Pravega OLAP

Integration of Pravega and online analytical processing (OLAP) database with Java

Project Testing and Accepting Plans

Dell Technologies

Jose Robles, Nate Tsige, Boxiang Lin

10/28/2022

Contents

Project Description

I. Introduction

Dell Technologies currently takes charge of an open-source project that is known as Pravega. This is an infrastructure that serves as a storage system that implements data streams to store/serve data [1]. These data streams are made up of sections which contain events. These are sets of bytes in a stream that represent some sort of data – Pravega is effective at storing/ingesting these due to its data streams being consistent, durable, elastic, and append-only [2].

Pravega stores ingested data from many different sources in a row-oriented manner which allows for all data points relating to one object to be stored in the same data block. This is beneficial for queries needing to read and manipulate an entire object, but it is slow to analyze large amounts of data. This is an issue because when we want to process events via big data analytics queries, efficiency is poor due to the row-oriented structure of Pravega. A column-oriented processing engine in which columns store similar data points for distinct objects within a block would allow for a quicker analysis of data points, as well as the compression of columns which is efficient for storing lots of data. Without ingesting Pravega events into a proper big data analytics engine, queries against the events are very slow and not feasible for a system storing as much data as Pravega.

II. Background and Related Work

Businesses are always looking for ways optimize their efficacy and to profit. One way to do that is to analyze data from the data source. In the early days, businesses were having difficulties as they navigated data due to the intense on-the-fly processing needed which resulted in the rise of OLAP databases. OLAPs pre-possess the data obtained from the source and store them. The processed data is instantly available for analysis.

One popular distributed event store and stream-processing platform is Apache Kafka. Kafka is a message-oriented middleware. While Kafka is great for transaction and event streams it lacks many feature that are necessary for modern data-intensive applications. Dell's Pravega further enhances programming models like Kafka and provides a cloud-native streaming infrastructure that enables a wider scope of applications by providing additional features like long-term retention, durability, auto-scale, and ingestion of large data to name a few [3].

However, Pravega is not an analytic engine hence it cannot process the data it ingests. The primary objective is to create plug-ins for either Apache Druid or Apache Pinot to enable integration to perform analysis of events from Pravega stream. This will allow users to make big data analytic queries on data that is passing through their stream.

In order to successfully provide a solution to the problem, the team will need some background knowledge of the problem space. A comprehensive understanding of Pravega and Apache Druid or Pinot is essential to solving the problem. In addition, the team will need to have a fundamental knowledge of data organization such as row-oriented and column-oriented databases, as well as experience developing in Java and familiarity with the SQL language.

III. Project Overview

Compared to similar streaming storage systems like Apache Kafka and Apache Pulsar, Pravega provides a more extended data retention period, auto-scaling of partition, and more [3]. However, since Pravega is a storage engine and not an analytics engine, streams don't get analyzed inside Pravega. Online analytical processing (OLAP) database is a software that enables users to quickly, consistently, and interactively observe information from all aspects to gain a deep understanding of data [4]. Hence, integrating Pravega with the OLAP database will allow users to perform log-based analysis against the stream data they stored in Pravega and therefore strengthen Pravega's ecological system, establishing a bridge between the storage engine to the analytics engine.

Integration of a storage engine and analytics engine for big data streaming is powerful and so beneficial as the needs for the Industrial Internet of Things, Internet of Vehicles, and real-time fraud risk control are developing rapidly. For technology to provide better services and customer experiences, we need applications to respond quickly to customer needs while still learning and adapting to changing behavior patterns. To be able to achieve that, an excellent streaming storage engine like Pravega and a great streaming analytics engine like Apache Druid are needed.

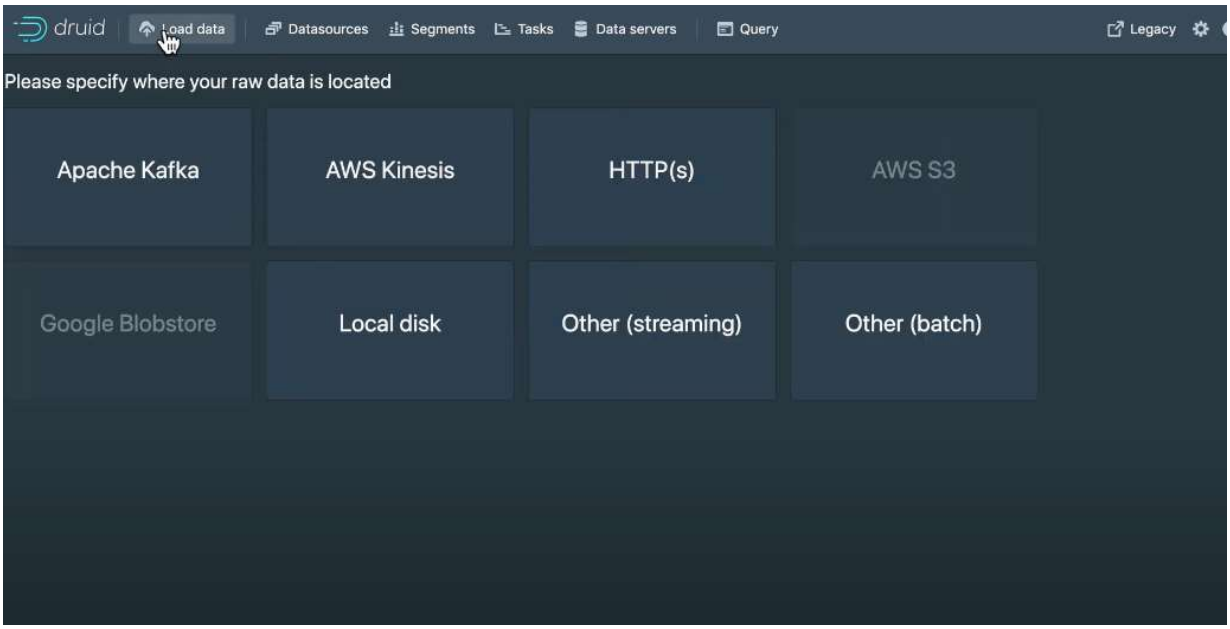
To integrate Pravega with Apache Druid, we will need to develop a plugin for the existing Apache Druid database system. This plugin will serve as a connector that first locates the source stream data in Pravega, then retrieves the stream data and transposes the data from row-oriented format to column-oriented format, and finally ingest them into the Apache Druid database. Since the Apache Druid and its provided client APIs are written in Java. We are expected to develop this plugin primarily using Java. We will also write some SQLs for data transposition and modify some configuration settings for Pravega connection.

We will develop this plugin project in Agile mode that cycles through processes of planning, executing, and evaluating. We are planning to have each sprint last for 3 weeks and each user story or use case to be tracked on GitHub Issue.

We will employ test-driven development (TDD) by having black-box test cases written before actual implementation and then finalizing the test cases after the actual implementation [5]. We might have functional test cases where we will try to validate a small amount of sample data by writing scripts to insert a few rows of sample data into Pravega and test the functionality of the plugin.

We will first develop the plugin in our local environment and version controlled by Git to our course remote repository. Once all test cases and validation are passed, we will then request a pull request to submit our plugin into Apache Druid GitHub repository.

Below is the screenshot of Apache Druid. As you can see on the “Load data” page there are plugins for Apache Kafka and AWS Kinesis stream storage system. We will develop a plugin for Pravega here.



IV. Client and Stakeholder Identification and Preferences

The primary client and stakeholder for this project is Dell Technologies, as they lead the development of the open-source distributed storage system Pravega. Stakeholders also include the JNB team comprised of WSU students that are working to develop this project, as well as the Pravega team at Dell and our Dell software engineer mentors.

