**主な最適化ポイント**

1. **チャンク処理**: 大きなファイルを小さな部分に分割して処理することで、メモリ使用量を抑制
2. **SQLiteデータベース**: 非常に大きなデータセット用に、メモリ内ではなくSQLiteを使用するオプション
3. **ハッシュベースの重複検出**: MD5ハッシュを使用してキー生成を効率化
4. **並列処理**: マルチプロセッシングによるハッシュ計算の高速化
5. **進捗表示とログ**: 詳細なログと進捗状況の表示
6. **エラーハンドリング**: 堅牢なエラー処理とリソースのクリーンアップ
7. **メモリ使用量の監視**: 処理中のメモリ使用量を監視し、必要に応じてガベージコレクションを実行
8. **ファイルタイプごとの最適化**: ExcelファイルとCSVファイルに対して異なる読み込み戦略を実装
9. **パラメータ調整**: 環境やデータサイズに応じて調整可能なパラメータ群
10. **インデックス使用**: SQLite使用時にインデックスを活用して検索を高速化

**パフォーマンス比較**

以下は様々なデータサイズでの処理時間の目安です（一般的なPCでの目安）：

| **データサイズ** | **行数** | **メモリ内処理** | **SQLite使用** | **並列処理有効時** |
| --- | --- | --- | --- | --- |
| 小 | ~10,000 | 数秒 | 数秒 | ほぼ同じ |
| 中 | ~100,000 | 20-30秒 | 30-40秒 | 30-40%高速 |
| 大 | ~1,000,000 | 3-5分 | 6-8分 | 40-50%高速 |
| 超大 | ~10,000,000 | メモリ不足の可能性 | 60-90分 | 30-40%高速 |

**使用シナリオと推奨設定**

**小〜中規模データの場合（数万行まで）**

python

processor = DuplicateProcessor(

chunk\_size=100000,

use\_sqlite=False,

use\_parallel=True

)

**大規模データの場合（数十万〜数百万行）**

python

processor = DuplicateProcessor(

chunk\_size=50000,

use\_sqlite=True,

sqlite\_path="temp\_large\_data.sqlite",

use\_parallel=True

)

**メモリが限られた環境の場合**

python

processor = DuplicateProcessor(

chunk\_size=10000,

use\_sqlite=True,

sqlite\_path="temp\_db.sqlite",

use\_parallel=False

)

**高速処理が必要な場合（多数のCPUコアがある）**

python

processor = DuplicateProcessor(

chunk\_size=100000,

use\_parallel=True,

max\_workers=16, *# 利用可能なコア数に応じて調整*

hash\_method='md5' *# 高速なハッシュ計算*

)

**追加の便利な機能**

このコードには、データ処理において有用な機能も含まれています。必要に応じて活用できます：

**1. ファイル形式の変換**

異なる形式で出力するには、出力ファイルの拡張子を変更するだけです：

python

*# CSVで出力*

processor.process\_files(file\_paths, output\_path="処理済みデータ.csv")

*# Excelで出力*

processor.process\_files(file\_paths, output\_path="処理済みデータ.xlsx")

**2. ログファイルの活用**

処理中の詳細なログは deduplication.log ファイルに自動的に保存されます。このログには以下の情報が含まれます：

* 処理されたファイルと行数
* 重複の詳細情報
* エラーと警告
* パフォーマンスメトリクス

**3. カスタム列での重複チェック**

重複チェックに使用する列をカスタマイズするには、初期化時に指定できます：

python

processor = DuplicateProcessor()

processor.duplicate\_check\_columns = [

'日付', '病棟コード', '診療科名', *# キー列*

'在院患者数' *# 必要な数値列のみに限定*

]

**4. 統計情報の抽出**

重複の詳細な統計情報を取得するためのメソッドも追加できます：

python

def get\_duplicate\_statistics(self, result\_df):

"""

重複に関する詳細統計情報を取得

Args:

result\_df: 処理結果のデータフレーム

Returns:

dict: 統計情報の辞書

"""

stats = {}

*# ソース（ファイル）ごとの行数*

stats['source\_counts'] = result\_df.groupby('元ファイル').size().to\_dict()

*# 日付ごとの行数*

stats['date\_counts'] = result\_df.groupby(result\_df['日付'].dt.date).size().to\_dict()

*# 診療科別の行数*

stats['department\_counts'] = result\_df.groupby('診療科名').size().to\_dict()

*# 病棟別の行数*

stats['ward\_counts'] = result\_df.groupby('病棟コード').size().to\_dict()

return stats

**5. メモリ使用量のグラフ表示**

処理中のメモリ使用量を視覚化するためのオプションを追加できます：

python

def plot\_memory\_usage(self, log\_data):

"""

メモリ使用量の履歴をグラフ表示

Args:

log\_data: メモリ使用量のログデータ

"""

import matplotlib.pyplot as plt

timestamps = [entry['timestamp'] for entry in log\_data]

memory\_usage = [entry['memory']['rss\_mb'] for entry in log\_data]

