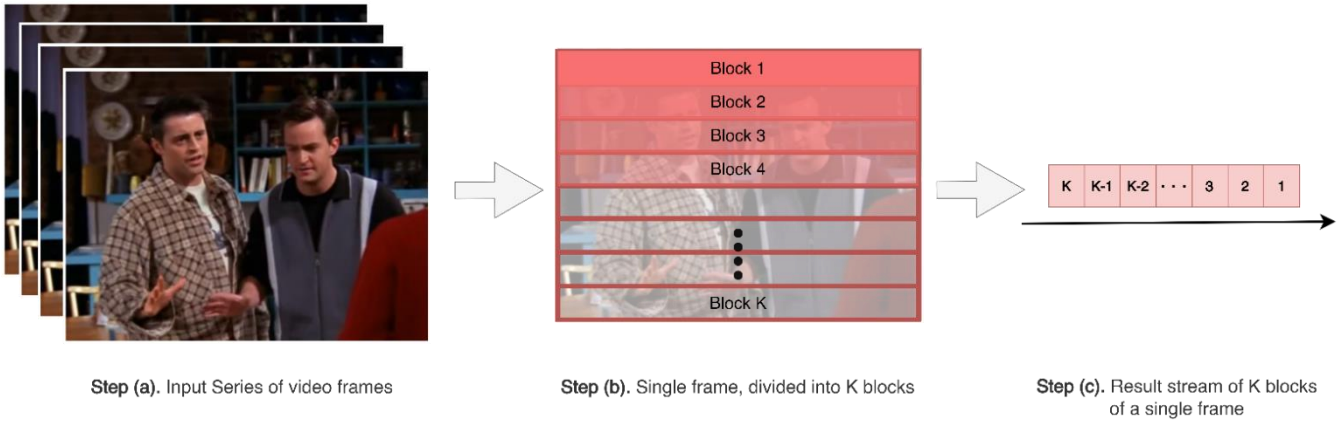


Figure 15: How each frame is split into K blocks. The result is a stream of blocks that will be sent to Kafka. Each block shown in this figure (step b), has a size that is a multiple of the frame's row (e.g., 1 block = 15 rows of the frame).

All the messages are evenly distributed across $\boldsymbol{N}$ partitions of the Kafka input topic. This is ensured by the Kafka Producer on the Clients, that distribute the messages in a Round Robin approach. The Kafka Producer queries the Kafka Broker during runtime to find out the current number of partitions of the input topic (N). The messages are evenly distributed in N partitions so they can be also evenly consumed by N parallel Flink Operators.

Figure 17 is a visual representation of how the blocks of each frame are distributed. For this example, the input topic has 4 partitions and each frame is split into 8 blocks. In the case of a single Client, blocks from multiple frames will be placed in order per partition, as shown in step b of Figure 17. With multiple Clients, the blocks of each specific video will still be in order inside each partition. However, blocks from multiple different videos will be shuffled together. This is shown in step b of Figure 16, where the blocks of the same video (red and green) are in order inside each partition. The blocks from the different video (blue blocks) are placed between the red and green blocks, in the order they arrive.

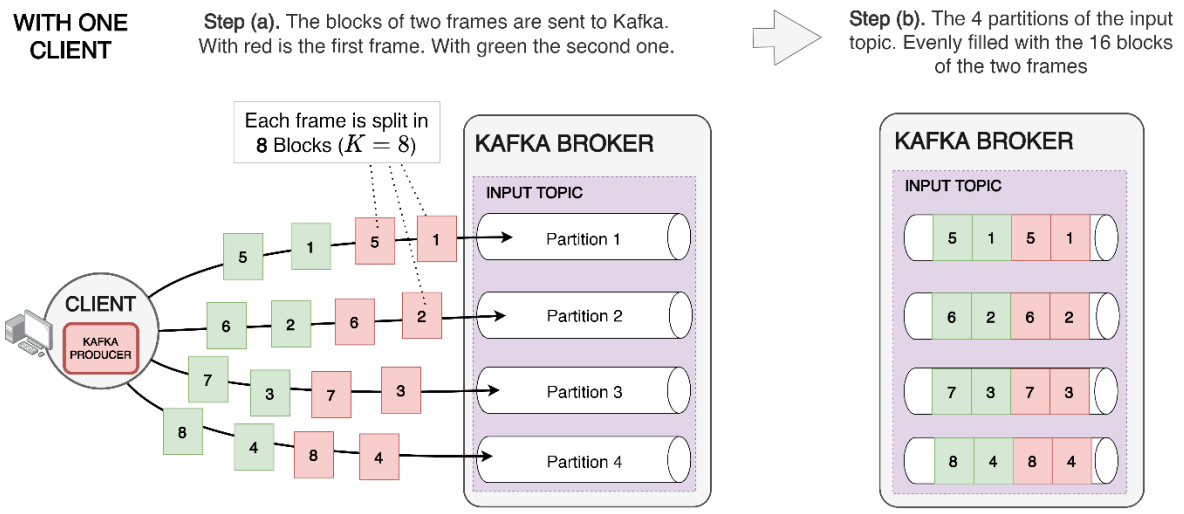


Figure 17: How blocks are distributed inside the input topic partitions with one client. In this example: Both frames are split into 8 blocks ($K = 8$). The input topic has 4 partitions ($N = 4$).

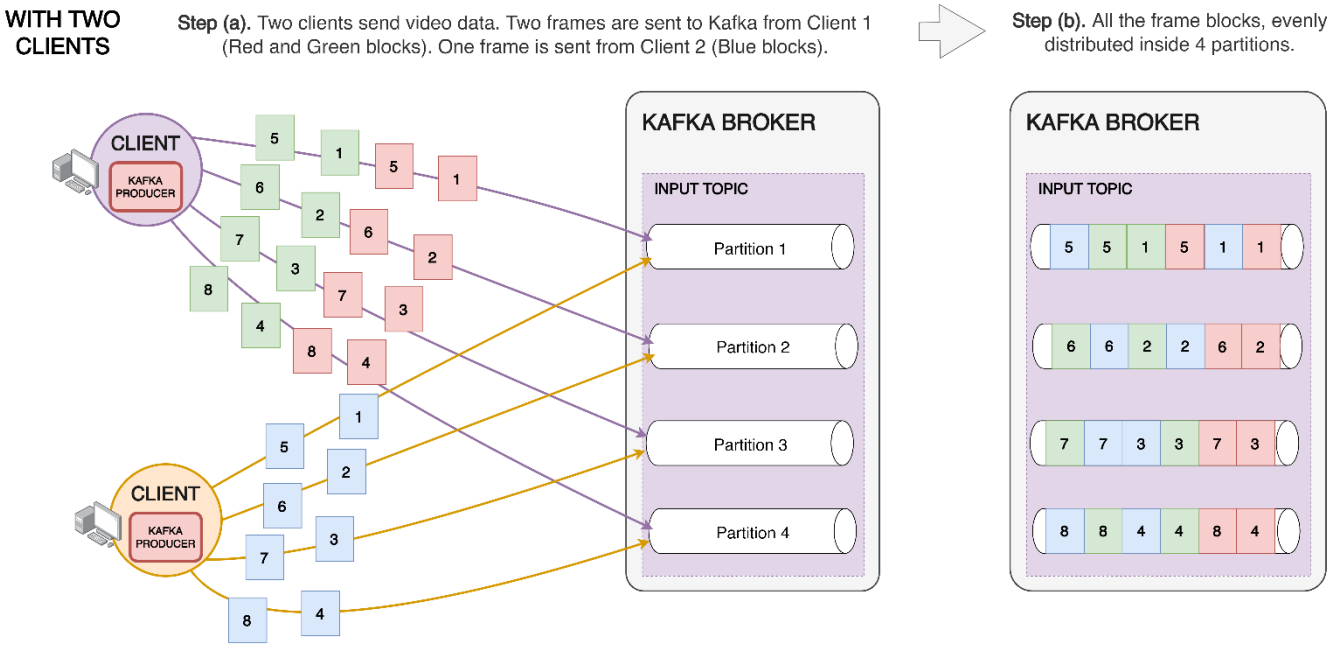


*Figure 16: How blocks are distributed in the input topic partitions with two Clients. In this example: All frames are split into 8 blocks ($K = 8$). The input topic has 4 partitions ($N = 4$). The blocks of **each video** are still in order inside each partition (as seen in step b).*

3.1.2 Key generation for Blocks

As shown in the previous figures, multiple clients can send different videos to be processed at the same time. All blocks of all the different videos are sent to the input topic, distributed into the N partitions of that topic. A way to distinguish between all blocks is required.

For that reason, an id is created for each unique video. Each block is assigned a key that contains the id of their origin video. In addition, on that key is also added an id of the origin frame and one for the block itself. In other words, each block is assigned a key that helps identify from which video and frame it is from as well as its own block id. Other characteristics of the video are also added to the key: the video's resolution, how it is encoded and in how many blocks each frame was split.

The final key of each block is a concatenation of the three IDs and the additional information. With this key, the Flink Operators will know exactly how to process and redistribute each block between the parallel pipelines.

An example block key is seen in Figure 18. It contains the unique id of its origin video (*a* and *b*). It also contains the id of the block itself (*h*) and the frame it came from (*g*). Component (*a*) is the time when this specific video started being read by the Client (in milliseconds since the Unix epoch[12]). Component (*b*) is a random integer between 0 and 10,000. Together, (*a*) and (b*)* compose a unique id for each video. (*c*) is the resolution and (*d*) is the total number of frames. Block size is defined in (*e*). It indicates how many rows each block contains. With the information of the frame's height, we can also deduct the number of blocks per frame. Part (*f*) of the key represents the encoding of the pixel data; 0 is for grayscale, 1 for RGB color.

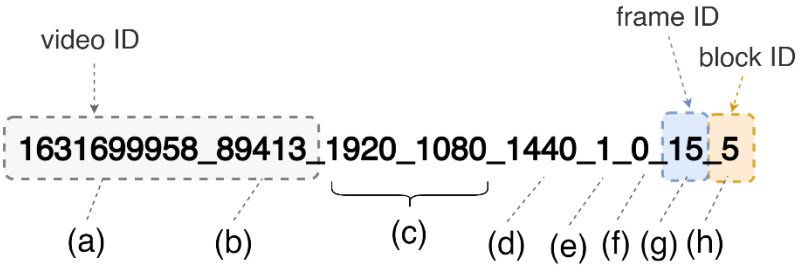


Figure 18: Example key with its various components. In this example, it's the key for the 5th block (h) of the 15th frame(g) from video with id (a)_(b).

Building up on the example from Figure 16, in Figure 19 we see some of the assigned keys on specific blocks. Three sets of blocks are shown. The red and green blocks are from the 1st and 2nd frame respectively of the 1st video. The blue blocks are part of the 1st frame of the 2nd video.

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

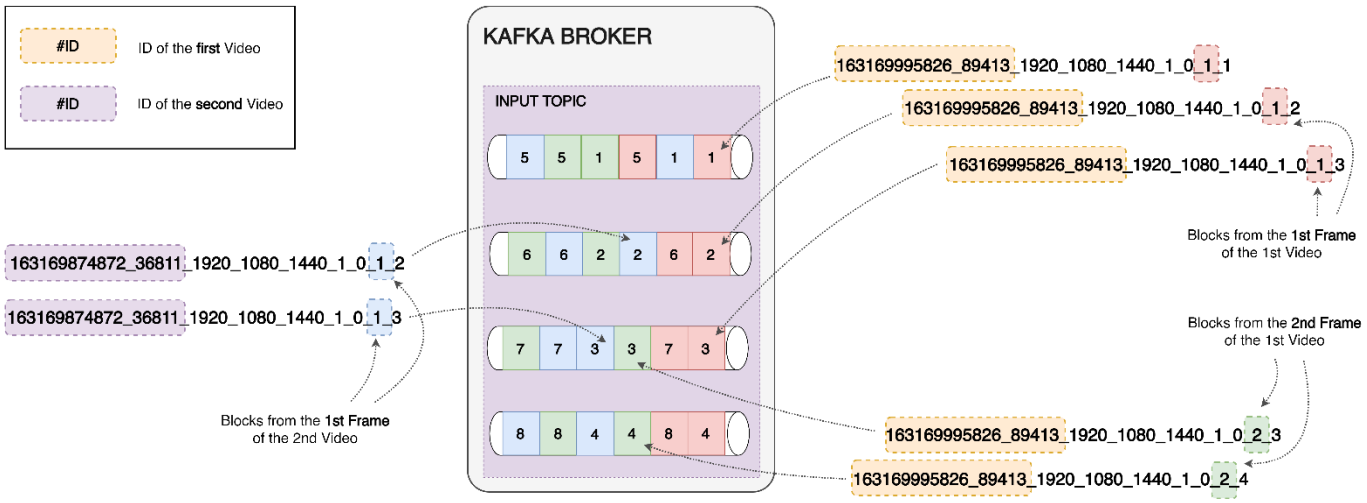


Figure 19: Example scenario with blocks from 2 videos. Red and green blocks are part of the 1st and 2nd frame respectively of the 1st video. The blue blocks are part of the 1st frame of the 2nd video. Some keys assigned to specific blocks are shown.

3.1.3 Video File Format

Our Flink Application can process raw pixel data that is either grayscale or in color format. Therefore, we needed the encoding of our input video files to directly contain that information (i.e., either raw grayscale or raw color frames). For this reason, we used video files with either raw RGB (RGB888) or YUV (YUV420p) encodings[13]. Videos of RGB encoding contain the raw red, green, and blue data of each pixel. Whereas the YUV encoding contains the raw grayscale pixel data (i.e., pixel brightness), separate from the color data (chroma components). By using either of these two formats, the clients do no processing on the input video file. Instead, they simply read the video file and send the video data to our Kafka input topic.

With RGB888 we transfer 24 bits per pixel (8 bits per color channel). The first Flink operator will then transform it to grayscale data. However, with the YUV420p encoding, the Y' component of each frame is exactly the grayscale frame we need. The client simply reads the grayscale frame and sends it to Kafka (8 bits per pixel). The U and V components of each frame are skipped from the file stream. The YUV encoding reduces the bandwidth used compared to the RGB, since we send only 8 bits of grayscale data instead of 24 bits of color data. It also saves processing time on the Flink Cluster, since no grayscale conversion needs to happen.

3.1.3 Optimization Parameters

Block Size

A frame of a high-resolution video in raw color or even grayscale encoding has a significant size. For a color frame (24 bits per pixel, 8 bits per color channel) of 2560x1440 resolution, its total size is:

$$(2560 * 1440)\ pixels * 24\frac{bits}{pixels} = 11059200\ Bytes \cong \mathbf{10.54\ MBytes}$$

For a grayscale frame, it is exactly one third of the previous value at **3.5MB**, since only one channel of 8 bits is used (only the brightness/intensity of each pixel). However, we can split this same frame in multiple blocks. If each block is the size of a single row, this will result in 1440 blocks, all of which

[13] Their complete definitions can be seen in Chapter 2

containing 2560 pixels. Each block will be either $2560 * 24\,Bits = \mathbf{7.5\,KBytes}$ or $2560 * 8\,bits = \mathbf{2.5KBytes}$, for color and grayscale frames respectively.

Initially we set the block size to be equal to a row. However, our system supports the use of different block sizes for different video resolutions. For example, we can set higher resolution videos to use blocks the size of a single row, while lower resolution videos can use blocks that contain the data of 20 rows. This can allow all blocks to be the same size, regardless of the video resolution. At chapter 4 we experiment with different block sizes to determine how they impact performance.

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. Smaller blocks of data obviously require less processing power to be processed. 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. Alternatively, with weaker nodes, a much smaller size can be configured. This allows us to use the Flink Cluster and its resources optimally.

