Implementation

In this chapter, we will discuss how we implement all those systems and configure them to provide an integrated platform that will be effective in the three operations that we mentioned before: integration data, storing data, analyze and visualization.

So we used apache NiFi for ingestion of the data, Apache Hadoop and Apache Hive to the storage system, and Apache Spark for analyzing and visualization. Also, we install some of Hadoop echo system in order of building and utilizing the platform: Oozei, Zookeeper, Ambari, HUE. We install all of that on Cloudera CDH through Cloudera Manager.

1. Implementing Data Ingestion Layer

As we said before, we installed Apache NiFi on Cloudera CDH by adding CFM to it, and this allows us to unify entire subsystems into one system because Cloudera CDH without adding CFM will allow us to store, analyze, visualize and even ingestion but not with NiFi. The alternative was to install NiFi on other devices and connect to the main node after configuring the channel. Moreover, taking a round trip which is the bottleneck of speed, which means intensive use for network resources, and that will slow down the system.

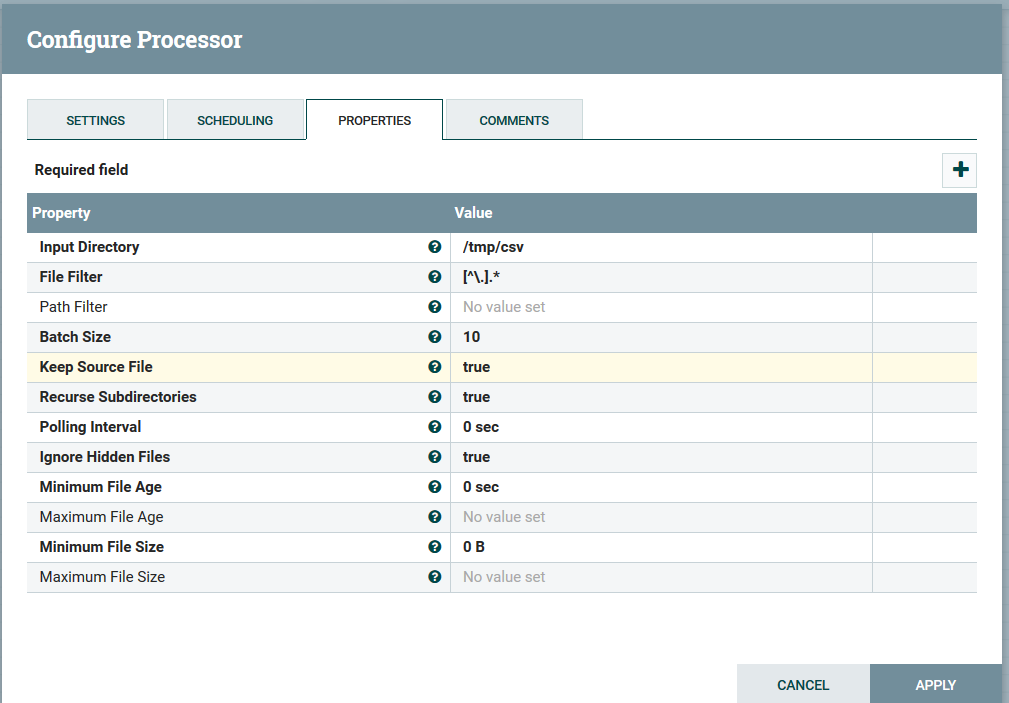
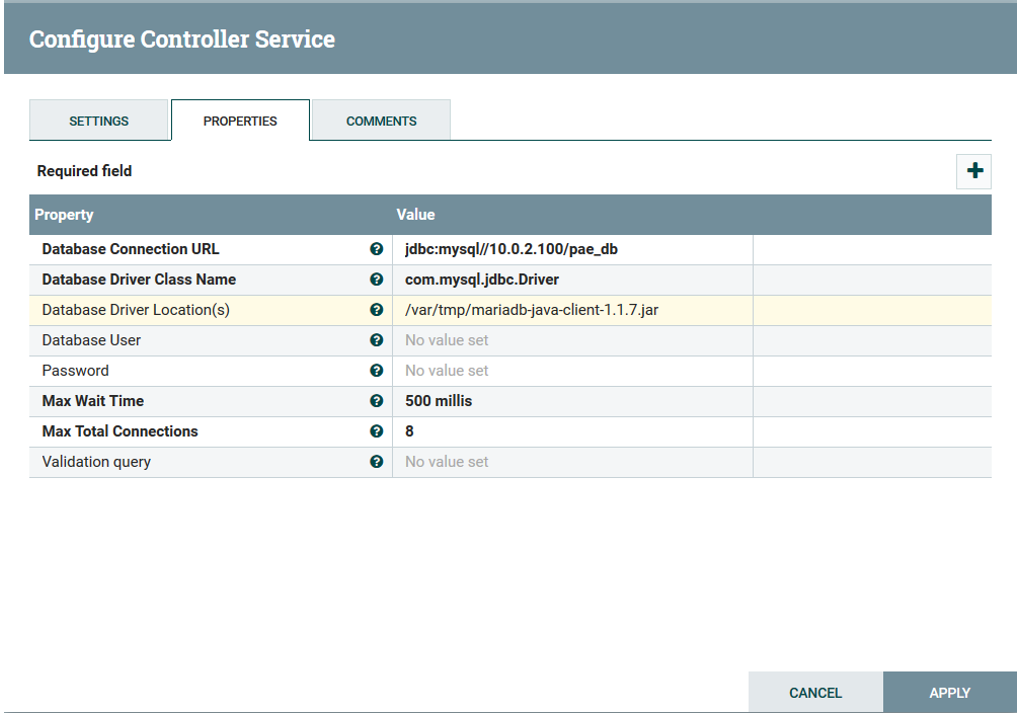
In the Ingestion process, we ingest data from three different sources:

* the CSV file, which contains a different file from many offices and departments
* MySQL database contains important information about crops and lands.
* Weather data from Openweathermap.com using API.
  1. Developing Process to Ingest CSV files

Processors that ingest data from the CSV data source to the main repository in Apache Hive are all placed into a single process group. A process group enables the grouping of processors into one. These processors extract data from each CSV file and stream it directly into Apache Hive.

The following steps below are followed to build the data ingestion platform for the CSV data source.

1. The GetFile processor is added and configured to ingest data from the external location of the PAE data to Apache Nifi.

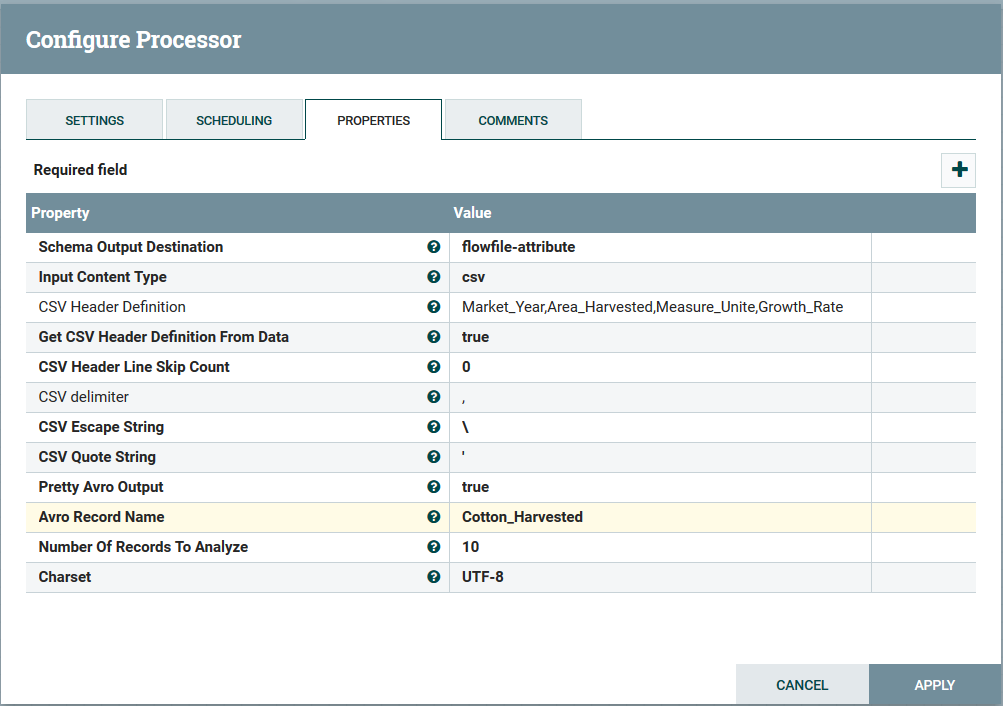
Below is the GetFile processor with its configurations.

GetFile processor to get CSV data to Apache NiFi

The Input Directory: specifies the location of the CSV file containing the PAE data. It is located in “???????????????”.