The primary need of the Pravega team is for a plugin to be created that allows for an OLAP database to be integrated with Pravega such that the automatic ingestion of Pravega data streams can be enabled to be sent to an OLAP database. A stretch goal that has been highlighted is to produce plugins for both Apache Druid and Pinot (OLAP databases), but the primary goal is to produce just one plugin.

Other stake holders include companies that utilize Pravega to ingest data. Two companies that are public about their adoption of Pravega are Link Labs and Wheels up. Companies such as these will benefit from the new Pravega plugin that will allow for much more efficient big data analytic queries through the integration of an OLAP database.

Requirements and Specifications section

I. Introduction

Dell Technologies takes charge of an open-source project known as Pravega. This is an infrastructure that serves as a storage system that implements data streams to store/serve data [1]. These data streams are made up of sections which contain events. These are sets of bytes in a stream that represent some sort of data – Pravega is effective at storing/ingesting these due to its data streams being consistent, durable, elastic, and append-only [2].

Pravega stores ingested data from many different sources in a row-oriented manner which allows for all data points relating to one object to be stored in the same data block. This is beneficial for queries needing to read and manipulate an entire object, but it is slow to analyze large amounts of data. This is an issue because when we want to process events via big data analytics queries, efficiency is poor due to the row-oriented structure of Pravega. A column-oriented processing engine in which columns store similar data points for distinct objects within a block would allow for a quicker analysis of data points, as well as for the compression of columns which is space efficient. Without ingesting Pravega events into a proper big data analytics engine, queries against the events are very slow and not feasible for a system storing as much data as Pravega.

II. System Requirements Specification

The overall goal for the integration of Pravega and an OLAP database is so that the automatic ingestion of data from Pravega can be enabled to be sent to a Apache Druid (OLAP). With this, we would want a user to be able to perform log-based analytics against the “events” in the data streams. The integration of the OLAP database with Pravega as the processing engine will allow for very efficient big data analytics queries due to its column-oriented structure. Apache Pino is another OLAP database that is available for integration with Pravega. Currently this is a stretch goal, and the team will be focusing on integrating Druid with Pravega first.

This integration has several use cases in which users would use the plugin in a variety of different scenarios in order to make big data analytics queries against the events stored in Pravega’s data streams. We assume that the client in these use cases has already integrated their own system with Pravega so that Pravega can automatically ingest information from their system (this data is the one being queried). This plugin also has several features (functional and non-functional) that will be highlighted below.

II.1. Use Cases

Story: A large retail corporation WalmartABC operates a chain of hypermarkets all around the world. The IT department of WalmartABC built an app that keeps track of product sales in different regions. There are tons of sales data produced every second, WalmartABC stores those data in Pravega with real-time insertion. WalmartABC wants to know what kinds of products are most popular in different parameters to prevent product shortages. To learn the usage patterns, they need to do some big data analytics and machine learning. However, since queries are slow in Pravega, they will ingest part of the diagnostic data into Apache Druid or Apache Pinot and perform the big data analytics there.

Source: Senior Software Engineer from DELL provides the idea of the story.

Story: A large electrical power system protection corporation SchweitzerABC Engineering Laboratories operates power protection services all around the United States. The software IT department is now building a cloud visualization application “SynchrowaveABC” to visualize the real-time state of their protection equipment at different stations, improve understanding of system events and expedite root cause analysis with high-resolution time-series data. SynchrowaveABC brings synchrophasor data and relays event reports together into one place so engineers can analyze both the high-level system impact of an event and the detailed oscillography data.

They set up a connection for real-time data to be stored in Pravega. They planned to support different filter interfaces to visualize the state of equipment for the past 30 days. They ingest the real-time data of the past 30 days into Apache Druid or Apache Pinot. The app SynchrowaveABC will then be able to retrieve the data from Apache Druid or Apache Pinot for different filtering and display it with signal charts, tables, reports, etc.
Source: Senior Software Engineer from DELL provides the idea of the story.

Story: A large E-commerce corporation, Bmazon provides online shopping services all around the world. Each month, there are more than 197 million people visiting the website and purchasing items.

To better serve customers, they need to have a good recommendation system that learns each user’s usage pattern and displays items highly matched to each individual user on the front page. They have the data of each individual browse and purchase history stored in Pravega with a real-time connection.

Now they have designed a few Machine Learning algorithms to train those data in different parameters. They must ingest part of the data from Pravega to Apache Druid or Apache Pinot. Since their search and filter capabilities enable rapid, easy drill-downs of users along any set of attributes.
Source: Principal Software Engineer from DELL provides the idea of the story.

II.2. Functional Requirements

III.2.1 Automatic ingestion data stream into Distributed OLAP

Automatic ingestion: Our plug-in should enable automatic ingestion of data stream into an OLAP database. This feature may vary based on which distributed OLAP is ingesting the stream. In our case, we are integrating Pravega with Druid or Pinot. Druid and Pinot have two streaming ingestion methods (Kafka and Kinesis). Both Kafka and Kinesis provide Exactly-once guarantees a feature discussed in the transaction feature.

Source: This requirement was requested by Dell in the original abstract of the project and is the core of the project. This feature is useful for anyone doing log-based data analytic against the event in their stream.

Priority: Priority Level 0: Essential and required functionality.

III.2.2. Checkpoints for the plugin

Checkpoints: The plug-in must support checkpoints that keep track of logs during a transaction. This checkpoint is used to verify and validate modified data in real-time. The checkpoints will also be useful to create backup and recovery prior to any application of data

modification in the database. This will allow us to resume from clean or unclean shutdown using the recovery system to return to the checkpoint state.

Source: This requirement was requested by a senior principal engineer at Dell and Pravega.io. The requirement is necessary for clients that may encounter a failure during a transaction allowing them to easily default back to the checkpoint state.

Priority: Priority Level 0: Essential and required functionality.

III.2.3. Transactions

Transactions: The plug-in must support transaction such that when a client says commit, the whole data is appended to the log. Transactions are all or nothing / exactly once semantic. This means an event is delivered and processed exactly once.

Source: This requirement was requested by a senior principal engineer at Dell and Pravega.io. This feature is useful for anyone doing log-based data analytic against the event in their stream.

Priority: Priority Level 0: Essential and required functionality.

III.2.4. Schema Registry

Schema Registry: Given the data stored in the Pravega stream is unstructured to minimize payload size Pravega has a stand-alone schema registry feature that contains the data type definition for events within the stream. The plugin must support this feature.

Source: This requirement was requested by a senior principal engineer at Dell and Pravega.io. This feature is necessary for anyone running big data query on Pravega. Pravega receives raw bytes and delivers raw bytes. This feature is intended to help clients who want deserialize the raw bytes which requires knowing the data type definition.

Priority: Priority Level 0: Essential and required functionality.

III.2.5 Operation on data

Search: This feature will take an input of data request and returns the result from the database.

Source: Chief data and analytics Engineer at Dell and Pravega.io. This function will help clients do the easiest data query task which is to search.

Priority: Priority Level 0: Essential and required functionality.

III.2.6 Proactive Creation of Tables within Apache Druid

Table Creation: Pravega data streams store events of certain data types – streams will vary in the data in which they contain. With this, we want to store data streams to certain tables within the OLAP database. When we encounter a data stream of x data type, we should not store it within an existing table for data streams of y data. The database should expect to see streams of different types and create new tables to store them accordingly.

Source: This requirement was suggested by a Dell engineer. This would be good to have to make the OLAP database efficient, it is important, but it is not the highest priority.

Priority: Priority Level 1: Important feature, not highest priority

II.3. Non- Functional Requirements

Easy to Use: The plugin should be easy to use. Its user interface should be user friendly and self-explanatory. Users should be able to easily make queries against certain Pravega data streams for processing within the OLAP database.

