Creating PySpark Data Pipeline - b4bb813e3

USER

Your job is expert cloud engineer. You have data json nested data in databricks and now you want to create pipeline using pyspark for reading, transformation and send it to s3 buckets. We are primaryly using aws service. Can you help me to complete this code
The input format is json nested file and output format is parquet in s3 bucket. Also ensure to include secutiry configuration, IAM permission to send it back to S3.

BOT

Certainly! As an expert cloud engineer, I can guide you through the process of creating a pipeline using PySpark to read nested JSON data from Databricks, transform it, and then write it to an S3 bucket. We'll also ensure that the necessary security configurations and IAM permissions are in place for secure data transfer.

Step 1: Setting Up the Environment

First, ensure you have the necessary dependencies and configurations in place. You'll need to have PySpark installed and configured to work with AWS services. If you're using Databricks, you can skip this step as PySpark is already integrated.

Step 2: Reading Nested JSON Data

To read the nested JSON data, you can use the `spark.read.json()` method. Assuming your JSON data is stored in a Databricks file system or an S3 bucket, you can read it as follows:

```python
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("JSON to Parquet").getOrCreate()

# Read the JSON data
json_data = spark.read.json("path/to/your/json/data")
```

Step 3: Transforming the Data

Next, you'll need to transform the data as required. This might involve flattening the nested structure, handling missing values, or performing any other necessary data manipulation. For example, if your JSON data has a nested structure like this:

```json
{
    "id": 1,
    "name": "John",
    "address": {
        "street": "123 Main St",
        "city": "Anytown",
        "state": "CA",
        "zip": "12345"
    }
}
```

You might want to flatten this structure. Here's an example of how you can do it:

```python
from pyspark.sql.functions import col

# Flatten the nested structure
flattened_data = json_data.select(
```

    col("id"),
    col("name"),
    col("address.street").alias("street"),
    col("address.city").alias("city"),
    col("address.state").alias("state"),
    col("address.zip").alias("zip")
)
```

### Step 4: Writing to Parquet in S3

To write the transformed data to an S3 bucket in Parquet format, you'll need to specify the output format and the S3 path. Make sure you have the necessary AWS credentials set up to access your S3 bucket. You can do this by setting environment variables or using the `spark.hadoop.fs.s3a` configuration.

Here's how you can write the data to S3:

```python
# Write the data to S3 in Parquet format
flattened_data.write.parquet("s3a://your-bucket-name/path/to/parquet/data")
```

Step 5: Security Configuration and IAM Permissions

To ensure secure data transfer, you'll need to configure your Spark environment to use your AWS credentials. You can do this by setting the `spark.hadoop.fs.s3a` configuration. Here's an example:

```python
spark._jsc.hadoopConfiguration().set("fs.s3a.access.key", "YOUR_ACCESS_KEY")
spark._jsc.hadoopConfiguration().set("fs.s3a.secret.key", "YOUR_SECRET_KEY")
spark._jsc.hadoopConfiguration().set("fs.s3a.endpoint", "s3.amazonaws.com")
```

Replace `YOUR_ACCESS_KEY` and `YOUR_SECRET_KEY` with your actual AWS access key and secret key. Make sure these credentials have the necessary permissions to read from the source and write to the target S3 bucket.

Step 6: IAM Permissions

Ensure that the IAM role or user you're using has the necessary permissions to read from the source (if it's an S3 bucket) and write to the target S3 bucket. The minimum permissions required are:

- `s3:GetObject` for reading from the source S3 bucket (if applicable)
- `s3:PutObject` for writing to the target S3 bucket

You can create a custom IAM policy or use an existing one that grants these permissions.

Complete Code

Here's the complete code snippet:

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# Create a SparkSession
spark = SparkSession.builder.appName("JSON to Parquet").getOrCreate()

# Read the JSON data
json_data = spark.read.json("path/to/your/json/data")

# Flatten the nested structure
flattened_data = json_data.select(
    col("id"),
    col("name"),
    col("address.street").alias("street"),
    col("address.city").alias("city"),
    col("address.state").alias("state"),
    col("address.zip").alias("zip")

)

# Configure AWS credentials
spark._jsc.hadoopConfiguration().set("fs.s3a.access.key", "YOUR_ACCESS_KEY")
spark._jsc.hadoopConfiguration().set("fs.s3a.secret.key", "YOUR_SECRET_KEY")
spark._jsc.hadoopConfiguration().set("fs.s3a.endpoint", "s3.amazonaws.com")

# Write the data to S3 in Parquet format
flattened_data.write.parquet("s3a://your-bucket-name/path/to/parquet/data")
```

