BluePrint for Software Decoding Through Documentation Analysis

Conceptual Architecture

By: Analysis Paralysis

Authors:
Aisha Mahmood - aisha895@my.yorku.ca
Harsimran Saini - saini19@my.yorku.ca
Esha Vij (218123307) - eshavij@my.yorku.ca
Harrish Elango - elangoha@my.yorku.ca
Adewusi Fashokun (218329540) - korede@my.yorku.ca
Siddhanth Bakshi - sbakshi@my.yorku.ca

Abstract

This conceptual architecture report provides a thorough overview of Apache Flink, specifically version 1.17.1. Apache Flink is a powerful framework for distributed data processing. The report gives a high-level overview of the system's structure, functionality, and interactions among its parts. The report examines Apache Flink's essential components and the overall functionality of the system. The report explores Apache Flink's evolution, discussing how Flink adjusts to changing requirements and technological advancements. The report also mentions how data flows among its components and, offering insights into how a directed acyclic graph (DAG) manages and processes data. Moreover, Flink's concurrency is examined, showing the distributed and parallel processing features that enable Flink to handle large amounts of data. In addition, the report discusses the implications for division of responsibilities among participating developers. The report also includes use cases on how Apache Flink can be used to achieve specific outcomes. To facilitate a newcomer's understanding of Apache Flink, the report includes sequence diagrams to show the messages passed between Flink's objects. The sequence diagrams give a visual picture of the system's interactions and processes, which makes it simpler to understand its functionality.

Introduction and Overview

Apache Flink, which is an open source software system, is a distributed processing engine and framework for stateful operations across **bounded** and **unbounded** data streams. Flink has been developed to operate in all typical cluster environments and conduct computations at any scale and in-memory speed. Flink offers a variety of capabilities that in today's data-driven environment, the demand for strong and scalable data processing frameworks is more pronounced than ever. Apache Flink is an adaptable distributed data processing framework that has become a key response to this expanding need. This report begins with a thorough investigation of the conceptual architecture that supports Apache Flink and followed with providing a high-level overview of this technology. This report is especially useful for anyone looking for an in-depth understanding of Apache Flink. Both novices and experienced individuals can utilise this report to gain the conceptual architecture of Apache Flink. By the end of this report, readers will have a better understanding of the fundamental concepts and procedures that make Apache Flink an essential tool for a variety of data-driven applications. The report offers the crucial insights needed to successfully navigate through the world of distributed data processing, regardless of the reader's level of expertise with the system.

Architecture

When dealing with the transfer of large volumes of data, it is essential to carefully consider the methods employed and strategies for cost reduction. In this context, stream processing emerges as a critical approach, offering the advantage of reducing data transmission while enabling real-time responses to new data events. This capability greatly simplifies the task of integrating data from multiple operational systems by providing a common architectural foundation.
Streaming data holds immense significance in enhancing organizational insights and delivering superior customer services. Instant data accessibility has become an expectation of modern consumers, driving the need for efficient data processing. A stream represents an uninterrupted sequence of events, lacking distinct starting or ending points. To handle such continuous data streams effectively, a system must be capable of receiving and processing new elements continuously and promptly. Apache Flink addresses this requirement by providing the DataStream API, which empowers applications to work seamlessly with unbounded streams of data. Additionally, some event streams may necessitate storage for later reprocessing, forming bounded streams with well-defined beginnings and ends. For handling bounded data sets with specific components, Apache Flink offers the DataSet API, enabling users to iterate over these elements effectively.

The Apache Flink framework finds widespread adoption among a diverse range of companies seeking to analyze streaming data at a massive scale in real-time. What sets Flink apart is its adaptability, offering support for both stream and batch processing frameworks. It boasts a rich ecosystem with APIs available in Java, Python, and SQL, providing users the flexibility to choose the programming language in which they are most comfortable and productive.

