# combined\_csv\_chunking\_app.py

import pandas as pd

import numpy as np

import io

import json

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

from dataclasses import dataclass

from datetime import datetime, timedelta

import hashlib

import logging

import uuid

from enum import Enum

import streamlit as st

# Configure logging

logging.basicConfig(level=logging.INFO)

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

# Enums for better type handling

class ChunkingMethod(Enum):

FIXED = "fixed"

RECURSIVE = "recursive"

class QualityStatus(Enum):

PASS = "pass"

FAIL = "fail"

PENDING = "pending"

@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]

quality\_status: QualityStatus = QualityStatus.PENDING

quality\_issues: List[str] = None

def \_\_post\_init\_\_(self):

if self.quality\_issues is None:

self.quality\_issues = []

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()

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

return {

'numeric': numeric\_cols,

'text': text\_cols,

'datetime': datetime\_cols,

'boolean': bool\_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

}

class FixedChunker:

"""Fixed-size chunking implementation"""

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

self.chunk\_size = chunk\_size

self.overlap\_ratio = overlap\_ratio

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

self.preserve\_headers = preserve\_headers

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}\_{str(uuid.uuid4())[:8]}"

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

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)

# Check if group size is within acceptable limits

if group\_size <= self.max\_chunk\_size and group\_size >= self.min\_chunk\_size:

# Group size is OK, add to current chunk

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

# Finalize current chunk if adding this group would exceed limit

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

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

# Group is too large, split it recursively

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"))

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)

else: # group\_size < self.min\_chunk\_size

# Group is too small, add to current chunk

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

# Finalize current chunk if adding this group would exceed limit

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}\_{str(uuid.uuid4())[:8]}",

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}\_{str(uuid.uuid4())[:8]}",

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}\_{str(uuid.uuid4())[:8]}",

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

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

all\_chunk\_rows = set()

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

issue\_msg = f"Schema mismatch in chunk {metadata.chunk\_id}"

results['issues'].append(issue\_msg)

metadata.quality\_issues.append(issue\_msg)

metadata.quality\_status = QualityStatus.FAIL

# Count rows

total\_rows += len(chunk\_df)

# Check for duplicate rows across chunks

for idx, row in chunk\_df.iterrows():

row\_hash = hashlib.md5(row.to\_string().encode()).hexdigest()

if row\_hash in all\_chunk\_rows:

issue\_msg = f"Duplicate row detected in chunk {metadata.chunk\_id}, row {idx}"

results['issues'].append(issue\_msg)

metadata.quality\_issues.append(issue\_msg)

metadata.quality\_status = QualityStatus.FAIL

all\_chunk\_rows.add(row\_hash)

# Boundary validation

if len(chunk\_df) == 0:

results['boundary\_validation'] = False

issue\_msg = f"Empty chunk detected: {metadata.chunk\_id}"

results['issues'].append(issue\_msg)

metadata.quality\_issues.append(issue\_msg)

metadata.quality\_status = QualityStatus.FAIL

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

# Data completeness check

if total\_rows != self.original\_row\_count:

results['data\_completeness'] = False

issue\_msg = f"Row count mismatch: {total\_rows} vs {self.original\_row\_count}"

results['issues'].append(issue\_msg)

# Update metadata for all chunks if quality passed

if results['schema\_consistency'] and results['data\_completeness'] and results['boundary\_validation']:

for \_, metadata in chunks:

if metadata.quality\_status == QualityStatus.PENDING:

metadata.quality\_status = QualityStatus.PASS

return results

class CSVChunkingBackend:

"""Backend class for CSV chunking operations"""

def \_\_init\_\_(self):

self.df = None

self.analysis = None

self.chunks = None

self.quality\_results = None

def load\_data(self, file\_path\_or\_buffer):

"""Load CSV data from file path or buffer"""

try:

self.df = pd.read\_csv(file\_path\_or\_buffer)

return True, f"Data loaded successfully: {len(self.df):,} rows × {len(self.df.columns)} columns"

except Exception as e:

return False, f"Error loading file: {str(e)}"

def analyze\_data(self):

