Flink Data Stream System

Tianhua Xu
Data Science, Faculty of Math
University of Waterloo
Waterloo, ON, Canada
t72xu@uwaterloo.ca

Figure 1: Apache Flink framework for stream data processing

ABSTRACT

[1]Apache Flink is a distributed computing engine, it can be used to figure out bounded and unbounded batch processing. That is, processing static data sets, historical data sets; doing stream processing, such as processing lots of real-time data streams, data results, and event-based applications is another function of this engine. Flink is a stateful computational tool over streams, it believes that bounded datasets are special cases of unbounded datasets, which indicates bounded data sets are also a kind of event data stream, therefore everything is streams. At the meantime, people give the understanding that Flink can be used to process any data, doing batch processing, stream processing, AI, Machine Learning model training, and so on.

The popularity of Flink is inseparable from most of its feathures, including excellent performance, high scalability, fault tolerancing, and a pure memory-based computing engine that did a lot of optimizations in memory management. Flink performances well at supporting eventime processing with super-stateful jobs, with highly efficient exactly-once processing.

This project is working on implementing a Flink data stream processing program by Java, regrading on vehicle GPS data that uploaded to Kafka every 30 secs, calculate the distance of each trip that reference to the GPS coordinates. Since we already played around with Spark Streaming in Assignment 7, this report will also do comparison with Flink and Spark Streaming working procedure, to find out which technique is more suitable in certain real-world circumstances.

CS-651, Fall 2022, University of Waterloo, Waterloo, Ontario

CCS CONCEPTS

• Information systems, Data management systems; Big Data distributed computing; Flink programming

KEYWORDS

Big data, Flink, Kafka, Stateful, Spark, Data Streaming, Stream Data processing, distributed computing, query

ACM Reference format:

Tianhua Xu. 2022. Flink Data Stream System. In *(CS 651 Data-Intensive Distributed Computing), 2 pages.*

1 INTRODUCTION

With the proliferation of internet users, devices, services, etc., a majority amount of data has been generated exponentially in various business scenarios, which comes with a challenge about how to effectively process and optimize these data. In the past, people are familiar with the MapReduce, Storm, Spark Streaming and other frameworks, however they may have been unable to fully meet the requirements of users within short processing time in some scenarios, or the cost of implements is too high to take. Also, the amount of code or the complexity of the architecture occasionally may not achieve the expected standards. These new puzzles have pushed the development of new technologies: Flink, which offers a new option for processing live stream data. Apache Flink is a distributed processing framework that has been active in the big data area in past recent years. With the promotion of big data processing in North America, it is believed that it will have more advantages in the future competition. Flink's relatively simple programming model, combined with its high throughput, low latency, high performance, and exactly-once semantics,

makes it an excellent choice for industrial applications data processing technology. As many blog posts and others comments had put it, “Flink will become the dominant data processing framework for enterprises and ultimately the next standard for big data processing.”

The keyword of Flink data stream processing is [2]Stateful, which stands for stateful computation. Stateful computation is a feature that has been increasingly requested by users in recent years. To illustrate the meaning of state, for example, a website has the count of user visited in a single day, then this count of user visited can be regard as ‘state’. Flink provides built-in consistency of states, which gives strong fault tolerance: even if the task is failed over, its state will not lose, or miscounted. This consistency gives Flink a very high performance.

Apache Flink is relatively new in the area of big data streaming computing, but it has been developing rapidly and showed up on the market since 2016. This prototype project was first launched in December 2010 by Volker Markl [2] [4], a professor at the Technical University of Berlin in Germany. The main developers include Stephan Ewen and Fabian Hueske. They are aiming on a system that goes beyond MapReduce, along with Spark at the University of California, Berkeley’s AMP lab. In fact, Flink started out doing batch processing, but by then Spark has already established itself in batch processing, so Flink decided to move away from batch processing and focus on stream processing instead. Therefore, Professor Volker Markl referred to the latest paper[5] MillWheel of Google Stream computing, and decided to develop a distributed stream computing engine ‘Flink’ based on stream computing. Flink entered the Apache Incubator in March 2014 and graduated as an Apach top-level project in November 2014.