1. The InferAvroSchema is added and configured to examine the content of the file before it is ingested to the main repository in Hive.

Below is the InferAvroSchema for Cotton Harvest Area with configurations.



InferAcroSchema processor to examine the content of the file

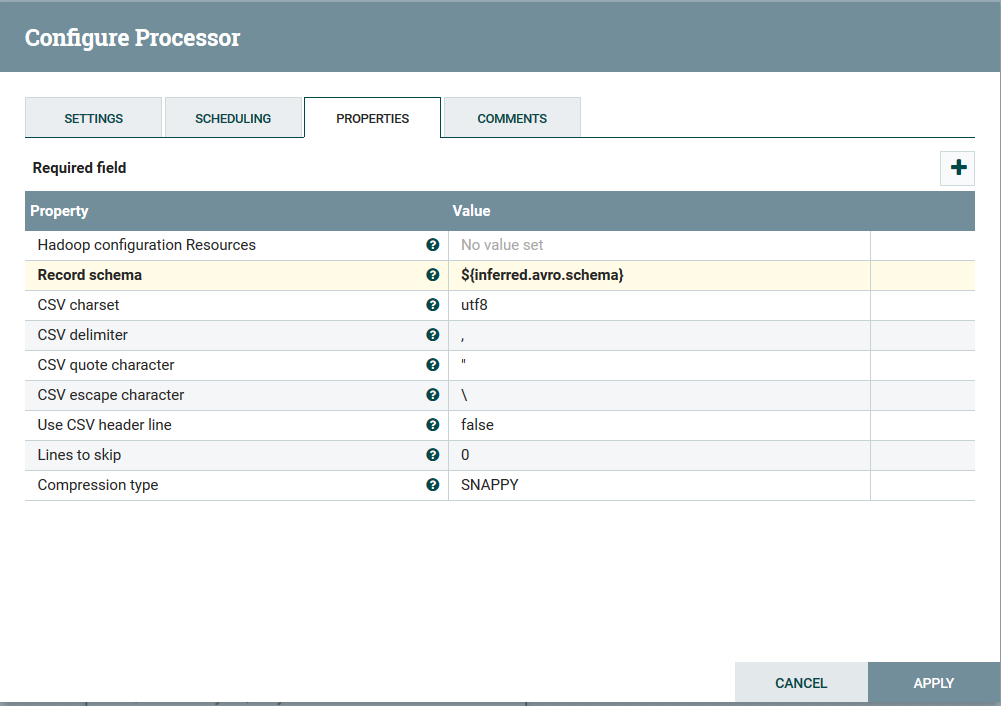
The Schema Output Destination specifies whether the Avro schema is written as a new attribute “inferred.avro.schema.”

The Input Content-Type specifies the content type of the incoming flow file. It supports files that are either in CSV or JSON.

The CSV Header Definition specifies a list of columns separated by commas that serve as the Header for the CSV file. This applies only to CSV files. These columns will be retrieved as the header definition of the CSV file.

iii. The ConvertCSVToAvro processor is configured to convert the CSV file to Avro format. This is because the PutHiveStreaming processor only understands data in Avro format.

Below is the ConvertCSVToAvro processor and its configuration:



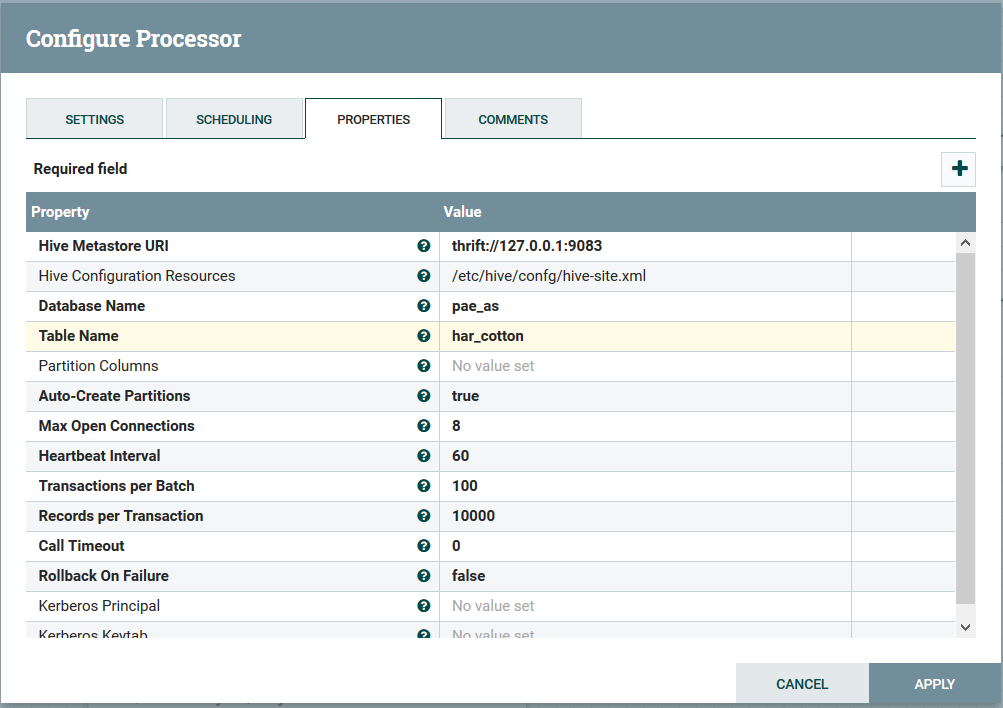
ConvertCSVToAvro processor to convert the CSV file to AVRO format

The Hadoop configuration Resources specifies the list of files that contain Hadoop’s configuration. It will use the default configuration to search for the files when left empty.

The Record schema specifies the outgoing Avro schema for each record created from a CSV row.

1. The PutHiveStreaming processor is configured to ingest the data to Hive.

Below is the PutHiveStreaming processor with configurations.



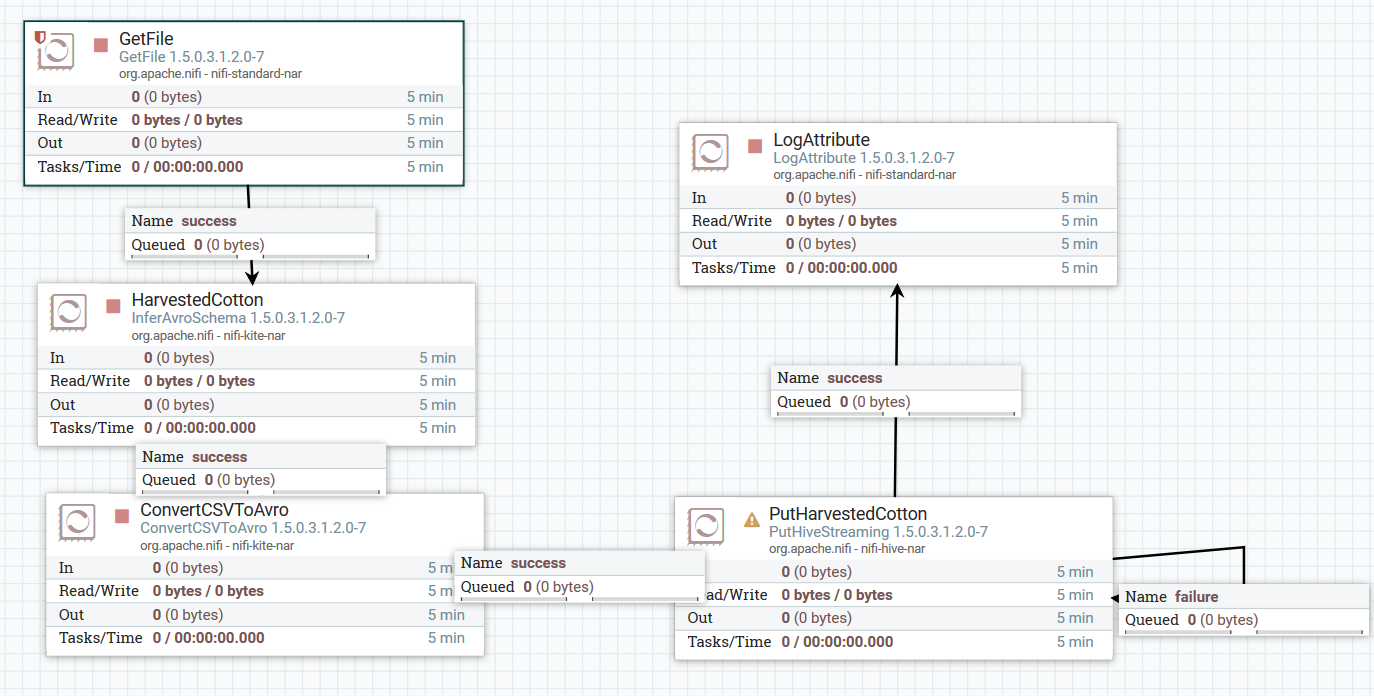
PutHiveStreaming processor to Ingest PAE data to Hive

Below is the SQL syntax, which creates the table in Hive.

