# Cell 1: Imports and Setup

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

import hashlib

import logging

import matplotlib.pyplot as plt

import seaborn as sns

from IPython.display import display, HTML

import warnings

warnings.filterwarnings('ignore')

# Configure logging

logging.basicConfig(level=logging.INFO)

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

print("✅ All imports successful!")

# Cell 2: Data Classes and Metadata

@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]

def to\_dict(self):

"""Convert to dictionary for easy display"""

return {

'Chunk ID': self.chunk\_id,

'Method': self.method,

'Start Row': self.start\_row,

'End Row': self.end\_row,

'Total Rows': self.total\_rows,

'Overlap Rows': self.overlap\_rows,

'Entity Group': self.entity\_group or 'N/A',

'Created': self.created\_at

}

print("✅ Data classes defined!")

# Cell 3: Schema Analyzer 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

}

def display\_analysis(self):

"""Display comprehensive analysis in Jupyter"""

schema = self.analyze\_schema()

data\_types = self.detect\_data\_types()

relationships = self.analyze\_column\_relationships()

print("📊 SCHEMA ANALYSIS REPORT")

print("=" \* 50)

print(f"📏 Dimensions: {schema['total\_rows']:,} rows × {schema['total\_columns']} columns")

print(f"💾 Memory Usage: {schema['memory\_usage'] / 1024 / 1024:.1f} MB")

print(f"🔄 Duplicate Rows: {schema['duplicate\_rows']:,}")

print()

print("📈 COLUMN TYPES:")

print(f" 🔢 Numeric: {len(data\_types['numeric'])} columns")

if data\_types['numeric']:

print(f" {', '.join(data\_types['numeric'][:5])}{'...' if len(data\_types['numeric']) > 5 else ''}")

print(f" 📝 Text: {len(data\_types['text'])} columns")

if data\_types['text']:

print(f" {', '.join(data\_types['text'][:5])}{'...' if len(data\_types['text']) > 5 else ''}")

print(f" 📅 DateTime: {len(data\_types['datetime'])} columns")

if data\_types['datetime']:

print(f" {', '.join(data\_types['datetime'])}")

print()

print("🔑 PRIMARY KEY CANDIDATES:")

if relationships['primary\_keys']:

for pk in relationships['primary\_keys']:

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

print(f" • {pk} (Uniqueness: {unique\_ratio:.1%})")

else:

print(" ⚠️ No clear primary keys detected")

print()

# Display sample data

print("👀 SAMPLE DATA:")

display(self.df.head())

print("✅ SchemaAnalyzer class defined!")

# Cell 4: Fixed Chunker 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 = []

print(f"🔄 Starting fixed chunking...")

print(f" Chunk size: {self.chunk\_size:,} rows")

print(f" Overlap: {self.overlap\_size:,} rows ({self.overlap\_ratio:.1%})")

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))

print(f" ✓ Created chunk {chunk\_id}: {len(chunk\_df):,} rows")

if end\_idx >= len(df):

break

print(f"✅ Fixed chunking complete: {len(chunks)} chunks created")

return chunks

print("✅ FixedChunker class defined!")

# Cell 5: Recursive Chunker 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 = []

print(f"🔄 Starting recursive chunking...")

print(f" Max chunk size: {self.max\_chunk\_size:,} rows")

print(f" Primary keys: {primary\_keys if primary\_keys else 'None detected'}")

if not primary\_keys:

print(" ⚠️ Falling back to simple row-based chunking")

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

print(f" 📊 Processing {len(groups)} entity groups...")

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, "mixed\_entities"))

print(f" ✓ Created mixed entity chunk: {len(chunk\_df):,} rows")

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)

print(f" ✓ Split large entity '{name}': {len(sub\_chunks)} 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"))

print(f" ✓ Created mixed entity chunk: {len(chunk\_df):,} rows")

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"))

print(f" ✓ Created final chunk: {len(chunk\_df):,} rows")

print(f"✅ Recursive chunking complete: {len(chunks)} chunks created")

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

print("✅ RecursiveChunker class defined!")

# Cell 6: Quality Validator 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

print("🔍 Validating chunk quality...")

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 (accounting for overlaps in fixed chunking)

expected\_total = self.original\_row\_count

if total\_rows < expected\_total:

results['data\_completeness'] = False

results['issues'].append(f"Row count mismatch: {total\_rows} vs {expected\_total}")

# Display results

print("📋 VALIDATION RESULTS:")

print(f" ✅ Schema Consistency: {'PASS' if results['schema\_consistency'] else 'FAIL'}")

print(f" ✅ Data Completeness: {'PASS' if results['data\_completeness'] else 'FAIL'}")

print(f" ✅ Boundary Validation: {'PASS' if results['boundary\_validation'] else 'FAIL'}")

print(f" 📊 Rows Preserved: {results['total\_rows\_preserved']:,}")

if results['issues']:

print(" ⚠️ Issues Found:")

for issue in results['issues']:

print(f" • {issue}")

else:

print(" 🎉 No issues found!")