2 FLINK[1]

2.1 Flink Cornerstone

Flink’s popularity is from four of its most important building blocks: Checkpoint, State, Time, and Window.

The first is the Checkpoint mechanism, which is one of the most important features of Flink. Checkpoint provides consistency semantics by implementing a distributed snapshot of consistency based on the Chandy-Lamport algorithm[6]. The Chandy-Lamport algorithm was proposed in 1985, but it wasn’t widely used. However, Flink brings it to the next level, at the same time, Spark has implemented continuous streaming, which aims to reduce the latency of its processing and need to provide this kind of consistency. This indicates that the Chandy-Lamport algorithm has received some recognition in the industry.

In addition to providing a consistent semantics, Flink also provides a very straightforward StateAPI to make it easier for users to manage state during programming, including ValueState, ListState, Mapstate, and so on. With the recent addition of Broadcaststate, users can automatically enjoy this consistent semantics using StateAPI.

Other than above, Flink also implemented the Watermark[7] mechanism to support timely event-based processing, or system time processing, which gives the whole system more ability to tolerate data latency, delays and disorder.

Last, windowing in Flink stream computing is generally performed before operating on stream data, that is, the window type decides what to do with this calculation. Flink provides a variety of Window type in its box, such as Sliding Windows, Rolling Windows, Session Windows, and very flexible customized Windows.

2.2 Flink Architecture

Flink involves two following processes: JobManager and TaskManager, as the figure below from Flink website:

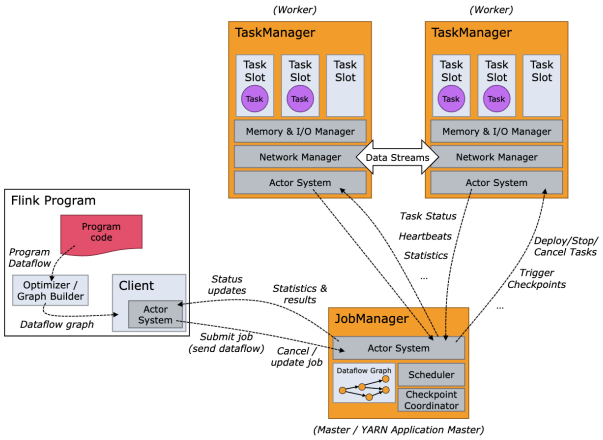


Figure 3: Flink Architecture[8]

The JobManager is responsible for scheduling tasks, coordinating checkpoint and recover the dataset from processing errors, etc. When the client submits the packaged tasks to the JobManager, it will assign the task to the TaskManager with resources according to the registered TaskManager resource information, and then start to run the task. The TaskManager gets the task information from JobManager and runs the task using available slot resources.

The TaskManager manages tasks that executes the data stream. A task may have multiple subtasks by setting the parallelism. Each TaskManager is running as an independent virtual machine process, which is mainly responsible for operators that are executed in a separate thread. The number of operators that can be executed depends on the number of slots available from each TaskManager, where a task slot is the smallest resource unit in Flink. For example, if a TaskManager has four slots, it allocates a quarter of the memory to each slot so that the according slots will not be CPU isolated. Slots within the same TaskManager share network resources and heartbeat information. Also, Flink does not allow a single task to be executed in a slot, in all cases, it is necessary to execute multiple tasks.

2.3 Flink API

The Flink API has three main layers, as shown below:

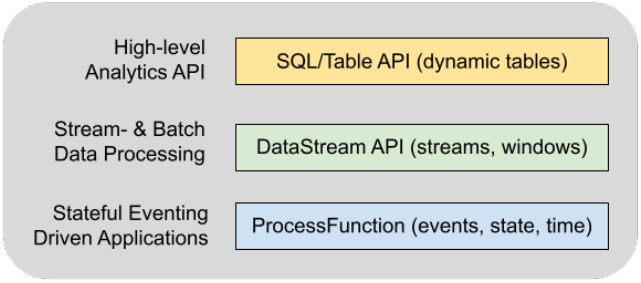


Figure 2: Flink API layers[1]

ProcessFunction is the bottom layer, which provides very flexible functionality. It can access various states to register some timers and implement event-driven applications using the mechanism of timer callbacks.