Create table har\_cotton***(***Market\_Year int, Area\_Harvested Float, Measure\_Unite String, Growth\_Rate Float

***) CLUSTERED BY(Market\_Year) INTO 2 BUCKETS STORED AS ORC tblproperties("transactional"="true");***

1. The LogAttribute processor is configured to tell whether the data is successfully ingested or not.



a complete flow with processors to ingest CSV data to the main repository

we did this for a cotton harvested area by year, but of course, we have the following file table ?? which will get the same setting:

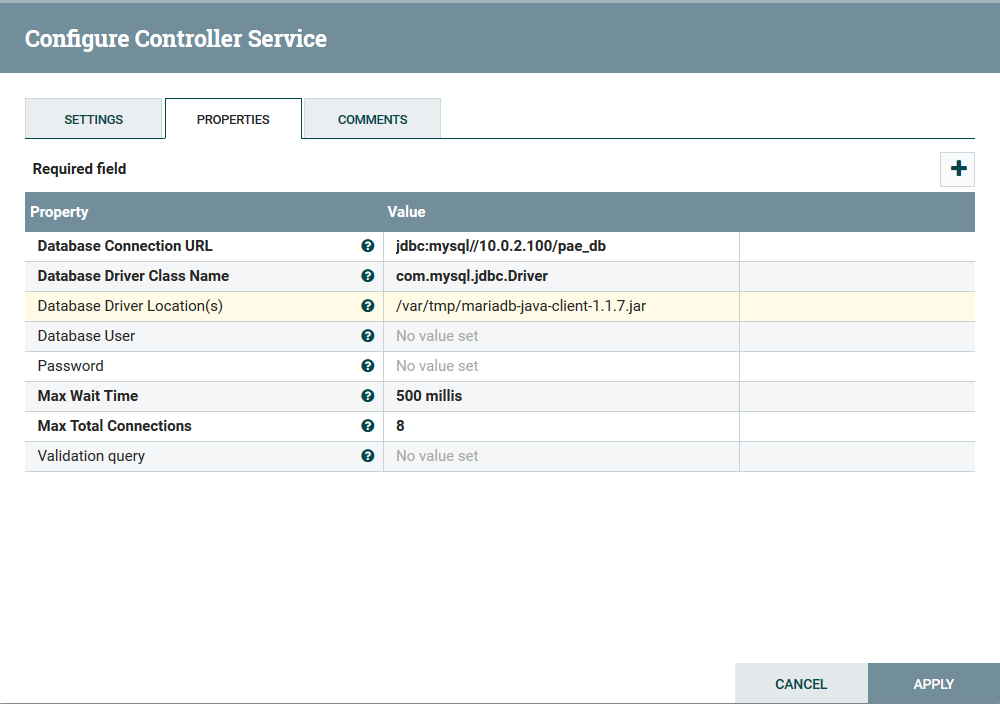
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| CSV file by year | cotton | Sorghum | Millet | Wheat |
| Area Harvested | har\_cotton | har\_sorg | har\_ millet | har\_wheat |
| Beginning Stock | bstock\_cotton | bstock\_ sorg | bstock\_ millet | bstock\_ wheat |
| Ending Stock | estock\_cotton | estock\_ sorg | estock\_ millet | estock\_ wheat |
| Exports | exports\_cotton | exports\_ sorg | exports\_ millet | exports\_ wheat |
| Imports | imports\_cotton | imports\_ sorg | imports\_ millet | imports\_ wheat |
| Production | production\_cotton | production\_ sorg | production\_ millet | production\_ wheat |
| Yield | yield\_cotton | yield\_ sorg | yield\_ millet | yield\_ wheat |

The tables name for all database table

* 1. Developing Process to Ingest MySQL to Hive

In order to ingest data from MySQL database to Hive, some NiFi processors had been used. The step to perform such operation are described as follow:

* To group all the processors, a **Process Group** is added and configured
* The **database controller** service is added and configured to establish a connection to the external database, as shown in the figure ??



Controller Service to connect to PAE\_DB external database source

From the figure?? above, the Database Connection URL to connect to the PAE\_DB data source is “jdbc:mysql//10.0.2.100/pae\_db”

The Database Driver Class Name for MySQL database “com.mysql.jdbc.Driver”.

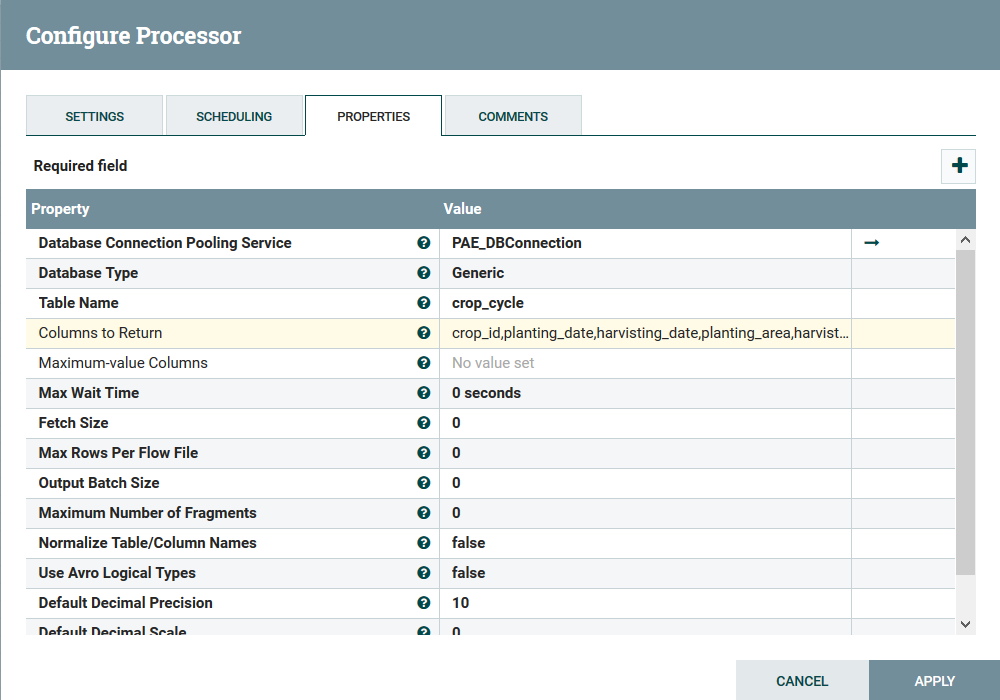
The database driver is located in “/var/tmp/mysql-connector-java-5.1.47.jar”.

The Database User is “root”

* To extract the data from the external database to Apache Nifi, a **QueryDatabaseTable** is added and configured for each table in the database.

The figure?? below is the QueryDatabaseTable processor with configurations to connect to the PAE\_DB data source (i.e., MySQL) and ingest data to Apache Nifi from a table in the PAE\_DB data source.

Configurations are done to ingest data from the crop\_cycle table in the pae\_db database.



QueryDatabaseTable processor to ingest data to Apache Nifi form the crop table in the pae\_db data source

As shown in configurations above:

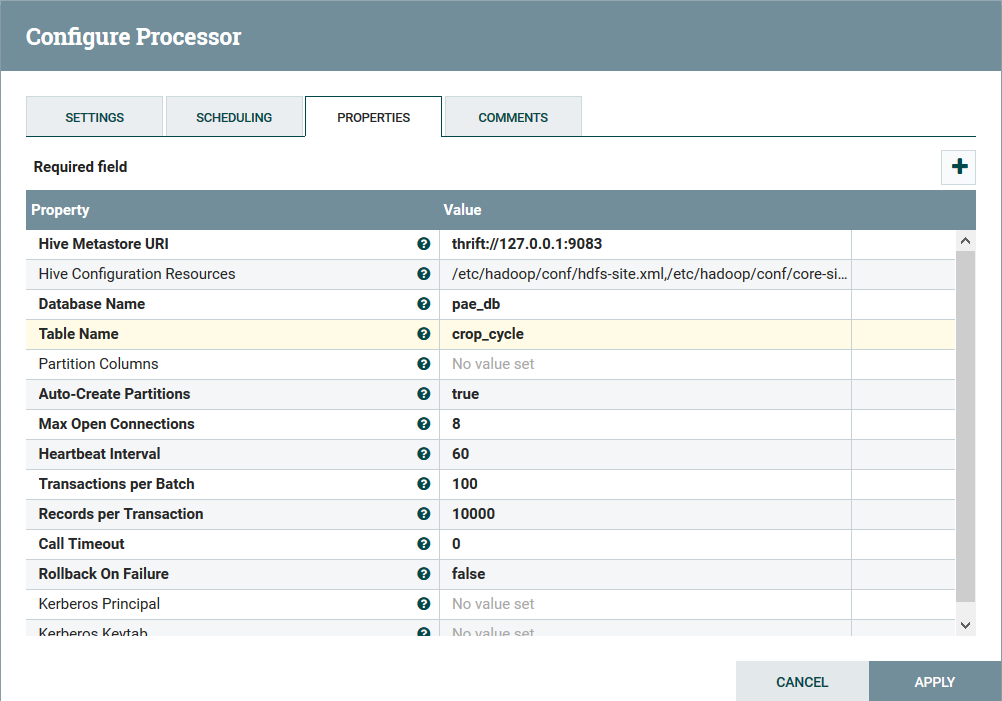
The QueryDatabaseTable processor connects to the pae\_db database through the pae\_db controller service. Therefore the Database Connection Pooling Service is “PAE\_DBConnection”.

