



PROJECT DOCUMENTATION

Author:	innData Analytics Pvt Ltd
Creation Date:	06 th Dec 2024
Last Updated:	
Document Ref:	
Version:	v1.0

Prepared For: Aerosimple
Prepared by: Vijay K, Senior Data Engineer
06 December 2024

Confidentiality

Each party shall treat as confidential all the information obtained from the other pursuant to this Proposal and shall not divulge such information to any person (except to its own employees and other persons and only to those employees and persons who need to know the same) without the other party's prior written consent provided that this clause shall not extend to the information which was rightfully in the possession of such party prior to the commencement of the negotiations; or which is already in public knowledge or becomes so at a future date (otherwise than as a result of a breach of this clause). Each party shall ensure that its employees are aware of and comply with the provisions of this clause. The foregoing obligations as to confidentiality are applicable in perpetuity.

Document Control

Version	Name	Comments	Dated
1.0	Vijay K	Original Response Creation	06 th - DEC-2024

Project Contacts

innData Contacts

Name	Title/Role	Phone	Email
Vidya G	CEO		

AeroSimple Contacts

Name	Title/Role	Phone	Email

Contents

1. Scope of Work	5
2. NRs/CRs.....	6
3. Solution Overview	8
4. Proposed Solution In detail for each module	10
5. Data Process Workflow	12
6. Architecture	13
7. Main flow for OLAP	16
Form Answer Table Mapping.....	16
Data Approval Mapping.....	44
DataBase and Table creation	57
Data Insertion	70
Delete Operation Flow	78
8. Incremental / Manual refresh of data based on Airport Codes and Form parent ID.....	80
9. Performance Considerations in Database	82
Partitioning	82
Indexing.....	83
10. Final Deliverables.....	84

1. Scope of Work

Our objective is to devise a solution for processing and converting real-time data emanating from airports. This solution will empower the client to construct precise Key Performance Indicators (KPIs). The data, gathered through dynamic forms at airports, is currently stored in JSON format within a PostgreSQL database. Our aim is to extract, transform, and load this data into well-organized tables optimized for the purpose of generating KPI reports.

The key components of the scope include:

Initial Data Load:

For the initial data ingestion process, the complete dataset will be extracted from the source PostgreSQL database, containing dynamic form data in JSON format. This raw data will then be directly transferred to the staging area. Within the staging database, the unprocessed data will undergo a transformation to achieve a normalized structure. The transformation and loading procedures will be managed by Apache NiFi, facilitating the storage of the refined data in the designated target database.

Incremental Data Synchronization using Nifi:

Upon completion of the initial data load, the system will transition to real-time data synchronization employing Nifi. Nifi captures all modifications (inserts, updates, and deletes) from the source PostgreSQL database through logical replication. These modifications will be continuously streamed to the staging database for further processing. Subsequently, NiFi will undertake the transformation and integration of these modifications into the final KPI tables, ensuring near real-time updates.

Data Transformation and Storage:

The core of the data transformation will be handled by NiFi, which will apply business rules to flatten the JSON data, normalize it, and aggregate the required fields for KPI computation. Each airport's data will be stored in separate tables, allowing for both standardization and customization across various airport operations.

Monitoring and Alerts:

Grafana will be utilized for the purpose of monitoring the performance of ETL pipelines and triggering alerts in the event of errors or system inefficiencies. Real-time dashboards will offer insights into the operational status of data processing, encompassing aspects such as throughput, latency, and data quality assessments.

Integrating static and dynamic data files into OLAP environment, starting with a one-time data ingestion process that includes cleansing, normalizing, and validating the data for completeness and accuracy. Following this, continuous synchronization of dynamic data will be implemented using Change Data Capture (CDC) to ensure the OLAP Mobility remains up to date. Transformations will then be applied to merge static and dynamic data into OLAP tables optimized for analysis.

Testing and validation will be conducted to confirm accurate and efficient data retrieval from the OLAP tables. Finally, automated job scheduling will be established to manage ongoing data extraction, transformation, and loading processes, ensuring that dynamic data is regularly updated and integrated seamlessly with the static data.

2. NRs/CRs

SL. No	Type	Description	Date Requested	Resolution date
1	CR	Connected the approval tables' mapped data with the form answer data. Ensured that for multiple steps with the same step name, the step is selected based on the most recent completed_date column	30/08/2024	09/10/2024
2	CR	Mapped form answer data with the inspection schema using the select module key and type as the inspection checklist.	30/08/2024	09/10/2024
3	CR	Implemented dynamic database creation based on airport_id.	30/08/2024	06/9/2024
4	CR	Appended titles and descriptions for foreign keys (SRA, Hazard, Snow) in the mapped form answer data.	30/08/2024	06/9/2024
5	CR	Removed the following columns from the mapped form answer data: index_key, form_parent_id, and form_version_id.	30/08/2024	05/9/2024
6	CR	Mapped various field types (select, multiselect, multi_field) in form answer data.	13/09/2024	01/10/2024
7	CR	Mapped the response column from the hazard and SRA tables, supporting different field types (select, multiselect, multi_field).	13/09/2024	20/09/2024
8	CR	Used the version from the form answer table to match the number column in the inspections_inspection table. If no version is available, the form_date is compared against the publish_date and expiry_date in the inspections_inspection table. Applied similar logic for approvals data.	13/09/2024	15/10/2024
9	CR	Added the prefix 'd*' for repeated/duplicate titles in the form answer data, hazard/SRA responses, and approvals data.	13/09/2024	11/10/2024
10	CR	Replaced airport_id with airport_code for database creation.	14/10/2024	16/10/2024
11	CR	Replaced completed_by and created_by with full names.	14/10/2024	16/10/2024
12	CR	Removed parent_key and type:annotations from the final record.	14/10/2024	16/10/2024
13	CR	Implemented incremental loading from the source to the staging database (using CDC or table columns).	21/10/2024	14/11/2024
14	CR	Developed incremental loading logic from the staging database to the target database.	21/10/2024	14/11/2024
15	CR	Added upsert functionality for data insertion into the target database.	21/10/2024	15/11/2024
16	CR	Created indexes for newly created or altered tables using NiFi Groovy scripts.	25/11/2024	04/12/2024
17	CR	Implemented partitions for newly created tables using NiFi Groovy scripts.	25/11/2024	03/12/2024

18	CR	Added a default column is_deleted (default value false) to all tables. Managed deletions using a maintain_history_config_flag in NiFi.	25/11/2024	03/12/2024
19	CR	Enabled manual full refresh for: i. All data in one airport by code, refreshing only that specific database. ii. All data in one form for a specific airport, refreshing only that form's data.	25/11/2024	04/12/2024

3. Solution Overview

The solution for integrating and managing data within the OLAP Mobility environment involves several key steps, from continuous data synchronization and transformation of static and dynamic data in CSV and JSON formats to testing and scheduling. Below is a high-level elaboration of the proposed solution:

Continuous Data Synchronization to OLAP Mobility

Maintaining the OLAP Mobility environment synchronized with source systems by capturing and applying changes (inserts, updates, deletes) from the provided stage database.

- The client will provide access to the stage database, which contains the source data changes through CDC mechanisms.
- Establish connections to the stage database using ETL tools, such as Apache NiFi processors, to facilitate data extraction.
- Process and transform the stage-level data using ETL tools to load it into the target database or OLAP Mobility tables.
- Ensure the data extraction, transformation, and loading processes are efficient and maintain the integrity of the data.
- Validate that the OLAP Mobility is always up to date, reflecting the latest changes propagated from the stage database.

Apply Transformations from Base Tables to OLAP Tables

Transforming raw or base-level data into OLAP (Online Analytical Processing) tables optimized for querying and reporting.

- Define transformation logic based on business requirements, including aggregation, filtering, and deriving new metrics.
- Use ETL tools or data transformation engines to implement these transformations within the OLAP Mobility environment.
- Load the transformed data into OLAP tables, structured to support complex queries and analytical workloads.
- Ensure that the transformations are performed efficiently, considering scalability and performance.

Testing Connection with OLAP Tables

To validate that the OLAP tables are correctly populated and accessible for analytical purposes:

- Establish connections between the OLAP tables and the analytics or BI (Business Intelligence) tools used by the organization.
- Perform test queries to verify that the OLAP tables return accurate and expected results.
- Check the performance of queries to ensure that they meet the required response times and handle the expected load.

- Confirm that security protocols are in place to control access to sensitive data within the OLAP tables.

Scheduling Jobs at Respective Times to Pull Necessary Data

To automate the ongoing data extraction, transformation, and loading processes by scheduling jobs at predefined intervals:

- Define job schedules based on the data update frequency, business requirements, and operational windows.
- Use Apache NiFi processors to schedule and automate ETL processes for seamless data movement from the stage to the target database.
- Configure the jobs to run at the appropriate times, ensuring that data is consistently updated and available for analysis.
- Implement monitoring and alerting mechanisms to track job execution status and quickly resolve any issues that arise.

4. Proposed Solution In detail for each module

Data Integration and Schema Management Solution using NiFi and Postgres

This solution utilizes Apache NiFi to efficiently retrieve, process, and manage data from stage databases in Postgres, enabling seamless transformation and synchronization with OLAP Mobility. The system ensures flexibility, scalability, and automation to handle evolving data environments and dynamic schemas.

1. Data Retrieval from Stage Database Using Apache NiFi

Apache NiFi serves as the cornerstone for extracting data from the client-provided stage database in Postgres. By leveraging NiFi processors, secure connections to the stage database are established, enabling efficient data retrieval in real-time or batch mode. The system is designed to handle data with varying schemas and structures dynamically, ensuring compatibility with diverse data formats such as CSV and JSON. Incremental data retrieval processes ensure that only changes, such as inserts, updates, or deletes, are fetched, optimizing performance and minimizing redundancy in data synchronization.

2. Handling Dynamic Schemas and Data Mapping

The solution dynamically manages schema variations using NiFi's built-in capabilities. Incoming data from the stage database is mapped to align with the required structure of the OLAP Mobility environment. NiFi detects discrepancies between source and target schemas in real-time and resolves them through automated schema management. This ensures smooth integration and transformation of data, including applying custom business rules such as aggregations, filtering, and enriching datasets. Dynamic mapping guarantees seamless processing, even for evolving data structures, making the solution robust and adaptable.

3. Automated Database and Table Management

Automation is a key feature of this solution, with NiFi dynamically managing the creation and updates of target tables in OLAP Mobility. When data is processed, NiFi verifies the existence of corresponding tables in the target database. If a table does not exist, it is created automatically based on the mapped schema. For existing tables, any missing columns are added dynamically to accommodate new fields. This automation eliminates the need for manual interventions, streamlines database schema management, and ensures seamless compatibility with evolving data structures.

4. Data Insertion and Transformation into OLAP Tables

NiFi handles the transformation and insertion of data into OLAP Mobility tables with precision and efficiency. Transformation logic applies business-specific rules, such as deriving new metrics, filtering irrelevant records, and enriching datasets. Both batch and real-time processing options ensure flexibility, with NiFi validating every record for accuracy before insertion. This guarantees high data integrity and consistency, preparing the data in OLAP tables for complex analytical workloads while supporting rapid decision-making processes.

5. Real-Time Data Adaptability and Scalability

The solution's architecture is designed to adapt dynamically to frequent changes in source data schemas. NiFi's capability to handle schema modifications ensures that any new data formats from the stage database are integrated seamlessly without disrupting existing workflows. The scalable design accommodates growing data volumes and complexity without requiring significant architectural changes. Real-time monitoring and alerting mechanisms further enhance adaptability, ensuring uninterrupted operations in fast-evolving environments.

6. Scalable and Automated Job Scheduling

NiFi's flow scheduling capabilities automate the entire data pipeline, from extraction to transformation and loading into OLAP Mobility. Jobs are scheduled based on the frequency of data updates, ensuring continuous synchronization with minimal manual oversight. Monitoring tools track the status of job execution, providing alerts for any failures or delays to enable prompt resolution. By balancing real-time processing with scheduled batch jobs, the solution optimizes resource utilization while meeting business and operational requirements effectively.

5. Data Process Workflow

The data process workflow comprises two key components: the **initial data load** and the **incremental data load** using CDC with Nifi. Each workflow ensures that data from dynamic forms is processed and transformed into structured formats for KPI calculation.

- **Initial Data Load Process**

For the initial data migration, all existing records from the source PostgreSQL database will be extracted in bulk. This process is intended to transfer the complete dataset, encompassing historical records, to the staging area. Subsequently, the data will undergo transformation and processing before being transferred to the final KPI tables.

The steps involved are:

1. **Extract:** NiFi will extract the entire dataset from the source database.
2. **Transform:** JSON data will be flattened, and the business rules will be applied to normalize and clean the data. This includes validating field types, handling nested fields, and applying aggregation logic.
3. **Load:** Once transformed, the data will be loaded into the target database. Separate KPI tables will be created for each airport, ensuring that customized data points are captured.

- **Incremental Load Process using Nifi**

Following the completion of the initial data load, Nifi will handle the management of incremental data loads by capturing real-time changes from the Stage database through logical replication. These captured changes will then be transferred to the target database, where they will undergo processing by NiFi.

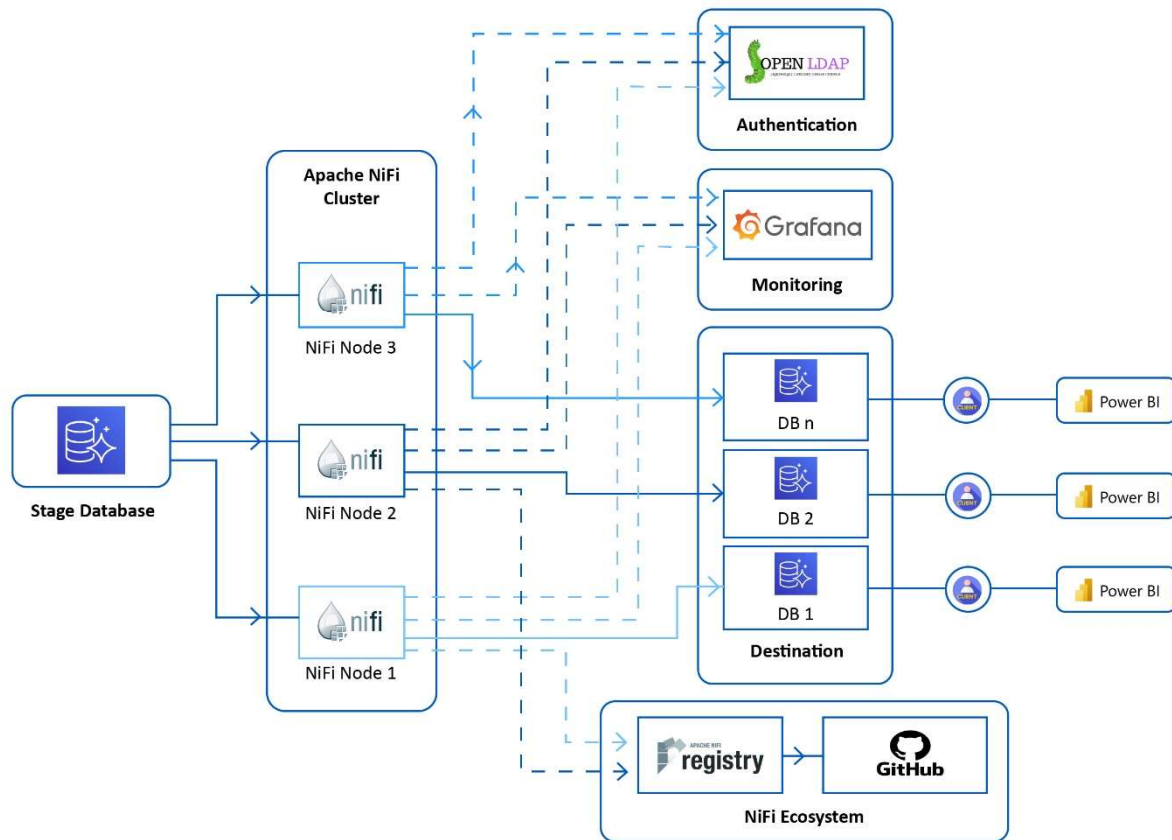
Transformation with NiFi: NiFi identifies the newly added or updated records, transforms the data based on the business rules, and loads it into the target KPI tables in near real-time.

Real-time Updates: This process ensures that KPI data is updated in real-time, providing the client with timely and accurate metrics.

- **Error Handling and Data Quality Checks**

Throughout both the initial and incremental data loading processes, robust data validation and error-handling mechanisms will be implemented. In the event of any errors encountered during data extraction, transformation, or loading, the system will meticulously log these errors and redirect the problematic records for thorough review. Stringent data quality checks will be conducted to ensure accurate field population, consistent data types, and adherence to the specified business rules.

6. Architecture



1. Stage Database

- Starting Point: The Stage Database acts as the source where the initial data is stored.
- Data flows from the Stage Database into the Apache NiFi Cluster for further processing.

2. Apache NiFi Cluster

- NiFi Cluster consists of multiple NiFi Nodes.
- These nodes work together to process and manage the data flow in a distributed manner.

Data Flow:

- NiFi Nodes ingest, process, and route the data from the Stage Database.
- The processed data is sent to the Destination Databases (DB 1, DB 2, DB n) for storage and further analysis.

3. Authentication (OpenLDAP)

- Purpose: OpenLDAP provides authentication services for secure access to the Apache NiFi Cluster.
- Integration: Authentication ensures only authorized users or processes can interact with NiFi nodes.