Acknowledgements

Kafka Producers have three possible levels of acknowledgements from the Kafka Brokers. These acknowledgements notify the Producer that their messages where successfully sent. We use the lowest level, where the Producer does not wait for any acknowledgement from the Broker. The messages are sent asynchronously and are immediately considered sent. This improves the latency of sending it to the Broker. However, this way the Producers are not notified in case of an error with the Kafka Broker.

3.2 Kafka Broker

The Kafka Broker receives all the messages from the Clients and stores them in the partitions of the input topic. The Broker is also responsible for providing the Kafka consumers of the Flink Application with messages when they request for more.

Two topics are used. A topic for the input data and another one for the output data, as seen in Figure 14. The Clients use the input topic to send the video data blocks. These are consumed by the first Flink Operator. The last Flink Operator announces through the output topic any shots it detected. These announcements can be consumed by the end-user. The output topic uses only one partition. The input topic has as many partitions as the number of parallel Flink pipelines. This is required to provide each pipeline with its own set of data to process. Using multiple Kafka consumers with multiple partitions also greatly improves Kafka's overall throughput, compared to a single partition and a single consumer.

3.3 Flink

3.3.1 Flink Pipeline Design

The Flink pipeline uses as input raw video data and outputs any potential shot detections. The input video data is partitioned into multiple small blocks. Each block comes with a unique key to identify its origin (i.e., which video and frame it comes from). The pipeline can process both streams and batches of raw video data. Multiple videos can be processed concurrently. They can have different resolutions, encodings or even block sizes. The distinguish between them is achieved through the unique key that the clients generate for each block.

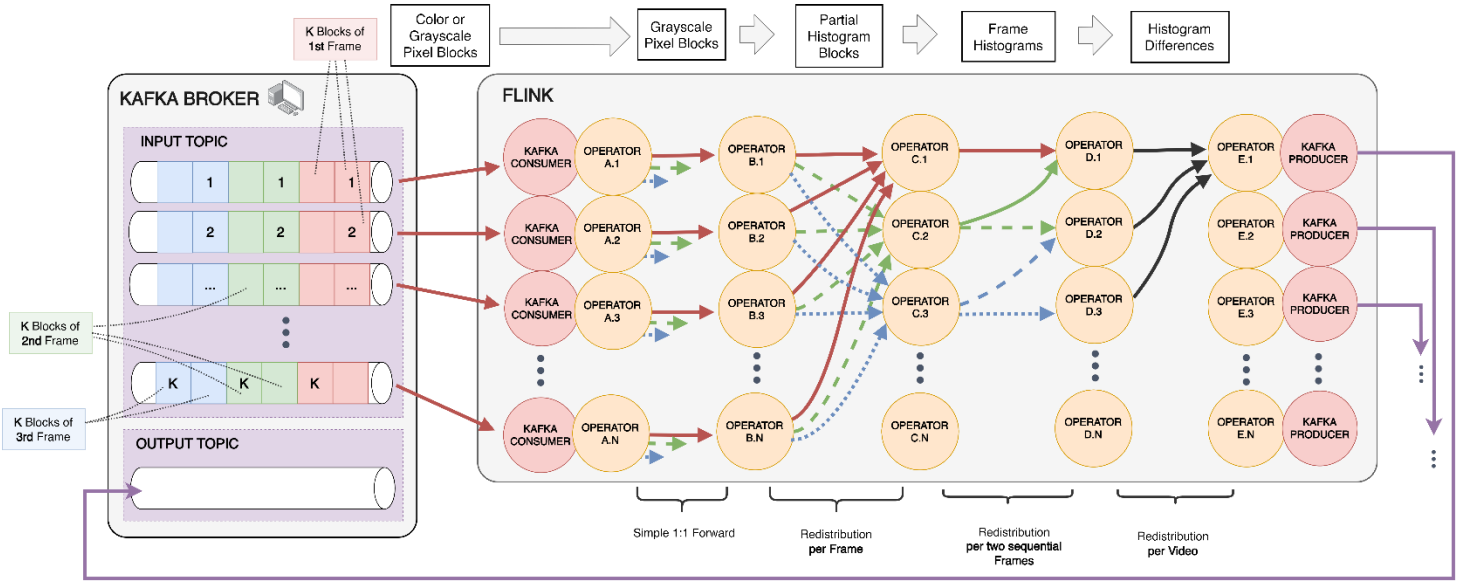


Figure 20: Detailed figure of our Flink Application. As an example, 3 frames from the same video are being processed. Each frame is split into K blocks of red, green or blue color. 5 Operators can be seen with a parallelism of N. At the top of the figure is the chain of data transformations of all operators. The arrows show how the data of each frame flows through the Operators.

In Figure 20 a more detailed view of our Flink Application can be seen. As an example, 3 Frames of the same video are seen in the Kafka Broker, split into *K* blocks each and evenly distributed in the *N* partitions of the input topic. In our Flink Application, 5 different operators can be seen (A, B, C, D, E). The parallelism is *N*, therefore *N* copies (*instances*) of each operator exist. Each Operator A is coupled with a Kafka Consumer that consumes from a partition of the input topic. Therefore, providing operator A with input data. Each Operator E is coupled with a Kafka Producer that produces the application's output messages to the output topic.

The specific transformation that each Operator does to the input data can be seen at the top (e.g., "Grayscale Pixel blocks" to "Block Histograms" for operator B). The small arrows between the operators represent the data of each frame. Red, green, blue for the 1st, 2nd and 3rd frame respectively. The arrows also indicate how and where redistribution of data happens (e.g., Per frame redistribution between operator B and C). The black arrows at the end represent histogram differences of this specific video. The purple arrows are descriptive messages of detected shots, sent to the output topic of our Kafka Broker.

Parallelism

A pipeline is a single set of instances of different operators (e.g., A.1→B.1→C.1→D.1→E.1). Parallelism is the number of parallel pipelines that exist. The parallelism can be freely adjusted. Our Flink Application can run with only one pipeline or multiple. For each pipeline we have we also need another partition for our input Kafka topic.

All operators execute their task only by using the data they receive as input. This means they do not request or send data to an external database or between parallel operators. They either process each input data immediately (e.g., Operator A and B) and provide an output, or they wait for multiple input data (e.g., Operators B, C, D). Some operators make use of a local on-memory keyed state.

The optimal parallelism of the system is calculated through experiments. In these experiments we increase the number of parallel pipelines until we no longer observe any performance increase. A performance increase is considered when a specific video is processed faster than the previous experiment run. We run tests with one Client only (i.e., one video input) and multiple Clients (i.e., multiple concurrent video inputs). These experiments are described in depth in chapter 4.

Keys and Keyed State

In Flink, a normal stream of data (e.g., the output of all parallel instances of Operator B) can be transformed to a **keyed stream** based on a specific **key** (e.g., each frame's unique id). This is used to group the related results of all the previous operator instances (e.g., all partial histograms of a frame). Each group of related results will be processed together on the same instance of the next operator. For each specific key, an associated state is created. This state of each key is entirely stored on a single operator instance. All output of the previous operator with the same key will be redirected to that same operator instance.

In Figure 21, an example data flow between operator B and operator C can be seen. Operator B outputs partial histograms. Operator C receives the partial histograms of each frame and outputs the full histogram. In the figure, the partial histograms of frame "1" of video "1" are all sent to the instance C.1 of Operator C. On that operator instance, the partial histograms are summed up. The full histogram is therefore calculated and sent to the next Operator. In order to sum the partial histograms that will arrive at different times, that key's *associated state* is used to store them locally, in memory.

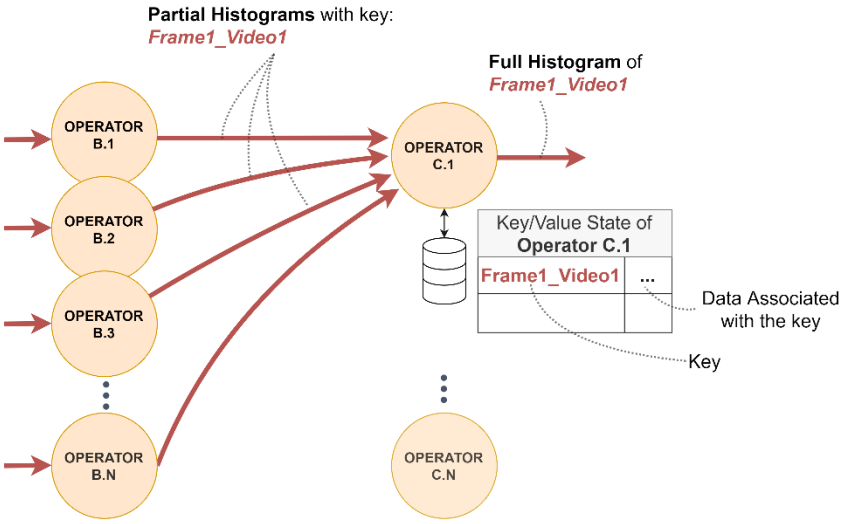


Figure 21:A key is used to group related data and process them together. The operator's key/value state is always local and in-memory.

A more complete visual example of that is seen in Figure 22, showing multiple data with different keys. Both operators have a parallelism of 2 (i.e., 2 operator instances each). Data from 4 frames is redistributed among the two operator instances, based on the frame and video id. Both data blocks for frame 1 (red solid arrows) are assigned the same key of "Frame1_Video1". As a result, they will all be sent to the same operator and share a local state. The same is true for the rest of the data from the other frames.

The associated states of these 4 keys are evenly distributed among the two parallel operator instances. We have **4 keyed states** and **2 operator instances**. Thus, each instance will hold 2 keyed states. This can be seen on each operator's local Key/Value total state on the right of the figure. From the operator's perspective, each time a data block with a specific key arrives, the operator uses that key's state, ignoring everything else. In the figure is also shown data for the 5th and 6th frames of the same video and the 1st and 2nd frame of a second video in the two Key/Value states.

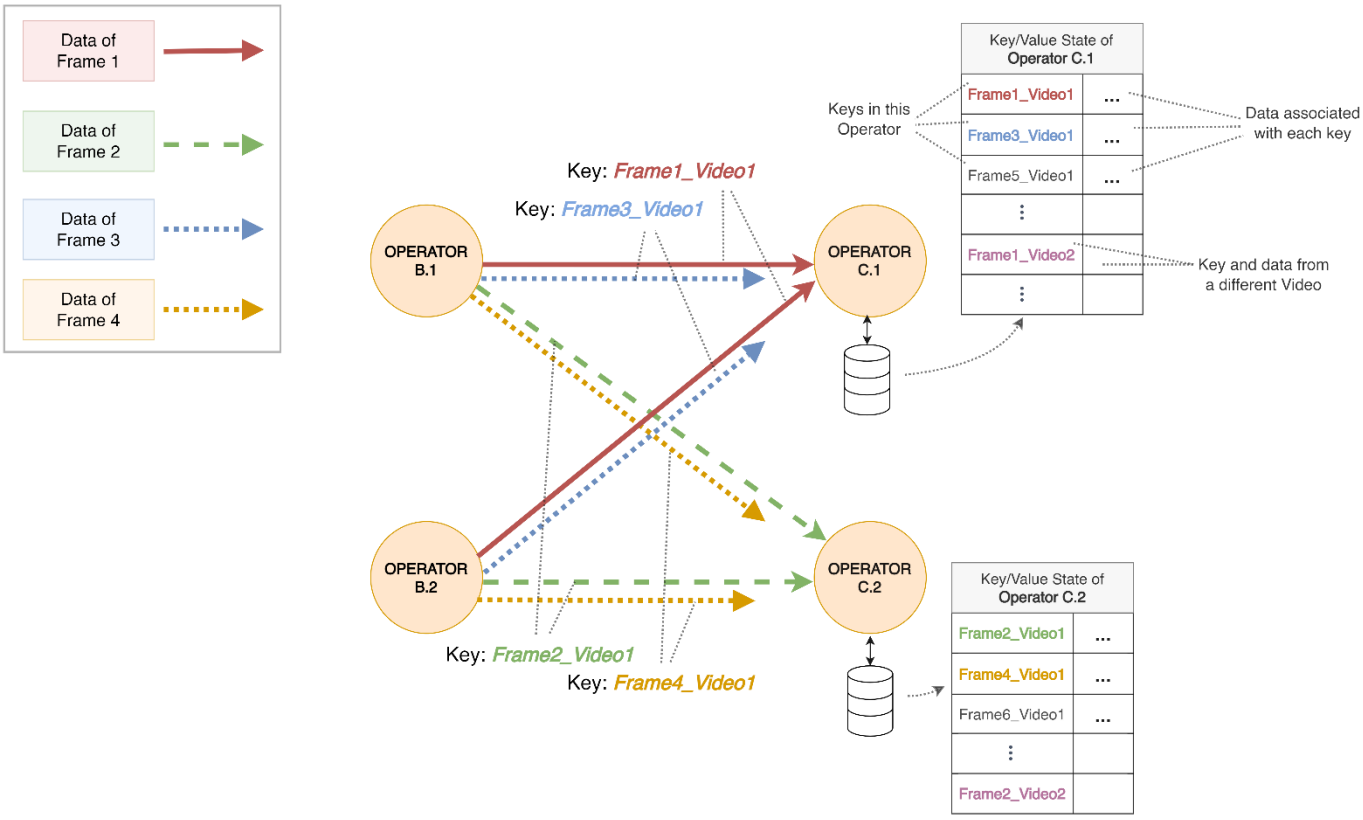


Figure 22: Example use case of keys and keyed states. The operators have a parallelism of 2. Each type of arrow represent data with the same key (e.g., red solid arrows are data of frame 1 with key Frame1_Video1). Each operators Key/Value state is shown next to it. The Key/Value states are local to each operator instance and stored in memory.

3.3.2 Operators

Our Flink Job consists of 5 operators plus a Source Operator (Kafka Consumer) and a Sink Operator (Kafka Producer). The Kafka Consumer is used for data acquisition from the input topic and the Producer for the shot change announcements. Each operator can be thought of as a single transformation. The overall actions of these operators are equivalent to the shot detection algorithm described in Chapter 2.5.

Each operator has the same degree of parallelism, as shown in Figure 20. A typical parallelism value for our work was 6. In other words, each operator had 6 instances that all worked in parallel. We also set the number of partitions for the input Kafka topic to the same (or higher) value.

Source Operator / Kafka Consumer

The source operator consists of a Kafka Consumer who requests messages from the input topic. Each Consumer is automatically assigned a specific partition. No other consumer can use the same partition. The messages from the input topic contain the block pixel data and that block's unique key. The different key components are parsed in order to be used by the operators. The key and the raw pixel data are sent to the first operator.

Operator A – RGB to Grayscale Conversion

If the input block pixel data is in color (24 bits per pixel), it will be converted to grayscale (8 bits per pixel) in this step. Otherwise, it is simply forwarded to the next operator unchanged. The Operator recognizes if it's in color or in grayscale through the block's key. It is a 1 to 1 map function (i.e., One input, One output).

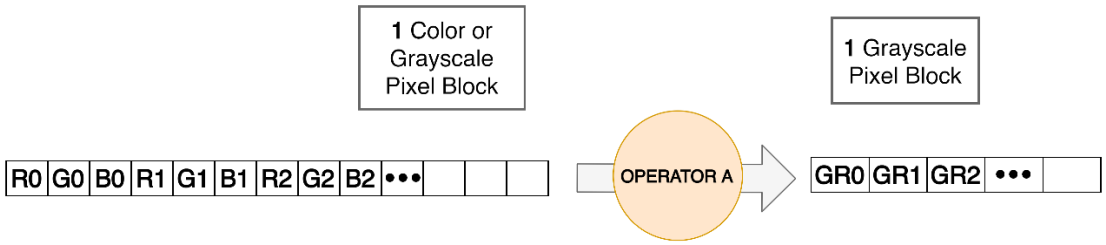


Figure 23: Operator A input and output. On the input array, R0, G0, B0 are the color channel data for pixel 0 (24 bits total). On the output array, the corresponding grayscale value "GR0" is calculated (8 bits).