The Database Type is MySQL. The Table Name is crop.

The Columns to return are crop\_id, planting\_date, harvisting\_date, planting\_area, harvisting\_area, production, yield, land. This means that only data for the specified columns are ingested.

* The **PutHiveStreaming** processor is added and configured to stream data from Apache Nifi directly into the HIVE data warehouse.

The figure?? is the PutHiveStreaming processor with configurations to stream data of the land table from the QueryDatabaseTable processor to HIVE



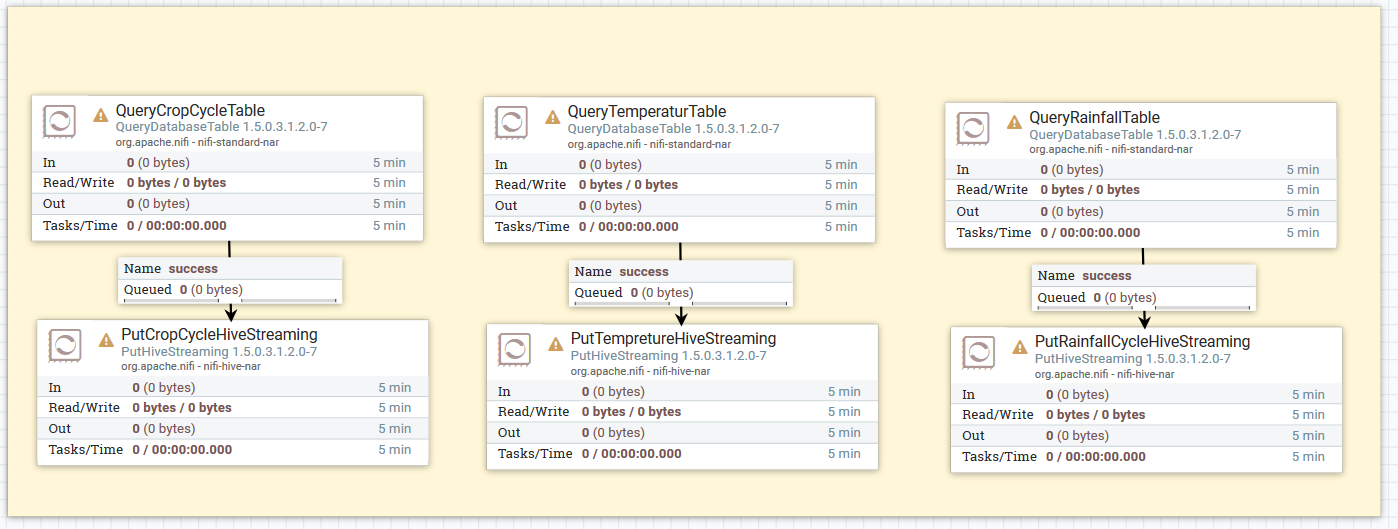
PutHiveStreaming processor to stream pae\_db data to Hive

Data from all data sources are streamed to a single data warehouse (i.e., pae-repo) in Hive.

Therefore the configurations for the Hive Metastore URI, Hive Configuration Resources, and the Database Name are the same PutHiveStreaming processors for all the data sources. Below is an SQL syntax that creates a table in HIVE to ingest data for the crop table.

Create table crop\_cycle(crop\_id int, planting\_date date, harvisting\_date date, planting\_area float, harvisting\_area float, production float, yield float , land int) CLUSTERED BY(crop\_id) INTO 3 BUCKETS STORED AS ORC tblproperties("transactional"="true");

Data from crop\_cycle, temperatures, and rainfall tables are ingested to Hive from the PAE data source.



flow with processors to ingest data to the main repository from the pae\_db external data source

* 1. Developing Process to Ingest RestFUL API to Hive

To connect to the OpenWeatherMap API using Autonomous REST connector, we can point it to the endpoint with the default values.

I connected to the weather API series offered by openweathermap.com. I want to get all-time series intraday values for the weather with an interval of 1 day, here would be my request:

[https://samples.openweathermap.org/data/2.5/weather?id={City-ID}&appid={Key}](https://samples.openweathermap.org/data/2.5/weather?id=%7bCity-ID%7d&appid=%7bKey%7d)

Open weather API basics

Open weather API Web Service returns information about the weather, but we will retrieve only the following: City geolocation, longitude, and latitude (coord.lon, coord.lat), Temperature min and max (main.temp\_min, main.temp\_max), Humidity(main.humidity ), Wind speed (wind.speed) , Rain volume (rain.3h) , Time of data (dt), Sunrise (sys.sunrise ), Sunset time(sys.sunset ), and City name(name). this data of interest had been based on Latitude and Longitude coordinates that are passed into HTTP requests. Or we can pass the city ID or name also.

All requests are accessed through an HTTP request and return either JSON or XML response. the necessary components to use the weather API is https:// protocol, API Key, and city identifier.

ValidateWeatherTransitData process group

Obtain API key for NiFi’s InvokeHTTP process

location is represented as Latitude and Longitude coordinates, but in this process, we use city id. So to get the previous weather data, we use the following HTTP URL of the following definition, which we will need for NiFi:

<https://samples.openweathermap.org/data/2.5/weather?id=379252&appid=b6907d289e10d714a6e88b30761fae22>

The output can come in two formats: JSON or XML. We will use JSON for this process. We have the other required parameters also.

The HTTP URL had built with the parameters below, so we can insert the URL as a property value into InvokeHTTP later in the tutorial.

* API Key = ccb227ec77119477b33263561add35da
* City id for khartoum=379252

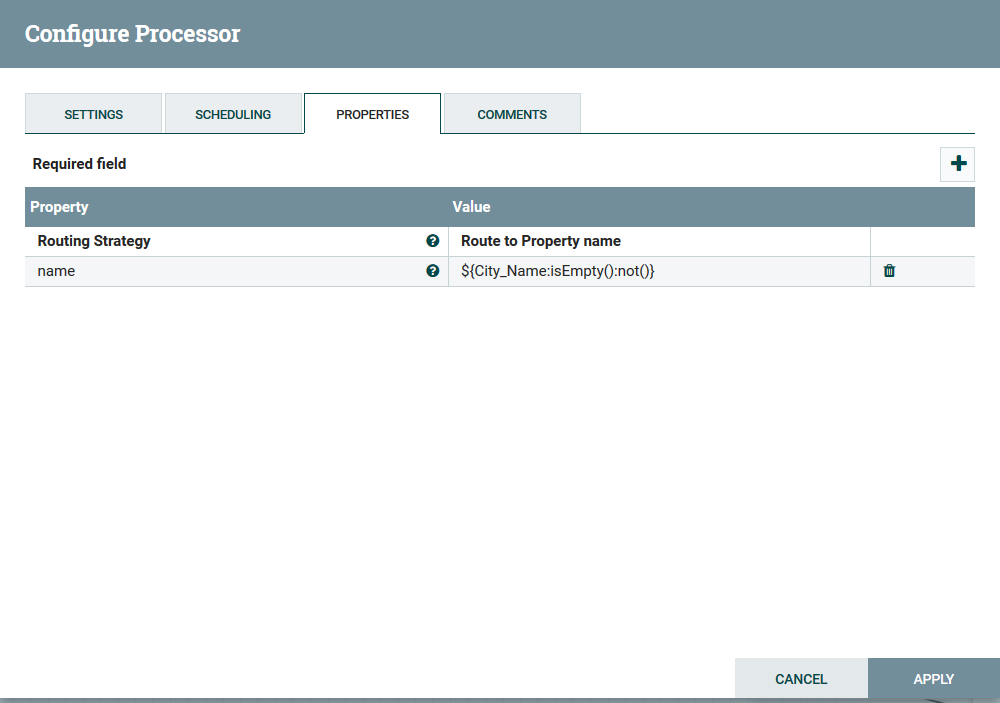
Create a process group

We separate the whole process into two process groups. The first one ValidateWeatherTransitData to get data from OpenWeatherMap API and the second one StoreDataAsJSONToDisk to save the data into JSON file on the HDFS

1. The first processor group  ValidateWeatherTransitData

We added four processor: RouteOnAttribute, InvokeHttp, EvaluateJsonPath, and RouteOnAttribute again.

