Data Ingestion Framework: Job Flow Design Document

Table of Contents

1. **Introduction**
2. **Framework Overview**
3. **Job Flow Design**
   - 3.1 Job Configuration
   - 3.2 Data Ingestion Flow
   - 3.3 Validation Flow
   - 3.4 Partitioning and Writing Flow
   - 3.5 Logging and Error Handling Flow
   - 3.6 Self-Healing and Alerting
4. **Technical Stack**
5. **Performance and Scalability**
6. **Security Considerations**
7. **Conclusion**

1. Introduction

This document outlines the job flow design for the **Data Ingestion Framework**, which facilitates the ingestion of data from various source systems (e.g., Oracle, MySQL, PostgreSQL) into a target data lake (S3) using Apache Iceberg. The framework supports both full and incremental data loads, along with pre-ingestion and post-ingestion data validation, partitioning, logging, and error handling.

2. Framework Overview

The **Data Ingestion Framework** is built to automate and manage the data ingestion pipeline through:

- **Modular design** that decouples configurations and ingestion logic.
- **Job configurability** using `job_config` and `log_config` tables.
- **Pre-ingestion** and **post-ingestion validation** through customizable validation rules.
- **Partitioning support** for optimized data storage and query performance.
- **Error logging, alerting**, and **self-healing** for resilient operations.

3. Job Flow Design

The **Data Ingestion Job Flow** consists of several stages, each governed by specific configuration settings and validation rules. Below are the key stages of the flow.

3.1 Job Configuration

1. **Job Initialization**:
    - Fetch the job configuration details from the `job_config` table.
    - Load the necessary database connection details from `source_db_config.json` and `target_db_config.json`.
    - Set up logging using `log_config` from the `log_config` table for error, info, and debug logs.
2. **Key Configuration**:
    - `source_db_reference`: Database connection details (e.g., Oracle, MySQL).
    - `target_db_reference`: Target S3 bucket and Iceberg configuration.
    - `load_type`: Full or incremental load.
    - `cdc_column`: Change data capture (CDC) column for incremental loads.
    - `validation_rules_table`: Reference to pre/post-ingestion validation rules.

3.2 Data Ingestion Flow

1. **Build Query**:
    - Based on the job configuration, construct the SQL query to extract data from the source table.

    - If incremental, add the necessary filter on the `cdc_column`.
2. **Execute Query**:
    - Run the constructed query using JDBC to fetch the data from the source system into a Spark DataFrame.
3. **Key Configurations**:
    - `source_table`: Name of the source table.
    - `source_fields`: Fields to extract (optional).
    - `source_where_clause`: Optional filtering clause for source data.
4. **Load Data into DataFrame**:
    - Read the source data using Spark's JDBC connector with appropriate credentials and filters.

3.3 Validation Flow

1. **Pre-Ingestion Validation**:
    - **Schema Validation**: Validate if the schema of the extracted data matches the expected schema.
    - **Record Count Validation**: Check if the record count matches the expected count.
    - **Data Freshness Validation**: Ensure that the latest data is being ingested.
2. These validations are configurable using the `validation_rules` table.
3. **Post-Ingestion Validation**:
    - Perform business rule validations after the data is written to the target.
    - Validate that all partitions are correctly written.

3.4 Partitioning and Writing Flow

1. **Partition Data**:
    - Apply partitioning logic on the DataFrame before writing the data to the target (S3 or Hive).
    - The partition columns are defined in `target_partition_by` in the `job_config` table.
2. **Write Data to Target**:

- Write the partitioned data to the target database or S3 bucket in the specified format (e.g., Parquet, CSV).
- Ensure that data is written using the appropriate partitioning and output format settings.

3. **Key Configuration**:
   - `target_output_format`: Defines the output file format (e.g., Parquet, ORC, CSV).
   - `target_partition_by`: Columns to partition by.

3.5 Logging and Error Handling Flow

1. **Logging**:
   - Capture logs at different levels (`info`, `debug`, `error`) throughout the ingestion process.
   - Use the logging configuration from the `log_config` table to determine log destinations (e.g., S3, CloudWatch).
2. **Error Handling**:
   - If an error occurs during ingestion, log the issue and send alerts based on the `alert_threshold` defined in `log_config`.
   - Provide detailed error messages to aid in debugging.
3. **Retries**:
   - Automatically retry the job if it fails, based on the retry policy defined in the logging configuration.

3.6 Self-Healing and Alerting

1. **Self-Healing**:
   - The framework attempts self-healing actions in the case of transient errors (e.g., network failure, temporary unavailability of the source system).
   - Retry logic is implemented with a backoff mechanism.
2. **Alerting**:
   - If the job fails multiple times or encounters a critical error, send alerts via the configured alerting channel (e.g., email, PagerDuty).
   - Alert thresholds are defined in `log_config` to trigger actions after a set number of failures.

4. Technical Stack

The **Data Ingestion Framework** is built on the following technology stack:

- **Apache Spark**: Core engine for data extraction and transformation.
- **Apache Iceberg**: Storage layer on S3 or Hive for managing large datasets.
- **Airflow**: Scheduling and orchestration of the data ingestion jobs.
- **PyDeequ**: For data quality validation.
- **AWS S3**: Storage for ingested data.
- **Oracle, MySQL, PostgreSQL**: Supported source databases.

5. Performance and Scalability

The framework is designed to handle both **small** and **large-scale** datasets efficiently:

- **Partitioning** enables faster queries and reduces the size of data reads.
- **Incremental Loads** minimize the amount of data being processed in each run.
- **Auto-Scaling** is supported using the underlying Spark infrastructure (e.g., Amazon EMR, Databricks).

6. Security Considerations

1. **Data Encryption**:
    - Data is encrypted at rest in S3 using server-side encryption.
    - Secure connections to databases via JDBC with encrypted credentials.
2. **Access Control**:
    - Access to source and target systems is restricted through role-based access control (RBAC).
    - Apache Ranger is integrated for access management and audit logging.

7. Conclusion

The **Data Ingestion Framework** is a flexible and robust solution for ingesting data into an S3 data lake with support for:

- **Configurable job setup** through JSON and database configuration tables.
- **Data validation** at various stages of ingestion.
- **Comprehensive logging**, alerting, and error-handling mechanisms.
- **Partitioning** for optimized data storage and querying.

This design ensures the ingestion framework can scale to meet the needs of both small and large datasets while ensuring data integrity and security.

This design document should provide the necessary information for developers and data engineers to understand and contribute to the ingestion framework.

The data ingestion framework using config tables for job configuration, logging, validation, and parallelism with job group IDs. This approach replaces the JSON file configurations with table-based configurations for `job_config`, `log_config`, and `validation_rules`.

Directory Structure

bash

Copy code

```
data_ingestion_framework/
├── config/
│   ├── source_db_config.json         # Source DB configurations (e.g., Oracle, MySQL)
│   ├── target_db_config.json         # Target DB configurations (e.g., S3, Hive)
├── utils/
│   ├── ingestion_utils.py            # Data ingestion logic
│   ├── validation_utils.py           # Data validation functions (PyDeequ integration)
│   ├── logging_utils.py              # Logging issues
│   ├── alerting_utils.py             # Alerts for job failures
│   ├── self_healing.py               # Error recovery logic
```

```
│   ├── partition_utils.py          # Partitioning logic
├── main.py                         # Entry point for running jobs
├── tests/
│   ├── test_data_validation.py     # Unit tests using PyDeequ
└── resources/
    ├── validation_rules.json       # Validation rules for data quality checks
```

1. SQL Table Definitions

Job Config Table

sql
Copy code

```
CREATE TABLE config.job_config (
    job_id STRING PRIMARY KEY,                  -- Unique identifier for the job
    job_name STRING,                            -- Name of the job
    load_type STRING,                           -- 'full' or 'incremental'
    cdc_column STRING,                          -- CDC field name (timestamp or ID)
    cdc_column_type STRING,                     -- 'timestamp' or 'increasing_key'
    validation_rules_table STRING,              -- Reference to validation rules table
    source_table STRING,                        -- Source table name
    source_db_reference STRING,                 -- Reference to source DB in
source_db_config.json
    source_fields STRING,                       -- Comma-separated list of fields to ingest
    source_where_clause STRING,                 -- Optional filter for data ingestion
    target_table STRING,                        -- Target table name
    target_db_reference STRING,                 -- Reference to target DB in
target_db_config.json
    target_output_format STRING,                -- Output file format (e.g., parquet, csv)
    target_partition_by STRING,                 -- Comma-separated list of partitioning columns
    log_config_reference STRING,                -- Reference to logging/alerting config
```

```
    job_group_id STRING                          -- Group ID for job parallelism
);
```

Log Config Table

sql

Copy code

```sql
CREATE TABLE config.log_config (
    log_id STRING PRIMARY KEY,                   -- Unique identifier for logging config
    log_type STRING,                             -- 'error', 'info', 'debug'
    log_destination STRING,                      -- e.g., 's3://my-bucket/logs/', 'cloudwatch'
    alert_threshold INT,                         -- Number of retries before triggering alert
    alert_destination STRING                     -- 'email', 'pagerduty', etc.
);
```

Validation Rules Table

sql

Copy code

```sql
CREATE TABLE config.validation_rules (
    validation_rules_table STRING PRIMARY KEY,  -- Unique identifier for validation rules
    rule_name STRING,                            -- Name of the rule (e.g.,
'schema_validation')
    rule_type STRING,                            -- 'pre_ingestion' or 'post_ingestion'
    rule_expression STRING                       -- SQL-like expression or validation logic
);
```