In Figure 24, a detailed view of Operator A can be seen (with an example N blocks of a frame). Each block that arrives to the Kafka Consumer from the input topic is sent to that pipeline's Operator A. Each Operator A converts each input block if needed. Therefore, each operator B always receives a grayscale pixel block. Operator A only uses the key of each block to find out if it's in color or grayscale and its dimensions. It ignores which frame or video each block is from.

Each operator instance of operator A will process K/N blocks of data per frame for each video. K is the number of blocks that each frame is divided in and N is the parallelism of the operator. The same number of grayscale blocks will be outputted from each operator instance.

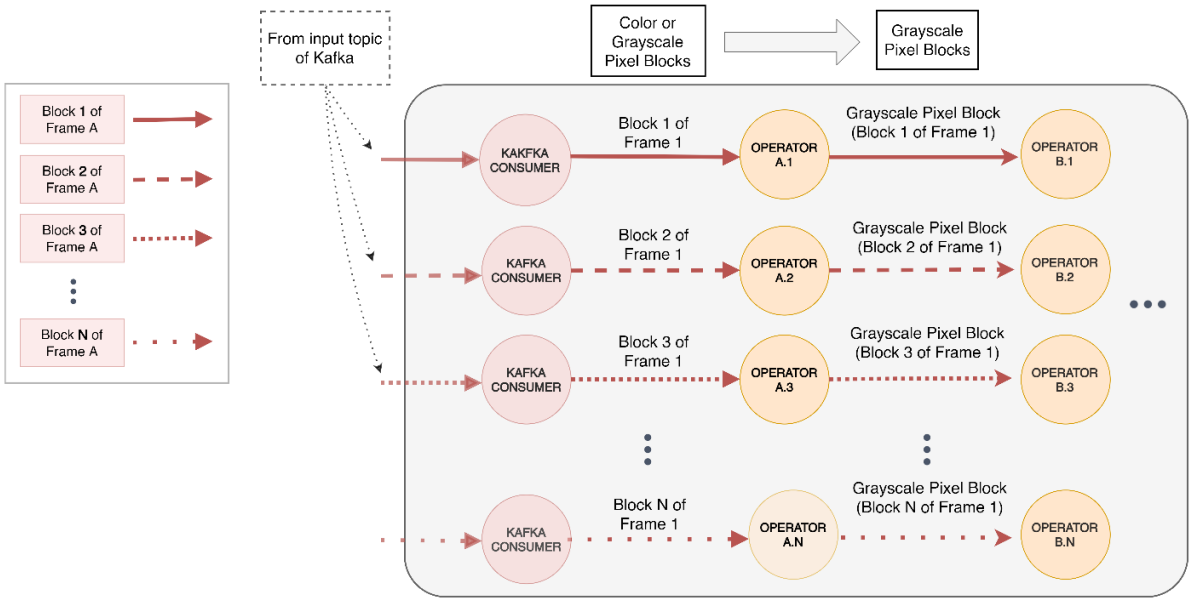


Figure 24: Detailed graph of operator A. Each Kafka consumer reads a block from the input topic and sends it to Operator A of the same pipeline. Each Operator A converts the pixels in the block to grayscale, if in color.

Operator B – Partial Frame Histogram Creation

This operator receives blocks with grayscale pixel data. For each input block, a histogram of intensity of those pixels is the output. In other words, a partial histogram of the source frame is generated. It is a 1 to 1 map function.

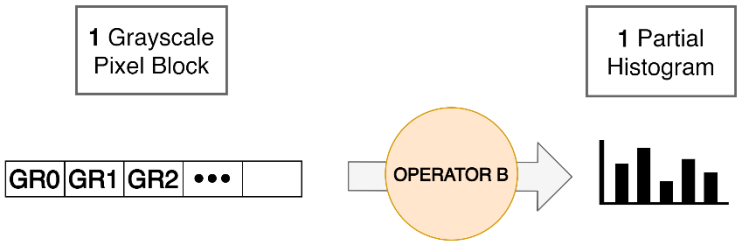


Figure 25: Operator B input and output.

As shown in Figure 26, the input of this operator is simply forwarded from the previous Operator of the same pipeline. However, the output data is redistributed across Operators C, based on the source frame of each block. This is accomplished by using the **frame and video id** of each block as a key, with Flink's keyed state.

Like operator A, each operator instance of operator B will process K/N blocks of data, per frame for each video and output the same number of partial histograms.

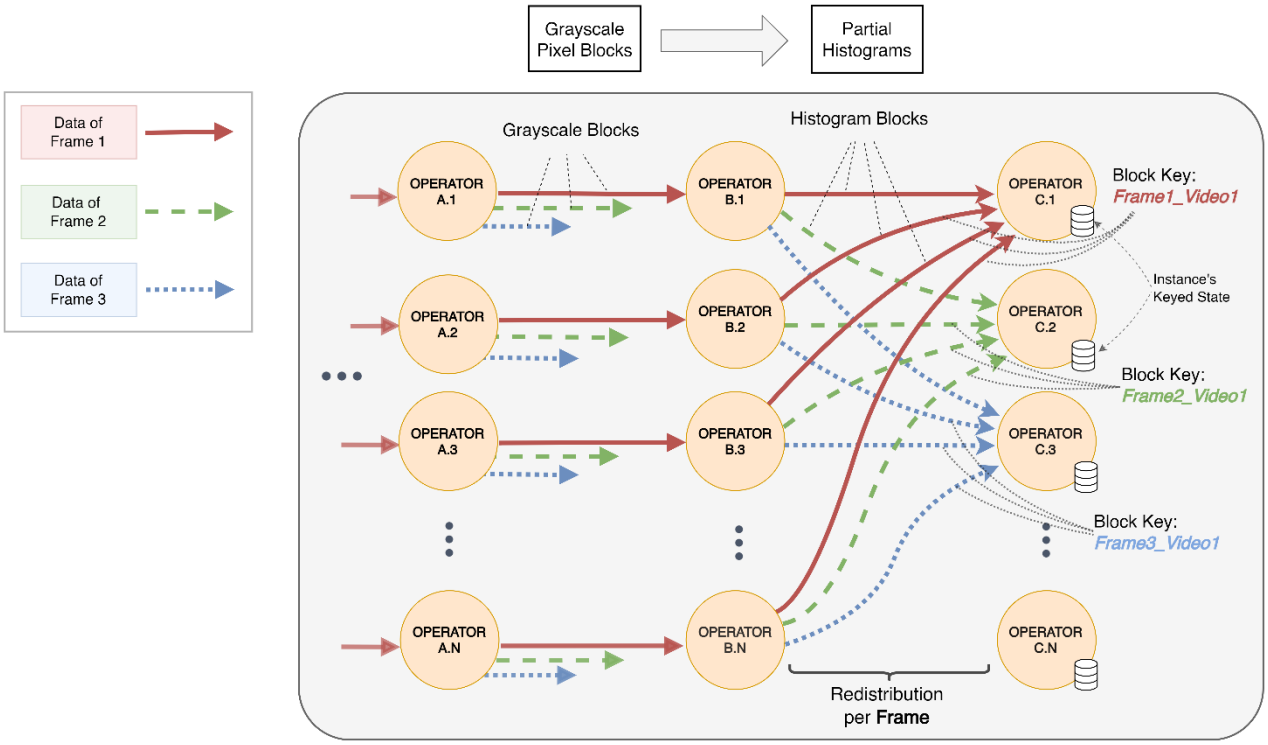


Figure 26: Detailed view of Operator B. Input data (Grayscale pixel blocks) is simply forwarded from operator A. Output data (Partial Histograms) is redistributed based on the frame ID.

Chained Operators: Operator A, operator B and the Source Kafka Consumer are *chained* together (see Figure 27). Operators of the same pipeline that are a 1 to 1 map functions can be "chained" together and co-located in the same thread. Their output data is immediately forwarded to the next chained operator. Between these operators, no redistribution to remote operators occurs. This increases the performance and latency of these operators, by reducing the communication overhead to the next operators.

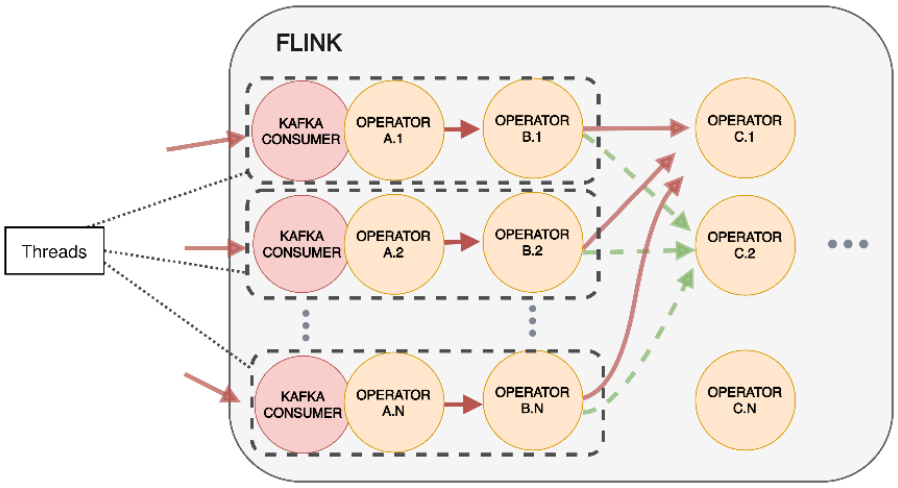


Figure 27: Chained operators share the same thread, reducing latency and increasing performance. These threads can exist in entirely different worker machines.

Operator C –Full Frame Histogram Creation

The operator receives as input the histograms of all the blocks of a frame. The operator sums up all the partial histograms and generates the total histogram of that frame. If a frame is split into *K* blocks, then the operator will wait for *K* blocks and then output the result histogram. The result histogram is sent to the next operator.

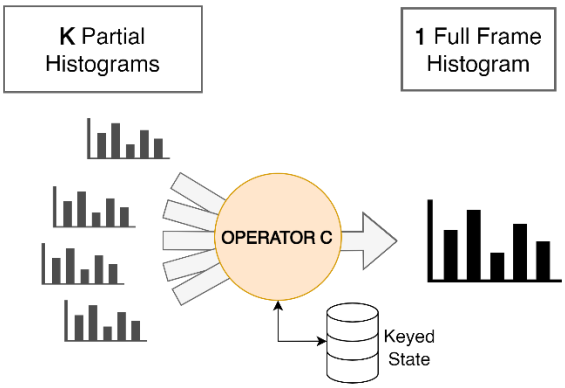


Figure 28: Operator C inputs and output. It receives K partial histograms and outputs a full frame histogram. K is the number of blocks that the original frame was divided in.

For this operator we create a keyed state for each unique frame. In other words, each single frame from any number of videos has its own **unique key** and **associated keyed state**. All the blocks of each frame are sent to the specific operator that keeps that frame's state. Inside each frame's state the operators keep the number of blocks that have already arrived for this frame and the up to now result total histogram.

Multiple frames can be processed on the same operator concurrently, even from different videos. Whenever a new partial histogram arrives, the operator uses its corresponding state (e.g., the state assigned to key "Frame10_Video3"). It adds up the partial histogram to the result total one. If *K* blocks have already arrived, the operator outputs the result histogram of that frame.

In Figure 29, we can see the partial histograms of 4 frames being redistributed. Each instance of operator B outputs K/N partial histograms (only 4 shown in the figure). All partial histograms of frame 1 from video 1 are routed to the first instance of operator C (C.1). This is accomplished by having the same key (e.g., "Frame1_Video1"). The same operator instance will output that specific frame's full histogram. The routing of the keyed data to the correct operator instances is handled by Flink. In our work, we simply assign the correct key to each data block.

Each output (full histogram of a frame) is sent out twice, as shown in Figure 29. Each time with a different key in order to calculate the difference between two consecutive frames. This is explained in detail with the next operator.

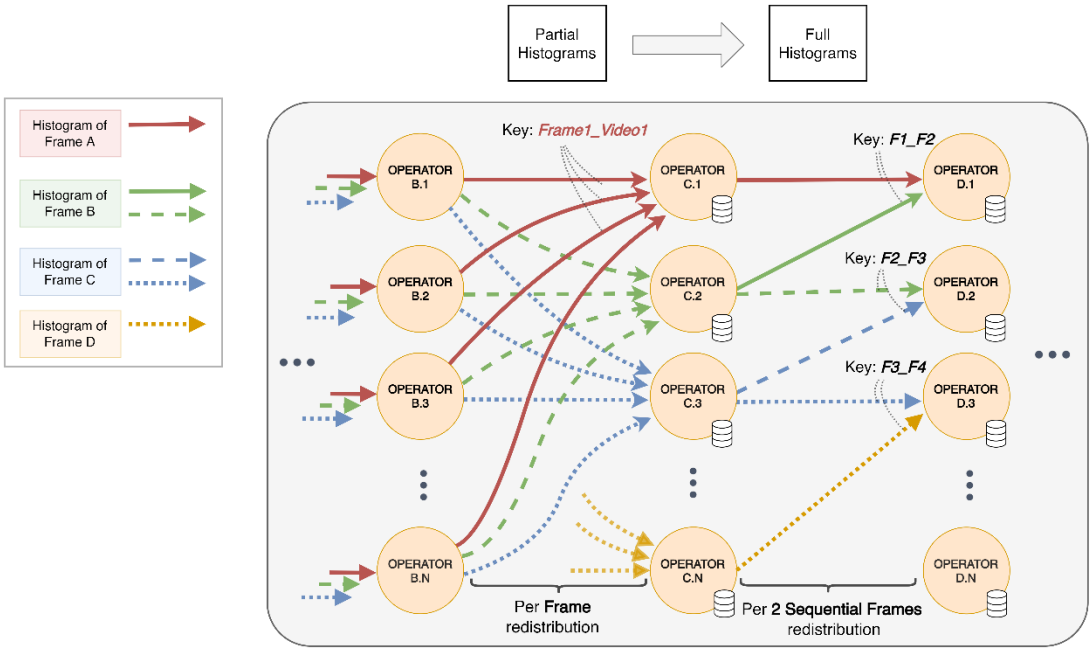


Figure 29: Detailed view of operator C. Each operator instance receives all the partial histograms of specific frames. The output is the full histogram of those frames. Each full histogram of each frame is sent out twice, each time with a different key.

Operator D – Sequential Frames Histogram Difference Calculation

The operator receives the histogram of two consecutive frames. It calculates their difference and outputs the result.

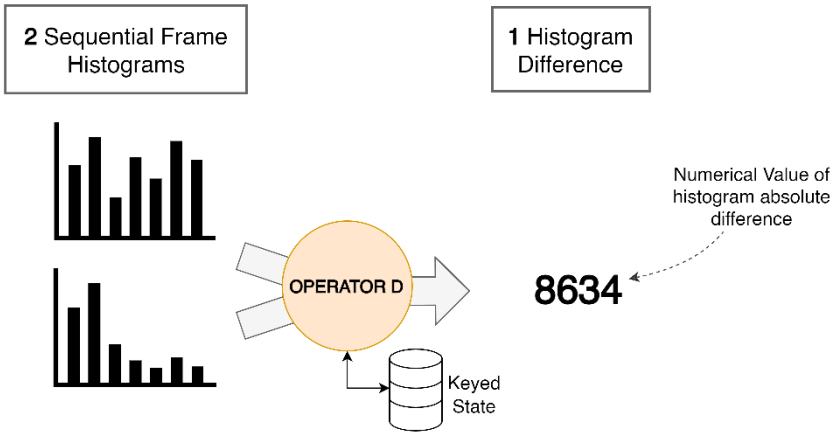


Figure 30: Operator D inputs and output. The difference between each histogram bin is added up to calculate the total numerical value of histogram difference.