The first processor is RouteOnAttribute it is just as an initiator for InvokeHttpAdd to make sure we will pass the city name to the invokeHttp processor



Configuring RouteOnAttribute with the name ValidateWeatherCityData

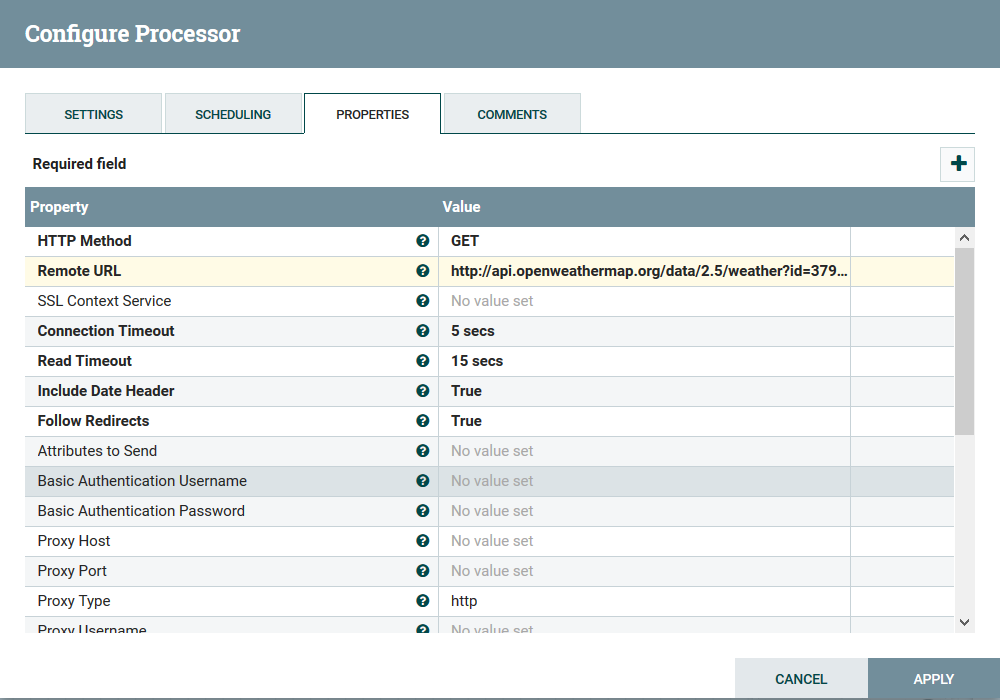
We inserted new properties : name with the following value ${City\_Name:isEmpty():not()} to make sure that city is not empty

The second processor was InvokeHttp. Where we put the API URL to pull weather data from OpenWeatherMap

in InvokeHTTP configure properties we configured the Remote URL with the following value

<http://api.openweathermap.org/data/2.5/weather?id=379252&APPID=ccb227ec77119477b33263561add35da>

with Http Method equal GET

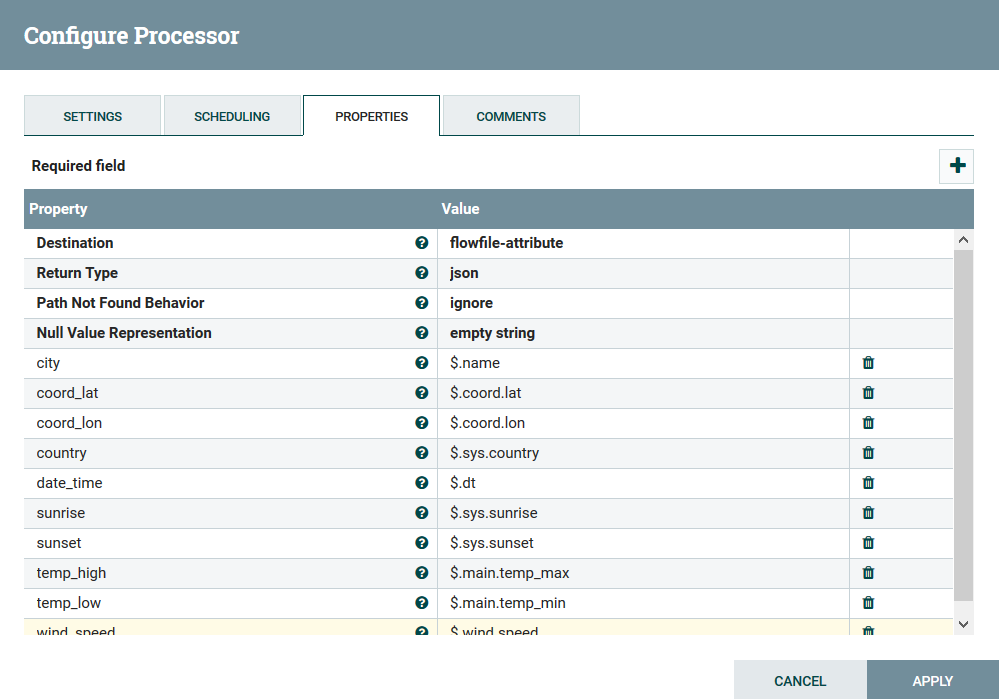


Configure InvokeHTTP to pull data from API URL

The third processor is EvaluateJsonPath; it used to determine which data we need from the response of the OpenWeatherMap site. We add ten properties, as shown in the table ??.

Table 3: Update EvaluateJsonPath Properties Tab

| **Property** | **Value** |
| --- | --- |
| City | $.name |
| Coord\_lat | $.coord.lat |
| coord\_lon | $.coord.lon |
| country | $sys.country |
| date­\_time | $.dt |
| sunrise | $sys.sunrise |
| sunset | $.sys.sunset |
| temp\_low | $.main.temp\_max |
| temp\_hight | $.main.temp\_max |
| Wind\_speed | $.wind.speed |

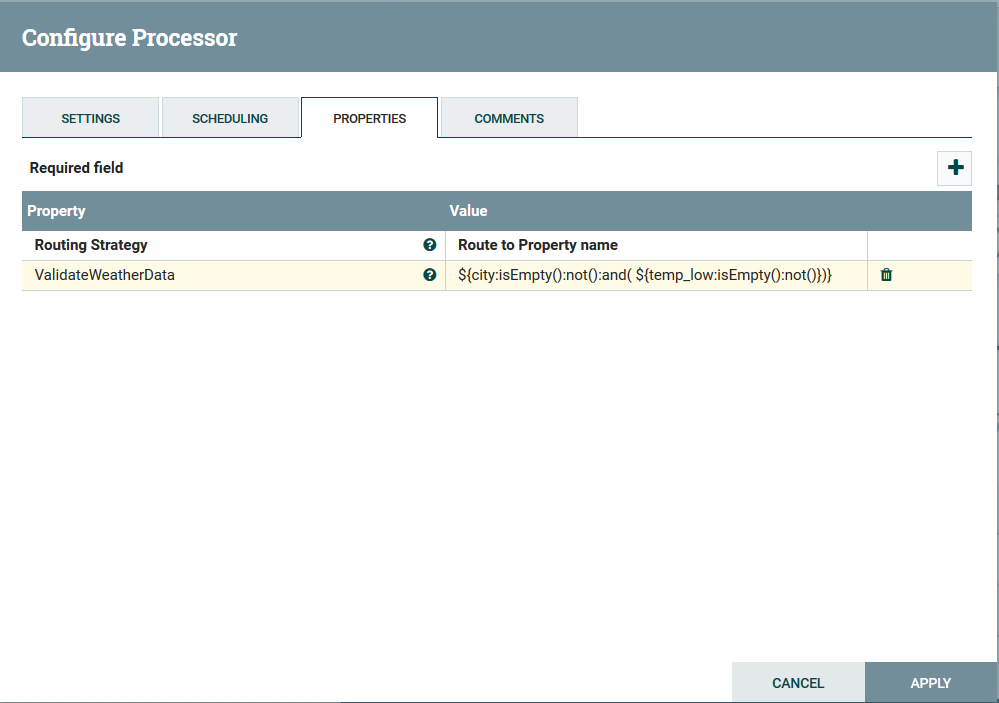


Configuration of EvaluateJsonPath to extract weather data

, the fourth processor was RouteOnAttribute to Validate Open Weather Map Data. It configured to check wither we obtain data from the previous operations or not.