Conclusion

This pipeline reads nested JSON data, transforms it, and writes it to an S3 bucket in Parquet format, ensuring secure data transfer by using AWS credentials and IAM permissions. Make sure to replace the placeholders (`YOUR_ACCESS_KEY`, `YOUR_SECRET_KEY`, `path/to/your/json/data`, and `your-bucket-name`) with your actual values.

If you need to handle more complex transformations or have specific requirements for error handling, feel free to ask, and I can guide you further.

USER

Your job is expert cloud engineer. You should do below tasks

You have data json nested data in databricks and now you want to create pipeline using pyspark for reading, transformation and send it to s3 buckets. We are primarily using aws service. Can you help me to complete this code

The input format is json nested file and You should do complex trandformation and output format is parquet in s3 bucket. Also ensure to include secutiry configuration, IAM permission to send it back to S3.
Write end to end etl pileline for it

BOT

Step-by-step Plan

- Step 1: Read Nested JSON Data - Use PySpark's `read.json` method to read the nested JSON data into a DataFrame, ensuring proper handling of nested structures.

- Step 2: Perform Complex Transformations - Apply the necessary transformations to the DataFrame, including any data cleaning, filtering, or aggregations required for the output format.

- Step 3: Write to S3 in Parquet Format - Use PySpark's `write.parquet` method to write the transformed DataFrame to an S3 bucket, ensuring the output format is correct and security configurations are in place.

- Step 4: Configure Security and IAM Permissions - Ensure the necessary IAM permissions are set up for the Databricks environment to access and write to the S3 bucket securely.

Certainly! As an expert cloud engineer, I'll guide you through the process of creating an end-to-end ETL pipeline using PySpark to read nested JSON data from Databricks, perform complex transformations, and write the data to an S3 bucket in Parquet format. We'll also ensure that the necessary security configurations and IAM permissions are in place for secure data transfer.

Step 1: Reading Nested JSON Data

First, let's read the nested JSON data into a DataFrame using PySpark's `read.json` method. This method can handle nested structures, but you need to ensure that the JSON data is properly formatted and that the schema is inferred correctly.
```

```
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("JSON to Parquet").getOrCreate()

# Read the JSON data
json_data = spark.read.json("path/to/your/json/data")
```

Step 2: Performing Complex Transformations

Next, we'll perform complex transformations on the DataFrame. This might involve flattening the nested structure, handling missing values, or performing any other necessary data manipulation. For example, if your JSON data has a nested structure like this:

```
{
    "id": 1,
    "name": "John",
    "address": {
        "street": "123 Main St",
        "city": "Anytown",
        "state": "CA",
        "zip": "12345"
    },
    "orders": [
        {
            "order_id": 1,
            "order_date": "2022-01-01",
            "total": 100.0
        },
        {
            "order_id": 2,
            "order_date": "2022-01-15",
            "total": 200.0
        }
    ]
}
```

You might want to flatten this structure, explode the `orders` array, and perform aggregations. Here's an example of how you can do it:

```
from pyspark.sql.functions import col, explode, sum

# Flatten the nested structure
flattened_data = json_data.select(
    col("id"),
    col("name"),
    col("address.street").alias("street"),
    col("address.city").alias("city"),
    col("address.state").alias("state"),
    col("address.zip").alias("zip"),
    explode(col("orders")).alias("orders")
)

# Perform aggregations
transformed_data = flattened_data.groupBy("id", "name", "street", "city", "state", "zip").agg(
    sum("orders.total").alias("total_orders")
)
```

Step 3: Writing to S3 in Parquet Format

To write the transformed data to an S3 bucket in Parquet format, you'll need to specify the output format and the S3 path. Make sure you have the necessary AWS credentials set up to access your S3 bucket. You can do this by setting environment variables or using the `spark.hadoop.fs.s3a` configuration.

Here's how you can write the data to S3:

```
# Write the data to S3 in Parquet format
transformed_data.write.parquet("s3a://your-bucket-name/path/to/parquet/data")
```

Step 4: Configuring Security and IAM Permissions

To ensure secure data transfer, you'll need to configure your Spark environment to use your AWS credentials. You can do this by setting the `spark.hadoop.fs.s3a` configuration. Here's an example:

```
spark._jsc.hadoopConfiguration().set("fs.s3a.access.key", "YOUR_ACCESS_KEY")
spark._jsc.hadoopConfiguration().set("fs.s3a.secret.key", "YOUR_SECRET_KEY")
spark._jsc.hadoopConfiguration().set("fs.s3a.endpoint", "s3.amazonaws.com")
```

Replace `YOUR_ACCESS_KEY` and `YOUR_SECRET_KEY` with your actual AWS access key and secret key. Make sure these credentials have the necessary permissions to read from the source and write to the target S3 bucket.

IAM Permissions

Ensure that the IAM role or user you're using has the necessary permissions to read from the source (if it's an S3 bucket) and write to the target S3 bucket. The minimum permissions required are:

- `s3:GetObject` for reading from the source S3 bucket (if applicable)
- `s3:PutObject` for writing to the target S3 bucket

You can create a custom IAM policy or use an existing one that grants these permissions.

Complete Code

Here's the complete code snippet:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, explode, sum

# Create a SparkSession
spark = SparkSession.builder.appName("JSON to Parquet").getOrCreate()

# Read the JSON data
json_data = spark.read.json("path/to/your/json/data")

# Flatten the nested structure
flattened_data = json_data.select(
    col("id"),
    col("name"),
    col("address.street").alias("street"),
    col("address.city").alias("city"),
    col("address.state").alias("state"),
    col("address.zip").alias("zip"),
```

```
    explode(col("orders")).alias("orders")
)

# Perform aggregations
transformed_data = flattened_data.groupBy("id", "name", "street", "city", "state", "zip").agg(
    sum("orders.total").alias("total_orders")
)

# Configure AWS credentials
spark._jsc.hadoopConfiguration().set("fs.s3a.access.key", "YOUR_ACCESS_KEY")
spark._jsc.hadoopConfiguration().set("fs.s3a.secret.key", "YOUR_SECRET_KEY")
spark._jsc.hadoopConfiguration().set("fs.s3a.endpoint", "s3.amazonaws.com")

# Write the data to S3 in Parquet format
transformed_data.write.parquet("s3a://your-bucket-name/path/to/parquet/data")
```

Conclusion

This pipeline reads nested JSON data, performs complex transformations, and writes it to an S3 bucket in Parquet format, ensuring secure data transfer by using AWS credentials and IAM permissions. Make sure to replace the placeholders (`YOUR_ACCESS_KEY`, `YOUR_SECRET_KEY`, `path/to/your/json/data`, and `your-bucket-name`) with your actual values.

If you need to handle more complex transformations or have specific requirements for error handling, feel free to ask, and I can guide you further.

USER

Your job is expert cloud engineer. You should do below tasks

You have data json nested data in databricks and now you want to create pipeline using pyspark for reading, transformation and send it to s3 buckets. We are primarily using aws service. Can you help me to complete this code

The input format is json nested file and You should do complex trandformation and output format is parquet in s3 bucket. Also ensure to include secutiry configuration, IAM permission to send it back to S3.
Write end to end etl pileline for it

