Distributed Video Analytics using Kafka & Faust

Jeevithiesh Duggani
jeevithieshd@iisc.ac.in

Sasanka Sekhar Sahu
sasankasahu@iisc.ac.in

Abstract— **With growing rate of data being used by enterprises globally, the notion is Big-data is not a new concept anymore. However, some use cases in the industry relies heavily on Fast Data processing. Fast data or streaming data is a continuous stream of data arriving from various sources like mobile devices, sensor networks, financial transactions, logs, etc. To process streaming data, various stream processing frameworks are available. One of the most popular event stream processing systems. In this project we build a distributed streaming pipeline for video analytics using Kafka & Faust. Faust a python stream processing library which is heavily inspired from Kafka Streams. We perform experiments to demonstrate weak and strong scaling of the pipeline.**

Keywords— Distributed Computing, \Video Analytics, Kafka, Kafka Streams, Faust

I. Introduction

In today's world data is being generated in an insanely fast rate. Enterprises are already being powered by huge amounts of data, also called big data, and becoming more and more reliant on data, analytics, AI, automation and IoT technology as time progresses. AI and data analytics have been shaping business operations and generating revenue in billions across global economy. So much so that, an article by An article by Forbes claims that every company is eventually going to become a data and analytics company now. Soon, data itself will become a main focus for nearly every business be it a apparel company, an insurance company, or a food application. And data analytics will form the core of every company's business model. Existing information is taken as input, pre-processed, modified, analysed and more useful information is generated. Every service or application generates data, whether it is log messages, metrics, user activity, outgoing messages, or something else. Every byte of data is considered significant and has something of importance that will inform the next thing to be done. But, like everything else, this doesn't come without challenges.

IBM, a leader in this sector, breaks big data into four dimensions (four V's): Volume, Velocity, Variety & Veracity. Volume refers to the scale in which the data is getting generated or transferred and Variety refers to the different forms of data being generated e.g., music, images, videos, logs, events (button clicks, likes on facebook, etc.). Velocity refers to the speed with which the data is produced or transferred e.g., The New York Stock Exchange captures more than 1 TB of trade information during each trading session. Veracity refers to the uncertainty of the data that is being collected i.e., whether the data can be trusted to make business-critical decisions based on it. This dynamic nature of data makes the end-to-end data-analytics process challenging and motivates further for well-designed data-processing system.

It can be easily observed in e-commerce websites like flipkart and amazon that the clicks on products are recorded and even the amount of time we spend on a certain product is recorded. These information are then turned to recommendations in almost blazing fast speeds which are shown to the user only a little later. As can be seen, this has direct effect on business. The faster this can be done, not only it helps in more sales of the product but also makes the platform smoother and responsive. To achieve this, data movement between services become critical. Focusing less on moving the data around can allow organisations to pay more attention to the core business logic at hand. Hence, pipelining becomes a critical component in such data-driven enterprises. Figuring out optimal movement of data becomes as important as the data itself. That is, our focus should be as equally on analysing and processing fast data as it on analysis big data.

Big data is data at rest: collections of structured and unstructured data that have been stored in popular file systems like HDFS and are waiting to be analysed historically. Streaming data (or fast data), on the other hand, is data that is always changing. Big data can be evaluated via batch processing at a later time, but real-time data streams must be handled with quickly. Fast data is data in motion, streaming from hundreds of thousands to millions of endpoints such as mobile devices, sensor networks, financial transactions, logs, retail systems, telecom call routing and authorization systems, and more into applications and computing environments. Businesses that rely on data, such as telecommunications, financial services, health/medical, energy, and others, are the ones who are exploiting big-data with real-time applications built on top of fast data. It's also changing the game for developers, who now have to construct apps that can handle ever-increasing data streams.