return results

def display\_chunk\_statistics(self, chunks: List[Tuple[pd.DataFrame, ChunkMetadata]]):

"""Display detailed chunk statistics"""

chunk\_data = []

for chunk\_df, metadata in chunks:

chunk\_data.append({

'Chunk ID': metadata.chunk\_id,

'Method': metadata.method,

'Rows': metadata.total\_rows,

'Overlap': metadata.overlap\_rows,

'Entity Group': metadata.entity\_group or 'N/A',

'Memory (MB)': chunk\_df.memory\_usage(deep=True).sum() / 1024 / 1024

})

stats\_df = pd.DataFrame(chunk\_data)

print("📊 CHUNK STATISTICS:")

print(f" Total Chunks: {len(chunks)}")

print(f" Average Rows per Chunk: {stats\_df['Rows'].mean():,.0f}")

print(f" Min Rows: {stats\_df['Rows'].min():,}")

print(f" Max Rows: {stats\_df['Rows'].max():,}")

print(f" Total Memory: {stats\_df['Memory (MB)'].sum():.1f} MB")

print()

display(stats\_df)

# Visualize chunk sizes

plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)

plt.bar(range(len(stats\_df)), stats\_df['Rows'])

plt.title('Chunk Sizes (Rows)')

plt.xlabel('Chunk Index')

plt.ylabel('Number of Rows')

plt.xticks(range(0, len(stats\_df), max(1, len(stats\_df)//10)))

plt.subplot(1, 2, 2)

plt.hist(stats\_df['Rows'], bins=min(20, len(stats\_df)), alpha=0.7, edgecolor='black')

plt.title('Distribution of Chunk Sizes')

plt.xlabel('Number of Rows')

plt.ylabel('Frequency')

plt.tight\_layout()

plt.show()

print("✅ QualityValidator class defined!")

# Cell 7: Utility Functions

def load\_sample\_data(rows=50000, cols=10, with\_entities=True):

"""Generate sample data for testing"""

print(f"🎲 Generating sample data: {rows:,} rows × {cols} columns")

np.random.seed(42) # For reproducible results

data = {}

# Add ID column (potential primary key)

data['id'] = range(1, rows + 1)

# Add entity grouping column if requested

if with\_entities:

num\_entities = max(10, rows // 5000) # Aim for ~5000 rows per entity on average

data['entity\_id'] = np.random.choice(range(1, num\_entities + 1), rows)

data['entity\_name'] = [f"Entity\_{eid:03d}" for eid in data['entity\_id']]

# Add various data types

for i in range(cols - (3 if with\_entities else 1)):

if i % 3 == 0: # Numeric columns

data[f'numeric\_{i}'] = np.random.normal(100, 15, rows).round(2)

elif i % 3 == 1: # Text columns

categories = [f'Category\_{j}' for j in range(1, 11)]

data[f'category\_{i}'] = np.random.choice(categories, rows)

else: # More numeric

data[f'value\_{i}'] = np.random.exponential(50, rows).round(2)

# Add some datetime data

base\_date = pd.Timestamp('2023-01-01')

data['created\_at'] = [base\_date + pd.Timedelta(days=np.random.randint(0, 365)) for \_ in range(rows)]

df = pd.DataFrame(data)

print(f"✅ Sample data generated!")

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

print(f" Memory: {df.memory\_usage(deep=True).sum() / 1024 / 1024:.1f} MB")

return df

def compare\_chunking\_methods(df, fixed\_params=None, recursive\_params=None):

"""Compare both chunking methods side by side"""

print("🆚 COMPARING CHUNKING METHODS")

print("=" \* 50)

# Default parameters

fixed\_params = fixed\_params or {'chunk\_size': 10000, 'overlap\_ratio': 0.1}

recursive\_params = recursive\_params or {'max\_chunk\_size': 50000, 'min\_chunk\_size': 1000}

# Analyze schema first

analyzer = SchemaAnalyzer(df)

relationships = analyzer.analyze\_column\_relationships()

# Test Fixed Chunking

print("\n🔧 TESTING FIXED CHUNKING:")

fixed\_chunker = FixedChunker(\*\*fixed\_params)

fixed\_chunks = fixed\_chunker.chunk(df, {'method': 'fixed', 'params': fixed\_params})

fixed\_validator = QualityValidator(df)

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

# Test Recursive Chunking

print("\n🌳 TESTING RECURSIVE CHUNKING:")

recursive\_chunker = RecursiveChunker(\*\*recursive\_params)

recursive\_chunks = recursive\_chunker.chunk(df, {'method': 'recursive', 'params': recursive\_params}, relationships['primary\_keys'])

recursive\_validator = QualityValidator(df)

