# **Apache NiFi Deployment Documentation**

## **1. Introduction**

This document serves as a comprehensive guide to setting up an Apache NiFi-based ELT (Extract, Load, Transform) pipeline using version 1.28.1. It includes installation and environment setup issues, detailed processor configurations, pipeline structure, common errors encountered, and their respective resolutions. The goal is to serve as documentation for the Data Warehouse Implementation to enable team members to debug, understand, replicate, and troubleshoot both ETL & ELT pipelines setup in the future.

## **2. System Environment**

* **Apache NiFi Version**: 1.28.1 (stable version as NiFi 2.4.0 had bugs during the time of implementation)
* **Java Version**: 17 (Java 21 was incompatible with NiFi 1.28.1)
* **Database**: PostgreSQL
* **Operating System**: Linux (Ubuntu)
* Apache NiFi can be deployed in two primary configurations:
* Standalone Mode (Single Node) – Runs on a single machine, suitable for simple or small-scale pipelines. {our current implementation for tests}
* Cluster Mode (Multi-Node) – Distributes workloads across multiple nodes, including ***master node(s)*** for coordination and***slave node(s)*** for execution. This setup enhances ***fault tolerance*** and ***processing efficiency*** for larger or more complex workflows.
* The choice between these architectures depends on the ***pipeline complexity*** and ***performance requirements***. For lightweight tasks, a standalone installation is sufficient, while demanding workloads benefit from a clustered deployment.
* Additionally,***disk capacity*** is a critical consideration since running pipelines consume storage for data buffering and processing.

## **3. NiFi Installation and Setup**

### **3.1 Overview of Apache Nifi**

Nifi is a flow automation tool, like Apache Airflow. But it was built to work via GUI instead of programming.

**Understanding “FlowFiles” and “Processors”**

**FlowFiles** are objects in **NiFi** that represent data. They consist of:

* **Content** (the actual data payload)
* **Attributes** (metadata describing the data)

It is important to distinguish between these two because all operations in NiFi work on either the **content** or the **attributes**.

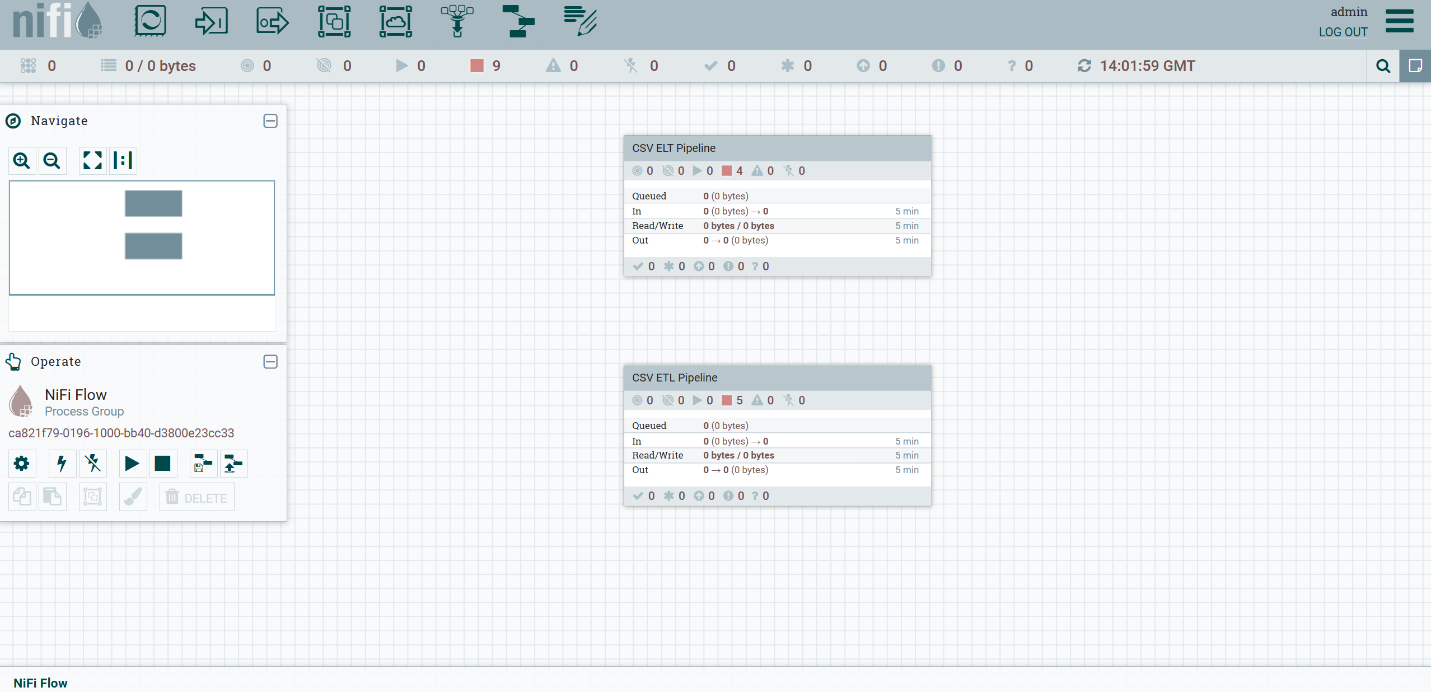
In our test for instance, the “content” will be the rows of data, and we will use the **filename** “attribute” to filter and route different datasets to different processors.

**Processors** are pieces of code that perform operations or transformations on FlowFiles. Think of processors as reusable functions—they receive FlowFiles, process them, and then pass them to other processors.

### **3.2 Initial Installation Issue with Java 21**

* **Problem**: After installing Java 21 and attempting to start NiFi 1.28.1, the application failed to run.
* **Cause**: NiFi 1.28.1 is not compatible with Java 21.
* **Resolution**: Uninstalled Java 21 and installed Java 17 instead.
  + Verified Java version using java -version.
  + Restarted NiFi successfully after setting JAVA\_HOME to the Java 17 directory.
* Note: Apache Nifi requires Java to run, thus if not already installed on your server, install Java and all required packages.
* Steps into setting up NIFI version 1.28.1 and Java 17 are all included in the reference section at the end of the document. Point 10 “***Appendix and References***” (same steps can be followed to set up newer versions of both NiFi and Java)

## **4. Interface Navigation Notes**



* To open a processor configuration, double-click its icon.
* Connect processors by clicking the arrow icon on one and drag to the next processor.
* Use the right-click context menu to start/stop/enable processors.
* Use the data provenance tab to trace data flow and inspect payloads.
* To create a “Process Group,” In NiFi UI, right-click on the canvas and Select "Create Process Group"
* To Add Required Processors, drag them from the top palette and drop them into your process group on your canvas