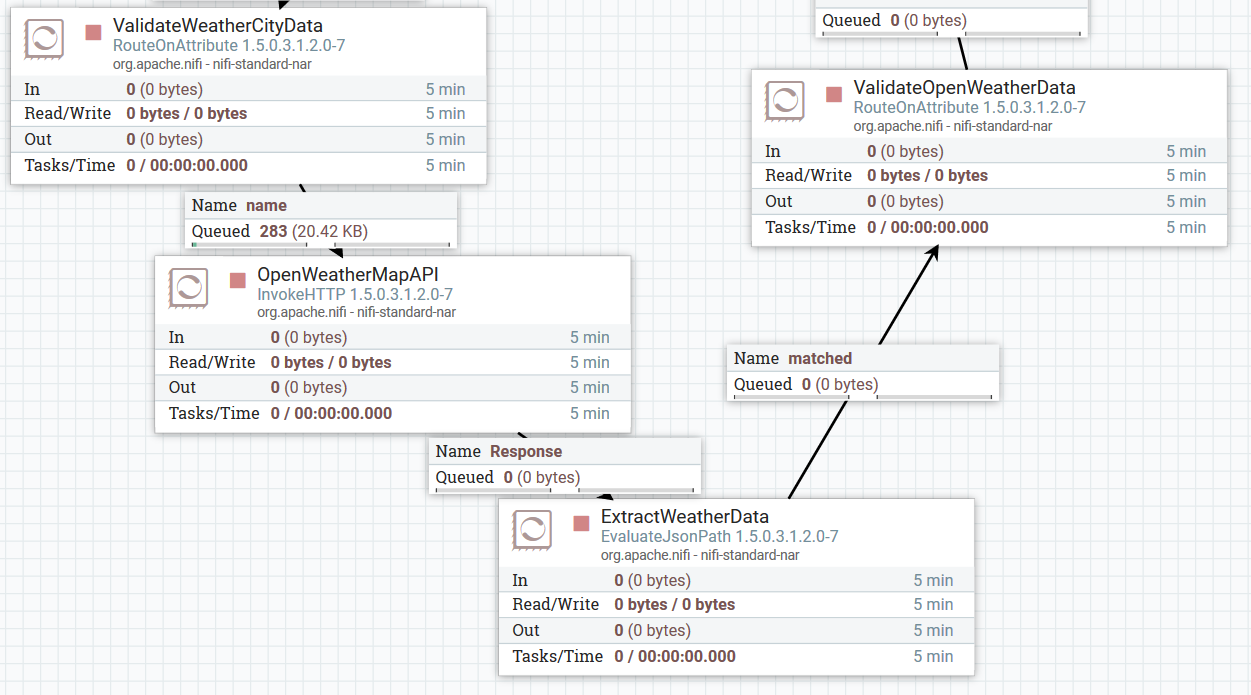
And as it shown in picture ??. we set a new propery ValidateWeatherData to ${city:isEmpty():not():and(

${temp\_low:isEmpty():not()})}



Configuration of RouteOnAttribute to Validate the weather data

The last layout for processor group as shown in figure ??

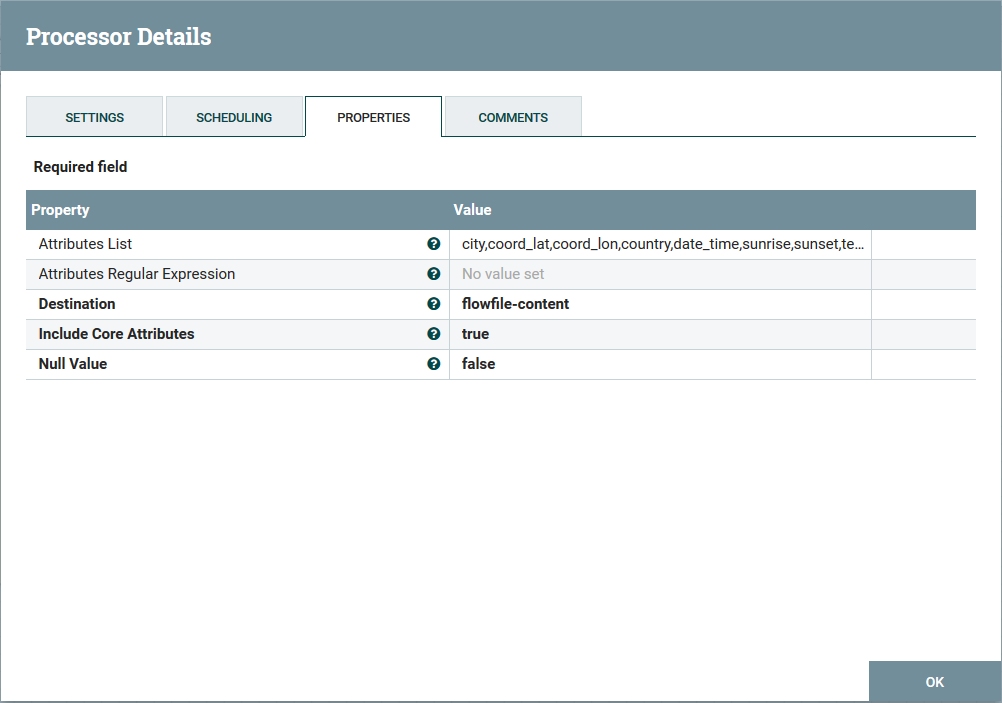


The layout of ValidateWeatherTransitData processor group

1. the second processor group is StoreDataAsJSONToDisk and we used it to store the weatherdata in json format on HDFS

here we also had four processor: AttributeToJSON, MeregeContent, UpdateAttribute, and PutFile.

1. The first processor is AttributeToJSON. We collect all data that we net from the series we obtained from OpenWeatherMap API. For that we set Attribute List properties to city,coord\_lat,coord\_lon,country,date\_time,sunrise,sunset,temp\_high,temp\_low,wind\_speed to ensure that we do not miss a property. As shown in figure ??. Attributes List takes FlowFile attribute parameters and presents them in JSON format. Destination stores the output as content in the FlowFile.



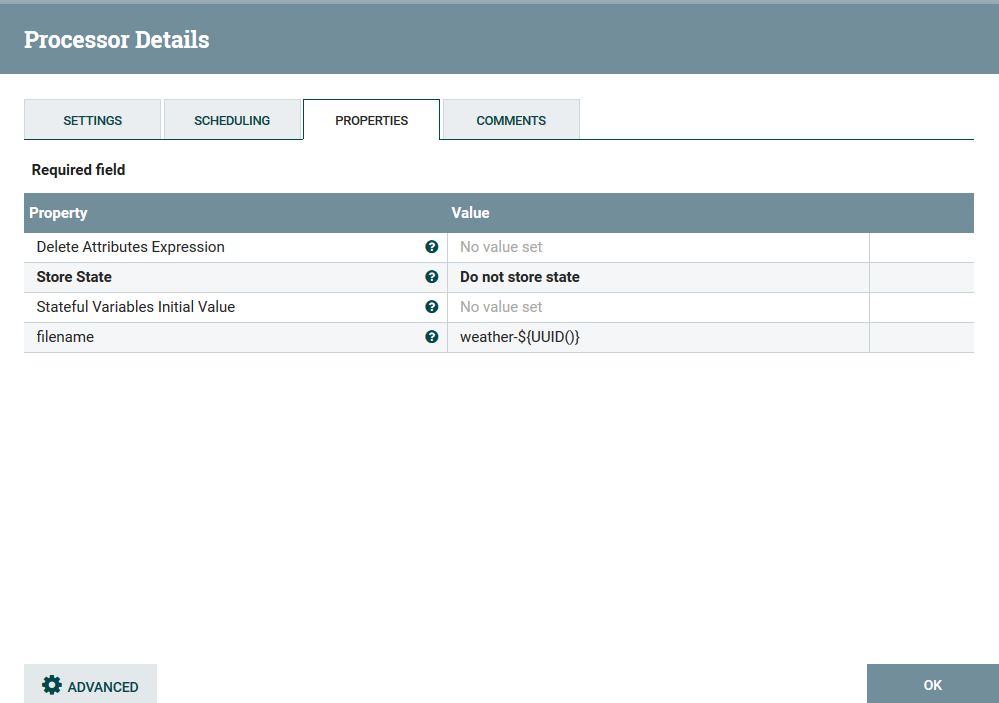
The setting for AttributeToJSON processor to map all attribute to variables

1. The second processor is MeregeContent to combine multiple FlowFiles together. We make some changes in the default value, as shown in the table ??.