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

# Comparison summary

print("\n📊 COMPARISON SUMMARY:")

print("=" \* 50)

comparison\_data = {

'Metric': [

'Number of Chunks',

'Avg Rows per Chunk',

'Min Chunk Size',

'Max Chunk Size',

'Schema Consistency',

'Data Completeness',

'Total Issues'

],

'Fixed Chunking': [

len(fixed\_chunks),

f"{np.mean([len(chunk[0]) for chunk in fixed\_chunks]):,.0f}",

f"{min(len(chunk[0]) for chunk in fixed\_chunks):,}",

f"{max(len(chunk[0]) for chunk in fixed\_chunks):,}",

'✅ PASS' if fixed\_quality['schema\_consistency'] else '❌ FAIL',

'✅ PASS' if fixed\_quality['data\_completeness'] else '❌ FAIL',

len(fixed\_quality['issues'])

],

'Recursive Chunking': [

len(recursive\_chunks),

f"{np.mean([len(chunk[0]) for chunk in recursive\_chunks]):,.0f}",

f"{min(len(chunk[0]) for chunk in recursive\_chunks):,}",

f"{max(len(chunk[0]) for chunk in recursive\_chunks):,}",

'✅ PASS' if recursive\_quality['schema\_consistency'] else '❌ FAIL',

'✅ PASS' if recursive\_quality['data\_completeness'] else '❌ FAIL',

len(recursive\_quality['issues'])

]

}

comparison\_df = pd.DataFrame(comparison\_data)

display(comparison\_df)

return {

'fixed': {'chunks': fixed\_chunks, 'quality': fixed\_quality},

'recursive': {'chunks': recursive\_chunks, 'quality': recursive\_quality}

}

print("✅ Utility functions defined!")

# Cell 8: Load and Analyze Your Data

# Replace this with your actual data loading

# df = pd.read\_csv('your\_data.csv')

# For demonstration, let's use sample data

df = load\_sample\_data(rows=25000, cols=8, with\_entities=True)

# Analyze the data

analyzer = SchemaAnalyzer(df)

analyzer.display\_analysis()

# Cell 9: Run Fixed Chunking

print("\n" + "="\*60)

print("🔧 FIXED CHUNKING DEMONSTRATION")

print("="\*60)

# Configure fixed chunking parameters

CHUNK\_SIZE = 5000

OVERLAP\_RATIO = 0.1

fixed\_chunker = FixedChunker(chunk\_size=CHUNK\_SIZE, overlap\_ratio=OVERLAP\_RATIO)

fixed\_chunks = fixed\_chunker.chunk(df, {

'method': 'fixed',

'params': {'chunk\_size': CHUNK\_SIZE, 'overlap\_ratio': OVERLAP\_RATIO},

'original\_shape': df.shape

})

# Validate and display results

fixed\_validator = QualityValidator(df)

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

fixed\_validator.display\_chunk\_statistics(fixed\_chunks)

# Cell 10: Run Recursive Chunking

print("\n" + "="\*60)

print("🌳 RECURSIVE CHUNKING DEMONSTRATION")

print("="\*60)

# Get primary keys from analysis

analyzer = SchemaAnalyzer(df)

relationships = analyzer.analyze\_column\_relationships()

primary\_keys = relationships['primary\_keys']

# Configure recursive chunking parameters

MAX\_CHUNK\_SIZE = 15000

MIN\_CHUNK\_SIZE = 1000

recursive\_chunker = RecursiveChunker(max\_chunk\_size=MAX\_CHUNK\_SIZE, min\_chunk\_size=MIN\_CHUNK\_SIZE)

recursive\_chunks = recursive\_chunker.chunk(df, {

'method': 'recursive',

'params': {'max\_chunk\_size': MAX\_CHUNK\_SIZE, 'min\_chunk\_size': MIN\_CHUNK\_SIZE},

'original\_shape': df.shape

}, primary\_keys)

# Validate and display results

recursive\_validator = QualityValidator(df)

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

recursive\_validator.display\_chunk\_statistics(recursive\_chunks)

# Cell 11: Compare Methods and Export Results

print("\n" + "="\*60)

print("🆚 FINAL COMPARISON & EXPORT")

print("="\*60)

# Side-by-side comparison

results = compare\_chunking\_methods(df,

fixed\_params={'chunk\_size': CHUNK\_SIZE, 'overlap\_ratio': OVERLAP\_RATIO},

recursive\_params={'max\_chunk\_size': MAX\_CHUNK\_SIZE, 'min\_chunk\_size': MIN\_CHUNK\_SIZE}

)

# Export chunks if needed

def export\_chunks(chunks, method\_name):

"""Export chunks to individual CSV files"""

print(f"\n💾 Exporting {method\_name} chunks...")

for i, (chunk\_df, metadata) in enumerate(chunks):

filename = f"{method\_name}\_chunk\_{metadata.chunk\_id}.csv"

chunk\_df.to\_csv(filename, index=False)

if i < 3: # Show first 3 files

print(f" ✓ Exported {filename} ({len(chunk\_df):,} rows)")

elif i == 3 and len(chunks) > 3:

print(f" ... and {len(chunks) - 3} more files")

break

# Uncomment to export chunks

# export\_chunks(fixed\_chunks, "fixed")

# export\_chunks(recursive\_chunks, "recursive")

print("\n🎉 Analysis complete! Choose your preferred chunking method based on the comparison above.")

