#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

"""

Hive Table Validator

This script provides functionality to validate Hive tables using PySpark.

It includes methods for schema validation, data validation, count validation,

data quality checks, and detailed reporting of validation results.

"""

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, count, lit, when, isnan, isnull, length, trim

from pyspark.sql.types import StructType, NumericType, StringType, DateType, TimestampType

import logging

from typing import List, Dict, Tuple, Optional, Union

import json

import os

from datetime import datetime

class HiveTableValidator:

    """

    A class for validating Hive tables using PySpark.

    This class provides methods to validate schema, data quality, row counts,

    and perform comparisons between Hive tables.

    """

    def \_\_init\_\_(self, spark: SparkSession, table\_name: str,

                 reference\_table: str = None, key\_columns: List[str] = None,

                 exclude\_columns: List[str] = None, tolerance: Dict = None,

                 output\_path: str = "./validation\_results"):

        """

        Initialize the HiveTableValidator class.

        Args:

            spark: SparkSession object with Hive support enabled

            table\_name: Name of the Hive table to validate

            reference\_table: Optional reference table name for comparison

            key\_columns: List of columns to use as keys for row-by-row comparison

            exclude\_columns: List of columns to exclude from validation

            tolerance: Dictionary mapping column names to tolerance values for numeric comparisons

            output\_path: Path to save validation results

        """

        self.spark = spark

        self.table\_name = table\_name

        self.reference\_table = reference\_table

        self.key\_columns = key\_columns or []

        self.exclude\_columns = exclude\_columns or []

        self.tolerance = tolerance or {}

        self.output\_path = output\_path

        # Set up logging

        self.logger = logging.getLogger(\_\_name\_\_)

        self.logger.setLevel(logging.INFO)

        if not self.logger.handlers:

            handler = logging.StreamHandler()

            formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

            handler.setFormatter(formatter)

            self.logger.addHandler(handler)

        # Initialize result dictionary

        self.results = {

            "validation\_time": datetime.now().strftime("%Y-%m-%d %H:%M:%S"),

            "table\_name": table\_name,

            "reference\_table": reference\_table,

            "validation\_summary": {

                "table\_exists": None,

                "schema\_valid": None,

                "row\_count": None,

                "data\_quality": None,

                "reference\_comparison": None

            },

            "details": {

                "schema": {},

                "row\_count": {},

                "data\_quality": {},

                "reference\_comparison": {}

            }

        }

    def load\_table(self, table\_name: str) -> Optional["pyspark.sql.DataFrame"]:

        """

        Load a Hive table as a DataFrame.

        Args:

            table\_name: Name of the Hive table to load

        Returns:

            DataFrame or None if table doesn't exist

        """

        try:

            self.logger.info(f"Loading table: {table\_name}")

            # Check if table exists

            tables = self.spark.sql(f"SHOW TABLES LIKE '{table\_name}'").count()

            if tables == 0:

                self.logger.error(f"Table {table\_name} does not exist")

                return None

            # Load the table

            df = self.spark.table(table\_name)

            return df

        except Exception as e:

            self.logger.error(f"Error loading table {table\_name}: {str(e)}")

            return None

    def validate\_table\_exists(self) -> bool:

        """

        Validate that the table exists in Hive.

        Returns:

            Boolean indicating if the table exists

        """

        self.logger.info(f"Validating table existence: {self.table\_name}")

        try:

            tables = self.spark.sql(f"SHOW TABLES LIKE '{self.table\_name}'").count()

            table\_exists = tables > 0

            self.results["validation\_summary"]["table\_exists"] = table\_exists

            if table\_exists:

                self.logger.info(f"Table {self.table\_name} exists")

            else:

                self.logger.error(f"Table {self.table\_name} does not exist")

            return table\_exists

        except Exception as e:

            self.logger.error(f"Error validating table existence: {str(e)}")

            self.results["validation\_summary"]["table\_exists"] = False

            self.results["details"]["error"] = f"Failed to validate table existence: {str(e)}"

            return False

    def validate\_schema(self, expected\_schema: Optional[Dict] = None) -> bool:

        """

        Validate the schema of the table against expected schema if provided.