Up a level is DataStream API, it is mainly focus on data streams, is the layer of API we need to focus.

The SQL/Table API are for data analyzing, doing query after data stream are processed.

2.4 Flink & Kafka

Flink need its data source for stream processing, Apache Kafka is the most common one that is used together with Flink.

Kafka is a distributed, high-throughput, and easily scalable token-based messaging system. Kafka could work as a message queue, with built in message topic partitioning, backup, fault tolerance and other features, make it more suitable for use in large-scale, high-intensity message data processing systems.

We are mainly focus Kafka as a data source for streaming system. As the producer of Kafka, the stream data generation system distributes the data stream to Kafka message topics, and the stream data computing system (Flink in our project) consumes and calculates the data in real time.

Kafka could also be log aggregator and event sources in an event-driven system. Users can design events to a reasonable format and store them as Kafka message for real-time or periodic process by corresponding system modules. This mechanism makes Kafka a robust and efficient system that is widely used.

A detailed work combination is shown as figure below.

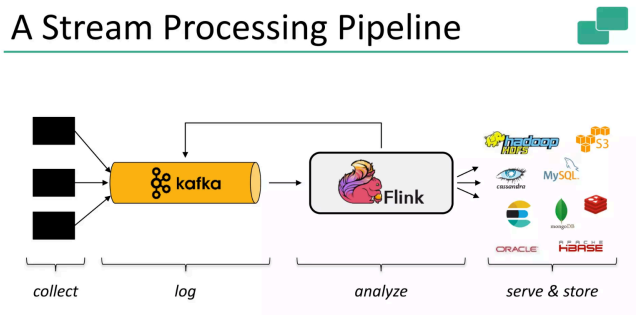


Figure 3: Advanced Streaming Analytics with Apache Flink and Apache Kafka, Stephan Ewen

2.5 Flink Application Methodology[10]

A template Flink application consists of the following components:

1. A source that reads data from Kafka (KafkaConsumer)
2. A timing windowed meeting up operation
3. A sink that writes results back to Kafka (KafkaProducer)

If the KafkaProducer supports exactly once semantics, it must write back to Kafka as event, so that all writes between two checkpoints are committed as a single transaction when the transaction was committed, it ensures that these writes can be rolled back in an accident of failure or crash.

KafkaProducer checkpointing starts in the pre-commit phase. Specifically, once a checkpoint has started, Flink's JobManager writes a checkpoint barrier to the input stream to split the messages for the current stream and next stream. The checkpoint barrier also flows between operations. For each operator, this barrier triggers the operator's backend state to take a snapshot of that state, which saves the consumption offset and passes the checkpoint barrier to the next operator once the displacement saving is complete.

The pre-commit phase is complete only after the checkpoint barrier has been passed to all operators and snapshots have been successfully completed. All snapshots created during this process are considered as part of checkpoint, which is the global state of the entire application, as well as the external state of the pre-commit. When a crash occurs, we can roll back the state to the point in time when the latest snapshot was successfully completed.

Then the system will commit to all operators that checkpoints has successfully completed. This is the second phase of the tow-phase commit protocol: the commit phase. In this phase, the JobManager initiates the checkpoint callback logic for each operator in the application.

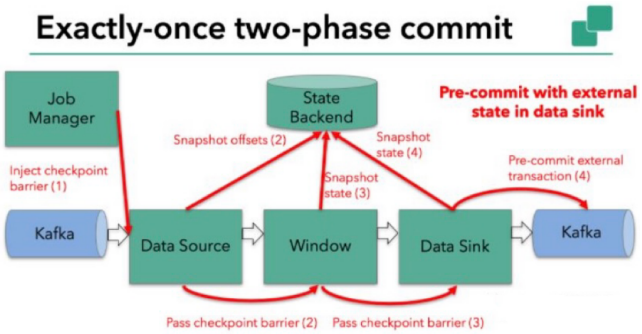


Figure 4: exactly-once work phase[1]

This architecture consists of Kafka cluster that provides data to a stream processor. The results of the stream transformation are published in Radius and can be used for applications outside the architecture. We can see from real-world usage Flink maintain low latency even at high throughput. Flink not only outperformed

the other streaming engines, but also fully loaded Kafka links at a rate of nearly 3 million connections per second.

