ActiveDataFlow: Heartbeat Trigger System Design

1. Introduction

This document outlines the database schema and controller logic for a heartbeat-triggered execution model within the `active_data_flow` engine. In this model, a central `/dataflow/heartbeat` endpoint is called at a regular interval (e.g., by a cron job or an external scheduler). This endpoint is responsible for identifying and triggering any `DataFlow` processes that are due to run.

This approach provides a simple, centralized, and resilient mechanism for scheduling long-running tasks without relying on complex, stateful background job schedulers within each application instance.

2. Database Schema Design

To support the heartbeat model, we need a database schema that can store the configuration and state of each data flow. The schema is composed of two main tables: `data_flows` to store the static configuration of each flow class, and `data_flow_runs` to log each individual execution.

2.1. `data_flows` Table

This table acts as the central registry for all defined `DataFlow` classes in the application. It stores their configuration, scheduling interval, and the status of their most recent run.

Migration

```
# db/migrate/YYYYMMDDHHMMSS_create_data_flows.rb

class CreateDataFlows < ActiveRecord::Migration[7.0]
  def change
    create_table :data_flows do |t|
      t.string :name, null: false, index: { unique: true } # Class name of
the DataFlow
      t.text :description
      t.boolean :enabled, default: true, null: false
      t.jsonb :configuration, default: {}, null: false # Stores source,
sink, and other settings
      t.integer :run_interval # Interval in seconds for heartbeat-based
execution
      t.datetime :last_run_at # Timestamp of the last time the flow was
triggered
      t.string :last_run_status # e.g., 'success', 'failed', 'in_progress'
      t.text :last_run_error

      t.timestamps
    end
  end
end
```

Model

The corresponding `DataFlow` model includes validations, scopes for querying due flows, and methods to manage its state.

```ruby
# app/models/data_flow.rb

class DataFlow < ApplicationRecord
  validates :name, presence: true, uniqueness: true
  validates :run_interval, numericality: { only_integer: true, greater_than:
0 }, allow_nil: true

  # Scope to find all flows that are enabled, have a run_interval, and are
due to be run.
  scope :due_to_run, -> {
    enabled
      .where.not(run_interval: nil)
      .where("last_run_at IS NULL OR last_run_at <= ?
 NOW() - INTERVAL '1 second' * run_interval")
  }

  # Triggers the execution of the data flow by enqueuing an ActiveJob.
  def trigger_run!
    # Update the status to prevent concurrent runs from the same heartbeat.
    update!(last_run_at: Time.current, last_run_status: 'in_progress')
    DataFlowRunJob.perform_later(self.id)
  end
end
```

2.2. `data_flow_runs` Table

This table provides a detailed audit log of every individual execution of a `DataFlow`. It stores the status, start and end times, and any errors that occurred. This is invaluable for monitoring, debugging, and understanding the history of each process.

Migration

```
# db/migrate/YYYYMMDDHHMMSS_create_data_flow_runs.rb

class CreateDataFlowRuns < ActiveRecord::Migration[7.0]
  def change
    create_table :data_flow_runs do |t|
      t.references :data_flow, null: false, foreign_key: true
      t.string :status, null: false # e.g., 'pending', 'in_progress',
'success', 'failed'
      t.datetime :started_at
      t.datetime :ended_at
      t.text :error_message
      t.text :error_backtrace

      t.timestamps
    end
  end
end
```

Model

The `DataFlowRun` model represents a single execution and is primarily used for logging and status tracking.

```
# app/models/data_flow_run.rb

class DataFlowRun < ApplicationRecord
  belongs_to :data_flow

  validates :status, presence: true, inclusion: { in: %w(pending in_progress
success failed) }

  # Convenience method to calculate the duration of the run.
  def duration
    return nil unless started_at && ended_at
    ended_at - started_at
  end
end
```

3. Controller and Job Logic

The controller and job are the dynamic components that bring the heartbeat system to life. The controller responds to the heartbeat requests, and the job executes the actual data flow logic asynchronously.

3.1. `DataFlowsController`

This controller exposes the `/dataflow/heartbeat` endpoint. It is designed to be lightweight and fast, as it only needs to identify which flows are due and enqueue them for background processing. It does not execute the flows directly.

```
# app/controllers/data_flows_controller.rb

class DataFlowsController < ApplicationController
  # A POST request to /dataflow/heartbeat triggers this action.
  # This endpoint should be secured to prevent unauthorized access.
  def heartbeat
    due_flows = DataFlow.due_to_run
    triggered_count = 0

    # Using a transaction ensures that we don't have race conditions
    # if multiple heartbeats arrive at the same time.
    DataFlow.transaction do
      # We lock the records to prevent other transactions from modifying
them.
      due_flows.lock("FOR UPDATE SKIP LOCKED").each do |flow|
        flow.trigger_run!
        triggered_count += 1
      end
    end

    render json: {
      status: "Heartbeat processed",
      flows_due: due_flows.count,
      flows_triggered: triggered_count
    }, status: :ok
  end
end
```