Reliability: The plugin is expected to ingest all data selected without any loss. We aim to maintain system integrity by ensuring that no data is lost during the transfer of data streams from Pravega to the OLAP database.

Maintainable: The plugin will be available as open source on Apache Druid GitHub Repository, it should be maintainable to the public.

Cross Platform: The plugin should be working on different operating systems.

Code Comment: Every method in the Plugin should be properly commented.

Usage Manual: There should be a usage manual provided along with the Plugin in production to allow users to clone the repository and make working queries against Pravega streams.

Extensible: The plugin should be extensible so future developers can easily add additional features to it without changing the existing structure of the code.

Performance: The plugin should be efficient in ingesting data from Pravega to Apache Druid.

Storage: Apache Druid should be able to store ingested data from Pravega with immediate concerns of running out of storage.

Security: Plugin must be secure to ensure that data is not tampered with in transit from Pravega to Apache Druid.

Scalability: The system should be able to be scaled to allow for different systems to be integrated with Pravega.

III. System Evolution

The delivery of the plugin should be functional, bug free, and versatile in the sense that it is able to adapt to unforeseen changes in requirements. Assumptions must be made for potential changes that we may anticipate during development. With this list of possible changes/risks, the team can quickly adapt to the change and refer to this section in order to decipher what the next steps are to handle the scope/requirement change.

Deployment: Currently, the Dell team is aiming for us to deploy our plugin via Kubernetes. We are assuming the efficiency of Kubernetes will be sufficient for our project to be deployed and perform as expected. If change occurs, whether that be a software change of Kubernetes, or a change in the client's requirements, we propose deploying instead using AWS.

Development (Java): This project will be completed using Java8 development. This is the most recent version of Java, and we are assuming that Pravega and the OLAP database are compatible with it. We should anticipate potential compatibility issues with Java8 and Pravega or Apache Druid or Pino. Should this occur, we aim to instead develop with an older version of Java7 in hopes of eliminating compatibility issues.

OLAP Database: This project allows for our team to select either Apache Druid or Pino as our processing engine for the Pravega streams. As of now we are undecided on which we will be integrating with Pravega, but we are leaning towards Druid. We should anticipate the possibility of Druid having less functionalities or being less efficient than Apache Pino. If we integrate Druid and find our big data analytics queries performing poorly (assuming everything was integrated correctly), we should be able to adapt quickly and integrate Apache Pino with Pravega instead.

Testing: Once the integration is complete, we will need to test the processing engine by creating sample queries that we can make to fetch and process information within the Pravega streams. If sample queries prove to be inefficient for testing this plugin, or if we are unable to fetch data from Pravega in a testing environment, then we will use Debezium, another data source/platform so that we can test our sample queries on Debezium data sources.

Integration of both Apache Druid and Pino: Primary goal of this project is to just integrate one OLAP database with Pravega. We should anticipate the possibility of finishing the Druid integration early and also integrating/creating a plugin for Apache Pino to process data from Pravega.

Solution Approach

I. Introduction

The Purpose of this document/section is to provide a general description of the functionality and design of the Pravega OLAP Java project. In the Architecture design section, a bird's-eye view of the architectural design pattern is given to highlight the different components involved in this project. The intended audience for this document is our Cpts 421 capstone professor as well as the Dell engineers mentoring us. This document will provide both audiences with the solution approach designed by the team with the help of the Dell engineers.

Pravega stores ingested data from different sources in a row-oriented manner which allows for all data points relating to one object to be stored in the same data block. This is beneficial for queries needing to read/manipulate an entire object, but it is slow to analyze large amounts of data. This is an issue because when we want to process events via big data analytics queries, efficiency is poor due to the row-oriented structure of Pravega. A column-oriented processing engine in which columns store similar data points for distinct objects within a block would allow for a quicker analysis of data points, as well as for the compression of columns which is space efficient. Without ingesting Pravega events into a proper big data analytics engine, queries against the events are very slow and not feasible for a system storing as much data as Pravega.

II. System Overview

This project aims to provide a plugin that allows Pravega's ingested data streams to be sent to/stored within Apache Druid, an OLAP database. The general process/flow of the project is as follows:

First Apache Druid is instantiated which causes the plugin to be loaded. Once loaded, a user can opt to use the feature via the Druid UI where they can specify certain conversions or parameters for the data streams stored in Pravega. These specifications will be read and understood within the plugin. Once this is done, a request containing user specifications will be sent to Pravega and data is retrieved. The plugin will require a reading component to retrieve the Pravega data streams, another that will take this data and transform it into an understandable manner for Druid. One for writing the newly transformed data into Druid's column-oriented storage, and a component for storing metadata associated with the user's request for Pravega data streams to be stored within the OLAP database.

III. Architecture Design

III.1. Overview

An ingestion plugin that ingests stream data from Pravega to Apache Druid requires a series of stream conversions. These conversions contain a necessary part and an optional part. The necessary part of conversion is to convert the Pravega stream events into "segments" that are compatible with Apache Druid's storage. The optional part of conversion includes filtering and transforming the data structure.

Ideally, there exist graphical interfaces provided to users that allow users to pick certain conversions and to set the parameters of conversions in detail. A config loader will then retrieve the information from graphical interfaces and save it into a JSON file for specification of conversion that user selected.

With the specification of conversion ready, the plugin system is now able to perform a series of conversions subscribed by the user. To do that, we need to have a scheduler that organizes a series of conversions step by step.

We defined a controller sub-system to be a central processing unit of the plugin system. The controller will then instantiate all the sub-systems with appropriate parameters specified from the specification JSON. Then the controller will establish a pipeline execution for those sub-systems.

The execution of the Plugin's sub-systems is as follows: connection probe with Pravega, stream read from Pravega, stream parse from stream event, schema customization, tune setting, and finally a segment loader that loads the segments into the deep storage of Apache Druid.

In each step of sub-task pipeline execution. We have 2 databases: one is for meta data and the other one for actual data that records the data at any given stage.