## **5. Overview of the ELT Pipeline**

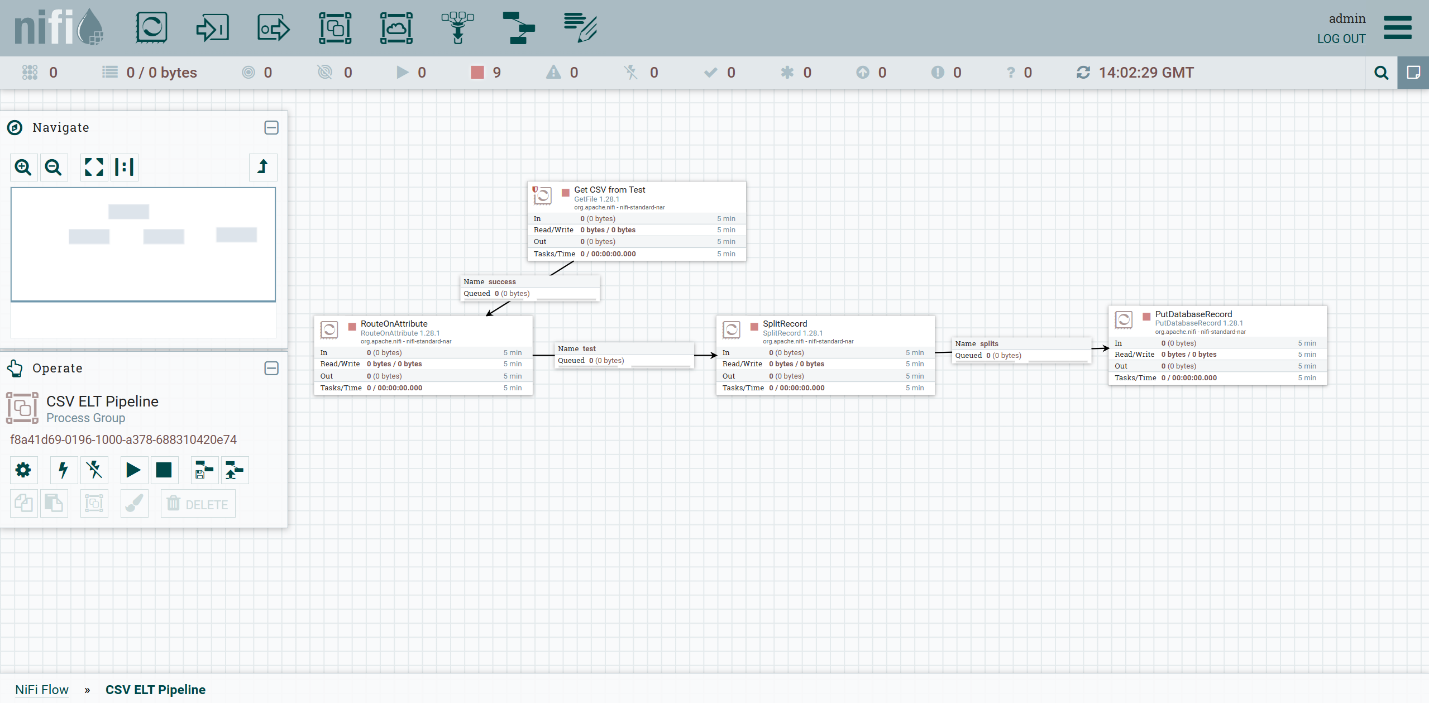
### **5.1 Pipeline Type**

* **ELT (Extract, Load, Transform)**: Data is extracted from a CSV file, loaded into PostgreSQL as-is, with no transformation occurring before the load step.

**5.2 Objective**

1. Create a sample CSV file with mock data.
2. Create a matching PostgreSQL table.
3. Build a NiFi ELT pipeline to:

* Extract the CSV file.
* Split the CSV file.
* Load it into the database.



**Create Sample CSV File**

Created a file named test.csv in a folder on my system (e.g., /home/username/nifi-data/in/).

**test.csv content:**

staff\_id,first\_name,last\_name,age

210,Kofi,Bamfo,20

212,Ama,Adu,21

214,Kojo,Adjei,24

216,Abena,Tetteh,23

218,Kwame,Gyamfi,24

**Create PostgreSQL Table**

CREATE TABLE data.test (

staff\_id INTEGER PRIMARY KEY,

first\_name TEXT(50) NOT NULL,

last\_name TEXT(50) NOT NULL,

age INTEGER NOT NULL

);

This matches what the output will be.

**Step 3: Build ELT Flow in NiFi**

**Final Processor Flow:**

GetFile → RouteOnAttribute → SplitRecord → PutDatabaseRecord

**5.3 Explanation of Each Processor**

***GetFile***

* **Purpose**: Extracts the CSV file from a local directory.

***RouteOnAttribute***

* **Purpose**: Directs flowfiles based on attributes (e.g., filename pattern).

***SplitRecord***

* **Purpose**: Splits a CSV file into individual records.

***PutDatabaseRecord***

* **Purpose**: Inserts records into a PostgreSQL table.

Created a **new Process Group** in NiFi and double-clicked to enter it.

Configuring each Processor;

**1. GetFile**

**Purpose**: Pulls the CSV file from the local directory.

***Steps:***

1. Search for "GetFile" in the processor palette and drag it to the canvas
2. Double-click it to configure
3. In the Properties tab configure:
4. **Input Directory**: The folder where the csv file(s) is located (e.g., /home/username/nifi-data/in)
5. **Keep Source File**: false (optional, moves files after processing) (if set to “true” the file will be re-processed over and over, which is only ideal when running tests)
6. **File Filter**: .\*\.csv (or your specific filename)
7. **Polling Interval**: 10 sec
8. In the Scheduling tab configure:
9. **Run Schedule**: 0 sec (continuous) (or your preferred interval):
10. **Concurrent Tasks**: 1
11. Click **Apply** to save the configuration.

**2. RouteOnAttribute**

**Purpose**: Filters non-CSV files.

***Steps:***

1. Drag “RouteOnAttribute” processor.
2. Connect “GetFile” → “RouteOnAttribute.”
3. Double-click it:
4. Go to **Properties**.
5. Add a new property:
6. **Name**: “test”
7. **Value**: ${filename:equals('test.csv')}
8. Go to **Settings → Automatically Terminate Relationships**:
9. Check all **except** “test”’.
10. Click **Apply** to save the configuration.

**3. SplitRecord**

**Purpose**: Converts the CSV into one JSON record per row.

***Steps:***

1. Drag SplitRecord processor.
2. Connect “RouteOnAttribute” → SplitRecord using “test” relationship.
3. Configure:
4. **Record Reader**: Click ➕ to create a new “CSVReader”
5. CSVReader Controller Configuration
6. Search for CSVReader → Add
7. Properties

* **Schema Access Strategy**: "Use String Fields from Header"
* **First Line is Header**: "true" # This handles header skipping
* **CSV Format**: Custom Format
* **Character Set**: "UTF-8"
* **Delimiter Character:** ","
* **Quote Character**: '"
* **Escape Character**: '\'
* **Trim Fields**: "true"
* **Schema Access Strategy**: Infer Schema
* **Enable the Service**

1. **Record Writer**: Click ➕ to create “CSVRecordSetWriter”
2. CSVRecordSetWriter Controller Configuration
3. Search for CSVRecordSetWriter → Add
4. Properties

* **Schema Write Strategy**: Do Not Write Schema

1. Click **Apply** to save the configuration.

**4. PutDatabaseRecord**

**Purpose**: Loads transformed data into PostgreSQL.

***Steps:***

1. Drag “PutDatabaseRecord” processor.
2. Connect “QueryRecord” → “PutDatabaseRecord.”
3. Configure:
4. **Record Reader**: “CSVReader” (same as above)
5. **Database Connection Pooling Service**:
6. Click ➕ to create the “DBCPConnectionPool”
7. Configure:

**Database Connection URL**: jdbc:postgresql://localhost:5432/your\_database

**Driver Class Name**: org.postgresql.Driver

**Driver Location(s)**: Full path to the PostgreSQL JDBC driver .jar

**Database Username** and **Password**

**Table Name**: your\_table

**Schema Name**: schema\_name

**Statement Type**: INSERT

**Schema Access Strategy**: Infer Schema

**Enable the service.**

1. Click **Apply** to save the configuration.

**5.4 Connect the Processors**

Connect the processors in this order with success relationships:

GetFile → RouteOnAttribute

RouteOnAttribute → SplitRecord

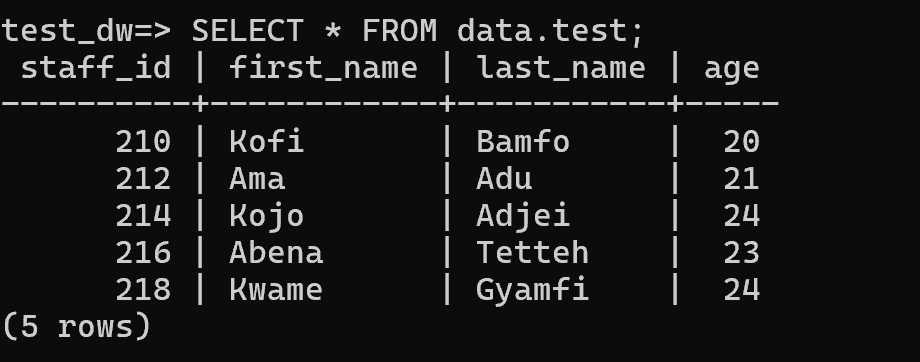
SplitRecord → PutDatabaseRecord

**Start the Processors**

* Right-click inside the canvas → Select "Start"
* Watch the flow by clicking on the button at the top-right corner of your NiFi Dashboard and in the drop-down select "Data Provenance".

**5.5 Success Output**

Check PostgreSQL to confirm that transformed rows are inserted.



**5.6 Troubleshooting**

**If files are not being picked up:**

* Check /your\_directory/ permissions
* Verify NiFi has read access (try sudo chmod -R 755 /datasources)
* If PostgreSQL connection fails:
* Verify your JDBC URL includes the correct port/database
* Check the PostgreSQL driver is in the correct location

## **6. Errors Encountered and Resolutions**

### **6.1 Validation Warning in NiFi**

* Unconfigured relationships ('failure' and 'original') in a NiFi workflow.
* Hovered over the invalid badge on a processor in the initial process flow and what I saw was " 'Relationship Failure' is invalid because Relationship 'failure' is not connected to any component and is not auto-terminated" and also " 'Relationship original' is invalid because Relationship 'original' is not connected to any component and is not auto-terminated"
* This happened because NiFi requires all relationships to be either: Connected to another processor or auto terminated(acknowledged/dropped)

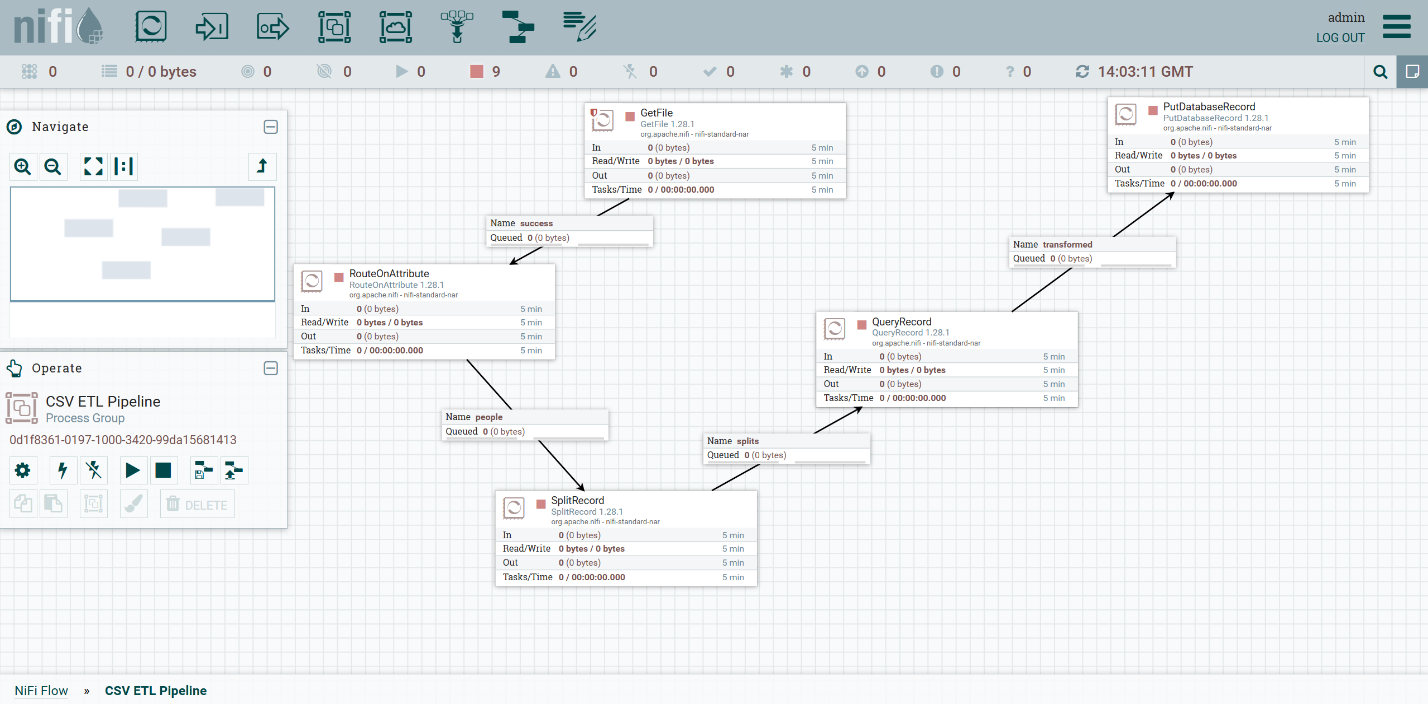
### **6.2 PutDatabaseRecord Errors**

* I had not yet added the PostgreSQL JDBC driver; thus, I downloaded it from <https://jdbc.postgresql.org/download.html>, and placed it somewhere accessible (e.g., /opt/nifi/drivers/postgresql.jar), then referenced that path in Database Driver Location(s).
* Database Connection Failure due to incorrect PostgreSQL configuration (localhost:5500 unreachable) default port was 5432 being used, configured the connection URL and updated it to :5432.

## **7. ETL Pipeline: Minimal Working Setup**

## **7.1 Objective**

1. Create a sample CSV file with mock data.
2. Create a matching PostgreSQL table.
3. Build a NiFi ETL pipeline to:
   1. Extract the CSV file.
   2. Transform data (e.g., uppercase names, filter adults).
   3. Load it into the database.



## **7.2 Create Sample CSV File**

Create a file named people.csv in a folder on your system (e.g., /home/username/nifi-data/in/).

