```
import pandas as pd
import numpy as np
import logging
from pathlib import Path
from typing import Dict, List, Tuple, Any, Optional
try:
    from fuzzywuzzy import fuzz
    FUZZY_AVAILABLE = True
except ImportError:
    FUZZY_AVAILABLE = False

import openpyxl
from openpyxl.styles import Font, PatternFill, Alignment
import json
import warnings
warnings.filterwarnings('ignore')

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


class CSVComparator:
    """
    A comprehensive tool for comparing two large CSV files with different delimiters.
    Supports manual and automatic column mapping, side-by-side attribute comparison,
    and detailed difference reporting.

    Key Features:
    - Configurable file names (not source/target)
    - No memory monitoring
    - Side-by-side attribute comparison for matched records
    - Column-wise difference statistics
    - Excel report with format: [col]_[file1], [col]_[file2], [col]_Match
    """

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.file1_df = None
        self.file2_df = None
        self.file1_name = config['file1']['name']
        self.file2_name = config['file2']['name']
        self.column_mapping = {}
        self.comparison_results = {}

    def load_csv_file(self, file_config: Dict[str, Any], file_key: str) -> pd.DataFrame:
        """Load and validate CSV file with optimization"""
        file_path = file_config['file_path']
        file_name = file_config['name']
        logger.info(f"Loading {file_name} file: {file_path}")

        if not Path(file_path).exists():
            raise FileNotFoundError(f"{file_name} file not found: {file_path}")

        try:
            # First, read a sample to infer dtypes and optimize memory
            sample_df = pd.read_csv(
                file_path,
                delimiter=file_config['delimiter'],
                encoding=file_config['encoding'],
                na_values=file_config['na_values'],
                nrows=1000
            )

            # Optimize dtypes
            dtypes = {}
            for col in sample_df.columns:
                if sample_df[col].dtype == 'object':
                    try:
                        # Try to convert to numeric
                        pd.to_numeric(sample_df[col], errors='raise')
                        dtypes[col] = 'float64'
                    except:
                        # Keep as string but use category if many repeats
                        if sample_df[col].nunique() / len(sample_df) < 0.5:
                            dtypes[col] = 'category'

            # Load full file with optimized dtypes
            df = pd.read_csv(
                file_path,
                delimiter=file_config['delimiter'],
                encoding=file_config['encoding'],
                na_values=file_config['na_values'],
                dtype=dtypes,
                low_memory=False
            )

            logger.info(f"{file_name} file loaded successfully: {df.shape[0]} rows, {df.shape[1]} columns")
            return df

        except Exception as e:
            raise Exception(f"Error loading {file_name} file: {str(e)}")

    def detect_column_types(self, df: pd.DataFrame, file_name: str) -> Dict[str, str]:
        """Detect and log column types"""
        type_info = {}
        logger.info(f"\nColumn analysis for {file_name} file:")

        for col in df.columns:
            dtype = str(df[col].dtype)
            null_count = df[col].isnull().sum()
            null_pct = (null_count / len(df)) * 100

            if df[col].dtype in ['int64', 'float64']:
                type_category = 'numeric'
            elif df[col].dtype == 'object':
                type_category = 'text'
            elif df[col].dtype == 'category':
                type_category = 'categorical'
            else:
                type_category = 'other'

            type_info[col] = {
                'dtype': dtype,
                'category': type_category,
```

```
            'null_count': null_count,
            'null_percentage': null_pct
        }

        logger.info(f"  {col}: {dtype} ({null_pct:.1f}% null)")

    return type_info

def simple_string_similarity(self, str1: str, str2: str) -> int:
    """Simple string similarity calculation when fuzzywuzzy is not available"""
    str1, str2 = str1.lower(), str2.lower()

