1. The **getIncorrectColsRecords** function reads the input file as text data and splits each record by the pipe character (**|**). It then adds a new column to the DataFrame that represents the number of columns in each record. Finally, it filters out any records that do not have 19 columns (which is the expected number of columns).
2. The **getValidRecords** function takes the original DataFrame and the DataFrame containing the invalid records as input. It then performs an inner join on the two DataFrames to get only the valid records.
3. The **getInvalidRecords** function takes the original DataFrame and the DataFrame containing the invalid records as input. It then performs a left anti-join on the two DataFrames to get only the invalid records.
4. The **main** function calls the **getIncorrectColsRecords** function to get the DataFrame containing the invalid records. It then calls the **getValidRecords** and **getInvalidRecords** functions to get the DataFrames containing the valid and invalid records, respectively.
5. The main function then logs the number of total, valid, and invalid records. It then calls the **writeOutput** function to write the valid and invalid records to two separate output directories.
6. Finally, the main function logs the completion of writing the valid and invalid records to their respective output directories.

Vsolfilevalidations

The code defines an object called **VsolFileValidation** that contains a **main** function and several helper functions. The purpose of this object is to validate a batch file containing verification data, and the code uses the Apache Spark framework to do so.

The **main** function takes in a command-line argument array and attempts to parse the arguments using the **scopt** library. If the parsing is successful, the function extracts the arguments from the **commandLineArgs** object that **scopt** generates and initializes various variables. It then initializes a Spark context and sets some Hadoop configuration variables.

The function then generates a YAML file based on the input file and validates the headers of the input file using the YAML file. If the headers are valid, the function proceeds to validate the rest of the input file. If the validation succeeds, the function performs various checks on the input file, such as ensuring that the input file exists and has a valid name. Finally, the function performs file validation on the input file, using the **FileValidation** class.

The **validateFileName** function takes in an input file name and a **DataFrame** representing the configuration for the batch process. It extracts the file name from the input file path and checks if it has a valid format. If the format is not valid, the function logs an error and throws an exception.

Overall, the code performs several checks and validations on a batch file containing verification data, using Apache Spark and the **scopt** library.

enkins is an open-source automation server that helps automate software development processes such as building, testing, and deploying software applications. It can be used to manage and monitor software delivery pipelines, integrating with various tools and technologies used in the software development process.

Veracode is a cloud-based application security platform that helps developers and organizations secure their software applications. It provides a suite of security testing tools that can be used to detect and fix vulnerabilities in applications, ensuring that they are secure and compliant with industry standards and regulations.

SonarLint is an open-source static code analysis tool that helps developers identify and fix coding issues early in the development process. It integrates with various IDEs and editors and provides real-time feedback on coding standards, code quality, and potential security vulnerabilities.

Docker is a platform for building, shipping, and running distributed applications in containers. It provides a way to package and distribute software applications with all their dependencies and configurations, making it easy to deploy them across different environments and platforms. Docker allows developers to isolate their applications from the underlying infrastructure, providing a consistent and reliable way to run applications in different environments.

Top of Form

adppostprocessingThis is a Scala code snippet that defines an object **AdpPostProcessing**. The object has a **main** method that expects to receive an array of strings **args**, which will be parsed by a **scopt.OptionParser** instance. The **main** method will read some input CSV files, join them, generate two output CSV files, and rename them.

The **joinDataFrames** method flattens the input DataFrames, then joins them based on the **experianTransactionId** column in the first DataFrame, and the **transactionID** column in the second DataFrame. The resulting DataFrame is then returned.

The **generateAdpPinningDF** method takes the joined DataFrame as input and generates a new DataFrame with some additional columns. These columns are named and formatted according to some specific requirements, then the resulting DataFrame is returned.

There is also a **generatePostProcessingDF** method, which takes the joined DataFrame as input and generates another DataFrame. The columns in this DataFrame are again named and formatted according to specific requirements, then the resulting DataFrame is returned.

Finally, there are some utility methods, such as **readFile**, **writeOutput**, and **renamePartFileInS3**, which are used to read and write files and to rename the output files.

