python

# Cell 1: Import required libraries

import pandas as pd

import numpy as np

import io

import json

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

from dataclasses import dataclass

from datetime import datetime, timedelta

import hashlib

import logging

import random

import string

# Configure logging

logging.basicConfig(level=logging.INFO)

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

python

# Cell 2: Define data classes

@dataclass

class ChunkMetadata:

"""Metadata for each chunk"""

chunk\_id: str

method: str

start\_row: int

end\_row: int

total\_rows: int

columns: List[str]

overlap\_rows: int

entity\_group: Optional[str]

created\_at: str

source\_info: Dict[str, Any]

python

# Cell 3: SchemaAnalyzer class

class SchemaAnalyzer:

"""Analyzes CSV schema and relationships"""

def \_\_init\_\_(self, df: pd.DataFrame):

self.df = df

self.analysis\_results = {}

def analyze\_schema(self) -> Dict[str, Any]:

"""Perform comprehensive schema analysis"""

return {

'total\_rows': len(self.df),

'total\_columns': len(self.df.columns),

'column\_types': dict(self.df.dtypes.astype(str)),

'memory\_usage': self.df.memory\_usage(deep=True).sum(),

'null\_counts': dict(self.df.isnull().sum()),

'duplicate\_rows': self.df.duplicated().sum()

}

def detect\_data\_types(self) -> Dict[str, List[str]]:

"""Classify columns by data type"""

numeric\_cols = self.df.select\_dtypes(include=[np.number]).columns.tolist()

text\_cols = self.df.select\_dtypes(include=['object']).columns.tolist()

datetime\_cols = self.df.select\_dtypes(include=['datetime64']).columns.tolist()

return {

'numeric': numeric\_cols,

'text': text\_cols,

'datetime': datetime\_cols

}

def detect\_primary\_keys(self) -> List[str]:

"""Detect potential primary key columns"""

candidates = []

for col in self.df.columns:

unique\_ratio = self.df[col].nunique() / len(self.df)

null\_ratio = self.df[col].isnull().sum() / len(self.df)

# High uniqueness, low nulls = potential primary key

if unique\_ratio > 0.9 and null\_ratio < 0.1:

candidates.append(col)

return candidates

def analyze\_column\_relationships(self) -> Dict[str, Any]:

"""Analyze relationships between columns"""

relationships = {}

primary\_key\_candidates = self.detect\_primary\_keys()

for col in self.df.columns:

if col in primary\_key\_candidates:

continue

# Check for foreign key relationships

for pk\_col in primary\_key\_candidates:

if col != pk\_col:

# Simple heuristic: if values in col exist as values in pk\_col

intersection = set(self.df[col].dropna()) & set(self.df[pk\_col].dropna())

if len(intersection) > 0:

relationships[col] = {'potential\_fk\_to': pk\_col, 'match\_ratio': len(intersection) / self.df[col].nunique()}

return {

'primary\_keys': primary\_key\_candidates,

'foreign\_keys': relationships

}

python

# Cell 4: FixedChunker class

class FixedChunker:

"""Fixed-size chunking implementation"""

def \_\_init\_\_(self, chunk\_size: int = 10000, overlap\_ratio: float = 0.1):

self.chunk\_size = chunk\_size

self.overlap\_ratio = overlap\_ratio

self.overlap\_size = int(chunk\_size \* overlap\_ratio)

def chunk(self, df: pd.DataFrame, source\_info: Dict[str, Any]) -> List[Tuple[pd.DataFrame, ChunkMetadata]]:

"""Generate fixed-size chunks with overlap"""

chunks = []

for i in range(0, len(df), self.chunk\_size - self.overlap\_size):

start\_idx = i

end\_idx = min(i + self.chunk\_size, len(df))

chunk\_df = df.iloc[start\_idx:end\_idx].copy()

# Create metadata

chunk\_id = f"fixed\_{i // (self.chunk\_size - self.overlap\_size):04d}"

metadata = ChunkMetadata(

chunk\_id=chunk\_id,

method="fixed",

start\_row=start\_idx,

end\_row=end\_idx,

total\_rows=len(chunk\_df),

columns=chunk\_df.columns.tolist(),

overlap\_rows=self.overlap\_size if i > 0 else 0,

entity\_group=None,

created\_at=datetime.now().isoformat(),

source\_info=source\_info

)

chunks.append((chunk\_df, metadata))

if end\_idx >= len(df):

break

return chunks

python

# Cell 5: RecursiveChunker class

class RecursiveChunker:

"""Recursive entity-aware chunking implementation"""

def \_\_init\_\_(self, max\_chunk\_size: int = 50000, min\_chunk\_size: int = 1000):

self.max\_chunk\_size = max\_chunk\_size

self.min\_chunk\_size = min\_chunk\_size

def chunk(self, df: pd.DataFrame, source\_info: Dict[str, Any], primary\_keys: List[str]) -> List[Tuple[pd.DataFrame, ChunkMetadata]]:

"""Generate entity-aware chunks with recursive splitting"""

chunks = []

if not primary\_keys:

# Fallback to simple row-based chunking if no primary keys

return self.\_fallback\_chunking(df, source\_info)

# Group by primary key combinations

if len(primary\_keys) == 1:

groups = df.groupby(primary\_keys[0])

else:

groups = df.groupby(primary\_keys)

current\_chunk\_data = []

current\_chunk\_size = 0

chunk\_counter = 0

for name, group in groups:

group\_size = len(group)

# If single group is too large, split it recursively

if group\_size > self.max\_chunk\_size:

# Create chunks from the accumulated data first

if current\_chunk\_data:

chunk\_df = pd.concat(current\_chunk\_data, ignore\_index=True)

chunks.append(self.\_create\_chunk(chunk\_df, chunk\_counter, source\_info, str(name)))

chunk\_counter += 1

current\_chunk\_data = []

current\_chunk\_size = 0

# Split the large group recursively

sub\_chunks = self.\_split\_large\_group(group, chunk\_counter, source\_info, str(name))

chunks.extend(sub\_chunks)

chunk\_counter += len(sub\_chunks)

# If adding this group exceeds limit, finalize current chunk

elif current\_chunk\_size + group\_size > self.max\_chunk\_size and current\_chunk\_data:

chunk\_df = pd.concat(current\_chunk\_data, ignore\_index=True)

chunks.append(self.\_create\_chunk(chunk\_df, chunk\_counter, source\_info, "mixed\_entities"))

chunk\_counter += 1

current\_chunk\_data = [group]

current\_chunk\_size = group\_size

else:

current\_chunk\_data.append(group)

current\_chunk\_size += group\_size

# Handle remaining data

if current\_chunk\_data:

chunk\_df = pd.concat(current\_chunk\_data, ignore\_index=True)

chunks.append(self.\_create\_chunk(chunk\_df, chunk\_counter, source\_info, "mixed\_entities"))

return chunks

def \_split\_large\_group(self, group: pd.DataFrame, start\_counter: int, source\_info: Dict[str, Any], entity\_name: str) -> List[Tuple[pd.DataFrame, ChunkMetadata]]:

"""Recursively split large groups"""

chunks = []

for i in range(0, len(group), self.max\_chunk\_size):

chunk\_df = group.iloc[i:i + self.max\_chunk\_size].copy()

metadata = ChunkMetadata(

chunk\_id=f"recursive\_{start\_counter:04d}\_{i // self.max\_chunk\_size:02d}",

method="recursive",

start\_row=i,

end\_row=min(i + self.max\_chunk\_size, len(group)),

total\_rows=len(chunk\_df),

columns=chunk\_df.columns.tolist(),

overlap\_rows=0,

entity\_group=f"{entity\_name}\_split\_{i // self.max\_chunk\_size}",

created\_at=datetime.now().isoformat(),

source\_info=source\_info

)

chunks.append((chunk\_df, metadata))

return chunks

def \_create\_chunk(self, chunk\_df: pd.DataFrame, chunk\_counter: int, source\_info: Dict[str, Any], entity\_group: str) -> Tuple[pd.DataFrame, ChunkMetadata]:

"""Create chunk with metadata"""

metadata = ChunkMetadata(

chunk\_id=f"recursive\_{chunk\_counter:04d}",

method="recursive",

start\_row=0,

end\_row=len(chunk\_df),

total\_rows=len(chunk\_df),

columns=chunk\_df.columns.tolist(),

overlap\_rows=0,

entity\_group=entity\_group,

created\_at=datetime.now().isoformat(),

source\_info=source\_info

)

return (chunk\_df, metadata)

def \_fallback\_chunking(self, df: pd.DataFrame, source\_info: Dict[str, Any]) -> List[Tuple[pd.DataFrame, ChunkMetadata]]:

"""Fallback to simple chunking when no entities detected"""

chunks = []

for i in range(0, len(df), self.max\_chunk\_size):

chunk\_df = df.iloc[i:i + self.max\_chunk\_size].copy()

metadata = ChunkMetadata(

chunk\_id=f"recursive\_fallback\_{i // self.max\_chunk\_size:04d}",

method="recursive\_fallback",

start\_row=i,

end\_row=min(i + self.max\_chunk\_size, len(df)),

total\_rows=len(chunk\_df),

columns=chunk\_df.columns.tolist(),

overlap\_rows=0,

entity\_group="no\_entities\_detected",

created\_at=datetime.now().isoformat(),

source\_info=source\_info

)

chunks.append((chunk\_df, metadata))

return chunks

python

# Cell 6: QualityValidator class

class QualityValidator:

"""Quality validation for chunks"""

def \_\_init\_\_(self, original\_df: pd.DataFrame):

self.original\_df = original\_df

self.original\_schema = set(original\_df.columns)

self.original\_row\_count = len(original\_df)

def validate\_chunks(self, chunks: List[Tuple[pd.DataFrame, ChunkMetadata]]) -> Dict[str, Any]:

"""Comprehensive quality validation"""

results = {

'schema\_consistency': True,

'data\_completeness': True,

'boundary\_validation': True,

'total\_rows\_preserved': 0,

'issues': []

}

total\_rows = 0

for chunk\_df, metadata in chunks:

# Schema consistency check

chunk\_schema = set(chunk\_df.columns)

if chunk\_schema != self.original\_schema:

results['schema\_consistency'] = False

results['issues'].append(f"Schema mismatch in chunk {metadata.chunk\_id}")

# Count rows

total\_rows += len(chunk\_df)

# Boundary validation

if len(chunk\_df) == 0:

results['boundary\_validation'] = False

results['issues'].append(f"Empty chunk detected: {metadata.chunk\_id}")

results['total\_rows\_preserved'] = total\_rows

# Data completeness check

if total\_rows < self.original\_row\_count:

results['data\_completeness'] = False

results['issues'].append(f"Row count mismatch: {total\_rows} vs {self.original\_row\_count}")

return results

python

# Cell 7: Generate synthetic dataset with text, numeric, categorical, and datetime data

def generate\_synthetic\_dataset(n\_rows=10000):

"""Generate a comprehensive synthetic dataset with multiple data types"""

np.random.seed(42)

random.seed(42)

# Generate unique IDs (numeric)

ids = list(range(1, n\_rows + 1))

# Generate categorical data

categories = ['Electronics', 'Clothing', 'Books', 'Home', 'Sports', 'Beauty', 'Toys', 'Food']

subcategories = {

'Electronics': ['Phones', 'Laptops', 'Tablets', 'Accessories'],

'Clothing': ['Men', 'Women', 'Kids', 'Accessories'],

'Books': ['Fiction', 'Non-Fiction', 'Educational', 'Children'],

'Home': ['Furniture', 'Decor', 'Kitchen', 'Garden'],

'Sports': ['Outdoor', 'Fitness', 'Team Sports', 'Water Sports'],

'Beauty': ['Skincare', 'Makeup', 'Haircare', 'Fragrance'],

'Toys': ['Educational', 'Outdoor', 'Puzzles', 'Dolls'],

'Food': ['Snacks', 'Beverages', 'Frozen', 'Fresh']

}

categories\_col = np.random.choice(categories, n\_rows)

subcategories\_col = [random.choice(subcategories[cat]) for cat in categories\_col]

# Generate numeric data

prices = np.round(np.random.uniform(5, 1000, n\_rows), 2)

quantities = np.random.randint(1, 100, n\_rows)

ratings = np.round(np.random.uniform(1, 5, n\_rows), 1)

weights = np.round(np.random.uniform(0.1, 50, n\_rows), 2)

# Generate datetime data

start\_date = datetime(2023, 1, 1)

dates = [start\_date + timedelta(days=np.random.randint(0, 365)) for \_ in range(n\_rows)]

timestamps = [dt + timedelta(seconds=np.random.randint(0, 86400)) for dt in dates]

# Generate text data

def generate\_text(length=10):

return ''.join(random.choices(string.ascii\_letters + string.digits, k=length))

product\_names = [f"Product\_{generate\_text(8)}" for \_ in range(n\_rows)]

descriptions = [f"This is a detailed description of {name} with features and benefits."

for name in product\_names]

# Generate boolean data

in\_stock = np.random.choice([True, False], n\_rows, p=[0.8, 0.2])

on\_sale = np.random.choice([True, False], n\_rows, p=[0.3, 0.7])

# Create DataFrame

df = pd.DataFrame({

'product\_id': ids,

'product\_name': product\_names,

'description': descriptions,

'category': categories\_col,

'subcategory': subcategories\_col,

'price': prices,

'quantity': quantities,

'weight\_kg': weights,

'customer\_rating': ratings,

'created\_date': dates,

'last\_updated': timestamps,

'in\_stock': in\_stock,

'on\_sale': on\_sale

})

# Add some null values to make it more realistic

for col in ['price', 'quantity', 'customer\_rating']:

null\_indices = np.random.choice(n\_rows, size=int(n\_rows \* 0.05), replace=False)

df.loc[null\_indices, col] = None

return df

python

# Cell 8: Demo function for Jupyter Notebook

def demo\_chunking():