One of the most compelling use-cases of fast data processing is streaming analytics. Data arrives in the enterprise in fast-moving streams as it is created. A stream of data can contain a variety of data kinds and formats. This includes data from new sources including sensor data, as well as web server clickstreams, machine data, and data from consumer interactions devices. Because of the rise in fast data processing, it's now possible to analyse data as it comes in, rather than after it's been transferred to a data warehouse for longer-term analysis. For data-driven applications, the ability to examine streams of data and make in-transaction decisions based on this fresh data is the most enticing vision.

II. Literature

A. *Apache Kafka*

In this project, we use one such event stream processing platform called Apache Kafka. Apache Kafka is a distributed publish-subscribe messaging system that receives data from a diverse sources, called Kafka producers, and makes it available in real time to target systems of interest, called Kafka Producers. Kafka is a real-time event stream processing system for big data that is written in Scala and Java. Like other message brokers systems, Kafka facilitates the asynchronous data exchange between processes, applications and servers. Unlike other messaging systems, however, Kafka has very low overhead because it does not track consumer behaviour and delete messages that have been read. Instead, Kafka retains all messages for a set amount of time and makes the consumer responsible for tracking which messages have been read. Kafka software runs on one or more servers and each node in a Kafka cluster is called a broker.

Kafka uses Apache ZooKeeper to manage clusters; the broker's job is to help producer applications write data to topics and consumer applications read from topics. Topics are divided into partitions to make them more manageable and Kafka guarantees strong ordering for each partition. Because messages are written into a partition in a particular order and are read in the same order, each partition essentially becomes a commit log that can function as a single source of truth for a distributed system's events.

B. *Kafka Streams*

Kafka Streams [2] is a library for building streaming applications, specifically applications that transform input Kafka topics into output Kafka topics or updates to databases. It lets you do this with concise code in a way that is distributed and fault-tolerant. Stream processing is a computer programming paradigm, equivalent to event stream processing that allows some applications to exploit a limited form of parallel processing more easily. Stream processing apps were most often software that implemented core functions in the business rather than computing analytics about the business.

There were two major ways streaming pipelines are built: build an application that uses the Kafka producer and consumer APIs directly, or adopt a full-fledged stream processing framework. Using the Kafka APIs directly works well for simple things. It doesn't pull in any heavy dependencies to your app. This works well for simple one-message-at-a-time processing, but the problem comes when you want to do something more involved, say compute aggregations or join streams of data. In this case inventing a solution on top of the Kafka consumer APIs is fairly involved. Kafka Streams is much more focused in the problems it solves. It does the following: Balance the processing load as new instances of your app are added or existing ones crash, maintain local state for tables, and recover from failures. This is accomplished by using the exact same group management protocol that Kafka provides for normal consumers. The result is that a Kafka Streams app is just like any other service. It may have some local state on disk, but that is just a cache that can be recreated if it is lost or if that instance of the app is moved elsewhere.

C. *Faust*

Faust is a stream processing library, porting the ideas from Kafka Streams to Python [4]. It is used at Robinhood to build high performance distributed systems and real-time data pipelines that process billions of events every day. Faust provides both stream processing and event processing, sharing similarity with tools such as Kafka Streams, Apache Spark, Apache Flink & Apache Storm. Primary objective of faust to implement Kafka Streams we can use all your favourite Python libraries when stream processing: NumPy, PyTorch, Pandas, NLTK, Django, Flask, etc.