Overall, this code appears to be a batch processing application that performs some data manipulation on input CSV files, generates some output CSV files, and renames them. It is built on top of Apache Spark, which is a distributed computing framework, and it reads and writes data to Amazon S3, which is a cloud-based storage service.

This is a Scala code that seems to be performing some data processing tasks. The main function is **generatePostProcessingDF**, which takes a **DataFrame** as input and returns a processed **DataFrame** as output.

The code consists of several functions, each performing a specific task. The **explodeEmploymentHistory** function explodes the **employmentHistory** column, which is an array of structs, into separate columns. The **processPaymentHistory** function transforms, sorts, and explodes the **paymentHistory** column, which is an array of structs, into separate columns. It also filters the data based on a date value and generates some new columns such as **currentYear** and **previousYear1**.

The **processRemunerationSummary** function is not included in the code provided, but it is likely similar to the **processPaymentHistory** function, as it also involves exploding and imploding arrays.

The **ADP\_POST\_PROCESSING\_MAP** and **ADP\_PLACEHOLDER\_ARRAY** are two arrays that are used to rename and add null placeholders to columns, respectively. The final step is to join the **DataFrame** with the **employmentCount** and **reportGenerationDate** columns, rename columns, and select the desired columns.

Overall, this code seems to be a part of a larger data processing pipeline that takes raw data and transforms it into a more structured and useful format.

Postprocessing

This is a Scala code for a batch processing job that performs pre-processing of data. The code reads input data from a CSV file, selects required columns, renames them, and writes the resulting data to a compressed CSV file. It also generates a JSON metadata file and writes it to a separate file. The input and output file paths are passed as command-line arguments.

The code uses the Apache Spark framework for distributed processing of data. The Spark session is initialized and Hadoop configurations are set for reading and writing data to Amazon S3.

The **generateRequiredDF()** function selects the required columns from the input DataFrame and renames them. The **generateMetadataDF()** function creates a DataFrame from a hardcoded JSON string. The **writeCompressedOutput()** and **writeJsonOutput()** functions write the DataFrames to compressed CSV and JSON files, respectively.

Finally, the **renamePartFileInS3()** function is used to rename the files stored in S3 bucket, and a log message is printed at the end of the job indicating that the pre-processing is completed.

**df.coalesce** is a DataFrame operation in Apache Spark that is used to reduce the number of partitions in a DataFrame. It does not shuffle data across the network like **repartition**, instead it merges adjacent partitions to create new partitions.

The **coalesce** method can be useful when you want to reduce the number of partitions in a DataFrame, for example, to make the data more manageable or to reduce overhead costs. In the code you provided, **df.coalesce(1)** is used to merge all the partitions into a single partition. This is often done when writing the data to a single file or when reducing the data size to fit in a single machine's memory

Ees

This code seems to be a Scala program that reads data from a CSV file, performs some data processing, and writes the processed data to another CSV file.

It starts by defining a **Logger** object and initializing it with an informational message. Then, it defines a **scopt.OptionParser** object, which is a command-line argument parser library for Scala. The parser is used to parse the command-line arguments passed to the program.

The **main** function uses the parser to parse the command-line arguments and extracts the relevant arguments. It then initializes a SparkSession, sets some Hadoop configuration properties, reads two CSV files (**inputEES** and **fileValidationOutput**) into DataFrames using the **readFile** function, joins the DataFrames using the **joinDataFrames** function, and applies some transformations to the resulting DataFrame using the **generateProcessingDF** function. Finally, it writes the processed DataFrame to a CSV file using the **writeOutput** function.

The **generateProcessingDF** function performs several data transformations on the input DataFrame. It first flattens the input DataFrame, filters some columns based on certain conditions, and adds some new columns. It then pivots the DataFrame using the **pivotRenumerationYear** function, performs some additional transformations, and selects a subset of columns based on the **EES\_PROCESSING\_DF** constant.