BOT

I'll create a comprehensive end-to-end ETL pipeline with error handling, logging, and complex transformations. I'll break this down into multiple components for better organization and maintainability.

```python
import os
from datetime import datetime
from pyspark.sql import SparkSession
from pyspark.sql.functions import (
    col, explode, sum, current_timestamp,
    to_date, when, concat, lit, array_contains,
    from_json, struct
)
from pyspark.sql.types import (
    StructType, StructField, StringType,
    IntegerType, ArrayType, DoubleType
)
import logging

class ETLPipeline:
    def __init__(self):
        self.spark = self._create_spark_session()
        self.logger = self._setup_logger()
        self.current_date = datetime.now().strftime("%Y-%m-%d")

    def _setup_logger(self):
```

```
        """Configure logging"""
        logging.basicConfig(
            level=logging.INFO,
            format='%(asctime)s - %(levelname)s - %(message)s'
        )
        return logging.getLogger(__name__)

    def _create_spark_session(self):
        """Create Spark session with AWS configurations"""
        spark = (SparkSession.builder
                .appName("JSON_to_Parquet_Pipeline")
                .config("spark.sql.legacy.timeParserPolicy", "LEGACY")
                .config("spark.sql.sources.partitionOverwriteMode", "dynamic")
                .getOrCreate())

        # Configure AWS credentials
        aws_access_key = os.environ.get('AWS_ACCESS_KEY')
        aws_secret_key = os.environ.get('AWS_SECRET_KEY')

        spark._jsc.hadoopConfiguration().set("fs.s3a.access.key", aws_access_key)
        spark._jsc.hadoopConfiguration().set("fs.s3a.secret.key", aws_secret_key)
        spark._jsc.hadoopConfiguration().set("fs.s3a.endpoint", "s3.amazonaws.com")
        spark._jsc.hadoopConfiguration().set("fs.s3a.impl", "org.apache.hadoop.fs.s3a.S3AFileSystem")

        return spark

    def read_json_data(self, input_path):
        """Read nested JSON data with schema validation"""
        try:
            # Define schema for nested JSON
            schema = StructType([
                StructField("id", StringType(), True),
                StructField("timestamp", StringType(), True),
                StructField("customer", StructType([
                    StructField("customer_id", StringType(), True),
                    StructField("name", StringType(), True),
                    StructField("email", StringType(), True),
                    StructField("address", StructType([
                        StructField("street", StringType(), True),
                        StructField("city", StringType(), True),
                        StructField("state", StringType(), True),
                        StructField("zip", StringType(), True)
                    ]))
                ])),
                StructField("orders", ArrayType(StructType([
                    StructField("order_id", StringType(), True),
                    StructField("product_id", StringType(), True),
                    StructField("quantity", IntegerType(), True),
                    StructField("price", DoubleType(), True)
                ])))
            ])

            df = self.spark.read.schema(schema).json(input_path)
            self.logger.info(f"Successfully read JSON data from {input_path}")
            return df

        except Exception as e:
            self.logger.error(f"Error reading JSON data: {str(e)}")
            raise

    def transform_data(self, df):
        """Perform complex transformations on the data"""
        try:
            # Flatten nested structures
            flattened_df = df.select(
                col("id"),
                col("timestamp"),
                col("customer.customer_id"),
```

```
            col("customer.name"),
            col("customer.email"),
            col("customer.address.street"),
            col("customer.address.city"),
            col("customer.address.state"),
            col("customer.address.zip"),
            explode("orders").alias("order")
        )

        # Add derived columns
        transformed_df = flattened_df.select(
            col("id"),
            col("customer_id"),
            col("name"),
            col("email"),
            col("street"),
            col("city"),
            col("state"),
            col("zip"),
            col("order.order_id"),
            col("order.product_id"),
            col("order.quantity"),
            col("order.price"),
            (col("order.quantity") * col("order.price")).alias("total_amount"),
            to_date(col("timestamp")).alias("order_date")
        )

        # Add data quality checks
        quality_checked_df = transformed_df.withColumn(
            "data_quality_check",
            when(col("email").isNull(), "Invalid Email")
            .when(col("total_amount") <= 0, "Invalid Amount")
            .otherwise("Valid")
        )

        # Add audit columns
        final_df = quality_checked_df.withColumn(
            "processed_timestamp", current_timestamp()
        ).withColumn(
            "batch_id", lit(datetime.now().strftime("%Y%m%d_%H%M%S"))
        )

        self.logger.info("Data transformation completed successfully")
        return final_df

    except Exception as e:
        self.logger.error(f"Error during data transformation: {str(e)}")
        raise

def write_to_s3(self, df, output_path):
    """Write transformed data to S3 in Parquet format"""
    try:
        # Write to S3 with partitioning
        (df.write
         .mode("overwrite")
         .partitionBy("order_date")
         .parquet(output_path))

        self.logger.info(f"Successfully wrote data to {output_path}")

    except Exception as e:
        self.logger.error(f"Error writing to S3: {str(e)}")
        raise

def run_pipeline(self, input_path, output_path):
    """Execute the complete ETL pipeline"""
    try:
        self.logger.info("Starting ETL pipeline")
```

