ActiveDataFlow: Flink-Inspired Source Architecture

1. Introduction

This document details a new source processing architecture for the `active_data_flow` engine, implemented under the `ActiveDataFlow::SourceSupport` namespace. This design is directly modeled on the sophisticated, scalable, and fault-tolerant Data Source API of Apache Flink [1]. By adopting Flink's core concepts, we can provide a robust framework for data ingestion that is both powerful and extensible.

The Flink model separates the responsibility of discovering what data to read from the process of actually reading it. This is achieved through three primary components:

1. **Splits**: A `Split` is a portion of the total data source, such as a file, a part of a file, or a message queue partition. It represents the smallest unit of work that can be distributed.
2. **SplitEnumerator**: This is the central coordinator or "brain" of the source. It runs as a single, logical process and is responsible for discovering all the `Splits` and assigning them to available readers.
3. **SourceReader**: These are the "workers" that execute in parallel. Each reader requests `Splits` from the `SplitEnumerator`, reads the data described by the `Split`, and emits the records into the data flow.

This architecture provides excellent support for parallelism, dynamic work re-balancing, and fault tolerance, making it an ideal foundation for `ActiveDataFlow`.

2. Core Architecture and Components

The `ActiveDataFlow::SourceSupport` namespace will contain the base classes and modules that implement the Flink source model. The interaction between these

components is key to the system’s design.

Architectural Flow

1. A `DataFlow` is initiated, configured with a specific source type (e.g., `Rafka`).
2. The `DataFlow` starts a single `SplitEnumerator` process for that source.
3. The `DataFlow` also starts one or more `SourceReader` processes (e.g., one per ActiveJob worker or thread).
4. Each `SourceReader` registers itself with the `SplitEnumerator`.
5. The `SplitEnumerator` discovers the work to be done (e.g., lists partitions in a Kafka topic) and creates a backlog of `Splits`.
6. As `SourceReader`s become available, they request `Splits` from the `SplitEnumerator`.
7. The `SplitEnumerator` assigns a `Split` to a `SourceReader`.
8. The `SourceReader` reads all data from its assigned `Split`. Once finished, it requests another `Split`.
9. This continues until the `SplitEnumerator` reports that there are no more `Splits` to process (for a bounded/batch source) or continues indefinitely (for an unbounded/streaming source).

Figure 1: The Flink Source API architecture, which this design emulates. [1]

3. Core Class Implementation

This section provides the Ruby implementation of the core classes in the `ActiveDataFlow::SourceSupport` namespace. These classes provide the generic framework that specific source implementations will build upon.

3.1. `Source` Class

The `Source` class is the main entry point and factory for creating the other components. It is initialized with the user’s configuration and is responsible for

creating the enumerator and readers.

```
# app/models/active_data_flow/source_support/source.rb

module ActiveDataFlow
  module SourceSupport
    # The main factory class for a specific data source implementation.
    class Source
      attr_reader :configuration

      def initialize(configuration)
        @configuration = configuration
      end

      # Factory method to create a new SplitEnumerator.
      # @param context [SplitEnumeratorContext] The context for the
enumerator.
      def create_enumerator(context)
        raise NotImplementedError
      end

      # Factory method to create a new SourceReader.
      # @param context [SourceReaderContext] The context for the reader.
      def create_reader(context)
        raise NotImplementedError
      end

      # Provides the boundedness of the source (either :unbounded or
:bounded).
      def boundedness
        raise NotImplementedError
      end

      # Provides a serializer for the source's splits.
      def split_serializer
        raise NotImplementedError
      end
    end
  end
end
```

3.2. `Split` Class

A `Split` is a simple, serializable data object that represents a portion of work.

```
# app/models/active_data_flow/source_support/split.rb

module ActiveDataFlow
  module SourceSupport
    # A marker module for all Split implementations.
    # A Split is a serializable object that represents a unit of work.
    module Split
      def split_id
        raise NotImplementedError, "Splits must have a unique ID"
      end
    end
  end
end
```