### **people.csv content:**

id,name,age,email  
[1,Alice,25,alice@example.com](mailto:1,Alice,25,alice@example.com)[2,Bob,17,bob@example.com](mailto:2,Bob,17,bob@example.com)[3,Charlie,30,charlie@example.com](mailto:3,Charlie,30,charlie@example.com)[4,Diana,15,diana@example.com](mailto:4,Diana,15,diana@example.com)

## **7.3 Create PostgreSQL Table**

Log into PostgreSQL (e.g., using psql) and run:

CREATE TABLE your\_table (  
 id INT,  
 full\_name TEXT,  
 age INT,  
 email TEXT,  
 category TEXT  
);

This matches what the output will be after transforming the CSV data.

## **7.4 Build ETL Flow in NiFi**

### **Final Processor Flow:**

GetFile → RouteOnAttribute → SplitRecord → QueryRecord→ PutDatabaseRecord

#### **GetFile**

* **Purpose**: Extracts the source CSV file.

#### **RouteOnAttribute**

* **Purpose**: Filters flowfiles to allow only CSV files.

#### **SplitRecord**

* **Purpose**: Splits the CSV into individual records for transformation.

#### **QueryRecord**

* **Purpose**: Transforms the data using SQL-like queries (e.g., column renaming, filtering).

#### **PutDatabaseRecord**

* **Purpose**: Loads the transformed records into a PostgreSQL table.

Create a **new Process Group** in NiFi and double-click to enter it.

Now, follow these steps:

**1. GetFile**

**Purpose**: Pulls the CSV file from the local directory.

***Steps:***

1. Search for "GetFile" in the processor palette and drag it to the canvas
2. Double-click it to configure
3. In the Properties tab configure:
4. **Input Directory**: The folder where the csv file(s) is located (e.g., /home/username/nifi-data/in)
5. **Keep Source File**: false (optional, moves files after processing) (if set to “true” the file will be re-processed over and over, which is only ideal when running tests)
6. **File Filter**: .\*\.csv (or your specific filename)
7. **Polling Interval**: 10 sec
8. In the Scheduling tab configure:
9. **Run Schedule**: 0 sec (continuous) (or your preferred interval)
10. **Concurrent Tasks**: 1
11. Click **Apply** to save the configuration.

### **2. RouteOnAttribute**

**Purpose**: Filters non-CSV files.

#### **Steps:**

1. Drag “RouteOnAttribute” processor.
2. Connect “GetFile” → “RouteOnAttribute.”
3. Double-click it:
   1. Go to **Properties**.
   2. Add a new property:
      1. **Name**: “people”
      2. **Value**: ${filename:endsWith('.csv')}
4. Go to **Settings → Automatically Terminate Relationships**:
   1. Check all **except** “people.”
5. Click **Apply** to save the configuration.

### **3. SplitRecord**

**Purpose**: Converts the CSV into one JSON record per row.

#### **Steps:**

1. Drag SplitRecord processor.
2. Connect “RouteOnAttribute” → SplitRecord using “the ‘Name’ you set in B in the previous step” relationship.
3. Configure:
4. **Record Reader**: Click ➕ to create a new “CSVReader”

* CSVReader Controller Configuration
* Search for CSVReader → Add
* Properties
* **Schema Access Strategy**: "Use String Fields from Header"
* **First Line is Header**: "true" # This handles header skipping
* **CSV Format**: Custom Format
* **Character Set**: "UTF-8"
* **Delimiter Character:** ","
* **Quote Character**: '"
* **Escape Character**: '\'
* **Trim Fields**: "true"
* **Schema Access Strategy**: Infer Schema
* **Enable the Service**

1. **Record Writer**: Click ➕ to create “JsonRecordSetWriter”
   * 1. **Schema Write Strategy**: Do Not Write Schema
2. Click **Apply** to save the configuration.

### **4. QueryRecord**

**Purpose**: Transforms data using SQL.

**Steps:**

1. **Add the Processor**:
   1. Drag the QueryRecord processor onto the canvas.
   2. Connect it to the previous processor (SplitRecord).
2. **Configure Record Reader and Writer**:
   1. **Record Reader**:
      1. Click the gear icon to configure it.
      2. Choose “JsonTreeReader.”
      3. Set **Schema Access Strategy** to “Infer Schema.”
   2. **Record Writer**:
      1. Choose “JsonRecordSetWriter.”
      2. Set **Schema Write Strategy** to “Do Not Write Schema.”
3. **Define the SQL Query**:
   1. In the **Properties** tab, click the ➕ icon to add a new property.
   2. **Property Name**: transformed
   3. **Property Value**:

SELECT  
 id,  
 UPPER(name) AS full\_name,  
 age,  
 email,  
 CASE WHEN age > 18 THEN 'adult' ELSE 'minor' END AS category  
FROM FLOWFILE

1. **Set Relationships**:
   1. In the **Settings** tab, under **Automatically Terminate Relationships**, check all except transformed and failure.
2. **Connect to the Next Processor**:
   1. Connect the transformed relationship from QueryRecord to the PutDatabaseRecord processor.
3. Click **Apply** to save the configuration.

### **5. PutDatabaseRecord**

**Purpose**: Loads transformed data into PostgreSQL.

***Steps:***

1. Drag “PutDatabaseRecord” processor.
2. Connect “QueryRecord” → “PutDatabaseRecord.”
3. Configure:

* **Record Reader**: “JsonTreeReader” (same as above)
* **Database Connection Pooling Service**:
* Click ➕ to create the “DBCPConnectionPool”
* Configure:
* **Database Connection URL**: jdbc:postgresql://localhost:5432/your\_database
* **Driver Class Name**: org.postgresql.Driver
* **Driver Location(s)**: Full path to the PostgreSQL JDBC driver .jar
* **Database Username** and **Password**
* **Table Name**: your\_table
* **Schema Name**: schema\_name
* **Statement Type**: INSERT
* **Schema Access Strategy**: Infer Schema

1. **Enable the service.**
2. Click **Apply** to save the configuration.

**7.5 Connect the Processors**

Connect the processors in this order with success relationships:

GetFile → RouteOnAttribute

RouteOnAttribute → SplitRecord

SplitRecord → QueryRecord

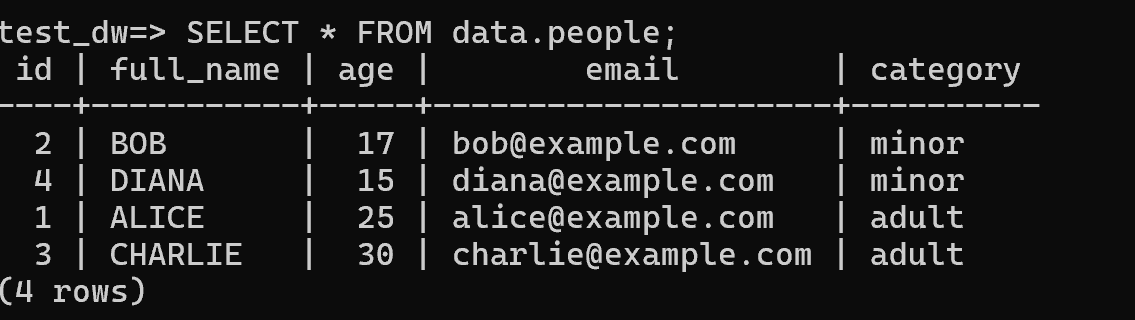
QueryRecord → PutDatabaseRecord

**Start the Processors**

* Right-click inside the canvas → Select "Start"
* Watch the flow by clicking on the button at the top-right corner of your NiFi Dashboard and in the drop-down select "Data Provenance".
* Monitor for errors using bulletins and data provenance.