3.2. `DataFlowRunJob`

This ActiveJob is the workhorse of the system. It is responsible for instantiating the correct `DataFlow` class, executing its `run` method, and recording the outcome.

```ruby
# app/jobs/data_flow_run_job.rb

class DataFlowRunJob < ApplicationJob
  queue_as :default

  def perform(data_flow_id)
    data_flow = DataFlow.find(data_flow_id)
    run_log = data_flow.data_flow_runs.create!(status: 'pending',
started_at: Time.current)

    begin
      run_log.update!(status: 'in_progress')

      # Instantiate the DataFlow class from its name.
      flow_class = data_flow.name.constantize
      flow_instance = flow_class.new(data_flow.configuration)

      # Execute the flow's logic.
      flow_instance.run

      # Record the successful outcome.
      data_flow.update!(last_run_status: 'success')
      run_log.update!(status: 'success', ended_at: Time.current)

    rescue => e
      # If any error occurs, record the failure details.
      data_flow.update!(last_run_status: 'failed', last_run_error:
e.message)
      run_log.update!(
        status: 'failed',
        ended_at: Time.current,
        error_message: e.message,
        error_backtrace: e.backtrace.join("\n")
      )
      # Re-raise the exception to allow for standard ActiveJob error
handling (e.g., retries).
      raise e
    end
  end
end
```

”’

4. Workflow and Example

This section illustrates the end-to-end workflow of the heartbeat system and provides a concrete example of a `DataFlow` configured for heartbeat-based execution.

4.1. End-to-End Workflow

The process unfolds in a clear sequence of events:

1. **External Scheduler**: A cron job or a similar scheduling mechanism sends a `POST` request to the `/dataflow/heartbeat` endpoint at a regular, predefined interval (e.g., every minute).
2. **Heartbeat Controller**: The `DataFlowsController#heartbeat` action is invoked.
    - It queries the `data_flows` table for all enabled flows that are due to run using the `due_to_run` scope.
    - It opens a database transaction and acquires a lock on the due flow records using `FOR UPDATE SKIP LOCKED` to prevent race conditions from concurrent heartbeat calls [1].
3. **Job Enqueueing**: For each locked `DataFlow` record, the controller calls the `trigger_run!` method.
    - This method immediately updates the `last_run_at` timestamp and sets the `last_run_status` to `in_progress`.
    - It then enqueues a `DataFlowRunJob` with the ID of the `DataFlow` record.
4. **Asynchronous Execution**: An ActiveJob worker picks up the `DataFlowRunJob` from the queue.
    - The job creates a new `DataFlowRun` log entry with a `pending` status.
    - It finds the `DataFlow` record and instantiates the corresponding `DataFlow` class (e.g., `ProductUpdateFlow`).
    - The job executes the `run` method of the `DataFlow` instance.
5. **Logging and Status Update**: Upon completion, the job updates the status of both the `DataFlow` and the `DataFlowRun` records.

- **On Success**: The `last_run_status` is set to `success` on the `DataFlow` record, and the `DataFlowRun` log is updated to `success` with an `ended_at` timestamp.
- **On Failure**: The job catches the exception, updates the `last_run_status` to `failed` with the error message, and records the full error details in the `DataFlowRun` log.

4.2. Example: Heartbeat-Triggered DataFlow

Here is how the `ProductUpdateFlow` would be configured for heartbeat-based execution. The core logic is encapsulated within the `run` method, which is called by the `DataFlowRunJob`.

```ruby
# app/data_flows/product_update_flow.rb

class ProductUpdateFlow
  include ActiveDataFlow::DataFlow # Provides the #initialize(config) method

  def run
    # The configuration is passed from the DataFlow record in the database.
    source_config = @configuration['source']
    sink_config = @configuration['sink']

    source = ActiveDataFlow::Source.new(source_config)
    sink = ActiveDataFlow::Sink.new(sink_config)

    # The core logic remains the same: read, transform, write.
    source.each do |message|
      transformed_message = transform(message)
      sink.write(transformed_message)
    end
  end

  private

  def transform(message)
    message.merge(processed_at: Time.current)
  end
end
```

To enable this flow to run via heartbeat, an administrator would create a `DataFlow` record in the database with the following attributes:

- **name**: `"ProductUpdateFlow"`
- **enabled**: `true`
- **run_interval**: `3600` (to run every hour)
- **configuration**:

```
{
  "source": {
    "type": "Rafka",
    "stream_name": "inventory_updates",
    "consumer_group": "data_flow_processors",
    "consumer_name": "product_update_flow_1"
  },
  "sink": {
    "type": "ActiveRecord",
    "model_name": "Product"
  }
}
```

5. Security Considerations

The `/dataflow/heartbeat` endpoint is a powerful entry point into the application that can trigger resource-intensive processes. It is critical that this endpoint be secured to prevent unauthorized access and potential denial-of-service attacks. Recommended security measures include:

- **IP Whitelisting**: Restrict access to the endpoint to known IP addresses of the scheduler.
- **Authentication Token**: Require a secret token in the request headers that is validated by the controller.
- **Network-Level Security**: Place the endpoint behind a firewall or within a private network (VPC) if possible.

6. References

[1] PostgreSQL Documentation. (n.d.). *The Locking Clause*. Retrieved from https://www.postgresql.org/docs/current/sql-select.html#SQL-FOR-UPDATE-SHARE
“