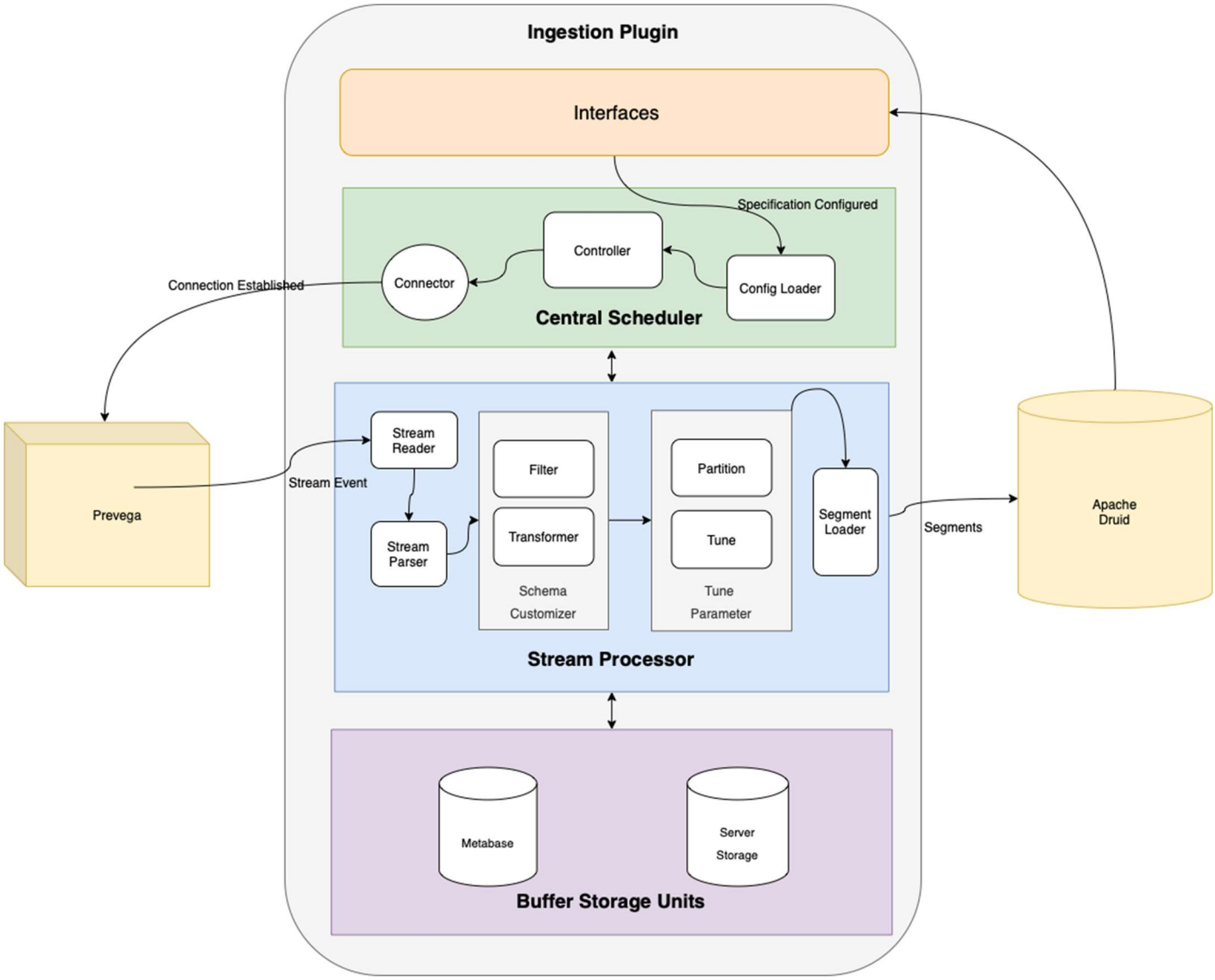


Figure III.1

III.2. System Decomposition

I.1.1. [Interfaces]

a. **Description**

The interface is the component that provides the user with an ability to provide inputs as parameters.

b. Concepts and Algorithms Generated

No algorithms involved.

Straightforward parameter inputs.

c. Interface Description

Ideally, a GUI dashboard provided in the Apache Druid load data page. The dashboard provides different ingestion methods to users.

Service Provided:

1. **Service name:** Specification Read-In

Service provided to: Config Loader
Description: Receive specification of conversion from user.
Services Required: N/A

I.1.2. [Config Loader]

a. **Description**

Config Loader receives a specification from the Apache Druid user. The config loader will then express the specification in a JSON file. The specification contains types of ingestion (stream or batch), IO config information for the Pravega source data address, and the schema transformation, filter, and tuning methods.

b. Concepts and Algorithms Generated

A very simple algorithm that reads parameter inputs from front-end and writes those inputs with certain structural format into a specification JSON file. The JSON file is supposed to be saved in a buffer storage unit – metabase.

c. Interface Description

Internal Logic

Service Provided:

2. **Service name:** Specification JSON Create

Service provided to: Controller
Description: The config loader saved the specification in JSON. The controller reads the specification and controls other subsystems to perform data ingestion.
Services Required: Interfaces

I.1.3. [Controller]

a. **Description**

Controller manages the entire ingestion procedures. Controller reads the specification generated by Config Loader to understand all ingestion subtasks subscribed by the user. The controller then manages the entire stream of event processing procedures according to the specification, including the connection of the Pravega data source.

b. Concepts and Algorithms Generated

Controller reads in the specification. Then, according to the specification a factory design pattern applied to auto-generate subsystem instances. A scheduling algorithm pipelining subsystem instances execution.

Details about the scheduling algorithm:

A predefined TreeMap<Integer, Object> where Integer defines a priority level, 0-highest and greater number the lower priority.
Since all associated sub-system instances are instantiated in the first place when our controller reads in the specification. We have them handy, perhaps in a List<Object>.
Next step we are going to traverse over the predefined TreeMap. By nature of TreeMap in Java, it is an OrderedMap based on the key, hence we are always traversing from high priority to low priority. In each traversal, we will compare the tree map value with List<Object> using the **instanceof** keyword and insert the match type instance into the tail of a LinkedList<Object>.
Now a LinkedList<Object> defines the series of execution of different sub-systems in series. We are then able to run the execution from the head of LinkedList to the tail of the LinkedList.

c. Interface Description

Internal Logic Controller, no interface.

Service Provided:

3. **Service name:** Ingestion Process Controller

Service provided to: Connection, Stream Reader, Stream Paser, Schema Customizer, and Tune Parameter.
Description: The controller manages the entire ingestion procedures, call its sub-system run in a pipelining schedule algorithm.
Services Required: Specification JSON from Config Loader.

I.1.4. [Connector]

a. **Description**

The connector is responsible for ensuring that the provided Pravega source data address is reachable.

b. Concepts and Algorithms Generated

The connector tries to connect with Pravega with the given address provided by Controller using HTTP protocols. If the connection is successful, the connector tells the controller that it is a valid connection.

c. Interface Description

Internal Logic.

Service Provided:

4. **Service name:** Connection Probing

Service provided to: Controller, Stream Reader
Description: In order for controller to proceed with the pipelining execution, a connection must be valid.
Services Required: Controller

I.1.5. [Stream Reader]

a. **Description**

The stream reader is responsible for reading the stream event from Pravega.

b. Concepts and Algorithms Generated

No algorithm involved.
Simply a wrapper class that uses Pravega client APIs to read and access the data stream, hand over the data stream to stream parser.
Note that ideally, the data stream read from Pravega is to be stored in a buffer storage unit with actual data and meta data.

c. Interface Description

Internal Logic.

Service Provided:

5. **Service name:** Source Data Read-In

Service provided to: Controller, Stream Parser
Description: In order for the controller to proceed with the pipelining execution, the stream reader must work functionally.
Services Required: Connector, Controller.

I.1.6. [Stream Parser]

a. **Description**

The stream parser is responsible for parsing the stream event that is obtained from Stream Reader into an Apache druid compatible format called “Segments”.

b. Concepts and Algorithms Generated

No algorithm involved.
Simply a wrapper class that uses Apache Druid client APIs to parse the stream events.
Note that ideally, the parsed segments are to be stored in a buffer storage unit with actual data and meta data.

c. Interface Description

Internal Logic.

Service Provided:

6. **Service name:** Parse Stream Event

Service provided to: Controller, Schema Customizer, Tune Parameters
Description: In order for the controller to proceed with the pipelining execution, the stream parser must work functionally.
Services Required: Connector, Controller, Stream Reader

I.1.7. [Schema Customizer]

a. **Description**

The Schema customizer contains two parts – filter and transformer. Provides users with the ability to customize the structural segments before actually loading them into the deep storage of Apache Druid.

b. Concepts and Algorithms Generated

No algorithm involved.
Simply a wrapper class that uses Apache Druid client APIs to handle the schema customization.
Note that ideally, the customized segments are to be stored in a buffer storage unit with actual data and meta data.

c. Interface Description

Internal Logic.

Service Provided:

7. **Service name:** Segment Customization

Service provided to: Controller, Tune Parameter
Description: In order for the controller to proceed with the pipelining execution, the Schema Customizer must work functionally.
Services Required: Connector, Controller, Stream Reader, Stream Parser.