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

plt.plot(timestamps, memory\_usage)

plt.title('メモリ使用量の推移')

plt.xlabel('時間')

plt.ylabel('メモリ使用量 (MB)')

plt.xticks(rotation=45)

plt.tight\_layout()

plt.savefig('memory\_usage.png')

plt.show()

**まとめ**

この拡張版重複処理モジュールは、以下のような特長を持っています：

1. **拡張性**: データサイズに応じて柔軟に設定を調整可能
2. **効率性**: 最小限のメモリ使用量で大規模データを処理
3. **堅牢性**: 包括的なエラーハンドリングとログ記録
4. **カスタマイズ性**: 様々なユースケースに対応可能

最も大きなデータセットでも効率的に処理できるよう、チャンクサイズとSQLiteオプションを適切に設定してください。特に、数百万行を超えるデータセットでは、SQLiteを使用することを強く推奨します。

なお、このコードは汎用的な重複処理のフレームワークとして設計されているため、具体的なプロジェクトのニーズに合わせて調整することも可能です。処理速度とメモリ使用量のバランスが重要な場合は、チャンクサイズとワーカー数を適宜調整してください。

再試行

G

今作成中のアプリで具体的に修正するファイルはどれですか？

編集

今回のアプリケーションで具体的に修正が必要なファイルは主に以下の2つです：

**1. integrated\_preprocessing.py**

このファイルが重複検出と処理の中核部分であり、前回修正を提案したファイルです。診療科の集約と重複処理のロジックがあります。

具体的な修正ポイント:

* 日付・病棟コード・診療科名の重複チェック方法
* 「その他」への集約後の集計処理

**2. data\_processing\_tab.py**

このファイルは integrated\_preprocessing.py を呼び出してデータ処理を実行するUI部分です。

具体的な修正ポイント:

* 複数ファイルの処理ロジック
* ファイル間重複の処理方法

現在の問題を解決するためには、2つのアプローチが考えられます：

**アプローチ1: 現在のコードを最小限に修正する（軽量な変更）**

このアプローチでは、現在のアプリケーション構造を維持しながら、integrated\_preprocessing.py の重複検出と処理ロジックのみを修正します。

python

*# integrated\_preprocessing.py の修正箇所*

*# データの初期集計 (重複削除ではなく集計)*

if '日付' in df\_processed.columns and '病棟コード' in df\_processed.columns and '診療科名' in df\_processed.columns:

*# 欠損値の処理*

df\_processed['診療科名'] = df\_processed['診療科名'].fillna("空白診療科").astype(str)

*# 最初のグループ化と集計 (診療科別のデータを保持)*

numeric\_columns = ['在院患者数', '入院患者数', '緊急入院患者数', '退院患者数', '死亡患者数']

available\_numeric\_cols = [col for col in numeric\_columns if col in df\_processed.columns]

*# グループ化して集計 (診療科レベルで)*

if available\_numeric\_cols:

df\_grouped = df\_processed.groupby(['日付', '病棟コード', '診療科名'])[available\_numeric\_cols].sum().reset\_index()

*# 以下、既存コード*

**アプローチ2: 新しい重複処理モジュールを追加する（大規模な変更）**

このアプローチでは、先ほど提案した堅牢な重複処理モジュール duplicate\_processor.py を新しく追加し、data\_processing\_tab.py からこのモジュールを呼び出すように変更します。

**1. 新規ファイル duplicate\_processor.py を追加**

* 先ほど提案した完全なコードをこのファイルに保存

**2. data\_processing\_tab.py の修正**

python

*# ファイルのインポート部分*

from integrated\_preprocessing import integrated\_preprocess\_data

from duplicate\_processor import DuplicateProcessor *# 新しいモジュールをインポート*

*# process\_data\_with\_progress 関数内で、複数ファイルを処理する箇所*

if new\_files\_uploaded:

*# 複数ファイル間の重複処理を行う*

all\_files = [base\_file\_uploaded] + new\_files\_uploaded

processor = DuplicateProcessor(chunk\_size=50000)

df\_raw = processor.process\_files(all\_files, return\_df=True)

*# 以下、既存コード*

**推奨するアプローチ**

お使いのアプリの状況とデータ量によって最適なアプローチは異なりますが、推奨は以下の通りです：

1. **データ量が少なめ（数万行まで）**：
   * アプローチ1を選択
   * integrated\_preprocessing.py の重複検出部分のみを修正
2. **データ量が多い（数十万行以上）または将来的な拡張性が重要**：
   * アプローチ2を選択
   * 新しい duplicate\_processor.py を追加
   * data\_processing\_tab.py からこれを呼び出すよう修正

あるいは段階的なアプローチとして：

1. まず、アプローチ1で integrated\_preprocessing.py の重複検出ロジックを修正
2. それでも問題が解決しない、または新たな問題が発生した場合は、アプローチ2を検討

いずれを選択する場合も、修正前にはコードのバックアップを取り、テストデータで十分に検証することをお勧めします。