2. Python Code

ingestion_utils.py

python

Copy code

```python
import logging
```

```python
from pyspark.sql import SparkSession
from utils.logging_utils import log_issue
from utils.validation_utils import validate_schema, validate_count, validate_data_freshness
from utils.partition_utils import apply_partitioning

def run_ingestion(spark: SparkSession, job_config):
    try:
        # Load source and target DB configurations
        source_db_config = load_source_db_config(job_config["source_db_reference"])
        target_db_config = load_target_db_config(job_config["target_db_reference"])

        # Build query for ingestion
        query = build_query(job_config)
        logging.info(f"Running query: {query}")

        # Load data from source
        df = spark.read.format("jdbc")\
            .option("url", source_db_config['jdbc_url'])\
            .option("dbtable", f"({query}) as source_data")\
            .option("user", source_db_config['username'])\
            .option("password", source_db_config['password'])\
            .load()

        # Pre-ingestion validations
        validate_schema(df, job_config)
        validate_count(df, job_config)
        validate_data_freshness(df, job_config)

        # Apply partitioning
        partition_cols = job_config['target_partition_by'].split(',')
```

```
        df.write.format(job_config['target_output_format'])\
            .partitionBy(partition_cols)\
            .mode("append")\
            .save(target_db_config['bucket'])

        # Post-ingestion validation
        validate_post_ingestion(df, job_config)

    except Exception as e:
        log_issue(f"Error in job {job_config['job_id']}: {str(e)}",
job_config['log_config_reference'])
        raise e
```

logging_utils.py

python
Copy code

```
import logging

def log_issue(message, log_config_reference):
    log_config = load_log_config(log_config_reference)

    if log_config['log_type'] == 'error':
        logging.error(message)
    elif log_config['log_type'] == 'info':
        logging.info(message)

    # Log to S3 or another destination
    if log_config['log_destination'].startswith("s3"):
        upload_log_to_s3(message, log_config['log_destination'])
```

```
def upload_log_to_s3(message, destination):
    # Code to upload logs to S3
    pass
```

validation_utils.py

python
Copy code

```python
import logging

def validate_schema(df, job_config):
    # Implement schema validation logic
    logging.info(f"Validating schema for job {job_config['job_id']}")

def validate_count(df, job_config):
    # Implement record count validation
    logging.info(f"Validating record count for job {job_config['job_id']}")

def validate_data_freshness(df, job_config):
    # Implement data freshness validation
    logging.info(f"Validating data freshness for job {job_config['job_id']}")
```

partition_utils.py

python
Copy code

```python
def apply_partitioning(df, job_config):
    partition_cols = job_config['target_partition_by'].split(',')
    df.write.partitionBy(partition_cols).mode("append").save(job_config['target_table'])
```

3. README.md

markdown

Copy code

```
# Data Ingestion Framework

## Overview
This ingestion framework facilitates the movement of data from various source systems (e.g.,
Oracle, MySQL) into an S3 bucket using Iceberg format, with support for both full and
incremental loads. The framework is configurable through metadata tables and integrates
logging, validation, and parallel execution using job group IDs.

## Features
- **Configurable**: Uses `job_config` and `log_config` tables for metadata-driven ingestion.
- **Source/Target Configurations**: External DB configurations stored as JSON files
(`source_db_config.json`, `target_db_config.json`).
- **Data Validation**: Pre-ingestion and post-ingestion validation rules (record count,
schema validation, data freshness).
- **Partitioning**: Supports partitioning by one or more columns for efficient data storage.
- **Error Handling**: Captures errors in S3 or other log destinations.
- **Self-Healing**: Implement basic error recovery strategies using `self_healing.py`.

## Table Structures
### Job Config Table
Contains metadata for each ingestion job, including source/target tables, load type,
partitioning, and CDC columns.

### Log Config Table
Specifies logging configuration, including log destination (e.g., S3) and alerting rules.

### Validation Rules Table
Defines validation logic (e.g., schema validation, record count validation) to ensure data
integrity.
```

Usage
1. Configure your source and target DBs in `source_db_config.json` and `target_db_config.json`.
2. Define your ingestion jobs in the `config.job_config` table.
3. Run the ingestion process using:
```bash
python main.py
```

Logging

Logs are stored based on the `log_config_reference` in the `log_config` table. Supported destinations include S3 and other logging platforms.

Alerts

Alerts are triggered based on thresholds specified in the `log_config` table. Alerts can be sent via email, PagerDuty, or other mechanisms.

bash
Copy code
Conclusion:
This ingestion framework offers flexibility, scalability, and a robust mechanism for managing data ingestion jobs across multiple systems, with support for logging, validation, and partitioning.

Below is a sample implementation of a `self_healing.py` script designed for a data ingestion framework. This script aims to handle transient errors during data ingestion by implementing retry logic with exponential backoff.

self_healing.py

python
Copy code
```python
import time
import logging
from pyspark.sql import SparkSession

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class DataIngestion:
    def __init__(self, spark: SparkSession, max_retries: int = 3, backoff_factor: float =
2.0):
        self.spark = spark
        self.max_retries = max_retries
        self.backoff_factor = backoff_factor

    def ingest_data(self, source_query: str, target_path: str, format: str):
        """
        Ingest data from source to target with self-healing capabilities.

        :param source_query: SQL query to fetch data from the source.
        :param target_path: Target path for writing the data.
        :param format: Format of the data to be written (e.g., 'parquet', 'csv').
        """
        attempts = 0

        while attempts < self.max_retries:
```

```
        try:
            # Attempt to fetch data from the source
            logger.info(f"Attempting to execute source query: {source_query}")
            df = self.spark.sql(source_query)

            # Check if DataFrame is empty
            if df.count() == 0:
                logger.warning("No data returned from the source query.")
                break

            # Write the DataFrame to the target path
            logger.info(f"Writing data to {target_path} in {format} format.")
            df.write.format(format).mode('overwrite').save(target_path)

            logger.info("Data ingestion completed successfully.")
            break  # Exit the loop if ingestion is successful

        except Exception as e:
            attempts += 1
            wait_time = self.backoff_factor ** attempts
            logger.error(f"Error during data ingestion: {e}. Attempt {attempts} of
{self.max_retries}.")

            # Check if maximum retries reached
            if attempts >= self.max_retries:
                logger.critical("Maximum retries reached. Data ingestion failed.")
                raise
            else:
                logger.info(f"Retrying in {wait_time} seconds...")
                time.sleep(wait_time)
```

```python
if __name__ == "__main__":
    # Example usage
    spark = SparkSession.builder \
        .appName("Self Healing Data Ingestion") \
        .getOrCreate()

    # Define your source SQL query and target path
    source_query = "SELECT * FROM your_source_table"
    target_path = "s3://your-bucket/path/to/store/data"
    format = "parquet"

    data_ingestion = DataIngestion(spark)
    data_ingestion.ingest_data(source_query, target_path, format)
```

Code Explanation

- **Logging Configuration**: The script sets up basic logging to monitor the ingestion process and errors.
- **DataIngestion Class**:
    - Initializes with a Spark session, maximum retry count, and backoff factor for exponential wait time between retries.
    - Contains the `ingest_data` method which:
        - Executes the provided SQL query to fetch data from the source.
        - Checks if the resulting DataFrame is empty.
        - Writes the DataFrame to the specified target path in the given format (e.g., Parquet, CSV).
        - Implements error handling that retries the operation in case of transient errors, increasing the wait time between attempts based on the backoff factor.
- **Main Execution**:
    - Sets up a Spark session and defines a source SQL query and target path.
    - Creates an instance of the `DataIngestion` class and calls the `ingest_data` method to start the ingestion process.

Usage

- Make sure to replace `"your_source_table"` and `"s3://your-bucket/path/to/store/data"` with your actual source table and target S3 path.
- You can run this script in an environment where Apache Spark is configured and integrated with your S3 access credentials.

Below is a detailed implementation of `validation_utils.py`, which includes functions for various types of data validation, such as schema validation, record count validation, and data freshness validation. The code also includes logging for monitoring the validation process.

validation_utils.py

python
Copy code

```python
import logging
from pyspark.sql import DataFrame
from pyspark.sql.functions import col, current_timestamp, datediff

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def validate_schema(df: DataFrame, job_config: dict):
    """
    Validates the schema of the DataFrame against the expected schema defined in the job configuration.
```

```
    :param df: DataFrame to validate
    :param job_config: Configuration for the job containing expected schema information
    """
    expected_schema = job_config.get("expected_schema")
    if expected_schema is None:
        logger.warning(f"No expected schema defined for job {job_config['job_id']}. Skipping
schema validation.")
        return

    actual_schema = df.schema
    if actual_schema != expected_schema:
        logger.error(f"Schema mismatch for job {job_config['job_id']}. Expected:
{expected_schema}, Found: {actual_schema}")
        raise ValueError(f"Schema mismatch for job {job_config['job_id']}")
    else:
        logger.info(f"Schema validated successfully for job {job_config['job_id']}.")


def validate_count(df: DataFrame, job_config: dict):
    """
    Validates the record count of the DataFrame against the expected count defined in the job
configuration.

    :param df: DataFrame to validate
    :param job_config: Configuration for the job containing expected count information
    """
    expected_count = job_config.get("expected_count")
    if expected_count is not None:
        actual_count = df.count()
```

```
        if actual_count != expected_count:
            logger.error(f"Record count mismatch for job {job_config['job_id']}. Expected:
{expected_count}, Found: {actual_count}")
            raise ValueError(f"Record count mismatch for job {job_config['job_id']}")
        else:
            logger.info(f"Record count validated successfully for job
{job_config['job_id']}.")


def validate_data_freshness(df: DataFrame, job_config: dict):
    """
    Validates the freshness of the data by checking the timestamp of the latest records.

    :param df: DataFrame to validate
    :param job_config: Configuration for the job containing freshness rules
    """
    freshness_threshold_days = job_config.get("freshness_threshold_days")
    if freshness_threshold_days is None:
        logger.warning(f"No freshness threshold defined for job {job_config['job_id']}.
Skipping freshness validation.")
        return

    # Assuming there's a timestamp column to check freshness (adjust as needed)
    if 'last_updated' not in df.columns:
        logger.error(f"'last_updated' column not found in DataFrame for job
{job_config['job_id']}.")
        raise ValueError(f"'last_updated' column not found for job {job_config['job_id']}")

    max_timestamp = df.agg({"last_updated": "max"}).collect()[0][0]
```

```
    logger.info(f"Latest timestamp in DataFrame for job {job_config['job_id']}:
{max_timestamp}")

    if max_timestamp is not None:
        days_since_last_update = (current_timestamp() - max_timestamp).days
        if days_since_last_update > freshness_threshold_days:
            logger.error(f"Data freshness violation for job {job_config['job_id']}. Last
update was {days_since_last_update} days ago.")
            raise ValueError(f"Data freshness violation for job {job_config['job_id']}")
        else:
            logger.info(f"Data freshness validated successfully for job
{job_config['job_id']}.")


def validate_post_ingestion(df: DataFrame, job_config: dict):
    """
    Placeholder function for post-ingestion validation. Add business rule checks here.

    :param df: DataFrame to validate
    :param job_config: Configuration for the job containing post-ingestion rules
    """
    logger.info(f"Running post-ingestion validations for job {job_config['job_id']}.")
    # Implement additional post-ingestion checks as needed
```

Code Explanation

1. **Logging Configuration**: The script sets up logging to monitor the validation process. It uses different log levels (INFO, WARNING, ERROR) to capture the status of each validation step.
2. **Validation Functions**:

- **`validate_schema`**:
  - Compares the actual schema of the DataFrame with the expected schema defined in the job configuration.
  - Raises a `ValueError` if the schemas do not match and logs an error message. If they match, it logs a success message.
- **`validate_count`**:
  - Checks the actual record count of the DataFrame against an expected count specified in the job configuration.
  - Raises a `ValueError` if the counts do not match, logging an error. If they match, it logs success.
- **`validate_data_freshness`**:
  - Checks the freshness of the data based on the `last_updated` column.
  - It raises an error if the latest update exceeds the defined freshness threshold (in days).
- **`validate_post_ingestion`**:
  - This is a placeholder for any additional business rule checks you want to implement after data ingestion.