    # Exact match
    if str1 == str2:
        return 100

    # Check if one is contained in the other
    if str1 in str2 or str2 in str1:
        return 90

    # Simple character overlap
    set1, set2 = set(str1), set(str2)
    overlap = len(set1.intersection(set2))
    total = len(set1.union(set2))

    if total == 0:
        return 0

    return int((overlap / total) * 100)

def fuzzy_match_columns(self, file1_cols: List[str], file2_cols: List[str],
                        threshold: int = 80) -> Dict[str, str]:
    """Automatically match columns using fuzzy string matching"""
    logger.info(f"\nPerforming fuzzy column matching (threshold: {threshold}%)")
    auto_mapping = {}

    for file1_col in file1_cols:
        best_match = None
        best_score = 0

        for file2_col in file2_cols:
            if FUZZY_AVAILABLE:
                # Try different matching algorithms
                ratio = fuzz.ratio(file1_col.lower(), file2_col.lower())
                partial_ratio = fuzz.partial_ratio(file1_col.lower(), file2_col.lower())
                token_sort_ratio = fuzz.token_sort_ratio(file1_col.lower(), file2_col.lower())
                score = max(ratio, partial_ratio, token_sort_ratio)
            else:
                # Use simple similarity
                score = self.simple_string_similarity(file1_col, file2_col)

            if score > best_score and score >= threshold:
                best_score = score
                best_match = file2_col

        if best_match:
            auto_mapping[file1_col] = best_match
            logger.info(f"  {file1_col} -> {best_match} (score: {best_score}%)")

    logger.info(f"Auto-matched {len(auto_mapping)} column pairs")
    return auto_mapping

def create_column_mapping(self) -> Dict[str, str]:
    """Create combined column mapping from manual and auto-detection"""
    logger.info("\nCreating column mapping...")

    file1_cols = list(self.file1_df.columns)
    file2_cols = list(self.file2_df.columns)

    # Start with manual mapping
    manual_mapping = self.config.get('mapping', {}).get('manual_mapping', {})
    final_mapping = manual_mapping.copy()

    logger.info(f"Manual mappings: {len(manual_mapping)}")
    for src, tgt in manual_mapping.items():
        logger.info(f"  {src} -> {tgt}")

    # Add auto-detection if enabled
    if self.config.get('mapping', {}).get('auto_detect', False):
        # Only auto-match columns not already manually mapped
        unmapped_file1_cols = [col for col in file1_cols if col not in final_mapping]
        mapped_file2_cols = list(final_mapping.values())
        unmapped_file2_cols = [col for col in file2_cols if col not in mapped_file2_cols]

        auto_mapping = self.fuzzy_match_columns(unmapped_file1_cols, unmapped_file2_cols)

        # Add auto mappings to final mapping
        final_mapping.update(auto_mapping)

    # Validate mappings
    missing_file1_cols = [col for col in final_mapping.keys() if col not in file1_cols]
    missing_file2_cols = [col for col in final_mapping.values() if col not in file2_cols]

    if missing_file1_cols:
        logger.warning(f"Mapped {self.file1_name} columns not found: {missing_file1_cols}")
    if missing_file2_cols:
        logger.warning(f"Mapped {self.file2_name} columns not found: {missing_file2_cols}")

    self.column_mapping = final_mapping
    logger.info(f"Final mapping contains {len(final_mapping)} column pairs")

    return final_mapping

def validate_key_columns(self) -> Tuple[str, str]:
    """Validate that key columns exist in both files"""
    file1_key = self.config['key_columns']['file1']
    file2_key = self.config['key_columns']['file2']

    if file1_key not in self.file1_df.columns:
        raise ValueError(f"{self.file1_name} key column '{file1_key}' not found in file")

    if file2_key not in self.file2_df.columns:
        raise ValueError(f"{self.file2_name} key column '{file2_key}' not found in file")

    logger.info(f"Key columns validated - {self.file1_name}: '{file1_key}', {self.file2_name}: '{file2_key}'")
    return file1_key, file2_key

def _create_side_by_side_comparison(self, file1_key: str, file2_key: str, common_keys: set) -> pd.DataFrame:
    """Create side-by-side comparison DataFrame for matched records"""
    logger.info("Creating side-by-side attribute comparison...")