4. Monitoring (Grafana)

- Purpose: Grafana is integrated into the workflow for monitoring purposes.
- Functionality:
 - It monitors the performance and health of the NiFi nodes.
 - Tracks system metrics, workflows, and potential issues to maintain optimal operation.

5. Destination Databases

- After processing by the Apache NiFi Cluster, data is sent to multiple Destination Databases:
- These databases serve as the final repositories for the transformed and processed data.

6. Power BI for Visualization

- Data stored in the Destination Databases is accessed by clients using Power BI.
- Purpose:
 - Power BI provides interactive visualizations and business intelligence reports.
 - Clients use these reports for data analysis and decision-making.

7. NiFi Ecosystem

The NiFi Ecosystem consists of:

- **Apache NiFi Registry:**
 - Used for version control and managing data flow definitions.
 - Ensures workflows are tracked, shared, and reusable.
- **GitHub:**
 - Acts as a repository for storing flow definitions, scripts, and other configurations.
 - Facilitates collaboration and version management for the development team.

Workflow Summary:

1. Stage Database → Initial data storage.
2. Apache NiFi Cluster (Nodes 1, 2, 3) → Processes and routes data.
3. Authentication (OpenLDAP) → Secures access to NiFi.
4. Monitoring (Grafana) → Tracks and monitors the NiFi system.
5. Destination Databases → Processed data is stored here (DB 1, DB 2, DB n).
6. Power BI → Clients visualize and analyze the data.
7. NiFi Ecosystem (Registry and GitHub) → Manages version control and collaboration.

7. Main flow for OLAP

The tables listed below are used in the NiFi flow:

SL. No	Table name
1	data_forms_formanswer
2	data_forms_formversion
3	data_forms_formparent
4	inspections_inspection
5	airport_airport
6	users_aerosimpleuser
7	auth_user
8	safety_management_hazard
9	safety_management_hazardschema
10	snow_management_snowevent
11	safety_management_safetyrisk
12	safety_management_sraschema
13	data_forms_formprocess
14	data_forms_formprocess_approvals
15	data_forms_formapprovals
16	data_forms_formapprovaldef

Form Answer Table Mapping

Importing Required Libraries

The script begins by importing essential libraries required for efficient data handling, JSON processing, and database operations. **IOUtils** simplifies operations related to input and output streams, such as converting streams to byte arrays or reading content efficiently. **JsonSlurper** enables easy parsing of JSON strings into Groovy objects like maps or lists, facilitating seamless traversal and manipulation of JSON data. Additionally, **JsonOutput** converts Groovy objects into JSON strings, making it convenient to serialize structured data for storage or further processing. For database operations, **DriverManager**, **Connection**, **PreparedStatement**, and **ResultSet** form the core of JDBC components. **DriverManager** manages database connections, while **Connection** establishes the connection to the database. **PreparedStatement** allows the execution of parameterized SQL queries, ensuring enhanced security and performance. **ResultSet** helps in row-by-row retrieval of query results.

To handle input streams efficiently, `BufferedReader` and `InputStreamReader` are utilized. `InputStreamReader` converts raw byte streams into character streams, while `BufferedReader` adds buffering for efficient memory usage, particularly when dealing with large datasets. The `StreamCallback` utility from NiFi is employed to define callbacks for reading and writing `FlowFile` content, enabling efficient processing of large data streams within the workflow.

FlowFile Retrieval

The script begins its execution by attempting to retrieve a `FlowFile`, which acts as a data packet in NiFi workflows, using the `session.get()` method. A `FlowFile` contains both data and associated metadata, which the script processes. If no `FlowFile` is available, the script exits gracefully without performing further actions. This ensures that errors or unnecessary computation are avoided when no input data exists in the session, preserving system resources and maintaining workflow stability.

Variable Declarations

Reader Setup

The **reader** variable is initialized to hold a `BufferedReader` instance, which reads the content of the `FlowFile` efficiently. `BufferedReader` ensures memory efficiency when dealing with large data volumes.

Header and Value Variables

- **headers:** This array stores column names or field identifiers extracted from the `FlowFile`'s first line.
- **values:** A corresponding array that holds actual values aligned with the parsed headers. Together, these arrays serve as the backbone for organizing the input data for downstream processing.

Data Organization Maps

- **dataSchema:** Stores schema definitions, including field names and metadata, specific to the current form being processed.
- **flattenedAttributes:** Simplifies nested or hierarchical structures into key-value pairs, ensuring easier access and integration during mapping and database storage.
- **inspectionSchemas and inspectionFormIds:** Manage inspection-related schemas and corresponding IDs for specific form modules. These variables ensure proper organization of module-specific data.
- **schemaMap:** Maintains a mapping of field IDs to their respective titles or metadata.
- **valueReplacementMap:** Stores replacement mappings, particularly for dropdowns, multi-select values, or enumerated fields, allowing consistent value representation.

Tracking Titles

To handle potential duplicate field titles, **titleCountMap** tracks occurrences of each title in the schema. Additional variables like **mappedTitles**, **hazardMappedTitles**, and **sraMappedTitles** ensure that column names remain unique in the output data. Separate mappings are maintained for general fields, hazard-specific fields, and Safety Risk Assessment (SRA) fields.

Key Identifiers and Metadata

- **id**: Represents the unique identifier for the current form answer or record being processed.
- **formDate**: Captures the date associated with the form submission.
- **hazardId**, **snowId**, and **sraId**: Represent unique IDs for hazard, snow, and SRA-specific modules. These IDs enable conditional processing for corresponding modules.
- **hazardSchemaJson**, **hazardResponse**, **sraSchemaJson**, **sraResponse**: These variables store raw or processed schema and response data for hazards and SRAs. They play a crucial role in conditionally handling module-specific data, ensuring flexibility in processing form-related content.

General Data Handling

The **jsonData** variable temporarily holds parsed JSON data extracted from the FlowFile content. Transformation tracking is managed through variables like **updatedSchema**, **updatedHazardSchema**, and **updatedSraSchema**, which ensure consistent and unified outputs for general, hazard, and SRA fields after applying necessary modifications.

Processing the FlowFile Content

The script efficiently processes FlowFile content using the `session.write()` method, which defines a callback function. This callback mechanism ensures that potentially large data is handled in a streaming fashion rather than being loaded entirely into memory.

Reading Data

The content of the FlowFile is accessed through its input stream, allowing real-time data retrieval. The **inputStream** variable represents the stream's source, and **BufferedReader** wraps the stream for efficient, character-based reading. Encoding is explicitly set to **UTF-8**, ensuring compatibility with multi-language input data and special characters.

Buffered Reading

The combination of **InputStreamReader** and **BufferedReader** allows memory-efficient reading of FlowFile content. While **InputStreamReader** handles the conversion of byte streams to character streams, **BufferedReader** introduces buffering to improve performance, especially for large datasets.

Header and Value Extraction

- **headerLine**: The first line of the FlowFile content is read and stored as **headerLine**. This line contains the names of fields or identifiers.
- **valueLine**: The second line, which contains the actual data values, is extracted and stored as **valueLine**.

The headers provide the structural context, while the values supply the actual content. These two lines are pivotal in organizing the data for further processing.

Purpose of Headers and Values

The headers and values act as the foundation for the script's processing logic. Headers provide a reference to identify each data field, while values contain the corresponding data. The combination of these arrays enables:

- **Field Mapping:** Aligning data with the appropriate database schema.
- **Data Transformation:** Applying transformations such as flattening JSON, replacing values, or enriching data.
- **Validation and Integration:** Ensuring the data aligns with schema definitions before integrating it into target systems.

The extracted content is subsequently split, validated, and mapped to prepare it for database interactions or downstream output generation.

Validating and Parsing Header and Value Lines

The script begins by ensuring that both the headerLine and valueLine are available and valid before proceeding. These lines are fundamental for extracting and organizing data from the **FlowFile** content. A **conditional check** is applied to confirm their presence. If either line is found to be null, the script logs an error message: "Error reading headers or values from flow file content" and throws an **IllegalStateException**. This exception halts further execution, preventing the script from processing incomplete or invalid data, ensuring data integrity and robustness.

Once validated, the **headers** and **values** are parsed for clean extraction. The headerLine is split using a **delimiter defined as a parameter**, which is configured in the NIFI global parameters. The trimming operation ensures any leading or trailing whitespace is removed from each header, resulting in clean and consistent field names. The same steps are applied to the valueLine, producing corresponding values aligned with the headers. The resulting **arrays of headers and values** maintain a consistent relationship, where each header corresponds to its value at the same index.

To ensure the **original content** of the FlowFile remains intact, the data is converted into a byte array using `IOUtils.toByteArray(inputStream)` and written to the output stream. This step guarantees that the input content is preserved while the script processes it further for transformations and validations.

Initializing Form Identifiers

The script initializes placeholders for critical form identifiers, namely **formParentId** and **formVersionId**, both of which are initially set to null. These identifiers are later extracted dynamically from the FlowFile content during the header and value mapping process.

- **formParentId:** This identifier links the current form data to its parent form, and database table selection during downstream processing.
- **formVersionId:** Captures the specific version of the form, which is crucial for schema retrieval and enabling relational mapping and organization.

These variables serve as the foundational keys for subsequent database queries, schema selection, and conditional logic execution throughout the script.

Iterating Through Headers and Values

The script iterates over each **header** and its corresponding **value** using the `headers.forEachWithIndex` loop. This ensures that both arrays are **processed in alignment, allowing each header to be mapped to its respective value. During iteration, the script performs the following operations:**

- I. **Mapping Headers to Values:** The script checks whether the current header has a corresponding value in the values array. If the value array is shorter than the headers, a default empty string (") is assigned to maintain alignment. This Prevents null issues during processing.
- II. **Mapping Key Fields:** Key variables are assigned based on specific header names to identify critical fields:
 - a. **form_parent_id → formParentId:** Identifies the parent form ID for organizing or linking forms.
 - b. **form_version_id → formVersionId:** Captures the version of the form for schema validation and table selection.
 - c. **First Header → id:** The value of the first header is assigned to the **id** variable, which serves as a unique identifier for the current record.
 - d. **hazard_id, snow_id, sra_id:** These headers are mapped to respective variables (`hazardId`, `snowId`, and `srald`), enabling the script to conditionally process hazard, snow, and Safety Risk Assessment (SRA) modules based on their presence.
 - e. **form_date → formDate:** The form date is captured and stored for validation, chronological organization, or record filtering.
- III. **Handling data Fields:** If the header is **data**, the corresponding value is treated as a JSON string and parsed using `JsonSlurper`. The parsed JSON content is stored in the **jsonData** map, allowing further traversal and manipulation of structured data later in the workflow.
- IV. **Flattening Attributes:** For all headers other than data, the script flattens nested structures into simplified key-value pairs. This is achieved using the **flattenJson** function, which processes the value and combines it with the header name. The resulting flattened attributes are stored in the **flattenedAttributes** map, a unified data structure that simplifies further processing and integration.

This iterative process ensures that all headers and values are efficiently mapped, with critical fields identified and structured data prepared for downstream operations.

Validating Required Fields

To maintain data integrity, the script performs validation on essential fields after mapping:

- I. **formVersionId Validation:** The script checks whether `formVersionId` has been assigned a value. If it remains null, an error is logged: "form_version_id is missing". This identifier is critical because it determines the database schema and table to be used for the current form data. Without this field, the script cannot proceed with schema validation or table mapping.

- II. **formDate Validation:** Like `formVersionId`, the **formDate** field is validated to ensure it is not null. If the form date is missing, the script logs an error message: "form_date is missing". The form date plays an important role in the logical organization of records, chronological validations, or filtering processes for downstream workflows.

The validation of required fields ensures that incomplete or invalid data does not propagate further into the workflow, preventing errors during schema validation, enrichment, or database insertion.

Determining the Table Name

The script dynamically determines the target table name based on the availability of specific form identifiers. If the `formParentId` is available, the table name is constructed as **`tbl_form_${formParentId}`**. In cases where `formParentId` is not provided, the script defaults to **`tbl_form_${formVersionId}_v`**. This dynamic logic ensures that the correct database table is selected for querying or inserting form-related data based on the specific form or version identifiers in the input.

Database Connection Initialization

The script initializes a connection to the database using JDBC, leveraging globally configured parameters from NiFi. These include **`dbUrlAero`**, **`dbUser`**, and **`dbPassword`**, which are securely stored in NiFi to ensure:

- I. **Reusability:** The same database credentials can be used across different scripts and workflows.
- II. **Security:** Sensitive information, such as passwords, is not hardcoded and is protected within the NiFi configuration.

The connection is established using `DriverManager.getConnection()`, which initializes the `Connection` object (`connectionAero`). This connection facilitates SQL query execution and data retrieval. Additional placeholders are defined to store key variables, including:

- I. **`airportId`:** Placeholder for the airport identifier, retrieved later during processing.
- II. **`completedByFullname` and `createdByFullname`:** Store the full names of the users who completed and created the form, enabling proper metadata enrichment.

Querying the Schema

To validate and process form answers, the script queries the schema for the current form. A **parameterized SQL query** is prepared to fetch the schema column from the `data_forms_formversion` table. The query uses the following parameters:

- I. **`form_id`:** Mapped to `formParentId`, representing the parent form in `data_forms_formversion` table.
- II. **`id`:** Mapped to `formVersionId`, identifying the specific version of the form in `data_forms_formversion` table.

Before binding these parameters, `formParentId` and `formVersionId` are converted to integers for query execution. The query is executed using **`schemaPreparedStatement.executeQuery()`**, returning a `ResultSet` containing the rows that match the provided identifiers.

Tracking Field Titles

To ensure the uniqueness of field titles in the output, the script tracks occurrences of each title using a **map** (`titleCountMap`). This mechanism resolves duplicate titles that may appear within the schema with the **D*** added to their title.

- I. **Initialization:** For each field in the schema, the script checks if the title already exists in `titleCountMap`. If the title does not exist. It is initialized with a count of.
- II. **Incrementing Count:** Each subsequent occurrence of the title increments its count in the map.
- III. **Handling Duplicates:** The **D*** is later used to append unique identifiers to titles, ensuring no two fields have identical names in the output.

Iterating Over Schema Fields

The script iterates through the fields in the schema, specifically within the **`schemaMap.fields`** structure. Each field contains metadata such as:

- **field.id:** A unique identifier for the field.
- **field.title:** The name or label of the field.
- **field.type:** The type of the field (e.g., `select`, `multi_field`, `inspection_checklist`).
- **Additional Properties:** Fields may include attributes like `selected_module` or predefined values for specific types.

Each field is processed individually based on its type, and necessary actions are applied to map, track, and transform the field data.

Processing Fields of Type "inspection_checklist"

If the field's type is **inspection_checklist**, and it includes a `selected_module`, it indicates the field is linked to a predefined inspection module.

The following actions are taken:

- I. **Form ID Extraction:** The `selected_module` contains a property called `key`, which uniquely identifies the inspection module. This key is extracted and stored in the variable `formId`.
- II. **Mapping Titles:** The field's ID is mapped to its title in the **`dataSchema`** map, ensuring the title is accessible for downstream transformations.
- III. **Inspection Form Tracking:** The extracted `formId` is stored in `inspectionFormIds[field.id]`. This links the field ID to its associated inspection module, enabling future references during processing.

Processing Fields of Types "select", "multiselect", and "multi_field"

For fields of types **select**, **multiselect**, or **multi_field**, the script identifies that these fields allow users to select one or multiple options from a predefined list of values.

The script performs the following actions:

- I. **Value Replacement Mapping:**
 - a. The field includes property values, which contains a list of options.
 - b. Each option consists of a **key** (internal representation) and a **value** (human-readable label).
- II. **Mapping Generation:** The script creates a mapping of key → value pairs and stores it in `valueReplacementMap[field.id]`.

This mapping enables transformations where internal keys are replaced with their corresponding labels during output preparation.

Processing Other Field Types

For fields that are not of types of **inspection_checklist**, **select**, **multiselect**, or **multi_field**, the script treats them as standard fields requiring no special handling. The script directly maps each field's ID to its title in the **dataSchema** map. These fields are typically simple text or numeric inputs, which do not require additional transformations or value replacements.

Purpose of Error Logging

The script incorporates robust error-logging mechanisms to ensure traceability and clarity when specific conditions are not met.

- I. **Condition:** After executing the SQL query to fetch the schema, the script uses **resultSet.next()** to check for results.
 - a. If no rows are returned, the script logs an error indicating that the schema for the given `formParentId` and `formVersionId` was not found.
- II. **Error Message:** The message is logged in the format: "No schema found for `form_parent_id=${formParentId}` and `form_version_id=${formVersionId}`".

The log dynamically includes the form identifiers, allowing developers to identify the exact inputs that caused the error.

Importance of Logging This Error

Error logging is crucial for the following reasons:

- I. **Debugging and Diagnostics:** The error message provides immediate feedback to developers or administrators, helping them identify whether the issue lies with the input data or the database configuration.
- II. **Workflow Integrity:** Logging ensures that failures are identified at the earliest possible stage, preventing undefined behavior or incomplete processing.
- III. **Example Scenarios:**
 - a. If the input data is missing formParentId or formVersionId, the query will fail to retrieve the schema.
 - b. If the data_forms_formversion table lacks the necessary schema, the error highlights potential issues with database integrity or configuration.

Actions to Take Based on the Log

When the error is logged, the following corrective measures can be taken:

- I. **Investigating the Cause:**
 - a. Verify that the formParentId and formVersionId are correctly extracted from the FlowFile content.
 - b. Check if the input FlowFile contains valid form identifiers.
- II. **Corrective Measures:**
 - a. If the issue is with the input data, validate and correct the input FlowFile before reprocessing.
 - b. If the issue is with the database, verify that the data_forms_formversion table contains the expected schema for the given IDs.