Usage

- This validation utility is intended to be integrated into your data ingestion framework.
- Make sure to provide the necessary fields in your job configuration (like `expected_schema`, `expected_count`, and `freshness_threshold_days`) for the validations to function correctly.
- You can call these validation functions from the `ingestion_utils.py` script after loading the data to ensure that all necessary checks are performed before proceeding with further processing or writing to the target.

Sample Job Configuration Example

Here's an example of what your job configuration might look like to include expected schema, expected count, and freshness threshold:

json
Copy code

```json
{
    "job_id": "job_001",
    "expected_schema": {
        "type": "struct",
        "fields": [
            {"name": "id", "type": "integer"},
            {"name": "name", "type": "string"},
```

```
            {"name": "email", "type": "string"},
            {"name": "last_updated", "type": "timestamp"}
        ]
    },
    "expected_count": 100,
    "freshness_threshold_days": 30
}
```
This detailed implementation provides robust validation capabilities for your data ingestion framework.

---

enhanced version of the `ingestion_utils.py` that includes the complete logic for `load_source_db_config`, `load_target_db_config`, and `build_query`. This example assumes you have a way to retrieve database configurations and query parameters, typically from a configuration file, environment variables, or a database.

Complete `ingestion_utils.py`

python
Copy code
```
import logging
import json
import os
from pyspark.sql import SparkSession
from utils.logging_utils import log_issue
from utils.validation_utils import validate_schema, validate_count, validate_data_freshness,
validate_post_ingestion
```

```
from utils.partition_utils import apply_partitioning
from concurrent.futures import ThreadPoolExecutor, as_completed

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def load_source_db_config(source_db_reference):
    """
    Load and return the source DB configuration based on the reference.

    :param source_db_reference: Reference ID for the source DB configuration
    :return: A dictionary containing the source DB configuration
    """
    config_path = os.getenv("SOURCE_DB_CONFIG_PATH")  # Path to your configuration JSON file
    try:
        with open(config_path, 'r') as f:
            config_data = json.load(f)
            source_db_config = config_data.get(source_db_reference)
            if not source_db_config:
                raise ValueError(f"No configuration found for source_db_reference:
{source_db_reference}")
            return source_db_config
    except Exception as e:
        logger.error(f"Error loading source DB config: {str(e)}")
        raise e

def load_target_db_config(target_db_reference):
    """
    Load and return the target DB configuration based on the reference.
```

```
    :param target_db_reference: Reference ID for the target DB configuration
    :return: A dictionary containing the target DB configuration
    """
    config_path = os.getenv("TARGET_DB_CONFIG_PATH")  # Path to your configuration JSON file
    try:
        with open(config_path, 'r') as f:
            config_data = json.load(f)
            target_db_config = config_data.get(target_db_reference)
            if not target_db_config:
                raise ValueError(f"No configuration found for target_db_reference:
{target_db_reference}")
            return target_db_config
    except Exception as e:
        logger.error(f"Error loading target DB config: {str(e)}")
        raise e

def build_query(job_config):
    """
    Build and return the SQL query string for ingestion based on the job configuration.

    :param job_config: The job configuration dictionary
    :return: A SQL query string
    """
    table_name = job_config.get("source_table_name")
    filters = job_config.get("filters", "")

    if not table_name:
        raise ValueError("source_table_name is required in the job configuration.")
```

```
    query = f"SELECT * FROM {table_name}"

    if filters:
        query += f" WHERE {filters}"

    logger.info(f"Built query: {query}")
    return query

def run_ingestion(spark: SparkSession, job_config):
    try:
        # Load source and target DB configurations
        source_db_config = load_source_db_config(job_config["source_db_reference"])
        target_db_config = load_target_db_config(job_config["target_db_reference"])

        # Build query for ingestion
        query = build_query(job_config)
        logger.info(f"Running query for job {job_config['job_id']}: {query}")

        # Load data from source
        df = spark.read.format("jdbc")\
            .option("url", source_db_config['jdbc_url'])\
            .option("dbtable", f"({query}) as source_data")\
            .option("user", source_db_config['username'])\
            .option("password", source_db_config['password'])\
            .load()

        # Pre-ingestion validations
        validate_schema(df, job_config)
        validate_count(df, job_config)
        validate_data_freshness(df, job_config)
```

```
        # Apply partitioning and save the data
        partition_cols = job_config['target_partition_by'].split(',')
        df.write.format(job_config['target_output_format'])\
            .partitionBy(partition_cols)\
            .mode("append")\
            .save(target_db_config['bucket'])

        # Post-ingestion validation
        validate_post_ingestion(df, job_config)

        logger.info(f"Ingestion completed for job {job_config['job_id']}")

    except Exception as e:
        log_issue(f"Error in job {job_config['job_id']}: {str(e)}",
job_config['log_config_reference'])
        raise e

def run_parallel_ingestions(spark: SparkSession, job_configs: list, run_group_id: str):
    """
    Run multiple ingestion jobs in parallel within the same job run group ID.

    :param spark: SparkSession instance
    :param job_configs: List of job configurations
    :param run_group_id: Identifier for the group of job runs
    """
    logger.info(f"Starting parallel ingestions for run group: {run_group_id}")
    results = []

    with ThreadPoolExecutor() as executor:
```

```
        future_to_job = {executor.submit(run_ingestion, spark, job_config): job_config for
job_config in job_configs}

        for future in as_completed(future_to_job):
            job_config = future_to_job[future]
            try:
                future.result()  # This will raise an exception if the job failed
                results.append((job_config['job_id'], "Success"))
            except Exception as exc:
                logger.error(f"Job {job_config['job_id']} generated an exception: {exc}")
                results.append((job_config['job_id'], "Failed"))

    logger.info(f"All ingestions completed for run group: {run_group_id}")
    return results

# Sample usage
if __name__ == "__main__":
    spark = SparkSession.builder.appName("DataIngestion").getOrCreate()

    # Example job configurations
    job_configs = [
        {
            "job_id": "job_001",
            "source_db_reference": "source_db_ref_1",
            "target_db_reference": "target_db_ref_1",
            "source_table_name": "your_table_1",
            "filters": "condition_1",
            "target_partition_by": "date",
            "target_output_format": "parquet",
            "log_config_reference": "log_ref_1"
```

```
        },
        {
            "job_id": "job_002",
            "source_db_reference": "source_db_ref_2",
            "target_db_reference": "target_db_ref_2",
            "source_table_name": "your_table_2",
            "filters": "condition_2",
            "target_partition_by": "date",
            "target_output_format": "parquet",
            "log_config_reference": "log_ref_2"
        }
    ]

    run_group_id = "run_group_001"
    run_parallel_ingestions(spark, job_configs, run_group_id)
```

Explanation of Key Components

1. **Configuration Loading**:
    - `load_source_db_config` and `load_target_db_config` functions read configuration data from JSON files specified by the environment variables `SOURCE_DB_CONFIG_PATH` and `TARGET_DB_CONFIG_PATH`. Ensure these files are structured to include all necessary DB configuration details.
2. **Sample JSON Structure for Source and Target DB Configs**:
   json
   Copy code

```
{
```

3. `"source_db_ref_1": {`

```
        "jdbc_url": "jdbc:mysql://localhost:3306/source_db",
        "username": "user",
        "password": "pass"
    },
    "target_db_ref_1": {
        "bucket": "s3://your-target-bucket"
    }
}
```

13. **Query Building**:
    - The `build_query` function constructs a SQL query based on the `source_table_name` and optional `filters`provided in the job configuration. The function raises an error if the required `source_table_name` is not specified.
14. **Ingestion Logic**:
    - The `run_ingestion` function is responsible for executing the ingestion process, including reading data from the source, validating it, saving it to the target, and logging the results.
15. **Parallel Ingestion**:
    - The `run_parallel_ingestions` function uses a `ThreadPoolExecutor` to run multiple ingestion jobs in parallel, logging successes and failures.
16. **Sample Job Configurations**:
    - In the `__main__` block, sample job configurations are provided to demonstrate how to define multiple jobs for ingestion. You should replace the table names and conditions with your actual database schema.

How to Use

1. Ensure you have the necessary environment variables set for the configuration file paths.
2. Update the JSON configuration files with your actual source and target database credentials.
3. Run the script, and it will perform the ingestion jobs as defined in the job configurations.