# Cell 12: Interactive Chunk Exploration

def explore\_chunk(chunks, chunk\_index=0):

"""Explore a specific chunk interactively"""

if chunk\_index >= len(chunks):

print(f"❌ Chunk index {chunk\_index} out of range. Available: 0-{len(chunks)-1}")

return

chunk\_df, metadata = chunks[chunk\_index]

print(f"🔍 EXPLORING CHUNK: {metadata.chunk\_id}")

print("=" \* 50)

# Metadata

meta\_dict = metadata.to\_dict()

for key, value in meta\_dict.items():

print(f" {key}: {value}")

print(f"\n📊 Data Preview:")

display(chunk\_df.head())

print(f"\n📈 Statistics:")

display(chunk\_df.describe())

return chunk\_df, metadata

# Example: Explore first chunk from each method

print("🔧 EXPLORING FIXED CHUNKS:")

fixed\_sample = explore\_chunk(fixed\_chunks, 0)

print("\n🌳 EXPLORING RECURSIVE CHUNKS:")

recursive\_sample = explore\_chunk(recursive\_chunks, 0)

print("\n✅ Jupyter notebook conversion complete! Run cells sequentially for full demonstration.")

# Cell 13: Advanced Analysis - Chunk Distribution Visualization

def visualize\_chunk\_analysis(fixed\_chunks, recursive\_chunks):

"""Create comprehensive visualizations comparing both methods"""

fig, axes = plt.subplots(2, 3, figsize=(18, 12))

fig.suptitle('CSV Chunking Methods Comparison', fontsize=16, fontweight='bold')

# Extract data for visualization

fixed\_sizes = [len(chunk[0]) for chunk in fixed\_chunks]

recursive\_sizes = [len(chunk[0]) for chunk in recursive\_chunks]

fixed\_overlaps = [chunk[1].overlap\_rows for chunk in fixed\_chunks]

# 1. Chunk Size Distribution Comparison

axes[0,0].hist(fixed\_sizes, alpha=0.7, label='Fixed', bins=20, color='skyblue', edgecolor='black')

axes[0,0].hist(recursive\_sizes, alpha=0.7, label='Recursive', bins=20, color='lightcoral', edgecolor='black')

axes[0,0].set\_title('Chunk Size Distribution')

axes[0,0].set\_xlabel('Chunk Size (rows)')

axes[0,0].set\_ylabel('Frequency')

axes[0,0].legend()

axes[0,0].grid(True, alpha=0.3)

# 2. Chunk Sizes Over Index

axes[0,1].plot(range(len(fixed\_sizes)), fixed\_sizes, 'o-', label='Fixed', alpha=0.7, color='blue')

axes[0,1].plot(range(len(recursive\_sizes)), recursive\_sizes, 's-', label='Recursive', alpha=0.7, color='red')

axes[0,1].set\_title('Chunk Sizes by Index')

axes[0,1].set\_xlabel('Chunk Index')

axes[0,1].set\_ylabel('Chunk Size (rows)')

axes[0,1].legend()

axes[0,1].grid(True, alpha=0.3)

# 3. Box Plot Comparison

axes[0,2].boxplot([fixed\_sizes, recursive\_sizes], labels=['Fixed', 'Recursive'])

axes[0,2].set\_title('Chunk Size Distribution (Box Plot)')

axes[0,2].set\_ylabel('Chunk Size (rows)')

axes[0,2].grid(True, alpha=0.3)

# 4. Overlap Analysis (Fixed method only)

axes[1,0].bar(range(len(fixed\_overlaps)), fixed\_overlaps, color='orange', alpha=0.7)

axes[1,0].set\_title('Overlap Rows in Fixed Chunking')

axes[1,0].set\_xlabel('Chunk Index')

axes[1,0].set\_ylabel('Overlap Rows')

axes[1,0].grid(True, alpha=0.3)

# 5. Memory Usage Comparison

fixed\_memory = [chunk[0].memory\_usage(deep=True).sum() / 1024 / 1024 for chunk in fixed\_chunks]

recursive\_memory = [chunk[0].memory\_usage(deep=True).sum() / 1024 / 1024 for chunk in recursive\_chunks]

x\_pos = np.arange(2)

avg\_memory = [np.mean(fixed\_memory), np.mean(recursive\_memory)]

bars = axes[1,1].bar(x\_pos, avg\_memory, color=['skyblue', 'lightcoral'], alpha=0.7)

axes[1,1].set\_title('Average Memory Usage per Chunk')

axes[1,1].set\_xlabel('Method')

axes[1,1].set\_ylabel('Memory (MB)')

axes[1,1].set\_xticks(x\_pos)

axes[1,1].set\_xticklabels(['Fixed', 'Recursive'])

axes[1,1].grid(True, alpha=0.3)