By following these steps, administrators can resolve the issue and ensure smooth execution of the workflow.

Querying Airport Information

The script retrieves the **airport code** associated with a given formParentId. This operation is essential for linking form answers to their respective airport contexts, providing valuable metadata for downstream processes.

SQL Query Details

To fetch the airport code, the script performs a **join operation** between the data_forms_formparent table (aliased as dp) and the airport_airport table (aliased as aa):

- **dp.airport_id = aa.id:** Matches the `airport_id` field in the `data_forms_formparent` table with the corresponding record in the `airport_airport` table.
- **dp.id = ?:** Filters the results to include only the row associated with the specified `formParentId`.

This ensures that the retrieved airport code corresponds specifically to the form's parent ID.

Setting Query Parameters

The placeholder `?` in the query is dynamically replaced with the integer value of `formParentId` using the following operation:

`airportPreparedStatement.setInt(1, formParentId.toInteger())`: Safely binds the parameter to the SQL query.

Executing the Query

The query is executed using **`airportPreparedStatement.executeQuery()`**, which retrieves the matching record from the database as a **ResultSet**.

Processing Query Results

Once the query execution is complete, the script processes the retrieved data as follows:

- I. If a matching row is found:
 - a. The **airport code** is extracted from the `code` column using **`resultSet.getString("code")`**.
 - b. To standardize the identifier, the script prepends the prefix `"db_"` to the retrieved airport code. For example, an airport code `"ABC"` becomes **`db_ABC`**.
 - c. A log message confirms the successful retrieval and update of the airport ID:
`"Airport ID retrieved and updated: db_ABC"`.
- II. If no matching row is found:
 - a. The script logs a detailed error message indicating the absence of an airport code for the provided `formParentId`:
`"No airport_id found for form_parent_id=${formParentId}"`.
 - b. This error helps identify missing or incorrect relationships in the `data_forms_formparent` table or potential database misconfigurations.

Querying Full Names of Users

The script enriches the form data by retrieving the **full names of users** who:

1. **Completed the form:** The name is stored in the `completedByFullname` variable.
2. **Created the form:** The name is stored in the `createdByFullname` variable.

SQL Query Details

The query selects user full names by joining relevant tables:

- **dfFa.completed_by → uas.fullname**: The `completed_by` field in the `data_forms_formanswer` table (`dfFa`) is matched with the `username` in the `users_aerosimpleuser` table (`uas`) to retrieve the full name.
- **dfFa.created_by → uas2.fullname**: Similarly, the `created_by` field in the `data_forms_formanswer` table is matched with the `users_aerosimpleuser` table to fetch the creator's full name.

This join operation ensures that **human**-readable user names are retrieved for both fields.

Setting Query Parameters

The placeholder `?` in the query is dynamically replaced with the integer value of the `id` variable:

- **`fullNamePreparedStatement.setInt(1, id.toInteger())`**: Safely binds the parameter to the query.

Executing the Query

The query is executed using **`fullNamePreparedStatement.executeQuery()`**, which fetches the user details matching the specified `id` from the database.

Processing Query Results

Once the query results are returned, the script processes the data as follows:

- I. If a matching row is found:
 - a. **`completedByFullName`**: Retrieves the full name of the user who completed the form.
 - b. **`createdByFullName`**: Retrieves the full name of the user who created the form.
- II. These values are then added to the **`flattenedAttributes`** map, ensuring that the enriched metadata becomes part of the output:
 - a. **`"completed_by"`**: Updated with the value of `completedByFullName`. If no value is found, the existing value in `flattenedAttributes` is retained as a fallback.
 - b. **`"created_by"`**: Updated with the value of `createdByFullName`. Similarly, if no full name is retrieved, the script retains the existing value.

This approach ensures that user metadata is included wherever available while maintaining fallback mechanisms to handle missing data.

Error Handling and Logging

The script incorporates specific error-handling mechanisms for the airport query and user query processes to ensure traceability and robustness.

Airport Query Error Handling

If the airport query does not return a matching record, the script logs a detailed error message:

- **Error Message:** "No airport_id found for form_parent_id=\${formParentId}".

This message identifies the form parent ID for which the relationship could not be resolved, allowing for quick diagnosis of missing or incorrect data in the `data_forms_formparent` table.

User Query Error Handling

In the case of the user query, no explicit error logging is performed when user data is missing. Instead:

- If no matching record is found in the database, the `completedByFullname` and `createdByFullname` variables remain null.
- The script does not update the corresponding fields in the `flattenedAttributes` map, ensuring that no invalid or incomplete data is introduced.

This silent handling of missing user data allows the workflow to continue smoothly without introducing unnecessary interruptions.

Initializing Schema Maps

At the start of schema processing, the script initializes the required schema maps to ensure they are clean and contain only the current processed data. The following schema maps are reset to empty:

- **updatedSchema:** Holds the processed general schema for form fields.
- **updatedHazardSchema:** Stores schema details specific to hazard-related fields.
- **updatedSraSchema:** Maintains the schema structure specific to Safety Risk Assessments (SRAs).

Additionally, a **titleOccurrences** map is initialized to track how often field titles appear. This map plays a key role in identifying and handling duplicate field titles, ensuring the final schema output has unique field names.

Updating General Schema

The script processes the general schema for form fields by iterating through each entry in `schemaMap.fields`, which contains metadata about the form fields. The metadata for each field includes:

- **field.id:** A unique identifier for the field.
- **field.title:** The human-readable name or title assigned to the field.

Handling Duplicate Titles

To ensure field titles remain unique, the script tracks occurrences of each title using the `titleOccurrences` map:

- **Check for Title Existence:**
 - If the title is not already in titleOccurrences, it is added with an initial count of **1**, and the field ID is mapped directly to the title in updatedSchema.
 - If the title already exists in titleOccurrences, its count is incremented. To ensure uniqueness, the script appends the field ID (D*) to the title (e.g., "field_id_title") and updates the entry in updatedSchema.

Logging Updated Schema

After all fields have been processed, the updated schema is logged to confirm changes and verify that duplicate titles were appropriately handled. This log ensures transparency into how the script manages schema processing and title resolution.

Handling Foreign keys

1. Handling Hazard-Specific Data

The script conditionally processes hazard-specific data if a **hazardId** is present. This enables targeted enrichment of form data with hazard-related details, which are retrieved from the database.

Querying Hazard Data

A parameterized SQL query is used to fetch hazard-specific details:

- I. **Data Retrieved:**
 - a. **title:** The name or label for the hazard.
 - b. **description:** A textual description of the hazard.
 - c. **response:** The response data related to the hazard.
 - d. **schema:** The JSON schema that defines the structure of the hazard-related fields.

The query performs joins between the following tables:

- I. **safety_management_hazard (smh)** and **data_forms_formanswer (dff)**: Joins via hazard_id to fetch hazard details.
- II. **safety_management_hazardschema (smh2)**: Retrieves hazard schema by linking with the form_id in smh.

Setting Query Parameters

The query placeholder? is replaced dynamically with the integer value of hazardId using:

```
hazardPreparedStatement.setInt(1, hazardId.toInteger()).
```

Executing the Query

The script executes the query using **hazardPreparedStatement.executeQuery()**, which returns a **ResultSet** containing the matching row, if found.

Processing Hazard Query Results

When the query executes successfully, the script processes the returned data:

- I. **Data Extraction:**
 - a. **title:** Retrieved from the title column in the result set.
 - b. **description:** Extracted from the description column.
 - c. **hazardResponse:** Stores response-related data for the hazard.
 - d. **hazardSchemaJson:** Captures the schema definition for the hazard fields in JSON format.

A log message confirms the successful retrieval of hazard details, ensuring visibility into the process before further handling of the hazard schema.

Parsing and Processing Hazard Schema

If **hazardSchemaJson** is not null, the script proceeds to parse and process the hazard schema:

Parsing Hazard Schema

The hazard schema in JSON format is parsed into a Groovy object using **JsonSlurper**, allowing the script to traverse and manipulate the schema data programmatically. The parsed schema contains metadata for hazard-specific fields.

Processing Hazard Fields

The script iterates through each field in **hazardSchema.fields**, performing the following actions:

- I. **Identifying Hazard Field Types:**
 - a. If the field ID begins with the letter **d**, it is marked for further processing.
 - b. The field's ID is mapped to its title in the **hazardSchemaMap**, ensuring a referenceable structure for downstream processes.
- II. **Handling Select-Like Fields:** For fields of types **select**, **multiselect**, or **multi_field** that include predefined values:
 - a. A **value replacement mapping** is created to associate internal keys with human-readable labels.
 - b. Each key-value pair in the values property is stored in the **hazardValueReplacementMap**.

This mapping enables transformations where internal representations (e.g., "1", "2") are replaced with meaningful labels (e.g., "Yes", "No"), ensuring the final output is user-friendly and interpretable.

Handling Duplicate Titles in Hazard Schema

The script begins by addressing the issue of duplicate titles within the hazard schema to ensure all field titles remain unique. To achieve this, it initializes an empty map called **hazardTitleOccurrences**. This map tracks the occurrence of each title as the hazard schema is processed, preventing potential conflicts caused by duplicate names. The script then iterates through each key-value pair in the **hazardSchemaMap**, where the **key** represents the field ID, and the **value** represents the field title.

For each field, the script checks whether the title already exists in **hazardTitleOccurrences**. If the title is not present, it is added to the map with an initial count of 1, and the field ID is directly mapped to its title in the **updatedHazardSchema**. However, if the title already exists in the map, its count is incremented, and the script appends the field ID to the title to ensure uniqueness.

Processing Hazard Response

After the schema has been processed, the script moves on to handle hazard-specific response data. To store this response data, an empty map called **updatedHazardResponse** is initialized. The script first checks whether the **hazardResponse** data is present. If a hazard response exists, the JSON-formatted content of the response is parsed using **JsonSlurper**, converting it into a Groovy map known as **hazardResponseMap**. This parsed response map contains key-value pairs, where the **key** corresponds to the field ID in the hazard schema, and the **value** represents the specific response provided for that field.

Mapping Hazard Response Fields

As the script iterates through the **hazardResponseMap**, it ensures that the response fields align with the schema. For each key (field ID), the script checks if it exists in the **updatedHazardSchema**. If the key is present, it generates a **column name** based on the field title. To maintain a consistent format, the field title is converted to lowercase. Duplicate column names are managed through the **hazardMappedTitles** map, which tracks occurrences of each column name. If a column name is encountered for the first time, it is used directly. If a duplicate column name is detected, the field ID is appended to the column name to ensure uniqueness, and the occurrence count in **hazardMappedTitles** is updated accordingly.

For fields that require value replacements, the script checks if the key exists in **hazardValueReplacementMap**. This map contains mappings of internal keys (e.g., numeric codes) to human-readable labels. If a corresponding mapping exists, the script replaces the original value with its equivalent label. If no replacement is found, the original value is retained as-is. The processed value is then stored in the **updatedHazardResponse** map under a new key following the format **hazard_response_{finalColumnName}**. This naming convention ensures that hazard-specific response data is stored with unique, descriptive keys that are ready for further downstream usage.

Flattening Hazard Response

Once the hazard response data is processed and stored in the **updatedHazardResponse** map, the script flattens the data to simplify its structure. Using a helper function such as **flattenJson**, any nested elements within the response map are converted into a single-level key-value format. This flattened data is then added to the **flattenedAttributes** map, which consolidates all processed attributes, including general form fields, hazard responses, and additional metadata.

To provide context to the hazard-specific data, the script adds hazard-related metadata to **flattenedAttributes**, including:

- **hazard_title**: Contains the title of the hazard.
- **hazard_description**: Stores the description of the hazard.

These metadata fields ensure that hazard responses are enriched with descriptive information, improving clarity and usability. Finally, a log entry confirms the successful addition of hazard details, including the hazard title and description. This message ensures visibility into the workflow and verifies the successful integration of hazard response data.

Handling Missing Hazard Data

In scenarios where the query for hazard-specific data does not return any results, the script gracefully handles the absence of data. Instead of halting the workflow, it logs a clear message indicating that no hazard details were found for the provided **hazardId**. The log entry, formatted as "**No hazard details found for hazard_id=\${hazardId}**", helps identify missing or invalid hazard references in the database. This mechanism ensures that workflows continue to execute smoothly even when certain hazard data is unavailable, while also providing traceable feedback for debugging and resolution.

2. Snow Event Data Retrieval and Processing

The script includes a dedicated section to retrieve **snow event information** based on the value of **snowId**. Unlike other components that involve complex schema processing, this part of the script directly fetches specific details from the **snow_management_snowevent** table in the database. The focus here is on retrieving specific attributes, namely the **storm name** and **description** of the snow event, to enrich the form answer data. The process begins with a **conditional check** to determine whether the **snowId** is provided. If the **snowId** is present, the script proceeds to execute a database query to fetch the corresponding snow event details. This conditional approach ensures that snow data is processed only when relevant and avoids unnecessary operations if no **snowId** is available.

To retrieve the snow event details, a **parameterized SQL query** is prepared. The query targets two key columns in the **snow_management_snowevent** table:

- **storm_name**: Represents the name of the storm associated with the specific snow event.
- **description**: Provides additional descriptive context or details about the snow event.

The query filters the data by matching the id column in the table with the provided snowId. To ensure secure and efficient query execution, the placeholder ? in the query is dynamically replaced with the integer value of snowId using the statement:

```
snowPreparedStatement.setInt(1, snowId.toInteger())
```

Once the parameterized query is ready, it is executed using `snowPreparedStatement.executeQuery()`, which interacts with the database and returns a `ResultSet`. The `ResultSet` contains the matching row for the specified snowId, if available.

When the query successfully retrieves a matching row, the script processes the results by extracting the required attributes from the `ResultSet`:

- **stormName:** Retrieved using `getString("storm_name")`, this field contains the name of the storm linked to the snow event.
- **description:** Retrieved using `getString("description")`, this field offers a descriptive text providing additional context about the snow event.

Once these values are extracted, they are added to the `flattenedAttributes` map to ensure they become part of the consolidated dataset used for downstream processing. The extracted data is stored under descriptive and identifiable keys:

- **snow_storm_name:** Contains the name of the storm.
- **snow_description:** Holds the description of the snow event.

This step enriches the form answer data with snow event metadata, providing meaningful context that can be leveraged for reporting, analysis, or display purposes. To confirm the successful retrieval and inclusion of snow details, the script logs a message that includes the values of the retrieved attributes:

```
"Snow details retrieved and added: storm_name=${stormName}, description=${description}"
```

In cases where the query does not return any results—indicated by `snowResultSet.next()` evaluating to false—the script gracefully handles the absence of snow data. Instead of halting execution or raising an error, a log entry is generated to note the issue:

```
"No snow details found for snow_id=${snowId}"
```

This log message ensures that the absence of snow event details is recorded for future debugging, auditing, or analysis while allowing the workflow to continue without interruption. This approach prevents potential workflow failures caused by missing data and maintains the robustness and reliability of the overall script execution.

By handling both successful retrieval and missing data scenarios, the script ensures that snow event metadata is processed effectively when available and logged appropriately when absent. This makes the

snow event section an integral part of the data enrichment process, contributing to the completeness and context of the consolidated form answer data.

3.Processing Safety Risk Assessment (SRA) Data

The script includes a section dedicated to retrieving and processing Safety Risk Assessment (SRA) data, which is conditional upon the presence of a valid `srald`. This process involves retrieving essential SRA details, such as the **title**, **description**, and **response**, while also handling and processing the associated schema if it exists. The SRA data serves as a critical component for enriching the form answer data, providing structured and meaningful information related to safety risks.

The workflow begins with a **conditional check** to verify if a `srald` is available. If the `srald` is provided, the script proceeds to query the database to fetch the relevant SRA details and its corresponding schema. The query performs a join operation between two key tables:

- **safety_management_safetyrisk (sms)**: This table stores the primary SRA data, including the title, description, and response.
- **safety_management_sraschema (sch)**: This table contains the schema definitions associated with the specific SRA form.

The query is designed to filter rows based on the provided `srald`, matching it with the `sra_id` field in the `data_forms_formanswer` table. To ensure security and accuracy, the placeholder `?` in the query is dynamically replaced with the integer value of `srald` using:

```
sraPreparedStatement.setInt(1, srald.toInteger());
```

Once the query is prepared, it is executed using `sraPreparedStatement.executeQuery()`, which returns a `ResultSet` containing the matching record, if any.

Processing Query Results

If the query successfully retrieves a matching row, the script extracts the following key pieces of data from the `ResultSet`:

- **Title**: Retrieved using `getString("title")`, this field represents the name or label of the SRA.
- **Description**: Extracted using `getString("description")`, this field provides additional context or descriptive details about the SRA.
- **SRA Response**: Retrieved using `getString("response")`, this contains the raw response data for the SRA, typically formatted as JSON.
- **SRA Schema**: Extracted using `getString("schema")`, this contains the schema definition for the SRA in JSON format.

These extracted values are essential for processing both the response and schema components of the SRA, enabling the script to map and structure the data appropriately.

Processing the SRA Schema

Once the SRA schema is retrieved, the script initializes two empty maps to organize and manage the schema data effectively:

- **sraSchemaMap**: This map stores field IDs as keys and their corresponding titles as values.
- **sraValueReplacementMap**: This map is used to store value replacement mappings for fields that have predefined options, such as dropdowns or multi-select lists.