In order to achieve this functionality, a key is used for every two sequential frames. For example, for 4 example frames, F1, F2, F3, F4, three keys are created: keys "*F1_F2"*, "*F2_F3"* and "*F3_F4"*. Each key corresponds to the two consecutive frames (e.g., key "*F1_F2"* is about frame F1 and frame F2). For each key, a unique state is created and stored on an operator instance. To distinguish between frames of different videos, the id of each video is also added to each key, as shown in Figure 31.

Therefore, each full frame histogram is outputted twice from the previous operator C. Each time assigned one of the two related key it is related to. For example, the histogram of frame F3 will be sent from operator C twice. Once with the key "F2_F3" and another time with the key "F3_F4". This is shown in Figure 31 with the 2 blue arrows. The histogram of frame F3 and key "F2_F3" will be compared against frame F2 on operator instance D.2. That operator instance will output their difference. Likewise, in instance D.3, the difference between frame F3 and F4 will be calculated.

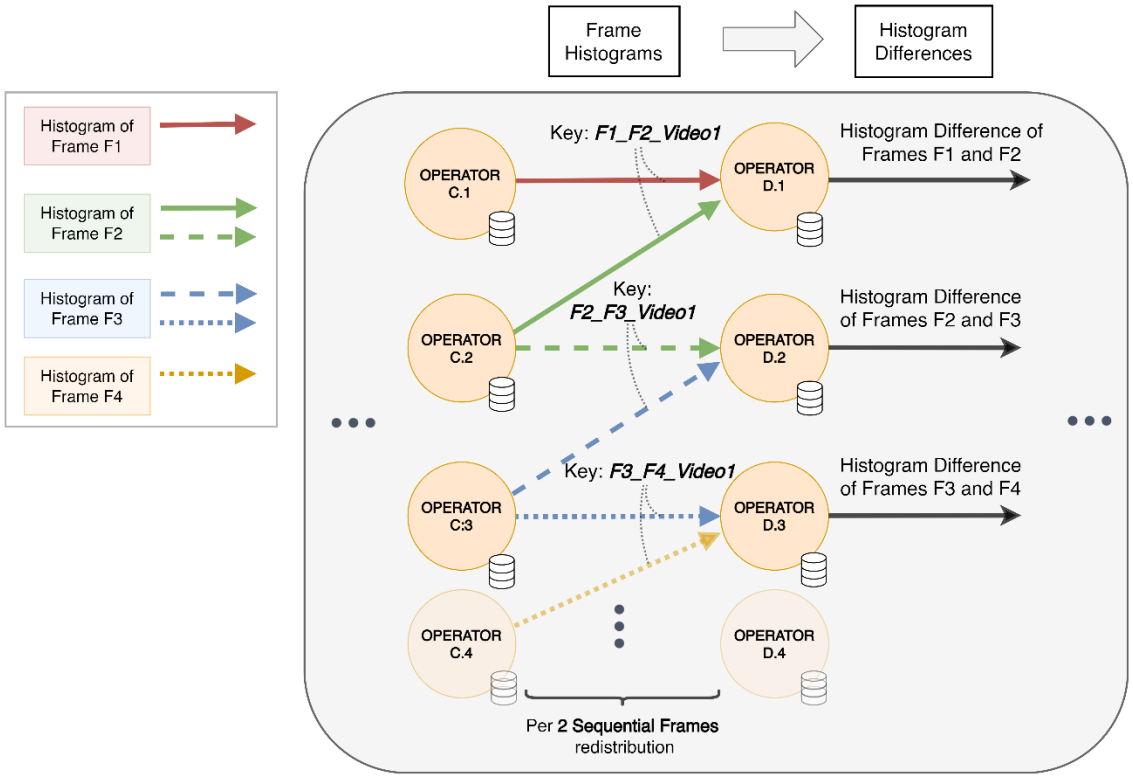


Figure 31: Visualization of how keys are used to properly distribute histograms to operator D.

Each parallel operator instance will hold data of multiple keys (keyed states). For each key, it waits for both histograms of the sequential frames to arrive. When the first histogram arrives, it saves it on the key's local state. When the second histogram arrives, it calculates their difference and outputs the result.

Operator E – Finding Shot changes

The operator receives all histogram differences between all the consecutive frames. If it finds any kind of shot changes (camera cuts or gradual fades) it outputs a descriptive message to the Kafka Producer. A key is also used on this operator, to distinguish between different videos being processed. Each unique video that is being processed by the application has its unique key and keyed state. The histogram differences do not come in complete order.

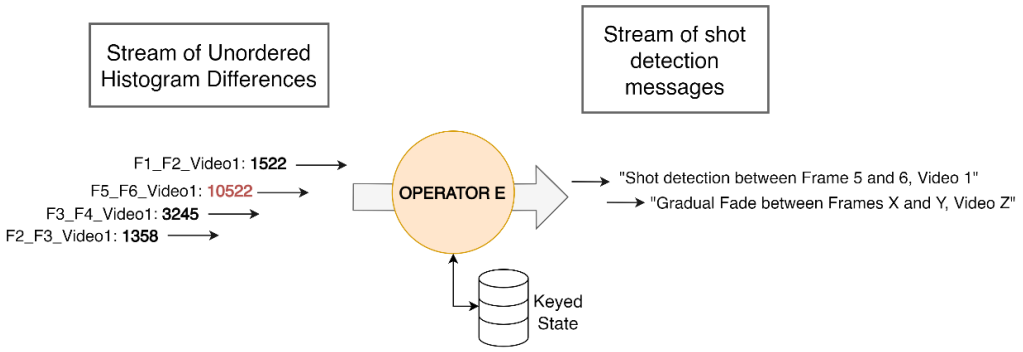


Figure 32: Operator E input and output. The histogram differences of all consecutive frames are the input per video. The operator then looks for camera cuts. Using the local state, it also searches for gradual fades from multiple consecutive frames.

In Figure 33, we can see a detailed view of the operator. The histogram differences are grouped based on the video they come from. This is done by assigning them the same key (e.g., "Video1"). All histogram differences with the same key are routed by Flink to the same operator instance. For example, in the same figure, histogram differences from video 1 (black arrows) are all sent to instance E.1. Likewise, all histogram differences from video 2 are sent to instance E.2 (purple arrows). On each instance, the histogram differences of the same video share a local state. Any shot detection messages from the operator instances are sent to the Kafka Producer. The Kafka Producer sends these messages to the Kafka output topic. The operator instances can process multiple videos concurrently by using the different states that exist for each key (example shown in Figure 34).

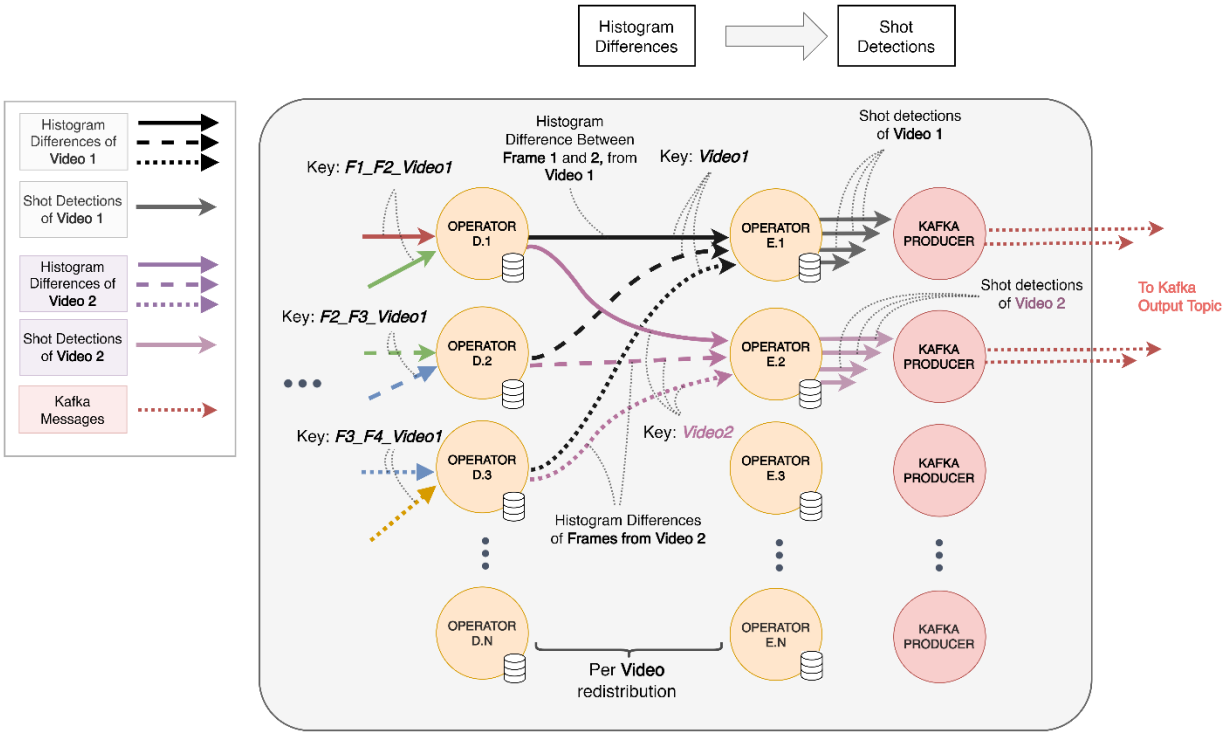


Figure 33: Detailed view of Operator E. The black arrows represent histogram differences of consecutive frames from video 1. They all have the same key: "video1". Likewise, the purple arrows are from video 2 with key "video2". All histogram differences with the same key are routed by Flink to the same operator instance. They are processed together and share a local state. The output shot detection messages are sent to the Kafka output topic, via the Kafka Producer.

In order to detect potential shot changes, two different threshold values are required. T_b (upper) and T_S (lower) which are are calculated beforehand, as described in Chapter 2.6 Video Segmentation.

For the camera cuts, immediately as a histogram difference arrives, it is compared against the T_b upper threshold. If the difference value is above that threshold, a descriptive message is outputted to the next Operator.

For the gradual fade shots, the original sequence of histogram differences needs to be used. The operator checks all histograms differences in their original order. If any of them surpasses the lower threshold T_S, it begins adding up all the histogram differences in the original order. If the sum of the differences reaches the high threshold T_b, a camera fade is considered found. A descriptive message is used as output. The operator stops summing the histogram differences in two cases. Either the next histogram difference is below the lower threshold T_S, or too many histogram differences have been summed up. We set that number of maximum histogram differences to be summed up to 30 frames. That is a little over a second on a 24fps video. Through our experiments, this number provided great results in consistently finding gradual fades and overcoming false positives.

Since the histogram differences are generated without taking into account the order of the frames, a Priority Queue is kept in each key's state, as shown in Figure 34. The Priority Queue of each video holds all the histogram differences that have arrived. At the top (i.e., first element) of the Priority queue is always the earliest available histogram difference. Whenever the next histogram difference is required (by the shot detection algorithm), the operator first looks up the first element of the Priority Queue. If it's in the Priority Queue, it continues with the shot detection algorithm. Otherwise, it stops and waits for the next element to arrive.

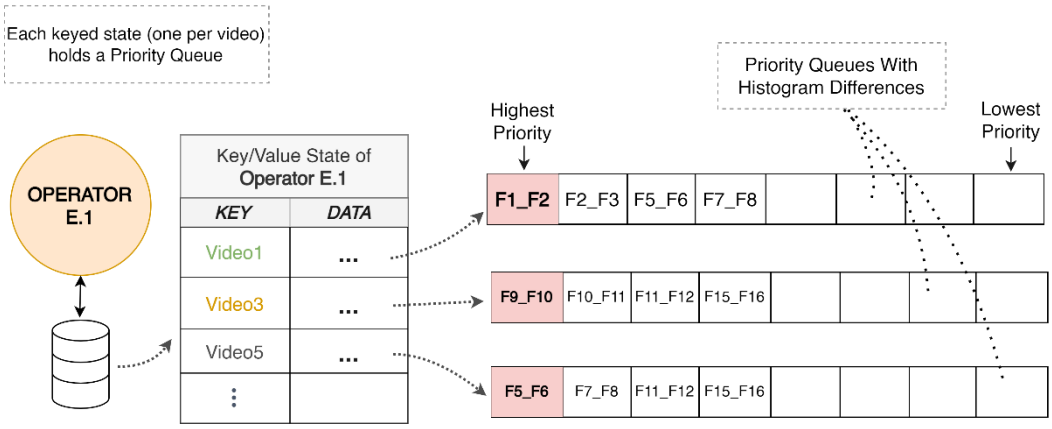


Figure 34: Example Key/Value state of an operator instance. 3 keys with their corresponding data are shown to be stored in the total state. In each key's state, a priority queue is stored. The priority queues hold the histogram differences that have arrived.

Sink Operator / Kafka Producer

At the end of the pipeline, we have a sink to the Kafka output topic. It consists of a Kafka Producer, who receives messages from the previous operator and sends them to Kafka (shown in Figure 14).

Important Note: Some Operators use a keyed state and wait for a specific set of data for each key (e.g., Operator B waits for all blocks of a specific frame, Operator C waits for two specific frames each time). In case of an error where some data may go missing (e.g., the last block of a frame is never sent), each operator has a time limit where it stops waiting for data for that specific key and clears the state associated with that key to prevent wasting memory. However, in normal conditions and our experiments, no errors occurred and no data went missing, mainly due to Kafka's high reliability.

4. Experiments

4.1 Purpose of the experiments

The experiments have two main goals. First, we confirm that the distributed shot detection algorithm works as intended (i.e., detecting the correct number of shots). Then, we compare the Flink Application's performance against a non-distributed algorithm. We expect to see a significant speedup to performance. Lastly, through these experiments, we will determine how well this system scales.

4.2 Systems used

The architecture of the experimental system has two main parts. The first is a Virtual Machine that was used to run the clients and the Kafka Broker. The second one is a Kubernetes Cluster containing the Flink Cluster with our Flink Application. Both of these systems were hosted on the Google Cloud Platform.

4.2.1 Virtual Machine for the Clients and the Kafka Broker

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[14]. For storage, a high-performant HDD was used with a theoretical maximum of above 700MB/s for sequential reads/writes. Ubuntu 16.04 was used for the VM's Operating System.

The VM was used to concurrently run any number of Clients while also hosting the Kafka Broker with multiple topic partitions. Both processes require great sequential I/O performance (i.e., the data is all in one place and read in order) and multiple cores for the parallel workload and multiple I/O threads. As such, the 8 vCPUs and especially the high-performant HDD were a crucial part of the system. A bottleneck at this part of the system was highly undesired.

4.2.2 Kubernetes and Flink Cluster

We initiated a Kubernetes cluster with a pool of up to 8 nodes. Each node had 2 vCPUs (2.0/2.8GHz Base/Turbo Frequency) and 8GB of RAM[15]. For storage, we used the minimum available option of 30GB, since our Flink Application required no local storage. Each node run on a Container-Optimized OS with Containerd provided by Google. The number of nodes was manually managed. The management of Flink Pods (JobManager or TaskManagers pods) was handled by Flink. For initiating a Flink Job (e.g., our Flink Application) we only needed to submit that job to the Flink Cluster along with the desired number of parallel Flink pipelines.

[14]Machine type: c2-standard-8, "*Compute Optimized*"
[15]Machine type: e2-standard-2, "*Cost Optimized*"

4.3 Experiments

In the following Sections 4.3.1 through 4.3.5, the experiments we run are described in detail. Each experiment section contains 4 subsections:

i. *Introduction*: A brief introduction with the aim of the experiment and an expected result.
ii. *Procedure*: A section describing the whole procedure of the experiment
iii. *Results*: The results of the experiments with their explanation
iv. *Discussion*: The analysis of the results.

4.3.1 Distributed Algorithm Correctness

Introduction

In this experiment, we want to confirm the distributed algorithms correctness compared to the regular shot detection algorithm. Our Flink Application should detect and output the correct number of shots (camera cuts or gradual fades) for the input videos.