"""Analyze the loaded data"""

if self.df is None:

return False, "No data loaded"

try:

analyzer = SchemaAnalyzer(self.df)

self.analysis = {

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

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

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

}

return True, "Data analysis completed successfully"

except Exception as e:

return False, f"Error analyzing data: {str(e)}"

def perform\_chunking(self, method: str, params: Dict[str, Any]):

"""Perform chunking with the specified method and parameters"""

if self.df is None:

return False, "No data loaded"

try:

source\_info = {

'method': method,

'params': params,

'original\_shape': self.df.shape,

'timestamp': datetime.now().isoformat()

}

if method == "Fixed Row Chunking":

chunker = FixedChunker(

chunk\_size=params['chunk\_size'],

overlap\_ratio=params['overlap\_ratio'],

preserve\_headers=params['preserve\_headers']

)

chunks = chunker.chunk(self.df, source\_info)

else: # Recursive

primary\_keys = params.get('primary\_keys', [])

chunker = RecursiveChunker(

max\_chunk\_size=params['max\_chunk\_size'],

min\_chunk\_size=params['min\_chunk\_size']

)

chunks = chunker.chunk(self.df, source\_info, primary\_keys)

# Quality validation

validator = QualityValidator(self.df)

quality\_results = validator.validate\_chunks(chunks)

self.chunks = chunks

self.quality\_results = quality\_results

# Generate error report if quality issues

error\_report = None

if quality\_results['issues']:

error\_report = self.\_generate\_error\_report(quality\_results, chunks)

success = quality\_results['schema\_consistency'] and quality\_results['data\_completeness'] and quality\_results['boundary\_validation']

message = f"Chunking completed! Generated {len(chunks)} chunks"

return success, message, error\_report

except Exception as e:

return False, f"Chunking failed: {str(e)}", None

def \_generate\_error\_report(self, quality\_results, chunks):

"""Generate a detailed error report"""

report = {

"timestamp": datetime.now().isoformat(),

"total\_issues": len(quality\_results['issues']),

"issues\_by\_type": {},

"chunks\_with\_issues": [],

"detailed\_issues": quality\_results['issues']

}

# Count issues by type

for issue in quality\_results['issues']:

issue\_type = issue.split(":")[0] if ":" in issue else "Other"

report["issues\_by\_type"][issue\_type] = report["issues\_by\_type"].get(issue\_type, 0) + 1

# Identify chunks with issues

for chunk\_df, metadata in chunks:

if metadata.quality\_issues:

report["chunks\_with\_issues"].append({

"chunk\_id": metadata.chunk\_id,

"issues": metadata.quality\_issues,

"row\_count": len(chunk\_df)

})

return report

def get\_chunk\_summary(self):

"""Get a summary of all chunks"""

if self.chunks is None:

return None

chunk\_summary = []

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

chunk\_summary.append({

'Chunk ID': metadata.chunk\_id,

'Method': metadata.method,

'Rows': metadata.total\_rows,

'Start Row': metadata.start\_row,

'End Row': metadata.end\_row,

'Overlap': metadata.overlap\_rows,

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

'Quality Status': metadata.quality\_status.value,

'Issue Count': len(metadata.quality\_issues)

})

return pd.DataFrame(chunk\_summary)

def get\_chunk\_data(self, chunk\_index):

"""Get data for a specific chunk"""

if self.chunks is None or chunk\_index >= len(self.chunks):

return None, None

return self.chunks[chunk\_index]

def download\_chunk(self, chunk\_index):

"""Download a specific chunk as CSV"""

if self.chunks is None or chunk\_index >= len(self.chunks):

return None, None

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

# Convert DataFrame to CSV string

csv\_buffer = io.StringIO()

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

csv\_data = csv\_buffer.getvalue()

filename = f"{metadata.chunk\_id}.csv"

return csv\_data, filename

class CSVChunkingFrontend:

"""Frontend class for CSV chunking UI"""

def \_\_init\_\_(self):

self.backend = CSVChunkingBackend()