If the **sraSchemaJson** is present, it is parsed into a Groovy object using **JsonSlurper**. This allows the script to traverse the schema structure programmatically and process each field in the schema. The script iterates through the **array of fields** within the parsed schema object, performing specific actions for each field. During this iteration, the script first checks whether the **field ID** starts with the letter 'd'. Fields with IDs that meet this condition are considered valid for further processing and are added to the **sraSchemaMap**, mapping their field.id to their corresponding field.title. This ensures that all relevant SRA fields are mapped to human-readable titles, which can be used later for data enrichment and output generation.

For fields of types **select**, **multiselect**, or **multi_field** that include predefined values, the script processes the values property. The values property typically contains key-value pairs

The mapping ensures that raw response values, which may contain internal keys, can later be replaced with their corresponding human-readable labels for improved clarity and usability. The processed schema and value replacements are stored for downstream use in transforming and enriching the SRA response data.

Handling Duplicate Titles in the SRA Schema

To ensure the uniqueness of field titles in the SRA schema, the script begins by tracking occurrences of each title. For this purpose, an empty map called **sraTitleOccurrences** is initialized at the start of the process. This map plays a critical role in identifying and handling duplicate titles, which might otherwise cause conflicts during schema processing.

The script iterates through each key-value pair within the **sraSchemaMap**, where the **key** represents the field ID and the **value** represents the corresponding field title. As it processes each field, the script checks whether the title already exists in the sraTitleOccurrences map. If the title does not exist, it is added to the map with an initial count of **1**, and the field ID is directly mapped to the title in the **updatedSraSchema**. However, if the title already exists in the map, the count is incremented to reflect the duplicate occurrence. To ensure uniqueness, the script appends the field ID (D*) to the title, generating an updated title like "field_id_title". This updated title is then stored in the updatedSraSchema, resolving the duplication issue.

Once all fields in the SRA schema are processed, the script logs the updated schema to confirm that duplicate titles have been effectively handled. This log provides visibility into the changes made and ensures that the schema is ready for downstream processing without conflicts.

Processing the SRA Response

After handling the schema, the script proceeds to process the **SRA response data**. To organize this data, it initializes an empty map called **updatedSraResponse**, which serves as the storage for the processed SRA response fields. Before any processing, the script checks if the **sraResponse** data is present. If the response exists, it is parsed from its JSON format into a Groovy map called **sraResponseMap** using the **JsonSlurper** utility. This step allows the script to iterate over the key-value pairs contained in the response data.

In this context, the **key** represents the field ID from the SRA schema, and the **value** holds the raw response value associated with that field. By iterating over these key-value pairs, the script systematically maps and transforms the response data to align with the processed SRA schema.

Mapping SRA Response Fields

For each key-value pair in the **sraResponseMap**, the script checks if the **key** exists in the **updatedSraSchema**. If it does, the corresponding field title is used to generate a **column name** by converting the title to lowercase. To manage duplicate column names, the script maintains a tracking map called **sraMappedTitles**.

Once the column name is finalized, the script performs **value replacement** if applicable. If the field ID exists in the **sraValueReplacementMap**, the script checks whether the raw response value has a corresponding replacement value (e.g., mapping a numeric code like 1 to a descriptive label such as "High"). If a match is found, the value is replaced with its human-readable equivalent. If no replacement mapping exists, the original value is retained as-is.

The processed response value is then added to the **updatedSraResponse** map under a descriptive key that follows the format: **sra_response_\${finalColumnName}**. This key structure ensures that SRA-specific response data remains unique, identifiable, and ready for integration into the consolidated output.

Flattening SRA Response

After mapping and processing the SRA response fields, the script simplifies the **updatedSraResponse** map by flattening it into a single-level structure. This is achieved using a helper function, such as **flattenJson**, which ensures that nested structures within the response data are converted into a straightforward key-value format. The flattened data is then added to the **flattenedAttributes** map, which consolidates all processed attributes, including general form fields, hazard data, and SRA responses.

To further enrich the output, the script adds SRA-specific metadata to the **flattenedAttributes** map:

- **sra_title**: Stores the title of the SRA.
- **sra_description**: Contains the descriptive text associated with the SRA.

A log entry is generated to confirm the successful addition of SRA details, including the title and description:

"SRA details retrieved and added: title=\${title}, description=\${description}"

This log provides clear visibility into the successful processing of the SRA response and ensures that the enriched data is ready for downstream workflows.

Handling Missing SRA Data

In cases where the query for SRA data does not return any results, the script handles the absence of data gracefully. If the **ResultSet** from the query is empty—indicated by `resultSet.next()` returning `false`—the script logs a message noting that no SRA details were found for the given **sraId**. The log entry is formatted as follows:

```
"No SRA details found for sra_id=${sraId}"
```

This approach ensures that missing data is recorded without interrupting the workflow, providing a clear trace for debugging or auditing purposes. By logging the absence of SRA details, the script allows developers or administrators to identify and resolve potential issues with the input data or database configuration.

Processing Inspection Form IDs and Retrieving Schemas

The script processes **inspectionFormIds**, which is a map containing **dSeries** (field IDs) linked to their corresponding **formId** values. These form IDs represent inspection modules, and the script's primary purpose in this section is to fetch and process schema details for each inspection form. The schema retrieval is guided by parameters like the **form version** (if available) or the **formDate** (if version is missing), ensuring that the inspection form data is enriched accurately.

Checking for Version in JSON Data

For each **dSeries** and **formId** pair in **inspectionFormIds**, the script begins by examining whether the **field data** stored in `jsonData[dSeries]` contains a **version** key. This check determines the appropriate logic for schema retrieval.

The SQL query used to fetch the inspection form schema depends on whether the version is available:

- I. **When version is available:** A straightforward SQL query retrieves the schema by filtering records based on the **formId** and the specific version. This guarantees that the schema fetched matches the exact version of the form as indicated in the input data.
- II. **When version is missing:** A more complex SQL query is constructed to retrieve the schema using the **formId** and the **formDate**. In this scenario, additional logic is applied:
 - a. The query checks whether the **formDate** falls within the range defined by the schema's **publish_date** and **expiry_date**.
 - b. If **publish_date** is null, the script assumes no restrictions and selects the **most recent schema** based on the version number.

This approach ensures that even when the version is not explicitly provided, the most relevant schema is selected based on the form's historical timeline.

PreparedStatement Setup

Once the appropriate query is defined, a **PreparedStatement** is created. The script dynamically binds the required parameters to the query:

- The **formId** is bound as a string parameter.
- Either the **version** or the **formDate** is bound depending on which value is available for the current dSeries.

This parameterized query setup ensures security and efficiency during database execution.

Executing the Query

The prepared statement is executed, and the script processes the resulting **ResultSet** to determine if a matching schema is found.

- **If a matching schema is found:**

The **schema column** from the **ResultSet** is retrieved as a JSON string. This schema is then parsed using **JsonSlurper**, converting it into a structured Groovy object. The parsed schema is stored in the **inspectionSchemas[dSeries]** map, effectively linking the schema to the corresponding dSeries field. This allows the script to seamlessly use the schema for downstream processing, enabling inspection-specific data enrichment.

- **If no schema is found:**

The script does not explicitly log an error. However, the dSeries field remains unmodified in **inspectionSchemas**, and the absence of schema data is handled gracefully without disrupting the workflow.

Error Handling

To ensure robustness, the script includes comprehensive error-handling mechanisms to manage potential issues that might arise during database operations.

- **Database or Query Errors:**

If an exception occurs during the connection setup or query execution, the script logs a detailed error message: "Failed to connect to aero database or execute query".

In such cases, the **FlowFile** is transferred to the failure relationship (**REL_FAILURE**), and the script exits early to prevent further processing of invalid or incomplete data. This ensures that downstream components remain unaffected by errors.

Closing the Database Connection: After the query execution, the script attempts to close the database connection in the **finally block** to release resources. If an error occurs during this operation, it logs a warning message: "Error closing the connection". This structured error-handling approach ensures that all failures are logged clearly, and resources are managed efficiently, minimizing the risk of unexpected behavior.

Iterating Through Headers and Processing Data

The script begins by looping through all headers in the **headers list** to process the corresponding values. Special attention is given to the header labeled **data**, as its value typically contains structured or nested JSON data. The **data** section is significant because it requires additional processing to flatten the nested structures into key-value pairs, making it easier for downstream workflows to consume and analyze the data efficiently.

Iterating Over Keys in jsonData

As the script processes the **jsonData** object, it checks whether each key exists in the **updatedSchema** map. If a match is found, the script retrieves the corresponding column name from the **updatedSchema**. At this stage, the script also fetches additional metadata for the field from **schemaMap.fields** to determine the field's type and its properties. These metadata details play a critical role in guiding the next steps, such as how the value should be transformed, mapped, or flattened.

Handling Non-Annotation Fields

Fields labeled with the type **annotation** are deliberately skipped because they do not contribute to data transformation or downstream outputs. For all other field types, the script standardizes the column names for consistency by converting them to lowercase. These normalized column names are stored as **finalColumnName**, ensuring a uniform structure across all processed attributes.

Processing Field Types

The script employs distinct strategies to process different field types, ensuring the structured data is transformed into a flattened, single-level representation.

Multiselect Fields

For fields that allow multiple selections, where the values are stored as a list, the script processes each item in the list iteratively:

- A unique key is generated for each value in the format: **d_\${finalColumnName}_\${idx}**, where **idx** represents the index of the value within the list.
- The script uses the **valueReplacementMap** to replace internal keys (e.g., numeric codes) with their corresponding human-readable labels. If no mapping exists, the raw value is retained.

Multi-Field Fields

For fields containing multiple **sub-fields**, the script processes each sub-field iteratively:

- The script retrieves the **sub-field schema** (from `multiple_fields`) to map sub-field keys to their respective titles.
- If the sub-field value is itself a nested structure (e.g., a map), the script flattens it recursively to ensure all values are represented in a single-level format.
- Flattened keys are generated in the format: **d_\${finalColumnName}_\${index}_\${subFieldTitle}**.

This approach enables the script to process multi-structured data while maintaining clarity and hierarchy in the flattened output.

Select Fields

For **select** fields, where only a single value is selected, the script uses the **valueReplacementMap** to replace internal values (e.g., numeric keys) with their human-readable equivalents. The flattened attribute is then stored using a key formatted as **d_\${finalColumnName}**. This ensures select fields remain simple, yet informative, in the final output.

Inspection Checklist Fields

For fields associated with **inspection modules**, the script processes the linked schema and checklist details:

- It retrieves the **inspection schema** from the `inspectionSchemas` map based on the field ID.
- The script iterates over each inspection item and matches the **inspection ID** to its corresponding checklist.
- Each checklist item and its response are flattened into descriptive key-value pairs, creating clear and readable output.

Default Fields

For all other field types, including nested JSON structures, the script performs a straightforward flattening operation:

- Nested maps are recursively flattened to produce single-level key-value pairs.
- Flattened keys are generated in the format **d_\${finalColumnName}** for clarity.

This approach ensures that all data, regardless of complexity, is represented consistently and flattened into a format suitable for downstream use.

Cleaning Flattened Attributes

After all fields have been processed and flattened, the script iterates through the **flattenedAttributes** map to ensure data consistency. During this step:

- Any null or empty values are replaced with an empty string ("), standardizing the output and ensuring no invalid or incomplete data propagates further in the workflow.

This final cleaning phase ensures the processed attributes are robust, complete, and ready for downstream processes such as database insertion, reporting, or analysis.

Filtering Flattened Attributes

The script begins by refining the **flattenedAttributes** map to ensure only relevant and meaningful data is retained for further processing. To accomplish this, specific keys are filtered out to clean up unnecessary or irrelevant entries. These include keys that:

- Start with the prefix **index_key_**, which are typically temporary or internal keys not required in the final dataset. The filtering is performed in a **case-insensitive** manner to ensure comprehensive cleanup.
- Exactly match keys like **form_parent_id**, **form_version_id**, or **parent_key**, as these identifiers are already handled elsewhere in the workflow and do not need to remain in the flattened attributes.

After this filtering step, the **flattenedAttributes** map contains only meaningful and relevant data, reducing noise and preparing the attributes for subsequent transformations.

Preparing Utility Functions

To ensure smooth and standardized handling of attribute names, the script includes two utility functions:

- I. **Quote Identifier Function:** This function ensures safe handling of attribute names, particularly for database interactions. It converts attribute names to **lowercase**, escapes existing double quotes by doubling them, and wraps the result in double quotes. This safeguards column names, ensuring they are valid and free from syntax errors when interacting with databases.
- II. **Truncate Column Name Function:** Given that many databases enforce column name length limits (commonly **63 characters**), this function ensures compliance by truncating attribute names that exceed the maximum length. Names are converted to lowercase, and truncation is performed while preserving the uniqueness of the column names.

These utility functions are essential for maintaining a clean, safe, and database-compliant structure in the flattened attributes.

Ensuring Unique Column Names

To prevent conflicts caused by duplicate or excessively long column names, the script ensures that all attribute keys in **flattenedAttributes** are unique. This is achieved through the following process:

- A new map called **uniqueFlattenedAttributes** is initialized to store the processed attributes with unique column names.
- A **columnNameMap** is created to maintain a mapping between the unique column names and their original keys, ensuring traceability.

The script iterates over each key-value pair in **flattenedAttributes**:

- **Truncating Key Names:** Keys are truncated to comply with the 63-character limit using the `truncateColumnName` function.
- **Storing Unique Keys:** The script updates the **columnNameMap** with the newly generated unique key and adds the key-value pair to **uniqueFlattenedAttributes**.

As a result, the script produces a clean and conflict-free version of the attributes, ready for downstream use without concerns about duplicate or invalid column names.

Logging Final Attributes

Once the attributes have been processed and standardized, the script logs the final map of **unique attributes**. Logging this information serves two key purposes:

- It allows verification of the integrity and correctness of the processed data.
- It provides a valuable debugging tool in case issues arise downstream in the workflow.

The log acts as a checkpoint, ensuring transparency into the state of the flattened attributes before they are serialized and used further.

Preparing Flattened Attributes for Output

The next step involves preparing the **uniqueFlattenedAttributes** map for output. The map is serialized into a JSON string using **JsonOutput.toJson**, producing a structured representation of the attributes. This serialized JSON, stored in the variable **flattenedAttributesJson**, is ideal for integration into workflows that require structured and readable data formats. It can be stored, transferred, or further processed without requiring additional transformations.

Adding Metadata to the FlowFile

The script enriches the **FlowFile** with key metadata attributes that are critical for downstream workflows. These attributes include:

- **Form_Answer_id:** Represents the unique identifier of the form answer.
- **flattenedAttributes:** Contains the flattened attributes as a serialized JSON string, ensuring all transformed data is available in a structured format.
- **tableName:** Specifies the database table name associated with the form, providing context for data storage.
- **airport_id:** Captures the airport ID, linking the data to its geographical or operational context.
- **form_version_id:** Includes the version ID of the form to maintain version-specific integrity in the workflow.

By embedding these metadata values into the FlowFile, the script ensures that all relevant information travels with the data, enabling smooth handoff to subsequent stages of the pipeline.

Transferring the FlowFile

After enriching the FlowFile with flattened attributes and metadata, the script transfers it to the **REL_SUCCESS** relationship. This step marks the FlowFile as successfully processed and ready for the next stage in the workflow.

Transferring the FlowFile to REL_SUCCESS is an indication of completion, ensuring seamless progression through the workflow pipeline.

Handling Data Parsing

The script also includes a **parseValue** function to handle the parsing of raw values in the dataset. This function attempts to safely parse values into JSON objects, ensuring that malformed data does not interrupt the workflow. If parsing fails, the function gracefully returns the original value. Additionally, to ensure compliance with JSON standards, single quotes (') in the raw data are replaced with double quotes ("). This adjustment guarantees that the data remains robust and compatible with JSON-based systems.

The **parseValue** function plays a crucial role in managing inconsistent or potentially malformed data, providing resilience to the overall script execution.

Purpose of the Functions

The functions **flattenJson** and **flattenNestedJson** are specifically designed to simplify hierarchical or nested data structures by transforming them into flat, single-level key-value pairs. This flattening process makes JSON data more manageable and compatible with systems that require straightforward, non-nested formats, such as relational databases, analytics pipelines, or integrations with other processing systems. By expanding complex structures into unique and descriptive keys, these functions improve data accessibility and usability for downstream workflows.

Flattening Logic in flattenJson

The **flattenJson** function handles a variety of data types—maps, lists, and primitive types—ensuring all nested levels of the input structure are flattened recursively.

Processing Maps

A **map** represents a collection of key-value pairs where keys are strings and values can range from simple types (e.g., strings, numbers, booleans) to more complex structures like nested maps or lists.

- When the input data is a **map**, the function evaluates its content:
 - If the map is **empty**, the prefix (which acts as the current key name) is mapped to an empty JSON object ({}), representing an absence of content in a structured way.
 - If the map is **non-empty**, the function iterates over each key-value pair within the map. For each pair:
 - A new **prefix** is generated by appending the current key (converted to lowercase for consistency) to the existing prefix, separated by an underscore.
 - The function calls itself recursively with the value associated with the current key. This recursive approach ensures that any further nested maps or lists are processed and flattened appropriately.
 - The flattened results from all recursive calls are merged into the final result map, preserving the hierarchical structure through unique, descriptive keys.

Processing Lists

A list is an ordered collection of values that may contain elements of any data type, including maps or other lists.

- When the input data is a **list**, the function evaluates its content:
 - If the list is **empty**, the prefix is mapped to an empty JSON array ('[]'), maintaining clarity about the absence of list elements.
 - If the list is **non-empty**, the function iterates over each value in the list. For each value:
 - The current index of the value is appended to the prefix, ensuring each element in the list is assigned a unique key.
 - The function is called recursively with the current list element, enabling further flattening of any nested structures within the list.