Updated `ingestion_utils.py`

python
Copy code
```
import logging
import json
import os
import pandas as pd
from pyspark.sql import SparkSession
from pyspark.sql import DataFrame
from utils.logging_utils import log_issue
from utils.validation_utils import validate_schema, validate_count, validate_data_freshness,
validate_post_ingestion
from utils.partition_utils import apply_partitioning
from concurrent.futures import ThreadPoolExecutor, as_completed

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def load_source_db_config(source_db_reference):
    """Load and return the source DB configuration based on the reference."""
    config_path = os.getenv("SOURCE_DB_CONFIG_PATH")
    try:
        with open(config_path, 'r') as f:
            config_data = json.load(f)
            source_db_config = config_data.get(source_db_reference)
            if not source_db_config:
                raise ValueError(f"No configuration found for source_db_reference:
{source_db_reference}")
```

```
            return source_db_config
    except Exception as e:
        logger.error(f"Error loading source DB config: {str(e)}")
        raise e

def load_target_db_config(target_db_reference):
    """Load and return the target DB configuration based on the reference."""
    config_path = os.getenv("TARGET_DB_CONFIG_PATH")
    try:
        with open(config_path, 'r') as f:
            config_data = json.load(f)
            target_db_config = config_data.get(target_db_reference)
            if not target_db_config:
                raise ValueError(f"No configuration found for target_db_reference:
{target_db_reference}")
            return target_db_config
    except Exception as e:
        logger.error(f"Error loading target DB config: {str(e)}")
        raise e

def build_query(job_config):
    """Build and return the SQL query string for ingestion based on the job configuration."""
    table_name = job_config.get("source_table_name")
    filters = job_config.get("filters", "")

    if not table_name:
        raise ValueError("source_table_name is required in the job configuration.")

    query = f"SELECT * FROM {table_name}"
```

```
    if filters:
        query += f" WHERE {filters}"

    logger.info(f"Built query: {query}")
    return query

def run_ingestion(spark: SparkSession, job_config):
    """Execute the ingestion process."""
    try:
        # Load source and target DB configurations
        source_db_config = load_source_db_config(job_config["source_db_reference"])
        target_db_config = load_target_db_config(job_config["target_db_reference"])

        # Build query for ingestion
        query = build_query(job_config)
        logger.info(f"Running query for job {job_config['job_id']}: {query}")

        # Load data from source
        df = spark.read.format("jdbc")\
            .option("url", source_db_config['jdbc_url'])\
            .option("dbtable", f"({query}) as source_data")\
            .option("user", source_db_config['username'])\
            .option("password", source_db_config['password'])\
            .load()

        # Pre-ingestion validations
        validate_schema(df, job_config)
        validate_count(df, job_config)
        validate_data_freshness(df, job_config)
```

```
        # Apply partitioning and save the data
        partition_cols = job_config['target_partition_by'].split(',')
        df.write.format(job_config['target_output_format'])\
            .partitionBy(partition_cols)\
            .mode("append")\
            .save(target_db_config['bucket'])

        # Post-ingestion validation
        validate_post_ingestion(df, job_config)

        logger.info(f"Ingestion completed for job {job_config['job_id']}")

    except Exception as e:
        log_issue(f"Error in job {job_config['job_id']}: {str(e)}",
job_config['log_config_reference'])
        raise e

def run_parallel_ingestions(spark: SparkSession, job_configs: list, run_group_id: str):
    """Run multiple ingestion jobs in parallel within the same job run group ID."""
    logger.info(f"Starting parallel ingestions for run group: {run_group_id}")
    results = []

    with ThreadPoolExecutor() as executor:
        future_to_job = {executor.submit(run_ingestion, spark, job_config): job_config for
job_config in job_configs}

        for future in as_completed(future_to_job):
            job_config = future_to_job[future]
            try:
                future.result()  # This will raise an exception if the job failed
```

```
                results.append((job_config['job_id'], "Success"))
            except Exception as exc:
                logger.error(f"Job {job_config['job_id']} generated an exception: {exc}")
                results.append((job_config['job_id'], "Failed"))

    logger.info(f"All ingestions completed for run group: {run_group_id}")
    return results

def fetch_job_configs(spark: SparkSession, config_table: str) -> list:
    """
    Fetch job configurations from the job config table.

    :param spark: SparkSession instance
    :param config_table: Name of the job config table in the database
    :return: List of job configuration dictionaries
    """
    logger.info(f"Fetching job configurations from table: {config_table}")

    # Read the job configurations from the config table
    job_configs_df = spark.read.format("jdbc")\
        .option("url", os.getenv("JDBC_URL"))\
        .option("dbtable", config_table)\
        .option("user", os.getenv("DB_USERNAME"))\
        .option("password", os.getenv("DB_PASSWORD"))\
        .load()

    job_configs = job_configs_df.toPandas().to_dict(orient='records')  # Convert to list of
dictionaries
    logger.info(f"Fetched {len(job_configs)} job configurations.")
    return job_configs
```

```python
# Sample usage
if __name__ == "__main__":
    spark = SparkSession.builder.appName("DataIngestion").getOrCreate()

    # Fetch job configurations from the specified job config table
    config_table = "job_config_table"  # Replace with your actual job config table name
    job_configs = fetch_job_configs(spark, config_table)

    # Define the run group ID
    run_group_id = "run_group_001"

    # Run the ingestion jobs in parallel
    run_parallel_ingestions(spark, job_configs, run_group_id)
```

Key Updates

1. **Fetching Job Configurations**:
    - The `fetch_job_configs` function reads job configurations from a specified database table using JDBC. It retrieves the configurations into a DataFrame, which is then converted to a list of dictionaries.
2. **Database Credentials**:
    - The function uses environment variables (`JDBC_URL`, `DB_USERNAME`, and `DB_PASSWORD`) to fetch the database connection details, making it flexible and secure.
3. **Sample Usage**:
    - In the `__main__` block, the script now fetches job configurations from the specified `job_config_table`instead of hardcoding them.

Assumptions

- Ensure the database table (`job_config_table`) contains the required fields that match the expected structure in your job configurations, including:
  - `job_id`
  - `source_db_reference`
  - `target_db_reference`
  - `source_table_name`
  - `filters` (optional)
  - `target_partition_by`
  - `target_output_format`
  - `log_config_reference`

Environment Variables

- Ensure the following environment variables are set correctly in your execution environment:
  - `SOURCE_DB_CONFIG_PATH`: Path to the JSON file containing source database configurations.
  - `TARGET_DB_CONFIG_PATH`: Path to the JSON file containing target database configurations.
  - `JDBC_URL`: JDBC URL for connecting to your database.
  - `DB_USERNAME`: Username for the database.
  - `DB_PASSWORD`: Password for the database.

———

`validation_utils.py` that uses PyDeequ for data validation in a Spark environment, we can integrate common validation checks such as schema validation, record count validation, data freshness validation, and post-ingestion validations. PyDeequ is a powerful tool for automating data quality checks in Spark, and we'll use it to build a set of utilities that can be customized as needed.

Below is the implementation of `validation_utils.py` using PyDeequ for robust data validation:

`validation_utils.py`

python
Copy code

```
import logging
```

```python
from pyspark.sql import DataFrame
from pydeequ.checks import Check, CheckLevel
from pydeequ.verification import VerificationSuite
from pydeequ.suggestions import ConstraintSuggestionRunner, Rules
from pydeequ.analyzers import *
from pydeequ.repository import FileSystemMetricsRepository, ResultKey
from pydeequ.verification import VerificationResult
from datetime import datetime

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Function to validate schema
def validate_schema(df: DataFrame, job_config: dict) -> bool:
    """
    Validate that the DataFrame schema matches the expected schema in the job configuration.
    :param df: Input DataFrame
    :param job_config: Job configuration with expected schema details
    :return: True if schema is valid, False otherwise
    """
    logger.info(f"Validating schema for job: {job_config['job_id']}")
    expected_schema = job_config.get("expected_schema", {})

    if not expected_schema:
        logger.warning(f"No expected schema provided for job {job_config['job_id']}")
        return True

    actual_schema = {field.name: field.dataType for field in df.schema.fields}
    mismatches = []
```

```
    for col, data_type in expected_schema.items():
        if col not in actual_schema:
            mismatches.append(f"Column {col} is missing")
        elif str(actual_schema[col]) != data_type:
            mismatches.append(f"Column {col} has incorrect type {actual_schema[col]}
(expected {data_type})")

    if mismatches:
        logger.error(f"Schema validation failed for job {job_config['job_id']}:
{mismatches}")
        return False

    logger.info(f"Schema validation passed for job {job_config['job_id']}")
    return True

# Function to validate record count
def validate_count(df: DataFrame, job_config: dict) -> bool:
    """
    Validate that the DataFrame record count matches the expected range in the job
configuration.
    :param df: Input DataFrame
    :param job_config: Job configuration with expected record count details
    :return: True if record count is valid, False otherwise
    """
    logger.info(f"Validating record count for job: {job_config['job_id']}")
    expected_min_count = job_config.get("expected_min_count", 0)
    expected_max_count = job_config.get("expected_max_count", float('inf'))

    record_count = df.count()
```

```
    if not (expected_min_count <= record_count <= expected_max_count):
        logger.error(f"Record count validation failed for job {job_config['job_id']}:
{record_count} records found, expected between {expected_min_count} and
{expected_max_count}")
        return False

    logger.info(f"Record count validation passed for job {job_config['job_id']}:
{record_count} records found")
    return True

# Function to validate data freshness (based on a timestamp column)
def validate_data_freshness(df: DataFrame, job_config: dict) -> bool:
    """
    Validate that the DataFrame contains data within the expected freshness window.
    :param df: Input DataFrame
    :param job_config: Job configuration with freshness validation details
    :return: True if data is fresh, False otherwise
    """
    logger.info(f"Validating data freshness for job: {job_config['job_id']}")
    freshness_column = job_config.get("freshness_column")
    freshness_threshold = job_config.get("freshness_threshold_days", 1)

    if not freshness_column:
        logger.warning(f"No freshness column provided for job {job_config['job_id']}")
        return True

    max_timestamp = df.agg({freshness_column: "max"}).collect()[0][0]
    if max_timestamp is None:
        logger.error(f"No data in the freshness column {freshness_column} for job
{job_config['job_id']}")
```

```
        return False

    current_date = datetime.now()
    freshness_diff = (current_date - max_timestamp).days

    if freshness_diff > freshness_threshold:
        logger.error(f"Data freshness validation failed for job {job_config['job_id']}: Last
data is {freshness_diff} days old, expected within {freshness_threshold} days")
        return False

    logger.info(f"Data freshness validation passed for job {job_config['job_id']}: Last data
is {freshness_diff} days old")
    return True

# Function to perform additional post-ingestion validations
def validate_post_ingestion(df: DataFrame, job_config: dict) -> bool:
    """
    Perform post-ingestion data quality checks using PyDeequ.
    :param df: Input DataFrame after ingestion
    :param job_config: Job configuration for the validation
    :return: True if post-ingestion validation passes, False otherwise
    """
    logger.info(f"Performing post-ingestion validation for job: {job_config['job_id']}")

    # Define a check suite for data validation
    check = Check(spark_session=df._sc, level=CheckLevel.Error, description=f"Post-ingestion
validation for {job_config['job_id']}")

    # Add specific checks based on job configuration
    check_is_not_empty = job_config.get("check_is_not_empty", True)
```

```
    check_null_percentage = job_config.get("check_null_percentage", {})

    # Check if DataFrame is not empty
    if check_is_not_empty:
        check.hasSize(lambda x: x > 0, "Data is empty")

    # Check null percentage for specific columns
    for col, max_null_pct in check_null_percentage.items():
        check.hasMax(fractionNull(col), lambda x: x <= max_null_pct, f"Too many nulls in
column {col}")

    # Run the verification suite
    verification_result =
VerificationSuite(spark_session=df._sc).onData(df).addCheck(check).run()

    # Parse the verification result
    result = VerificationResult.successMetricsAsDataFrame(df._sc, verification_result)
    result.show(truncate=False)

    if verification_result.status != "Success":
        logger.error(f"Post-ingestion validation failed for job {job_config['job_id']}")
        return False

    logger.info(f"Post-ingestion validation passed for job {job_config['job_id']}")
    return True
```

Detailed Breakdown of Each Function

1. **Schema Validation (`validate_schema`):**
   - Compares the actual schema of the DataFrame to the expected schema provided in the job configuration.