```

```
        # Filter dataframes to only common keys
        file1_matched = self.file1_df[self.file1_df[file1_key].isin(common_keys)].copy()
        file2_matched = self.file2_df[self.file2_df[file2_key].isin(common_keys)].copy()

        # Sort by key for consistent order
        file1_matched = file1_matched.sort_values(file1_key).reset_index(drop=True)
        file2_matched = file2_matched.sort_values(file2_key).reset_index(drop=True)

        # Create the comparison DataFrame
        comparison_data = {}

        # Add the key column first
        comparison_data[file1_key] = file1_matched[file1_key]

        # For each mapped column pair, create three columns: file1_value, file2_value, match_flag
        for file1_col, file2_col in self.column_mapping.items():
            if file1_col in file1_matched.columns and file2_col in file2_matched.columns:
                # Column values from both files
                file1_values = file1_matched[file1_col]
                file2_values = file2_matched[file2_col]

                # Create column names with file names (clean names for Excel)
                file1_clean = self.file1_name.replace(' ', '_').replace('-', '_')
                file2_clean = self.file2_name.replace(' ', '_').replace('-', '_')

                col1_name = f"{file1_col}_{file1_clean}"
                col2_name = f"{file1_col}_{file2_clean}"
                match_name = f"{file1_col}_Match"

                # Add values
                comparison_data[col1_name] = file1_values.values
                comparison_data[col2_name] = file2_values.values

                # Calculate match flag
                matches = []
                for val1, val2 in zip(file1_values, file2_values):
                    # Handle NaN values
                    val1_is_na = pd.isna(val1)
                    val2_is_na = pd.isna(val2)

                    if val1_is_na and val2_is_na:
                        matches.append("Yes")  # Both are NaN, consider as matching
                    elif val1_is_na != val2_is_na:
                        matches.append("No")   # One is NaN, other is not
                    else:
                        # Compare string representations after stripping whitespace
                        str_val1 = str(val1).strip() if not val1_is_na else ""
                        str_val2 = str(val2).strip() if not val2_is_na else ""
                        matches.append("Yes" if str_val1 == str_val2 else "No")

                comparison_data[match_name] = matches

        comparison_df = pd.DataFrame(comparison_data)
        logger.info(f"Created side-by-side comparison with {len(comparison_df)} matched records")

        return comparison_df

    def _calculate_column_difference_stats(self, comparison_df: pd.DataFrame) -> Dict[str, Any]:
        """Calculate difference statistics for each column"""
        stats = {}

        # Find all match columns
        match_columns = [col for col in comparison_df.columns if col.endswith('_Match')]

        for match_col in match_columns:
            # Extract base column name (remove '_Match' suffix)
            base_col = match_col.replace('_Match', '')

            # Count matches and mismatches
            total_records = len(comparison_df)
            matches = (comparison_df[match_col] == "Yes").sum()
            mismatches = (comparison_df[match_col] == "No").sum()

            stats[base_col] = {
                'total_records': total_records,
                'matches': matches,
                'mismatches': mismatches,
                'mismatch_percentage': (mismatches / total_records) * 100 if total_records > 0 else 0
            }

        return stats

    def compare_records(self) -> Dict[str, Any]:
        """Compare records between files with side-by-side attribute comparison"""
        logger.info("\nStarting record comparison...")

        file1_key, file2_key = self.validate_key_columns()

        # Prepare results dictionary
        results = {
            'matched_attributes_df': None,
            'file1_only_records': [],
            'file2_only_records': [],
            'summary_stats': {},
            'column_difference_stats': {}
        }

        # Get sets of keys for comparison
        file1_keys = set(self.file1_df[file1_key].dropna())
        file2_keys = set(self.file2_df[file2_key].dropna())

        common_keys = file1_keys.intersection(file2_keys)
        file1_only_keys = file1_keys - file2_keys
        file2_only_keys = file2_keys - file1_keys

        logger.info(f"Record analysis:")
        logger.info(f"  {self.file1_name} records: {len(file1_keys)}")
        logger.info(f"  {self.file2_name} records: {len(file2_keys)}")
        logger.info(f"  Common keys: {len(common_keys)}")