Table 2: Update MergeContent Properties Tab

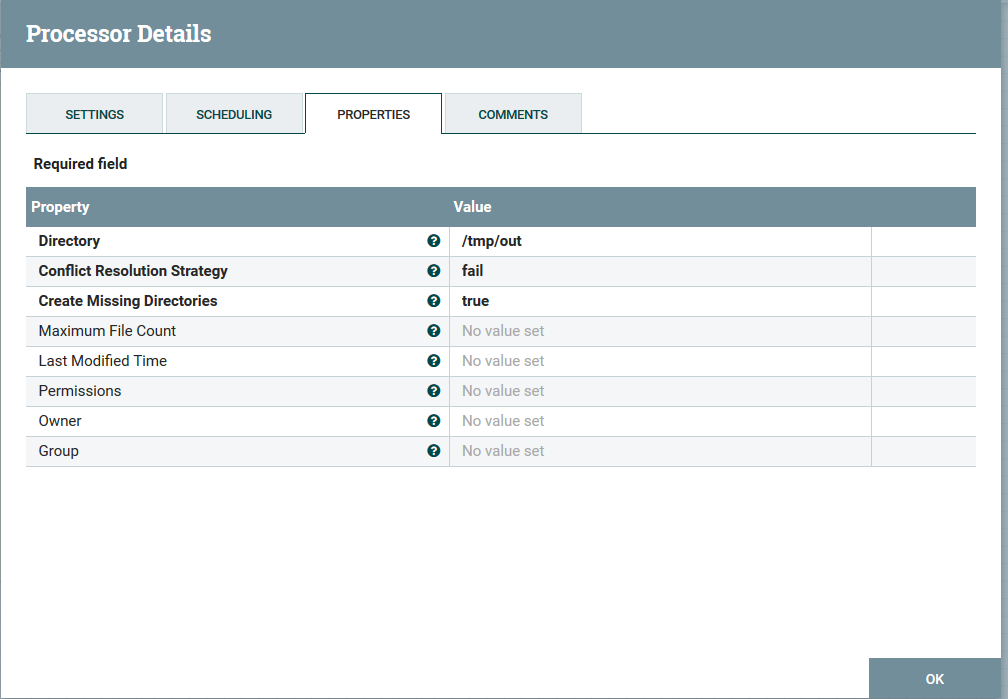
| **Property** | **Value** |
| --- | --- |
| Minimum Number of Entries | 10 |
| Maximum Number of Entries | 15 |
| Delimiter Strategy | Text |
| Header | [ |
| Footer | ] |
| Demarcator | , |

1. The third processor is UpdateAttribute to Make Each FlowFile name Unique. At the processor configuration properties tab. We Added the filename property with the value weather-${UUID()}. As shown in figure ??.

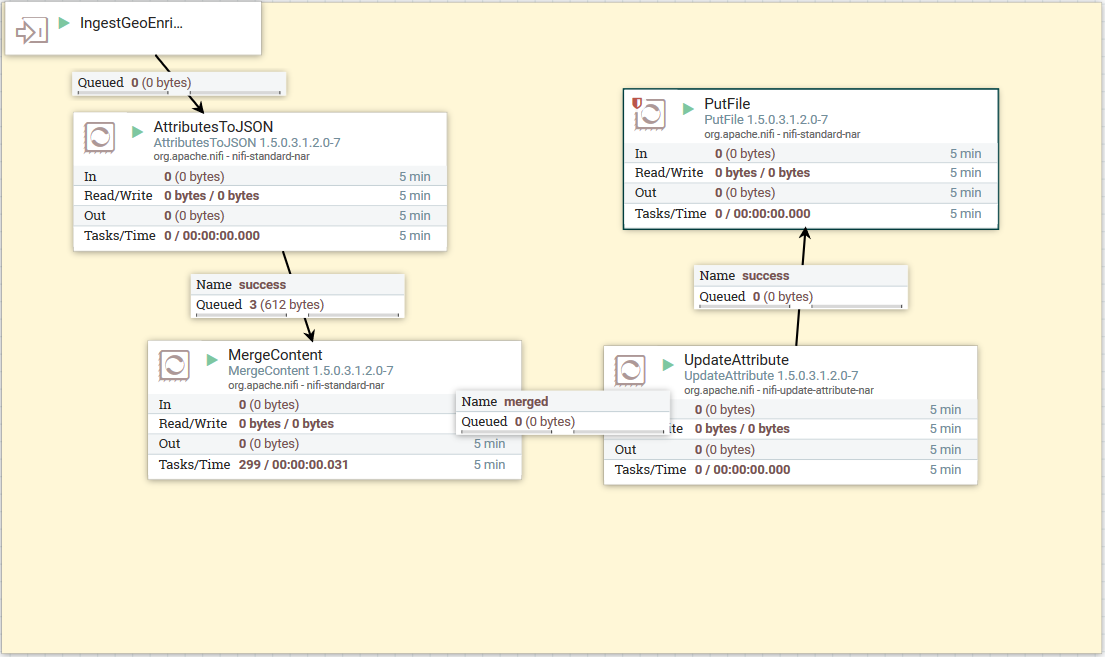


Setting for UpdateAttribute to set a unique name for the file

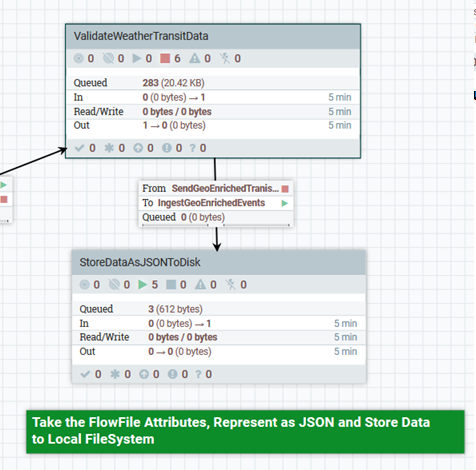
1. The last processor is PutFile to Store Data to Local File System. In this processor we change the Directory property only we set it to /tmp/out



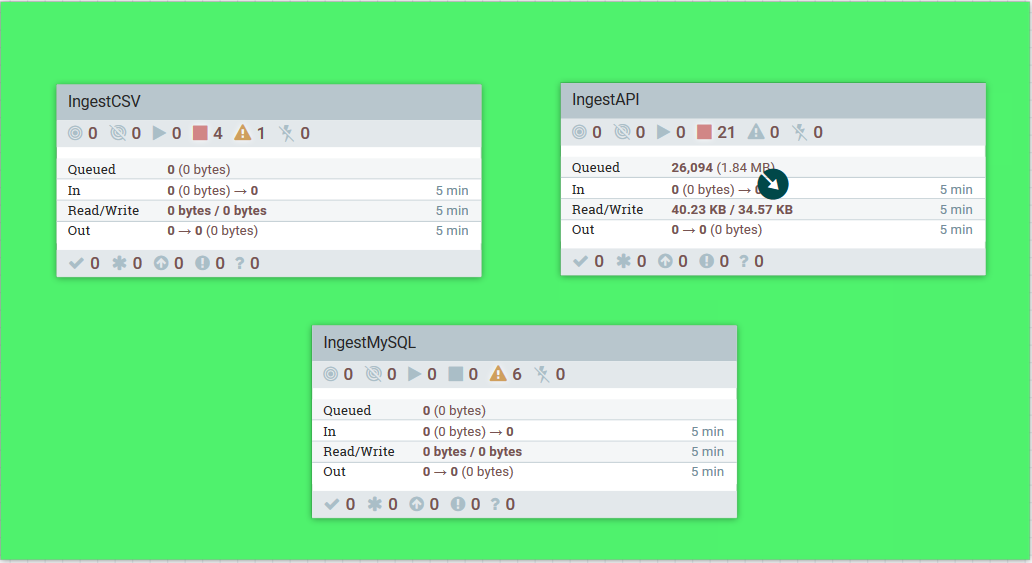
Settings for PutFile Processor to store data into the local file system; it the last four processors will look as shown in figure ??.



The connection between the processor of StoreDataAsJSONToDisk processor group



Show the connection between ValidateWeatherTransit PG and StoreDataAsJSONToDisk PG



The complete flow with the processor to ingest PAE data to the main repository

1. Implementing Data Storage Layer

To get a functional Data storage subsystem that serves its purpose, Apache Hadoop and Hive are installed and configured appropriately to ensure that the Data Integration subsystem is able to communicate with the data.

We use Cloudera CDH installed over Centos 7 Linux is first installed in a virtual environment to make a cluster of five machines. In order to succeed in installing the system, the following configurations are made to Centos 7 to ensure that it meets the required prerequisite.

***hostnamectl set-hostname mgt01.cloudera***

***echo "vm.swappiness = 10" >> /etc/sysctl.conf***

***sed -i 's/SELINUX=enforcing/SELINUX=disabled/' /etc/selinux/config***

***yum -y install ntp***

***systemctl enable ntpd***

***systemctl start ntpd***

***echo "echo never > /sys/kernel/mm/transparent\_hugepage/enabled " >> /etc/rc.local***

***echo "echo never > /sys/kernel/mm/transparent\_hugepage/defrag " >> /etc/rc.local***

***systemctl disable firewalld***

Cloudera CDH downloaded and installed on centos 7. then, required systems and services are installed and configured on CDH.