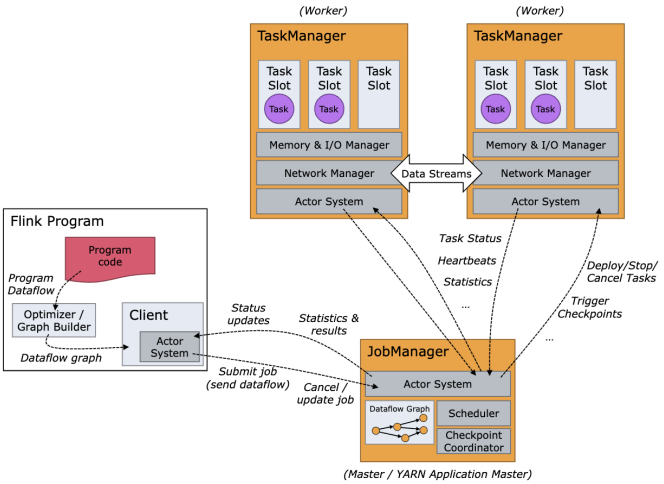
Flink is based on a dataflow engine which does not allow it to have its own storage layer and makes use of external storage systems with a set of connectors, that gives Flink the advantage to process data from any source. Apache Flink's architecture consists of a distributed execution engine that supports various workloads, including batch processing, streaming, graph processing, and machine learning. In the next layer, there’s a deployment management layer, facilitating both local and distributed deployment modes for test/development and production use. This layer includes components like Flink-runtime, Flink-client, Flink-web UI, Flink-distributed shell, and Flink-container, working together to manage the deployment and execution of Flink applications in a distributed cluster.

Flink kernel is the main framework of the Apache Flink. The distributed processing, fault tolerance, reliability and native iterative processing capability is handled in the run-time layer. Flink-tasks are handled by the execution engine which are components for distributed computations scattered over various cluster-nodes.

In the Apache Flink architecture, JobManager and TaskManager are crucial components responsible for managing and executing Flink applications. They play distinct roles in the distributed data processing framework. Here's an overview of each:

1. JobManager is the master node in the Flink cluster. It is responsible for overseeing the entire Flink application. Its primary role is to manage and coordinate the execution of Flink jobs, which represent the data processing tasks within an application. JobManager schedules and manages the tasks submitted to Flink, orchestrating their execution efficiently. It constructs a plan for executing the tasks, ensuring that they are distributed across the available resources in the cluster. JobManager handles the high-level control, job coordination, and resource allocation aspects of Flink applications.

2. Task Manager nodes are responsible for executing the actual data processing tasks on the provided resources. They work on distributed cluster nodes and are responsible for executing user-defined functions. Each Task Manager is assigned a set of tasks by the JobManager, and it processes these tasks in parallel. Task Managers are designed to handle the actual data processing workload, making them responsible for executing the code that transforms, filters, and computes results on the data. They provide the necessary computational resources, such as CPU and memory, for the tasks they execute. [5]



To illustrate the practical applications of Apache Flink, consider its deployment in the Uber application. Uber relies on this software to efficiently match drivers with riders and calculate estimated arrival times, delivering a seamless user experience. Another prominent example is Netflix, where Flink plays a crucial role in generating personalised content recommendations based on users' viewing history and recent selections. Furthermore, Reddit employs Flink to safeguard its user community from harassment and spam, showcasing its versatility in addressing diverse operational needs. [4]

Evolution of Apache Flink

As Apache Flink is an open-source product, it has a large community of dedicated developers who actively help to make the product faster and more efficient. The software evolves as changes are made and released through regular release cycles of updated versions. The updates are documented in the roadmap for the product, as it lets any potential developers know where they can help contribute to making the product better.

Apache Flink has a community area where developers and users can vote on what improvements to the software are of the highest priority, and therefore, what to include in the roadmap of the software. This is facilitated through Flips, or, Flink Improvement Proposals. Flips allow for a central place for the community behind Apache Flink to collect and document any planned changes or improvements for Apache Flink at a high level and also allow developers to see what changes are currently the main focus for evolving Apache Flink. These Flips can be created by anybody and are then voted on by the community to determine if the proposals are accepted to be used to improve Apache Flink. These Flips are considered to be more important than changes in actual code as they lay the foundation of the future roadmaps that dictate the evolution of Apache Flink, as it is what developers use to identify which parts of the product they can work on improving.

As Flips are a high-level overview of what parts of the software the community is currently improving, contributing code to the Apache Flink software as a developer is done using a different approach called Jira tickets, which are used to “track tasks, bugs, and progress” of changes to Apache Flink.