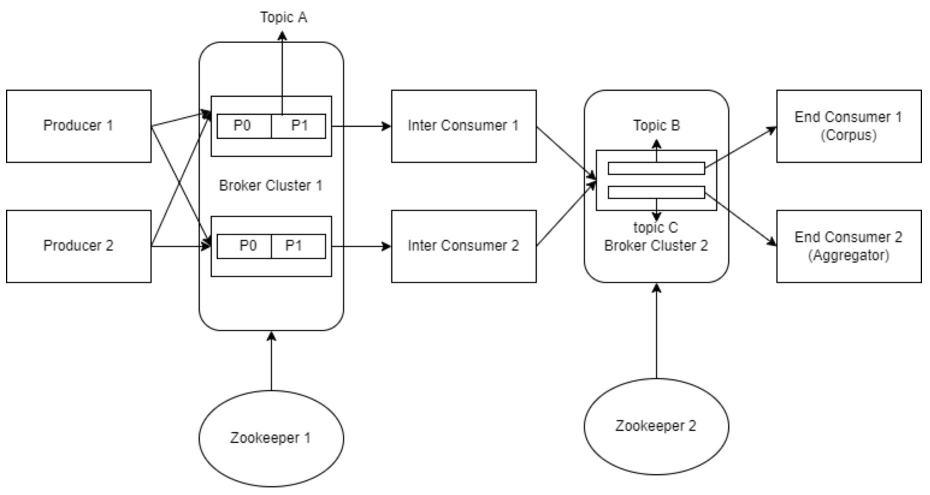

Figure 1: Architecture of Kafka Pipeline

III. Technical Approach

A. Architecture

The architecture of the project is as follows: We have multiple Kafka producer clients sending out video data as parallel streams of frames to the Kafka brokers. Multiple Kafka consumer (called InterConsumers) clients can receive this parallel sequence of frames. There are multiple brokers either running on the same machine or distributed across multiple machines. We have multiple replicated partitions (number of partitions are configurable) in each of the brokers. The number of consumer clients reading from brokers can be either less than or equal to the number of partitions, more consumers than partitions are wastage of resources. This gives us data parallelism across multiple video streams and across different frames in a stream. The InterConsumers can then perform inferencing on the incoming frames by using a pre-trained deep neural network model called Yolov5, which is explained in detail in section 3.2. The pretrained Yolov5 model takes a frame as input and outputs a list of bounding boxes along with labels denoting the objects in the frame. The output of the Yolov5 model is then again published to separate set of Kafka Brokers. Here, we have two types of Kafka consumer clients (called EndConsumer1 & EndConsumer2). Both these consumer clients will receive the output streams of InterConsumer and perform separate analytics. EndConsumer1 crops the bounding boxes out of the images and saves it in the file system along with it's label. This creates a corpus of images per label. EndConsumer2 receives the frames and keeps a moving count of vehicles per frame. Here, we use Kafka Producer & Consumer APIs to implement the entire pipeline. As a separate exercise, just to use Kafka Streams, we also implemented the InterConsumers & EndConsumer's in Faust – python stream processing library. Figure 1 shows a high-level architecture diagram of our implementation.

B. Inferencing using YOLOv5

Object detection is one of the challenging tasks of computer vision, which has been widely applied in people's life, such as monitoring security, autonomous driving and so on, with the purpose of locating instances of semantic objects of a certain class. With the rapid development of deep learning networks for detection tasks, the performance of object detectors has been greatly improved. Pre-existing domain specific image object detectors usually can be divided into two categories, the one is two-stage detector, the most representative one, Faster R-CNN. The other is one-stage detector, such as YOLO. The highest accuracy object detectors to date are based on a two-stage approach popularized by R-CNN, where a classifier is applied to a sparse set of candidate object locations. In contrast, one-stage detectors that are applied over a regular, dense sampling of possible object locations have the potential to be faster and simpler, but have trailed the accuracy of two-stage detectors traditionally. YOLO (you only look once) is a one-stage object detector proposed by Redmon et al. after Faster R-CNN in 2016. The main contribution was real-time detection of full images and videos. Initially, two-stage detectors had high localization and object recognition accuracy, whereas the one-stage detectors achieved high inference speed. However, in recent years, many newer versions of YOLO like YOLOv3, YOLOv4 and YOLOv5 have come up in literature whose accuracy is comparable and sometimes even much better than one-stage detectors [13]. In our project, we are using YOLOv5 [12]. Since, our project concerns stream-processing (fast data processing), fast inferencing like YOLO is a good choice & also we are not training the model on the dataset, rather just inferencing using it. We use the YOLO v5 model pretrained on the COCO dataset [11]. COCO is a large-scale object detection, segmentation, and captioning dataset published by Microsoft which has been the standard for object detection for a very long time.