**7.6 Success Output**

Check PostgreSQL to confirm that transformed rows are inserted.



## **8. API JSON to CSV File (ETL Pattern)**

* **Flow Name:** API JSON Ingestion
* **Flow Type:** ETL (Extract → Transform → Load)

**Objective**

To demonstrate a successful end-to-end data ingestion pipeline that:

* Pulls JSON data from a public API
* Extracts desired fields using EvaluateJsonPath
* Transforms JSON records into CSV format
* Writes the final CSV data to a local directory for downstream usage

### Summary

This NiFi flow demonstrates successful data ingestion from a dynamic public JSON API, transforming and converting it into structured CSV files. It is intended to simulate external API ingestion in a real-time data pipeline.

## **Flow Overview**

|  |  |  |
| --- | --- | --- |
| Step | Processor | Description |
| 1 | InvokeHTTP (Fetch\_JSON\_Data) | Pulls JSON data from a public REST API |
| 2 | SplitJson | Splits JSON array into individual record objects |
| 3 | EvaluateJsonPath | Extracts specific fields from each JSON object |
| 4 | AttributesToCSV | Converts flowfile attributes into CSV-formatted data |
| 5 | UpdateAttribute | Updates filename with dynamic timestamp |
| 6 | PutFile | Writes the CSV content to a local directory |
| 7 | LogAttribute | Logs flowfile metadata on failure or debug paths |

##### **Processor Configuration Details**

🔹 **InvokeHTTP (Fetch\_JSON\_Data)**

* HTTP Method: GET
* Remote URL: https://api.somethinggoeshere.com/qwe/asd/users.json(JSON endpoint)
* Send Body: false
* Follow Redirects: true
* Output Response Regardless: true
* Add Response Headers: false
* SSL Context Service: (none if not HTTPS-secured with cert)

🔹 **SplitJson**

* JsonPath Expression: “$[\*]”(depends on API response structure)

🔹 **EvaluateJsonPath**

* Destination: flowfile-attribute
* Return Types: Auto-detect (or you can specify, in our case “json”)
* Leave other properties with the default setting, and since we will be picking user data and not all user data, extract specific fields from each json object.(In the properties tab click on the + and include these field expressions based on the fields you want to extract) E.g., refer below
* Expressions:
  + city → $.address.city
  + email → $.email
  + name → $.name

🔹 **AttributesToCSV**

* Attributes List: name, email, city (based on the extracted fields from the “EvaluateJsonPath” processor i.e., the attributes you set “city → $.address.city”)
* Destination: flowflie-content
* Include Schema: false
* Include Core Attributes: false

🔹 **UpdateAttribute**

* Leave other fields with default settings.
* Adds dynamic filename: (add a new property and set;)
  + filename = user\_${id}.csv

🔹 **PutFile**

* Directory: /pathtodirectory/
* Conflict Resolution Strategy: replace
* Create Missing Directories: true

🔹 **LogAttribute**

* Used on failure path to capture and log flowfile metadata and error context

#### **Controller Services**

No specific Controller service was used in this implementation but here are some optional services to consider for fine-tuning;

* StandardSSLContextService (Optional)

- Truststore/Keystore File: Not configured (used default settings)

- Enabled: No, as the API endpoint was unsecured HTTP/HTTPS without cert validation

* JsonPathReader (for downstream use)
* Not used in this flow but documented for schema-aware variants

## **Errors Encountered During Implementation**

|  |  |
| --- | --- |
| Issue | Resolution |
| SplitJson produced no records | Incorrect path like `$[] ` used; fixed by adjusting to `$[\*] ` based on response structure |
| FlowFile empty after AttributesToCSV | Attribute names did not match EvaluateJsonPath keys (case-sensitive). Confirmed names: `name`, `email`, `city` |
| Unclear output after PutFile | Default filename overwritten in UpdateAttribute, resolved by appending .csv file with filename with ` user\_${id}.csv ` |
| Incomplete API response | Confirmed NiFi only handles body on `Output Response Regardless = true` |
| Blank CSV content | Processor was writing attributes before they were added — reordered processors: EvaluateJsonPath → AttributesToCSV |

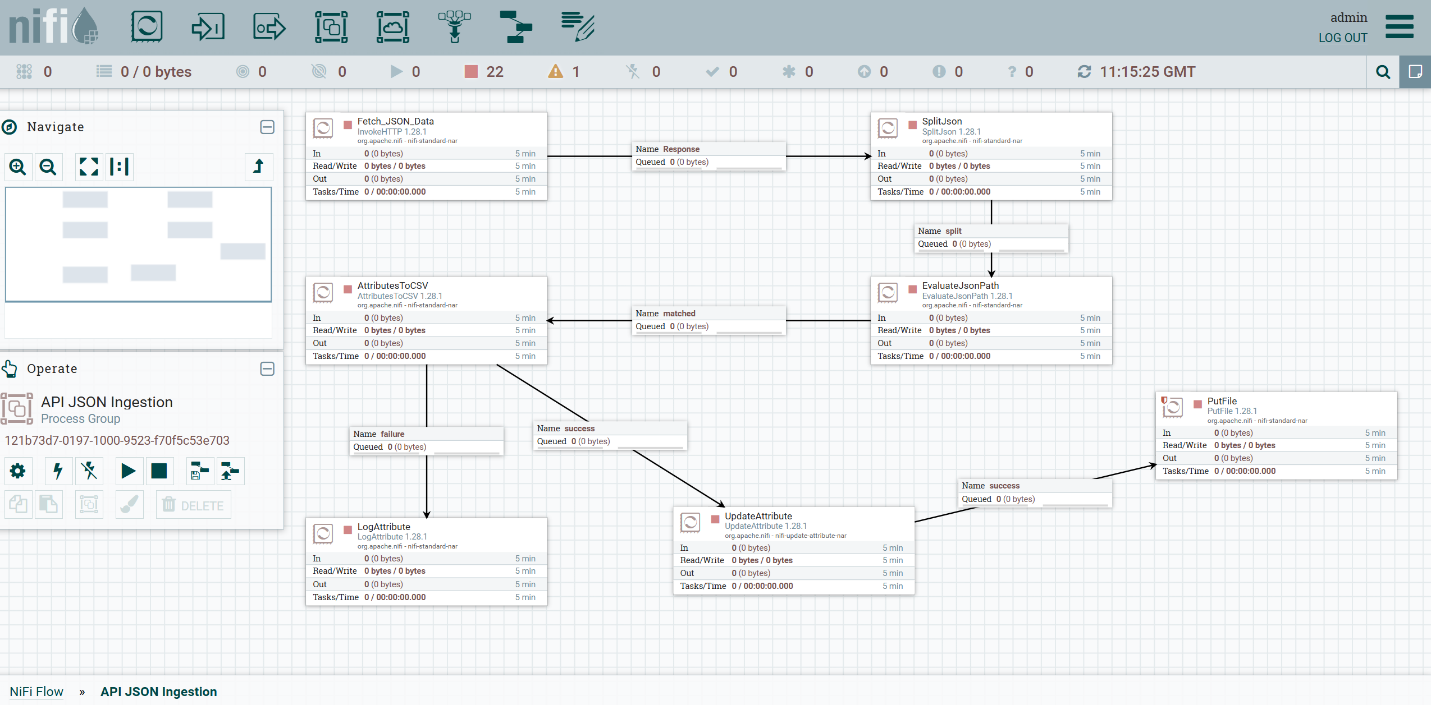
**Testing Notes**

* Ran successfully every 5 minutes on schedule.
* Confirmed individual JSON records were parsed and written as CSV rows.
* Output files verified in the target directory.
* Failures (e.g., API downtime) routed to LogAttribute for troubleshooting.

