### Proposal: Efficient Handling of New Batches of Input Data to Avoid Overriding Existing Transactions

In data processing pipelines, especially when handling transaction data, it's crucial to avoid overwriting historical data from previous batches when processing new batches. The goal is to maintain the integrity of the existing data while incorporating new data. Here's a detailed proposal on how to handle this scenario efficiently.

### Problem Definition

You are dealing with batch processing of transaction data. Each batch includes new transactions that need to be merged with the historical data. The challenge is to ensure that:

- **New transactions are added** without overwriting existing ones.

- **Existing transactions are updated** only if necessary, based on business rules.

- **Historical data remains intact** unless explicitly updated.

### Key Considerations:

1. **Data Consistency**: Ensure that existing transactions are not overwritten unless necessary, and new transactions are seamlessly added.

2. **Primary Key Integrity**: Based on the provided schema, the primary key columns (`CONTRACT_SOURCE_SYSTEM`, `CONTRACT_SOURCE_SYSTEM_ID`, and `NSE_ID`) need to be used to identify unique transactions.

3. **Batch Processing**: Handle new batches of data incrementally so that the historical dataset grows and retains transactional consistency.

4. **Data Deduplication**: Ensure no duplicate records are created, based on the primary key.

5. **Performance**: Ensure the solution can scale as the data volume grows.

### Solution Overview

We propose using **Merge (Upsert)** functionality with **Partitioning** and **Versioning** to handle new batches of input data efficiently without overriding existing transactions.

### Solution Components

#### 1. **Parquet Storage with Partitioning**

- Store transaction data in **Parquet** format (which you are already doing), but ensure the data is **partitioned** based on key columns or dates (e.g., `BUSINESS_DATE`). This helps in efficiently accessing and updating the relevant portions of data when new batches arrive.

- **Partitioning** will also improve read/write performance and make the data scalable.

```python
# Partition data by BUSINESS_DATE to optimize querying and updates
transactions_df.write.mode("overwrite").partitionBy("BUSINESS_DATE").parquet("transactions_output.parquet")
```

#### 2. **Schema Evolution**

- Ensure the schema can evolve over time to accommodate new fields or modifications. Since Parquet supports schema evolution, adding new columns or changing data types can be done without affecting existing data.

- This ensures backward compatibility and smooth transitions when handling new fields in future batches.

#### 3. **Merge (Upsert) Operation**

For handling new batches of data, a **merge** (upsert) strategy should be employed. PySpark does not have a native "upsert" functionality, but you can simulate it using **Delta Lake**, which extends Spark's functionality and adds support for atomic operations like merge.

##### How Merge Works:

- **Insert**: Add new transactions from the incoming batch that do not exist in the historical data.

- **Update**: Modify existing transactions in the historical data if the new batch contains updated information for the same primary key.

- **No Action**: Leave existing transactions unchanged if they are not part of the new batch.

Example using Delta Lake (upsert operation):

```python
from delta.tables import *

# Load the historical data as a Delta table
delta_table = DeltaTable.forPath(spark, "path_to_existing_transactions")

# New batch of transactions as a DataFrame
new_batch_df = get_dataframes(spark, new_batch_file)
```

```
# Perform merge (upsert) operation based on primary keys
delta_table.alias("old").merge(
    new_batch_df.alias("new"),
    "old.CONTRACT_SOURCE_SYSTEM = new.CONTRACT_SOURCE_SYSTEM AND \
     old.CONTRACT_SOURCE_SYSTEM_ID = new.CONTRACT_SOURCE_SYSTEM_ID AND \
     old.NSE_ID = new.NSE_ID"
).whenMatchedUpdateAll().whenNotMatchedInsertAll().execute()
```

With this strategy:

- **New transactions** from the incoming batch that don’t exist in the historical dataset are **inserted**.
- **Existing transactions** that match the primary keys are **updated**.
- Transactions in the historical data that are not part of the new batch remain **unchanged**.

#### 4. **Incremental Data Loading**

- Process data **incrementally** by loading only the new batches and merging them into the historical data. This avoids processing the entire dataset each time a new batch arrives.
- Use timestamps or batch identifiers to track changes and process only the new or changed data. For example, use the `CREATION_DATE` or `SYSTEM_TIMESTAMP` column to filter out transactions already processed in previous batches.

```python
# Load only new transactions since the last batch
new_transactions_df = new_transactions_df.filter(col("SYSTEM_TIMESTAMP") >
last_processed_timestamp)
```

#### 5. **Data Versioning and Auditing**

- Implement **data versioning** to maintain a history of changes. This allows tracking the changes to transactions over time and ensures that if something goes wrong, you can revert to a previous version of the data.
- **Delta Lake** supports versioning, so you can easily roll back to a previous state of the data if necessary.

```python
# Show the version history of the Delta table
delta_table.history().show()
```

#### 6. **Deduplication Mechanism**

- Ensure that before inserting new transactions, the incoming batch is deduplicated based on the primary key columns (`CONTRACT_SOURCE_SYSTEM`, `CONTRACT_SOURCE_SYSTEM_ID`, and `NSE_ID`).

```python
# Remove duplicates based on primary keys
new_batch_df = new_batch_df.dropDuplicates(["CONTRACT_SOURCE_SYSTEM",
"CONTRACT_SOURCE_SYSTEM_ID", "NSE_ID"])
```

#### 7. **Validation and Integrity Checks**

- Before inserting or updating the data, perform **validations** to check for nulls and duplicates in primary key columns. This can be done using the validation functions proposed earlier to enforce primary key constraints.

```python
validate_primary_keys(new_batch_df)  # Ensure new batch has no nulls or duplicates
```

### Implementation Flow

1. **Load Historical Data**: Read the existing transactions from the storage (e.g., Parquet or Delta Lake).

2. **Load New Batch**: Load the new batch of transactions.

3. **Validate New Data**: Ensure the new batch is valid (e.g., check for nulls, primary key violations).

4. **Merge Data**: Use Delta Lake to merge new transactions with historical data.

5. **Write Back**: Write the updated transactions back to storage, partitioned by `BUSINESS_DATE` or another relevant field.

6. **Track Changes**: Maintain a version history of the data for auditing purposes.

### Benefits of This Approach

1. **Efficient Updates**: By using merge operations, only the relevant data is updated or inserted, reducing the overall processing time.

2. **Historical Data Integrity**: Historical transactions remain intact unless explicitly updated, ensuring the accuracy of past data.

3. **Scalability**: Partitioning and incremental processing allow the system to scale to handle large datasets.

4. **Versioning**: Data versioning ensures that historical states are preserved, allowing for auditing and rollback if necessary.

5. **Future-Proof**: The schema can evolve as needed, and new fields can be added without affecting the existing data.

### Conclusion

This proposal outlines a robust and scalable solution for handling new batches of data while preserving existing transactions. Using Delta Lake for merge operations, partitioning for performance, and validation mechanisms for primary keys ensures that data consistency and integrity are maintained.