        logger.info(f"  {self.file1_name} only: {len(file1_only_keys)}")
        logger.info(f"  {self.file2_name} only: {len(file2_only_keys)}")

        # Store file-only records
        results['file1_only_records'] = list(file1_only_keys)
        results['file2_only_records'] = list(file2_only_keys)

        # Create side-by-side comparison for matched records
        if common_keys:
            results['matched_attributes_df'] = self._create_side_by_side_comparison(
```

```
            file1_key, file2_key, common_keys
        )
        results['column_difference_stats'] = self._calculate_column_difference_stats(
            results['matched_attributes_df']
        )

    # Calculate summary statistics
    total_file1 = len(file1_keys)
    total_file2 = len(file2_keys)
    matched_count = len(common_keys)

    # Count perfectly matched records (all attributes match)
    perfect_matches = 0
    records_with_differences = 0

    if results['matched_attributes_df'] is not None:
        match_columns = [col for col in results['matched_attributes_df'].columns if col.endswith('_Match')]
        if match_columns:
            # Count rows where all match columns are "Yes"
            perfect_matches = results['matched_attributes_df'][match_columns].eq("Yes").all(axis=1).sum()
            records_with_differences = matched_count - perfect_matches

    results['summary_stats'] = {
        'total_file1_records': total_file1,
        'total_file2_records': total_file2,
        'matched_records': matched_count,
        'perfectly_matched_records': perfect_matches,
        'records_with_differences': records_with_differences,
        'file1_only_records': len(file1_only_keys),
        'file2_only_records': len(file2_only_keys),
        'match_rate': (perfect_matches / matched_count) * 100 if matched_count > 0 else 0,
        'data_completeness_file1': ((total_file1 - len(file1_only_keys)) / total_file1) * 100 if total_file1 > 0 else 0,
        'data_completeness_file2': ((total_file2 - len(file2_only_keys)) / total_file2) * 100 if total_file2 > 0 else 0
    }

    self.comparison_results = results
    return results

def generate_excel_report(self) -> str:
    """Generate comprehensive Excel report with multiple sheets"""
    logger.info("\nGenerating Excel report...")

    output_path = self.config['output']['report_path']

    with pd.ExcelWriter(output_path, engine='openpyxl') as writer:
        # Sheet 1: Summary Statistics
        self._create_summary_sheet(writer)

        # Sheet 2: Column Mappings
        self._create_mapping_sheet(writer)

        # Sheet 3: Matched Attributes (side-by-side comparison)
        self._create_matched_attributes_sheet(writer)

        # Sheet 4: File1 Only Records
        self._create_file1_only_sheet(writer)

        # Sheet 5: File2 Only Records
        self._create_file2_only_sheet(writer)

        # Sheet 6: Difference Statistics
        self._create_difference_stats_sheet(writer)

    # Apply formatting
    self._format_excel_report(output_path)

    logger.info(f"Excel report generated: {output_path}")
    return output_path

def _create_summary_sheet(self, writer):
    """Create summary statistics sheet"""
    stats = self.comparison_results['summary_stats']

    summary_data = [
        ['Metric', 'Value'],
        [f'{self.file1_name} Name', self.file1_name],
        [f'{self.file2_name} Name', self.file2_name],
        [f'Total {self.file1_name} Records', stats['total_file1_records']],
        [f'Total {self.file2_name} Records', stats['total_file2_records']],
        ['Common Records (Matched Keys)', stats['matched_records']],
        ['Perfectly Matched Records', stats['perfectly_matched_records']],
        ['Records with Differences', stats['records_with_differences']],
        [f'{self.file1_name} Only Records', stats['file1_only_records']],
        [f'{self.file2_name} Only Records', stats['file2_only_records']],
        ['Perfect Match Rate (%)', f"{stats['match_rate']:.2f}%"],
        [f'{self.file1_name} Data Completeness (%)', f"{stats['data_completeness_file1']:.2f}%"],
        [f'{self.file2_name} Data Completeness (%)', f"{stats['data_completeness_file2']:.2f}%"],
        ['Total Column Mappings', len(self.column_mapping)],
        ['', ''],
        ['Column-wise Mismatch Summary', ''],
    ]

    # Add column-wise mismatch statistics
    if self.comparison_results.get('column_difference_stats'):
        for col_name, col_stats in self.comparison_results['column_difference_stats'].items():
            summary_data.append([f'{col_name} Mismatches', col_stats['mismatches']])
            summary_data.append([f'{col_name} Mismatch %', f"{col_stats['mismatch_percentage']:.2f}%"])

    summary_df = pd.DataFrame(summary_data)
    summary_df.to_excel(writer, sheet_name='Summary', index=False, header=False)