    - Ensures that all required columns are present and have the correct data types.
2. **Record Count Validation (`validate_count`):**
    - Checks that the number of records in the DataFrame is within the expected range (`expected_min_count` and `expected_max_count`) specified in the job configuration.
    - Logs a validation failure if the record count falls outside of this range.
3. **Data Freshness Validation (`validate_data_freshness`):**
    - Checks if the most recent record (based on a timestamp column) is within an acceptable freshness window (`freshness_threshold_days`).
    - This helps to ensure that the data being ingested is not too old.
4. **Post-Ingestion Validation (`validate_post_ingestion`):**
    - Uses PyDeequ to run a series of data quality checks post-ingestion.
    - Can check if the data is not empty, and validate null percentages in specific columns.
    - This step leverages PyDeequ's `VerificationSuite` and `Check` API to ensure that the data conforms to quality rules defined in the job configuration.

PyDeequ Installation

To use PyDeequ, make sure that you have installed it. You can install PyDeequ using the following command:

bash
Copy code
```
pip install pydeequ
```
Or include it in your `requirements.txt` file for your Spark job if you're running this in a cluster.

Example Job Configurations

Here is an example of what the job configuration might look like for validation purposes:

json
Copy code
```
{
```

```
  "job_id": "job_123",
  "expected_schema": {
    "id": "IntegerType",
    "name": "StringType",
    "created_at": "TimestampType"
  },
  "expected_min_count": 100,
  "expected_max_count": 1000,
  "freshness_column": "created_at",
  "freshness_threshold_days": 1,
  "check_is_not_empty": true,
  "check_null_percentage": {
    "name": 0.1,
    "id": 0.05
  }
}
```

How to Use

The `validation_utils.py` functions can be called from within your data ingestion script, ensuring that the data pipeline meets the quality standards at each step. Here's how they can be integrated:

python
Copy code

```
from utils.validation_utils import validate_schema, validate_count, validate_data_freshness,
validate_post_ingestion

# Example usage
validate_schema(df, job_config)
validate_count(df, job_config)
validate_data_freshness(df, job_config)
```

```
validate_post_ingestion(df, job_config)
```

CDC (Change Data Capture) values, we can store the CDC metadata in an Iceberg table, which will allow us to leverage Iceberg's powerful table management features such as schema evolution, partitioning, and versioned data storage. Instead of storing CDC values in a relational database, we will store them in an Iceberg table that tracks the job's `cdc_field` and the `last_cdc_value` after each job run.

Design Approach:

1. **Create a CDC Iceberg Table**: The Iceberg table will store the `job_id`, `cdc_field`, and `last_cdc_value` for each job.
2. **Load CDC State**: At the beginning of each job, retrieve the last CDC value from the Iceberg table.
3. **Save CDC State**: After each job completes, save the latest CDC value back into the Iceberg table for future use.

Updated `ingestion_utils.py` with Iceberg CDC Tracking:

python
Copy code

```python
import logging
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from utils.logging_utils import log_issue
from utils.validation_utils import validate_schema, validate_count, validate_data_freshness,
validate_post_ingestion
from utils.partition_utils import apply_partitioning
from datetime import datetime

# Setup logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Iceberg CDC Tracking Table Configuration
```

```
CDC_TRACKING_TABLE = "metadata.cdc_tracking"

def load_last_cdc_value(spark: SparkSession, job_id, cdc_field):
    """
    Load the last CDC value for the given job from the Iceberg CDC tracking table.
    :param spark: SparkSession
    :param job_id: The job ID for which the CDC value is tracked
    :param cdc_field: The CDC field (timestamp or ID) being tracked
    :return: The last CDC value, or None if no value is found
    """
    # Check if the CDC tracking table exists
    if not spark.catalog.tableExists(CDC_TRACKING_TABLE):
        logger.info(f"CDC tracking table {CDC_TRACKING_TABLE} does not exist. Starting
fresh.")
        return None

    # Query the Iceberg table for the last CDC value for this job and cdc_field
    cdc_df = spark.read.format("iceberg").load(CDC_TRACKING_TABLE)\
        .filter(F.col("job_id") == job_id)\
        .filter(F.col("cdc_field") == cdc_field)\
        .orderBy(F.col("updated_at").desc())\
        .limit(1)

    cdc_value_row = cdc_df.collect()
    if cdc_value_row:
        last_cdc_value = cdc_value_row[0]['last_cdc_value']
        logger.info(f"Last CDC value for job {job_id}: {last_cdc_value}")
        return last_cdc_value
    else:
        logger.info(f"No CDC value found for job {job_id}. Starting fresh.")
```

```
        return None

def save_cdc_value(spark: SparkSession, job_id, cdc_field, last_cdc_value):
    """
    Save the latest CDC value for the given job in the Iceberg CDC tracking table.
    :param spark: SparkSession
    :param job_id: The job ID for which the CDC value is tracked
    :param cdc_field: The CDC field (timestamp or ID) being tracked
    :param last_cdc_value: The last ingested CDC value to be stored
    """
    # Create the DataFrame to be saved
    cdc_df = spark.createDataFrame(
        [(job_id, cdc_field, last_cdc_value, datetime.now())],
        ["job_id", "cdc_field", "last_cdc_value", "updated_at"]
    )

    # Write the CDC value to the Iceberg table (append mode)
    cdc_df.write.format("iceberg").mode("append").save(CDC_TRACKING_TABLE)
    logger.info(f"CDC value {last_cdc_value} for field {cdc_field} saved for job {job_id}")

def build_query(job_config, last_cdc_value=None):
    """
    Build the ingestion query based on the job configuration.
    :param job_config: The job configuration containing the source table, cdc field, etc.
    :param last_cdc_value: The last ingested CDC value for incremental load
    :return: The SQL query to load the data
    """
    source_table = job_config['source_table']
    cdc_field = job_config.get("cdc_field", "last_updated")
```

```
    if last_cdc_value:
        query = f"SELECT * FROM {source_table} WHERE {cdc_field} > '{last_cdc_value}'"
    else:
        query = f"SELECT * FROM {source_table}"

    logger.info(f"Built query: {query}")
    return query

def run_ingestion(spark: SparkSession, job_config):
    try:
        # Load source and target DB configurations
        source_db_config = load_source_db_config(job_config["source_db_reference"])
        target_db_config = load_target_db_config(job_config["target_db_reference"])

        # Load the last CDC value
        last_cdc_value = load_last_cdc_value(spark, job_config["job_id"],
job_config.get("cdc_field", "last_updated"))

        # Build query for ingestion (use CDC value for incremental load)
        query = build_query(job_config, last_cdc_value)
        logger.info(f"Running query: {query}")

        # Load data from source
        df = spark.read.format("jdbc") \
            .option("url", source_db_config['jdbc_url']) \
            .option("dbtable", f"({query}) as source_data") \
            .option("user", source_db_config['username']) \
            .option("password", source_db_config['password']) \
            .load()
```

```
        # Pre-ingestion validations
        validate_schema(df, job_config)
        validate_count(df, job_config)
        validate_data_freshness(df, job_config)

        # Apply partitioning and write data to the target
        partition_cols = job_config['target_partition_by'].split(',')
        df.write.format(job_config['target_output_format']) \
            .partitionBy(partition_cols) \
            .mode("append") \
            .save(target_db_config['bucket'])

        # Get the last CDC value from the current ingestion (e.g., max timestamp or max ID)
        last_cdc_value_ingested = df.agg({job_config.get("cdc_field", "last_updated"):
"max"}).collect()[0][0]

        # Save the CDC value for future incremental loads
        save_cdc_value(spark, job_config["job_id"], job_config.get("cdc_field",
"last_updated"), last_cdc_value_ingested)

        # Post-ingestion validation
        validate_post_ingestion(df, job_config)

    except Exception as e:
        log_issue(f"Error in job {job_config['job_id']}: {str(e)}",
job_config['log_config_reference'])
        raise e

# Example function to load source DB configuration
def load_source_db_config(source_db_reference):
```

```
    # Dummy function, replace with actual DB config logic
    return {
        'jdbc_url': 'jdbc:mysql://source_db_host:3306/source_db',
        'username': 'source_user',
        'password': 'source_pass'
    }

# Example function to load target DB configuration
def load_target_db_config(target_db_reference):
    # Dummy function, replace with actual DB config logic
    return {
        'bucket': 's3://target-bucket/data',
        'target_output_format': 'parquet'
    }
```

Key Changes:

1. **CDC Tracking with Iceberg**:
    - `load_last_cdc_value`: Reads the last ingested CDC value for the job from the Iceberg table. If no value exists, the job starts fresh.
    - `save_cdc_value`: Saves the CDC field and its last ingested value to the Iceberg table. This allows the ingestion job to pick up from where it left off in subsequent runs.
    - The Iceberg CDC tracking table (`metadata.cdc_tracking`) stores each job's CDC state.
2. **Spark SQL**:
    - The CDC Iceberg table is queried via Spark SQL, and the CDC values are stored and retrieved through this table for each job.