"""Demo function to show chunking capabilities in Jupyter"""

# Create synthetic data with multiple data types

df = generate\_synthetic\_dataset(15000)

print(f"Created synthetic DataFrame with {len(df)} rows and {len(df.columns)} columns")

# Display data info

print("\nDataFrame Info:")

print(f"Shape: {df.shape}")

print(f"Columns: {list(df.columns)}")

print(f"Data Types:\n{df.dtypes}")

# Show sample data

print("\nSample Data:")

print(df.head(3))

# Analyze schema

analyzer = SchemaAnalyzer(df)

analysis = {

'schema': analyzer.analyze\_schema(),

'data\_types': analyzer.detect\_data\_types(),

'relationships': analyzer.analyze\_column\_relationships()

}

print("\nSchema Analysis:")

print(f"- Primary keys detected: {analysis['relationships']['primary\_keys']}")

print(f"- Foreign key relationships: {len(analysis['relationships']['foreign\_keys'])}")

print(f"- Data types: {analysis['data\_types']}")

# Test fixed chunking

print("\n--- Fixed Chunking Test ---")

fixed\_chunker = FixedChunker(chunk\_size=2500, overlap\_ratio=0.1)

fixed\_chunks = fixed\_chunker.chunk(df, {'method': 'fixed', 'params': {'chunk\_size': 2500, 'overlap\_ratio': 0.1}})

print(f"Generated {len(fixed\_chunks)} fixed chunks")

for i, (chunk\_df, metadata) in enumerate(fixed\_chunks[:3]): # Show first 3 chunks

print(f"Chunk {i}: {metadata.chunk\_id}, rows: {len(chunk\_df)}, overlap: {metadata.overlap\_rows}")

# Test recursive chunking

print("\n--- Recursive Chunking Test ---")

recursive\_chunker = RecursiveChunker(max\_chunk\_size=4000, min\_chunk\_size=800)

recursive\_chunks = recursive\_chunker.chunk(df, {'method': 'recursive', 'params': {'max\_chunk\_size': 4000, 'min\_chunk\_size': 800}}, ['category'])

print(f"Generated {len(recursive\_chunks)} recursive chunks")

for i, (chunk\_df, metadata) in enumerate(recursive\_chunks[:3]): # Show first 3 chunks

print(f"Chunk {i}: {metadata.chunk\_id}, rows: {len(chunk\_df)}, entity: {metadata.entity\_group}")

# Validate quality

print("\n--- Quality Validation ---")

validator = QualityValidator(df)

fixed\_quality = validator.validate\_chunks(fixed\_chunks)

print(f"Fixed chunking quality: Schema={fixed\_quality['schema\_consistency']}, Completeness={fixed\_quality['data\_completeness']}")

recursive\_quality = validator.validate\_chunks(recursive\_chunks)

print(f"Recursive chunking quality: Schema={recursive\_quality['schema\_consistency']}, Completeness={recursive\_quality['data\_completeness']}")

return df, fixed\_chunks, recursive\_chunks, analysis

python

# Cell 9: Run the demo

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

# This will run when executed as a script

df, fixed\_chunks, recursive\_chunks, analysis = demo\_chunking()

# Display some results

print(f"\nOriginal DataFrame shape: {df.shape}")

print(f"Fixed chunks created: {len(fixed\_chunks)}")

print(f"Recursive chunks created: {len(recursive\_chunks)}")

# Show first chunk details

if fixed\_chunks:

chunk\_df, metadata = fixed\_chunks[0]

print(f"\nFirst fixed chunk: {metadata.chunk\_id}")

print(f"Rows: {len(chunk\_df)}, Columns: {len(chunk\_df.columns)}")

print(f"Overlap: {metadata.overlap\_rows} rows")

print(f"Data types in chunk:\n{chunk\_df.dtypes}")

if recursive\_chunks:

chunk\_df, metadata = recursive\_chunks[0]

print(f"\nFirst recursive chunk: {metadata.chunk\_id}")

print(f"Rows: {len(chunk\_df)}, Columns: {len(chunk\_df.columns)}")

print(f"Entity group: {metadata.entity\_group}")

print(f"Categories in chunk: {chunk\_df['category'].unique()}")

This enhanced synthetic dataset includes:

1. **Numeric data**: product\_id, price, quantity, weight\_kg, customer\_rating
2. **Text data**: product\_name, description
3. **Categorical data**: category, subcategory
4. **Datetime data**: created\_date, last\_updated
5. **Boolean data**: in\_stock, on\_sale
6. **Realistic null values** in some numeric columns

The dataset represents a product catalog with various attributes that would be typical in e-commerce or retail systems. You can run this in a Jupyter Notebook by executing each cell in order.