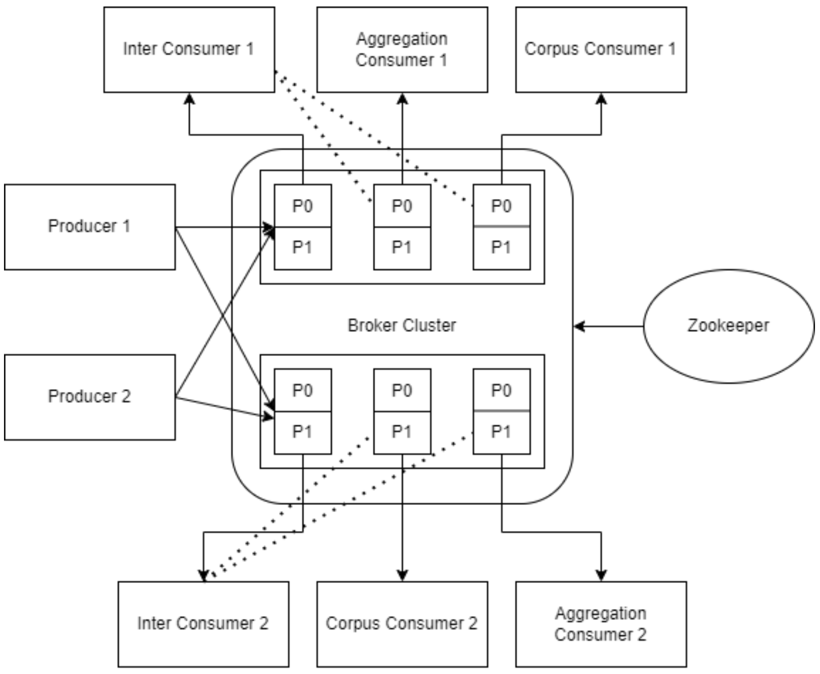

Figure 2: Architecture of Faust Pipeline

C. Datasets

The focus of this project is not on the video analytics part, rather the distributed stream-processing part. Hence, we didn't bother much on finding the best dataset for this project. However, we did consider some datasets, out of which, some we used. We looked at the VIRAT Video Dataset [6] which a diverse video surveillance dataset of over 1000 short videos but the videos in the dataset have very less activity and the activities are slow. So a lot of frames will be very similar which is not what we required. However, a few videos must be helpful but finding such videos was a lot of work so we decided not to use this. We also considered the KITTI Vision Benchmark Suite video dataset [5]. It is a autonomous driving video dataset by Karlsruhe Institute of Technology. They collected this dataset by fitting a camera over a car and moving through the local neighborhood. However, they do not video clips, rather they provide the videos in the form of sequences of images. This is not really a problem as we process the videos frame by frame anyway. So, we decided to use some videos from this dataset. We also considered Waymo Open Dataset [6] and BOSCH Boxy-Dataset [7], but Waymo dataset doesn't have raw video data, they have some pre-processed form of data and BOSCH dataset was very large in size (more than 600 GB) and the servers had very less download speed so we decided to skip these. We also considered ImageNet Video Visual Relation Dataset [8] but the video clips in this dataset are very small in length i.e., 2-3 seconds and contains mostly just one object so this was not useful for our purpose. We also looked at Road, Cars & People Video Dataset [9] by summalinguae.com but it was a paid dataset. However, we could download some sample videos from the dataset, and they were appropriate for our use case. So, we used this dataset as well.

D. Implementation

We wanted to use Yolov5 for analytics which is part of PyTorch library. Hence, we decided to use python to implement the entire Kafka pipeline. For this purpose, the most popular kafka-python package was used. It is a python client for the Apache Kafka distributed stream processing system. It is designed to function much like the official java client.