Procedure

For this test, we used two short-duration videos from a TV show, which we distinguish as "Video A" and "Video B". For both videos, we created four copies of each one with four different resolutions (e.g., "Video_A_1920x1080", "Video_A_2560x1440", etc.). The characteristics of each video and the resolution of each copy can be seen in Table 1.

	Duration	**Total Frames**	**Frames Per Second**	**Encoding**	**Resolutions Used** (Width*Height)
Video A	60 sec	1440	24	YUV420p	960*540, 1280*720, 1920*1080, 2560*1440
Video B	27 sec	648	24	YUV420p	960*540, 1280*720, 1920*1080, 2560*1440

Table 1: Characteristics of videos used in this experiment.

Threshold Calculation: The two threshold values T_b and T_S, required by the shot detection algorithm had to be calculated before the experiments. This was done using a second algorithm described in Chapter 2.6 ("Video Segmentation"), that calculates these values for a specific video. We gave the algorithm Video A as input (once for each of the 4 resolutions) and it output the thresholds T_b and T_S that we can use in our experiments. We can use the same thresholds to detect shot changes for Video B too. This is possible since their content is thematically the same (TV show scenes). The result thresholds can be seen in Table 2 along with the respective links to each clip's original video.

	Link	**Clip timestamp**	**Resolution**	**Result T_b Threshold**	**T_b Threshold (Normalized)**	**T_S Threshold (Normalized)**
Video A	*https://youtu.be/C3Kc9vigrrs*	0:00 to 1:00	960*540	172,930	0.5004	0.15012
			1280*720	459,880	0.4990	0.1497
			1920*1080	1,033,890	0.4986	0.14958
			2560*1440	1,822,556	0.4944	0.14832
Video B	*https://youtu.be/AeKizzQpPpY?t=202*	3:22 to 3:49	-	-	0.5	0.15

Table 2: Result Thresholds used in the following experiments and links to the source videos of the used clips.

The normalized T_b thresholds are calculated by simply dividing the result T_b by the total number of pixels in a frame of the respective resolution. The T_s threshold is a fraction of T_b. We empirically chose a value of 30% of the respective T_b value, since the original algorithm didn't specify a specific ratio. This ratio later proved to be reliable in correctly detecting gradual fades. It is clear that the normalized T_b and T_s values are almost identical. This was expected since the thresholds should not be dependent from the resolution of the video. Different threshold values are instead the result of different types of videos (e.g., TV shows, news, movies). After having taken in account all of the above, we used for all the resolutions and for both Video A and Video B a T_b of 0.5 and a T_S of 0.15 (normalized).

To determine the actual number of shots, we implemented the shot detection algorithm as described in Chapter 2.6 as a regular Java program. Using this program, we get a shot detection count for both videos (Video A and Video B). We then manually counted the shots to confirm this algorithm.

For the Flink Application test, we set the parallelization to four pipelines (i.e., four parallel instances of each operator). We also create four partitions for the Kafka input topic. We then run two Clients concurrently. We give to the first client Video A as input and to the second client we give Video B. The clients send the video data to the Flink Application through Kafka. Our Flink App computes the number of shots in both videos from the two Clients. The Flink App then sends the results to the output topic where we can extract them.

This process is then repeated for all four copies with different resolutions of both videos.

Results

In the table below are the results of all the test runs. The detected shots are displayed per type of shot (camera cuts or gradual fades). Each column is a different method of detection (manual, regular non-distributed Java program, and the Flink application). For our Flink Application, the result of the videos each Client sent is presented. Each row is a different set of experiments, where a copy of the original video with a different resolution is used.

All values per shot type are equal. The Flink application gave the same results for all the different resolutions and for both videos sent by the two clients. These results matched both the manual count and the regular Java program count.

Discussion

Through these results, we see that the distributed algorithm of our Flink Application works as intended. It outputs the current number of shots for all resolutions. It also manages to distinguish video data coming from two clients with different videos simultaneously and give the correct result.

		Camera Cuts Detected				Gradual Fades Detected			
Width	Height	Manual Count	Regula Java Program Count	Flink Application		Manual Count	Regular Java Program Count	Flink Application	
				Video of Client 1	Video of Client 2			Video of Client 1	Video of Client 2
Results of Video A									
960	**540**	14	14	14	14	1	1	1	1
1280	**720**		14	14	14		1	1	1
1920	**1080**		14	14	14		1	1	1
2560	**1440**		14	14	14		1	1	1
Results of Video B									
960	**540**	7	7	7	7	1	1	1	1
1280	**720**		7	7	7		1	1	1
1920	**1080**		7	7	7		1	1	1
2560	**1440**		7	7	7		1	1	1

Table 3: Number of shots detected in Video A and Video B with three different methods.

4.3.2 Speedup to Input Throughput – YUV encoded videos

Introduction

The goal of this experiment is to determine the speedup gained as we increase the input throughput, while the rest of the system's variables remain unchanged. We expect to see the speedup to increase as we increase the input's throughput.

Procedure

First, we determine a list of increasing input throughputs. Each input throughput can derive from any number of clients with videos of the same or even different resolutions. As an example, we can achieve an input throughput of 35.6MB/s with two clients sending a 540p video (17.8MB/s bitrate) each.

We use two different clips of 60 seconds and 24 frames per second each. Both clips have the same bitrate per resolution (see Table 4). On the right of Table 5, we can see the list of input throughputs that we use in the experiment and how each input throughput is achieved. Each checkmark indicates a separate client that sends the respective video with a respective resolution. Thus, for the *first* input throughput, two parallel clients are used. One that sends Video A at a 960x540 resolution and another one that sends Video C at the same resolution, resulting in an input throughput of 35.60 MB/s. For the *third* input throughput, 4 parallel clients are used: 2 clients send Video A and C at a resolution of 960x540 and another 2 clients at a resolution of 1280x720.

	Duration	Total Frames	Frames Per Second	Encoding	Link	Clip timestamp
Video A	60 sec	1440	24	YUV420p	*https://youtu.be/C3Kc9vigrrs*	0:00 to 1:00
Video C	60 sec	1440	24	YUV420p	*https://youtu.be/C3Kc9vigrrs*	1:00 to 2:00

Table 4: Characteristics of Video A and C (YUV encoded)

For each result input throughput, we run an experiment once with parallelization of 1 and once with parallelization of 8. For each of the two runs with different parallelization, we compare the average response time of each video sent. **As response time,** we assume the time passed between when a client starts sending a video and when that video is fully processed (visualized in Figure 35). For both parallelization values, the rest of the system variables remain fixed: 8 Kafka partitions are used for the input topic and a fixed block size of 200KB is used for all resolutions.

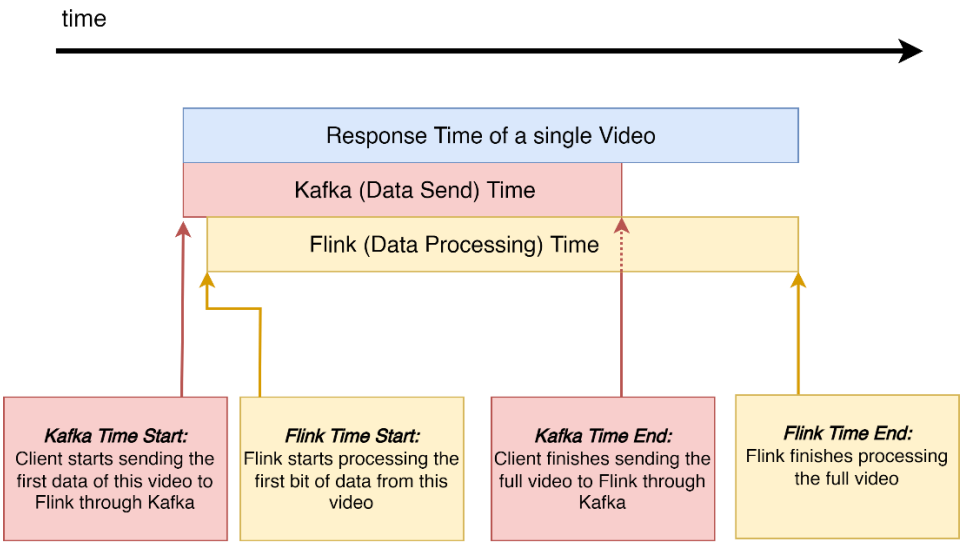


Figure 35: Response Time is the time between sending a video and fully processing it. The slight delay between Flink Start and Kafka Start occurs when Flink is busy processing other videos. The full Kafka time can usually be shorter than the Flink processing time.

	Video A				Video C				
Resolution	960x540	1280x720	1920x1080	2560x1440	960x540	1280x720	1920x1080	2560x1440	
Bitrate (MB/s)	17.80	31.64	71.19	126.56	17.80	31.64	71.19	126.56	**Result Input Throughput (MB/s)**
	✓				✓				**35.60**
		✓				✓			**63.28**
	✓	✓			✓	✓			**98.88**
			✓				✓		**142.38**
	✓		✓		✓		✓		**177.98**
		✓	✓			✓	✓		**205.66**
				✓				✓	**253.13**

Table 5: Input Throughput Matrix (YUV)

Results

In Table 6 we can see the results of this experiment. The first column is the input throughput and the next two columns (2nd and 3rd) display the result Response Time (in seconds) with parallelization of 1 and 8 respectively. The last column is the speedup gained between the two parallelization values. Figure 36 displays the absolute values of the result Response Times and Figure 37 displays the Speedup gained on Response Times after increasing the parallelization from 1 to 8.

Input Throughput (MB/s)	**Response Time (s)**		**Response Time Speedup**
	Par. 1	**Par. 8**	
35.60	47.79	14.72	**3.25**
63.28	71.51	19.60	**3.65**
98.88	142.22	35.18	**4.04**
142.38	166.57	34.88	**4.78**
177.98	234.08	32.57	**7.19**
205.66	254.55	42.08	**6.05**
253.13	329.23	82.66	**3.98**

Table 6: Results of the Speedup to Input Throughput experiment (YUV)

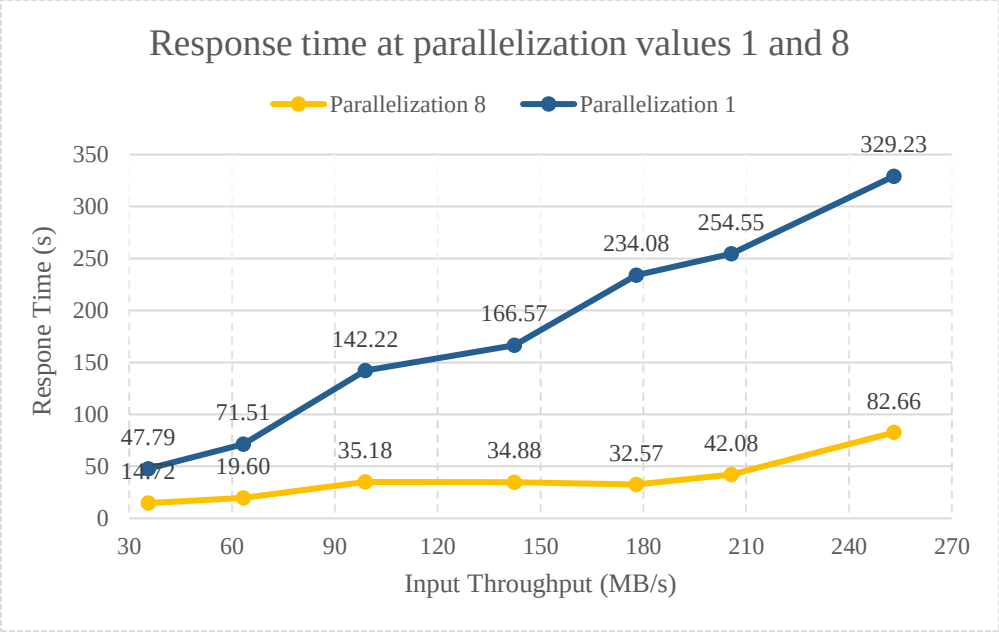


Figure 36: Chart with the result Response Time values (in seconds) of parallelization of 1 (blue line) and 8 (yellow line) (YUV videos).

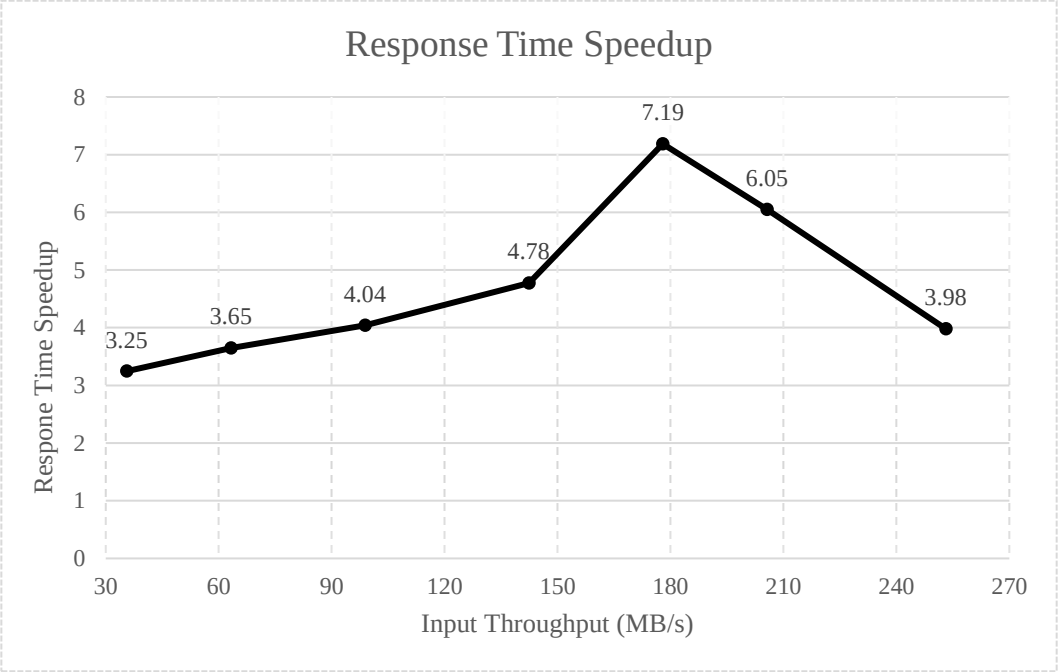


Figure 37: Result Speedup gained on Response Times through increasing parallelization (YUV videos).

Discussion

As we can observe from the results a significant speedup was achieved by increasing the parallelization of our Flink Cluster from 1 to 8. Across all input throughputs we had at least 3 times faster response times with a maximum speedup of 7.19 faster response time. The speedup increases relatively to the input throughput up until a maximum at 178 MB/s input throughput. After this point we see a small decline on the next input throughput and an even steeper decline at the last one.

The decline can be attributed to Kafka's performance. At the highest input throughputs (over 200MB/s) Kafka struggles to send data fast enough to Flink. Increasing the parallelization at this point would not improve the performance of our system. The performance bottleneck of Kafka is most likely due to the storage/hard disk Kafka relies on to read, write, and send data from. This can be further supported by the chart seen in Figure 38 whose data is derived from Table 7. This table shows the respective Kafka and Flink times as well as the Flink to Kafka lag. The Flink to Kafka lag indicates how many seconds slower was Flink to process a video compared to the time it took Kafka to fully send the same video. For example, at the lowest input throughput, Kafka was done sending a video 10 seconds (on average) earlier than Flink was to process that same video. On the contrary, during higher input throughputs, the processing time of Flink was almost equal to the video send time of Kafka. In other words, at that point Flink's performance was bottlenecked by Kafka's performance. Thus, in Figure 38, a clear correlation between the Response Time speedup and the Flink to Kafka lag time can be observed during higher input throughputs.

