## Problem Statement

Currently, most stream processing systems, such as Flink, operate with a combined approach to computation and storage. For instance, in Flink, each Task Manager contains an inbuilt key-value database (RocksDB) that is used to store its state. The tasks that are deployed on this Task Manager can only access the local state store, meaning that the computation and storage are co-located. However, this arrangement poses a problem for stateful streaming dataflow operators.

State backends in Flink can be divided into in-flight state and state snapshots. In-flight state, also known as working state, is the state a Flink job is working on. It is always stored locally in memory (with the possibility to spill to disk) and can be lost when jobs fail without impacting job recoverability. State snapshots, i.e., checkpoints and save points, are stored in a remote durable storage, and are used to restore the local state in the case of job failures.

The appropriate state backend for a production deployment depends on scalability, throughput, and latency requirements. Such operators are typically partitioned based on their keys and spread across multiple workers. This means that in the event of a reconfiguration, the state of the dataflow must be migrated, leading to a pause in the dataflow. During this migration, the data must be moved from one set of workers to another, which often requires it to be transferred across a network. This process is time-consuming and results in high latency and downtime, which can have a significant impact on the overall efficiency of the system.

This problem cannot be ignored when stream processing systems are used in a variety of applications, from real-time data analytics to online advertising. The ability to perform state migrations without downtime is crucial for these applications, as any interruption to the dataflow can result in significant financial losses or reduced efficiency. When the problem is solved, a range of stakeholders will benefit, including businesses and organizations that rely on stream processing systems for real-time data analysis and decision-making.

## Problem Solution

In our solution, the computation and storage of the streaming processing system are disaggregated, and they are not located in the same place in the system. A control panel is introduced into the system to act as the information broker to distribute the location of the states of each task manager. This stand alone control panel is separate from the runtime, which means that it can maintain the location of the states without being participating into the streaming processing system.

Suppose that we want to start a new operator to process the streaming data on another location, during the sate querying state, it goes to control plane to require the location of the states rather than migrate all the states from the existing operator. Then, it can directly access the state of the existing task manager. Under this scheme, the control plane serve as a broker to distribute the states across the whole processing network.

As for the implementation of this design, Apache Kafka can be used to serve as the control plane address this problem. The following key features make it well-suited for this architecture:

1. Decoupled Architecture: Apache Kafka provides a decoupled architecture in which the producers and consumers are separated from each other without noticing the states of each other. This makes it easy to add or shrink the number of tasks and nodes without affecting the rest of the system.
2. Publish-Subscribe architecture: Kafka uses a Pub-Sub model, where tasks can subscribe to the desired topics and receive messages as they are produced. This allows tasks to receive the messages they need to process in a timely manner.
3. Scalability: Apache Kafka is a highly scalable messaging system that can handle high volume of data, making it well-suited for streaming data processing. It can be scaled up or down horizontally and can be deployed in a distributed manner.
4. Durability: The durable message storage in Apache Kafka ensures that the messages in the storage are not lost in the event of a node failure. This makes it suitable for use in a control plane in this design, where the metadata and routing information must be reliable and available whenever a task manager asks for it.

These features make Apache Kafka a powerful tool for implementing a control plane for a disaggregated streaming data processing system. By using Apache Kafka as the message broker, the control plane can efficiently manage the metadata and routing information for the tasks, allowing for a scalable, durable, and flexible data processing architecture.

## Expectations

This disaggregated control plane system for streaming processing are expected to handle requests from the task managers whenever and wherever the task manager requires. Although delay is inevitable in this architecture as the location of the states are remote and it requires the task manager to query the control plane the location of the state and then get the data of the states from the existing task managers, the system can minimize the down time when a new task manager is up comparing to traditional states-data co-located architecture. Because when a new task manager is up, it doses not have to migrate all the states of the operator to the local of the task manager. Thus, the whole system will not have to halt to wait for the transmission of the data, which significantly minimize the down time of the system. In addition, parallelism which enables the system to scale up and down horizontally is also expected. Under this manner, the operators should be able to multiply its quantity and shrink its number without affecting other operators.