# Add value labels on bars

for bar, value in zip(bars, avg\_memory):

height = bar.get\_height()

axes[1,1].text(bar.get\_x() + bar.get\_width()/2., height + 0.01,

f'{value:.2f}', ha='center', va='bottom')

# 6. Entity Distribution in Recursive Chunks

if any(chunk[1].entity\_group and chunk[1].entity\_group != "mixed\_entities" for chunk in recursive\_chunks):

entity\_groups = [chunk[1].entity\_group for chunk in recursive\_chunks]

entity\_counts = {}

for group in entity\_groups:

if group and group != "mixed\_entities" and "no\_entities" not in group:

entity\_counts[group] = entity\_counts.get(group, 0) + 1

if entity\_counts:

axes[1,2].pie(entity\_counts.values(), labels=entity\_counts.keys(), autopct='%1.1f%%')

axes[1,2].set\_title('Entity Distribution in Recursive Chunks')

else:

axes[1,2].text(0.5, 0.5, 'No Clear Entity\nDistribution', ha='center', va='center', transform=axes[1,2].transAxes)

axes[1,2].set\_title('Entity Distribution')

else:

axes[1,2].text(0.5, 0.5, 'No Entity Groups\nDetected', ha='center', va='center', transform=axes[1,2].transAxes)

axes[1,2].set\_title('Entity Distribution')

plt.tight\_layout()

plt.show()

# Summary Statistics Table

summary\_data = {

'Metric': [

'Total Chunks',

'Average Chunk Size',

'Standard Deviation',

'Min Chunk Size',

'Max Chunk Size',

'Total Memory (MB)',

'Average Memory per Chunk (MB)'

],

'Fixed Method': [

len(fixed\_chunks),

f"{np.mean(fixed\_sizes):,.0f}",

f"{np.std(fixed\_sizes):,.0f}",

f"{min(fixed\_sizes):,}",

f"{max(fixed\_sizes):,}",

f"{sum(fixed\_memory):.2f}",

f"{np.mean(fixed\_memory):.2f}"

],

'Recursive Method': [

len(recursive\_chunks),

f"{np.mean(recursive\_sizes):,.0f}",

f"{np.std(recursive\_sizes):,.0f}",

f"{min(recursive\_sizes):,}",

f"{max(recursive\_sizes):,}",

f"{sum(recursive\_memory):.2f}",

f"{np.mean(recursive\_memory):.2f}"

]

}

print("\n📊 DETAILED COMPARISON STATISTICS:")

print("=" \* 80)

summary\_df = pd.DataFrame(summary\_data)

display(summary\_df)

# Run the advanced visualization

visualize\_chunk\_analysis(fixed\_chunks, recursive\_chunks)

# Cell 14: Performance Benchmarking

import time

from memory\_profiler import profile

def benchmark\_chunking\_performance(df, iterations=3):

"""Benchmark the performance of both chunking methods"""

print("⏱️ PERFORMANCE BENCHMARKING")

print("=" \* 50)

print(f"Running {iterations} iterations for each method...")

# Fixed Chunking Benchmark

fixed\_times = []

fixed\_chunker = FixedChunker(chunk\_size=5000, overlap\_ratio=0.1)

for i in range(iterations):

start\_time = time.time()

fixed\_chunks\_bench = fixed\_chunker.chunk(df, {'method': 'fixed'})

end\_time = time.time()

fixed\_times.append(end\_time - start\_time)

print(f" Fixed iteration {i+1}: {end\_time - start\_time:.3f}s")

# Recursive Chunking Benchmark

analyzer = SchemaAnalyzer(df)

relationships = analyzer.analyze\_column\_relationships()

primary\_keys = relationships['primary\_keys']

recursive\_times = []

recursive\_chunker = RecursiveChunker(max\_chunk\_size=15000, min\_chunk\_size=1000)

for i in range(iterations):

start\_time = time.time()

recursive\_chunks\_bench = recursive\_chunker.chunk(df, {'method': 'recursive'}, primary\_keys)

end\_time = time.time()

recursive\_times.append(end\_time - start\_time)

print(f" Recursive iteration {i+1}: {end\_time - start\_time:.3f}s")

# Performance Summary

print(f"\n📈 PERFORMANCE RESULTS:")

print(f" Fixed Chunking:")

print(f" Average Time: {np.mean(fixed\_times):.3f}s ± {np.std(fixed\_times):.3f}s")

print(f" Best Time: {min(fixed\_times):.3f}s")

print(f" Recursive Chunking:")

print(f" Average Time: {np.mean(recursive\_times):.3f}s ± {np.std(recursive\_times):.3f}s")

print(f" Best Time: {min(recursive\_times):.3f}s")

speedup = np.mean(recursive\_times) / np.mean(fixed\_times)

if speedup > 1:

print(f" 🏆 Fixed chunking is {speedup:.2f}x faster than recursive")

else:

print(f" 🏆 Recursive chunking is {1/speedup:.2f}x faster than fixed")