        Args:

            expected\_schema: Optional dictionary of expected column names and types

        Returns:

            Boolean indicating if schema is valid

        """

        self.logger.info(f"Validating schema for table: {self.table\_name}")

        # Load the table

        df = self.load\_table(self.table\_name)

        if df is None:

            self.results["validation\_summary"]["schema\_valid"] = False

            self.results["details"]["schema"]["error"] = "Failed to load table"

            return False

        # Get actual schema

        actual\_schema = {field.name: str(field.dataType) for field in df.schema.fields}

        self.results["details"]["schema"]["actual\_schema"] = actual\_schema

        # If no expected schema provided, just record the actual schema

        if expected\_schema is None:

            self.results["validation\_summary"]["schema\_valid"] = True

            self.logger.info(f"Schema recorded (no validation performed)")

            return True

        # Compare with expected schema

        missing\_columns = []

        type\_mismatches = []

        for col\_name, expected\_type in expected\_schema.items():

            if col\_name not in actual\_schema:

                missing\_columns.append(col\_name)

            elif actual\_schema[col\_name] != expected\_type:

                type\_mismatches.append({

                    "column": col\_name,

                    "expected\_type": expected\_type,

                    "actual\_type": actual\_schema[col\_name]

                })

        # Check for extra columns

        extra\_columns = [col for col in actual\_schema if col not in expected\_schema]

        # Record schema validation results

        schema\_valid = len(missing\_columns) == 0 and len(type\_mismatches) == 0

        self.results["validation\_summary"]["schema\_valid"] = schema\_valid

        self.results["details"]["schema"]["validation"] = {

            "missing\_columns": missing\_columns,

            "extra\_columns": extra\_columns,

            "type\_mismatches": type\_mismatches

        }

        if schema\_valid:

            self.logger.info("Schema validation passed")

        else:

            self.logger.warning("Schema validation failed")

            if missing\_columns:

                self.logger.warning(f"Missing columns: {missing\_columns}")

            if type\_mismatches:

                self.logger.warning(f"Type mismatches: {type\_mismatches}")

        return schema\_valid

    def validate\_row\_count(self, expected\_count: Optional[int] = None,

                          min\_count: Optional[int] = None,

                          max\_count: Optional[int] = None) -> bool:

        """

        Validate the row count of the table.

        Args:

            expected\_count: Optional exact count to validate against

            min\_count: Optional minimum count to validate against

            max\_count: Optional maximum count to validate against

        Returns:

            Boolean indicating if row count is valid

        """

        self.logger.info(f"Validating row count for table: {self.table\_name}")

        # Load the table

        df = self.load\_table(self.table\_name)

        if df is None:

            self.results["validation\_summary"]["row\_count"] = False

            self.results["details"]["row\_count"]["error"] = "Failed to load table"

            return False

        # Get actual count

        actual\_count = df.count()

        self.results["details"]["row\_count"]["actual\_count"] = actual\_count

        # Validate against criteria

        count\_valid = True

        validation\_details = {}

        if expected\_count is not None:

            count\_matches = actual\_count == expected\_count

            count\_valid = count\_valid and count\_matches

            validation\_details["expected\_count"] = expected\_count

            validation\_details["count\_matches"] = count\_matches

        if min\_count is not None:

            above\_min = actual\_count >= min\_count

            count\_valid = count\_valid and above\_min

            validation\_details["min\_count"] = min\_count

            validation\_details["above\_min"] = above\_min

        if max\_count is not None:

            below\_max = actual\_count <= max\_count

            count\_valid = count\_valid and below\_max

            validation\_details["max\_count"] = max\_count

            validation\_details["below\_max"] = below\_max

        # Record row count validation results

        self.results["validation\_summary"]["row\_count"] = count\_valid

        self.results["details"]["row\_count"]["validation"] = validation\_details

        if count\_valid:

            self.logger.info(f"Row count validation passed: {actual\_count} rows")

        else:

            self.logger.warning(f"Row count validation failed: {actual\_count} rows")

        return count\_valid

    def validate\_data\_quality(self, rules: Dict[str, Dict]) -> bool:

        """

        Validate data quality using specified rules.