The result is a flattened representation of the list where each element is uniquely identified and appropriately prefixed, maintaining both order and hierarchy in a simplified form.

Processing Other Data Types

For primitive data types such as strings, numbers, or booleans, the function handles them directly without any further recursion. If the input data is neither a map nor a list:

- The current prefix is directly mapped to the corresponding data value.

The result of the **flattenJson** function is a fully flattened map where all nested structures—whether maps, lists, or other data types—are expanded into unique keys. Each key reflects its position and hierarchy in the original data structure, creating a single-level map that retains the depth and context of the input JSON.

Flattening Logic in flattenNestedJson

The **flattenNestedJson** function is like **flattenJson** but focuses primarily on **maps** and treats lists as atomic, non-recursive values. This distinction makes it particularly effective for use cases where only nested maps need to be expanded, while lists are preserved as single units.

Results of the Functions

The result of the **flattenJson** function is a fully flattened map where every element—regardless of depth or complexity—is represented as a unique key-value pair. The keys are descriptive, combining prefixes and indices to retain the hierarchy and order of the original JSON structure while simplifying its representation into a single level.

The result of the **flattenNestedJson** function, on the other hand, focuses solely on expanding **nested maps**, treating lists as standalone values. This approach ensures that deeply nested maps are flattened, while lists remain intact, preserving a balance between simplification and structural integrity.

Both functions play a vital role in preparing hierarchical data for workflows that require flat structures, such as database storage, data pipelines, or reporting systems. By transforming complex JSON data into simple, accessible key-value formats, these functions ensure compatibility, ease of use, and efficiency across downstream processes.

Data Approval Mapping

Importing Required Libraries

The script begins by importing essential libraries required for efficient data handling, JSON processing, and database operations. **IOUtils** simplifies operations related to input and output streams, such as converting streams to byte arrays or reading content efficiently. **JsonSlurper** enables easy parsing of JSON strings into Groovy objects like maps or lists, facilitating seamless traversal and manipulation of JSON data. Additionally, **JsonOutput** converts Groovy objects into JSON strings, making it convenient to serialize structured data for storage or further processing. For database operations, **DriverManager**, **Connection**, **PreparedStatement**, and **ResultSet** form the core of JDBC components. **DriverManager** manages database connections, while **Connection** establishes the connection to the database. **PreparedStatement** allows the execution of parameterized SQL queries, ensuring enhanced security and performance. **ResultSet** helps in row-by-row retrieval of query results.

To handle input streams efficiently, **BufferedReader** and **InputStreamReader** are utilized. **InputStreamReader** converts raw byte streams into character streams, while **BufferedReader** adds buffering for efficient memory usage, particularly when dealing with large datasets. The **StreamCallback** utility from NiFi is employed to define callbacks for reading and writing FlowFile content, enabling efficient processing of large data streams within the workflow.

FlowFile Retrieval

Now that the necessary libraries are in place, the script retrieves a FlowFile from the NiFi session using `session.get()`. A FlowFile is a core data object in NiFi, representing the unit of data being processed. If no FlowFile is available, the script terminates early with `return`. This check ensures the script does not continue executing unnecessarily when there's no data to process, making it more efficient. The FlowFile object is essential for interacting with the data being passed through the NiFi pipeline, and this retrieval step ensures that there is data to work with before proceeding to the next steps in the script.

Attribute Retrieval and Validation

Retrieving Attributes

The script retrieves two important attributes, `formAnswerId` and `formVersionId`, from the FlowFile using `flowFile.getAttribute()`. These attributes are essential for identifying the specific form and its version for further processing.

Validating Attributes

The script checks if both `formAnswerId` and `formVersionId` are present. If either of these attributes is missing, the script will log an error and move the FlowFile to the failure relationship. This validation ensures that the necessary information is available for the next steps in the process.

Error Logging and Transfer to Failure

If the validation fails (i.e., one or both attributes are missing), the script logs an error and transfers the FlowFile to the failure relationship (REL_FAILURE). The script terminates early to avoid further processing of invalid data.

This structure ensures that the script efficiently handles the missing attributes, providing appropriate error logging and FlowFile redirection.

Defining Database Connection Parameters

The script defines three variables for the database connection: `dbUrl`, `dbUser`, and `dbPassword`. These variables hold the database URL, username, and password, respectively, which are crucial for connecting to the database. The values for these parameters are provided as placeholders (e.g., `{Source_DB_URL}`), which will be replaced with actual values at runtime.

Declaring the Connection Object

A Connection object, named `connection`, is declared. This object will be used to establish and manage the connection to the database. Initially, it is set to null, as the actual connection will be created later in the script.

Preparing for Database Interaction

With the database connection parameters defined and the connection object declared, the script is now prepared to interact with the database. The next steps (not yet shown) would involve establishing the connection using the `DriverManager` and executing SQL queries to retrieve or update data.

This setup ensures that the necessary credentials are in place and the script is ready to work with the database securely.

Database Query Execution and Data Mapping

Initializing Maps for Data Storage

The script begins by initializing two empty maps: `inspectionSchemas` and `inspectionFormIds`. These maps will later be used to store data related to inspection schemas and form IDs, respectively. They will help in organizing the data retrieved from the database.

Establishing the Database Connection

The script proceeds to establish a connection to the database using the previously defined parameters (`dbUrl`, `dbUser`, `dbPassword`). The `DriverManager.getConnection()` method is called, and if successful, the connection object is populated with an active database connection.

Defining the SQL Query

Next, a SQL query is defined within the script. This query selects various fields from multiple tables (data_forms_formanswer,data_forms_formprocess,data_forms_formprocess_approvals,data_forms_formapprovals,data_forms_formapprovaldef etc.) by joining them on specific conditions. The query retrieves approval-related information, such as approval_id, approval_data, approval_step_id, completed_date, status, and approvals_schema. The query filters results based on the form answer ID (dff.id = ?), which will be dynamically passed later in the script, this query will retrieve multiple formapprovals data with different step_id related with same formAnswerId, if we have multiple steps with same step name we are taking step based on recent completed_date column.

Query Execution and Data Retrieval

Following the SQL definition, the script would use the PreparedStatement to bind the dynamic parameters (e.g., form answer ID) and execute the query. The results would be captured in a ResultSet, which will be processed to extract and map the data into the inspectionSchemas and inspectionFormIds maps for later use.

The combination of connection setup and query preparation ensures that the script is ready to retrieve approval data from the database based on the specific form answer ID.

Executing the SQL Query

Preparing the SQL Statement

The script uses the connection.prepareStatement() method to prepare the SQL query defined earlier for execution.

connection.prepareStatement(sql): This prepares the SQL query for execution by the database. It returns a PreparedStatement object that is pre-compiled for optimal performance, and it can accept parameters dynamically

Setting the Parameter for the SQL Query

Before executing the query, the script needs to replace the placeholder? in the SQL query (defined in the previous step) with the actual form answer ID. This is done by setting the value using the setInt() method on the preparedStatement.

preparedStatement.setInt(1, ...): The setInt() method sets the value of the first parameter (?), which is the form answer ID. formAnswerId.toInteger() converts the formAnswerId (which was retrieved as a string from the FlowFile attribute) into an integer type, as expected by the SQL query.

Executing the Query and Retrieving the Results

After setting the parameter, the query is executed using the `executeQuery()` method, which retrieves the result set from the database.

`preparedStatement.executeQuery()`: Executes the SQL query and returns a `ResultSet` object containing the data returned by the query. `ResultSet` is an object that holds the result of the executed SQL query, and the script will use this `ResultSet` to fetch the actual data (like `approval_id`, `approval_data`, etc.) for further processing. This completes the database query execution process, and the results can now be processed and stored as needed.

Processing Query Results

Initializing Data Containers:

The script begins by defining three variables to store the retrieved data:

- I. **`approvalsList`:** An empty list that will hold approval data for each row in the `ResultSet`.
- II. **`approvalsSchema`:** This is initialized as null and will later hold the schema for the approval data.
- III. **`approvalStatus`:** Initialized as null, this will store the status of the approval.

Iterating Through the ResultSet

The `while (resultSet.next())` loop starts to process each row in the `ResultSet`. For each row, the following steps occur:

- I. **`approval_id`:** Extracts the approval ID (`approval_id`) from the `ResultSet`. **`approval_data`:** Fetches the approval data (`approval_data`) as a JSON string and parses it using `JsonSlurper` into a Groovy map.
- II. **`approval_step_id`:** Gets the approval step ID (`approval_step_id`) from the `ResultSet`.
- III. **`completed_date`:** Extracts the completion date (`completed_date`) from the `ResultSet`. Each of these values is added to the approval map, which is then added to `approvalsList`.

Assigning Approval Status and Schema

- I. **`approvalStatus`:** If not already set, the status (`status`) of the approval is retrieved from the `ResultSet` and assigned to `approvalStatus`. This is done once for the first row.
- II. **`approvalsSchema`:** Similarly, the schema for approvals (`approvals_schema`) is retrieved from the `ResultSet` and parsed into a Groovy map using `JsonSlurper`. This is done only once.

Closing Database Resources

Finally, once all rows are processed: `resultSet.close()`: Closes the `ResultSet`, freeing resources.
`preparedStatement.close()`: Closes the `PreparedStatement`, ensuring the database connection is properly managed.

Handling Missing Approvals Schema and Mapping Inspection Forms

Checking for Missing Approvals Schema

The script first checks if `approvalsSchema` is null (i.e., no approvals schema was retrieved from the database). If this is the case:

Logging a Message: An info log is recorded that the approvals schema for the current `formVersionId` is missing, and as a result, the script skips approval mapping. This helps in tracking the state of the process in case the data is incomplete.

Processing Approval Schema if Found If the `approvalsSchema` exists:

Iterating Through Steps: The script iterates over each step in the `approvalsSchema`. For each step, it further iterates over fields associated with that step.

Checking Field Type and Module: For each field, it checks if the field's type is 'inspection_checklist' and if a `selected_module` is specified.

If both conditions are met, the script retrieves the `formId` from the `selected_module.key` and maps it to the `field.id` in the `inspectionFormIds` map. This is used to track form IDs related to inspection checklists. This logic ensures that the form IDs for inspection-related fields are stored for further use, and it handles the scenario where the `approvalsSchema` is missing in a way that does not break the script's flow.

Iterating Over Inspection Form IDs and Extracting Version Data

Iterating Through Inspection Form IDs

The script iterates over the `inspectionFormIds` map, where, `dSeries` represents the field ID for the inspection checklist. `formId` is the associated form ID that corresponds to the selected module in the schema.

Extracting Version Data

For each `dSeries` (field ID) and `formId`, the script checks the following:

It accesses the `approval_data` map in the first approval object in `approvalsList`. It checks if the `approval_data[dSeries]` is an instance of `Map` (ensuring the field is structured as expected). Then, it checks if

this map contains a key called 'version'. If the version key is found, it extracts the version value from the approval_data associated with the dSeries.

This step ensures that if a version exists in the approval data for the specified field ID (dSeries), it is extracted and stored in the version variable for further processing.

Retrieving Inspection Schema Data

Defining SQL Query and PreparedStatement for Inspection Data

The script defines a new inspectionSql variable to hold the SQL query for fetching the schema from the inspections_inspection table based on the formId and version.

The script's first logic focuses on retrieving the inspection schema when a **specific version** of the form is provided. If the version exists, the script directly uses the **form ID** and the provided version number (which is equal to 'number' column value in inspections_inspection table) to pinpoint the corresponding schema in the database. This approach assumes that the input data explicitly specifies the exact version required, simplifying the query execution. By combining these parameters, the system retrieves the precise schema linked to the specified version, ensuring accuracy and consistency. This logic is efficient because it eliminates ambiguity, directly fetching the relevant schema without additional conditions or iterations.

The second logic comes into play when no specific version is provided. In such cases, the script determines the appropriate schema by checking the form's **completion date** against its publishing and expiry timelines. This date-based logic ensures that the schema retrieved corresponds to the timeframe during which the form was completed. The script evaluates whether the completion date falls within the range defined by the schema's **publish date** and **expiry date**. If no expiry date exists, the system defaults to using the current date as the upper limit.

if the **publish_date** is NULL, the system prioritizes the **number** column to determine which schema to select. This logic is applied in the **ORDER BY** clause, where the query ensures that schemas without a **publish_date** are considered first and ordered based on the **number** column in descending order and limit 1 (which is equal to maximum value of 'number' column).

Preparing the SQL Statement

For the versioned case, the script sets the formId (converted to string) and version (converted to integer) as parameters in the prepared statement using inspectionPreparedStatement.setString(1,formId.toString()) and inspectionPreparedStatement.setInt(2, version.toInteger()).

For the non-versioned case, it sets formId and completed_date from the approvalsList to check if the approval date falls within the publish date range.

Executing the Query

The script then uses `inspectionPreparedStatement.executeQuery()` to execute the SQL query and retrieve the result into `inspectionResultSet`.

Parsing the Result

Inside the try block, the script checks if `inspectionResultSet.next()` returns data. If data exists, the schema (stored as a JSON string) is retrieved from the result set using `inspectionResultSet.getString("schema")` and parsed into a Groovy object using `JsonSlurper().parseText()`.

Storing the Inspection Schema

The parsed `inspectionSchemaJson` is then stored in the `inspectionSchemas` map using the `dSeries` as the key. This ensures that the inspection schema is associated with the specific field ID (`dSeries`).

Closing the Resources

The try block ensures that the `inspectionPreparedStatement` and `inspectionResultSet` are automatically closed after use, preventing any resource leaks.

Handling Duplicate Titles

To ensure field titles remain unique, the script tracks occurrences of each title using the `titleOccurrences` map:

- **Check for Title Existence:**
 - If the title is not already in `titleOccurrences`, it is added with an initial count of **1**, and the field ID is mapped directly to the title in `mappedData`.
 - If the title already exists in `titleOccurrences`, its count is incremented. To ensure uniqueness, the script appends the field ID (`D*`) to the title (e.g., `"field_id_title"`) and updates the entry in `mappedData`.

Mapping Data from Approval Data to Schema Fields

Iterating Through Approvals

The script begins by iterating over each approval in `approvalsList`. For each approval, it attempts to find the corresponding `stepSchema` in `approvalsSchema.steps` using the `approval.approval_step_id` as the matching key.

Step Schema Lookup: The script uses `find { it.id == approval.approval_step_id }` to find the matching step schema in the `approvalsSchema`.

Verifying the Existence of Fields

Once the correct stepSchema is found, the script checks if stepSchema.fields exists. If this is true, it proceeds to map the data for that particular approval.

Field Mapping: A new mappedData map is initialized to store the mapped data for the approval.

Iterating Over Fields

For each field in stepSchema.fields, the script checks if the approval.approval_data contains the key corresponding to the field.id. If the field exists in the approval data, it extracts its value into fieldValue.

Handling Inspection Checklist Fields

If the field.type is 'inspection_checklist' and there is a corresponding entry in inspectionSchemas, the script proceeds to process the checklist data:

- I. Inspection Schema Lookup: It retrieves the inspectionSchema for the field from inspectionSchemas[field.id].
- II. Inspection Data Parsing: If fieldValue is a Map and inspectionSchema.fields exist, the script iterates over the fieldValue (which is assumed to be a map of inspection IDs to inspection data).
- III. Checklist Mapping: For each inspectionId and inspectionData, it tries to find the corresponding checklist item in the inspectionSchema.fields.
- IV. Extracting Inner Data: If the checklist contains valid items, the script maps the inner values of each checklist item into mappedData using a dynamic key based on the field and checklist details (finalKey).

Handling Multi-Field Data

If the field.type is 'multi_field' and fieldValue is a List, the script assumes the field contains multiple subfields, each of which needs to be processed:

Subfield Processing: It iterates over each multiFieldValue in the list and then over each subkey and subvalue in multiFieldValue. Subfield Schema Lookup: It retrieves the corresponding subfield schema from field.multiple_fields and maps the subfield value to a key in mappedData.

Handling Select and Multiselect Fields

If the field.type is 'select' or 'multiselect', the script processes these fields as follows:

- I. **Select Field:** For a select field, the script finds the corresponding valueMapping in field.values and stores the mapped value in mappedData.
- II. **Multiselect Field:** For a multiselect field, if fieldValue is a list, it collects the mapped values for each selected option and stores them in mappedData.

Default Field Handling

If none of the specific field types (inspection checklist, multi-field, select, multiselect) match, the script stores the fieldValue directly in mappedData under the field's title.

Updating Approval Data

Once all fields are processed and mapped, the script updates approval.approval_data with the mappedData for that approval. This ensures that the approval data is transformed and flattened according to the schema.

Retrieving the Flattened Attributes JSON

The script retrieves the flattenedAttributesJson from the FlowFile attributes using flowFile.getAttribute('flattenedAttributes'). This attribute is expected to contain the flattened attributes data, typically in JSON format.

FlowFile Attributes: The flattenedAttributes are accessed as an attribute of the FlowFile, which is a key-value pair storage in NiFi, where FlowFiles can carry metadata and data attributes.

Checking for Missing Flattened Attributes: Next, the script checks whether the flattenedAttributesJson is null or empty. If this attribute is missing or not available, an error message is logged to indicate that the flattened attributes are missing.

Error Handling: The log.error function logs a message that includes the failure reason, in this case, that flattenedAttributes are missing.

Transferring FlowFile on Failure

If the flattenedAttributesJson is missing, the script transfers the FlowFile to the REL_FAILURE relationship, signaling a failure in processing. This helps in managing the FlowFile flow within NiFi, and it can be reprocessed or analyzed later.

Failure Path: Using session.transfer(flowFile, REL_FAILURE) ensures that the FlowFile will not be processed further and will be routed to a failure state in the NiFi flow.