Another important result that we can see, is that for all but the highest input throughputs, the Response Time on parallelization 8 is below 60 seconds, i.e., the duration of the sample videos. This means that our system can process in real time an input throughput of at most 200MB/s. This characteristic is useful if this system was used to process live video input (which needs to be processed in real time) and we wanted to know its maximum capabilities.

Input Throughput (MB/s)	Kafka Send Time (s)	Flink Process Time (s)	Flink To Kafka Lag (s)	Response Time Speedup
35.60	4.04	14.16	10.12	**3.25**
63.28	12.19	19.33	7.14	**3.65**
98.88	12.97	19.35	6.38	**4.04**
142.38	33.34	34.14	0.80	**4.78**
177.98	28.90	32.03	3.13	**7.19**
205.66	39.97	40.93	0.96	**6.05**
253.13	79.23	79.34	0.11	**3.98**

Table 7: Flink to Kafka lag (on parallelization 8) with Response Time Speedup

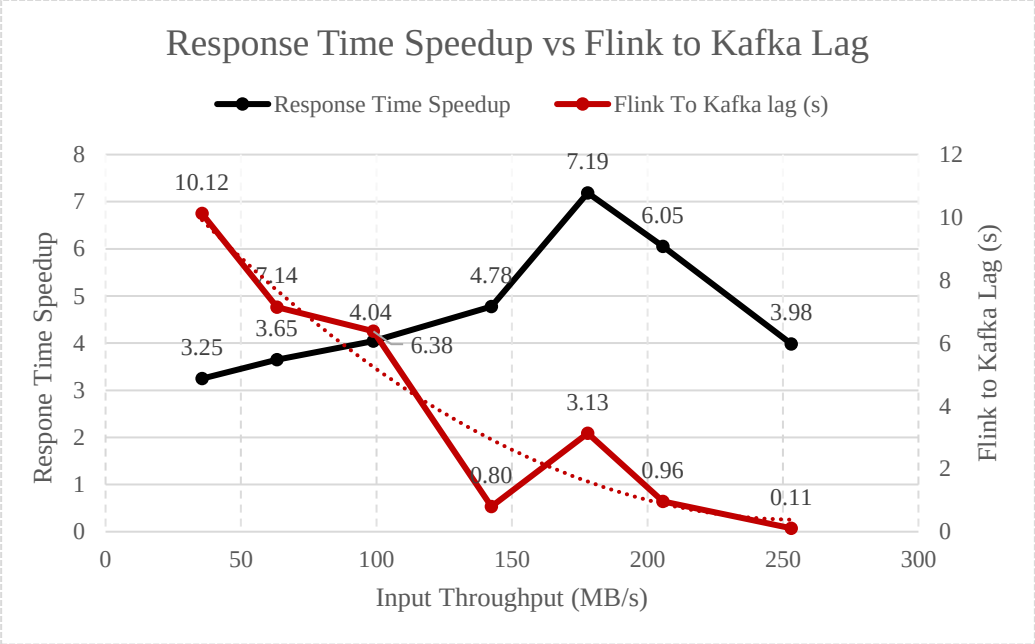


Figure 38: Response time vs Flink to Kafka Lag (trendline with dotted red line)

4.3.3 Speedup to Input Throughput – RGB encoded videos

Introduction

The goal of this experiment is identical to the previous one, but for RGB encoded videos. We want to determine the speedup gained as we increase the input throughput, while the rest of the system's variables remain unchanged. We expect to see the speedup to increase as we increase the input's throughput.

Procedure

The procedure is identical to the previous experiment. The same video A and video C are used as test videos, this time encoded with RGB888 (Table 8). Again, a list of input throughputs is created. On this experiment, smaller resolutions were used to stay within the systems capabilities during the highest chosen input throughput. The matrix of how each result input throughput was generated can be seen in Table 9.

Same as with the previous experiment, each result input throughput is tested against parallelization of 1 and 8. The average response time of each video sent and processed is the main result we will monitor. 8 Kafka partitions are used for all runs and the block size is fixed at 360KB.

	Duration	Total Frames	Frames Per Second	Encoding	Link	Clip timestamp
Video A	60 sec	1440	24	RGB888	*https://youtu.be/C3Kc9vigrrs*	0:00 to 1:00
Video C	60 sec	1440	24	RGB888	*https://youtu.be/C3Kc9vigrrs*	1:00 to 2:00

Table 8: Characteristics of Video A and C (RGB encoded)

	Video A				Video C				
Resolution	640x360	960x540	1280x720	1920x1080	640x360	960x540	1280x720	1920x1080	
Bitrate (MB/s)	15.82	35.60	63.28	142.38	15.82	35.60	63.28	142.38	**Result Input Throughput (MB/s)**
	✓				✓				**31.64**
		✓				✓			**71.19**
	✓	✓			✓	✓			**102.83**
			✓				✓		**126.56**
	✓		✓		✓		✓		**158.20**
		✓	✓			✓	✓		**197.75**
				✓				✓	**284.77**

Table 9: Input throughput matrix (RGB)

Results

In Table 10 we can see the results of this experiment. The first column is the input throughput and the next two columns (2nd and 3rd) display the result Response Time (in seconds) with parallelization of 1 and 8 respectively. The last column is the speedup gained between the two parallelization values. Figure 39 displays the absolute values of the result Response Times and Figure 40 displays the Speedup gained on Response Times after increasing the parallelization from 1 to 8.

Input Throughput (MB/s)	**Response Time (s)**		**Response Time Speedup**
	Par. 1	**Par. 8**	
31.64	49.18	28.54	1.72
71.19	86.57	27.93	3.10
102.83	144.78	32.07	4.51
126.56	154.49	38.27	4.04
158.20	194.63	38.37	5.07
197.75	276.97	50.60	5.47
284.77	365.09	103.35	3.53

Table 10: Results of the Speedup to Input Throughput experiment (RGB)

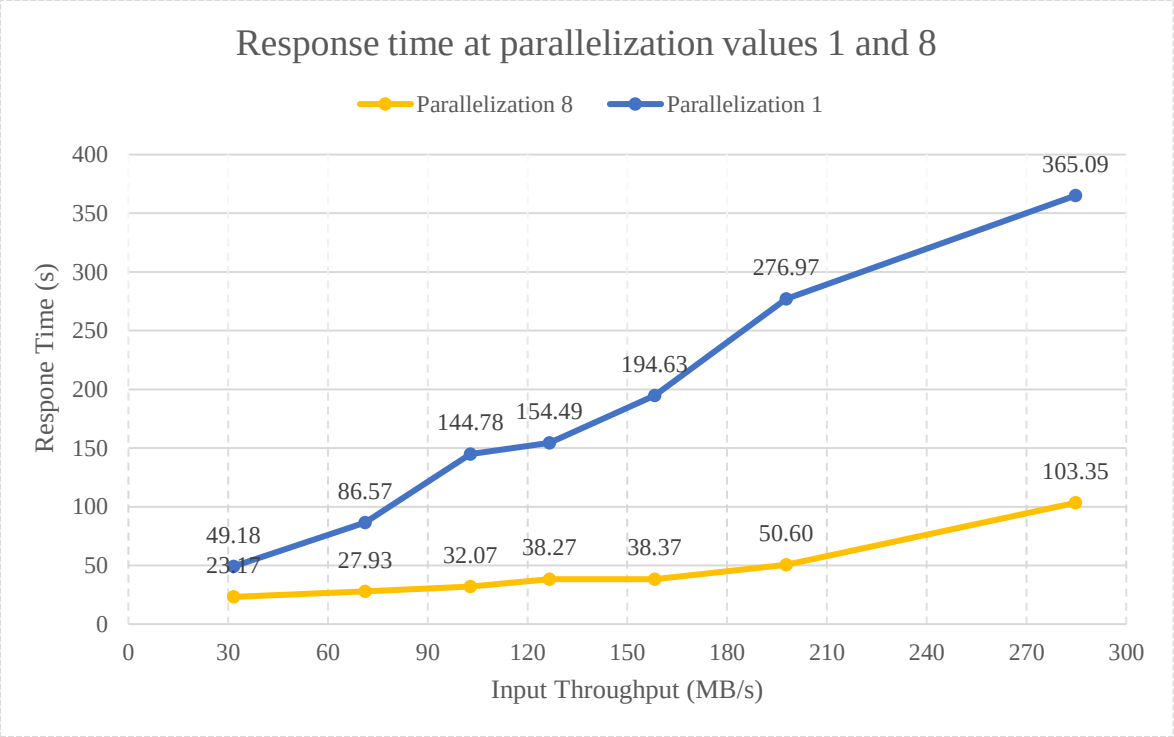


Figure 39: Chart with the result Response Time values (in seconds) of parallelization of 1 (blue line) and 8 (yellow line) (RGB videos).

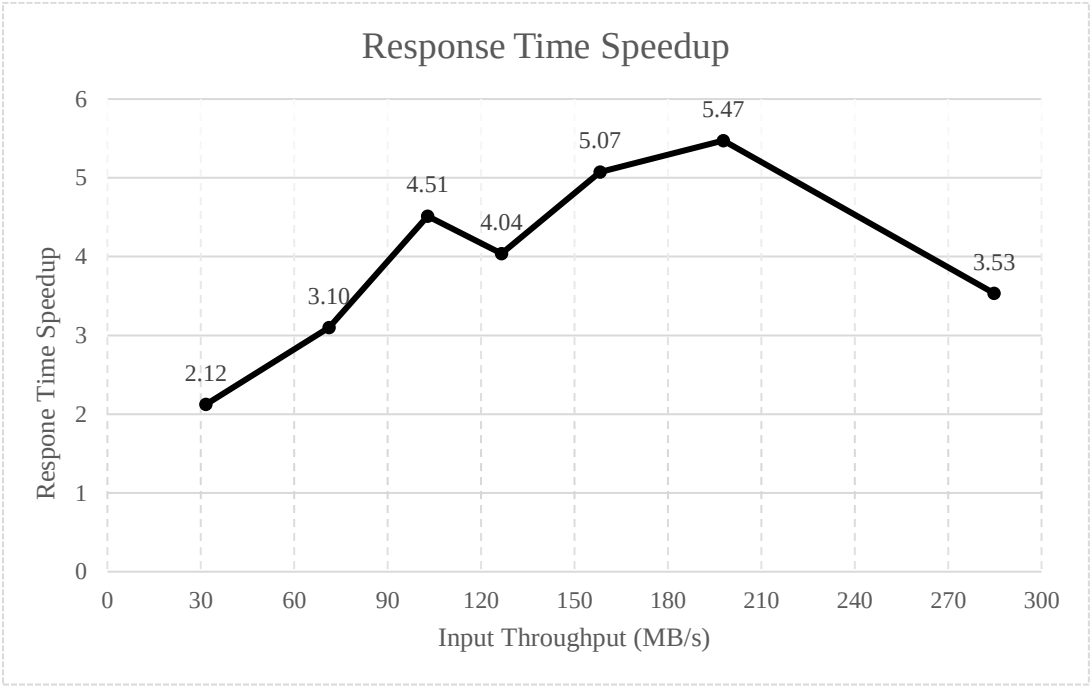


Figure 40: Result Speedup gained on Response Times through increasing parallelization (RGB videos).

Discussion

In this section we will compare the results of this experiment of RGB videos with the results of the equivalent experiment of YUV encoded videos. This is also possible since the list of input throughputs that we chose for both experiments ended up being quite similar (an average of 139.56MB/s input throughput for the YUV experiment and an average of 138.99MB/s for the RGB videos).

Speedup Comparison: Similarly to the previous YUV experiment, we observe an increasing speedup of the Response Time with the RGB encoded videos too. By comparing the result speedups of the two experiments, we observe a lower minimum and maximum on the RGB experiment but on average the speedups gained are relatively close (3.98 average speedup on the RGB videos, versus 4.70 on the YUV videos). This difference is due to the RGB videos being slightly harder to transfer and to process for both parallelization values. As a result, the performance *difference* between the two parallelization values is slightly lower.

The RGB videos are harder to transfer for the same input throughput since the full data has to be sent to Flink. In contrast, for the YUV videos, only 2/3rds of the video data (the Y component) have to be sent. Also, the RGB video data has to be transformed into grayscale whereas the Y component data of YUV videos is already in that form.

Response Time Comparison: For both types of videos, the system with a parallelization of 8 managed to keep a low response time. The average response time of YUV videos was slightly lower at 37.8 seconds, compared to the average response time of 44.82 seconds for the RGB videos. The slight difference in absolute seconds was highly expected for the reasons stated in the previous paragraph.

Again, we observe that the response times for parallelization value of 8 are for all but the highest input throughput, bellow 60 seconds (i.e., the video's actual duration). This characteristic would allow live video processing with an input throughput of up to 200MB/s.

Regarding the drop in performance in the highest resolution in this experiment, the same root cause seems to be the case again, similarly to the YUV experiment. The resulting *Flink to Kafka* lag shown in Table 11 (chart in Figure 41) supports this conclusion. The reason for the spikes on the 3rd and 5th input throughputs is that 4 clients were used to achieve the respective input throughputs (instead of using 2 clients, as with the rest of the input throughputs). Using more clients (i.e., more Kafka Producers) allows Kafka to perform moderately better for the same input throughput. The trendline (order 2 polynomial) however is still descending.

Input Throughput (MB/s)	Kafka Send Time (s)	Flink Process Time (s)	Flink To Kafka Lag (s)	Response Time Speedup
31.64	7.17	19.54	12.37	**2.12**
71.19	21.79	25.14	3.35	**3.10**
102.83	17.65	26.13	8.47	**4.51**
126.56	35.55	36.77	1.21	**4.04**
158.20	31.56	36.32	4.76	**5.07**
197.75	48.95	49.59	0.64	**5.47**
284.77	103.03	102.90	0.00	**3.53**

Table 11: Flink to Kafka lag (on parallelization 8) with Response Time Speedup

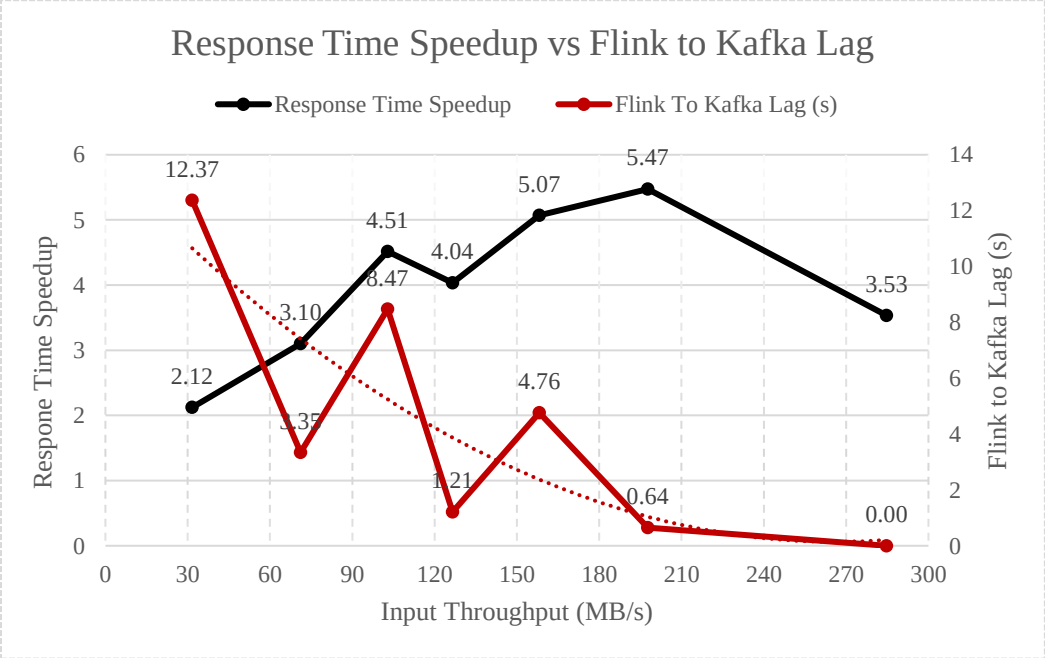


Figure 41: Response time vs Flink to Kafka Lag (trendline with dotted red line)

4.3.4 Speedup to Flink Parallelization – YUV encoded Videos

Introduction

The goal of this experiment is to determine the speedup gained as we increase the parallelism of our Flink Application, for YUV encoded videos. We expect to see a clear increase in performance (i.e., how fast each input video is processed) when we increase the number of parallel pipelines. We also expect to see a greater speedup for higher resolution videos.