            # Read data
            raw_df = self.read_json_data(input_path)

            # Transform data
            transformed_df = self.transform_data(raw_df)

            # Write to S3
            self.write_to_s3(transformed_df, output_path)

            self.logger.info("ETL pipeline completed successfully")

        except Exception as e:
            self.logger.error(f"Pipeline failed: {str(e)}")
            raise
        finally:
            self.spark.stop()

def main():
    # Configure environment variables
    os.environ['AWS_ACCESS_KEY'] = 'YOUR_AWS_ACCESS_KEY'
    os.environ['AWS_SECRET_KEY'] = 'YOUR_AWS_SECRET_KEY'

    # Define input and output paths
    input_path = "dbfs:/path/to/input/json/data"
    output_path = "s3a://your-bucket/path/to/output/parquet"

    # Create and run pipeline
    pipeline = ETLPipeline()
    pipeline.run_pipeline(input_path, output_path)

if __name__ == "__main__":
    main()
```

### IAM Policy Required:
Here's the minimum IAM policy needed for the S3 operations:

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:GetObject",
                "s3:ListBucket",
                "s3:DeleteObject"
            ],
            "Resource": [
                "arn:aws:s3:::your-bucket/*",
                "arn:aws:s3:::your-bucket"
            ]
        }
    ]
}
```

Features of this ETL Pipeline:

1. **Error Handling**: Comprehensive try-except blocks with logging
2. **Data Quality Checks**: Validates email and amount fields
3. **Audit Columns**: Adds processing timestamp and batch ID
4. **Partitioning**: Data is partitioned by order_date for optimal query performance
5. **Schema Validation**: Enforces schema during JSON reading
6. **Complex Transformations**:
   - Flattens nested JSON structure
```

  - Calculates derived columns
  - Adds data quality flags
7. **Security**:
  - Uses environment variables for AWS credentials
  - Configures S3A filesystem properly
8. **Logging**: Detailed logging for monitoring and debugging

To run this pipeline in Databricks:

1. Create a new Databricks notebook
2. Copy the code into the notebook
3. Configure the AWS credentials in Databricks secrets or cluster configuration
4. Update the input and output paths
5. Run the notebook

Additional Considerations:

1. **Performance Optimization**:
  - Add caching for frequently used DataFrames
  - Adjust partition sizes based on data volume
  - Configure Spark parameters based on cluster size

2. **Monitoring**:
  - Add metrics collection
  - Implement alerting for failures
  - Monitor data quality metrics

3. **Security**:
  - Use Databricks secrets for storing credentials
  - Implement row-level security if needed
  - Add column-level encryption for sensitive data

4. **Maintenance**:
  - Add data retention policies
  - Implement backup strategies
  - Add cleanup procedures for temporary files

This pipeline provides a robust foundation for processing nested JSON data and can be extended based on specific requirements.

USER

You should ensure to read data from Databricks and do complex ETL processes and send it to destination S3.

BOT

I'll create a comprehensive ETL pipeline specifically designed to read data from Databricks, perform complex transformations, and write to S3. Here's the enhanced version:

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import (
    col, explode, explode_outer, concat, lit, when, current_timestamp,
    from_json, to_json, array_contains, size, collect_list, struct,
    sum, avg, count, max, min, dense_rank, row_number
)
from pyspark.sql.window import Window
from pyspark.sql.types import *
from delta.tables import DeltaTable
import logging
from datetime import datetime
import json

class DatabricksToS3Pipeline:
    def __init__(self):
        self.spark = self._create_spark_session()
```

```
        self.logger = self._setup_logger()

    def _setup_logger(self):
        logging.basicConfig(
            level=logging.INFO,
            format='%(asctime)s - %(levelname)s - %(message)s'
        )
        return logging.getLogger(__name__)

    def _create_spark_session(self):
        """Initialize Spark Session with necessary configurations"""
        spark = SparkSession.builder \
            .appName("Databricks_to_S3_ETL") \
            .config("spark.sql.legacy.timeParserPolicy", "LEGACY") \
            .config("spark.databricks.delta.preview.enabled", "true") \
            .config("spark.sql.sources.partitionOverwriteMode", "dynamic") \
            .getOrCreate()

        # Configure AWS credentials
        spark.conf.set("fs.s3a.access.key", dbutils.secrets.get(scope="aws-credentials", key="access-key"))
        spark.conf.set("fs.s3a.secret.key", dbutils.secrets.get(scope="aws-credentials", key="secret-key"))
        spark.conf.set("fs.s3a.endpoint", "s3.amazonaws.com")

        return spark

    def read_from_databricks(self, database_name, table_name):
        """Read data from Databricks Delta table"""
        try:
            self.logger.info(f"Reading data from {database_name}.{table_name}")

            # Read from Delta table
            df = self.spark.table(f"{database_name}.{table_name}")

            # Cache the dataframe for better performance
            df.cache()

            self.logger.info(f"Successfully read {df.count()} records")
            return df

        except Exception as e:
            self.logger.error(f"Error reading from Databricks: {str(e)}")
            raise

    def complex_transformations(self, df):
        """Perform complex transformations on the data"""
        try:
            self.logger.info("Starting complex transformations")

            # 1. Flatten nested structures
            flattened_df = df.select(
                col("id"),
                col("timestamp"),
                col("customer.customer_id"),
                col("customer.profile.*"),
                explode_outer("orders").alias("order")
            )

            # 2. Create window specifications for analytics
            window_spec = Window.partitionBy("customer_id").orderBy("timestamp")

            # 3. Apply complex transformations
            transformed_df = flattened_df \
                .withColumn("row_number", row_number().over(window_spec)) \
                .withColumn("order_rank", dense_rank().over(window_spec)) \
                .withColumn("total_amount", col("order.quantity") * col("order.price")) \
                .withColumn("customer_segment",
                    when(col("total_amount") > 1000, "Premium")
                    .when(col("total_amount") > 500, "Gold")
```

```
                .otherwise("Regular")
            )

        # 4. Perform aggregations
        agg_df = transformed_df.groupBy("customer_id", "customer_segment") \
            .agg(
                count("order.order_id").alias("total_orders"),
                sum("total_amount").alias("lifetime_value"),
                collect_list(struct(
                    "order.order_id",
                    "total_amount",
                    "timestamp"
                )).alias("order_history")
            )

        # 5. Add data quality checks
        quality_df = agg_df \
            .withColumn("data_quality_flag",
                when(col("lifetime_value") < 0, "Invalid Amount")
                .when(col("total_orders") == 0, "No Orders")
                .otherwise("Valid")
            )

        # 6. Add audit columns
        final_df = quality_df \
            .withColumn("processed_timestamp", current_timestamp()) \
            .withColumn("batch_id", lit(datetime.now().strftime("%Y%m%d_%H%M%S"))) \
            .withColumn("source_system", lit("Databricks")) \
            .withColumn("process_date", current_timestamp())

        self.logger.info("Complex transformations completed successfully")
        return final_df

    except Exception as e:
        self.logger.error(f"Error during transformations: {str(e)}")
        raise

def write_to_s3(self, df, bucket_name, prefix):
    """Write transformed data to S3 in Parquet format"""
    try:
        output_path = f"s3a://{bucket_name}/{prefix}"

        # Write with partitioning and optimization
        (df.write
            .mode("overwrite")
            .partitionBy("customer_segment", "process_date")
            .option("maxRecordsPerFile", 1000000)
            .option("compression", "snappy")
            .parquet(output_path))

        self.logger.info(f"Successfully wrote data to {output_path}")

    except Exception as e:
        self.logger.error(f"Error writing to S3: {str(e)}")
        raise

def run_pipeline(self, config):
    """Execute the complete ETL pipeline"""
    try:
        self.logger.info("Starting ETL pipeline")

        # Read configuration
        database_name = config['source']['database']
        table_name = config['source']['table']
        bucket_name = config['destination']['bucket']
        prefix = config['destination']['prefix']

        # Execute pipeline steps
```

```
            source_df = self.read_from_databricks(database_name, table_name)
            transformed_df = self.complex_transformations(source_df)
            self.write_to_s3(transformed_df, bucket_name, prefix)

            # Clean up
            source_df.unpersist()

            self.logger.info("ETL pipeline completed successfully")

        except Exception as e:
            self.logger.error(f"Pipeline failed: {str(e)}")
            raise
        finally:
            self.spark.stop()

def main():
    # Pipeline configuration
    config = {
        "source": {
            "database": "your_database",
            "table": "your_table"
        },
        "destination": {
            "bucket": "your-s3-bucket",
            "prefix": "processed-data/year=2024/month=01"
        }
    }

    # Create and run pipeline
    pipeline = DatabricksToS3Pipeline()
    pipeline.run_pipeline(config)

if __name__ == "__main__":
    main()
```