# Initialize session state

if 'current\_method' not in st.session\_state:

st.session\_state.current\_method = None

if 'manual\_review\_data' not in st.session\_state:

st.session\_state.manual\_review\_data = None

if 'error\_report' not in st.session\_state:

st.session\_state.error\_report = None

# Set page config

st.set\_page\_config(

page\_title="CSV Chunking System",

page\_icon="📊",

layout="wide",

initial\_sidebar\_state="expanded"

)

def run(self):

"""Main application flow"""

st.title("📊 CSV Chunking System")

st.markdown("---")

# Render the sidebar for configuration

self.\_render\_sidebar()

# Main content area

if self.backend.df is not None:

self.\_render\_main\_content()

else:

st.info("👆 Please upload a CSV file to begin")

def \_render\_sidebar(self):

"""Render configuration sidebar"""

st.sidebar.header("⚙️ Configuration")

# File upload

uploaded\_file = st.sidebar.file\_uploader(

"Upload CSV File",

type=['csv'],

help="Upload a CSV file to start chunking"

)

if uploaded\_file is not None:

success, message = self.backend.load\_data(uploaded\_file)

if success:

st.sidebar.success(message)

# Analyze the data

success, message = self.backend.analyze\_data()

if not success:

st.sidebar.error(message)

else:

st.sidebar.error(message)

st.sidebar.markdown("---")

# Only show chunking options if data is loaded

if self.backend.df is not None:

# Chunking method selection

method = st.sidebar.selectbox(

"Select Chunking Method",

["Fixed Row Chunking", "Recursive Entity Chunking"],

help="Choose the chunking strategy"

)

st.sidebar.markdown("### Method Parameters")

if method == "Fixed Row Chunking":

chunk\_size = st.sidebar.number\_input(

"Chunk Size (rows)",

min\_value=100,

max\_value=100000,

value=1000,

step=100,

help="Number of rows per chunk"

)

overlap\_ratio = st.sidebar.slider(

"Overlap Ratio",

min\_value=0.0,

max\_value=0.3,

value=0.1,

step=0.05,

help="Fraction of chunk size to overlap between chunks"

)

preserve\_headers = st.sidebar.checkbox(

"Preserve Headers in Each Chunk",

value=True,

help="Include column headers in each chunk"

)

params = {

'chunk\_size': chunk\_size,

'overlap\_ratio': overlap\_ratio,

'preserve\_headers': preserve\_headers

}

else: # Recursive

max\_chunk\_size = st.sidebar.number\_input(

"Max Chunk Size (rows)",

min\_value=1000,

max\_value=100000,

value=5000,

step=500,

help="Maximum rows per chunk before splitting"

)

min\_chunk\_size = st.sidebar.number\_input(

"Min Chunk Size (rows)",

min\_value=100,

max\_value=10000,

value=500,

step=100,

help="Minimum viable chunk size"

)

# Primary key selection for recursive chunking

primary\_keys = []

if self.backend.analysis:

pk\_options = self.backend.analysis['relationships']['primary\_keys']

if pk\_options:

primary\_keys = st.sidebar.multiselect(

"Select Primary Keys for Entity Grouping",

options=pk\_options,

default=pk\_options[0] if pk\_options else None,

help="Select columns to use for entity-based chunking"

)

params = {

'max\_chunk\_size': max\_chunk\_size,

'min\_chunk\_size': min\_chunk\_size,

'primary\_keys': primary\_keys

}

# Chunking button

if st.sidebar.button("🚀 Start Chunking", type="primary"):

success, message, error\_report = self.backend.perform\_chunking(method, params)

st.session\_state.current\_method = method

st.session\_state.error\_report = error\_report

if success:

st.sidebar.success(message)

else:

st.sidebar.error(message)

def \_render\_main\_content(self):

"""Render main content area"""

st.subheader("📄 Data Overview")

# Data preview

with st.expander("🔍 Data Preview", expanded=True):

st.dataframe(self.backend.df.head(10), use\_container\_width=True)