        Args:

            rules: Dictionary mapping column names to validation rules

                  Example: {

                      "column1": {

                          "not\_null": True,

                          "min\_value": 0,

                          "max\_value": 100

                      },

                      "column2": {

                          "not\_null": True,

                          "min\_length": 5,

                          "regex": "^[A-Z].\*"

                      }

                  }

        Returns:

            Boolean indicating if data quality is valid

        """

        self.logger.info(f"Validating data quality for table: {self.table\_name}")

        # Load the table

        df = self.load\_table(self.table\_name)

        if df is None:

            self.results["validation\_summary"]["data\_quality"] = False

            self.results["details"]["data\_quality"]["error"] = "Failed to load table"

            return False

        # Initialize validation results

        all\_rules\_passed = True

        column\_results = {}

        # Process each column's rules

        for column, column\_rules in rules.items():

            if column not in df.columns:

                self.logger.warning(f"Column {column} not found in table, skipping validation")

                column\_results[column] = {"error": "Column not found in table"}

                all\_rules\_passed = False

                continue

            column\_result = {"rules\_checked": {}}

            column\_passed = True

            # Get column data type

            col\_type = [f.dataType for f in df.schema.fields if f.name == column][0]

            # Apply each rule for the column

            for rule, value in column\_rules.items():

                if rule == "not\_null":

                    if value:

                        null\_count = df.filter(col(column).isNull() | isnan(column)).count()

                        rule\_passed = null\_count == 0

                        column\_result["rules\_checked"]["not\_null"] = {

                            "passed": rule\_passed,

                            "null\_count": null\_count

                        }

                        column\_passed = column\_passed and rule\_passed

                elif rule == "unique":

                    if value:

                        distinct\_count = df.select(column).distinct().count()

                        total\_count = df.count()

                        rule\_passed = distinct\_count == total\_count

                        column\_result["rules\_checked"]["unique"] = {

                            "passed": rule\_passed,

                            "distinct\_count": distinct\_count,

                            "total\_count": total\_count

                        }

                        column\_passed = column\_passed and rule\_passed

                elif rule in ["min\_value", "max\_value"] and isinstance(col\_type, NumericType):

                    if rule == "min\_value":

                        below\_min\_count = df.filter(col(column) < value).count()

                        rule\_passed = below\_min\_count == 0

                        column\_result["rules\_checked"]["min\_value"] = {

                            "passed": rule\_passed,

                            "min\_value": value,

                            "below\_min\_count": below\_min\_count

                        }

                        column\_passed = column\_passed and rule\_passed

                    else:  # max\_value

                        above\_max\_count = df.filter(col(column) > value).count()

                        rule\_passed = above\_max\_count == 0

                        column\_result["rules\_checked"]["max\_value"] = {

                            "passed": rule\_passed,

                            "max\_value": value,

                            "above\_max\_count": above\_max\_count

                        }

                        column\_passed = column\_passed and rule\_passed

                elif rule in ["min\_length", "max\_length"] and isinstance(col\_type, StringType):

                    if rule == "min\_length":

                        below\_min\_length\_count = df.filter(length(trim(col(column))) < value).count()

                        rule\_passed = below\_min\_length\_count == 0

                        column\_result["rules\_checked"]["min\_length"] = {

                            "passed": rule\_passed,

                            "min\_length": value,

                            "below\_min\_length\_count": below\_min\_length\_count

                        }

                        column\_passed = column\_passed and rule\_passed

                    else:  # max\_length

                        above\_max\_length\_count = df.filter(length(trim(col(column))) > value).count()

                        rule\_passed = above\_max\_length\_count == 0

                        column\_result["rules\_checked"]["max\_length"] = {

                            "passed": rule\_passed,

                            "max\_length": value,

                            "above\_max\_length\_count": above\_max\_length\_count

                        }

                        column\_passed = column\_passed and rule\_passed

                elif rule == "regex" and isinstance(col\_type, StringType):