3 EXPLORAORY FLINK DATA STREAM SYSTEM ATTEMPT

In this section, I will generally describe the Flink application I implemented for enhanced learning of above statements.

The project is working with a dataset that comes from many vehicle GPS data uploads, contains nearly 15k real time data with the following schema:

- vehicle_id: a specific string that represents a corresponding vehicle.
- paragraph: not useful here.
- time_stamp: time stamp when the data was generated.
- heartbeat_gps_latitude: the GPS latitude location when the data was uploaded
- heartbeat_gps_longitude: the GPS longitude location when the data was uploaded

We will meanly focus on the latitude and longitude coordinates of each data instance for distance calculation (with couple helper class). We will group all trip information by vehicle_id + paragraph, which can be regarded as key of this dataset, then calculate the entire trip distance by the accumulative data. For example, we have 500 coordinate records for vehicle v1, the distance between time_stamp_1 and time_stamp_2 is calculated as d1, distance between time_stamp_2 and time_stamp_3 is d2, then the distance traveled from time_stamp_1 and time_stamp_3 is d1 + d2, we do such calculation and sum each of the results together, to get the accumulative v1 trip distance. The distance between the two time_stamps are calculated by latitude and longitude.

3.1 Flink Program

I implemented Flink by Java, the main program is named **VehicleStateJob**. The entry point to a Flink program is an instance of **StreamExecutionEnvironment** class: it defines the context in which the program is executed.

Let's create a **StreamExecutionEnvironment** to start our processing:

```
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecution
Environment();
```

Note when the application is started on the local machine, it preforms processing on the local JVM. If we want to start processing on a set of computers, we will need to install Apache Flink on the machine and configure the **StreamExecutionEnvironment** accordingly.

We set up **KafkaConsumerBuilder** that reads data instance from Kafka in the beginning, checkpoint, parallelism, watermark strategy after, read and convert time_stamp by **ISO8601DateTime**, map all Kafka instance for further processing by a new **EventFormat** instance, group by key which is **deviceEventMessage.getVehicleId() + "_" + deviceEventMessage.getParagraph()**. The data stream by now is set up properly, it will read data that uploaded to Kafka, consume it. Then, it will process a **VehicleMileageAndDurationAccumulation** instance, which does appreciate calculation for all vehicles. Finally, we setup the result execution: env.execute(), and print the message to the concole.

```
env.enableCheckpointing(30 * 1000, CheckpointingMode.EXACTLY_ONCE);
int sourceParallelism = 2;
env.fromSource(KafkaConsumerBuilder.buildConsumer(parameterTool), WatermarkStrategy.noWatermarks(),
        "Kafka Source")
    .setParallelism(sourceParallelism).uid("source-id").name("data-stream-source")
        .flatMap(new EventFormat())
    .assignTimestampsAndWatermarks(
        WatermarkStrategy.<DeviceEventMessage>forBoundedOutOfOrderness(
                Duration.ofSeconds(120))
            .withTimestampAssigner(
                (event, timestamp) -> ISO8601DateTime.parse(pevent.getTimeStamp()).toInstant(
                    ZoneOffset.UTC).toEpochMilli()))
    .keyBy((KeySelector<DeviceEventMessage, String>) deviceEventMessage ->
            deviceEventMessage.getVehicleId() + "_" + deviceEventMessage.getParagraph())
    .process(new VehicleMileageAndDurationAccumulation()).name("driving-time-accumulation")
        .print();

String jobName = "Vehicle gps realtime update Job";
JobExecutionResult result = env.execute(jobName);
System.out.printf("Flink Job, jobName[%s], jobId[$s].%n", jobName, result.getJobID());
```

The **VehicleMileageAndDurationAccumulation** is a process that we need to put another eye on. This is the accumulation process program that update state for each vehicle new record. It calculates the duration and distance for each new coordinate record for each vehicle, read and update the Flink ValueState so the accumulative distance result is preserved from each data update. We can keep track of all data updates and if any unforeseen questionable result was generated, the error result would be traceable.

```
private transient
ValueState<VehicleState> deviceState;

deviceState.update(previousVehicleState
);
collector.collect(previousVehicleState);
```

Where the collector is a collect of states that collects all state updates.