Procedure

We repeat the following procedure for each resolution we are going to test: Setup a number of clients that send video data (simultaneously) to Kafka which is afterwards sent to Flink. On the first iteration for this resolution, Flink has 1 parallel pipeline (i.e., no parallelism at all). All video data is sent to Flink, processed there and the process duration is timed. On the second iteration, the same volume of video data is sent. However, Flink will now have 2 parallel pipelines instead. Next, Flink will have 4 pipelines on the 3rd iteration, 6 on the 4th, and 8 on the 5th final iteration. The processing time between all 5 iterations will be compared. 8 Kafka partitions are used for the input topic during the whole experiment. This whole process is repeated for the next video input that has a different resolution.

In this experiment, we use "Video A" as input to our clients, which was also used in the previous experiment. The video's characteristics for each copy with a different resolution and some additional metrics can be seen in Table 12.

	Duration	**Frames**	**Frames Per Second**	**Encoding**		
Video A	60 sec	1440	24	YUV420p		
Width	**Height**	**Bitrate (MB/s) (1)**	**Bitrate Sent (MB/s) (2)**	**Parallel Clients (3)**	**Times sent per client (4)**	**Input throughput (MB/s) (5)**
960	**540**	17.80	11.87	3	4	**53.4**
1280	**720**	31.64	21.09	3	4	**63.28**
1920	**1080**	71.19	47.46	3	3	**142.38**
2560	**1440**	126.56	84.38	3	2	**253.13**

Table 12: Characteristics of Video A used in the speedup experiments.

Two bitrates are seen in the table (Columns (1) and (2)). *Bitrate* (1) corresponds to the video file's actual bitrate (i.e., how fast a client should read it from the disk). *Bitrate Sent* (2) is the bitrate of data sent to Kafka (i.e., how fast it is sent from the Client to the Kafka Broker). For a raw YUV420p encoded video, we only send the $2/3^{rds}$ of the video data we read. This happens because we only send the Y component of each frame, as discussed in more detail in Chapter 3.1.3. For a raw RGB encoded video these two values would be the same (i.e., we send to Kafka every bit we read from the file).

The *Bitrate Sent* value is important for multiple reasons. Any Client will need to be able to send data at least this rate through the network. The Kafka Broker will need to accept input at that rate (i.e., receive and save input data quickly on the disk). The Kafka Broker will also need output at the same rate through the network towards Flink. Similarly, Flink will have to accept data as input at this rate and most importantly process it at least this rate.

Parallel Clients (3), 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) increases linearly.

Times Sent per client [4], is the total number of times the video is sent (in sequence) by each client in every iteration. Each time a different video ID is generated by the client. Due to the different IDs, Flink will process each video as a different "task". 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.

Input Throughput (MB/s) [5], is the sum bitrate of all the videos being sent in parallel. For example, if three clients use as input a 1920x1080 resolution video (with YUV420p encoding), each one sends data to Kafka at a bit rate of 47.46MB/s. Therefore, the total input throughput will be $3 * 47.46\,MB/s = \mathbf{142.38\,MB/s}$.

In Figure 42 we see an example iteration with 3 clients where each one sends the same video 4 times. As a result, Flink will have to handle three times the original bitrate of each video (since 3 clients sent data concurrently). Each client reads the same video file (e.g., "Video 1" with a resolution of 1920x1080). A different video ID/key is assigned on each one (i.e., 12 different video IDs, since we use 3 clients and each one sends the video 4 times). This will allow Flink to process each one independently of each other. In other words, when all videos are processed, Flink should have output 12 different scene detection results. Of course, since it is the same video file, the resulting number of scene detections will have to match. For all the experiments we used a block size of 1/20th of the frame's height, i.e., each frame was partitioned into 20 blocks. For example, for the 2560x1440 video resolution the block size was 270KB.

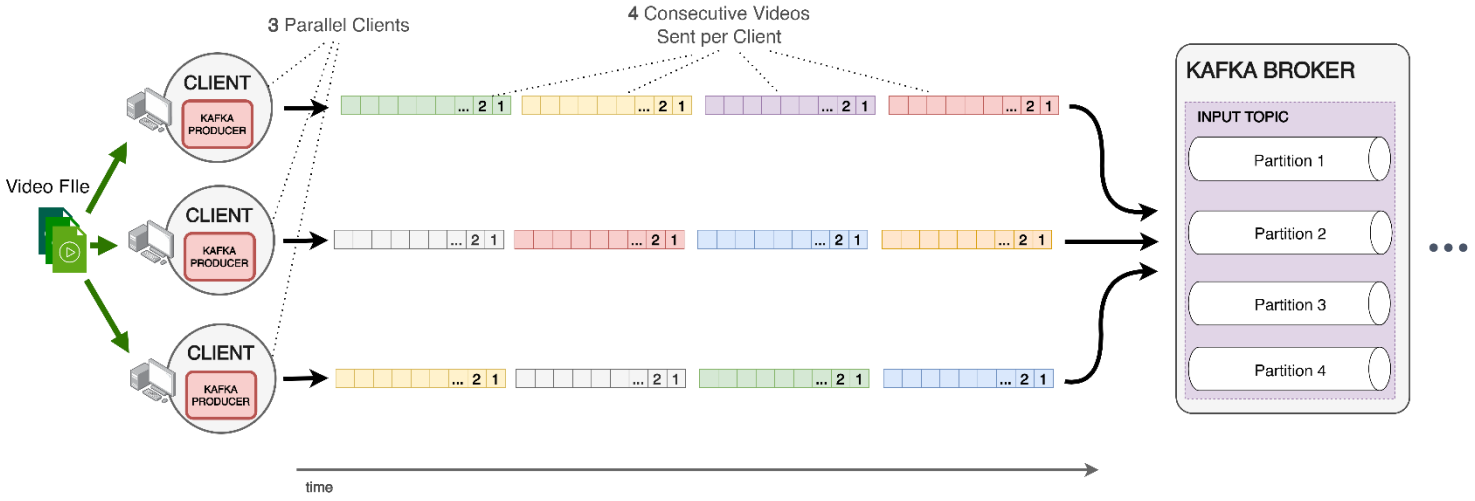


Figure 42: Example iteration with 3 parallel clients and 4 consecutive videos sent by each one. Each set of colored blocks represents the entire data sent for a single video. (The number of topic partitions in the Kafka Broker shown here is irrelevant)

Results

In Table 13 we see the results of the experiment. As described in the previous section, for each resolution 5 iterations/tests are executed, one for each different number of parallel Flink pipelines (i.e., 1, 2, 4, 6, 8). For each resolution and number of pipelines, we have 3 columns of result data.

Total GB Processed is the total size of all the videos on each iteration. In other words, how much data Flink processed for that iteration (i.e., for each parallelism value). *Flink Process Time* is how much time it took Flink to process the same amount of data and output the results. Total GB processed can be calculated beforehand since we know how many videos we will send for each resolution and iteration and what their size is. The total time processed is calculated by logging the exact time the processing starts

and when it ends (i.e., from the moment the first video starts being processed to the moment the last one finishes).

Total MB/s Processed is simply the division between the previous two values. It represents the output throughput of the system and is a very important metric to understand how well it performed.

Unfortunately, unlike the previous two experiment sections (4.3.2 and 4.3.3), we did not have access to the Response Time values of this experiment. For this reason, the speedup and the analysis further down is based on the Flink Processing time and the Output Throughput (MB/s).

Table 13: Results of the Speed up tests.

Video Width	Video Height	Parallel Clients	Input Throughput (MB/s)	Number of Parallel Pipelines	Total GB Processed	Flink Processing Time (s)	Output Throughput (MB/s)
960	540	3	**35.60**	1	8.34	101.1	**84.53**
				2	8.34	67.2	**127.09**
				4	8.34	48.5	**176.02**
				6	8.34	48.6	**175.62**
				8	8.34	45.6	**187.28**
1280	720	3	**63.28**	1	14.83	176.8	**85.91**
				2	14.83	132.8	**114.35**
				4	14.83	80.7	**188.30**
				6	14.83	85.1	**178.53**
				8	14.83	76.6	**198.17**
1920	1080	3	**142.38**	1	25.03	322.2	**79.54**
				2	25.03	192.9	**132.89**
				4	25.03	137.1	**186.91**
				6	25.03	116.9	**219.21**
				8	25.03	115.0	**222.90**
2560	1440	3	**253.13**	1	29.66	388.7	**76.31**
				2	29.66	386.2	**78.65**
				4	29.66	280.9	**108.13**
				6	29.66	242.0	**125.52**
				8	29.66	132.1	**229.95**

Discussion

If we focus our attention on the Total MB/s processed, we can see how the performance almost always increases relative to the number of parallel Flink pipelines. The performance increase is also a bit more drastic for the higher resolutions. For the first two lower resolutions (with equivalent low-performance requirements) the Flink application could keep up even with no parallelism (i.e., a single pipeline[16]).

For the 1920x1080 resolution video with a substantial input throughput, the Flink application managed to keep up with the input throughput only after having at least 4 parallel pipelines. Similarly, for the video with the highest tested resolution of 2560x1440, the system barely failed to keep up with the equally high input throughput. It managed to achieve a processing bitrate of 229.95 MB/s, while the input throughput was 253.13 MB/s.

[16] It should be noted, that even with one pipeline, the application is still multithreaded. This is an inherit functionality of Flink where each operator has its own thread (unless chained together).

It is important to note that having an output throughput slower than the input throughput does not mean that any video frames are lost, or that any shots are not correctly detected. It simply indicates that the system could not process the video in real time (e.g., if a video would be 60 seconds long, for these input throughputs the response time would be over 60 seconds).

In the Chart of Figure 43, we can see the speedup in output throughput. We compare against the single pipeline iteration of each resolution's set of tests. Looking at the chart, we can see that the performance gained for the two lower resolutions seems to reach a maximum after the 4 parallel pipelines. However, for the second to highest resolution tested (1920x1080), the performance seemed to increase even between 6 and 8 parallel pipelines (albeit less drastically). For the highest resolution, the performance increase between 6 and 8 parallel pipelines was vast. It indicates that probably with even more parallel pipelines we would keep seeing great performance increases.

In essence, we observe that the speed-up increases linearly with the parallelism, up to a maximum. We also observe that the magnitude of the speed-up increases with the amount of data (i.e., the resolution of the video).

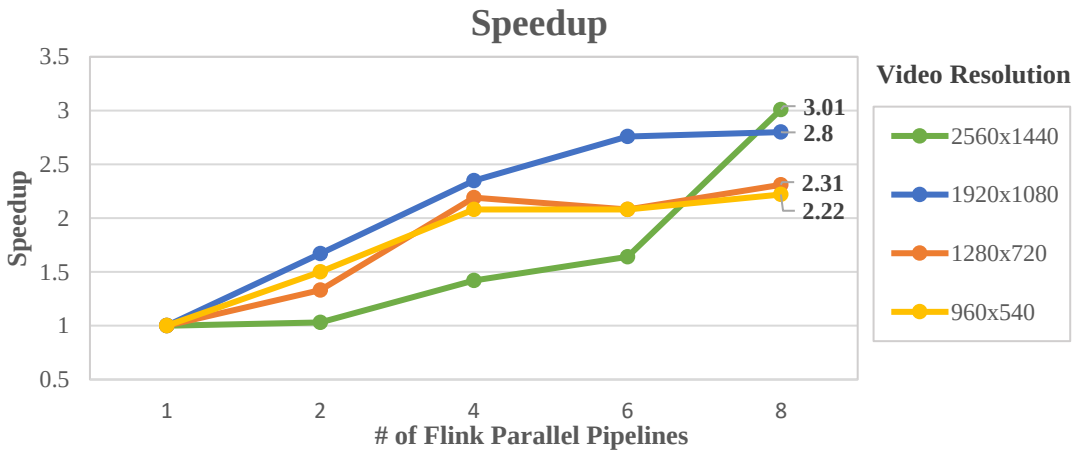


Figure 43: Relative Speedup compared to the number of Flink parallel pipelines

One question is *why* the speedup increases linearly until reaching a maximum. Additionally, a second question is why the maximum closely depends on the resolution of the video (i.e., why lower resolutions reached a maximum earlier than the higher resolutions).

The reason for a linear speedup increase at first, is because at first the bottleneck of the system is the performance of the Flink Nodes. A Flink Node can process *X* amount of data or blocks per second, whereas the Kafka Broker can provide *Y* amount of data per second (with *Y* obviously larger than *X*). Thus, as we increase the parallelism (i.e., the number of Flink Nodes), the rate at which data is processed increases linearly.

However, the speedup reaches at some point a maximum. The reason for this is that the max input rate (*X*) of the Flink Nodes reaches or goes past the max output rate (*Y*) of the Kafka Broker. In other words, the Kafka Broker cannot sufficiently provide additional input to all the Flink Nodes. This was also discussed during the two previous experiments. For this reason, increasing the parallelism further to 10 or more would not help increase the output throughput.

The second question was why this maximum depends on the resolution of the video. The experiments seem to indicate that when it comes to the raw throughput of Kafka, sending larger blocks of data is more efficient than multiple small ones. Since the block size was relative to the frame size of each video, higher resolution videos used larger blocks of data. As a result, the Kafka Broker could achieve a moderately higher max output with higher resolution videos. Thus, it could also provide for even more Flink Nodes before introducing a bottleneck.

This bottleneck can be improved in multiple ways. Some options are fine tuning the block sizes even further (i.e., using bigger block size even for smaller resolutions), adding multiple Kafka Brokers or increasing existing Broker's performance (i.e., by using a storage Disk/SSD with improved performance).

4.3.5 Speedup to Flink Parallelization – RGB encoded Videos

Introduction

The goal of this experiment is to prove that our application can also handle raw RGB encoded videos as well as increase its output throughput when we increase Flink's parallelism.

Processing raw color video instead of grayscale makes the overall processing more challenging for two reasons; First, it requires more data to be transferred between the Client, the Kafka Broker, and Flink. Secondly, the Flink Application will require additional processing power to convert the color data to grayscale. However, we still expect to see an increase in the total throughput of the application.

Procedure

The overall procedure is similar to the previous speedup experiment. The main difference is that the clients will use as input a raw RGB encoded video. The blocks they will generate and send to Kafka will contain color data. The first operator of the Flink Application will detect that the input blocks contain RGB encoded data (through the information embedded in the assigned key). It will convert them to grayscale and send them to the next operator. The rest of the operators will function as usual.

In this experiment, three resolutions will be tested, as shown in Table 14. The sample video that will be used as input is the same as "Video A" from the previous experiment but with an RGB888 encoding instead (i.e., 8 bits per color, 24 bits in total). The full characteristics of this video are seen in Table 14 along with the experiment's procedure details. The columns are the same as in Table 12. The only difference is that there is no distinction between the file's *Bitrate* [(1)] and the *Bitrate Sent* [(2)] to Kafka. This is of course because for this video's format we sent every single bit we read from the actual file.

For this experiment, we use 2 clients. Each one will send the input video twice, each time with a different video ID. We start the procedure with a single pipeline on the first iteration. We increase the number of parallel pipelines to 2, 4, 6, and 8 on each next iteration respectively. 8 Kafka partitions are used for the input topic during the whole experiment.