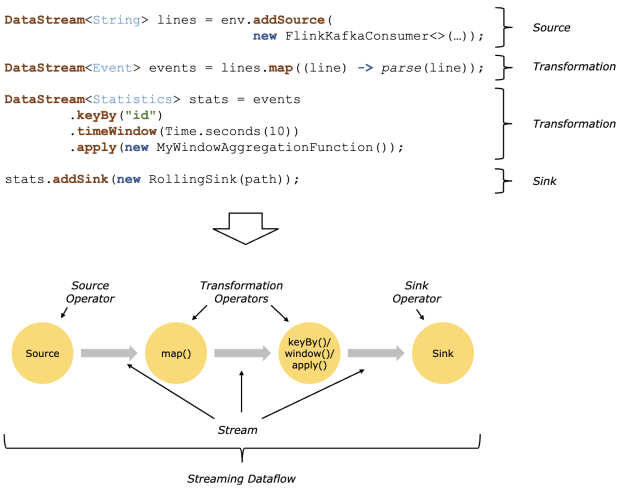
First, the developer must go to the Apache Flink community, usually done in the Apache Flink mailing list discussions, and reach a consensus regarding the scope, importance, and implementation approach of their proposed code change. In normal cases, the discussion happens in “Flink’s bug tracker: Jira” forums, but more serious or controversial changes are discussed in the “Flink Dev mailing list”. If a proposed change is of a much larger scope than what is normally contained in a Jira ticket, it is possible that it could be converted into a Flip to allow for a better understanding of said changes and to break down the tasks required to complete the proposed change. Only after the discussions have concluded can the Jira ticket be made, as the community will have to find a “committer” to issue you the Jira ticket to allow you to start working on the proposed improvement, as they are the only members of the Apache Flink community that can actually merge your new implementations into the existing source code of the software.

Once your Jira ticket has been issued, you are able to work on implementing the improvement by working on a clone of the source code in a “Flink development environment”. When making the changes to the code, you will have to follow the Code Style and Quality Guide, which helps developers make changes to the code in a way that future developers would easily be able to decipher what the code does and make changes to it in the future, as well as following the approach discussed earlier with the community in the forums. When you have completed your implementation, you must test the changes made to the code to make sure that the software builds successfully and passes all tests provided by the community, proving that the changes did not affect any other parts of the software.

After the testing is done, you can open a pull request, where the code will be reviewed to make sure that it adheres to the approach discussed in the forums, the Code Style and Quality Guide, and targets the issue assigned to the ticket. At the conclusion of the review, if the changes to the code meet all the requirements asked of it, then the “committer” assigned to you can merge your improved code to the Flink source code and close your Jira ticket, helping evolve Flink further.

The Flink community has steadily been making progress towards improving Apache Flink and is making efforts to upgrade Flink to a 2.0 version, where DFS will be used as its primary storage system, changes will be made to better support the current cloud-native era, and API's will be upgraded or removed, among other changes to help improve Flink's performance, stability, and usability now and into the future.

Streams and **transformations** are the fundamental building components of Flink programs. When Apache Flink programs are run, they are translated into streaming dataflows made up of streams and transformation operators. Each data flow has one or more sources at the beginning and one or more sinks at the conclusion. Indeterminate directed acyclic graphs (DAGs) are similar to the dataflows.

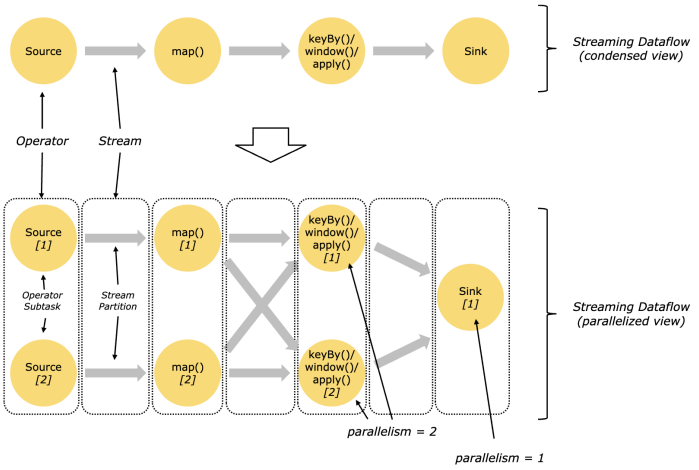


This code snippet is a simplified example of a data processing pipeline. The "keyBy() functions to partition the incoming data stream into parallel streams to introduce parallelism in the pipeline and the "process()" function calls an add operator that applies a function to each partitioned element in the stream. [3]

Apache Flink programs are distributed and parallel by nature. A stream may have one or more stream partitions throughout execution, and each operator may have one or more operator subtasks. The operator subtasks run in various threads, are independent of one another, and may execute on different machines. The **parallelism** of a particular operator can be identified by counting the number of operator subtasks. The parallelism of a stream is determined by its producing operator, and varying operators of the same program may have different levels of parallelism.