                    # Use rlike for regex matching

                    non\_matching\_count = df.filter(~col(column).rlike(value)).count()

                    rule\_passed = non\_matching\_count == 0

                    column\_result["rules\_checked"]["regex"] = {

                        "passed": rule\_passed,

                        "pattern": value,

                        "non\_matching\_count": non\_matching\_count

                    }

                    column\_passed = column\_passed and rule\_passed

                elif rule == "allowed\_values":

                    if isinstance(value, list):

                        non\_matching\_count = df.filter(~col(column).isin(value)).count()

                        rule\_passed = non\_matching\_count == 0

                        column\_result["rules\_checked"]["allowed\_values"] = {

                            "passed": rule\_passed,

                            "allowed\_values": value,

                            "non\_matching\_count": non\_matching\_count

                        }

                        column\_passed = column\_passed and rule\_passed

            # Record column validation result

            column\_result["passed"] = column\_passed

            column\_results[column] = column\_result

            all\_rules\_passed = all\_rules\_passed and column\_passed

        # Record data quality validation results

        self.results["validation\_summary"]["data\_quality"] = all\_rules\_passed

        self.results["details"]["data\_quality"]["column\_results"] = column\_results

        if all\_rules\_passed:

            self.logger.info("Data quality validation passed")

        else:

            self.logger.warning("Data quality validation failed")

        return all\_rules\_passed

    def compare\_with\_reference(self) -> bool:

        """

        Compare the table with a reference table if provided.

        Returns:

            Boolean indicating if comparison passed

        """

        if not self.reference\_table:

            self.logger.info("No reference table provided, skipping comparison")

            self.results["validation\_summary"]["reference\_comparison"] = None

            return True

        self.logger.info(f"Comparing table {self.table\_name} with reference {self.reference\_table}")

        # Load tables

        source\_df = self.load\_table(self.table\_name)

        reference\_df = self.load\_table(self.reference\_table)

        if source\_df is None or reference\_df is None:

            self.results["validation\_summary"]["reference\_comparison"] = False

            self.results["details"]["reference\_comparison"]["error"] = "Failed to load tables"

            return False

        # Initialize comparison results

        comparison\_results = {

            "schema\_match": None,

            "count\_match": None,

            "data\_match": None

        }

        # 1. Compare schemas

        source\_schema = {field.name: str(field.dataType) for field in source\_df.schema.fields

                         if field.name not in self.exclude\_columns}

        reference\_schema = {field.name: str(field.dataType) for field in reference\_df.schema.fields

                            if field.name not in self.exclude\_columns}

        schema\_match = source\_schema == reference\_schema

        comparison\_results["schema\_match"] = schema\_match

        if not schema\_match:

            # Find differences

            source\_cols = set(source\_schema.keys())

            reference\_cols = set(reference\_schema.keys())

            missing\_in\_source = reference\_cols - source\_cols

            missing\_in\_reference = source\_cols - reference\_cols

            # Check for type mismatches in common columns

            common\_cols = source\_cols.intersection(reference\_cols)

            type\_mismatches = []

            for col in common\_cols:

                if source\_schema[col] != reference\_schema[col]:

                    type\_mismatches.append({

                        "column": col,

                        "source\_type": source\_schema[col],

                        "reference\_type": reference\_schema[col]

                    })

            comparison\_results["schema\_differences"] = {

                "missing\_in\_source": list(missing\_in\_source),

                "missing\_in\_reference": list(missing\_in\_reference),

                "type\_mismatches": type\_mismatches

            }

        # 2. Compare row counts

        source\_count = source\_df.count()

        reference\_count = reference\_df.count()

        count\_match = source\_count == reference\_count

        comparison\_results["count\_match"] = count\_match

        comparison\_results["count\_details"] = {

            "source\_count": source\_count,

            "reference\_count": reference\_count,

            "difference": abs(source\_count - reference\_count)