Early Return if Missing: Finally, the script returns early if the flattenedAttributesJson is missing. This prevents the script from continuing and attempting further operations that depend on the flattenedAttributes. The return statement exits the script.

Flattening and Storing Attributes

Parsing Flattened Attributes JSON

The script begins by parsing the flattenedAttributesJson string (which was retrieved earlier) into a Groovy map using the JsonSlurper.

JsonSlurper: The JsonSlurper().parseText() method is used to parse the JSON string and convert it into a Groovy map. The resulting flattenedAttributes will contain the existing key-value pairs that need to be populated or modified during the processing.

Iterating Over Approvals and Flattening Data

Iterating Over Approvals: The approvalsList.each loop iterates over each approval in the list, and for each approval, it accesses the approval_data.

Flattening Data: The flattenJson function is called for each field in approval_data, which flattens nested data into a simpler key-value format. It adds a prefix ("ad_\${key}") to each field's key, helping to differentiate the flattened attributes from others.

Updating Flattened Attributes: The flattened data for each key-value pair is then added to the flattenedAttributes map. If there are nested fields, flattenJson ensures they are stored with a unique key, preserving the hierarchy but making it easier to handle.

Storing Approval Status

Finally, the approvalStatus value (which was set earlier from the database query result) is added to the flattenedAttributes map

Adding Approval Status: This step ensures that the approval status is included in the flattenedAttributes map, which is necessary for passing the final status information downstream.

Truncating Column Names and Preparing Data

Initializing the Truncated Attributes Map

The script begins by creating an empty map `truncatedFlattenedAttributes` to hold the modified keys

truncatedFlattenedAttributes: This map will store the flattened attributes after their keys have been processed (i.e., truncated).

Iterating Over Flattened Attributes

The script then iterates over the `flattenedAttributes` map (which contains all the flattened data) using each method

Iterating Over Keys and Values: For each key-value pair in `flattenedAttributes`, the script processes the key and applies a truncation method to it.

Converting Key to Lowercase: The `key.toLowerCase()` method ensures that the key is in lowercase before truncating, making the key uniform for further processing.

Truncating the Key: The `truncateColumnName` function is called to truncate the key. This ensures that if a column name exceeds a certain length or has any special formatting requirements (such as removing unsupported characters), it will be handled accordingly.

Storing Truncated Keys and Values

The truncated key and its associated value are then stored in the `truncatedFlattenedAttributes` map

Storing in the Map: The new key-value pair with the truncated key is added to `truncatedFlattenedAttributes`. This ensures that all keys are processed according to the required length/format before they are used downstream.

Transferring FlowFile and Handling Exceptions

Successfully Processing the FlowFile

Once all the steps are executed without any exceptions, the FlowFile is transferred to the `REL_SUCCESS` relationship

REL_SUCCESS: This relationship indicates that the script has successfully completed all tasks and the FlowFile is ready for further processing downstream. `session.transfer(flowFile, REL_SUCCESS)`: This line tells NiFi to move the FlowFile to the "success" path, signaling that the data processing was successful.

Catching Exceptions

If any errors occur during the execution of the database query, connection, or other parts of the script, the catch block will handle the exception

Exception Handling: The catch block captures any Exception (errors such as database connectivity issues, SQL exceptions, or any other runtime issues) and logs the error message.

Logging the Error: The log.error method logs the error message and the exception stack trace, making it easier to debug.

REL_FAILURE: If an error occurs, the FlowFile is transferred to the REL_FAILURE relationship. This signals to NiFi that the processing has failed, and the file can be handled according to the failure path (e.g., retry, alert, etc.).

Closing the Database Connection: Finally, whether the script succeeds or fails, the finally block ensures that the database connection is properly closed.

Ensuring Connection Closure: This block ensures that if the connection object was successfully created and is still open, it will be closed after the script execution ends. **Prevention of Connection Leaks:** Closing the connection ensures that database resources are freed, preventing connection leaks which can affect system performance or stability over time.

Flattening JSON Data

The flattenJson function is used to flatten nested JSON data into a simpler, key-value pair format, which is more manageable in NiFi workflows. It transforms hierarchical structures like maps and lists into a flat structure with composite keys, making it easier to store or pass the data downstream.

Purpose of the Method

The method aims to flatten a Map or List structure by recursively processing each element and creating a new map where nested objects are flattened by concatenating keys using a delimiter (in this case, an underscore _). The final map contains a flattened view of the original nested data, which is useful for storing in databases, creating reports, or sending as JSON data in downstream processes.

Function Breakdown

Flattening Function: The flattenJson function takes two parameters: data: The data to be flattened, which can be a Map, List, or other primitive types. prefix: A string used to prefix keys in the flattened structure. It starts as an empty string but accumulates key names as the recursion deepens.

Processing Maps: If the data is a Map, the function iterates over each key-value pair

Recursive Call: For each key-value pair, the function calls itself (`flattenJson(value, newPrefix)`), appending the current key to the prefix. The keys are converted to lowercase to maintain consistent naming conventions. **Merging Results:** The `putAll()` method is used to add the flattened results from the recursive call into the result map.

Processing Lists: If the data is a List, the function iterates over each element with an index: The index is appended to the prefix to differentiate between elements in the list, ensuring that each flattened key is unique (e.g., `field_0`, `field_1`, etc.). **Recursive Call:** It recursively flattens each element of the list.

Base Case: If data is neither a Map nor a List (i.e., a primitive value like a String, Integer, or Boolean), it adds the current prefix as the key and the data as the value in the result map.

Returning Result: After all recursion is completed, the flattened Map (result) is returned.

DataBase and Table creation

Importing Required Libraries

The script begins by importing essential libraries required for efficient data handling, JSON processing, and database operations. **JsonSlurper** parses JSON strings into Groovy objects like maps or lists. Enables seamless processing of JSON data retrieved from the database, making it easier to access and manipulate individual fields. This is especially useful when the script needs to work with nested or complex JSON structures. **DriverManager** manages database connections. Allows the script to establish a connection to a relational database by specifying a connection URL, username, and password. Without this, the script would not be able to interact with the database to fetch or manipulate data. **Connection** represents a persistent connection to the database. This object ensures that the database is accessible for the script to execute SQL queries. All subsequent database operations rely on this connection to remain open and valid. **PreparedStatement** facilitates the execution of parameterized SQL queries. Ensures secure and efficient interaction with the database by preventing SQL injection and optimizing query performance. This component is critical when the script involves dynamic SQL queries. **ResultSet** represents the output of an executed SQL query. Used to store and iterate through the data retrieved from the database, enabling the script to extract and use specific fields in further processing. **SQLException (from org.postgresql.util.SQLException)** specialized exception class for PostgreSQL database errors. Helps the script handle database-specific issues, such as connection failures or SQL syntax errors, ensuring robust error management during database interaction.

FlowFile Retrieval and Validation

Retrieving a FlowFile from the Session

The script attempts to retrieve a FlowFile from the current NiFi session using the `session.get()` method. A FlowFile is a key concept in NiFi, representing a unit of data that is processed through the system. By calling this method, the script checks whether there is a FlowFile available for processing. If a FlowFile is found, it is assigned to the variable `flowFile`. If no FlowFile is present, the variable will be null.

Checking if the FlowFile Exists

The script then checks if the `flowFile` variable is null using a conditional statement. If `flowFile` is null, this means that there is no FlowFile available to process, and the script will terminate early. The early return prevents the script from attempting to process non-existent data, thus avoiding errors or unnecessary operations.

Extracting FlowFile Attributes and Parsing JSON

Extracting FlowFile Attributes

The script extracts certain attributes from the FlowFile. These attributes are key-value pairs that provide metadata or additional information about the FlowFile's content. The three attributes extracted are:

- I. **tableName:** This attribute holds the name of the table associated with the FlowFile's data. It is retrieved using `flowFile.getAttribute('tableName')`.
- II. **flattenedAttributes:** This attribute contains JSON data in a string format, which represents a set of flattened attributes. It is retrieved using `flowFile.getAttribute('flattenedAttributes')`.
- III. **airport_id:** This attribute holds the identifier for the airport related to the data in the FlowFile, retrieved using `flowFile.getAttribute('airport_id')`. These attributes are stored in variables (`tableName`, `flattenedAttributesJson`, `airportId`) for use later in the script.

Parsing the Flattened Attributes JSON

After extracting the `flattenedAttributes` JSON string, the script parses it using `JsonSlurper`. This is done with the line `new JsonSlurper().parseText(flattenedAttributesJson)`. The `JsonSlurper` parses the JSON string and converts it into a Groovy object, typically a map, which allows for easy traversal and manipulation of the data. By converting the JSON into a Groovy object, the script can work with individual fields or perform necessary transformations on the data.

Database Connection Parameters

Defining Database Connection Parameters

In this section, the script defines the database connection parameters that will be used to establish a connection to the target database. These parameters are stored as variables and will be referenced later in the script for connecting to the database.

- I. **dbUrlBase:** This variable holds the base URL of the target database. It uses the placeholder `{Target_DB_URL}`, which will be replaced with the actual database URL at runtime. This URL typically includes the database hostname, port, and the database name.
- II. **dbUser:** This variable holds the username required for authenticating the database. It uses the placeholder `{Target_DB_User}` for runtime substitution with the actual database user.
- III. **dbPassword:** This variable stores the password for the database user, using the placeholder `{Target_DB_Pwd}`. This will be substituted with the actual password at runtime.

Connecting Database Parameters to the Script

At this point in the script, the necessary database parameters (`dbUrlBase`, `dbUser`, and `dbPassword`) have been defined. These parameters will be used to establish a connection to the database, enabling the script to interact with the database to retrieve or manipulate data.

For example, after extracting and parsing the flattened attributes, the script can proceed to use these parameters to connect to the database and perform operations like fetching approval data or inserting information into the target database.

Database Connection and Table Management Process

Constructing the Airport-Specific Database URL

The script constructs the database URL for a specific airport by appending the `airportId` to the base database URL (`dbUrlBase`). This dynamically generates the connection string that targets the specific airport's database, allowing the script to work with different airport databases based on the `airportId`.

dbUrlAirport: The concatenated URL for the airport-specific database, formed by appending the `airportId` to the base URL.

Establishing the Connection to the Database

The script uses a try-catch-finally block to manage the database connection and ensure it is closed properly. `connectionAirport = DriverManager.getConnection("${dbUrlBase}postgres", dbUser, dbPassword)`: Initially, the script connects to the postgres database (typically used to manage databases and user access). This is done using the parameters defined earlier (`dbUrlBase`, `dbUser`, `dbPassword`).

Verifying Database Existence

Once the connection to the postgres database is established, the script checks if the airport-specific database exists.

`checkDatabaseExists(connectionAirport, airportId)`: This method is used to verify whether the airport-specific database exists. If the database doesn't exist, the script will proceed to create it.

Creating Database and Initializing Tables

If the airport-specific database does not exist, the script creates the database and initializes it. `createDatabase(connectionAirport, airportId)`: Creates the airport-specific database. After the database creation, the script reconnects to the newly created database (`connectionAirport = DriverManager.getConnection(dbUrlAirport, dbUser, dbPassword)`) and creates the necessary table and partition.

Reconnecting to an Existing Database

If the airport-specific database already exists, the script connects directly to it using the previously defined `dbUrlAirport`.

Checking for the Existence of the Table

The script then checks whether the specified table exists within the airport-specific database. `checkTableExists(connectionAirport, tableName)`: If the table doesn't exist, it is created along with the necessary partition using the `createTable` and `createPartition` methods.

Handling Existing Tables and Columns

If the table exists, the script compares the existing columns with the flattened attributes to ensure that any missing columns are added. `getExistingColumns(connectionAirport, tableName)`: This method retrieves the existing columns in the table. `addMissingColumns(connectionAirport, tableName, flattenedAttributes, existingColumns)`: If any columns are missing, they are added to the table to align with the structure of the flattened attributes.

Ensuring Partitions Are Created

Regardless of whether the table already exists or was newly created, the script ensures that a partition is created based on the `updated_at` field in the flattened attributes.

`createPartition(connectionAirport, tableName, flattenedAttributes["updated_at"])`: Creates a partition for the table based on the `updated_at` field, ensuring efficient data storage and retrieval.

Successful FlowFile Transfer

Once the database and table operations are completed, the script transfers the FlowFile to the success relationship (`REL_SUCCESS`), indicating that the processing was successful.

Error Handling and Logging

If any exception occurs during the database connection or operations, the script logs an error message and transfers the FlowFile to the failure relationship (`REL_FAILURE`).

Ensuring Proper Connection Closure

In the finally block, the script ensures that the database connection is closed properly, whether the operations succeed or fail.

`connectionAirport.close()`: This ensures that the database connection is closed to prevent resource leaks.

Checking Database Existence

Purpose of the Helper Function

The helper function is designed to check whether a specific database exists in the PostgreSQL server. It accomplishes this by querying the system catalog of PostgreSQL to see if the database name is listed.

Query Execution and Parameterization

The function begins by preparing a SQL query to check for the existence of the database. The query used searches the `pg_database` system catalog for a record where the `datname` matches the provided database name.

The query is parameterized, meaning the database name is passed as a parameter to prevent SQL injection attacks and ensure safe execution. The prepared statement (`preparedStatement`) is used to execute this query.

Checking the Query Result

The result of the query is stored in a `ResultSet`. This object contains the rows returned by the query, which in this case, would only contain a single row if the database exists.

`resultSet.next()`: This method is called to check if the query returned any rows. If it returns `true`, it means the database with the specified name exists; otherwise, it does not.

Closing Resources

Once the check is performed, the function ensures that both the `ResultSet` and `PreparedStatement` are closed properly to release resources.

Closing the `ResultSet` and `PreparedStatement` is crucial for avoiding memory leaks and ensuring that the database connection remains efficient.

Returning the Result

Finally, the function returns a boolean value, `exists`, which indicates whether the database was found. If the database exists, `true` is returned; otherwise, `false`.

Creating a Database

Purpose of the Helper Function

The function is responsible for creating a new database on the PostgreSQL server. It uses the provided connection to execute a SQL command that generates the new database.

Executing the SQL Query

The function constructs a SQL query that creates a new database with the name provided (dbName). The query is dynamically formed by embedding the database name into the CREATE DATABASE SQL command. The database name is passed as a parameter to ensure that the correct database is created.

Preparing and Executing the Statement

After preparing the query, the function executes it using executeUpdate(). This method is used because it performs a modification operation (in this case, creating a new database) rather than a query that returns data.

Closing the Statement

After the query is executed, the prepared statement is closed to free up resources and ensure the connection remains clean.

Logging

Once the database is successfully created, a log message is generated to inform the user that the new database was created. This log message helps in tracking the execution of the script and confirming that the database creation was successful.

Checking Table Existence

Purpose of the Helper Function

The checkTableExists function is designed to check if a specified table exists within a PostgreSQL database. This is necessary to determine whether to create a new table or perform additional operations on an existing one.

Query for Checking Table Existence

The function constructs an SQL query that checks if the table exists in the `information_schema.tables` system catalog. This catalog stores metadata about all tables in the database. The SQL query uses `SELECT EXISTS`, which returns a boolean value indicating whether the specified table exists.

Preparing and Executing the Query

The query is parameterized by setting the table name in the prepared statement. This avoids SQL injection risks and ensures that the table name is safely included in the query. The `executeQuery()` method is used to run the query and retrieve the result.

Interpreting the Result

The result of the query is stored in a `ResultSet`. The function checks whether the query returned any data and if the result is true, meaning the table exists.

The `resultSet.next()` checks if the result set has any rows, and `resultSet.getBoolean(1)` retrieves the boolean value that indicates whether the table exists.

Closing Resources

Once the existence check is complete, the function closes both the `ResultSet` and `PreparedStatement` to release resources.

Returning the Result

The function returns a boolean value indicating whether the table exists. If the table exists, it returns `true`; otherwise, it returns `false`.

Creating a Table

Purpose of the Helper Function

The `createTable` function is responsible for creating a new table in the PostgreSQL database. It takes the database connection, the table name, and a map of attributes (columns) as inputs. The function constructs and executes a dynamic SQL query to create the table with the specified attributes.

Validating and Filtering Columns

The function starts by filtering the provided attributes to exclude certain columns, such as `"id"` and `"updated_at"`. These columns are handled separately in the table creation process.

validColumns: This list contains the columns from the attributes map that are valid (not null or excluded columns). These columns are then added to the SQL query.

Constructing the SQL Query

The function constructs a CREATE TABLE SQL query. The table creation statement includes:

- I. "id" SERIAL: A primary key column that auto-increments.
- II. "updated_at" TIMESTAMP WITH TIME ZONE: A column to store timestamps with time zone information. Valid columns: These are dynamically added based on the filtered attributes and are defined as TEXT columns in the table.
- III. "is_deleted" BOOLEAN DEFAULT FALSE: A column to mark records as deleted, defaulting to FALSE.
- IV. Primary Key: The primary key is set on both the "id" and "updated_at" columns to ensure unique records are identified by both.
- V. Additionally, the table is partitioned by the "updated_at" column using range partitioning to facilitate better performance with large datasets.

Executing the Query

The function prepares the SQL query using a PreparedStatement and executes it with executeUpdate() to create the table in the database.

Closing Resources

Once the table is created, the function closes the PreparedStatement to release resources and avoid memory leaks.

Logging the Table Creation

Finally, the function logs an informational message that indicates the table was created and specifies that partitioning is applied on the "updated_at" column.

Retrieving Existing Columns in a Table

Purpose of the Helper Function

The getExistingColumns function is used to retrieve the list of column names in a specific table from the database. This function is essential for comparing the current state of a table with the incoming data and determining if any new columns need to be added.