After the installation of the Ambari server and the Ambari agent, HDP is installed using Ambari UI, Apache Hadoop HDFS, Yarn, and Apache Hive are installed to create the directory and table needed to the storage layer.

Submitting Hive queries or MapReduce jobs to execute on zeppelin had many methods like REST API, .NET SDK, Azure CLI, PowerShell, Beeline, and the usage of Hive View.

Although “.hql” files/scripts were written containing HQL instructions that define the schema, that we can execute the use of a CLI command, we used to reproduction its content material and paste it into the Hive view window instead.

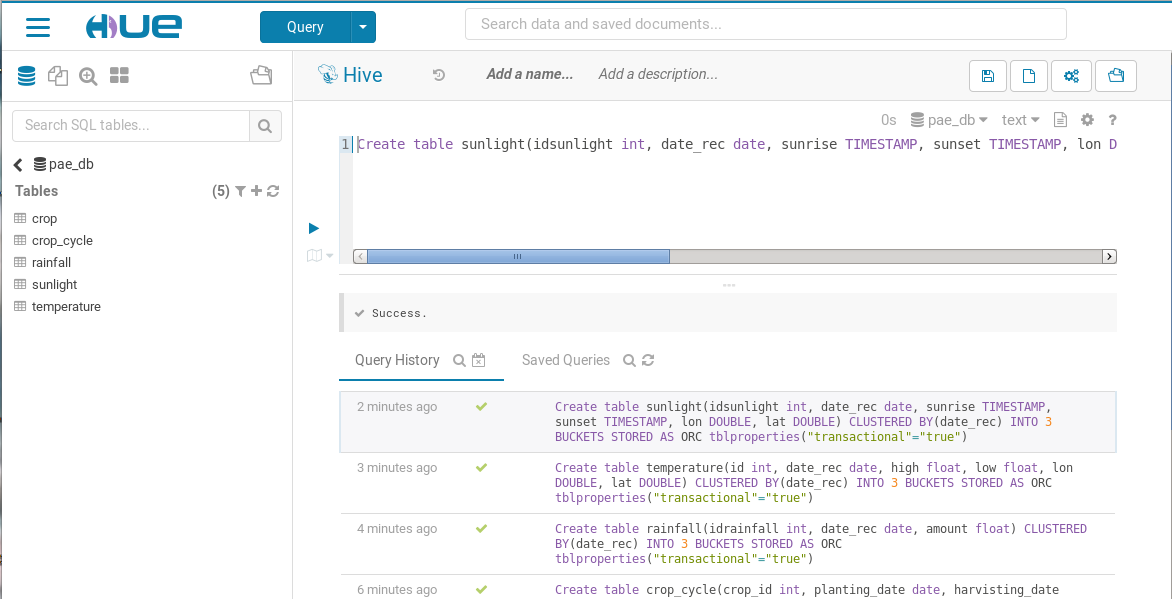
Create table sunlight(idsunlight int, date\_rec date, sunrise TIMESTAMP, sunset TIMESTAMP, lon DOUBLE, lat DOUBLE) CLUSTERED BY(date\_rec) INTO 3 BUCKETS STORED AS ORC tblproperties("transactional"="true");

Create table temperature(id int, date\_rec date, high float, low float, lon DOUBLE, lat DOUBLE) CLUSTERED BY(date\_rec) INTO 3 BUCKETS STORED AS ORC tblproperties("transactional"="true");

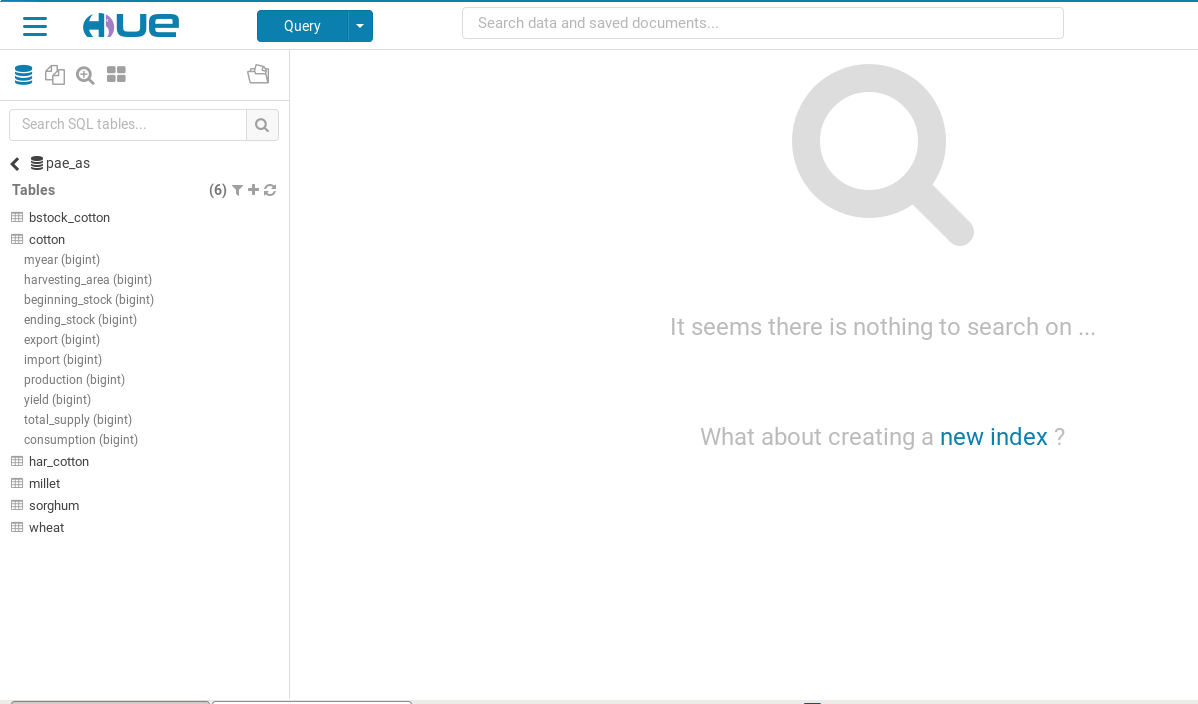
Create table rainfall(idrainfall int, date\_rec date, amount float) CLUSTERED BY(date\_rec) INTO 3 BUCKETS STORED AS ORC tblproperties("transactional"="true");

Create table crop\_cycle(crop\_id int, planting\_date date, harvisting\_date date, planting\_area float, harvisting\_area float, production float, yield float , land int) CLUSTERED BY(crop\_id) INTO 3 BUCKETS STORED AS ORC tblproperties("transactional"="true");

Create table crop(idcrop int, name varchar(255)) CLUSTERED BY(idcrop) INTO 3 BUCKETS STORED AS ORC tblproperties("transactional"="true");



Data ingested to pae\_db database Apache Hive



Data ingested to pae\_as database Apache Hive

1. Implementing Data Analysis and Visualization Layer

To ensure that the data analysis and visualization subsystem is functional, Apache Zeppelin is installed and configured on the Cloudera CDH using the Ambari UI. Interpreters for Apache Spark SQL and HIVEQL are configured on Apache Zeppelin to connect with the Data Storage System.

The Hive SQL query language (HiveQL) is used to perform the function of data selection. Hive queries are written on Apache Zeppelin to extract selected data from the main repository. The results of the data selected are also saved to the main repository. Apache Spark SQL is used to perform the data analysis.

After installation, the JDBC interpreter in Apache Zeppelin is configured to connect to the main repository in Apache Hive. With the JDBC interpreter, HIVE queries are executed to extract the information needed.

Below are the configurations for the JDBC interpreter.

|  |  |  |
| --- | --- | --- |
| **Property** | **Value** | **Description** |
| hive.driver | org.apache.hive.jdbc.HiveDriver | Specifies the class path of the jdbc driver for Hive. |
| hive.url | jdbc:hive2://localhost:10000 | url to connect to Apache Hive Warehouse. |

Table ?? Configurations on the JDBC interpreter to connect to the HIVE metastore

The Spark SQL interpreter is also configured to connect to the main repository in Apache Hive and execute SparkSQL queries. Apache Spark is also installed on Linux to ensure the possibility of running Spark SQL queries in Apache Zeppelin.

Below are the configurations for the Spark interpreter in Apache Zeppelin.

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Value** | **Description** |
| SPARK\_HOME | /usr/local/spark | The Apache Spark installation path. |
| zeppelin.interpreter.localRepo | /usr/local/zeppelin/local-repo/SYS078 | the local repository of Apache Zeppelin. |

Table 4. 2 Configurations on the Spark interpreter to connect to the HIVE metastore