# Schema information

if self.backend.analysis:

with st.expander("📋 Schema Analysis"):

col1, col2, col3 = st.columns(3)

schema = self.backend.analysis['schema']

with col1:

st.metric("Total Rows", f"{schema['total\_rows']:,}")

st.metric("Total Columns", schema['total\_columns'])

with col2:

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

st.metric("Duplicate Rows", f"{schema['duplicate\_rows']:,}")

with col3:

data\_types = self.backend.analysis['data\_types']

st.metric("Numeric Columns", len(data\_types['numeric']))

st.metric("Text Columns", len(data\_types['text']))

# Primary keys

pk\_cols = self.backend.analysis['relationships']['primary\_keys']

if pk\_cols:

st.success(f"🔑 Detected Primary Keys: {', '.join(pk\_cols)}")

else:

st.info("ℹ️ No clear primary keys detected")

# Chunks results

if self.backend.chunks:

st.subheader("📦 Chunking Results")

# Quality gate status

quality = self.backend.quality\_results

quality\_passed = quality['schema\_consistency'] and quality['data\_completeness'] and quality['boundary\_validation']

if quality\_passed:

st.success("✅ All quality checks passed!")

else:

st.error("❌ Quality issues detected:")

for issue in quality['issues']:

st.error(f"• {issue}")

# Show error handling options

self.\_render\_error\_handling()

# Results summary

col1, col2, col3, col4 = st.columns(4)

chunks = self.backend.chunks

with col1:

st.metric("Total Chunks", len(chunks))

with col2:

total\_rows = sum(len(chunk[0]) for chunk in chunks)

st.metric("Total Rows", f"{total\_rows:,}")

with col3:

avg\_size = total\_rows / len(chunks) if chunks else 0

st.metric("Avg Chunk Size", f"{avg\_size:.0f}")

with col4:

sizes = [len(chunk[0]) for chunk in chunks]

std\_dev = pd.Series(sizes).std() if sizes else 0

st.metric("Size Std Dev", f"{std\_dev:.0f}")

# Chunk details

self.\_render\_chunk\_details()

def \_render\_error\_handling(self):

"""Render error handling options"""

st.subheader("🛠️ Error Handling")

option = st.radio(

"Select action to resolve quality issues:",

["Adjust Parameters", "Switch Method", "Manual Review"]

)

if option == "Adjust Parameters":

st.info("Adjust the parameters in the sidebar and try again.")

if st.button("🔄 Retry with Adjusted Parameters"):

# This will use the existing method but with new parameters from sidebar

st.rerun()

elif option == "Switch Method":

new\_method = st.selectbox(

"Select new method:",

["Fixed Row Chunking", "Recursive Entity Chunking"],

index=0 if st.session\_state.current\_method == "Recursive Entity Chunking" else 1

)

if st.button("🔄 Switch Method"):

st.session\_state.current\_method = new\_method

st.rerun()

else: # Manual Review

st.warning("Manual review allows you to inspect and modify chunks with issues.")

if st.button("🔍 Start Manual Review"):

# Prepare data for manual review

problematic\_chunks = []

for chunk\_df, metadata in self.backend.chunks:

if metadata.quality\_issues:

problematic\_chunks.append((chunk\_df.copy(), metadata))

if problematic\_chunks:

st.session\_state.manual\_review\_data = problematic\_chunks

st.rerun()

else:

st.info("No chunks with issues found for manual review.")

def \_render\_chunk\_details(self):

"""Render detailed chunk information"""

with st.expander("📊 Chunk Details", expanded=True):

# Create summary dataframe

summary\_df = self.backend.get\_chunk\_summary()

if summary\_df is not None:

st.dataframe(summary\_df, use\_container\_width=True)

# Individual chunk inspector

st.subheader("🔍 Chunk Inspector")

if self.backend.chunks:

selected\_chunk\_idx = st.selectbox(

"Select chunk to inspect:",

range(len(self.backend.chunks)),

format\_func=lambda x: f"Chunk {x}: {self.backend.chunks[x][1].chunk\_id} ({len(self.backend.chunks[x][0])} rows)"