Example Schema for the Iceberg `cdc_tracking` Table:

sql

Copy code

```sql
CREATE TABLE metadata.cdc_tracking (
```

```
    job_id STRING,
    cdc_field STRING,
    last_cdc_value STRING,
    updated_at TIMESTAMP
) USING iceberg;
```
This Iceberg table will track the last CDC value for each `job_id` and `cdc_field`, ensuring that incremental data loads continue from where the last ingestion left off.

Benefits of Using Iceberg for CDC:

- **Version Control**: Iceberg maintains metadata for all table versions, so tracking CDC values fits well with its architecture.
- **Scalability**: Iceberg can handle large-scale CDC metadata for multiple jobs.
- **Schema Evolution**: Iceberg supports schema evolution, allowing you to add fields like `updated_at` without breaking existing queries.

----

restartability of failed jobs without re-triggering successful ones, we can make a few modifications to the `ingestion_utils.py` script. We will track the status of each job execution (success or failure) in a job tracking table. Failed jobs can be re-executed on subsequent runs, while successful jobs will not be triggered again.

Key Steps for Restartability:

1. **Job Tracking Table**: Track the execution status (e.g., `SUCCESS`, `FAILED`) and job execution timestamps.
2. **Restart Failed Jobs**: Only pick up jobs that have failed or haven't been executed yet.
3. **Status Update**: After a job runs, update the job tracking table with the result (`SUCCESS` or `FAILED`).
4. **Skip Successful Jobs**: Jobs marked as `SUCCESS` will not be re-executed in subsequent runs.

Updated `ingestion_utils.py` with Restartability:

python
Copy code
```python
import logging
from pyspark.sql import SparkSession
```

```
from concurrent.futures import ThreadPoolExecutor, as_completed
from utils.logging_utils import log_issue
from utils.validation_utils import validate_schema, validate_count, validate_data_freshness,
validate_post_ingestion
from utils.partition_utils import apply_partitioning
from datetime import datetime
from pyspark.sql import functions as F

# Setup logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Iceberg CDC Tracking and Job Tracking Table Configuration
CDC_TRACKING_TABLE = "metadata.cdc_tracking"
JOB_TRACKING_TABLE = "metadata.job_tracking"

def load_last_cdc_value(spark: SparkSession, job_id, cdc_field):
    """
    Load the last CDC value for the given job from the Iceberg CDC tracking table.
    """
    if not spark.catalog.tableExists(CDC_TRACKING_TABLE):
        logger.info(f"CDC tracking table {CDC_TRACKING_TABLE} does not exist. Starting
fresh.")
        return None

    cdc_df = spark.read.format("iceberg").load(CDC_TRACKING_TABLE)\
        .filter(F.col("job_id") == job_id)\
        .filter(F.col("cdc_field") == cdc_field)\
        .orderBy(F.col("updated_at").desc())\
        .limit(1)
```

```python
    cdc_value_row = cdc_df.collect()
    if cdc_value_row:
        last_cdc_value = cdc_value_row[0]['last_cdc_value']
        logger.info(f"Last CDC value for job {job_id}: {last_cdc_value}")
        return last_cdc_value
    else:
        logger.info(f"No CDC value found for job {job_id}. Starting fresh.")
        return None

def save_cdc_value(spark: SparkSession, job_id, cdc_field, last_cdc_value):
    """
    Save the latest CDC value for the given job in the Iceberg CDC tracking table.
    """
    cdc_df = spark.createDataFrame(
        [(job_id, cdc_field, last_cdc_value, datetime.now())],
        ["job_id", "cdc_field", "last_cdc_value", "updated_at"]
    )

    cdc_df.write.format("iceberg").mode("append").save(CDC_TRACKING_TABLE)
    logger.info(f"CDC value {last_cdc_value} for field {cdc_field} saved for job {job_id}")

def update_job_status(spark: SparkSession, job_id, status):
    """
    Update the job status (SUCCESS/FAILED) in the job tracking table.
    """
    job_tracking_df = spark.createDataFrame(
        [(job_id, status, datetime.now())],
        ["job_id", "status", "updated_at"]
    )
```

```
    job_tracking_df.write.format("iceberg").mode("append").save(JOB_TRACKING_TABLE)
    logger.info(f"Job {job_id} marked as {status}")

def get_jobs_to_run(spark: SparkSession, job_run_group_id):
    """
    Get jobs that have failed or have not been executed successfully yet.
    """
    if not spark.catalog.tableExists(JOB_TRACKING_TABLE):
        logger.info(f"Job tracking table {JOB_TRACKING_TABLE} does not exist. All jobs will
be executed.")
        return spark.sql(f"SELECT * FROM job_config WHERE job_run_group_id =
{job_run_group_id}").collect()

    # Get the last status for each job within the job_run_group_id
    job_tracking_df = spark.read.format("iceberg").load(JOB_TRACKING_TABLE)\
        .filter(F.col("job_run_group_id") == job_run_group_id)\
        .groupBy("job_id").agg(F.max("updated_at").alias("last_run_at"))\
        .join(spark.read.format("iceberg").load(JOB_TRACKING_TABLE), on=["job_id",
"updated_at"], how="left")

    # Filter jobs that are either failed or have no previous entry in job tracking
    jobs_to_run_df = spark.sql(f"""
        SELECT *
        FROM job_config jc
        LEFT JOIN ({job_tracking_df}) jt
        ON jc.job_id = jt.job_id
        WHERE job_run_group_id = {job_run_group_id}
        AND (jt.status IS NULL OR jt.status = 'FAILED')
    """)
```

```
    return jobs_to_run_df.collect()

def build_query(job_config, last_cdc_value=None):
    """
    Build the ingestion query based on the job configuration.
    """
    source_table = job_config['source_table']
    cdc_field = job_config.get("cdc_field", "last_updated")

    if last_cdc_value:
        query = f"SELECT * FROM {source_table} WHERE {cdc_field} > '{last_cdc_value}'"
    else:
        query = f"SELECT * FROM {source_table}"

    logger.info(f"Built query: {query}")
    return query

def run_ingestion(spark: SparkSession, job_config):
    """
    Runs the ingestion process for a single job.
    """
    try:
        # Load source and target DB configurations
        source_db_config = load_source_db_config(job_config["source_db_reference"])
        target_db_config = load_target_db_config(job_config["target_db_reference"])

        # Load the last CDC value
        last_cdc_value = load_last_cdc_value(spark, job_config["job_id"],
job_config.get("cdc_field", "last_updated"))
```

```
        # Build query for ingestion (use CDC value for incremental load)
        query = build_query(job_config, last_cdc_value)
        logger.info(f"Running query: {query}")

        # Load data from source
        df = spark.read.format("jdbc") \
            .option("url", source_db_config['jdbc_url']) \
            .option("dbtable", f"({query}) as source_data") \
            .option("user", source_db_config['username']) \
            .option("password", source_db_config['password']) \
            .load()

        # Pre-ingestion validations
        validate_schema(df, job_config)
        validate_count(df, job_config)
        validate_data_freshness(df, job_config)

        # Apply partitioning and write data to the target
        partition_cols = job_config['target_partition_by'].split(',')
        df.write.format(job_config['target_output_format']) \
            .partitionBy(partition_cols) \
            .mode("append") \
            .save(target_db_config['bucket'])

        # Get the last CDC value from the current ingestion (e.g., max timestamp or max ID)
        last_cdc_value_ingested = df.agg({job_config.get("cdc_field", "last_updated"):
"max"}).collect()[0][0]

        # Save the CDC value for future incremental loads
```

```
        save_cdc_value(spark, job_config["job_id"], job_config.get("cdc_field",
"last_updated"), last_cdc_value_ingested)

        # Post-ingestion validation
        validate_post_ingestion(df, job_config)

        # Mark job as successful
        update_job_status(spark, job_config["job_id"], "SUCCESS")

    except Exception as e:
        log_issue(f"Error in job {job_config['job_id']}: {str(e)}",
job_config['log_config_reference'])
        update_job_status(spark, job_config["job_id"], "FAILED")
        raise e

def run_parallel_jobs(spark: SparkSession, job_run_group_id):
    """
    Run all jobs within a job_run_group_id in parallel.
    """
    try:
        # Get jobs that need to be run (failed or not executed successfully)
        job_configs = get_jobs_to_run(spark, job_run_group_id)

        if not job_configs:
            logger.info(f"No jobs to run for job_run_group_id {job_run_group_id}.")
            return

        # Use ThreadPoolExecutor for parallel execution of jobs
        with ThreadPoolExecutor(max_workers=len(job_configs)) as executor:
```

```
            future_to_job = {executor.submit(run_ingestion, spark, job_config):
job_config['job_id'] for job_config in job_configs}

            for future in as_completed(future_to_job):
                job_id = future_to_job[future]
                try:
                    future.result()  # This will raise an exception if the job failed
                    logger.info(f"Job {job_id} completed successfully.")
                except Exception as e:
                    logger.error(f"Job {job_id} failed with error: {str(e)}")

    except Exception as e:
        log_issue(f"Error in running parallel jobs for job_run_group_id {job_run_group_id}:
{str(e)}", None)
        raise e

# Example function to load source DB configuration
def load_source_db_config(source_db_reference):
    # Dummy function, replace with actual DB config logic
    return {
        'jdbc_url': 'jdbc:mysql://source_db_host:3306
```

To implement restartability of failed jobs without re-triggering successful ones, we can make a few modifications to the `ingestion_utils.py` script. We will track the status of each job execution (success or failure) in a job tracking table. Failed jobs can be re-executed on subsequent runs, while successful jobs will not be triggered again.

Key Steps for Restartability:

1. **Job Tracking Table**: Track the execution status (e.g., `SUCCESS`, `FAILED`) and job execution timestamps.

2. **Restart Failed Jobs**: Only pick up jobs that have failed or haven't been executed yet.
3. **Status Update**: After a job runs, update the job tracking table with the result (`SUCCESS` or `FAILED`).
4. **Skip Successful Jobs**: Jobs marked as `SUCCESS` will not be re-executed in subsequent runs.