3.3. `SplitEnumerator` Class

The `SplitEnumerator` is the coordinator. It discovers splits and assigns them to readers. This base class provides the core API that all enumerator implementations must follow.

```ruby
# app/models/active_data_flow/source_support/split_enumerator.rb

module ActiveDataFlow
  module SourceSupport
    # The base class for all SplitEnumerators.
    class SplitEnumerator
      attr_reader :context

      def initialize(context)
        @context = context
      end

      # Starts the enumerator (e.g., to begin periodic discovery).
      def start
        # Default is no-op.
      end

      # Handles the registration of a new SourceReader.
      # @param reader_id [Integer] The unique ID of the reader.
      def add_reader(reader_id)
        # Default is no-op.
      end

      # Assigns splits to a requesting reader.
      # @param reader_id [Integer] The ID of the reader requesting work.
      def handle_split_request(reader_id)
        # Default is no-op.
      end

      # Takes back splits that were assigned to a failed reader.
      # @param splits [Array<Split>] The splits to be added back to the
backlog.
      def add_splits_back(splits)
        # Default is no-op.
      end

      # Closes the enumerator.
      def close
        # Default is no-op.
      end
    end
  end
end
```

3.4. `SourceReader` Class

The `SourceReader` is the worker. It requests splits, reads data, and emits records. This base class defines the interface for all reader implementations.

```ruby
# app/models/active_data_flow/source_support/source_reader.rb

module ActiveDataFlow
  module SourceSupport
    # The base class for all SourceReaders.
    class SourceReader
      attr_reader :context

      def initialize(context)
        @context = context
      end

      # Starts the reader.
      def start
        # Default is no-op.
      end

      # The main poll method to fetch records.
      # This method should be non-blocking.
      # @return [Array<Object>] An array of records.
      def poll
        raise NotImplementedError
      end

      # Notifies the reader that it has been assigned a set of splits.
      # @param splits [Array<Split>] The newly assigned splits.
      def add_splits(splits)
        # Default is no-op.
      end

      # Closes the reader.
      def close
        # Default is no-op.
      end
    end
  end
end
```

3.5. Context and Coordination

To facilitate communication between the `SplitEnumerator` and `SourceReader`s, we need context objects and a shared state/messaging mechanism. In a real Rails application, this could be implemented using Redis, a database table, or another shared memory store.

`SplitEnumeratorContext`

This object provides the `SplitEnumerator` with the means to interact with its readers.

```
# app/models/active_data_flow/source_support/split_enumerator_context.rb

module ActiveDataFlow
  module SourceSupport
    class SplitEnumeratorContext
      # Assigns one or more splits to a specific reader.
      def assign_split(split, reader_id)
        # Logic to send the split to the reader.
      end

      # Gets the number of currently registered readers.
      def current_parallelism
        # Logic to query the number of active readers.
      end
    end
  end
end
```

`SourceReaderContext`

This object provides the `SourceReader` with the means to request splits.

```ruby
# app/models/active_data_flow/source_support/source_reader_context.rb

module ActiveDataFlow
  module SourceSupport
    class SourceReaderContext
      # Sends a request for a split to the enumerator.
      def send_split_request
        # Logic to message the enumerator.
      end
    end
  end
end
```

4. Concrete Implementations

With the core framework in place, we can now create concrete implementations for different data sources. These classes will inherit from the base classes in `ActiveDataFlow::SourceSupport` and provide the specific logic for interacting with their respective external systems.

4.1. Rafka Source Implementation

This implementation provides a source for `Rafka`, a Kafka-compatible API backed by Redis streams. It is an **unbounded** source, meaning it will continuously listen for new messages.

RafkaSplit

A `RafkaSplit` represents a single partition of a Rafka (Redis) stream.

```ruby
# app/models/active_data_flow/source_support/rafka/rafka_split.rb

module ActiveDataFlow
  module SourceSupport
    module Rafka
      class RafkaSplit
        include ActiveDataFlow::SourceSupport::Split

        attr_reader :stream_name, :partition_id

        def initialize(stream_name, partition_id)
          @stream_name = stream_name
          @partition_id = partition_id
        end

        def split_id
          "#{stream_name}-#{partition_id}"
        end
      end
    end
  end
end
```

RafkaSplitEnumerator

The enumerator connects to Rafka/Redis to discover the partitions for the configured stream and assigns them to readers.

```ruby
# app/models/active_data_flow/source_support/rafka/rafka_split_enumerator.rb

module ActiveDataFlow
  module SourceSupport
    module Rafka
      class RafkaSplitEnumerator <
ActiveDataFlow::SourceSupport::SplitEnumerator
        def initialize(context, configuration)
          super(context)
          @configuration = configuration
          @stream_name = configuration[:stream_name]
          @splits = []
          @assigned_splits = {}
        end

        def start
          # Discover initial partitions.
          discover_splits
        end

        def handle_split_request(reader_id)
          # Assign the next available split to the reader.
          split = @splits.pop
          if split
            context.assign_split(split, reader_id)
            @assigned_splits[split.split_id] = reader_id
          end
        end

        private

        def discover_splits
          # In a real implementation, this would connect to Redis
          # and list the partitions for the stream.
          partitions = ::Rafka::Admin.new.partitions_for(@stream_name)
          @splits = partitions.map { |p_id| RafkaSplit.new(@stream_name,
p_id) }
        end
      end
    end
  end
end
```