)

if selected\_chunk\_idx is not None:

chunk\_df, metadata = self.backend.get\_chunk\_data(selected\_chunk\_idx)

col1, col2 = st.columns([3, 1])

with col1:

st.dataframe(chunk\_df.head(10), use\_container\_width=True)

with col2:

st.json({

'chunk\_id': metadata.chunk\_id,

'method': metadata.method,

'total\_rows': metadata.total\_rows,

'overlap\_rows': metadata.overlap\_rows,

'entity\_group': metadata.entity\_group,

'quality\_status': metadata.quality\_status.value,

'issue\_count': len(metadata.quality\_issues),

'columns': len(metadata.columns)

})

# Show quality issues if any

if metadata.quality\_issues:

st.warning("Quality Issues:")

for issue in metadata.quality\_issues:

st.write(f"• {issue}")

# Download button for individual chunk

csv\_data, file\_name = self.backend.download\_chunk(selected\_chunk\_idx)

if csv\_data:

st.download\_button(

label=f"📥 Download {metadata.chunk\_id}.csv",

data=csv\_data,

file\_name=file\_name,

mime="text/csv"

)

# Manual review interface if activated

if st.session\_state.manual\_review\_data:

self.\_render\_manual\_review()

def \_render\_manual\_review(self):

"""Render manual review interface"""

st.subheader("✏️ Manual Review Interface")

review\_data = st.session\_state.manual\_review\_data

st.info(f"Found {len(review\_data)} chunks with issues for manual review.")

# Select a chunk to review

chunk\_options = [f"{metadata.chunk\_id} ({len(chunk\_df)} rows, {len(metadata.quality\_issues)} issues)"

for chunk\_df, metadata in review\_data]

selected\_chunk = st.selectbox("Select chunk to review:", range(len(review\_data)), format\_func=lambda x: chunk\_options[x])

if selected\_chunk is not None:

chunk\_df, metadata = review\_data[selected\_chunk]

st.write(f"\*\*Reviewing Chunk:\*\* {metadata.chunk\_id}")

st.write(f"\*\*Issues:\*\* {', '.join(metadata.quality\_issues)}")

# Editable dataframe

edited\_df = st.data\_editor(chunk\_df, num\_rows="dynamic", use\_container\_width=True)

col1, col2, col3 = st.columns(3)

with col1:

if st.button("✅ Approve This Chunk", type="primary"):

# Mark this chunk as approved

metadata.quality\_status = QualityStatus.PASS

metadata.quality\_issues = []

st.success("Chunk approved!")

with col2:

if st.button("🔄 Revert Changes"):

# Reset to original

st.rerun()

with col3:

# Download the edited chunk

csv\_buffer = io.StringIO()

edited\_df.to\_csv(csv\_buffer, index=False)

st.download\_button(

label="📥 Download Edited Chunk",

data=csv\_buffer.getvalue(),

file\_name=f"edited\_{metadata.chunk\_id}.csv",

mime="text/csv"

)

# Final approval

st.markdown("---")

if st.button("✅ Finalize All Approved Chunks", type="primary"):

# Update the original chunks with approved ones

updated\_chunks = []

for orig\_idx, (orig\_chunk\_df, orig\_metadata) in enumerate(self.backend.chunks):

# Check if this chunk was reviewed and approved

reviewed = False

for review\_idx, (review\_df, review\_metadata) in enumerate(review\_data):

if orig\_metadata.chunk\_id == review\_metadata.chunk\_id and review\_metadata.quality\_status == QualityStatus.PASS:

# Replace with the reviewed version

updated\_chunks.append((review\_df, review\_metadata))

reviewed = True

break

if not reviewed:

updated\_chunks.append((orig\_chunk\_df, orig\_metadata))

# Update backend chunks

self.backend.chunks = updated\_chunks

# Revalidate quality

validator = QualityValidator(self.backend.df)

quality\_results = validator.validate\_chunks(updated\_chunks)

self.backend.quality\_results = quality\_results

st.session\_state.manual\_review\_data = None

st.success("Manual review completed! Chunks updated.")

st.rerun()

# Main execution

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

app = CSVChunkingFrontend()

app.run()