        }

        # 3. Compare data

        # Prepare DataFrames for comparison by selecting common columns

        source\_cols = set(source\_df.columns) - set(self.exclude\_columns)

        reference\_cols = set(reference\_df.columns) - set(self.exclude\_columns)

        common\_cols = list(source\_cols.intersection(reference\_cols))

        if not common\_cols:

            self.logger.error("No common columns found for comparison")

            comparison\_results["data\_match"] = False

            comparison\_results["data\_error"] = "No common columns found for comparison"

        else:

            # Select only common columns

            source\_df\_common = source\_df.select(\*common\_cols)

            reference\_df\_common = reference\_df.select(\*common\_cols)

            # Compare data

            if not self.key\_columns:

                # Use subtract for full dataset comparison

                diff1 = source\_df\_common.subtract(reference\_df\_common)

                diff2 = reference\_df\_common.subtract(source\_df\_common)

                diff1\_count = diff1.count()

                diff2\_count = diff2.count()

                data\_match = diff1\_count == 0 and diff2\_count == 0

                comparison\_results["data\_match"] = data\_match

                comparison\_results["data\_differences"] = {

                    "rows\_in\_source\_not\_in\_reference": diff1\_count,

                    "rows\_in\_reference\_not\_in\_source": diff2\_count,

                    "total\_differences": diff1\_count + diff2\_count

                }

                # Sample differences

                if not data\_match:

                    sample\_diff1 = diff1.limit(5).collect()

                    sample\_diff2 = diff2.limit(5).collect()

                    comparison\_results["data\_differences"]["sample\_differences"] = {

                        "source\_not\_in\_reference": [row.asDict() for row in sample\_diff1],

                        "reference\_not\_in\_source": [row.asDict() for row in sample\_diff2]

                    }

            else:

                # Use key-based comparison

                # Validate key columns exist in both DataFrames

                for key in self.key\_columns:

                    if key not in source\_df\_common.columns or key not in reference\_df\_common.columns:

                        self.logger.error(f"Key column '{key}' not found in both tables")

                        comparison\_results["data\_match"] = False

                        comparison\_results["data\_error"] = f"Key column '{key}' not found in both tables"

                        break

                else:

                    # Join DataFrames on key columns

                    join\_condition = " AND ".join([f"source.{key} <=> reference.{key}" for key in self.key\_columns])

                    joined\_df = source\_df\_common.alias("source").join(

                        reference\_df\_common.alias("reference"),

                        self.spark.sql(join\_condition),

                        "full\_outer"

                    )

                    # Identify rows missing in source or reference

                    missing\_in\_source\_condition = " AND ".join([f"source.{key} IS NULL" for key in self.key\_columns])

                    missing\_in\_reference\_condition = " AND ".join([f"reference.{key} IS NULL" for key in self.key\_columns])

                    # Create conditions for value mismatches

                    value\_columns = [col for col in common\_cols if col not in self.key\_columns]

                    value\_mismatch\_conditions = []

                    for col\_name in value\_columns:

                        # Apply tolerance for numeric columns if specified

                        if col\_name in self.tolerance:

                            tolerance\_value = self.tolerance[col\_name]

                            value\_mismatch\_conditions.append(

                                f"(source.{col\_name} IS NOT NULL AND reference.{col\_name} IS NOT NULL AND " +

                                f"ABS(source.{col\_name} - reference.{col\_name}) > {tolerance\_value})"

                            )

                        else:

                            value\_mismatch\_conditions.append(f"(source.{col\_name} <=> reference.{col\_name}) = false")

                    value\_mismatch\_condition = " OR ".join(value\_mismatch\_conditions) if value\_mismatch\_conditions else "false"

                    # Apply conditions to identify different types of mismatches

                    missing\_in\_source\_df = joined\_df.filter(missing\_in\_source\_condition)

                    missing\_in\_reference\_df = joined\_df.filter(missing\_in\_reference\_condition)

                    value\_mismatch\_df = joined\_df.filter(

                        f"NOT ({missing\_in\_source\_condition}) AND NOT ({missing\_in\_reference\_condition}) AND ({value\_mismatch\_condition})"