def _create_mapping_sheet(self, writer):
    """Create column mappings sheet"""
    mapping_data = []
    manual_mapping = self.config.get('mapping', {}).get('manual_mapping', {})

    for file1_col, file2_col in self.column_mapping.items():
        mapping_data.append({
            f'{self.file1_name} Column': file1_col,
            f'{self.file2_name} Column': file2_col,
            'Mapping Type': 'Manual' if file1_col in manual_mapping else 'Auto'
        })

    if mapping_data:
        mapping_df = pd.DataFrame(mapping_data)
        mapping_df.to_excel(writer, sheet_name='Column Mappings', index=False)

def _create_matched_attributes_sheet(self, writer):
    """Create the main side-by-side comparison sheet"""
    if self.comparison_results['matched_attributes_df'] is not None:
```

```
        # Limit to first 10,000 rows for Excel performance
        df_to_write = self.comparison_results['matched_attributes_df'].head(10000)
        df_to_write.to_excel(writer, sheet_name='Matched Attributes', index=False)

        if len(self.comparison_results['matched_attributes_df']) > 10000:
            logger.info(f"Note: Only first 10,000 records written to Excel (total: {len(self.comparison_results['matched_attributes_df'])})")

    def _create_file1_only_sheet(self, writer):
        """Create sheet for records only in file1"""
        file1_only_keys = self.comparison_results['file1_only_records']
        if file1_only_keys:
            file1_key = self.config['key_columns']['file1']
            file1_only_df = self.file1_df[self.file1_df[file1_key].isin(file1_only_keys)]
            file1_only_df.to_excel(writer, sheet_name=f'{self.file1_name} Only Records', index=False)

    def _create_file2_only_sheet(self, writer):
        """Create sheet for records only in file2"""
        file2_only_keys = self.comparison_results['file2_only_records']
        if file2_only_keys:
            file2_key = self.config['key_columns']['file2']
            file2_only_df = self.file2_df[self.file2_df[file2_key].isin(file2_only_keys)]
            file2_only_df.to_excel(writer, sheet_name=f'{self.file2_name} Only Records', index=False)

    def _create_difference_stats_sheet(self, writer):
        """Create difference statistics analysis sheet"""
        if not self.comparison_results.get('column_difference_stats'):
            return

        stats_data = []
        stats_data.append(['Column', 'Total Records', 'Matches', 'Mismatches', 'Mismatch %'])

        for col_name, col_stats in self.comparison_results['column_difference_stats'].items():
            stats_data.append([
                col_name,
                col_stats['total_records'],
                col_stats['matches'],
                col_stats['mismatches'],
                f"{col_stats['mismatch_percentage']:.2f}%"
            ])

        stats_df = pd.DataFrame(stats_data[1:], columns=stats_data[0])
        stats_df.to_excel(writer, sheet_name='Difference Statistics', index=False)

    def _format_excel_report(self, file_path: str):
        """Apply formatting to the Excel report"""
        try:
            workbook = openpyxl.load_workbook(file_path)

            # Define styles
            header_font = Font(bold=True, color="FFFFFF")
            header_fill = PatternFill(start_color="366092", end_color="366092", fill_type="solid")

            for sheet_name in workbook.sheetnames:
                worksheet = workbook[sheet_name]

                # Format header row
                if worksheet.max_row > 0:
                    for cell in worksheet[1]:
                        if cell.value:
                            cell.font = header_font
                            cell.fill = header_fill
                            cell.alignment = Alignment(horizontal='center')

                # Auto-adjust column widths
                for column in worksheet.columns:
                    max_length = 0
                    column = [cell for cell in column]
                    for cell in column:
                        try:
                            if len(str(cell.value)) > max_length:
                                max_length = len(str(cell.value))
                        except:
                            pass
                    adjusted_width = min(max_length + 2, 50)
                    worksheet.column_dimensions[column[0].column_letter].width = adjusted_width

            workbook.save(file_path)
            logger.info("Excel formatting applied successfully")

        except Exception as e:
            logger.warning(f"Could not apply Excel formatting: {str(e)}")

    def run_comparison(self) -> Dict[str, Any]:
        """Run the complete comparison process"""
        logger.info("Starting CSV comparison process...")