Updated `ingestion_utils.py` with Restartability:

python
Copy code

```
import logging
from pyspark.sql import SparkSession
from concurrent.futures import ThreadPoolExecutor, as_completed
from utils.logging_utils import log_issue
from utils.validation_utils import validate_schema, validate_count, validate_data_freshness,
validate_post_ingestion
from utils.partition_utils import apply_partitioning
from datetime import datetime
from pyspark.sql import functions as F

# Setup logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Iceberg CDC Tracking and Job Tracking Table Configuration
CDC_TRACKING_TABLE = "metadata.cdc_tracking"
JOB_TRACKING_TABLE = "metadata.job_tracking"

def load_last_cdc_value(spark: SparkSession, job_id, cdc_field):
    """
    Load the last CDC value for the given job from the Iceberg CDC tracking table.
    """
    if not spark.catalog.tableExists(CDC_TRACKING_TABLE):
```

```
        logger.info(f"CDC tracking table {CDC_TRACKING_TABLE} does not exist. Starting 
fresh.")
        return None

    cdc_df = spark.read.format("iceberg").load(CDC_TRACKING_TABLE)\
        .filter(F.col("job_id") == job_id)\
        .filter(F.col("cdc_field") == cdc_field)\
        .orderBy(F.col("updated_at").desc())\
        .limit(1)

    cdc_value_row = cdc_df.collect()
    if cdc_value_row:
        last_cdc_value = cdc_value_row[0]['last_cdc_value']
        logger.info(f"Last CDC value for job {job_id}: {last_cdc_value}")
        return last_cdc_value
    else:
        logger.info(f"No CDC value found for job {job_id}. Starting fresh.")
        return None

def save_cdc_value(spark: SparkSession, job_id, cdc_field, last_cdc_value):
    """
    Save the latest CDC value for the given job in the Iceberg CDC tracking table.
    """
    cdc_df = spark.createDataFrame(
        [(job_id, cdc_field, last_cdc_value, datetime.now())],
        ["job_id", "cdc_field", "last_cdc_value", "updated_at"]
    )

    cdc_df.write.format("iceberg").mode("append").save(CDC_TRACKING_TABLE)
    logger.info(f"CDC value {last_cdc_value} for field {cdc_field} saved for job {job_id}")
```

```
def update_job_status(spark: SparkSession, job_id, status):
    """
    Update the job status (SUCCESS/FAILED) in the job tracking table.
    """
    job_tracking_df = spark.createDataFrame(
        [(job_id, status, datetime.now())],
        ["job_id", "status", "updated_at"]
    )

    job_tracking_df.write.format("iceberg").mode("append").save(JOB_TRACKING_TABLE)
    logger.info(f"Job {job_id} marked as {status}")

def get_jobs_to_run(spark: SparkSession, job_run_group_id):
    """
    Get jobs that have failed or have not been executed successfully yet.
    """
    if not spark.catalog.tableExists(JOB_TRACKING_TABLE):
        logger.info(f"Job tracking table {JOB_TRACKING_TABLE} does not exist. All jobs will
be executed.")
        return spark.sql(f"SELECT * FROM job_config WHERE job_run_group_id =
{job_run_group_id}").collect()

    # Get the last status for each job within the job_run_group_id
    job_tracking_df = spark.read.format("iceberg").load(JOB_TRACKING_TABLE)\
        .filter(F.col("job_run_group_id") == job_run_group_id)\
        .groupBy("job_id").agg(F.max("updated_at").alias("last_run_at"))\
        .join(spark.read.format("iceberg").load(JOB_TRACKING_TABLE), on=["job_id",
"updated_at"], how="left")
```

```python
    # Filter jobs that are either failed or have no previous entry in job tracking
    jobs_to_run_df = spark.sql(f"""
        SELECT *
        FROM job_config jc
        LEFT JOIN ({job_tracking_df}) jt
        ON jc.job_id = jt.job_id
        WHERE job_run_group_id = {job_run_group_id}
        AND (jt.status IS NULL OR jt.status = 'FAILED')
    """)

    return jobs_to_run_df.collect()

def build_query(job_config, last_cdc_value=None):
    """
    Build the ingestion query based on the job configuration.
    """
    source_table = job_config['source_table']
    cdc_field = job_config.get("cdc_field", "last_updated")

    if last_cdc_value:
        query = f"SELECT * FROM {source_table} WHERE {cdc_field} > '{last_cdc_value}'"
    else:
        query = f"SELECT * FROM {source_table}"

    logger.info(f"Built query: {query}")
    return query

def run_ingestion(spark: SparkSession, job_config):
    """
    Runs the ingestion process for a single job.
```

```
    """
    try:
        # Load source and target DB configurations
        source_db_config = load_source_db_config(job_config["source_db_reference"])
        target_db_config = load_target_db_config(job_config["target_db_reference"])

        # Load the last CDC value
        last_cdc_value = load_last_cdc_value(spark, job_config["job_id"],
job_config.get("cdc_field", "last_updated"))

        # Build query for ingestion (use CDC value for incremental load)
        query = build_query(job_config, last_cdc_value)
        logger.info(f"Running query: {query}")

        # Load data from source
        df = spark.read.format("jdbc") \
            .option("url", source_db_config['jdbc_url']) \
            .option("dbtable", f"({query}) as source_data") \
            .option("user", source_db_config['username']) \
            .option("password", source_db_config['password']) \
            .load()

        # Pre-ingestion validations
        validate_schema(df, job_config)
        validate_count(df, job_config)
        validate_data_freshness(df, job_config)

        # Apply partitioning and write data to the target
        partition_cols = job_config['target_partition_by'].split(',')
        df.write.format(job_config['target_output_format']) \
```

```
            .partitionBy(partition_cols) \
            .mode("append") \
            .save(target_db_config['bucket'])

        # Get the last CDC value from the current ingestion (e.g., max timestamp or max ID)
        last_cdc_value_ingested = df.agg({job_config.get("cdc_field", "last_updated"):
"max"}).collect()[0][0]

        # Save the CDC value for future incremental loads
        save_cdc_value(spark, job_config["job_id"], job_config.get("cdc_field",
"last_updated"), last_cdc_value_ingested)

        # Post-ingestion validation
        validate_post_ingestion(df, job_config)

        # Mark job as successful
        update_job_status(spark, job_config["job_id"], "SUCCESS")

    except Exception as e:
        log_issue(f"Error in job {job_config['job_id']}: {str(e)}",
job_config['log_config_reference'])
        update_job_status(spark, job_config["job_id"], "FAILED")
        raise e

def run_parallel_jobs(spark: SparkSession, job_run_group_id):
    """
    Run all jobs within a job_run_group_id in parallel.
    """
    try:
        # Get jobs that need to be run (failed or not executed successfully)
```

```
        job_configs = get_jobs_to_run(spark, job_run_group_id)

        if not job_configs:
            logger.info(f"No jobs to run for job_run_group_id {job_run_group_id}.")
            return

        # Use ThreadPoolExecutor for parallel execution of jobs
        with ThreadPoolExecutor(max_workers=len(job_configs)) as executor:
            future_to_job = {executor.submit(run_ingestion, spark, job_config):
job_config['job_id'] for job_config in job_configs}

            for future in as_completed(future_to_job):
                job_id = future_to_job[future]
                try:
                    future.result()  # This will raise an exception if the job failed
                    logger.info(f"Job {job_id} completed successfully.")
                except Exception as e:
                    logger.error(f"Job {job_id} failed with error: {str(e)}")

    except Exception as e:
        log_issue(f"Error in running parallel jobs for job_run_group_id {job_run_group_id}:
{str(e)}", None)
        raise e

# Example function to load source DB configuration
def load_source_db_config(source_db_reference):
    # Dummy function, replace with actual DB config logic
    return {
        'jdbc_url': 'jdbc:mysql://source_db_host:3306/source_db',
        'username': 'source_user',
```

```
        'password': 'source_pass'
    }

# Example function to load target DB configuration
def load_target_db_config(target_db_reference):
    # Dummy function, replace with actual DB config logic
    return {
        'bucket': 's3://target-bucket/data/',
        'target_output_format': 'parquet'
    }
```

Key Changes:

1. **Job Tracking Table (`JOB_TRACKING_TABLE`)**:
   - Keeps track of the job's `status` (e.g., `SUCCESS`, `FAILED`) and the `job_id`.
   - When a job fails, its status is updated to `FAILED`. Successful jobs are marked as `SUCCESS`.
2. **Get Jobs to Run**:
   - The function `get_jobs_to_run` ensures only failed or new jobs are selected for execution.
   - It queries the `job_tracking` table to get the last execution status of each job and filters out successful jobs.
3. **Job Execution**:
   - For each job, ingestion is executed with validations, partitioning, and CDC value tracking.
   - If a job fails, it is marked as `FAILED` so that it can be re-executed later without affecting successful jobs.

With this design, you can restart only the failed jobs, allowing the framework to handle failure scenarios without re-triggering jobs that have already succeeded.

——-

Job Flow Diagram for Ingestion Process

1. **Start Job Run**
    - Trigger the job execution process with a specified `job_run_group_id`.
2. **Get Jobs to Run**
    - Query the job tracking table to retrieve jobs based on their status (i.e., `FAILED` or not previously executed).
3. **Iterate Through Jobs**
    - For each job that needs to be executed:
        - Load source and target database configurations.
        - Load the last CDC value from the Iceberg CDC tracking table.
        - Build the ingestion query using the last CDC value.
4. **Load Data from Source**
    - Execute the built query to load data from the source database into a DataFrame.
5. **Pre-Ingestion Validation**
    - Perform validations:
        - **Schema Validation**
        - **Count Validation**
        - **Data Freshness Validation**
6. **Data Processing**
    - Apply partitioning based on the job configuration.
    - Write the DataFrame to the target location (e.g., S3 bucket) in the specified format.
7. **Get Last CDC Value**
    - Extract the latest CDC value from the ingested DataFrame.
8. **Save CDC Value**
    - Store the latest CDC value in the Iceberg CDC tracking table for future incremental loads.

9. **Post-Ingestion Validation**
   - Perform post-ingestion validation to ensure data integrity.
10. **Update Job Status**
    - If ingestion is successful:
      - Update the job status in the job

——-

Transformation

enhance the Slowly Changing Dimension (SCD) Type 2 implementation with a hash of relevant columns for change detection, we can calculate a hash value for the combined key attributes of the dimension. This approach allows us to efficiently track changes and handle updates. Additionally, I will explain how to manage the detection of new columns in the data model.

Updated Implementation Using Hash for Change Detection

Here's how to modify the previous code to implement change detection using a hash of selected columns:

1. **Calculate the Hash of Relevant Columns**: We'll create a new column that computes a hash value from the relevant fields to detect changes.
2. **Check for New Columns**: We will keep track of the schema and compare it to the current schema in the dimension table to identify new columns.