# Visualization

plt.figure(figsize=(10, 6))

plt.subplot(1, 2, 1)

plt.bar(['Fixed', 'Recursive'], [np.mean(fixed\_times), np.mean(recursive\_times)],

color=['skyblue', 'lightcoral'], alpha=0.7)

plt.title('Average Execution Time')

plt.ylabel('Time (seconds)')

plt.grid(True, alpha=0.3)

plt.subplot(1, 2, 2)

plt.boxplot([fixed\_times, recursive\_times], labels=['Fixed', 'Recursive'])

plt.title('Execution Time Distribution')

plt.ylabel('Time (seconds)')

plt.grid(True, alpha=0.3)

plt.tight\_layout()

plt.show()

# Run performance benchmark (comment out if memory\_profiler not installed)

try:

benchmark\_chunking\_performance(df)

except ImportError:

print("⚠️ memory\_profiler not installed. Skipping detailed benchmarking.")

print(" Install with: pip install memory-profiler")

# Simple timing benchmark

print("\n⏱️ SIMPLE TIMING BENCHMARK")

print("=" \* 40)

# Time fixed chunking

start = time.time()

fixed\_chunker = FixedChunker(chunk\_size=5000, overlap\_ratio=0.1)

fixed\_result = fixed\_chunker.chunk(df, {'method': 'fixed'})

fixed\_time = time.time() - start

# Time recursive chunking

start = time.time()

analyzer = SchemaAnalyzer(df)

relationships = analyzer.analyze\_column\_relationships()

recursive\_chunker = RecursiveChunker(max\_chunk\_size=15000, min\_chunk\_size=1000)

recursive\_result = recursive\_chunker.chunk(df, {'method': 'recursive'}, relationships['primary\_keys'])

recursive\_time = time.time() - start

print(f"Fixed Chunking: {fixed\_time:.3f}s")

print(f"Recursive Chunking: {recursive\_time:.3f}s")

# Cell 15: Real-world Data Testing Helper

def test\_with\_real\_data(file\_path, sample\_size=None):

"""Helper function to test chunking with your real CSV data"""

print(f"📁 LOADING REAL DATA: {file\_path}")

print("=" \* 60)

try:

# Load data with optional sampling for large files

if sample\_size:

print(f" Sampling {sample\_size:,} rows for testing...")

df = pd.read\_csv(file\_path, nrows=sample\_size)

else:

df = pd.read\_csv(file\_path)

print(f" ✅ Data loaded: {len(df):,} rows × {len(df.columns)} columns")

print(f" 📊 Memory usage: {df.memory\_usage(deep=True).sum() / 1024 / 1024:.1f} MB")

# Analyze the data

print(f"\n🔍 ANALYZING YOUR DATA:")

analyzer = SchemaAnalyzer(df)

analyzer.display\_analysis()

# Get recommendations

print(f"\n💡 CHUNKING RECOMMENDATIONS:")

memory\_mb = df.memory\_usage(deep=True).sum() / 1024 / 1024

if memory\_mb < 100:

print(f" 🟢 Small dataset ({memory\_mb:.1f}MB) - Consider chunk sizes: 5K-10K rows")

rec\_fixed\_size = 5000

rec\_recursive\_max = 15000

elif memory\_mb < 500:

print(f" 🟡 Medium dataset ({memory\_mb:.1f}MB) - Consider chunk sizes: 10K-25K rows")

rec\_fixed\_size = 10000

rec\_recursive\_max = 25000

else:

print(f" 🔴 Large dataset ({memory\_mb:.1f}MB) - Consider chunk sizes: 25K-50K rows")

rec\_fixed\_size = 25000

rec\_recursive\_max = 50000

# Check for entity structure

relationships = analyzer.analyze\_column\_relationships()

if relationships['primary\_keys']:

print(f" 🎯 Entity structure detected - Recursive chunking recommended")

print(f" 🔑 Primary keys: {', '.join(relationships['primary\_keys'])}")

else:

print(f" 📏 No clear entities - Fixed chunking may be simpler")

# Run both methods with recommendations

print(f"\n🧪 TESTING BOTH METHODS WITH RECOMMENDED SETTINGS:")

# Fixed chunking test

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

start\_time = time.time()

fixed\_chunks = fixed\_chunker.chunk(df, {'method': 'fixed'})

fixed\_time = time.time() - start\_time

# Recursive chunking test

recursive\_chunker = RecursiveChunker(max\_chunk\_size=rec\_recursive\_max, min\_chunk\_size=1000)

start\_time = time.time()

recursive\_chunks = recursive\_chunker.chunk(df, {'method': 'recursive'}, relationships['primary\_keys'])

recursive\_time = time.time() - start\_time

# Compare results

print(f"\n📊 RESULTS SUMMARY:")

results\_data = {

'Method': ['Fixed', 'Recursive'],

'Chunks Created': [len(fixed\_chunks), len(recursive\_chunks)],

'Avg Chunk Size': [

f"{np.mean([len(chunk[0]) for chunk in fixed\_chunks]):,.0f}",

f"{np.mean([len(chunk[0]) for chunk in recursive\_chunks]):,.0f}"