**Outcome**

CSV files are successfully generated from JSON API data and stored locally.

##### **Final Data Flow**



## **Access & Permissions**

Ensure the output directory (`/pathtodirectory/`) is writable by the NiFi service user:  
  
 sudo mkdir -p /nifi\_output/  
 sudo chown nifi:nifi /nifi\_output/  
 ```

**Next Steps**

* Add timestamp field (ingested\_at) in the CSV for lineage tracking
* Add PutDatabaseRecord (optional) to load parsed data into PostgreSQL
* Add monitoring via Prometheus/Grafana
* Set up error flow notification (e.g., email/slack) on API failure

## **9. MySQL to PostgreSQL NiFi Flow**

**Flow Name:** MySQL to PostgreSQL Ingestion

**Flow Type:** ETL (Extract → Transform → Load)

## **Objective**

To demonstrate a successful database-to-database ingestion pipeline that:

* Extracts records from a MySQL source database
* Converts the data to JSON for transformation
* Converts to SQL INSERT statements
* Inserts the records into a PostgreSQL target database

## **Summary**

This NiFi flow demonstrates a complete data pipeline between two relational databases. The flow extracts data from a MySQL database, transforms it into JSON, and then inserts it into a PostgreSQL database. It is designed to test database integration capabilities for structured system migration or synchronization scenarios.

## **Flow Overview**

|  |  |  |
| --- | --- | --- |
| **Step** | **Processor** | **Description** |
| 1 | ExecuteSQL | Extracts data from MySQL using a SQL query |
| 2 | ConvertAvroToJSON | Converts Avro output to JSON format |
| 3 | SplitJson | Splits JSON array into individual records |
| 4 | ConvertJSONToSQL | Converts JSON records into SQL INSERT statements |
| 5 | PutSQL | Executes SQL inserts into PostgreSQL |
| 6 | LogAttribute | Logs flowfile attributes for debugging |

## **Processor Configuration Details**

* **ExecuteSQL**
  + **Purpose**: Run SQL query against MySQL database.
  + **SQL Query**: SELECT \* FROM my\_table
  + **Output**: Avro
  + **Controller Service**: DBCPConnectionPool (MySQL)
* **ConvertAvroToJSON**  
  + Converts the Avro output from ExecuteSQL into JSON format.
  + Required for easier handling of dynamic schema and transformation.
* **SplitJson**  
  + Splits JSON array into individual records for granular insertion.
* **ConvertJSONToSQL**  
  + Converts each JSON record into a SQL INSERT statement.
  + **Statement Type**: INSERT
  + **Database Type**: PostgreSQL
  + **Table Name**: Matches target PostgreSQL table.
* **PutSQL**  
  + Executes generated SQL statements into PostgreSQL.
  + **Controller Service**: DBCPConnectionPool (PostgreSQL)
* **LogAttribute**

Logs metadata for debugging or auditing purposes.

## **Controller Service Configurations**

### **1. MySQL DBCPConnectionPool**

* **Database Connection URL**: jdbc:mysql://localhost:3306/your\_database
* **Database Driver Class Name**: com.mysql.cj.jdbc.Driver
* **Database Driver Location(s)**: Path to your mysql-connector-java-X.X.X.jar
* **Database User**: your\_username
* **Password**: your\_password

### **2. PostgreSQL DBCPConnectionPool**

* **Database Connection URL**: jdbc:postgresql://localhost:5432/your\_target\_db
* **Database Driver Class Name**: org.postgresql.Driver
* **Database Driver Location(s):** Path to your postgresql-XX.X.jar
* **Database User**: your\_pg\_user
* **Password**: your\_pg\_password

## **Errors Encountered During Implementation**

|  |  |
| --- | --- |
| **Issue** | **Resolution** |
| Missing JDBC Driver | NiFi could not find the MySQL driver; fixed by downloading mysql-connector-java-X.X.X.jar and placing it in NiFi’s lib directory, then restarting NiFi. |
| SplitJson not outputting records | The JSON structure was flat, not an array, adjusted input/output format based on the query output. |
| Schema mismatch between MySQL and PostgreSQL | Fields did not align (e.g., case differences; column name changes). Verified both schemas and adjusted ConvertJSONToSQL table and mapping logic. |
| FlowFile content was null | Query returned no rows; confirmed the table had data and the query was accurate. |
| SQL Insert failed due to data types | Data type mismatch, updated PostgreSQL table schema |
| Avro to JSON conversion failed | Error: Not a data file. Ensured upstream processor (ExecuteSQL) output format was Avro, and no intermediary processor altered the FlowFile. |
| SQL Insert failed silently | Converted JSON contained null or malformed fields. Added schema validation before ConvertJSONToSQL. |

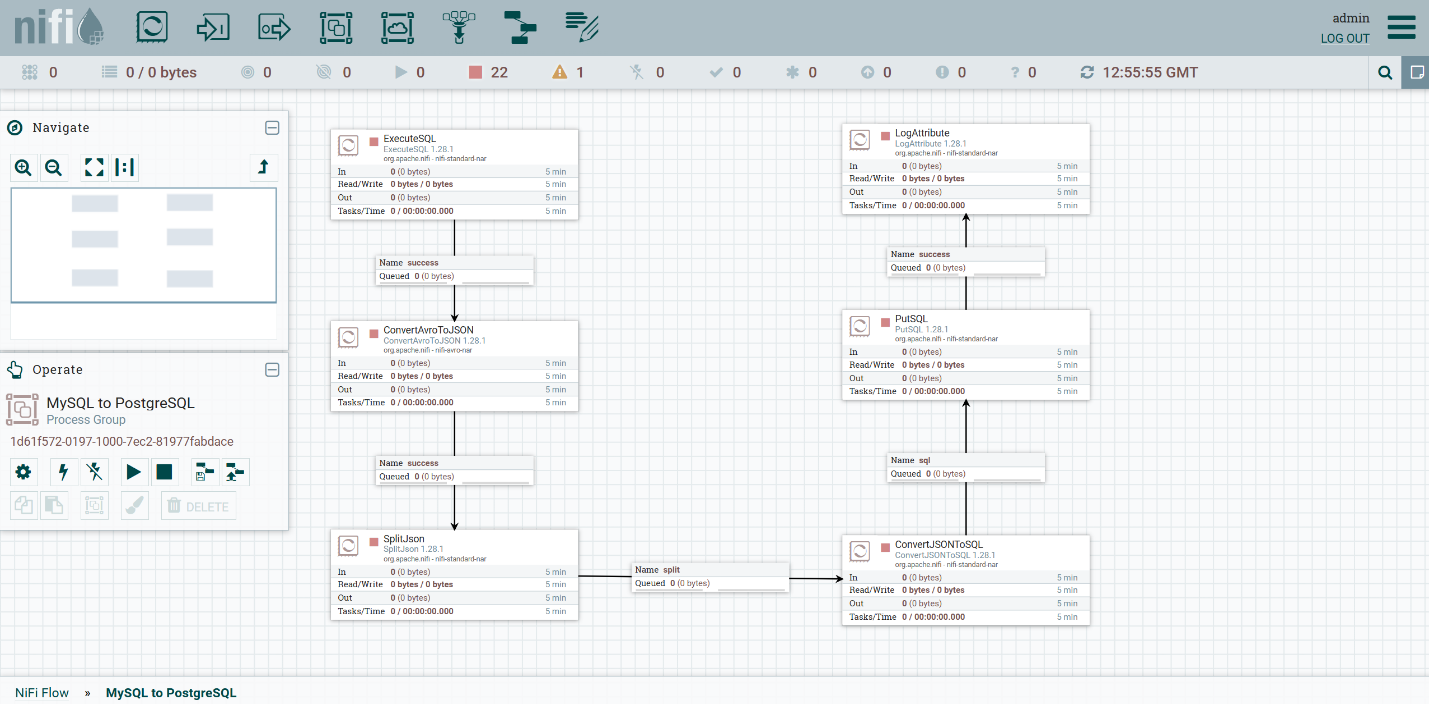
## **Testing Notes**

* Ran flow manually and on schedule using ExecuteSQL processor triggered every 10 minutes.
* Confirmed rows were extracted from MySQL and inserted into PostgreSQL.
* Verified row count using SQL queries in both databases.