Data can be transported between two operators using streams in a one-to-one or redistribution pattern:

- One-to-one streams maintain the elements' division and order.
- Redistribution pattern streams change the partitioning of the streams.



In a basic Flink application program, the programmer can define the following:

- Sources from which the data will be ingested, which can be one or more.
- Operations on the data in succession, including computations with and without states.
- Operation parallelism level (degree) to speed up computation
- One or more sinks to send the output of the computation

As data flows into Apache Flink, it is processed through a directed acyclic graph (DAG) of operators, including ingestion, transformations, stateful processing, windowing, and output sinks. The dataflow graph is one that consists of nodes representing the operators or tasks and edges defining the inputs/outputs and relationship between the nodes. By coordinating the execution of this data flow, the control flow makes sure that data is processed in accordance with the specified logic and fault tolerance requirements. Flink is able to process massive amounts of data effectively in both real-time and batch processing scenarios due to its parallel and distributed data flow design. Here is an overview of how what occurs at each step of a DAG and how data is processed:

- **Data ingestion:** The system receives data from a variety of sources, including file systems, Kafka, and Apache Pulsar. These sources are read for data by Flink's source connectors, which then results in the creation of data streams or datasets.
- **Data Processing and Transformation**: A sequence of data transformation and processing procedures are performed on data streams or datasets through Flink’s DataSet APIs or DataStream. Moreover, data transformation can occur through defining specific operations such as mapping, filtering, windowing and aggregation.
- **Stateful Functions:** It is an API that simplifies the creation of building stateful applications. Stateful is defined as when operations remember information across multiple events (ex. window operators). The state information is managed and updated as data is flowed through them When stateful operations occur, the data streams maintain their state.

- **Event Time Processing:** Data streams can be dependent on event time. This is done by assigning a timestamp to the data elements, followed by data processed according to the timestamp. If this is not the case, the system is able to handle out-of-order events.
- **Windowing**: Data can be divided into windows according to their assigned timestamps, or other criteria. When data is divided into windows (aka as windowed data), the data is processed in separate units.
- **Output (sinks):** The data is processed into its final destination, called output sinks which can be databases, external systems, or file systems.

By coordinating the execution of this data flow, the control flow makes sure that data is processed in accordance with the specified logic and fault tolerance requirements. Here is an overview on the how the control flow ensures this:

- **Job coordination and execution:** The data flow through the system is determined by the DAG, which is the execution plan. The JobManager coordinates the complete execution of the data, from start to finish.
- **Fault Tolerance and Checkpointing:** In order to guarantee fault tolerance and data consistency, data streams and state are periodically checkpointed. Throughout the data flow, checkpoints record the condition of the operators and data streams.
- **TaskManager Execution:** The JobManager sends tasks to TaskManagers, who then carry out their actions on the designated data partitions. TaskManagers work in parallel to process data.
- **Scaling:** By redistributing jobs or adding or removing TaskManagers, data flow can be scaled dynamically. This guarantees that data is processed effectively and evenly across all resources.

Division of responsibilities among developers ensures that the process of building, maintaining and debugging code can be handled with maximum efficiency and minimal interruption to development velocity when a developer is a victim of an unfavourable event.

When building a complex and densely-featured software project like Apache Flink, there are a number of benefits that dividing responsibilities among developers brings. The first is that the focus of individual developers on specific portions of the application ensures that maximum attention is given to even the most minute details of that part of the architecture. Developers are able to create implementation details that might not have been considered in the high level conceptualization of the project. In many cases, these details help to prevent near-undetectable issues in the future, or provide benefits that are useful in other portions of the system, and by extension, the overall structure. This also means that these developers grow to become domain experts, improving the quality of work delivered and their ability to train newer members of the team.

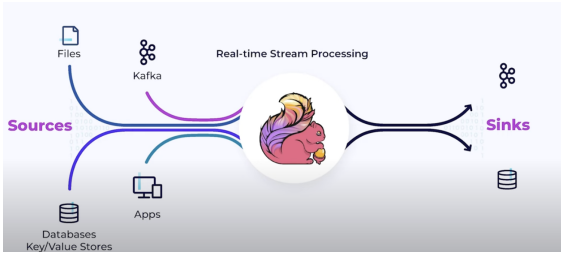
Another merit of this approach is its usefulness in coming up with time estimates and budgets for feature development. While it is notoriously difficult to estimate the time required to reach certain milestones in the development process, breaking up the project into smaller constituent parts and assigning them to developers makes it much easier to visualise or predict the time required to complete them. Similarly to parallelizing processes, it also makes it possible to build these pieces at the same time, as opposed to sequentially. In the same vein, budgeting resources (monetary or otherwise) required for development becomes simpler as the requirements are much clearer.