The provided code is a Scala program for ADP (Automatic Data Processing) pre-processing. The program performs the following steps:

1. It imports necessary libraries and defines constants.
2. It initializes a logger and logs the start of the ADP pre-processing.
3. It defines an **OptionParser** to parse command-line arguments.
4. The **main** method is the entry point of the program.
5. It parses the command-line arguments using the **OptionParser** and extracts the required parameters.
6. It initializes Spark and sets Hadoop configurations for S3 encryption and endpoint.
7. It reads the input file from the specified path using the **readFile** function.
8. It generates a new DataFrame by selecting the required columns and renaming them.
9. It writes the required DataFrame to an output file in compressed format (CSV with Gzip compression) using the **writeCompressedOutput** function.
10. It renames the part files generated by Spark to the final output file name using the **renamePartFileInS3** function.
11. It generates a metadata DataFrame using a JSON string and writes it to a separate JSON file using the **writeJsonOutput** function.
12. It renames the part files generated for the metadata DataFrame using the **renamePartFileInS3** function.
13. If any exceptions occur during the process, a **TechnicalException** is thrown.

The program leverages Spark for distributed processing and handles file compression and renaming. It performs transformations on the input DataFrame to generate the required output files.

Please note that this code snippet is meant to be part of a larger project and might reference functions or variables defined in other parts of the project.

It seems that you have provided a Scala code snippet for an ADP Post Processing application. This application generates a ".pinning.adp" file and a post-processing file that can be used by the Merge process.

The code defines a Scala object called **AdpPostProcessing** with a **main** function that serves as the entry point for the application. The code uses the **scopt** library for command-line argument parsing. The parsed arguments are then used to perform various operations on Spark DataFrames.

Here's an overview of the major components and operations in the code:

1. Import statements: The code imports various classes, constants, and utility functions required for the application.
2. Argument Parser: The **scopt.OptionParser** is used to define and parse command-line arguments for the application. The parsed arguments are then used to set various variables.
3. **main** function: This is the entry point for the application. It parses the command-line arguments using the **parser.parse** method and performs operations based on the parsed arguments.
4. Spark Initialization: The SparkSession is initialized using the **initSpark** utility function.
5. Hadoop Configurations: Hadoop configurations are set to specify encryption algorithm and S3 endpoint.
6. Reading Input: The code reads input data from CSV files using the **readFile** utility function.
7. Joining DataFrames: The **joinDataFrames** function is called to join two DataFrames (**fromAdp** and **fromFileValidation**) based on a common column.
8. Generating ".pinning.adp" File: The **generateAdpPinningDF** function is called to generate a new DataFrame (**AdpPinningDF**) with specific columns and transformations. The resulting DataFrame is then written to a CSV file using the **writeOutput** utility function.
9. Renaming Part Files: The **renamePartFileInS3** function is called to rename the generated output file in the S3 bucket.
10. Generating Post Processing File: The **generatePostProcessingDF** function is called to generate a new DataFrame (**postProcessingDF**) with specific columns and transformations. The resulting DataFrame is then written to a CSV file using the **writeOutput** utility function.

The code also includes helper functions like **flattenFirstLevelData**, **flattenResponseData**, **defaultToEmptyIfNull**, **explodeEmploymentHistory**, **processPaymentHistory**, and **processRemunerationSummary** that perform various data transformations and manipulations on DataFrames.

Please note that the code snippet you provided is incomplete and might contain missing dependencies and utility functions. Make sure you have all the required dependencies and utility functions to successfully run the application.

**sftpfile\_pull\_operator.py** class, I am unable to provide a detailed description of its functionality. However, I can provide a general overview of what a typical implementation of an SFTP file pull operator might include:

1. Import Dependencies: The class would likely import the necessary dependencies, such as the **BaseOperator** class from Apache Airflow and the **pysftp** library for SFTP operations.
2. Class Definition: The class would be defined, typically inheriting from the **BaseOperator** class or a related subclass. This allows the operator to be used as a task within an Airflow DAG.
3. Constructor: The class would have a constructor method that initializes the necessary attributes and parameters. This may include the SFTP connection details (hostname, port, username, password, or private key path), remote path, local path, and any other required parameters.
4. Execute Method: The class would implement an **execute** method, which contains the logic for connecting to the SFTP server, retrieving the specified file(s), and saving them to the local directory. This method would be executed when the operator is triggered within an Airflow workflow.
5. Error Handling: The class might include error handling mechanisms to handle exceptions that may occur during the SFTP operation, such as authentication failures, connection errors, or file not found errors.
6. Logging and Reporting: The class could include logging statements or reporting mechanisms to provide information about the execution and status of the SFTP file pull operation. This helps with monitoring and troubleshooting.