## Experimental plan

#### 4.1 Environment setup

In the implementation of this disaggregated streaming processing system, we utilized a simple dataflow application that continuously read from Wikipedia edit history stream and count the number of events on each server name. For the data source, we can get the unlimited data from a single URL: <https://stream.wikimedia.org/v2/stream/recentchange>. The returned data is Json objects and we need to record the server\_name entity in the Json objects in the data in specified window size.

#### System Architecture

Once we have the environment set up, we need to implement the control plane using Kafka and basic operators including data source, data sink, map, filter, window.

In this design, the tasks implementing each operator would run independently and consume messages from the input Kafka topic, apply the desired operation, and produce messages to the output Kafka topic. The control plane would manage the state location metadata, allowing the tasks to access their state as necessary, and the data sink operator would write the output data to the desired location.

##### 4.2.1 Control Plane

The control plane would be responsible for maintaining and updating the location metadata whenever there is a change, such as a state migration or the addition of a new state store.

Firstly, when an operator is created, it would query the control plane by consuming the appropriate topic to obtain the location of its state, if there’s no such states it will act as the producer to create a new topic in the Kafka and write in the corresponding metadata for this new operator. If there’s already one topic, the information of the location of the state would be transmitted to the tasks through Kafka topic, allowing the newly generated tasks managers to access the pre-stored state data without stopping the whole streaming processing system.

Then, when the state of one particular operator is changed, it will act as a producer to write to the particular topic of the Kafka. So that after updating the state location metadata, the control panel would allow subscribers that is assigned to the topic to access the updated information in a timely manner. This process would continue as needed, with the control panel persistently monitoring the state of the system and updating the state location metadata as changes occur.

##### 4.2.2 Operators

1. Data source The data source operator would be responsible for producing the messages that represent the input data to the predecessor operators. The data source operator would connect to the data source and read the Json objects, then output the messages to downstream operators. The data source would continuously produce the messages as the data arrives, allowing the operators to process the data in real-time.

2. Data Sink Operator: The data sink operator would be responsible for output the manipulated data of the upstream operators and serves as the output the whole system. It subscribes the topic of the last operator and consume the messages produced by the tasks. Finally, it can write the data of to desired location.

3. Filter Operator: This operator would be responsible for filtering the incoming data based on a specific condition. The filter operator would consume messages from the Kafka topic and get the location of its state and apply the filtering condition to each input message and produce messages that pass the filtering condition to the output Kafka topic.

4. Map Operator: This operator is responsible for transforming the incoming data using user-defined map functions, each input element corresponds to an output element, and the entire data stream is converted to a new DataStream. Like Filter Operator, it also can consume messages from the Kafka topic and get the location of its state and then produce messages to the output Kafka topic.

5. Window Operator: This operator would be able to aggregate the incoming data over a specified time window. Like the Filter and Map operators, it can also subscribe and consume topic of Kafka control panel.

## 5. Success Indicators

#### 5.1 Outcome of this work

This novel approach where Kafka serves as the control plane and routing table would offer several advantages over the traditional co-located computation and storage in streaming processing systems

1. Scalability: By decoupling computation and storage, the system can more easily scale computation and storage independently. Tasks can be added or removed without having to move data, and state can be stored on a separate storage cluster that can be horizontally scaled as necessary.
2. Flexibility: The system would allow for greater flexibility in task placement, as tasks could be placed near their state or near the data they are processing.
3. Availability: By decoupling computation and storage, the system can tolerate failures of individual tasks or state stores without impacting the overall system.

#### 5.2 Intermediate Milestones

1. Successfully implementation of the control panel using Apache Kafka as the backend for managing state location metadata

2. Implement the operators, including Data Source, Data Sink, Map, Filter and Window.

3. Validation of the flexibility in single node deployment, allowing for tasks to be placed in local machine.

4. Validation of the scalability of the system, demonstrating the ability to add or remove tasks without having to move data or impact the overall system.

5. Deploy the system in distributed machines to simulate the real cluster environment.

#### 5.3 Measures of Success

1. Increased scalability: The system should demonstrate the ability to handle larger data sets and more tasks without a significant decrease in performance.

2. Increased flexibility: The system should allow for more flexibility in task placement, demonstrating the ability to place tasks near their state or near the data they are processing.

3. Increased availability: The system should demonstrate improved availability, tolerating failures of individual tasks or state stores without impacting the overall system.