While all the aforementioned are positive implications of the division of responsibilities among participating developers, this is not a silver bullet, as there are a number of considerations to be made. Overspecialization of developers leads to information about specific parts of the system being hidden from members of the team, leading to a very high **bus factor.** It can also lead to **premature optimization**, since the fixation of participating developers on their area of specialty leads to implementation of features that are not necessarily required for the system's primary functionality. This is important because it takes away from the focus on the core use cases and also opens potential attack vectors. [6]

Concurrency

Event streaming captures events in real-time, from online orders and shipments to downloads and clicks. Therefore, business events can always be streamed. There are two types of streams, unbounded and bounded streams. An unbounded stream is manipulated, processed, and reacted to in real-time and the sequence of events forms an unbounded stream which extends indefinitely into the future. A bounded stream is where the streams can be stored for later retrieval and reprocessing. A bounded stream is time stamped with a starting and ending. Stream processing supports bounded and unbounded, but in batch processing only bounded streams are present.

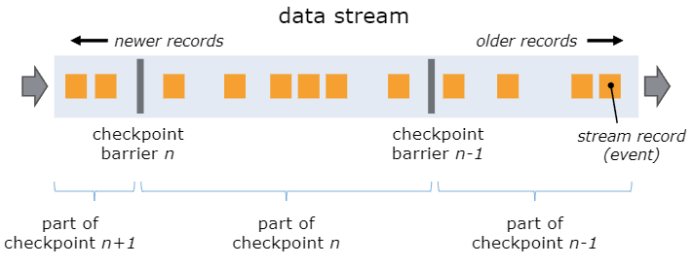
Flink applications consume data from one or more sources and produce sinks. But the main focus is what happens in the middle, between the sources and sinks. In the middle is where the stream processing happens. To execute a Flink cluster, a developer will need to use Flinks API by defining their business logic.



Each state of the node during the processing is checked via a snapshot. There are two types of snapshots – checkpoint and savepoint.

A checkpoint is an automatic snapshot, which is used for the purpose of failure recovery and is created by Flink. These checkpoints are written in a format that is optimised for quick recovery. All checkpointing can be done asynchronously and snapshot their own state as the checkpoint barriers don't travel in lockstep.

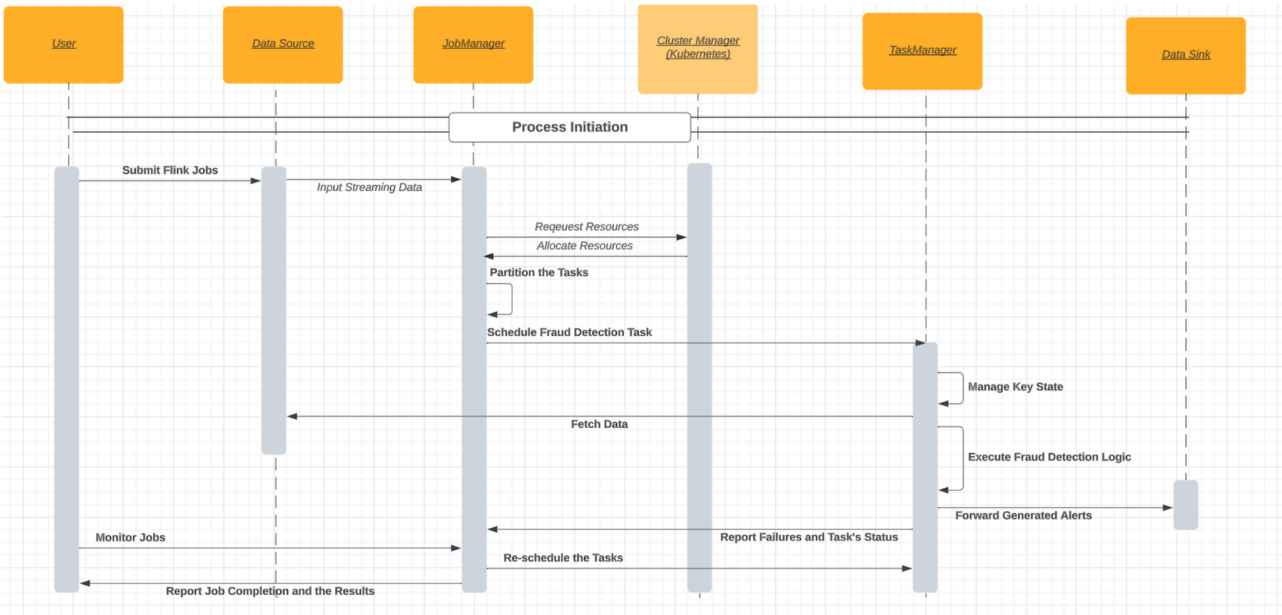
These barriers, also known as stream barriers, are a core element in Flink's distributed snapshotting. The barriers are inserted into the data stream and flow with the records as part of the data stream. They flow in a queue meaning they will not overtake records. In the data stream, the records are separated by the barriers. Every barrier carries its own ID of the snapshot. So checkpoints do happen in concurrence as multiple barriers from different snapshots can be in the stream at the same time. [2]