Databricks Notebook Implementation:

```python
# Databricks notebook source
# MAGIC %md
# MAGIC ## Databricks to S3 ETL Pipeline
# MAGIC This notebook implements an ETL pipeline to process data from Databricks and write to S3.

# COMMAND ----------
# MAGIC %md
# MAGIC ### Setup and Configuration

# COMMAND ----------

# DBTITLE 1,Import Dependencies and Configure Environment
# Import the pipeline class
from databricks_to_s3_pipeline import DatabricksToS3Pipeline

# Configure environment
pipeline_config = {
    "source": {
        "database": dbutils.widgets.get("database"),
        "table": dbutils.widgets.get("table")
    },
    "destination": {
        "bucket": dbutils.widgets.get("s3_bucket"),
        "prefix": dbutils.widgets.get("s3_prefix")
    }
}

# COMMAND ----------
```

# DBTITLE 1,Execute Pipeline
pipeline = DatabricksToS3Pipeline()
pipeline.run_pipeline(pipeline_config)
```

### Required IAM Policy:

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:GetObject",
                "s3:ListBucket",
                "s3:DeleteObject"
            ],
            "Resource": [
                "arn:aws:s3:::your-bucket/*",
                "arn:aws:s3:::your-bucket"
            ]
        }
    ]
}
```

Key Features:

1. **Databricks Integration**:
   - Native Delta table support
   - Databricks secrets integration
   - Optimized for Databricks runtime

2. **Complex Transformations**:
   - Nested structure flattening
   - Window functions for advanced analytics
   - Customer segmentation
   - Order history aggregation
   - Data quality checks

3. **Performance Optimizations**:
   - Dataframe caching
   - Partition optimization
   - File size management
   - Snappy compression

4. **Security**:
   - Databricks secrets for AWS credentials
   - Secure S3 communication
   - Audit trailing

5. **Monitoring and Logging**:
   - Comprehensive logging
   - Error handling
   - Process tracking

To Use This Pipeline:

1. **Setup in Databricks**:
   ```python
   dbutils.widgets.text("database", "default", "Source Database")
   dbutils.widgets.text("table", "source_table", "Source Table")
   dbutils.widgets.text("s3_bucket", "your-bucket", "S3 Bucket")
   dbutils.widgets.text("s3_prefix", "processed-data", "S3 Prefix")
   ```
   ```

2. **Configure Secrets**:
```python
# In Databricks Secrets UI
scope = "aws-credentials"
key = "access-key"
value = "YOUR_AWS_ACCESS_KEY"

key = "secret-key"
value = "YOUR_AWS_SECRET_KEY"
```

3. **Run the Pipeline**:
   - Upload the code as a Databricks notebook
   - Configure the widgets
   - Run the notebook

Best Practices Implemented:

1. **Error Handling**:
   - Comprehensive try-except blocks
   - Detailed error logging
   - Graceful failure handling

2. **Performance**:
   - Optimal partition strategy
   - Caching when beneficial
   - Resource cleanup

3. **Monitoring**:
   - Detailed logging
   - Process tracking
   - Data quality checks

4. **Maintainability**:
   - Modular code structure
   - Configuration-driven
   - Well-documented

This pipeline provides a robust foundation for processing data from Databricks to S3 and can be extended based on specific requirements.