Producer: Producer was implement by importing KafkaProducer from Kafka module under the kafka-python packager. The producer consists of a pool of buffer space that holds records that haven't yet been transmitted to the server. By default a buffer is available to send immediately even if there is additional unused space in the buffer. However, the linger_ms' config setting can be set to something greater than 0 to make the producer wait before pushing. We set it to 5. So that the buffer gets full before being sent. The compression-type used is snappy. We needed to install python-snappy to use it. The input frames were sent to Topic A. In producer, we kept the input rate to be 10 frames per second. This parameter was changed during experiments to evaluate scaling performance of the system. To read video from file system & convert the video streams into sequence of frames, opencv was used. It was also used in InterConsumers & EndConsumer's to perform analytics. Also, the IP-address & the port numbers of the brokers that producer needs to connect to needs to explicitly given as a configuration parameter.

InterConsumer: Since InterConsumer is both a producer and consumer, both KafkaConsumer and KafkaProducer were imported. All the configuration parameters of both the producer and the consumer were set similar to producer. Torch was imported to use pre-trained Yolov5 model as it is a part of PyTorch library. The weights of the pre-trained Yolov5 model were downloaded and kept in the root folder as the model uses it to output bounding boxes. The results of the inferencing were sent to both Topic B and Topic C. However, since EndConsumer2 doesn't really need the frame to calculate the moving vehicle count, we refrained from sending the actual frame to Topic C, rather we just sent the bounding box and label information.

EndConsumers: We skip the details of the configuration parameters as they are similar to the above setting. In EndConsumer1, we just use open-cv tools to crop the bounding boxes with confidence of prediction greater than 0.7. This is because, we want the corpus of images created to be trustworthy for future use. We save the cropped images in the file system with filename label with a count appended to it. In EndConsumer2, we just maintained a count of vehicles, which includes all vehicle classes present in the COCO dataset like car, trucks, motorcycles, etc. Here, we only count vehicles which have confidence of prediction greater than 0.5.

We have also implemented the InterConsumer and the EndConsumers in Faust.

Faust_InterConsumer: A Faust App is created in the InterConsumer and and an agent is used to start the streaming process. The agent receives messages from Topic A and the exact same implementation for inferencing in InterConsumer is followed. The results of this inferencing are sent as messages to Topic B and Topic C.

Faust_EndConsumers: The procedure for Faust_EndConsumer1 is the same as in EndConsumer1 where the same implementation is used within the agent to create a corpus of cropped vehicle images from the given data set. In Faust_EndConsumer2 the labels of the objects in the frames are read and the count of the different vehicles is updated in a Table that is maintained across all the Faust_EndConsumer2 workers.

IV. EXPERIMENTS & RESULTS

We used only the faust implementation for performing experiments. We performed two kinds of experiments: Demonstration of weak scaling and strong scaling. For Strong scaling, we kept the input size produced by the producers fixed and run three pipelines. In all of them, we used two laptops and run the broker and zookeeper in one of them and distributed the producers and consumers in both the laptops. In the first pipeline, we used one Faust_InterConsumer, one Faust_EndConsumer1 and one Faust_EndConsumer2. We run the pipeline five times and every time, we measured the end to end time in seconds. In the second pipeline, we used two Faust_InterConsumer, two Faust_EndConsumer1 and two Faust_EndConsumer2. Similarly, in the third pipeline, we used three Faust_InterConsumer, three Faust_EndConsumer1 and three Faust_EndConsumer2. Figure 3 shows the plot of total times in each of these pipelines. As observed, the total time is approximately halved we scaling is 2x and the total time became approximately one-third when we scaling is 3x.

For weak scaling, we increase the the input size and scale the system at the same time and measure the total time in each case. Here, also we run three pipelines but we also spawned 2 producers and 3 producers in second and third case. The running times are summarized in Figure 4. As observed, the running times are almost the same every time which demonstrate strong scaling.