Updated PySpark Code

Here’s how you can modify the dimension loading function with these enhancements:

python
Copy code

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, current_timestamp, lit, sha2, concat_ws
import logging

# Set up logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
logger = logging.getLogger(__name__)

# Initialize Spark session
spark = SparkSession.builder \
    .appName("Iceberg Standardization Layer with SCD Type 2 Using Hash") \
    .config("spark.sql.catalog.my_catalog", "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.my_catalog.type", "hive") \
    .config("spark.sql.catalog.my_catalog.warehouse", "s3://your_warehouse_path/") \
    .getOrCreate()

# Load config tables
dimension_config_df = spark.read.parquet("s3://your_path/dimension_config")
fact_config_df = spark.read.parquet("s3://your_path/fact_config")

def detect_new_columns(iceberg_table_name, source_df):
    # Get the existing schema from the Iceberg table
    existing_columns = set(spark.table(iceberg_table_name).columns)
    source_columns = set(source_df.columns)

    # Identify new columns
```

```
    new_columns = source_columns - existing_columns

    return new_columns

def load_dimension(dimension_row):
    try:
        # Load the source data for the dimension
        source_df = spark.read.parquet(f"s3://your_path/
source_{dimension_row['dimension_name']}_data")

        # Prepare to merge into the Iceberg table
        iceberg_table_name = f"my_catalog.{dimension_row['table_name']}"

        # Step 1: Detect new columns
        new_columns = detect_new_columns(iceberg_table_name, source_df)
        if new_columns:
            logger.warning(f"New columns detected in {dimension_row['dimension_name']}:
{new_columns}")

        # Step 2: Add a hash column for change detection
        source_df = source_df.withColumn("hash", sha2(concat_ws("||", *source_df.columns),
256))

        # Step 3: Create a temporary view for the source data
        source_df.createOrReplaceTempView("source_dim_data")

        # Step 4: MERGE statement for SCD Type 2 using hash for change detection
        merge_sql = f"""
        MERGE INTO {iceberg_table_name} AS target
        USING (
```

```
    SELECT customer_id, name, email, hash
    FROM source_dim_data
) AS source
ON target.customer_id = source.customer_id
WHEN MATCHED AND target.hash <> source.hash THEN
    UPDATE SET
        target.end_date = current_timestamp(),
        target.is_current = false,
        target.hash = source.hash
WHEN NOT MATCHED THEN
    INSERT (
        customer_id,
        name,
        email,
        start_date,
        end_date,
        is_current,
        hash
    ) VALUES (
        source.customer_id,
        source.name,
        source.email,
        current_timestamp(),
        NULL,
        true,
        source.hash
    )
"""

# Execute the merge operation
```

```
        spark.sql(merge_sql)

        logger.info(f"Successfully merged dimension with SCD Type 2:
{dimension_row['dimension_name']}")

    except Exception as e:
        logger.error(f"Error merging dimension {dimension_row['dimension_name']}: {str(e)}")

def load_fact(fact_row):
    # (The previous fact loading logic remains the same)
    pass

# Iterate over each dimension config row and load data
for row in dimension_config_df.collect():
    load_dimension(row)

# Iterate over each fact config row and load data
for row in fact_config_df.collect():
    load_fact(row)

# Stop the Spark session
spark.stop()
```

Key Enhancements

1. **Detect New Columns**: The `detect_new_columns` function compares the columns in the source DataFrame against the existing Iceberg table's schema. If new columns are detected, a warning is logged.

2. **Hash Calculation**: The hash for the relevant columns is calculated using the `sha2` function combined with `concat_ws`. The hash is computed for all columns, which helps identify changes efficiently.

3. **`MERGE` Logic**: In the `MERGE` statement:

    - The `WHEN MATCHED` clause checks if the hash of the current row in the target table differs from the source hash. If they differ, it updates the record to mark it as historical and sets the `end_date`.
    - The `WHEN NOT MATCHED` clause inserts new records along with their hashes.

Handling New Columns

- When new columns are detected, you may need to modify the schema of the Iceberg table accordingly. You can automate this process by adding additional logic to the `load_dimension` function. Here's an example approach:
    - For each new column, alter the Iceberg table schema using the `ALTER TABLE` statement in Spark SQL.

python
Copy code
```python
for column in new_columns:
    alter_sql = f"ALTER TABLE {iceberg_table_name} ADD COLUMN {column} STRING"  # Specify the correct type
    spark.sql(alter_sql)
```

Conclusion

This updated implementation effectively tracks changes using a hash of relevant columns and detects new columns in the incoming data. It leverages Iceberg's capabilities for efficient data management while implementing SCD Type 2 to maintain historical data.

1. Create `dimension_config` and `fact_config` tables in Iceberg

First, you will need to define and create these config tables in Iceberg. These tables will store the same metadata that was initially planned for CSV files but in Iceberg for easier access and scalability.

SQL Example to Create Config Tables in Iceberg:

sql
Copy code
```sql
-- Create dimension config table
```

```
CREATE TABLE iceberg_catalog.db.dimension_config (
  table_name STRING,
  primary_key_columns ARRAY<STRING>,
  hash_columns ARRAY<STRING>,
  scd_type STRING,
  scd_columns ARRAY<STRING>,
  additional_columns ARRAY<STRING>,
  transformation_rules STRING,
  schema_version INT
);

-- Create fact config table
CREATE TABLE iceberg_catalog.db.fact_config (
  table_name STRING,
  primary_key_columns ARRAY<STRING>,
  hash_columns ARRAY<STRING>,
  fact_columns ARRAY<STRING>,
  additional_columns ARRAY<STRING>,
  transformation_rules STRING,
  schema_version INT
);
```

- **`primary_key_columns`**: Columns used for the primary key.
- **`hash_columns`**: Columns used to detect changes (e.g., for SCD Type 2).
- **`scd_columns`**: Columns used in the Slowly Changing Dimension.
- **`transformation_rules`**: String or JSON with transformation logic.
- **`schema_version`**: Used to track schema changes and versioning.

2. PySpark Code to Read Config from Iceberg

You'll modify the existing code to read from the Iceberg tables rather than CSV files.

PySpark Code:

python
Copy code
```
from pyspark.sql import SparkSession

# Initialize Spark session
spark = SparkSession.builder \
    .appName("Standardization Layer") \
    .config("spark.sql.catalog.iceberg", "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.iceberg.catalog-impl", "org.apache.iceberg.aws.s3.S3Catalog")
\
    .config("spark.sql.catalog.iceberg.warehouse", "s3://your-bucket/warehouse") \
    .config("spark.sql.catalog.iceberg.io-impl", "org.apache.iceberg.aws.s3.S3FileIO") \
    .getOrCreate()

# Load dimension and fact config from Iceberg tables
dimension_config_df = spark.sql("SELECT * FROM iceberg_catalog.db.dimension_config")
fact_config_df = spark.sql("SELECT * FROM iceberg_catalog.db.fact_config")

# Show the loaded configs
dimension_config_df.show()
fact_config_df.show()

# Example: Use the config in your transformation logic
def process_dimension_table(config_row):
    table_name = config_row['table_name']
    primary_key_columns = config_row['primary_key_columns']
    hash_columns = config_row['hash_columns']
    scd_columns = config_row['scd_columns']
```

```
    transformation_rules = config_row['transformation_rules']

    # Apply transformation rules dynamically based on the config
    source_df = spark.sql(f"SELECT * FROM staging.{table_name}")

    # Implement transformation and validation logic (use primary key and hash columns)
    transformed_df = apply_transformations(source_df, transformation_rules)

    # Merge logic for SCD Type 2 or upsert logic
    merge_into_iceberg(transformed_df, table_name, primary_key_columns, hash_columns,
scd_columns)

def apply_transformations(df, transformation_rules):
    # Apply the transformations (this will depend on your specific rules)
    # You can parse the transformation_rules field from the config and apply them
    # For example: if transformation_rules is a JSON with field mappings
    return df  # Placeholder - implement your logic here

def merge_into_iceberg(transformed_df, table_name, primary_key_columns, hash_columns,
scd_columns):
    # Placeholder for the merge operation
    # Use Iceberg's MERGE INTO statement for SCD Type 2 handling

    transformed_df.createOrReplaceTempView("temp_view")

    # Example MERGE INTO statement in PySpark
    spark.sql(f"""
    MERGE INTO iceberg_catalog.db.{table_name} AS target
    USING temp_view AS source
    ON {" AND ".join([f"target.{col} = source.{col}" for col in primary_key_columns])}
```

```
    WHEN MATCHED AND {" OR ".join([f"target.{col} != source.{col}" for col in hash_columns])}
      THEN UPDATE SET *
    WHEN NOT MATCHED THEN
      INSERT *
    """)

# Process all dimension tables based on config
for row in dimension_config_df.collect():
    process_dimension_table(row)

# Process all fact tables similarly
def process_fact_table(config_row):
    # Similar processing logic for fact tables
    pass

for row in fact_config_df.collect():
    process_fact_table(row)

spark.stop()
```

Explanation:

- **`dimension_config_df`** **and** **`fact_config_df`**: These DataFrames are read from the Iceberg config tables.
- **`process_dimension_table`**: This function processes dimension tables based on the configuration. It reads the raw data from the staging area, applies the transformations, and merges into Iceberg tables using the config.
- **`apply_transformations`**: A placeholder function for applying transformations. You can implement this function based on your transformation rules.
- **`merge_into_iceberg`**: The `MERGE INTO` operation performs upserts based on the primary key and hash columns (for SCD Type 2).

3. Schema Versioning and Hash Storage

To track schema versioning and avoid rehashing, you can add a schema column and hash column to your target tables in Iceberg. This way, you can store the hash of relevant columns in the target table and avoid recomputing it if the schema has not changed.

Modifications to the Iceberg Tables:

When creating your target Iceberg tables (dimension or fact), you can add a column for schema versioning and another column to store the hash of relevant columns.

sql
Copy code
```
ALTER TABLE iceberg_catalog.db.<dimension_table>
ADD COLUMNS (
  hash_value STRING,
  schema_version INT
);
```

- **`hash_value`**: Store the computed hash of the relevant columns.
- **`schema_version`**: Store the version of the schema that was applied during this transformation.

When you read from the target table, you can compare the stored schema version and hash value to determine if the table needs to be reprocessed.