        try:
            # Load files
            self.file1_df = self.load_csv_file(self.config['file1'], 'file1')
            self.file2_df = self.load_csv_file(self.config['file2'], 'file2')

            # Detect column types
            file1_types = self.detect_column_types(self.file1_df, self.file1_name)
            file2_types = self.detect_column_types(self.file2_df, self.file2_name)

            # Create column mapping
            self.create_column_mapping()

            # Compare records
            results = self.compare_records()

            # Generate Excel report
            report_path = self.generate_excel_report()

            # Log final summary
            self._log_final_summary(results)

            return {
                'success': True,
                'results': results,
                'report_path': report_path
            }

        except Exception as e:
            logger.error(f"Comparison failed: {str(e)}")
            return {
                'success': False,
                'error': str(e)
            }
```

```
    def _log_final_summary(self, results: Dict[str, Any]):
        """Log final summary to console"""
        stats = results['summary_stats']

        print("\n" + "="*80)
        print("CSV COMPARISON SUMMARY")
        print("="*80)
        print(f"{self.file1_name} Records: {stats['total_file1_records']:,}")
        print(f"{self.file2_name} Records: {stats['total_file2_records']:,}")
        print(f"Common Records: {stats['matched_records']:,}")
        print(f"Perfectly Matched Records: {stats['perfectly_matched_records']:,}")
        print(f"Records with Differences: {stats['records_with_differences']:,}")
        print(f"{self.file1_name} Only Records: {stats['file1_only_records']:,}")
        print(f"{self.file2_name} Only Records: {stats['file2_only_records']:,}")
        print(f"Perfect Match Rate: {stats['match_rate']:.2f}%")
        print(f"Column Mappings: {len(self.column_mapping)}")

        # Column-wise differences
        if self.comparison_results.get('column_difference_stats'):
            print(f"\nCOLUMN-WISE DIFFERENCES:")
            print("-"*40)
            for col_name, col_stats in self.comparison_results['column_difference_stats'].items():
                print(f"{col_name}: {col_stats['mismatches']} mismatches ({col_stats['mismatch_percentage']:.2f}%)")

        # Actionable insights
        print("\nACTIONABLE INSIGHTS:")
        print("-"*40)

        if stats['match_rate'] < 90:
            print("âš  Low match rate detected. Consider reviewing:")
            print("  - Key column data quality")
            print("  - Data standardization processes")

        if stats['records_with_differences'] > stats['perfectly_matched_records'] * 0.1:
            print("âš  High number of attribute differences detected")
            print("  - Review data transformation rules")
            print("  - Check for data type inconsistencies")

        if stats['file1_only_records'] > 0 or stats['file2_only_records'] > 0:
            print("â„¹ Missing records detected:")
            if stats['file1_only_records'] > 0:
                print(f"  - {stats['file1_only_records']} records only in {self.file1_name}")
            if stats['file2_only_records'] > 0:
                print(f"  - {stats['file2_only_records']} records only in {self.file2_name}")

        print("\n" + "="*80)


def main():
    """Example usage of the modified CSV Comparator"""
    config = {
        "file1": {
            "name": "Production DB",
            "file_path": "file1.csv",
            "delimiter": ",",
            "encoding": "utf-8",
            "na_values": ["", "NULL", "N/A", "?", "#", "null", "nan"]
        },
        "file2": {
            "name": "Staging DB",
            "file_path": "file2.csv",
            "delimiter": "|",
            "encoding": "utf-8",
            "na_values": ["", "NULL", "N/A", "?", "#", "null", "nan"]
        },
        "mapping": {
            "manual_mapping": {
                "customer_id": "cust_id",
                "first_name": "fname",
                "last_name": "lname"
            },
            "auto_detect": True
        },
        "key_columns": {
            "file1": "customer_id",
            "file2": "cust_id"
        },
        "output": {
            "report_path": "comparison_report.xlsx"
        }
    }

    comparator = CSVComparator(config)
    result = comparator.run_comparison()

    if result['success']:
        print(f"âœ… Comparison completed successfully!")
        print(f"ðŸ“Š Report saved to: {result['report_path']}")
    else:
        print(f"â Œ Comparison failed: {result['error']}")


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