I.1.8. [Tune Parameter]

a. **Description**

The Tune Parameter contains two parts – partition and tune. Parameters for partition tell Apache Druid how to partition the data into segments by time. Parameters for tune tell Apache Druid a more flexible way to reprocess the segments.

b. Concepts and Algorithms Generated

No algorithm involved.
Simply a wrapper class that uses Apache Druid client APIs to handle the tune setting.
Note that ideally, the tuned segments are to be stored in a buffer storage unit with actual data and meta data.

c. Interface Description

Internal Logic.

Service Provided:

8. **Service name:** Tune Segments

Service provided to: Controller
Description: In order for the controller to proceed with the pipelining execution, the Tune Parameter must work functionally.
Services Required: Connector, Controller, Stream Reader, Stream Parser, Scheme Customizer (Optional, can leave as blank).

I.1.9. [Segment Loader]

a. **Description**

The segment loader is responsible for loading the segment into Apache Druid deep storage.

b. Concepts and Algorithms Generated

No algorithm involved.
Simply a wrapper class that uses Apache Druid client APIs to load the segments into its deep storage.

c. Interface Description

Internal Logic.

Service Provided:

9. **Service name:** Load Segment

Service provided to: Controller
Description: In order for the controller to proceed with the pipelining execution, the Segment Loader must work functionally.
Services Required: Connector, Controller, Stream Reader, Stream Parser, Scheme Customizer (Optional, can leave as blank), Tune parameter.

IV. Data design

Major Structures:

Pravega

Pravega will be serving as the source for all data streams for this plugin. Without Pravega, this plugin serves no purpose. Pravega will be providing our system with its ingested data streams at the user's request so that it can be transformed to the user's specifications and stored within Apache Druid. The plugin will utilize existing Pravega APIs to make calls to retrieve stored data streams so that they can be read by our read component within the plugin.

Apache Druid

Druid serves as the primary storage for this project. This is the target OLAP database in which we want to enable Pravega data streams to be read, transformed and stored into Druid via our plugin. Pravega's data streams are distinct and so each stream will contain varying data. This means that Druid cannot store all streams within a singular table. The plugin must be able to read the data stream and signal to Druid whether or not the data stream is compatible to be stored within existing tables. If not, Druid should be able to automatically create a new table and store the new data stream there.

Currently it is planned to make Druid tables manually for different Pravega streams for testing. Enabling the automatic creation of tables within Druid is a feature of lesser priority compared to the read, transform, and write features highlighted in the architecture section.

Buffer Storage Unit

The buffer storage will be a new data structure created for this project. It serves to gather/contain metadata within the Metabase and also provide server storage. The Metabase will have more emphasis as it will serve to gather metadata/telemetry related to our plugin. Things such as how much data we read/wrote. More specifically, the following subsystems highlighted in the architecture section will make use of the Metabase:

- Config loader – Responsible for taking in the user's data query and specifications (within a JSON file). This component will store the JSON file containing the query and its associated information within the Metabase.
- Stream Reader – The data stream read from Pravega by this component should ideally be stored in a buffer storage – Metabase. By storing the data streams read by this subsystem, we can keep track of which data streams users are making queries on using our plugin.
- Stream Parser – Responsible for parsing the stream obtained from the reader component in a compatible format for Druid (Segments). These segments should be stored in the buffer storage – Metabase. By storing parsed streams, we can keep records of the "before and after" of a data stream read from Pravega. If errors occur when writing to Druid, we can analyze the parsed streams stored in the Metabase to troubleshoot.
- Schema Customizer – Contains filter and transformer. Responsible for giving users the ability to customize the transformed segments before actually storing them in druid. These customized segments should be stored within the buffer storage's Metabase. Again, this can be useful for troubleshooting errors when writing to Druid. If a user transforms the parsed data in a certain manner that causes errors, we can simply analyze the Metabase to troubleshoot.
- Tune Parameter – contains partition and tune, tells druid how to partition the data into segments for storage. Tune tells druid how to reprocess the segments, the tuned segments should be stored within buffer storage unit with actual data and metadata

The other component within the Buffer Storage Unit is the Server Storage. This is a data store recommended by the Dell engineers. This plugin needs to be deployed in some server independent of Apache Druid or Pravega. This server storage will serve as a database to store the actual data after each step of the plugin process.

Minor Structures:

Data streams – Retrieved from Pravega via utilization of public Pravega API. Returned streams will be read, parsed, and transformed via the subsystems outlined in the architecture section. We propose storing raw data streams from Pravega in the Buffer Storage Unit for metadata/telemetry.

Transformed streams – Data streams read via the Stream Reader component must be parsed into segments and then potentially transformed/customized to a user's specifications before stored within Druid via our write component.

User Specification via JSON file – When a user initiates our plugin to be used, they input parameters in order to make a data query against ingested Pravega streams. User's input will be expressed via a JSON file containing information such as type of ingestion, IO config information, etc. (see I.1.2 [Config Loader]). We propose storing this JSON containing the user's input for our plugin within the Metabase so that we can track the kinds of queries users are making against Pravega with our plugin.

V. User Interface Design

The plugin will provide users with a command-line interface dashboard on Apache Druid load data page. The interface enables users to pass streams directly from Pravega as a parameter. The streams will then be automatically ingested by Druid. The interface will allow the user to pass specifications to the config loader. The interface will appear along with other data ingestion plugins like Apache Kafka and Amazon Kinesis shown in figure V.1.