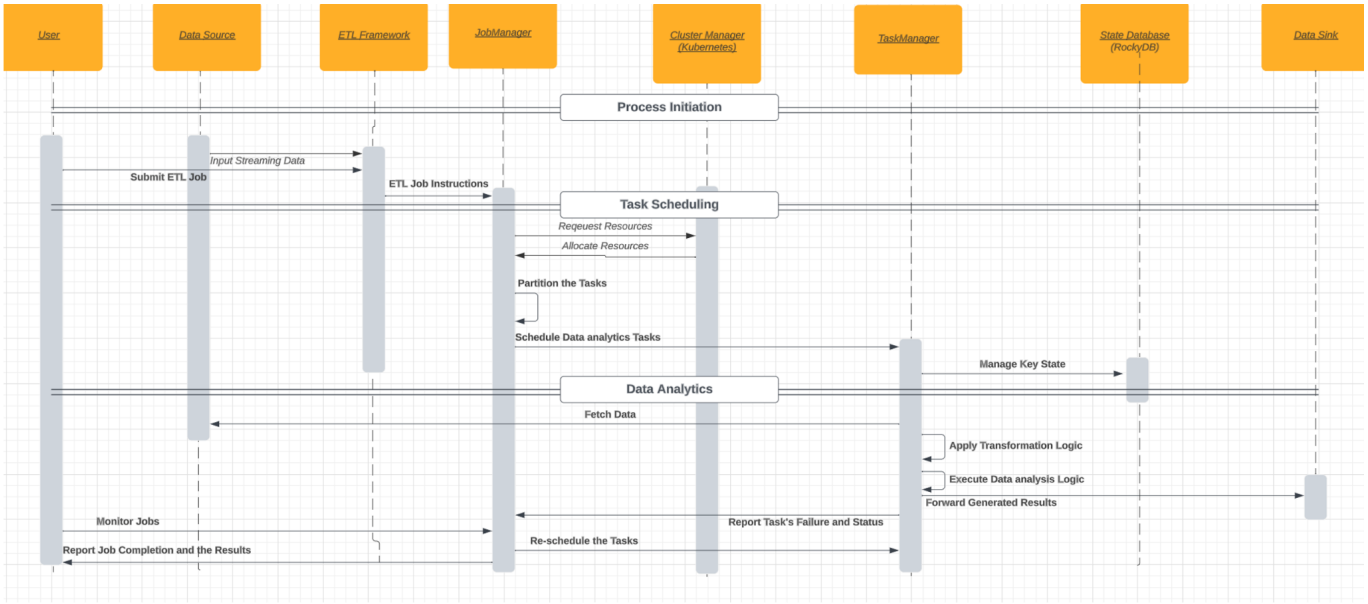
A savepoint is a manual snapshot and is written in a format that optimises operational flexibility and is created for some operational purpose such as upgrading to a new version of Flink or a new version of your application.

When a processing node of a Flink cluster fails, this can also be in parallel streaming processing as well. But when they fail, Flink needs to somehow resume the processing without making mistakes. Some common mistakes Flink might make are unexpected results or duplicates. Flink runtime does not experience a huge hiccup when these snapshots are happening, instead, the state is stored in data structures and supports multi-version concurrency control, meaning that Flink can continue to process events and produce results in new versions of states while the failure ones are still being processed by them being snapshotted.