3.2 Kafka Producer and Consumer

We create two jobs for Kafka communication: Producer and Consumer. Producer writes to Kafka, generates all read data instance from csv file and publishes them to stream topic using the Kafka Flink connector and its Producer API. Whereas Consumer read the same topic from Kafka, then print the message in std out using the Kafka Flink connector and its ConsumerConfig.

The **KafkaConsumerBuilder** is the consumer class I build here; it is used in the main Flink program as data source. With each vehicle state updated, we need to rely on the consumer for data update.

```
env.fromSource(KafkaConsumerBuilder.buildConsumer(parameterTool), WatermarkStrategy.noWatermarks(),
        "Kafka Source")
```

```
public static KafkaSource<String> buildConsumer(ParameterTool parameterTool) {
    Properties properties = new Properties();
    properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "node01:9092,node02:9092,node03:9092");
    properties.put(ConsumerConfig.GROUP_ID_CONFIG, "vehicle_state_job");
    properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest");
    properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true);
    properties.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, 5000);

    properties.putAll(parameterTool.getProperties());

    List<String> topics = Collections.singletonList("vehicle_heartbeat");

    return KafkaSource.<String>builder()
        .setProperties(properties)
        .setTopics(topics)
        .setValueOnlyDeserializer(new SimpleStringSchema())
        .build();
}
```

The Kafka producer program should be started in the beginning to send Kafka data, the config server node should be aligned between producer and consumer. Then producer will read the local vehicle_gps.csv file that contains all GPS data (This is for the project use, just like assignments), send it to Kafka.

```
String[] columns = line.split(DELIMITER);
JSONObject jsonObject = new JSONObject();
jsonObject.put("vehicle_id",columns[0]);
jsonObject.put("paragraph",columns[1]);
jsonObject.put("time_stamp",columns[2]);
jsonObject.put("heartbeat_gps_latitude",columns[3]);
jsonObject.put("heartbeat_gps_longitude",columns[4]);
producer.send(new ProducerRecord<>(TOPIC, jsonObject.getString("vehicle_id"), jsonObject.toJSONString()));
```

The initialization of producer is in a similar way as how we build the consumer.

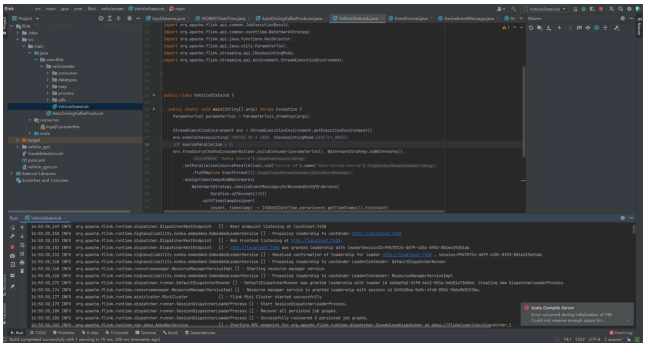
3.3 Other Helper Classes

There are couple other helper classes for the program, I separated them into different directories to indicate their different usages:

- DeviceEventMessage is for each data instance from csv file that we convert it to an object.
- VehicleState is a record for a certain vehicle state in a certain time stamp.
- GpsDistance is for calculate the distance between two sets of coordinates.
- ISO8601Datetime is for convert the timestamp format in csv file.