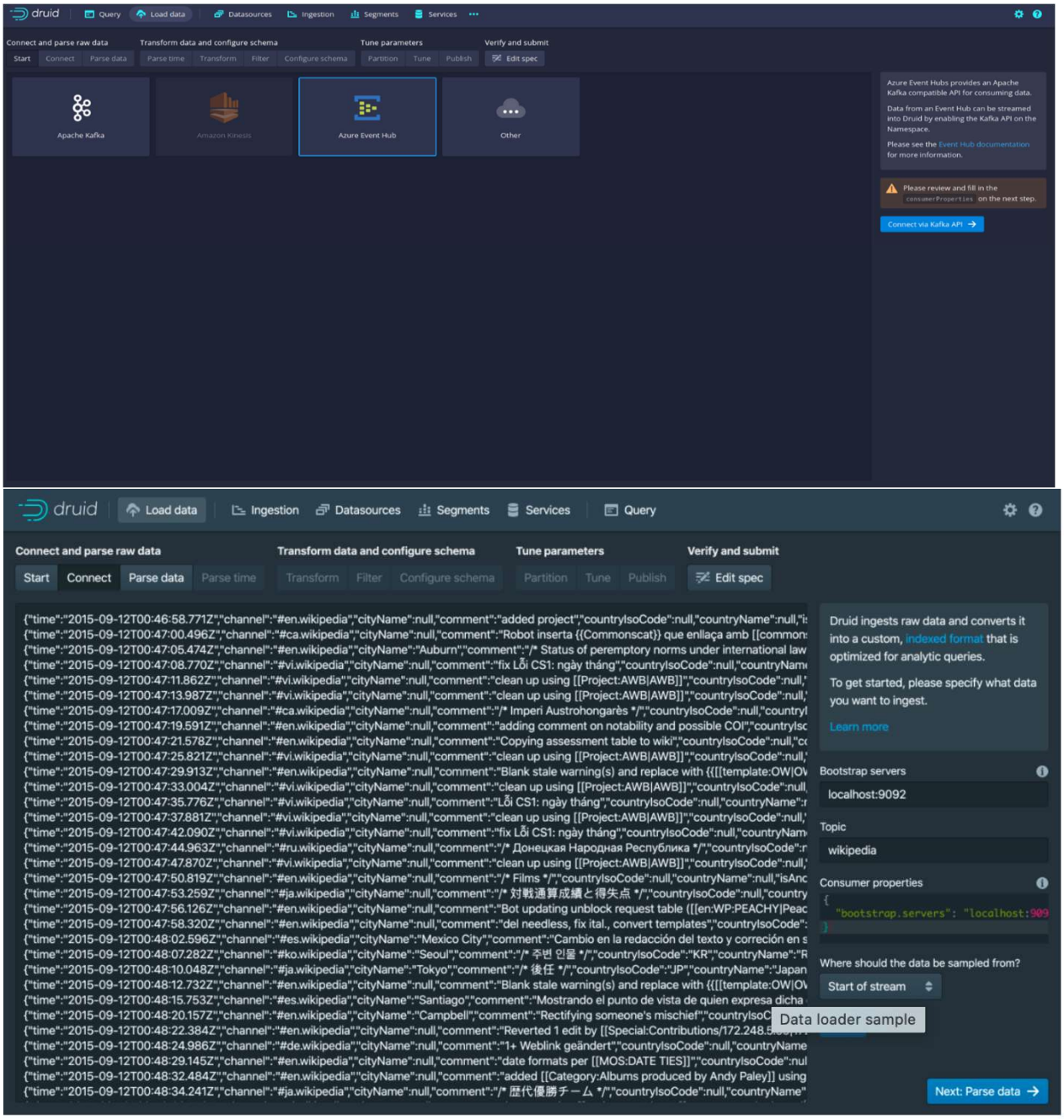

druid
Query
Load data
Datasources
Ingestion
Segments
Services
Connect and parse raw data
Start
Connect
Parse data
Transform data and configure schema
Parse time
Transform
Filter
Configure schema
Tune parameters
Partition
Tune
Publish
Verify and submit
Edit spec
Apache Kafka
Amazon Kinesis
Azure Event Hub
Other
Azure Event Hubs provides an Apache Kafka compatible API for consuming data.
Data from an Event Hub can be streamed into Druid by enabling the Kafka API on the Namespace.
Please see the Event Hub documentation for more information.
Please review and fill in the consumerProperties on the next step.
Connect via Kafka API
druid
Load data
Ingestion
Datasources
Segments
Services
Query
Connect and parse raw data
Start
Connect
Parse data
Parse time
Transform data and configure schema
Transform
Filter
Configure schema
Tune parameters
Partition
Tune
Publish
Verify and submit
Edit spec
Druid ingests raw data and converts it into a custom, indexed format that is optimized for analytic queries.
To get started, please specify what data you want to ingest.
Learn more
Bootstrap servers
localhost:9092
Topic
wikipedia
Consumer properties
{
"bootstrap.servers": "localhost:909
}
Where should the data be sampled from?
Start of stream
Data loader sample
Next: Parse data

Figure V.1

Project Testing and Accepting Plan

I. Introduction

I.1. Project Overview

The software to be tested is a Java plugin used to enable data streams from Pravega to be read, processed, and written into Apache Druid as data segments. The major functionalities that need to be tested are the central scheduler, and the stream processor components that are shown in the Figure III.1, architecture diagram. The Buffer Storage Units may also require testing in order to ensure the integrity of the data being moved from the stream processor to the Buffer Storage.

The acceptance criteria for this software to be considered complete is as follows: all test cases passing with expected output. Test cases for all subsystems should be passing such as reading from Pravega, processing the data streams into segments, and writing to Apache Druid. Another criterion is client satisfaction of the plugin.

I.2. Testing Objective and Schedule

The team aims to develop a Java plugin that needs to be integrated with two different technologies for Pravega streams to be processed by Apache Druid. This plugin has multiple subsystems planned to allow this integration, which means there are multiple possible points of failure at runtime. With this, we plan to implement test cases with the Java testing framework known as JUnit in order to ensure the integrity of the software when development is completed. We plan to use JUnit as it is a framework specifically designed for Java development and because it emphasizes test driven development. [6]. JDK is a prerequisite for JUnit to be installed since it is a Java framework.

With JUnit, we will be able to write unit and integration tests to cover every subsystem within the proposed plugin. Along with JUnit, GitLab's CI/CD pipeline will be used. This is a tool that "automates your software delivery process." It can be utilized to build the code for the plugin, run tests (CI), and deploy different versions of the application (CD). [7]. With this tool, the team can ensure that each change in code is validated by causing the project to be automatically built and tested with this tool. Another tool to be used for testing is the mocking framework designed for Java, Mockito. With this tool, we can mock objects/subsystems for testing interactions within components.

The major work activities are to provide unit tests for all subsystems within the Central Scheduler and Stream Processor components, as well as to provide integration tests for the components to ensure that they work/interact properly together. These tests will be contained in a GitLab repo that contains the pipeline as well as documentation explaining the tests. Tests will need to be thought about before writing the code as per test driven development. Developers will write code, then test, code, test etc. Milestones include producing a test as well as code for a certain subsystem, as well as integrating those tests with the pipeline and demoing the pipeline to ensure that the code can pass those tests.

I.3. Scope

The purpose of this document is to discuss the high-level testing plan for our plugin for Apache Druid to take in data segments from Pravega's data streams. This document will discuss the

overall testing strategy and will discuss the unit, integration, and system tests that will be added to our software to validate the plugin's functionality.

II. Testing Strategy

For the testing strategy the team plans to conduct testing for all client requirements. The team will be utilizing a test-driven development strategy to build each subsystem that the plugin requires. Afterwards we will make use of a bottom-up incremental integration test to ensure coherent integration. Once the plugin is fully integrated, we will evaluate the system compliance with respect to the requirement by doing system testing. Finally, we conduct an acceptance test based on tests that adheres to the requirements.

Each step of the testing will go through the following test-driven development cycle.

Plugin subcomponent test example:

a) **Test case for subcomponent:** The developer will come up with a test for a single unit that describes an aspect of the plugin.
b) **Run the test:** The developer will run the test. This test should fail because the feature has not been implemented yet.
c) **Implement minimal functionality** The developer will implement the function in a basic form such that it passes the test. Should the test fail due to a potential bug fix the bug and repeat the test.
d) **Refractor:** Repeat testing each incremental cycle. Should the function fail the test run, the developer will have to refactor the implementation and run the test again.
e) **Test refactored implementation:** Repeat testing each incremental cycle. Should the function fail test run, developer will have to refactor the implementation and run test again.
f) **Use CI/CD pipeline:** Here the developer will push the implementation of a unit to CI (Continuous Integration) and check if there is a conflict between the newly added functionality and existing implementation. Should a conflict arise, the developer will refactor the implementation and repeat the process d to f again until the implementation passes the pipeline.
g) **Document test data:** Test data, test cases, test configuration and bugs should be documented each cycle.
h) **Make a merge request:** The developer will make a merge request to the master from their development branch.
i) **Review merge request:** Each team member should review the merge request and approve the merge. If the merge request is not approved the team must discuss how to resolve the issue and refactor the implementation. Afterward the developer will repeat the process d to i.
j) **Use CD to deploy:** Once a merge is approved, CD will be used to deploy the new feature if there are no pipeline errors.

The above TDD cycle will be applied for integration testing, system testing, and acceptance testing as well.

III. Test Plan

III.1. Unit Testing

The primary goal of unit testing is to take the smallest unit of testable software in the application, isolate it from the remainder of the code and test it for bugs and unexpected behavior.

The team will follow test driven development in order to validate the functionality of all subsystems. We will take the requirements for each subsystem in order to create test cases before the subsystem is fully developed [9]. Our subsystems will have a minimum of two-unit tests. One failure, and one successful use case for the subsystem. Many of our subsystems are connected, so in our tests we will need to isolate them by providing the subsystem with mock data/input in order for an output to be created independently from other subsystems.