It's important to note that the actual implementation may vary depending on the specific requirements and design choices of the operator. Without the code or more specific information about the **sftpfile\_pull\_operator.py** class, it's not possible to provide a more detailed overview.

The provided code represents a custom **SFTPFilePushOperator** class, which is a subclass of **BaseOperator** in Apache Airflow. This operator is responsible for pushing files from an S3 bucket to an SFTP server.

Here's a breakdown of the code:

1. Imports: The necessary libraries are imported, including **paramiko** for SFTP operations, **BaseOperator** from Airflow, an **SSARAwsHook** for AWS interactions, and an **ExceptionUtility** for handling exceptions.
2. Class Definition: The **SFTPFilePushOperator** class is defined as a subclass of **BaseOperator**.
3. Constructor: The class constructor (**\_\_init\_\_**) initializes the required parameters: **team**, **env**, **bucket**, **object\_path**, **metadata\_path**, and **input\_file\_name**. These parameters represent the relevant information needed to push the files to the SFTP server.
4. Execute Method: The **execute** method is implemented to perform the SFTP file push operation. Here's a breakdown of the steps within the method:

a. Retrieve AWS credentials: The code initializes an **SSARAwsHook** with the provided **team** and **env** parameters to retrieve the AWS credentials required for accessing the S3 bucket.

b. Create an SFTP connection: The code creates a connection to the SFTP server using **paramiko.Transport** and the provided host (**aft-sftp-2e08a8483410f6de.elb.us-east-1.amazonaws.com**) and port (**22**).

c. Download and push files: The code downloads the file from the specified S3 bucket and object path using the **s3.download\_fileobj** method. It then writes the file to the SFTP server using **sftp.open** and the provided file paths.

d. Close the connections: After the file push operation is complete, the SFTP connection and transport are closed using **sftp.close()** and **transport.close()**.

1. Error Handling: Any exceptions that occur during the execution are caught and handled using the **ExceptionUtility** class. The **exception\_handler** method is called, passing the error and the operator's log for error logging and handling.

Please note that this code assumes certain parameter values such as the SFTP server's hostname, username, and password. You may need to modify these values to match your specific setup.

The provided code represents a custom **S3FilePushOperator** class, which is a subclass of **BaseOperator** in Apache Airflow. This operator is responsible for copying objects from one S3 bucket to another.

Here's a breakdown of the code:

1. Imports: The necessary libraries are imported, including **SSARS3Hook** for S3 operations, **BaseOperator** from Airflow, and **CustomConfig** for retrieving environment and team values.
2. Class Definition: The **S3FilePushOperator** class is defined as a subclass of **BaseOperator**.
3. Constructor: The class constructor (**\_\_init\_\_**) initializes the required and optional parameters for the operator. The required parameters are **source\_bucket\_key**, **dest\_bucket\_key**, and **server\_side\_encryption**. The optional parameters are **source\_bucket\_name**, **dest\_bucket\_name**, **sse\_kms\_keyid**, and **source\_version\_id**. These parameters represent the relevant information needed to copy objects from one S3 bucket to another.
4. Execute Method: The **execute** method is implemented to perform the S3 object copy operation. Here's a breakdown of the steps within the method:

a. Retrieve environment and team values: The code retrieves the environment and team values from **CustomConfig**.

b. Initialize the S3 hook: The code initializes an **SSARS3Hook** with the team and environment values.

c. Copy the S3 object: The code calls the **copy\_object** method on the S3 hook, passing the required and optional parameters. This method performs the actual copy operation from the source bucket and key to the destination bucket and key.

1. Response: The method assigns the response of the **copy\_object** operation to the **response** variable. You can use this variable to further process the response if needed.

Please note that this code assumes the presence of a custom **SSARS3Hook** class and a **CustomConfig** class. You may need to define these classes or modify the import statements to match your specific implementation.