3.4 Outcome

We will need to execute **VehicleStateJob** and **AutoDrivingKafkaProducer** to start our program, where **VehicleStateJob** is for data consuming and processing and **AutoDrivingKafkaProducer** is for data producing. These are easy to understand. Below are couple screenshots that outputs the results after the program is executed:



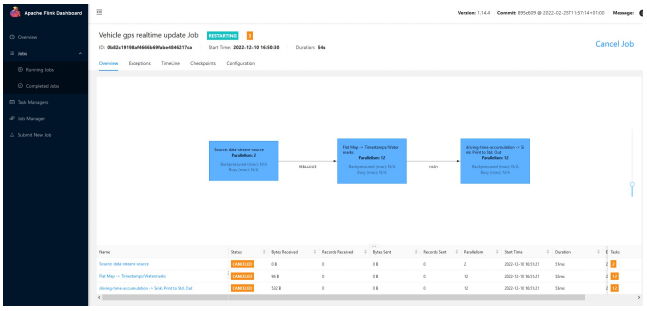
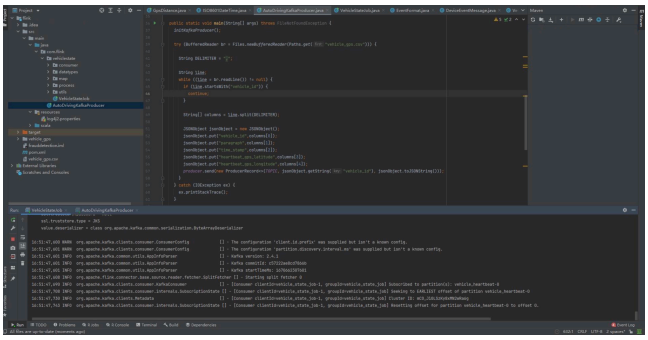


Figure 5 & 6: Execute main Flink program



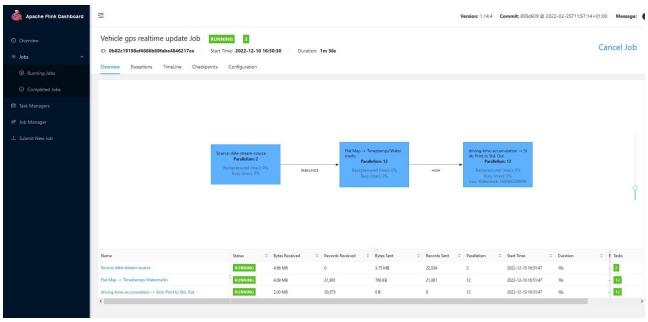


Figure 7 & 8: Execute Kafka producer and the running result from main program

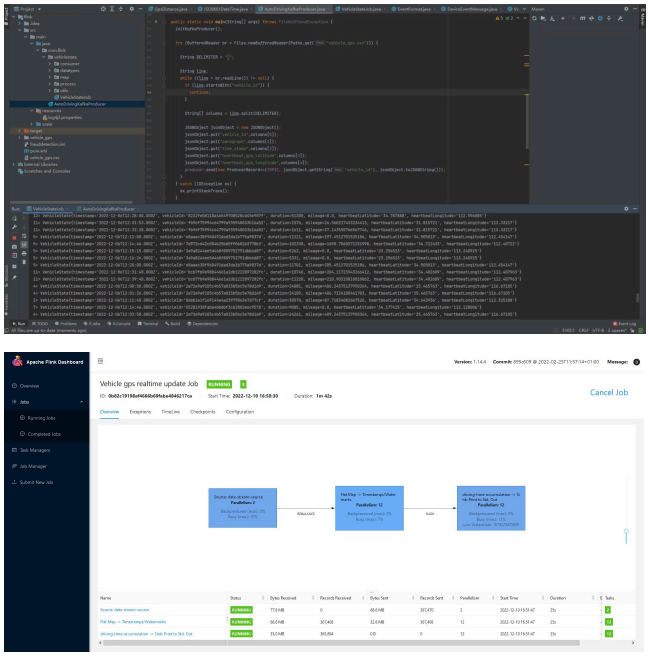


Figure 9 & 10: results after all vehicle states are processed

3.5 Conclusion & Future Task

We can easily read the vehicle trip result from stand out in console, this proves Flink could easily and efficiently process real-time data. In this project, we mocked some real-time data to simulate real world data stream process, the results showed us Flink could do this task perfectly with great fault tolerance and low latency. We will leave other Flink features like windowing, state managing, operators and other elements for future study, also, Flink SQL could be more interesting for higher level data analyzing for us to study.

4 SPARK STREAMING VS FLINK

Working together with Assignment 7, Flink has many pros and cons compare with Spark Streaming.