SQL Query to Fetch Columns

The function constructs an SQL query to retrieve the column names from the `information_schema.columns` view. This view contains metadata about all columns in the database, including their names and types.

SQL Query: The query filters columns based on the `table_name` provided, ensuring that only the columns for the specified table are returned.

Preparing and Executing the Query

The query is parameterized, and the table name is set in the prepared statement to prevent SQL injection and ensure safe query execution.

The `executeQuery()` method is used to run the query, retrieving the results from the database.

Processing the Result Set

The function processes the results by iterating over each row in the `ResultSet`. For each row, the column name is fetched using `resultSet.getString("column_name")` and added to a set called `existingColumns`.

Using a Set: A set is used here to store column names because it automatically handles duplicate values, ensuring that each column name appears only once.

Closing Resources

After retrieving all the column names, the function closes both the `ResultSet` and `PreparedStatement` to free up database resources and prevent memory leaks.

Returning the Column Names

The function returns the `existingColumns` set, which contains the names of all columns in the specified table.

Adding Missing Columns to the Table

Purpose of the Helper Function

The `addMissingColumns` function is responsible for adding columns to a table that do not already exist. This is important for keeping the table schema up to date with the incoming data, especially when new attributes are introduced.

Identifying Columns to Add

The function first filters the provided attributes map to identify columns that:

- i. Are not already present in the `existingColumns` set (the set of columns that already exist in the table).

- ii. Have a valid name (non-null, non-empty, and properly truncated using `truncateColumnName`).

- iii. Filtering Logic: The `findAll` method is used to filter out columns that are either empty, null, or already exist in the table.

- iv. The `truncateColumnName` method ensures that column names are in the proper format (e.g., truncating them if they exceed length limitations).

Adding the Columns

Once the missing columns are identified, the function iterates over them and attempts to add each column to the table using the `addColumn` function.

Error Handling

If a column already exists in the database, a `SQLException` will be thrown. The function catches this exception and logs a warning, indicating that the column already exists and is being skipped. If the error is not due to the column already existing, the exception is rethrown for further handling.

Skipping Invalid Columns

Before attempting to add a column, the function checks if the column name is valid (non-null and non-empty). If the column name is invalid, it is skipped, and the process moves to the next column.

Exception Handling

The function includes a try-catch block to gracefully handle any database exceptions, particularly the case where the column already exists. This ensures that the process continues without crashing even if the column is already present.

Adding a Column to the Table

Purpose of the Helper Function

The `addColumn` function is used to add a new column to a specified table in the database. This is necessary when the schema needs to be updated with new fields, allowing the table to accommodate incoming data attributes that were previously missing.

SQL Query for Adding the Column

The function constructs an SQL query using the `ALTER TABLE` statement to add a new column to the specified table. The `columnName` is set to `TEXT` type, which means that the new column will hold textual data.

SQL Query: The column name is wrapped with `quoteIdentifier` to ensure it has properly escaped, protecting against potential SQL injection issues and ensuring the name is valid, especially if it includes special characters or spaces.

Preparing and Executing the Query

Once the SQL query is created, the function prepares the statement with the `conn.prepareStatement(query)` method. The `executeUpdate()` method is used to run the query, which modifies the table structure by adding the new column.

Closing the PreparedStatement

After the column has been added successfully, the `PreparedStatement` is closed using `preparedStatement.close()` to release resources and prevent memory leaks.

Logging the Action

The function logs an informational message using `log.info` to confirm that the column has been successfully added to the table. This log helps track the schema changes made by the script.

Creating Partitions for the Table

Purpose of the Helper Function

The createPartition function is responsible for creating partitions for a given table in the database, based on a column (updated_at) that typically stores timestamp values. The function ensures that new partitions are created dynamically based on the year and month derived from the updated_at field, which is crucial for optimizing data management and query performance on large datasets.

Extracting Year and Month

The updatedAt parameter (which represents the timestamp) is first trimmed of any excess spaces. The date part is extracted using split(" ")[0], ensuring that only the date (and not the time) is considered. The year and month are further extracted by splitting the date string by hyphen "-".

Constructing the Partition Name

A partition name is dynamically generated using the table name and the extracted year and month, following the format: "tableName_year_month". This ensures that the partition name is both unique and easily identifiable.

Checking if the Partition Already Exists

The function first checks if a partition already exists for the given year and month by querying the pg_tables system catalog. If the partition exists, no further action is taken. If the partition does not exist, the function proceeds to create a new one.

SQL Query: A SELECT EXISTS query is used to check if a table (partition) with the generated partition name already exists. If it does, the function skips the creation process.

Creating the Partition

If the partition doesn't exist, the function constructs a CREATE TABLE statement to create a new partition. The partition is created for the specified table and is bound by a date range (from the first day of the month to the end of the month). The timestamp (updated_at) is used to determine the partition's date range.

Partition Date Range: The partition range is calculated using the updated_at value, adding a one-month interval to define the partition's upper boundary.

SQL Query: The query creates the partition using the FOR VALUES FROM ... TO ... clause, which specifies the valid range of values for the partition.

Preparing and Executing the Partition Creation Query

The query is then prepared and executed using `conn.prepareStatement(createQuery)`. After execution, the `PreparedStatement` is closed to free up resources.

Logging the Action

Finally, a log message is generated to confirm that the partition has been created successfully, including the year and month for which the partition was created. This helps in tracking the partition creation and ensuring that the process is completed correctly.

Helper Functions for Quoting Identifiers and Truncating Column Names

These functions are used to handle database table and column names safely, especially when working with PostgreSQL. They ensure that your database operations don't run into issues because of name length or special characters.

quoteIdentifier Function

This function makes sure that table and column names are formatted correctly for PostgreSQL. PostgreSQL requires that names with special characters or spaces, or that are case-sensitive, must be enclosed in double quotes ("). It takes a name (like a table or column) and adds double quotes around it. If the name already contains double quotes, it escapes them by doubling them up (e.g., " becomes ""), so it's valid in SQL.

Why It's Needed: If your table or column name has special characters (like spaces or hyphens) or is a reserved word (like SELECT), you need to quote it to avoid errors. It ensures that the database can correctly interpret the name.

truncateColumnName Function

PostgreSQL limits column names to a maximum of 63 characters. This function makes sure that column names stay within that limit. If a column name is too long, it cuts it down to 63 characters. It also makes sure that any double quotes (") in the name are escaped properly.

Why It's Needed: PostgreSQL will throw an error if a column name is longer than 63 characters. This function ensures that column names are always within the valid length and formatted properly.

Data Insertion

Importing Required Libraries

The script begins by importing essential libraries required as **JsonSlurper** used to parse JSON strings into Groovy objects like maps or lists. It simplifies working with JSON data, making it easier to access and manipulate nested structures. **DriverManager** java class that manages database connections. It establishes a connection to a database by using a connection URL, username, and password, enabling database interactions. **Connection** represents an open connection to the database, used to execute SQL queries and retrieve results. All database operations rely on this active connection. **PreparedStatement** class used to execute parameterized SQL queries. It helps prevent SQL injection and optimizes query performance by allowing dynamic query execution. **Types** utility class that provides constants for SQL data types, ensuring that correct types are used in SQL queries and preventing type mismatches. **OffsetDateTime** represents a date-time with an offset from UTC, used for handling time zone-aware date-time values, ensuring accurate time calculations across time zones. **DateTimeFormatter** class used to format and parse date-time objects. It ensures that date-time values are consistently represented and compatible with various systems. **Timestamp** represents a specific point in time, typically used for storing and manipulating timestamp data in databases, ensuring accurate time tracking.

FlowFile Retrieval and Validation

Retrieving a FlowFile from the Session

The script retrieves a FlowFile from the NiFi session by calling the `session.get()` method. In NiFi, a FlowFile represents a piece of data that flows through the system. This method checks if there is a FlowFile available for processing, and if so, assigns it to the variable `flowFile`. If no FlowFile is available, the method will return null, indicating that there is no data to process at that moment.

Checking if the FlowFile Exists

After attempting to retrieve the FlowFile, the script checks whether the `flowFile` variable is null. If it is null, this means no FlowFile was retrieved from the session. The script then exits early using a return statement to prevent any further processing. This step ensures that the script doesn't try to operate on a non-existent FlowFile, which could lead to errors or unnecessary processing steps.

Attribute Extraction and Validation

Extracting FlowFile Attributes

The script extracts three attributes from the FlowFile: `tableName`, `flattenedAttributes`, and `airport_id`. These attributes are essential for further processing, as they provide information about the target database table, the data structure to be inserted, and the specific airport ID that will be used to customize database interactions. The `flowFile.getAttribute()` method retrieves the values of these attributes from the FlowFile.

Validating the Table Name

The script checks if the `tableName` attribute is present and valid. This is done by verifying that the value is not null or empty and does not consist of only whitespace (using `trim()`). If the `tableName` is invalid or missing, the script logs an error and transfers the FlowFile to the `REL_FAILURE` relationship. This ensures that the script does not proceed with invalid or incomplete data, preventing potential issues in later stages of processing.

Parsing Flattened Attributes

The script then parses the `flattenedAttributesJson` (a JSON string) using `JsonSlurper`. This parsing step converts the JSON string into a Groovy map, allowing the script to easily work with the data. The parsed map, `flattenedAttributes`, will hold the key-value pairs that define the data to be processed or inserted into the database. This conversion is essential for working with structured data, such as JSON, in a programmatic way within Groovy scripts.

Validation of Required Columns

Checking for the 'id' Column

The script checks if the `flattenedAttributes` map contains a key named `id`, which is crucial for identifying the record in insert or update operations. The check is case-insensitive, meaning it will match any variation of "id" (e.g., "ID", "Id", etc.). If the `id` column is missing from the `flattenedAttributes` map, the script logs an error message indicating that the "id" column is required.

Handling Missing 'id' Column

If the `id` column is not present, the script transfers the FlowFile to the `REL_FAILURE` relationship, indicating that the current FlowFile cannot be processed further due to missing required data. The early return prevents further processing, ensuring that the script does not proceed with incomplete or invalid data.

Validation of Valid Columns

Dynamically Validating Columns

The script dynamically validates the columns by filtering the `flattenedAttributes` map to include only those entries where the key (column name) is non-null and non-empty after trimming any whitespace. This ensures that only valid columns are considered for insertion or update. The validation is done by calling the `findAll` method on the `flattenedAttributes` map, which checks if the column name (key) is valid.

Handling Empty Columns

If no valid columns are found after filtering, meaning the map contains no columns with non-null and non-empty names, the script logs an error message indicating that there are no valid columns to process. The FlowFile is then transferred to the REL_FAILURE relationship, stopping further execution to avoid processing data without any valid columns. This ensures data integrity by only allowing valid columns to be processed.

Database Connection Setup

Defining Database Connection Parameters

The script starts by defining the basic parameters required to establish a connection to the target database. These include:

- I. dbUrlBase: The base URL of the target database.
- II. dbUser: The username used for database authentication.
- III. dbPassword: The password associated with the database username.

These parameters are placeholders that will be replaced with actual values when the script is executed in a live NiFi environment.

Determining the Database URL

The script then defines dbUrlSample, which determines the full database connection URL. If the airportId is present, it appends the airportId to the dbUrlBase, creating a connection URL specific to that airport's database. If airportId is not provided, the URL defaults to a generic default_database. This dynamic URL creation allows the script to connect to different databases depending on the provided airportId. The connectionSample object is declared as null initially and will later be used to hold the actual database connection.

Establishing Database Connection and Data Validation

Attempting Database Connection

In this section, the script attempts to establish a connection to the target database using the DriverManager.getConnection() method. The connection is made using the previously defined connection URL (dbUrlSample), username (dbUser), and password (dbPassword). Once the connection is established, the script sets the autoCommit property to false. This ensures that database transactions are not committed automatically, giving the script full control over when to commit or rollback changes.

Retrieving Valid Column Names from the Database

The script calls the `getValidColumnNames()` function to retrieve a list of valid column names from the database schema for the specified `tableName`. This function queries the database metadata to get the list of columns that exist in the target table.

Filtering Incoming Attributes Based on Valid Columns

Next, the script filters the `flattenedAttributes` map to keep only those columns whose names match the valid columns retrieved from the database schema. This filtering ensures that only valid attributes (those that exist in the target table) are processed, and prevents attempts to insert or update invalid columns that do not exist in the table.

Handling Invalid or Missing Matching Columns

If the filtered `validAttributes` map is empty (i.e., there are no matching columns between the provided attributes and the table schema), the script logs an error indicating that no matching columns were found. It then transfers the `flowFile` to the `REL_FAILURE` relationship, terminating the processing of that particular `FlowFile`.

Handling Insert/Update Operations and Error Management

Checking if the Record Already Exists

The script starts by checking if a record with the provided `id` already exists in the target table. This is done by preparing a SQL query that looks for the `id` in the specified table. The `id` value is obtained from the `validAttributes` map. The query is executed using a prepared statement, which ensures that the `id` is securely and correctly inserted into the query. If the record exists (i.e., the query returns a result), the script proceeds to the update operation. If the record does not exist, it will proceed with the insert operation.

If the key corresponds to `updated_at`, the script ensures that the value is converted into a proper PostgreSQL-compatible timestamp with timezone (`timestampz`):

Date Parsing: The script attempts to parse the date-time string using a specific format (`yyyy-MM-dd HH:mm:ss.SSS Z`). This format includes the date, time, milliseconds, and timezone offset.

Conversion to Timestamp: After parsing the string into an `OffsetDateTime` object, the script converts it into a `Timestamp` object using `Timestamp.from()`. This conversion ensures compatibility with the database's expected data type for `updated_at`.

Error Handling: If the script fails to parse the date-time string (e.g., due to an invalid format), it logs an error and re-throws the exception to halt further execution. This prevents corrupted or improperly formatted timestamps from being stored.

Handling List Values: If the value is a list (e.g., a collection of strings or numbers), the script joins the elements into a single string, separated by commas. This allows multi-value data to be stored as a single database field, commonly used for storing delimited lists.

Handling Other Data Types: For all other data types, the script uses `preparedStatement.setObject()` to dynamically set the value in the prepared statement. This method ensures that the database driver determines the most appropriate type for the value.

Summary of Dynamic Handling

The `setPreparedStatementValue()` function ensures that each value is properly formatted and inserted into the SQL query according to its type:

Null values are stored as NULL. The `updated_at` column is correctly parsed and stored as a timestamp with timezone. Lists are serialized as comma-separated strings. Other types are handled generically to maintain flexibility. This robust handling ensures that the database receives clean, valid, and well-structured data during SQL execution.

Column Name Truncation

Purpose of Truncation

In PostgreSQL, the maximum allowed length for a column name is 63 characters. If a column name exceeds this length, it must be truncated to meet the database's constraints. The `truncateColumnName` function ensures that column names conform to this limit.

Truncating Column Names

Length Check: The function checks if the length of the provided column name (`name`) exceeds the maximum allowed length (63 characters).

Truncation Logic: If the column name exceeds 63 characters, it uses the `take(maxLength)` method to extract only the first 63 characters. If the column name is within the limit, it remains unchanged.

Escaping Quotes

After truncation (if necessary), the function escapes any double quotes (") in the column name by replacing them with two consecutive double quotes ("). This prevents syntax errors when the column name is used in SQL queries.

Example Use Cases

Input: A column name like

`very_long_column_name_exceeding_sixty_three_characters_in_length_but_needs_to_be_truncated.`

Truncated Output: `very_long_column_name_exceeding_sixty_three_characters_in_lengt.`

Escaped Name: If the column name contains a double quote, e.g., `column"name`, it becomes `column""name`.

Ensuring Compatibility

This function ensures that dynamically generated or user-provided column names remain compatible with PostgreSQL's length constraints and are safe to use in SQL queries without causing runtime errors. It is particularly useful in scenarios where column names are derived from user inputs or external sources.

Retrieving Valid Column Names

Overview

The `getValidColumnNames` function retrieves a list of column names for a specific table by querying the database's metadata. This ensures operations are performed only on valid columns as defined in the database schema.

Retrieving Metadata

Accessing Metadata: The script uses the `getMetaData()` method on the database connection to fetch metadata, which contains information about the database schema, tables, and columns.

Querying Columns: The `getColumns` method of `DatabaseMetaData` is used to fetch metadata about all columns in the specified table.

The parameters include `null` for schema and catalog, to target all schemas, the table name for which column metadata is required and `null` for the column pattern, to include all columns in the table.

Extracting Column Names

Iteration Through ResultSet: The function iterates over the result set returned by `getColumns` using a while loop.

For each entry: It retrieves the column name using `getString("COLUMN_NAME")`.

Normalization: Column names are converted to lowercase for consistent handling, as databases may treat column names differently with respect to case sensitivity.

Returning Valid Columns

Collecting Columns: Valid column names are appended to the `validColumns` list.

Final Output: The list of valid column names is returned, providing the caller with all the columns present in the specified table.

Performing Update Operations

Overview

The `performUpdate` function is responsible for updating an existing record in the database. It dynamically generates the necessary SET clause for updating specific columns and executes the query using secure parameterized inputs.

Generating the Update Clause

Column Validation and Formatting: The function iterates through the attributes provided in `validAttributes` (key-value pairs to update) and dynamically constructs the list of columns to be updated. Column names are validated and truncated if necessary. Quoting ensures that column names are compatible with the database's requirements.

Dynamic SQL Construction: The SET clause is built to include only the columns specified in `validAttributes`, making the query adaptable to different data inputs.

Preparing the Query

Dynamic Table Reference: The table name is properly quoted to handle special characters or reserved keywords.