                    )

                    # Count differences

                    missing\_in\_source\_count = missing\_in\_source\_df.count()

                    missing\_in\_reference\_count = missing\_in\_reference\_df.count()

                    value\_mismatch\_count = value\_mismatch\_df.count()

                    total\_differences = missing\_in\_source\_count + missing\_in\_reference\_count + value\_mismatch\_count

                    data\_match = total\_differences == 0

                    comparison\_results["data\_match"] = data\_match

                    comparison\_results["data\_differences"] = {

                        "rows\_in\_reference\_not\_in\_source": missing\_in\_source\_count,

                        "rows\_in\_source\_not\_in\_reference": missing\_in\_reference\_count,

                        "rows\_with\_value\_mismatches": value\_mismatch\_count,

                        "total\_differences": total\_differences

                    }

                    # Sample differences

                    if not data\_match:

                        sample\_differences = []

                        # Sample missing in source

                        if missing\_in\_source\_count > 0:

                            sample\_missing\_in\_source = missing\_in\_source\_df.select("reference.\*").limit(5).collect()

                            comparison\_results["data\_differences"]["sample\_missing\_in\_source"] = [

                                row.asDict() for row in sample\_missing\_in\_source

                            ]

                        # Sample missing in reference

                        if missing\_in\_reference\_count > 0:

                            sample\_missing\_in\_reference = missing\_in\_reference\_df.select("source.\*").limit(5).collect()

                            comparison\_results["data\_differences"]["sample\_missing\_in\_reference"] = [

                                row.asDict() for row in sample\_missing\_in\_reference

                            ]

                        # Sample value mismatches

                        if value\_mismatch\_count > 0:

                            sample\_value\_mismatches = []

                            sample\_rows = value\_mismatch\_df.limit(5).collect()

                            for row in sample\_rows:

                                mismatch = {"key\_values": {}, "value\_differences": {}}

                                # Extract key values

                                for key in self.key\_columns:

                                    mismatch["key\_values"][key] = row[f"source.{key}"]

                                # Extract value differences

                                for col\_name in value\_columns:

                                    source\_val = row[f"source.{col\_name}"]

                                    reference\_val = row[f"reference.{col\_name}"]

                                    if source\_val != reference\_val:

                                        mismatch["value\_differences"][col\_name] = {

                                            "source\_value": source\_val,

                                            "reference\_value": reference\_val

                                        }

                                sample\_value\_mismatches.append(mismatch)

                            comparison\_results["data\_differences"]["sample\_value\_mismatches"] = sample\_value\_mismatches

        # Record comparison results

        self.results["details"]["reference\_comparison"] = comparison\_results

        comparison\_passed = (comparison\_results["schema\_match"] if comparison\_results["schema\_match"] is not None else True) and \

                           (comparison\_results["count\_match"] if comparison\_results["count\_match"] is not None else True) and \

                           (comparison\_results["data\_match"] if comparison\_results["data\_match"] is not None else True)

        self.results["validation\_summary"]["reference\_comparison"] = comparison\_passed

        if comparison\_passed:

            self.logger.info("Reference comparison passed")

        else:

            self.logger.warning("Reference comparison failed")

        return comparison\_passed

    def run\_validation(self, expected\_schema: Optional[Dict] = None,

                      expected\_count: Optional[int] = None,

                      min\_count: Optional[int] = None,

                      max\_count: Optional[int] = None,

                      data\_quality\_rules: Optional[Dict] = None) -> Dict:

        """

        Run the full validation process.