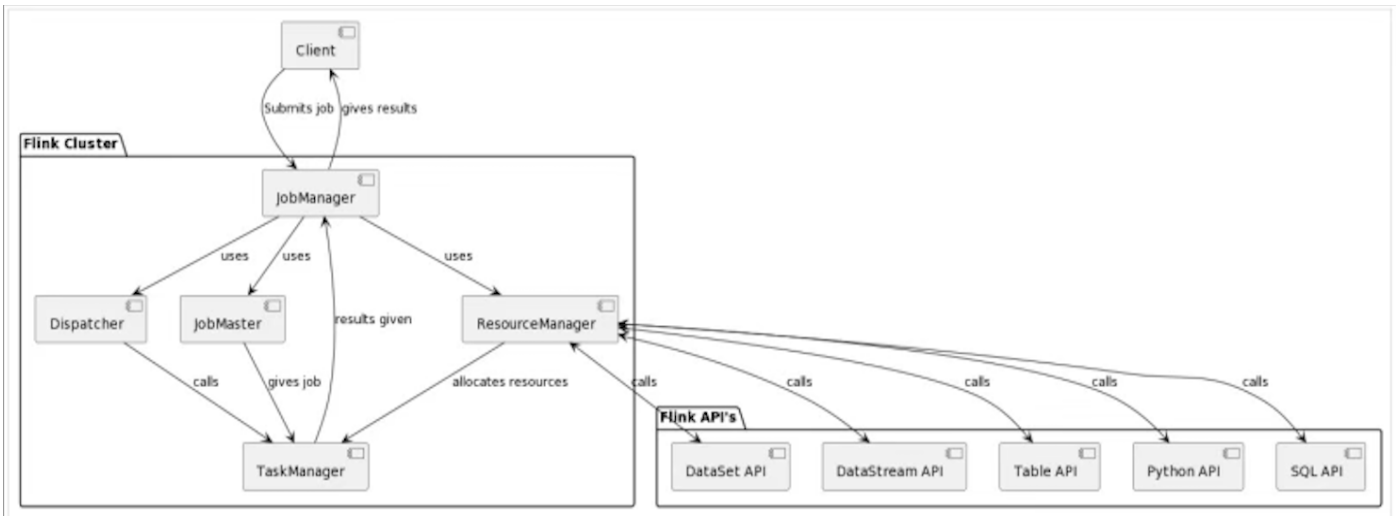
Diagrams
Sequence Diagram 1:



Sequence Diagram 2:



Component Diagram:



External Interfaces

1- Kafka Connector: Transmits data Streams from Kafka Topics to Kafka Topics and serves as a data ingestion and export interface.

2- Hadoop Integration: Transmits data through Hadoop Integration interface(structured data, batch processing tasks and huge datasets stored in Hadoop Distributed File System)

3- Restful API interface : Interface serves as a communication channel to send and receive data from Apache Flink Clusters.

4- JDBC Connector: Enables Flink to read and write data from databases.

Design Patterns:

1. Observer Design Pattern: The support for event-driven stream processing and event-time processing is where the observer design pattern is quite evident. The occurrences are monitored and analysed as they occur, causing actions to be taken in reaction to these occurrences.
2. Master-Slave Design Pattern: The JobManager performs functions within Flink that correspond to the idea of a “master.” The JobManager is responsible for organising, and coordinating the execution of jobs within Flink. The TaskManagers perform functions within Flink that correspond to the idea of a “slave.” TaskManagers receive tasks to complete from the JobManager for parallel execution.
3. Visitor Design Pattern: The chaining of numerous functions within a processing pipeline is where the visitor design pattern is visible. As data items move through the pipeline, the functions in the chains are able to modify or transform the data items.

Use Cases

Use Case 1: In this particular use case, a bank sought to bolster its fraud detection capabilities by swiftly identifying unusual transaction patterns for individual customers and promptly flagging any potentially fraudulent transactions. To achieve this goal, the bank's development team turned to Apache Splunk, a robust data processing tool, to effectively manage the continuous flow of transaction data and assist in the detection of fraudulent activities.