RafkaSourceReader

The reader connects to Rafka/Redis and fetches records for its assigned partition.

```ruby
# app/models/active_data_flow/source_support/rafka/rafka_source_reader.rb

module ActiveDataFlow
  module SourceSupport
    module Rafka
      class RafkaSourceReader < ActiveDataFlow::SourceSupport::SourceReader
        def initialize(context, configuration)
          super(context)
          @configuration = configuration
          @assigned_splits = []
          @consumer = nil # The Rafka consumer instance
        end

        def add_splits(splits)
          @assigned_splits.concat(splits)
          # A real implementation would likely handle one split at a time.
          # For simplicity, we connect to the first assigned split.
          connect_to_split(@assigned_splits.first)
        end

        def poll
          return [] unless @consumer
          # Fetch the next batch of records from the stream.
          @consumer.read(max_batch_size: 100)
        end

        private

        def connect_to_split(split)
          return unless split
          @consumer = ::Rafka::Consumer.new(
            split.stream_name,
            @configuration[:consumer_group],
            @configuration[:consumer_name]
          )
        end
      end
    end
  end
end
```

4.2. File Source Implementation

This implementation provides a source for reading files from a directory. It is a **bounded** source, meaning it will process all the files in the directory and then stop.

FileSplit

A `FileSplit` represents a single file to be processed.

```ruby
# app/models/active_data_flow/source_support/file/file_split.rb

module ActiveDataFlow
  module SourceSupport
    module File
      class FileSplit
        include ActiveDataFlow::SourceSupport::Split

        attr_reader :file_path

        def initialize(file_path)
          @file_path = file_path
        end

        def split_id
          file_path
        end
      end
    end
  end
end
```

FileSplitEnumerator

The enumerator scans the configured directory, creating a `FileSplit` for each file it finds.

```ruby
# app/models/active_data_flow/source_support/file/file_split_enumerator.rb

module ActiveDataFlow
  module SourceSupport
    module File
      class FileSplitEnumerator <
ActiveDataFlow::SourceSupport::SplitEnumerator
        def initialize(context, configuration)
          super(context)
          @directory_path = configuration[:directory_path]
          @splits = []
        end

        def start
          # Discover all files in the directory at startup.
          Dir.glob("#{@directory_path}/*").each do |file_path|
            @splits << FileSplit.new(file_path)
          end
        end

        def handle_split_request(reader_id)
          split = @splits.pop
          if split
            context.assign_split(split, reader_id)
          else
            # Signal that there is no more work to be done.
            context.signal_no_more_splits(reader_id)
          end
        end
      end
    end
  end
end
```

FileSourceReader

The reader opens the file from its assigned `FileSplit` and reads it record by record.

```ruby
# app/models/active_data_flow/source_support/file/file_source_reader.rb

require 'csv'

module ActiveDataFlow
  module SourceSupport
    module File
      class FileSourceReader < ActiveDataFlow::SourceSupport::SourceReader
        def initialize(context, configuration)
          super(context)
          @format = configuration[:format]
          @current_split = nil
          @file_handle = nil
        end

        def add_splits(splits)
          # This simple reader only handles one split at a time.
          @current_split = splits.first
          open_file if @current_split
        end

        def poll
          return [] unless @file_handle
          # Read a batch of records from the file.
          records = []
          case @format
          when 'csv'
            # In a real implementation, this would read in batches, not all
at once.
            @file_handle.each do |row|
              records << row.to_h
            end
            @file_handle.close
            @file_handle = nil
          end
          records
        end

        private

        def open_file
          case @format
          when 'csv'
            @file_handle = CSV.open(@current_split.file_path, headers: true)
          else
```

```
            raise "Unsupported format: #{@format}"
          end
        end
      end
    end
  end
end
```

5. Conclusion

By adopting the core architectural patterns of Apache Flink's Data Source API, the `ActiveDataFlow::SourceSupport` module provides a powerful, scalable, and extensible foundation for data ingestion in a Rails environment. This design separates the concerns of work discovery (Enumerator) from work execution (Reader), allowing for parallel processing and robust fault tolerance. The concrete implementations for `Rafka` and `File` sources demonstrate the flexibility of this pattern for both unbounded streaming and bounded batch use cases.

6. References

[1] Apache Flink. (n.d.). *Data Sources*. Retrieved from https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/sources/