        Args:

            expected\_schema: Optional dictionary of expected column names and types

            expected\_count: Optional exact count to validate against

            min\_count: Optional minimum count to validate against

            max\_count: Optional maximum count to validate against

            data\_quality\_rules: Optional dictionary of data quality rules

        Returns:

            Dictionary containing validation results

        """

        self.logger.info(f"Starting validation for table: {self.table\_name}")

        # 1. Validate table exists

        table\_exists = self.validate\_table\_exists()

        if not table\_exists:

            self.logger.error(f"Validation failed: Table {self.table\_name} does not exist")

            return self.results

        # 2. Validate schema

        schema\_valid = self.validate\_schema(expected\_schema)

        # 3. Validate row count

        row\_count\_valid = self.validate\_row\_count(expected\_count, min\_count, max\_count)

        # 4. Validate data quality

        data\_quality\_valid = True

        if data\_quality\_rules:

            data\_quality\_valid = self.validate\_data\_quality(data\_quality\_rules)

        else:

            self.results["validation\_summary"]["data\_quality"] = None

        # 5. Compare with reference table

        reference\_comparison\_valid = self.compare\_with\_reference()

        # Overall validation result

        overall\_valid = table\_exists and schema\_valid and row\_count\_valid and data\_quality\_valid and reference\_comparison\_valid

        self.results["overall\_valid"] = overall\_valid

        # Save results

        self.\_save\_results()

        self.logger.info(f"Validation completed. Overall valid: {overall\_valid}")

        return self.results

    def \_save\_results(self):

        """Save validation results to file."""

        try:

            os.makedirs(self.output\_path, exist\_ok=True)

            timestamp = datetime.now().strftime("%Y%m%d\_%H%M%S")

            filename = f"{self.output\_path}/validation\_results\_{self.table\_name}\_{timestamp}.json"

            with open(filename, 'w') as f:

                json.dump(self.results, f, indent=2)

            self.logger.info(f"Results saved to {filename}")

        except Exception as e:

            self.logger.error(f"Failed to save results: {str(e)}")

def create\_spark\_session(app\_name: str = "Hive Table Validator",

                         configs: Dict = None) -> SparkSession:

    """

    Create and configure a SparkSession with Hive support.

    Args:

        app\_name: Name of the Spark application

        configs: Dictionary of Spark configuration options

    Returns:

        Configured SparkSession with Hive support enabled

    """

    builder = SparkSession.builder.appName(app\_name).enableHiveSupport()

    # Apply additional configurations if provided

    if configs:

        for key, value in configs.items():

            builder = builder.config(key, value)

    # Create and return the SparkSession

    return builder.getOrCreate()

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

    import argparse

    # Parse command line arguments

    parser = argparse.ArgumentParser(description="Validate Hive tables using PySpark")

    parser.add\_argument("table\_name", help="Name of the Hive table to validate")

    parser.add\_argument("--reference", help="Optional reference table for comparison")

    parser.add\_argument("--keys", help="Comma-separated list of key columns for comparison")

    parser.add\_argument("--exclude", help="Comma-separated list of columns to exclude from validation")

    parser.add\_argument("--min-count", type=int, help="Minimum expected row count")

    parser.add\_argument("--max-count", type=int, help="Maximum expected row count")

    parser.add\_argument("--output", default="./validation\_results", help="Path to save validation results")

    args = parser.parse\_args()

    # Create Spark session

    spark = create\_spark\_session("Hive Table Validator")

    # Parse key columns and excluded columns

    key\_columns = args.keys.split(",") if args.keys else None

    exclude\_columns = args.exclude.split(",") if args.exclude else None

    # Create validator

    validator = HiveTableValidator(

        spark=spark,

        table\_name=args.table\_name,

        reference\_table=args.reference,

        key\_columns=key\_columns,

        exclude\_columns=exclude\_columns,

        output\_path=args.output

    )

    # Run validation

    results = validator.run\_validation(

        min\_count=args.min\_count,

        max\_count=args.max\_count

    )

    # Print summary

    print(f"\nValidation Summary for {args.table\_name}:")

    print(f"Overall valid: {results.get('overall\_valid', False)}")

    for check, result in results["validation\_summary"].items():

        status = "Passed" if result is True else "Failed" if result is False else "Not Checked"

        print(f"- {check.replace('\_', ' ').title()}: {status}")

    print(f"\nDetailed results saved to: {args.output}")