Use Case 2: An e-commerce website is looking to streamline its operations, enhance the customer experience, and make more informed decisions through the utilisation of data analytics. To achieve these objectives, the website plans to employ Apache Spark for the analysis of the constant data flow generated by user interactions on the platform. [7]

Data Dictionary

- Bounded data streams: The beginning and end of bounded data streams have been identified. By ingesting all data prior to conducting any computations, bounded streams can be processed.
- Unbounded data streams: These data streams have a clear beginning, but not a clear end. They continue and offer data as it is being produced. Unbounded streams need to be processed continuously.
- Streams: A stream is a flow of data records, which could possibly be endless.
- Transformations: A transformation is an operation that accepts one or more streams as input and outputs one or more streams as a result.
- Parallelism: The number of parallel instances of a task.
- Bus Factor: This means that the sudden incapacitation or unavailability of a developer drastically impacts the ability of the team to function smoothly, meet deadlines and mitigate issues.
- Premature optimization: Entails attempting to enhance something when it is too early to do so, especially with the intention of perfection.

Conclusion

In conclusion, Apache Flink's conceptual architecture offers a strong framework for understanding the idea that drives the success of this distributed data processing platform. As we look to the future, Apache Flink will remain at the forefront of data processing due to its ongoing development in developer resources, community collaboration and monitoring tools. Moreover, the integration with emerging technologies will keep Apache Flink a prominent tool of data processing. This report acts as a guide for both novices and experts in the field looking to take advantage of Apache Flink's features. The fundamental building blocks and procedures that support Flink have been demonstrated through this report. Moreover, this report has provided new information on how Flink works, develops, subsystems, deals with concurrency and the division of responsibilities among developers. The following is a summary of our key findings:

1. Flexibility: Apache Flink stands out as a flexible and versatile option for processing data in both batch and real-time applications.
2. Adaptability: The architecture of Flink is built to accommodate growing data and adapt though changing requirements. It is Flink's adaptability that demonstrates the relevance it will have in the future.
3. Utilising sequence diagrams to visualise interactions among systems: Sequence diagrams have been crucial in giving a visual picture of the interactions between Flink's components during particular processes. They provide useful perceptions into the operation of the system.

The following are our proposals for future directions:

1. Promoting community collaboration: As Apache Flink is open-source, the community plays a vital role in its development. Community collaboration can foster new use cases for Apache Flink, hence enhancing its functionality.
2. Emerging technologies integration: As technology is constantly evolving, it is important to keep up with it to remain relevant.
3. Providing more essential tools for developers: One of Apache Flink's future goals could be to provide even more tools, documentation and other resources to speed up process development and promote the education of all of its users.

Lessons Learned

As a group the lessons that we can apply going forward is to enhance our knowledge of complex systems. It taught us the importance of designing and architecture of our projects and how the outcome of the project can be inspired by creative solutions. By understanding the complexity of Apache Flink's architecture, our group has grasped a different perspective of distributed data processing. This presentation introduced concepts regarding data streaming, transformations and event time processing which is a concept that will come in handy while working on real-time data processing projects.

References

[1] “Architecture,” Apache Flink, https://flink.apache.org/what-is-flink/flink-architecture/ (accessed Oct. 13, 2023).

[2] “Stateful Stream Processing,” Stateful Stream Processing | Apache Flink, https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/concepts/stateful-stream-processing/ (accessed Oct. 8, 2023).

[3] “Dataflow programming model,” Apache Flink 1.2 Documentation: Dataflow Programming Model, https://nightlies.apache.org/flink/flink-docs-release-1.2/concepts/programming-model.html (accessed Oct. 14, 2023).

[4] “Introduction | apache flink 101,” YouTube, https://www.youtube.com/watch?v=3cg5dABA6mo&list=PLa7VYi0yPIH1UdmQcnUr8lvjbUV8JriK0&index=1 (accessed Oct. 9, 2023).

[5] “Flink architecture,” Flink Architecture | Apache Flink, https://nightlies.apache.org/flink/flink-docs-master/docs/concepts/flink-architecture/ (accessed Oct. 12, 2023).

[6] H. Temme, “3 reasons why you need apache flink for stream processing,” The New Stack, https://thenewstack.io/3-reasons-why-you-need-apache-flink-for-stream-processing/ (accessed Oct. 10, 2023).

[7] Wojciech-Gebis, “What is Apache Flink? architecture, use cases, and benefits,” nexocode, https://nexocode.com/blog/posts/what-is-apache-flink/#:~:text=Apache%20Flink%20is%20an%20open,and%20expressive%20data%20processing%20capabilities. (accessed Oct. 11, 2023).