Alongside the JUnit Java testing framework, the mocking tool known as Mockito will also be used for unit testing. It will be utilized in order to mock subsystems that a currently tested subsystem may be dependent on. This is useful for fully isolating our subsystems as well as being able to test subsystems that rely on incomplete subsystems.

III.2. Integration Test

Since the controller component is responsible for initiating and building up an execution pipeline for sub-components in the Stream Processor layer, we will use a bottom-up incremental integration test approach [8]. In bottom-up incremental integration, we will build a temporary controller component called “Driver” to customize the execution orders of sub-components in the Stream Processor layer accordingly with their completion. This way, we are able to verify the accuracy of data flows in between the sub-components in the Stream Processor layer in advance. We will do the integration test for the entire system with a real Controller after we are assured that sub-components of stream processors are indeed functioning properly.

III.3. System Testing

III.3.1. Functional Testing

Our objective for functional testing directly relies on our requirements and specifications. Every requirement in the requirement and specification will be treated as a unit and a test will be devolved around each of those requirements. In addition, we will also be developing tests around each component in our solution approach architecture where each component is treated as a unit. Each functionality is tested multiple times as the developer will be refactoring the code multiple times to meet the required criteria. Refactoring could be done for improvement purposes or due to a test failure. In case of failure, the developer will document the failure and refactor the implementation, and test again.

III.3.2. Performance Testing

For performance testing, we will be utilizing fuzzing to test for possible unbeknown bugs within our system. Our plugin takes in structured input with specifications from users. We can use Fuzzing [10] to test invalid, unexpected, and random data as input into our system. By using fuzzing, we can automate all possible input that may not be taken into consideration during development. This makes it possible for us to identify some exceptions we may not be able to identify during code auditing. Identifying such exceptions will also improve our system as it will prevent a malicious user from flooding the system with inputs that may crush the system.

III.3.3. User Accepting Testing

For user-accepting testing, the team will come up with a set of predefined test case that adheres to the overall requirements. The predefined test cases are defined incrementally throughout the development process. Each case showcases a functionality a user would make use of post-deployment. The set of predefined cases should include all the possible use cases a user would utilize. If our implementation passes all the predefined scenarios tests, the overall system is regarded as acceptable.

IV. Environmental Requirements

We will have to install Java Testing Frameworks. Junit for Unit Test. Mockito for Integration Test. We will configure a dev docker container with Junit and Mockito installed inside. We will also have to import our plugin and install standalone instances of Pravega and Apache Druid inside of the container.

We will use GitHub Action CI pipeline to continuously integrate the testcases when new changes are made to the master branch. DELL have resources of AWS, if possible, we will use AWS instance to run the GitHub Action CI/CD pipeline.

Glossary

Auto scaling: A Pravega concept that allows the number of Stream Segments in a stream to change over time, based on scaling policy.
Big Data – Refers to data sets that are too large or complex to be dealt with by traditional data processing software
Checkpoints - A kind of Event that signals all readers within a Reader Group to persist their state.
Data streams – method of organizing data, has a data source and a destination. Comprised of stream segments which contain events
Events – Set of bytes contained within stream segments to represent some data point
OLAP – Online analytical processing database
Pravega – Open-source storage system implementing streams for storing/serving continuous and unbounded data.
Query – Refers to a select query which retrieves data from a database and an action query that applies operations on the data.
Transaction – A collection of streams write operations that are applied atomically to the stream.

References

[1] “Pravega Concepts” *Concepts - Exploring Pravega*. [Online]. Available: https://cncf.pravega.io/docs/nightly/pravega-concepts/#introduction. [Accessed: 23-Sep-2022].

[2] “Pravega Overview” *Exploring Pravega*. [Online]. Available: https://cncf.pravega.io/docs/v0.11.0/. [Accessed: 23-Sep-2022].

[3] Pravega, “A reliable stream storage system,” *Pravega*, 21-Mar-2022. [Online]. Available: https://cncf.pravega.io/. [Accessed: 23-Sep-2022].

[4] Foundation, A., 2022. Druid | Database for modern analytics applications. [online] Druid.apache.org. Available at: <https://druid.apache.org> [Accessed 23 September 2022].

[5] Hamilton, T., 2022. What is Test Driven Development (TDD)? Tutorial with Example. [online] Guru99. Available at: <https://www.guru99.com/test-driven-development.html> [Accessed 23 September 2022].

[6] “Junit - Overview,” *Tutorials Point*. [Online]. Available: https://www.tutorialspoint.com/junit/junit_overview.htm. [Accessed: 26-Oct-2022].

[7] Written by: Marko Anastasov Marko Anastasov Marko Anastasov is a software engineer, “CI/CD pipeline: A gentle introduction,” Semaphore, 15-Jul-2022. [Online]. Available: https://semaphoreci.com/blog/cicd-pipeline. [Accessed: 26-Oct-2022].

[8] K. Asif, “What is bottom-up testing?,” Educative. [Online]. Available: https://www.educative.io/answers/what-is-bottom-up-testing. [Accessed: 26-Oct-2022].

[9] “Test-driven development,” *Wikipedia*, 07-Sep-2022. [Online]. Available: https://en.wikipedia.org/wiki/Test-driven_development. [Accessed: 27-Oct-2022].

[10] “Fuzzing,” *Wikipedia*, 25-Sep-2022. [Online]. Available: https://en.wikipedia.org/wiki/Fuzzing. [Accessed: 28-Oct-2022].