],

'Processing Time': [f"{fixed\_time:.2f}s", f"{recursive\_time:.2f}s"],

'Memory per Chunk': [

f"{np.mean([chunk[0].memory\_usage(deep=True).sum() / 1024 / 1024 for chunk in fixed\_chunks]):.1f} MB",

f"{np.mean([chunk[0].memory\_usage(deep=True).sum() / 1024 / 1024 for chunk in recursive\_chunks]):.1f} MB"

]

}

results\_df = pd.DataFrame(results\_data)

display(results\_df)

return df, fixed\_chunks, recursive\_chunks

except Exception as e:

print(f" ❌ Error loading data: {str(e)}")

print(f" 💡 Make sure the file path is correct and the file is a valid CSV")

return None, None, None

# Example usage (uncomment and modify for your data):

# df\_real, fixed\_real, recursive\_real = test\_with\_real\_data('path/to/your/data.csv', sample\_size=50000)

print("📝 To test with your real data, use:")

print(" test\_with\_real\_data('path/to/your/file.csv')")

# Cell 16: Export and Summary Functions

def export\_analysis\_report(df, fixed\_chunks, recursive\_chunks, filename="chunking\_analysis\_report"):

"""Export comprehensive analysis report"""

report\_sections = []

# Header

report\_sections.append("# CSV Chunking Analysis Report")

report\_sections.append(f"Generated on: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")

report\_sections.append("=" \* 50)

# Dataset Overview

report\_sections.append("\n## Dataset Overview")

report\_sections.append(f"- \*\*Shape\*\*: {df.shape[0]:,} rows × {df.shape[1]} columns")

report\_sections.append(f"- \*\*Memory Usage\*\*: {df.memory\_usage(deep=True).sum() / 1024 / 1024:.1f} MB")

report\_sections.append(f"- \*\*Data Types\*\*: {dict(df.dtypes.astype(str))}")

# Schema Analysis

analyzer = SchemaAnalyzer(df)

schema = analyzer.analyze\_schema()

relationships = analyzer.analyze\_column\_relationships()

report\_sections.append(f"\n## Schema Analysis")

report\_sections.append(f"- \*\*Duplicate Rows\*\*: {schema['duplicate\_rows']:,}")

report\_sections.append(f"- \*\*Primary Keys\*\*: {relationships['primary\_keys'] if relationships['primary\_keys'] else 'None detected'}")

# Fixed Chunking Results

fixed\_sizes = [len(chunk[0]) for chunk in fixed\_chunks]

report\_sections.append(f"\n## Fixed Chunking Results")

report\_sections.append(f"- \*\*Total Chunks\*\*: {len(fixed\_chunks)}")

report\_sections.append(f"- \*\*Average Chunk Size\*\*: {np.mean(fixed\_sizes):,.0f} rows")

report\_sections.append(f"- \*\*Size Range\*\*: {min(fixed\_sizes):,} - {max(fixed\_sizes):,} rows")

report\_sections.append(f"- \*\*Standard Deviation\*\*: {np.std(fixed\_sizes):,.0f} rows")

# Recursive Chunking Results

recursive\_sizes = [len(chunk[0]) for chunk in recursive\_chunks]

report\_sections.append(f"\n## Recursive Chunking Results")

report\_sections.append(f"- \*\*Total Chunks\*\*: {len(recursive\_chunks)}")

report\_sections.append(f"- \*\*Average Chunk Size\*\*: {np.mean(recursive\_sizes):,.0f} rows")

report\_sections.append(f"- \*\*Size Range\*\*: {min(recursive\_sizes):,} - {max(recursive\_sizes):,} rows")

report\_sections.append(f"- \*\*Standard Deviation\*\*: {np.std(recursive\_sizes):,.0f} rows")

# Entity Analysis for Recursive

entity\_groups = [chunk[1].entity\_group for chunk in recursive\_chunks if chunk[1].entity\_group]

if entity\_groups:

unique\_entities = len(set(entity\_groups))

report\_sections.append(f"- \*\*Entity Groups\*\*: {unique\_entities} unique groups")

# Recommendations

report\_sections.append(f"\n## Recommendations")

if relationships['primary\_keys']:

report\_sections.append("- ✅ \*\*Recursive chunking recommended\*\* - Clear entity structure detected")

report\_sections.append(f"- 🔑 Utilize primary keys: {', '.join(relationships['primary\_keys'])}")

else:

report\_sections.append("- ✅ \*\*Fixed chunking recommended\*\* - No clear entity structure")