4.1 Data Model

The most important difference between Spark Streaming and Flink is the data model they are using for stream processing. Spark was first adopted to RDD model, which is 100 times faster than MapReduce, and greatly updates the Hadoop ecosystem. The DStream in Spark Streaming is similar to the RDD model, which divides real-time infinite dataset into a small batch data set DStream, and the timer notifies the processing system to process these microbatch data. From the other side, The disadvantages are obvious, fewer API makes it difficult to be qualified for complex stream computing business. And Spark Streaming also difficult to handle out-of-order problems, it is only suitable for simple streaming processing since low latency is not concerned too much here.

Flink's basic data model is a data stream and a sequence of events. Dataflows as the basic model of data may not be intuitively readable as tables or data blocks, but they can prove to be perfectly equivalent. A stream can be an infinite stream without boundaries, that is flow processing in the general sense. It can also be a bounded finite stream, which is called batch processing. The Dataflow model used in Flink is different from Lambda pattern, it is a pure graph composed nodes structure. The nodes in the graph can perform batch computation, stream computation or machine learning model training. The stream dataflows between nodes and is applied by the processing function on the nodes in real time. The nodes are connected, and the connection between two nodes are kept alive, after logical and physical optimization, the Dataflow logical relationship is not very different from the physical topology at runtime. In this pure streaming design, the latency and throughput are theoretically designed to optimized.

4.2 State Storage

The Spark Streaming APIs are focusing on the basic capability of RDD. After a snapshot is periodically enabled, the entire memory data is persisted at the same time. Spark is used for computing large datasets, while Flink provides file, storage and value three types of state storage, for asynchronous persistence of running status data. Flink also support incremental snapshots, which can undoubtedly reduce network and disk overload in the face of large memory status data, and the Flink stream data could also be stored in many storage options.

4.3 Conclusion

Flink is a newer technology compared to Spark Streaming, while spark is more mature and widely used. In addition, spark has more supporting host that learners can easily found many existing cases and examples of best practice that shared by other users. Although Flink is not mature, it is useful for complex event processing or native streaming cases because it provides better performance, latency and extensibility. Flink also has better support for window and state management, can perform many raw operations that require custom logic development in Spark streaming.

ACKNOWLEDGMENTS

This project is supported by CS 651 Data-Intensive Distributed Computing, University of Waterloo. Thanks for Dr. Dan Holtby for providing advice for this project. I will be graceful to the online resources from Apache Flink and Apache Kafka, together with Youtube Flink tutorials at https://www.youtube.com/watch?v=aWJ7CkSKMpQ.

REFERENCES

[1] https://flink.apache.org/ (And it's following subdirectories)
[2] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., & Tzoumas, K. (2015). Apache flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 36(4).
[3] Rabl, T., Traub, J., Katsifodimos, A., & Markl, V. (2016). Apache Flink in current research. it-Information Technology, 58(4), 157-165.
[4] Imran, M., Gévay, G. E., & Markl, V. (2020). Distributed graph analytics with datalog queries in Flink. In Software Foundations for Data Interoperability and Large Scale Graph Data Analytics (pp. 70-83). Springer, Cham.
[5] Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., ... & Whittle, S. (2013). Millwheel: Fault-tolerant stream processing at internet scale. Proceedings of the VLDB Endowment, 6(11), 1033-1044.

[6] Chandy, K. M., & Lamport, L. (1985). Distributed snapshots: Determining global states of distributed systems. ACM Transactions on Computer Systems (TOCS), 3(1), 63-75.
[7] Akidau, T., Begoli, E., Chernyak, S., Hueske, F., Knight, K., Knowles, K., ... & Sotolongo, D. (2021). Watermarks in Stream Processing Systems: Semantics and Comparative Analysis ofApache Flink and Google Cloud Dataflow. Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States).
[8] https://nightlies.apache.org/flink/flink-docs-release-1.16/docs/concepts/flink-architecture/
[9] Javed, M. H., Lu, X., & Panda, D. K. (2017, December). Characterization of big data stream processing pipeline: a case study using Flink and Kafka. In Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (pp. 1-10).
[10] https://flink.apache.org/features/2018/03/01/end-to-end-exactly-once-apache-flink.html