* Used LogAttribute for debugging content.

## **Outcome**

Structured data successfully migrated from MySQL to PostgreSQL. The flow handles type conversion, schema matching, and connection reliability, and includes validation paths for future production hardening.

**Final Data Flow**



**Notes & Best Practices**

* Always test query output with LogAttribute or UpdateAttribute to verify flowfile content.
* Keep JDBC drivers in a consistent, accessible location for controller services.
* Use SplitJson if the JSON array is large or to maintain a 1:1 mapping between records and inserts.
* If column mapping issues arise, consider using PutDatabaseRecord instead of PutSQL.

## **Ready for Production?**

This flow meets the requirements for staging deployment.  
Before moving to production:

* Enable controller service protection policies.
* Set up error queues and alerts with email or Prometheus/Grafana.
* Secure DB credentials using NiFi’s parameter context or vault integration.

## **10. Parse Apache log JSON data and insert it into PostgreSQL**

**Flow Name:**

Ingest Apache Access Logs

**Flow Type:**

ETL (Extract → Transform → Load)

**Objective:**

To ingest, parse, enrich, and store Apache HTTP server access logs in PostgreSQL for analysis and monitoring. This flow:

* Extracts log lines from a live access log file
* Converts each line to structured JSON
* Parses and validates the JSON structure
* Enriches the data with an ingestion timestamp
* Inserts records into a PostgreSQL table

**Summary**

This Apache NiFi flow demonstrates ingestion of real-time access logs generated by a web server. It reads the logs using ***TailFile***, transforms raw log strings into JSON, enriches them with ingestion metadata, validates structure, and inserts into a PostgreSQL database. The flow includes error routing, logging, and is designed for ongoing ingestion in an on-premises data pipeline.

### **Flow Overview**

|  |  |  |
| --- | --- | --- |
| **Step** | **Processor** | **Description** |
| 1 | TailFile | Continuously monitors and ingests new log lines from Apache access logs |
| 2 | UpdateAttribute | Sets file-related attributes (e.g., filename, log type) |
| 3 | ReplaceText | Converts raw log lines into structured JSON format |
| 4 | RouteOnAttribute | Validates or classifies parsed JSON logs |
| 5 | UpdateRecord | Adds “ingested\_at” timestamp field to each record |
| 6 | PutDatabaseRecord | Inserts enriched JSON data into PostgreSQL using schema-aware insertion |
| 7 | LogAttribute | Logs flowfile metadata for parse failure or debugging |
| 8 | PutFile | Archives or stores failed records locally |

### **Processor Configuration Details**

#### **TailFile**

* **File to Tail**: /var/log/apache2/access.log (where your logs are located)
* **Initial Start Position**: Beginning of File or End of File (depending on goal)
* **Rolling Filename Pattern**: Optional (if logs rotate)
* **Output**: Raw string log line (1 per FlowFile)

#### **UpdateAttribute**

* Adds context-specific attributes like log.type = apache, host.name = ${hostname()}

**ReplaceText**

Converts the raw log line (e.g., Common/Combined Log Format) to structured JSON using regex or known parsing template. (in our test, regex was used)

* Replacement Strategy = Regex Replace
* Search Value = ^(\S+)\s+(\S+)\s+(\S+)\s+\[([^\]]+)\]\s+"([^"]\*)"\s+(\d{3})\s+(\d+|-)\s+"([^"]\*)"\s+"([^"]\*)"
* Replacement Value =

{

"clientip": "$1",

"ident": "$2",

"user": "$3",

"timestamp": "$4",

"request": "$5",

"response": "$6",

"bytes": "$7",

"referer": "$8",

"useragent": "$9"

}

* Character Set = UTF-8
* Maximum Buffer Size = 1 MB
* Evaluation Mode = Line-by-Line (Run the ‘Replacement Strategy’ against each line separately)
* Line-by-Line Evaluation Mode = All

#### **RouteOnAttribute**

* Checks for valid JSON structure (e.g., attribute like parse\_status = success)
* Routes success to UpdateRecord, failure to LogAttribute
* Routing Strategy = Route to Property name (Specifies how to determine which relationship to use when evaluating the Expression Language)

#### **UpdateRecord**

* Adds a new field: "/timestamp"
* **Record Reader**: JsonTreeReader (linked to inferred schema)
* **Record Writer**: JsonRecordSetWriter
* (Insert as new Property)/timestamp = ${timestamp:toDate("dd/MMM/yyyy:HH:mm:ss Z"):format("yyyy-MM-dd HH:mm:ss")}

#### **PutDatabaseRecord**

* **Database Connection Pooling Service**: DBCPConnectionPool (the pool you created)
* **Record Reader**: JsonTreeReader
* **Database Type:** PostgreSQL
* **Table Name**: data.logs (where data is the schema name)
* **Statement Type**: INSERT
* **Fail on Schema Mismatch**: true
* **Database Columns**: clientip, ident, user, timestamp, request, response, bytes, referer, useragent, ingested\_at (postgres table reference)