PreparedStatement Usage: A `PreparedStatement` is created, enabling the query to securely bind values to placeholders instead of directly embedding them.

Binding Parameters

Attribute Value Binding: Each attribute's value is assigned to the corresponding placeholder using the `setPreparedStatementValue` helper function. This ensures proper formatting (e.g., handling nulls, timestamps, and lists).

ID Binding: The `idValue` is appended as the last parameter to specify which record to update.

Query Execution and Cleanup

Executing the Update: The prepared query is executed, sending the update request to the database.

Resource Management: The `PreparedStatement` is closed after execution to release resources and maintain efficient database connections.

Error Handling and Flexibility:

This method provides:

Flexibility: Dynamically handles varying sets of columns without requiring predefined SQL. **Security:** Uses parameterized queries to prevent SQL injection. **Efficiency:** Only updates specified columns, reducing unnecessary operations.

Delete Operation Flow

The delete operation is handled through a specific flow, where the operation is performed based on records in the `data_forms_dataformsrecyclebin` table and its `delete_date` column. To manage this operation effectively, a boolean flag (true/false) is introduced for the records in the `data_forms_dataformsrecyclebin` table. This flag determines the action to be taken on the corresponding records in the target database/table.

If the flag is assigned as true in the Parameters of nifi, the corresponding record in the target database/table, which contains an `is_deleted` column (default boolean value set to false), will be updated to true.

If the flag is assigned as false in the Parameters of nifi, the corresponding record in the target table will be deleted.

Steps to Achieve the Flow:

1. QueryDatabaseTableRecord Processor:

- a. This processor is used to maintain a cache date.
- b. **Initial Schedule:** On the first execution, it fetches all records from the `data_forms_dataformsrecyclebin` table based on the `delete_date` column and stores the cache date (which is max of `delete_date` column) in the processor's viewstate.
- c. **Subsequent Runs:** For subsequent executions, the processor only fetches records with a `delete_date` greater than the cache date. This ensures incremental processing of records.

2. UpdateAttribute Processor:

- a. To maintain the flag, an UpdateAttribute processor is used as the second step in the flow.
- b. In the processor properties, a new property is created with the name `Maintain_history_config_flag`.
- c. The value of this property is configurable using **NiFi Global Parameters**, which allows dynamic updates without needing to open or modify the processor directly.
- d. The value of this flag can be set to true or false based on the requirements.

By following these steps, the delete operation can be dynamically managed to either update the `is_deleted` flag in the target table or completely delete the record as per the flag's value.

3. ExecuteGroovyScript:

To perform update or delete operations in the target database/table, the ExecuteGroovyScript processor is utilized. The script processes input data, extracts necessary attributes, and determines the appropriate operation—either updating a flag (`is_deleted = true`) or deleting the record—based on a configurable flag (`Maintain_history_config_flag`).

Below is the detailed explanation of the script's workflow:

Script Overview

The script executed within the **ExecuteGroovyScript** processor performs update or delete operations in the target database or table based on a configurable flag named `Maintain_history_config_flag`.

Input Data Processing

The script processes input data by reading records line-by-line, where the first line represents headers and the second line contains corresponding data values. The data is split using a defined delimiter (`μ`), mapped into key-value pairs, and attributes like `data_json` are extracted. From the parsed JSON content, fields such as `form_parent`, `id`, and `airport_key` are identified for use in subsequent operations.

Dynamic Attribute Construction

Using the extracted attributes, the script dynamically constructs database and table names. The database name is prefixed with `db_` followed by the `airport_key`, while the table name is prefixed with `tbl_form_` followed by the `form_parent`. These dynamic values are then used to identify the target database and table.

Database Connection

The script retrieves essential database connection details such as the base database URL, user credentials, and dynamically constructed database names. It uses these details to securely establish a connection with the target database.

Flag-Based Operations

Based on the `Maintain_history_config_flag`, the script determines the operation to be performed. If the flag is set to true, an **UPDATE** query is executed to set the `is_deleted` column to true, marking the record as logically deleted. If the flag is set to false, a **DELETE** query is executed to completely remove the record from the target table.

Error Handling

The script includes robust error handling mechanisms to capture and log any issues during database operations. If an error occurs, details such as the error message are stored in flow file attributes for further debugging and monitoring.

Resource Management

After completing the operation, the script ensures that all database connections and resources are properly closed. This prevents resource leaks and ensures efficient utilization of system resources.

8. Incremental / Manual refresh of data based on Airport Codes and Form parent ID

The document explains different methods for refreshing data in the database, including full refreshes, incremental updates, and manual refreshes based on specific parameters such as form parent IDs and airport codes. Each approach is defined to ensure optimized performance, data consistency, and flexibility in executing database operations. The flows and logic for these processes are established using **Apache NiFi** processors, primarily focusing on `queryDatabaseRecord` and `executeSQLRecord` processors, and incorporating **cache maintenance** techniques for incremental update

Full / Incremental Refresh of Databases with Maintaining Cached Date

To perform a full refresh of databases while maintaining a cache date, we utilize the **NiFi 'QueryDatabaseRecord' processor**. This processor allows us to efficiently manage incremental updates by leveraging its **view state** functionality. The view state stores the most recent record timestamp, which is then used as a reference point for subsequent data fetch operations.

In this implementation, the **updated_at** column in the table **data_forms_formanswer** serves as the timestamp column to track the most recent updates. When the processor runs, it queries the table and retrieves only those records that have a timestamp greater than the last cached value. This ensures that we fetch only new or updated records since the last refresh, optimizing the overall performance and reducing data duplication.

Full Data Refresh Without Cached Date

For scenarios where a full refresh of data or incremental refresh is required without maintaining a cached date, we use the **NiFi 'ExecuteSQLRecord' processor**. This processor executes a SQL query that fetches all records from the table where **form_parent_id** is not null.

The process selects all records starting from the beginning of the table, ensuring a complete data refresh. However, since the **id** column is a primary key in the target database, attempting to insert duplicate id values would result in a constraint violation. To address this issue, we modified the **data insertion script** to handle duplicate keys gracefully. Instead of throwing an error when duplicate id values are encountered, the script performs an **upsert operation**. If the same id is encountered again, the existing record in the database is updated rather than being reinserted. This approach ensures consistency and avoids data conflicts during a full refresh.

Manual Incremental Refresh for Form Parent ID/Airport with Cached Date

This process involves maintaining a cached date in a dedicated table to enable efficient manual refreshes of specific **form parent IDs** or entire databases. To achieve this, we first create a table to store the maximum **updated_at** date column from the **data_forms_formanswer** table. The SQL script for creating this table is as follows:

```
CREATE TABLE public.date_table (
    id SERIAL PRIMARY KEY,
    max_date TIMESTAMP NOT NULL
);
```


This table stores the most recent `updated_at` value and is used as a reference for incremental refreshes. A separate flow in NiFi is created to query the **`data_forms_formanswer`** table and fetch the maximum updated date using the SQL query:

```
SELECT MAX(updated_at) FROM data_forms_formanswer;
```

The resulting date is stored in the **`date_table`** using an upsert operation. This table contains a single row with two columns: `id` and `timestamp`. It serves as a cached date repository for all refresh operations.

To execute a full or partial refresh based on the cached date, global parameters such as **`dbname`** (database name) and **`form_id`** (form parent ID) are used. These parameters allow users to configure the refresh operation without modifying the processors directly. If a specific **`form parent ID`** needs to be refreshed, both **`dbname`** and **`form_id`** must be specified. If the entire database requires a refresh, the **`form_id`** parameter is set to `NULL`.

The **`ExecuteSQLRecord`** processor is then used to fetch only the records updated after the cached date.

Manual Refresh for Specific Form Parent ID/Airport Without Cached Date

In cases where a manual refresh is required for a specific **`form parent ID`** or an airport without using the cached date, the process is like the above but does not involve querying or referencing the cached date table. Instead, all records are refreshed starting from the beginning of the table.

The **`ExecuteSQLRecord`** processor is used to fetch all data, regardless of the `updated_at` column.

The flexibility of this approach lies in its ability to target specific datasets for refresh without relying on the cached date. This is particularly useful when the cache date is unavailable, corrupted, or irrelevant for the operation.

Description of Processor Workflow

Every refresh operation, whether full or incremental, passes through a series of pre-configured processors. The pre-processors are designed to align with the specific operation being performed. The processors include:

- I. **Form Answer Table Mapping:** Handles mapping of form answer data to the target database schema.
- II. **Approvals Data Mapping:** Processes approvals-related data and integrates it into the workflow.
- III. **Database and Table Creation:** Ensures the necessary database structures (tables, partitions & indexes) are created prior to data insertion.
- IV. **Data Insertion:** Inserts or upserts the data into the target database, ensuring no duplication and maintaining consistency.

Each of these processors plays a key role in ensuring that data refresh operations—whether full, incremental, or manual—are executed efficiently and without error.

9. Performance Considerations in Database

Partitioning and indexing are two critical strategies for optimizing database performance, particularly when dealing with large datasets. They help improve query execution times, reduce I/O overhead, and manage resource utilization effectively. When applied to a table, their impact varies depending on the structure of the data, the query patterns, and the storage architecture.

Partitioning

Partitioning is a technique that splits a table into smaller, manageable subsets, improving database performance and making data easier to manage. Partitioning is especially useful for large tables with millions of rows, where querying, updating, or managing the table as a whole would otherwise lead to significant performance bottlenecks. When partitioning is applied, a table remains logically intact, but the data is physically divided based on certain criteria. These subsets are stored as independent partitions, which can be accessed individually based on query requirements.

One of the most common partitioning methods is **range partitioning**, where data is divided into partitions based on ranges of column values. For example, in a table containing transaction records, data can be partitioned by the date column into monthly partitions. Queries filtering for a specific month will only access the relevant partition instead of scanning the entire table, resulting in reduced I/O and faster query execution. Similarly, **list partitioning** divides data based on discrete values in a column, such as dividing records based on region or airport code. Hash partitioning, on the other hand, distributes data across partitions using a hash function, ensuring an even spread of rows to avoid data skew. This method is useful when no natural ranges or lists are available for partitioning.

The benefits of partitioning are particularly evident when it comes to improving performance and managing large datasets. Partition pruning, a feature used by most modern databases, allows queries to target only relevant partitions. For example, a query that filters by a specific date range will automatically ignore irrelevant partitions, dramatically reducing the number of rows scanned. Partitioning also enables parallel processing, where queries or maintenance operations can run on multiple partitions simultaneously, leveraging multi-core systems for faster execution. Maintenance tasks such as deleting or archiving old data become much more efficient, as entire partitions can be dropped or moved without impacting other partitions. This is particularly valuable for managing time-series data, where older data can be archived by dropping partitions without the need for complex delete operations.

However, partitioning introduces challenges that must be carefully managed. Poorly designed partitions can lead to **data skew**, where some partitions contain far more rows than others, negating performance benefits. **Queries that do not use the partition key will require scanning all partitions, resulting in no performance improvement.** Additionally, each partition comes with storage and management overhead, and excessive partitioning can lead to inefficiencies during query planning. For optimal performance, partitioning must be aligned with query patterns and storage capacity.

We have implemented range partitioning for the target tables, using the 'updated_at' column as the partitioning key. The ranges are defined based on the values in the 'updated_at' column to optimize query performance and data management.

Indexing

Indexing is a technique that improves the speed of data retrieval operations on a table by creating an optimized data structure that allows the database to locate rows efficiently. An index functions much like a book index, where keywords point directly to the page number, bypassing the need to search through every page. Similarly, an index enables the database to reduce the number of rows scanned when executing queries, enhancing overall performance.

Indexes can be created on one or more columns, depending on the query requirements. For example, a **single-column index** on a field like `airport_code` will optimize queries that filter data using this column. When queries involve filtering, sorting, or grouping by multiple columns, a **composite index** can be created on those specific columns to speed up execution. Indexes are particularly beneficial for operations like **SELECT**, **JOIN**, **ORDER BY**, and **GROUP BY**, where faster row access can significantly reduce query response times. A **clustered index** determines the physical order of rows in a table, ensuring rows are stored in the same sequence as the index. Conversely, a **non-clustered index** provides a logical ordering of rows and stores pointers to their physical locations, enabling faster searches without altering row order.

While indexes improve query performance, they also come with trade-offs. **Storage overhead** is a notable consideration, as indexes require additional space proportional to the table size and the number of indexed columns. For instance, a table with multiple indexes will consume more disk space, increasing database storage requirements. Moreover, maintaining indexes can impact the performance of write operations such as **INSERT**, **UPDATE**, and **DELETE**. Each write operation requires the database to update all associated indexes to keep them consistent, leading to overhead during high-volume data modifications. Over-indexing—creating unnecessary or redundant indexes—can further degrade performance by slowing down writes and consuming excessive storage.

Regular maintenance of indexes is essential to ensure they remain efficient. Over time, indexes can become **fragmented**, meaning their structure no longer aligns optimally with the table's physical data layout. This fragmentation can lead to slower read performance as the database must perform additional I/O to retrieve scattered data. Tasks like **index rebuilding** or **reorganizing** are critical for defragmenting indexes and restoring their performance. Properly balancing the number of indexes and their relevance to query patterns is key to achieving optimal database performance.

We have created indexes for all the columns in the target tables. However, for columns where the column value length exceeds 8KB, we have intentionally skipped creating indexes. This is because attempting to index columns with lengths greater than 8KB triggers an error, as the database does not support indexes of this size.

10. Final Deliverables

The following deliverables are included as part of the project setup and implementation:

Technologies Used

Apache NiFi :-Apache NiFi is an open-source data integration tool that automates the movement of data between disparate systems. It provides a web-based interface to design data flows, enabling users to create complex data pipelines through a drag-and-drop interface.

Features:

- **Data Ingestion:** NiFi can ingest data from various sources, including databases, file systems, and cloud services.
- **Data Transformation:** It allows for data transformation through processors that can filter, aggregate, and modify data as it moves through the pipeline.
- **Data Routing:** NiFi can route data based on content, enabling conditional data flows.
- **Security:** It offers secure data transfer through SSL and supports user authentication and authorization.

NiFi Registry :- NiFi Registry is a complementary service to Apache NiFi, used for versioning and managing the flow configurations. It allows users to store and manage different versions of their data flows, enabling better collaboration and change management.

Features:

- **Version Control:** Tracks changes to data flows, allowing users to revert to previous versions.
- **Collaboration:** Multiple users can work on the same data flow and track changes made by others.
- **Deployment:** Facilitates the deployment of data flows across different NiFi instances.

GitHub :- GitHub is a web-based platform used for version control and collaborative software development. It hosts repositories where code and related files can be stored and managed.

Features:

- **Repository Management:** Stores and manages code repositories, including version history and branching.
- **Collaboration:** Enables multiple users to collaborate on projects, submit pull requests, and review code changes.
- **Integration:** Supports integration with various CI/CD tools for automated testing and deployment.

LDAP (Lightweight Directory Access Protocol):- LDAP is a protocol used for accessing and maintaining distributed directory information services. It is commonly used for managing user information and authentication in a centralized directory.

Features:

- **User Authentication:** Manages user credentials and authenticates users across different systems.
- **Directory Services:** Stores and retrieves directory information such as user details, organizational units, and network resources.
- **Access Control:** Implements access control policies based on user roles and attributes.

Apache Ranger:- Apache Ranger is a framework to enable, monitor, and manage comprehensive data security across the Hadoop ecosystem. It provides centralized security administration, fine-grained access control, and auditing capabilities.

Features:

- **Centralized Security:** Manages security policies from a central administration console.
- **Fine-Grained Access Control:** Defines access policies at the file, table, or column level.
- **Auditing:** Monitors and logs all access requests for data, providing detailed audit reports.

AeroSimple Documentation

Description: This document provides an in-depth explanation of the NiFi ETL pipeline for handling forms and workflows data. It covers the entire NiFi flow, describing each component and its role within the pipeline.

Contents:

- **NiFi Flow Overview:** Detailed descriptions of each component in the NiFi pipeline.
- **Functional Usage Guidelines:** Instructions on how to use and customize the pipeline for specific data processing needs.

These deliverables ensure a comprehensive understanding and smooth implementation of the NiFi ETL process for handling forms and workflows data. They provide both technical setup instructions and functional usage guidelines, facilitating easy adoption and integration into your existing data infrastructure.

GitHub Repository Contents

1. FORMS_&_WORKFLOWS.json

Description: This is a NiFi template that streamlines the ETL process for forms and workflows data. The template can be imported directly into your Apache NiFi instance via the NiFi Registry. It provides a pre-configured workflow that is ready for immediate use, allowing you to handle forms and workflows data efficiently.

2. Forms & Workflows Document.pdf

Description: This document serves as a comprehensive guide to setting up and using the provided NiFi template. It includes detailed instructions on configuring database connections and integrating the NiFi template with your existing data infrastructure.

Contents:

- **Setup Instructions:** Step-by-step guidance on importing and configuring the NiFi template from Nifi registry.
- **Database Connection Configuration:** Instructions on setting up and managing database connections to ensure smooth data integration.

3. NIFI-README-2.0.md

Description: This file acts as a comprehensive setup guide for all the technologies used in the project. It includes detailed steps required to configure your environment to work seamlessly with the provided NiFi template.

Contents:

- **Apache NiFi Configuration:** Instructions for installing and configuring NiFi.
- **NiFi Registry Configuration:** Steps for setting up the NiFi Registry.
- **GitHub Integration:** Guide on how to integrate the templates and documents from the GitHub repository.
- **LDAP Setup:** Instructions for configuring LDAP for user authentication and access control.
- **Apache Ranger Configuration:** Steps for setting up Apache Ranger to manage data security and access policies.