	Duration	**Frames**	**Frames Per Second**	**Encoding**	
Video A	60 sec	1440	24	RGB888	
Width	**Height**	**Bitrate (MB/s) [(1)] Bitrate Sent (MB/s) [(2)]**	**Parallel Clients [(3)]**	**Times sent per client [(4)]**	**Total MB/s [(5)]**
920	**540**	35.60	2	2	**71.2**
1280	**720**	63.28	2	2	**126.56**
1920	**1080**	142.38	2	2	**284.76**
2560	**1440**	253.13	2	2	**506.26**

Table 14: Characteristics of Video A used in the RGB speedup experiments.

Results

The results of the RGB Speedup tests can be seen in Table 15. The table's columns are the same as with the previous experiment's result, Table 13. The most important values to pay attention to are the *Total MB/s processed* (how fast Flink handled its input data) against the *Total MB/s Required* (how fast input data was generated by the clients). Ideally, the former has to exceed the latter.

The results show a significant speedup when we increase the number of parallel Flink pipelines. The two lower resolutions managed to reach the required processing rate even with no parallelization. As expected though, with the highest setting of 8 Flink pipelines, almost double the initial processing rate was achieved. However, for the two highest tested resolutions of 1920x1080 and 2560x1440, our system could not reach the total required throughput for each resolution. With two parallel clients, (i.e., double the base throughput requirement of each resolution), we needed to get at least 284 MB/s and 506 MB/s, but instead, we reached 253MB/s and 239MB/s respectively.

It should also be emphasized that multiple problems arose during testing the 2560x1440 resolution videos with multiple Kafka partitions. When one or two Kafka partitions with an equal number of Flink Nodes were used, the system naturally underperformed (due to Flink) but it was relatively stable. However, when over 6 partitions and Flink nodes were used, the system performed better but it was highly unstable at first. The cause of this was that a lot more data was being requested by the Flink Nodes. This resulted in the Kafka Broker (and the machine it was on) reaching its performance limits and struggling to fill these requests. As a chain reaction, the Kafka Producers of the clients sometimes couldn't receive confirmations that their messages were successfully received. Furthermore, this resulted in multiple timeouts for the clients, some lost messages on rare occasions, and mainly uneven performance for the Kafka Producers and the Broker. However, after finely tuning the Kafka Producers, the JVM they were running on, and the Broker itself, the system was much more stable (by allowing larger timeouts, larger internal buffers, and giving more memory to the Java Apps).

Video Width	Video Height	Parallel Clients	Input Throughput (MB/s)	Number of Parallel Pipelines	Total GB Processed	Flink Processing Time (s)	Output Throughput (MB/s)
960	540	2	**71.2**	1	8.34	54.93	**155.45**
				2	8.34	47.6	**179.48**
				4	8.34	38.1	**224.21**
				6	8.34	29.6	**288.42**
				8	8.34	30.0	**284.33**
1280	720	2	**126.56**	1	14.83	98.5	**154.19**
				2	14.83	86.9	**174.80**
				4	14.83	79.1	**191.96**
				6	14.83	62.4	**243.53**
				8	14.83	61.9	**245.34**
1920	1080	2	**284.76**	1	33.37	335.4	**101.89**
				2	33.37	259.5	**131.70**
				4	33.37	230.6	**148.22**
				6	33.37	134.1	**254.84**
				8	33.37	135.0	**253.07**
2560	1440	2	**506.26**	1	59.33	845.3	**71.87**
				2	59.33	440.6	**137.89**
				4	59.33	410.0	**148.18**
				6	59.33	328.5	**184.93**
				8	59.33	254.5	**238.71**

Table 15: Results of the RGB Speedup tests.

Discussion

The resulting speedup of the experiment was highly expected (shown in Figure 44). For the highest resolution of 2560x1440, having 8 parallel pipelines allowed our system to process video data 232% faster, compared to no parallelization. For the 1920x1080 resolution, with maximum parallelization, we processed video data 148% faster. A quite large speedup, similarly to the previous experiment. However, it should be noted that during the processing of these resolutions and **especially the 2560x1440**, the overall system struggled to both **transfer and process** the video with a single Kafka partition and a single Flink node. As a result, these initial runs greatly underperformed. This led to a possibly misleading high speed up when compared to the best-case scenario of using 8 Kafka partitions and 8 Flink nodes. In contrast, the absolute performance of the system during testing these two resolutions was close to the lower resolution tests (see Figure 45).

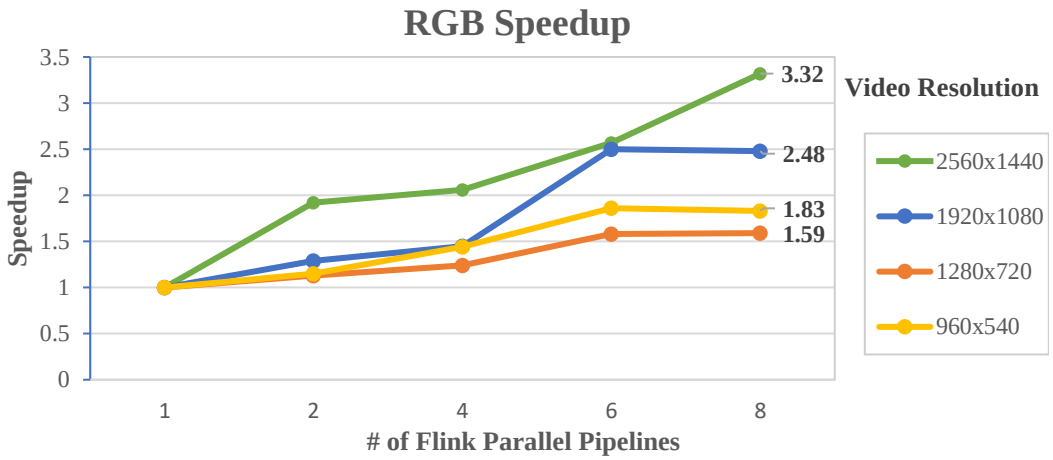


Figure 44: RGB speedup compared to the number of Flink parallel pipelines

The two lower resolutions saw a smaller relative speedup compared to no parallelization (83% and 59% for 960x540 and 1280x720, respectively), similarly to the previous experiment. However, in contrast to that experiment, in this one, they had almost equal or even better absolute processing rates compared to the higher 1920x1080 resolution, as seen in Figure 45. Between these two experiments (grayscale and RGB experiments), our Kafka and Flink Cluster configuration has been further optimized to our specific needs. This is the most probable reason for this mismatch in absolute performance between the two experiments.

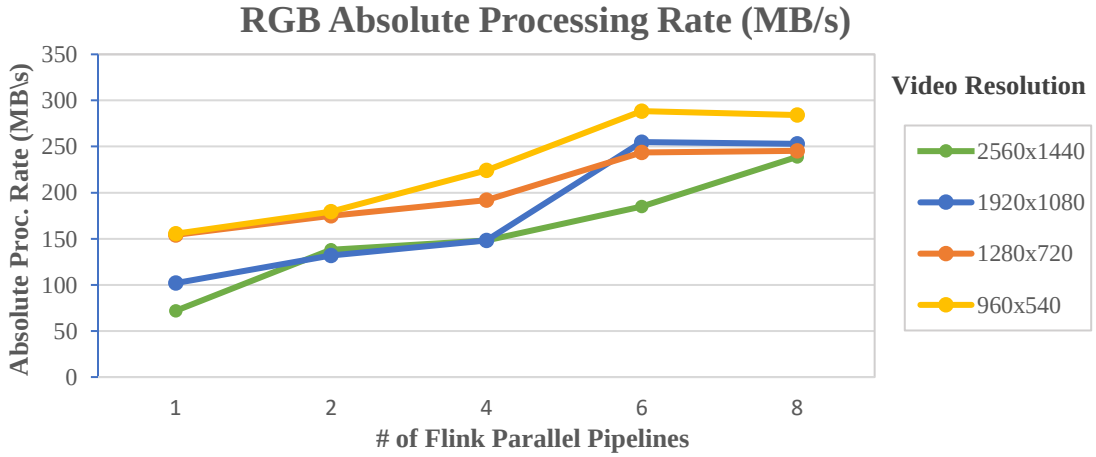


Figure 45: RGB Experiment: Absolute Processing rate compared to the number of Flink Parallel Pipelines.

Another interesting result of the tests is the plateau that all resolutions seem to reach when 6 parallel Flink pipelines were used. This is the same issue that came up in the previous experiments, and again, it is most probably tied to Kafka's overall max available performance on our current system.

If we ignore the performance bottlenecks of our test system (mainly Kafka's bottlenecked performance during high input throughputs), the speedup seems to increase linearly with the parallelism of Flink. Overall, the addition of the necessary RGB to grayscale conversion doesn't seem to negatively affect the Flink worker's performance. It does however have an impact on the overall system's performance due to the increased amount of data that needs to be transferred from the clients to the Flink processing cluster.

5. Conclusion

In this thesis we designed and implemented a system for distributed video processing in real time using Apache Flink's distributed stream-processing engine fueled by the high-speed distributed event streaming platform of Apache Kafka. The main focus of our work was detecting shot changes in raw high-resolution video on a scalable and flexible manner. The biggest challenge we had to overcome was designing the parallel pipelines that would processes the blocks of the partitioned frames and how we would efficiently and rapidly supply these blocks to them. Thankfully with the excellent tools and options provided by both the Flink's framework and Kafka, we managed to create both a processing pipeline and a streaming platform that provides for it, which can be parallelized as much as required, allowing for extensive scalability, achieving great performance in distributed raw video processing. Reaching this point however required deeply analyzing the system at hand, the goal's requirements as well as learning about Kafka's and Flink's extensive features, strengths and weaknesses.

The clients read the video files, split them up in multiple blocks and send them to Kafka through a their Kafka Producer. Each block is sent with a unique key, which identifies the video and the frame it is from, in how many blocks that frame was divided, each block's sequence number, along with extra video metadata (e.g., resolution, total frames). The Kafka Broker then receives the message and places them in the topic's partitions using a round robin pattern for equal distribution. In the next step, the moment the Flink parallel pipelines are ready, they start requesting messages; each parallel Flink source operator from exactly one Kafka topic partition. If the pixel data received is in RGB encoding, it is first converted to grayscale. The next processing operator then receives blocks of grayscale pixel data and creates their histogram of intensity. At the next step, all histogram blocks are redistributed in a such a way that all blocks of each frame are gathered to the same instance of the next operator. This allows the operator to aggregate the histogram of the blocks and output the full frame's histogram. This is then sent then sent downstream twice to two operator instances. With a properly designed key, each parallel operator instance down the line will receive two adjacent frames' histograms, in order to compare them and generate their difference. At this point a stream with all the histogram differences are received by the last operator that has two functions. Firstly, it announces camera shot changes if the differences are above a precalculated threshold. Secondly it detects gradual fade shot change transitions between multiple frames by utilizing its own in-memory state. Any kind of shot changes that are detected are sent as descriptive messages over to another Kafka topic.

The main advantage of this work is the fact that we reduced the size of each message by splitting each frame into smaller blocks. This both increased the transfer rates and latency of our data and allowed for more efficient parallelization and system scalability. This is of course allowed by the fact that no communication occurs (or is required) between the parallel tasks due to the nature of the current processing and the minimal correlation between the data blocks. Another great benefit is the general flexibility of processing videos of different resolutions, encodings, and block sizes concurrently. The use of Kafka as the data streaming platform with its "pull" subscribe system, which allows the use of Flink nodes with a big difference in performance. Additionally, from the client's side, the application is transparent about the parallelization factor of the system, decreasing the client's complexity.

Future Work

There are multiple ways to improve the current system, either in the form of small incremental improvements or more impactful changes that could greatly increase performance or add useful features. These were not implemented either due to time constraints or due to requiring more extensive research.

Improving System's Utilization – Configurations Improvement
The overall system is a reasonably complex system with a plethora of intertwined configurations. Even though we have done extensive research with corresponding experiments in order to pinpoint each configurations best option and increase the system's final performance, there is still plenty room to grow in order to fully utilize this system. Some examples are determining the best Kafka Consumers' polling rate versus batch size which is of course also depended on Flink's TaskManager's performance per Pod as well as the size of the partitioned frame's blocks.

Minimizing Data used
Another approach in increasing throughput and the final performance of the system would be to decrease the amount of data sent to Flink either by decreasing the number of blocks sent (e.g., send half the blocks/rows per frame) or even the number of frames (e.g., bypassing one or two frames each time) without of course impacting the correctness of the algorithm. Finding the optimal setting of this data reduction could provide a substantial increase in performance.

Continuously update the threshold values
The shot detection algorithm could be further improved by continuously updating based on a model the now-fixed cutoff thresholds for scene detections. This should be an easy feature to implement as, depending on what model we would use, it could require minimal additional state to be kept.

MPEG Video's processing
Processing MPEG encoded videos with minimal transcoding would ultimately solve multiple of this system's bottlenecks; mainly its Kafka transferring times between the video providers and the Flink processing pipeline. By using the encoding's I-Frames[17], we would theoretically be able to compare them directly, while also possibly utilizing their Macroblocks[18] by comparing the DCT coefficients directly. Using this type of information however poses multiple other difficulties; If we still want to processes each frame via multiple partitions (e.g., a row of Macroblocks or simply a single Macroblock) we need to decode the video in a previous stage, possible on the Video Preprocessor. Of course, this requires designing and implementing a custom decoder that will be able to extract the DCT coefficient information as well as slightly increasing the load the Video Preprocessor has to handle. The other option would be to use full jpeg I-frames on Flink (which exists as an existing feature) and extracting their histogram on Flink via an existing external performant feature-full image and video manipulation library such as OpenCV. In turn, this would also require to redesign both the Flink pipeline and the individual Kafka records.

Automatic Scaling
Since the system is easily scaled up and relatively simple through the use of a single topic with each data block holding all the required info in order to be correctly processed, it would greatly benefit from an

[17] https://en.wikipedia.org/wiki/Video_compression_picture_types
[18] https://en.wikipedia.org/wiki/Macroblock

autoscaling algorithm such as the one described in [7], by monitoring the Kafka input vs message request rate, allowing to utilize a Flink Cluster on top of a Kubernetes Cluster to its fullest.

References

[1] Q. Huang *et al.*, “SVE: Distributed Video Processing at Facebook Scale,” *SOSP 2017 - Proc. 26th ACM Symp. Oper. Syst. Princ.*, pp. 87–103, 2017, doi: 10.1145/3132747.3132775.

[2] 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 (Basel).*, vol. 11, no. 7, 2019, doi: 10.3390/sym11070911.

[3] Y. Kim and C. Jeong, “Large Scale Image Processing in Real-Time Environments With Kafka,” pp. 207–215, 2017, doi: 10.5121/csit.2017.70120.

[4] K. Yu, Y. Zhou, D. Li, Z. Zhang, and K. Huang, “A large-scale distributed video parsing and evaluation platform,” *Commun. Comput. Inf. Sci.*, vol. 664 CCIS, no. April 2018, pp. 37–43, 2016, doi: 10.1007/978-981-10-3476-3_5.

[5] F. J. Seinstra, J. Geusebroek, D. Koelma, C. G. M. Snoek, and A. W. M. Smeulders, “High-Performance Video Content Analysis with Parallel-Horus,” *Computing*, pp. 64–75, 2007.

[6] H. J. Zhang, A. Kankanhalli, and W. S. Smoliar, “Automatic partitioning of full-motion video,” *Multimed. Syst.*, pp. 10–28, 1993.

[7] P. Giannakopoulos and E. G. M. Petrakis, “Smilax: Statistical Machine Learning Autoscaler Agent for Apache FLINK,” *Lect. Notes Networks Syst.*, vol. 226 LNNS, pp. 433–444, 2021, doi: 10.1007/978-3-030-75075-6_35.