Appendices

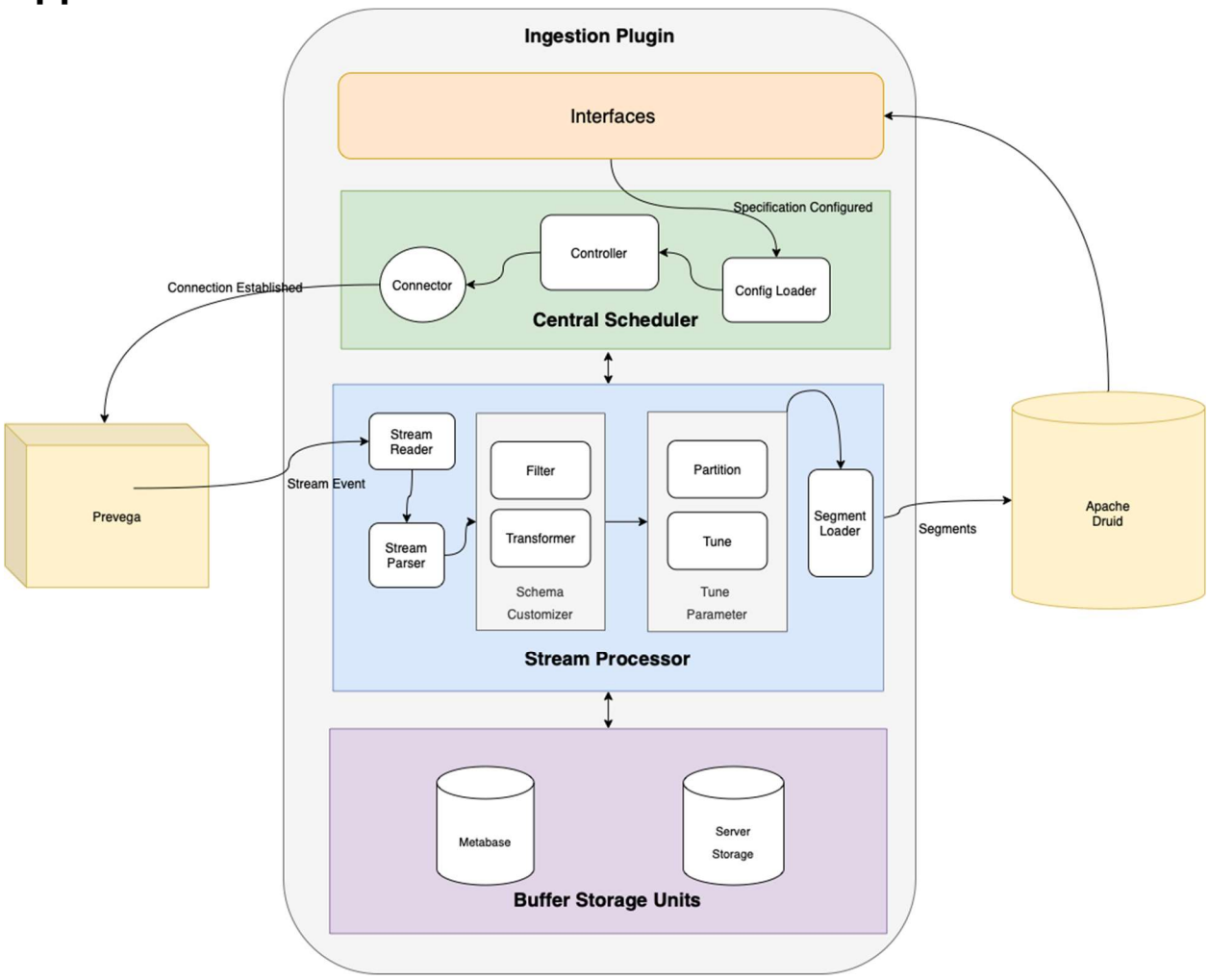
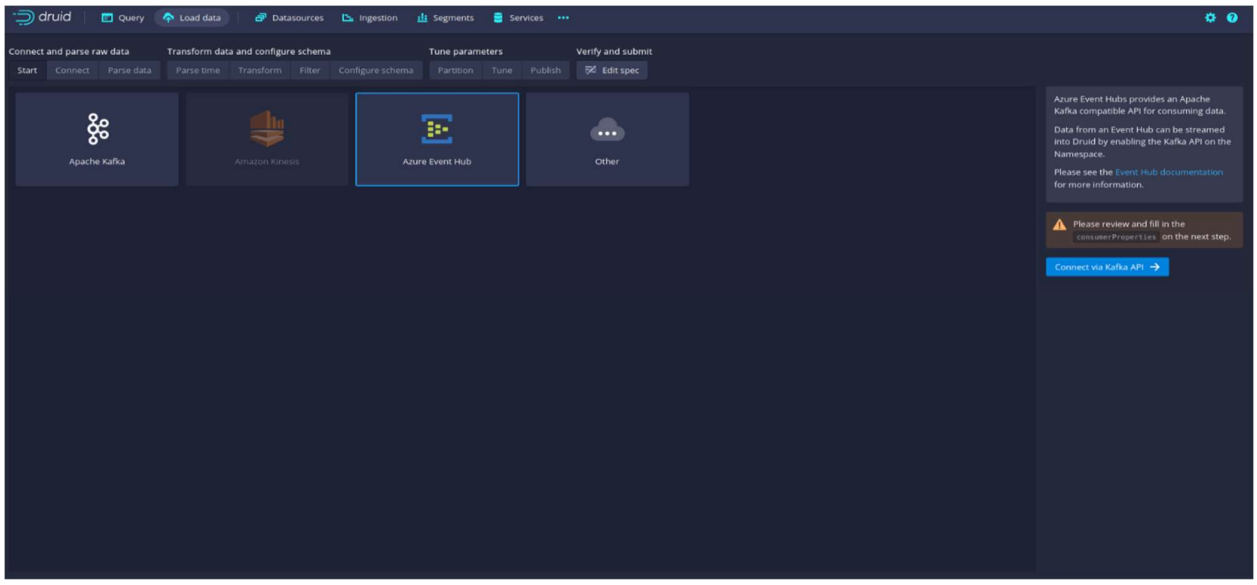


Figure III.1



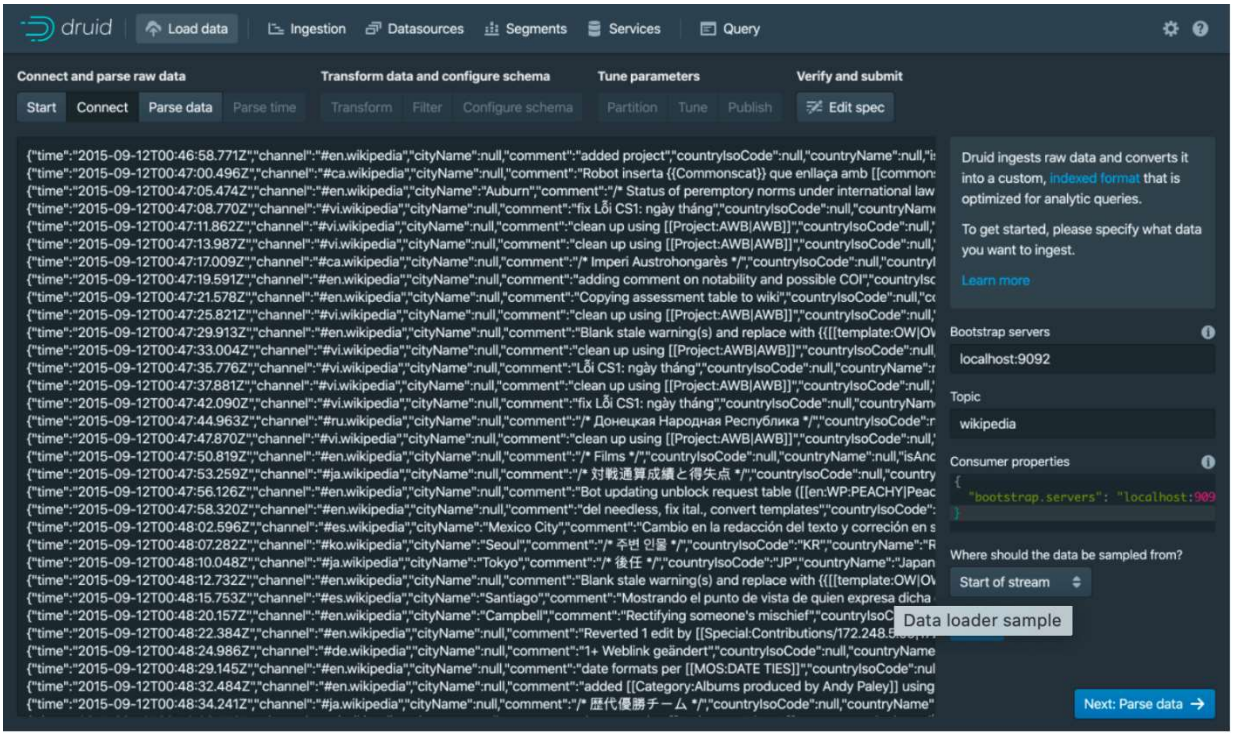

druid
Load data
Ingestion
Datasources
Segments
Services
Query
Connect and parse raw data
Transform data and configure schema
Tune parameters
Verify and submit
Start
Connect
Parse data
Parse time
Transform
Filter
Configure schema
Partition
Tune
Publish
Edit spec
Druid ingests raw data and converts it into a custom, indexed format that is optimized for analytic queries.
To get started, please specify what data you want to ingest.
Learn more
Bootstrap servers
localhost:9092
Topic
wikipedia
Consumer properties
"bootstrap.servers": "localhost:909
Where should the data be sampled from?
Start of stream
Data loader sample
Next: Parse data

Figure V.1