# Performance comparison

size\_variance\_fixed = np.std(fixed\_sizes) / np.mean(fixed\_sizes)

size\_variance\_recursive = np.std(recursive\_sizes) / np.mean(recursive\_sizes)

if size\_variance\_fixed < size\_variance\_recursive:

report\_sections.append("- 📊 \*\*Fixed chunking\*\* provides more consistent chunk sizes")

else:

report\_sections.append("- 📊 \*\*Recursive chunking\*\* provides better entity preservation")

# Write report to file

report\_content = '\n'.join(report\_sections)

with open(f"{filename}.md", 'w') as f:

f.write(report\_content)

print(f"📄 Analysis report exported to: {filename}.md")

# Also create a JSON summary for programmatic use

json\_summary = {

'dataset': {

'rows': int(df.shape[0]),

'columns': int(df.shape[1]),

'memory\_mb': float(df.memory\_usage(deep=True).sum() / 1024 / 1024),

'primary\_keys': relationships['primary\_keys']

},

'fixed\_chunking': {

'total\_chunks': len(fixed\_chunks),

'avg\_chunk\_size': float(np.mean(fixed\_sizes)),

'std\_chunk\_size': float(np.std(fixed\_sizes)),

'min\_chunk\_size': int(min(fixed\_sizes)),

'max\_chunk\_size': int(max(fixed\_sizes))

},

'recursive\_chunking': {

'total\_chunks': len(recursive\_chunks),

'avg\_chunk\_size': float(np.mean(recursive\_sizes)),

'std\_chunk\_size': float(np.std(recursive\_sizes)),

'min\_chunk\_size': int(min(recursive\_sizes)),

'max\_chunk\_size': int(max(recursive\_sizes))

},

'generated\_at': datetime.now().isoformat()

}

with open(f"{filename}.json", 'w') as f:

json.dump(json\_summary, f, indent=2)

print(f"📊 JSON summary exported to: {filename}.json")

return report\_content, json\_summary

def save\_chunks\_to\_files(chunks, method\_name, output\_dir="chunks\_output"):

"""Save all chunks to individual CSV files with detailed metadata"""

import os

# Create output directory

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

print(f"💾 Saving {len(chunks)} {method\_name} chunks to {output\_dir}/")

metadata\_list = []

for i, (chunk\_df, metadata) in enumerate(chunks):

# Save chunk data

chunk\_filename = f"{method\_name}\_chunk\_{metadata.chunk\_id}.csv"

chunk\_path = os.path.join(output\_dir, chunk\_filename)

chunk\_df.to\_csv(chunk\_path, index=False)

# Collect metadata

metadata\_dict = metadata.to\_dict()

metadata\_dict['filename'] = chunk\_filename

metadata\_dict['file\_size\_mb'] = os.path.getsize(chunk\_path) / 1024 / 1024

metadata\_list.append(metadata\_dict)

if i < 5: # Show first 5 files

print(f" ✓ {chunk\_filename} ({len(chunk\_df):,} rows)")

elif i == 5 and len(chunks) > 5:

print(f" ... and {len(chunks) - 5} more files")

# Save metadata

metadata\_df = pd.DataFrame(metadata\_list)

metadata\_filename = f"{method\_name}\_chunks\_metadata.csv"

metadata\_path = os.path.join(output\_dir, metadata\_filename)

metadata\_df.to\_csv(metadata\_path, index=False)

print(f" ✓ {metadata\_filename} (metadata file)")

print(f"✅ All {method\_name} chunks saved successfully!")

return metadata\_df

# Export final results (uncomment to save files)

# export\_analysis\_report(df, fixed\_chunks, recursive\_chunks, "my\_chunking\_analysis")

# fixed\_metadata = save\_chunks\_to\_files(fixed\_chunks, "fixed")

# recursive\_metadata = save\_chunks\_to\_files(recursive\_chunks, "recursive")

print("🎯 FINAL SUMMARY")

print("=" \* 50)

print("✅ Jupyter notebook setup complete!")

print("📋 Key functions available:")

print(" • load\_sample\_data() - Generate test data")

print(" • test\_with\_real\_data() - Test with your CSV files")

print(" • export\_analysis\_report() - Generate comprehensive reports")

print(" • save\_chunks\_to\_files() - Export chunks to individual files")

print(" • visualize\_chunk\_analysis() - Create detailed visualizations")

print()

print("🚀 Next steps:")

print(" 1. Replace sample data with your actual CSV using test\_with\_real\_data()")

print(" 2. Adjust chunk parameters based on your data size and structure")

print(" 3. Run comparison analysis to choose the best method")

print(" 4. Export results using the provided export functions")

print()

print("💡 Pro tips:")

print(" • For datasets with clear entities/groups → Use Recursive Chunking")

print(" • For datasets without clear structure → Use Fixed Chunking")

print(" • Always validate chunk quality before production use")

print(" • Monitor memory usage for very large datasets")