#### **LogAttribute**

* Captures metadata of failed FlowFiles for debugging purposes

#### **PutFile**

* Writes failed FlowFiles to a local directory: /data/nifi/failed\_logs/ (in this implementation we wrote successful flowflies to a local directory and logged all failed flowfiles)

### **Controller Service Configurations**

#### **PostgreSQL DBCPConnectionPool**

* **Database Connection URL**: jdbc:postgresql://localhost:5432/your\_db
* **Driver Class**: org.postgresql.Driver
* **Driver Location**: /pathtoyour/postgresql-42.7.1.jar
* **Database User**: Enter database user
* **Password**: \*\*\*\* (secured with Parameter Context)

#### **JsonTreeReader**

* **Schema Access Strategy**: Infer Schema

#### **JsonRecordSetWriter**

* Schema Write Strategy = Do Not Write Schema
* Schema Cache = No value set
* Schema Access Strategy = Inherit Record Schema

### **Errors Encountered During Implementation**

|  |  |
| --- | --- |
| **Issue** | **Resolution** |
| **DBCPConnectionPool not connecting to PostgreSQL** | Corrected JDBC URL and placed PostgreSQL driver in an accessible directory |
| **PutDatabaseRecord failed with null schema** | Configured JsonTreeReader with infer strategy |
| **Timestamp format mismatch** | Used UpdateRecord to ensure timestamps are ISO-8601 formatted |
| **Missing timestamp field in DB** | Added column to PostgreSQL table and ensured record schema matched |
| **Flowfile parse failure** | Routed via RouteOnAttribute → LogAttribute for inspection |
| **Unstructured log lines** | Used ReplaceText with regex to transform to JSON |

### **Testing Notes**

* Flow was tested using a sample Apache access log file.
* Inserted logs were verified via SQL queries:

SELECT COUNT(\*) FROM apache\_access\_logs;

SELECT \* FROM apache\_access\_logs ORDER BY ingested\_at DESC LIMIT 10;

* Used LogAttribute and PutFile to capture and inspect failed flowfiles.

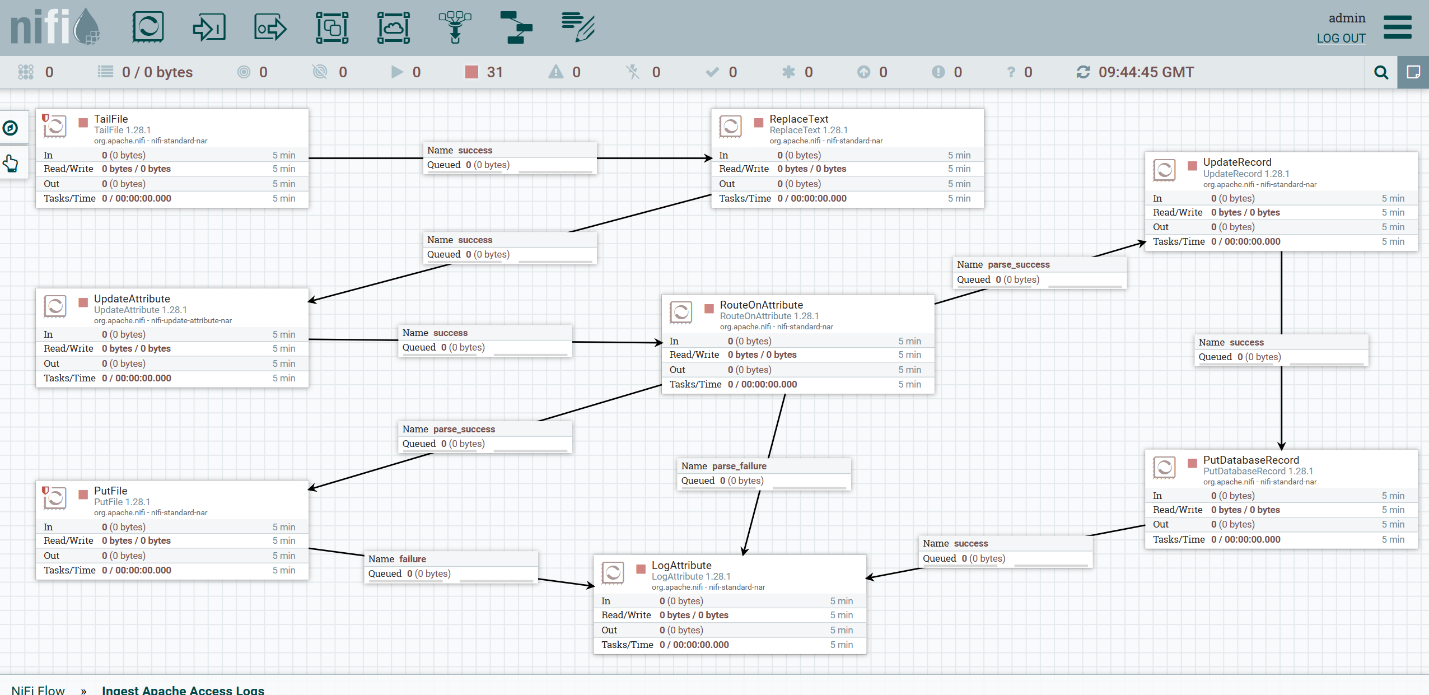
### **Outcome**

The Apache Access Log ingestion pipeline is fully functional. It handles:

* Log monitoring
* JSON transformation
* Timestamp enrichment
* Schema-aware PostgreSQL insertion
* Routing and archiving of failed records

This provides a foundational ingestion layer for operational dashboards or security audits.

**Final Data Flow**



### **Notes & Best Practices**

* Use PutDatabaseRecord with JsonTreeReader for structured insertion
* Always verify logs with LogAttribute during initial setup
* Ensure schema alignment between reader and target DB.
* Use UpdateRecord instead of manual attribute manipulation for timestamp fields.
* Secure credentials using Parameter Context.

### **Ready for Production?**

This flow is staging-ready. Before full production deployment:

* Configure alerting for failure paths
* Secure PostgreSQL access and NiFi endpoints
* Integrate with Prometheus/Grafana for health monitoring
* Enable backpressure and retry logic on database write failures

## **11. Recommendations and Best Practices**

* Always test with a small CSV sample before full-scale runs.
* Maintain naming consistency between CSV headers and DB columns.
* Validate schema early using “ValidateRecord” to avoid downstream issues.
* Avoid using hardcoded values where NiFi Expression Language can be applied.

## **12. Conclusion**

* Mastering NiFi can be complex, as understanding these concepts requires time and practice.

## **13. Appendix and References**

* [Apache NiFi Docs](https://nifi.apache.org/docs.html)
* <https://medium.com/turknettech/installation-of-apache-nifi-2856fca5bbdc>
* <https://downloads.apache.org/nifi/1.28.1/>
* <https://medium.com/@esdraslimasilva83/working-with-csv-and-nifi-febc942c7d60>
* PostgreSQL JDBC Setup: Ensure the JDBC driver is added to NiFi's lib directory
* Java 17 Compatibility: Confirmed NiFi 1.28.1 works with Java 17
* In Apache NiFi version 1.28.1, the **QueryRecord** processor does not have a dedicated "Queries" section. Instead, you define your SQL queries as **user-defined properties**, where:
* **Property Name**: Becomes a new relationship (e.g., transformed)
* **Property Value**: The SQL query to execute (e.g., SELECT ... FROM FLOWFILE)
* This design allows for multiple queries, each directing matching records to different relationships.
* Important Notes:
* ✅ Always keep success un-checked (this routes data to the next processor)
* ✅ Check failure to auto-drop files that cannot be processed
* ❌ Never auto-terminate all relationships (would discard all data)
* Why Auto-Terminate Failure?
* Prevents buildup of failed files in the connection queue
* Cleans up invalid data automatically
* Required for processors that do not route failures elsewhere
* If you do not see the failure relationship:
* Some processors only have success (check documentation)
* Ensure you are using the correct processor name (spelling matters)