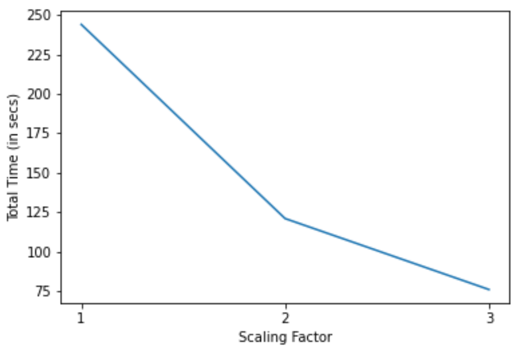

Figure 3: Demonstration of Strong Scaling

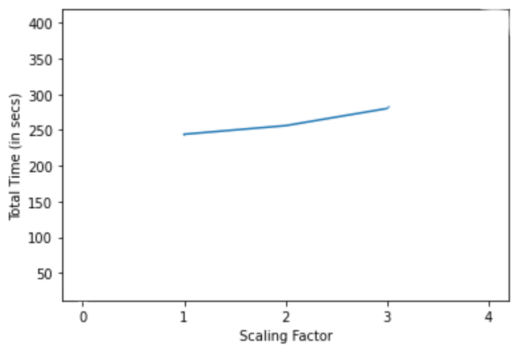

Figure 4: Demonstration of Weak Scaling

V. CHALLENGES & FUTURE WORK

One of the initial challenges we faced in this project was to run the entire pipeline in an distributed setting. We wanted to run the producers, InterConsumers and the EndConsumers in different machines. Initially when we tried connecting two different laptops via a broker, it didn't work as expected. When a broker service is spawned in any machine, it is available at some IP-address and port number. However, by default the IP-address of the broker is just localhost i.e., 127.0.0.1. We tried changing the IP-address and port number of the broker to the public IP-address in the broker.config configuration file. But still remote producers and consumers weren't able to connect. We finally figured out that we also required to advertise the IP and port which exposes this port to remote producers and consumers. This was also done by setting a configuration parameter.

As an improvement to the current project, we can perform multiple experiments to see what parameters give the best performance. For example, instead of sending one frame every time, we can try batching some images and then publish them to the broker. Batching may decrease the overall end-to-end time. We can play around with different batch sizes to find out what batch size gives the best performance. And, we can try and tune the frame input rate. Apart from that, we can also try and implement some complex analytics on the video frames. For example, something like counting only moving vehicles in a video and not the stationary ones. process multiple video frames at once. However, processing multiple frames at once in KafkaConsumer API is hard. This is exactly where KafkaStreams have an upper hand because they have aggregation functions over multiple messages. We tried implementing the same by replacing InterConsumers and EndConsumers by a faust consumers but we are struggling to use the aggregation functions because the message format of the KafkaProducer is different from what faust consumers expect. This can be a challenging problem to solve. Also, a comparision study of the Kafka Producer & Consumer pipeline implementation and the Faust Pipeline Implementation can be performed to see which framework scales better.

REFERENCES

[1] https://kafka.apache.org/
[2] https://kafka.apache.org/documentation/streams/
[3] https://kafka-python.readthedocs.io/en/master/
[4] https://faust.readthedocs.io/en/latest/
[5] http://www.cvlibs.net/datasets/kitti/raw_data.php
[6] https://waymo.com/open/data/motion/
[7] https://boxy-dataset.com/boxy/
[8] https://xdshang.github.io/docs/imagenet-vidvrd.html
[9] https://summalinguae.com/data-sets/roads-cars-and-people-video/
[10] https://viratdata.org/
[11] https://cocodataset.org/
[12] https://github.com/ultralytics/yolov5
[13] https://towardsdatascience.com/yolov5-compared-to-faster-rcnn-who-wins-a771cd6c9fb4