Análise Técnica Completa - Dashboard SINAN

Índice

---

1. Visão Geral

O dashboard SINAN é uma aplicação Streamlit que analisa dados reais de notificações de violência contra crianças e adolescentes (0-17 anos) do Sistema de Informação de Agravos de Notificação (SINAN) do Ministério da Saúde.

Características principais:

- Processa milhões de registros de arquivos Parquet (2019-2024)
- Interface interativa com filtros dinâmicos
- 10 hipóteses de pesquisa respondidas através de visualizações
- Otimizado para grandes volumes de dados
- Conformidade com LGPD (dados agregados e anonimizados)

---

2. Bibliotecas e Dependências

2.1 Bibliotecas Principais

Streamlit (`streamlit>=1.49.0`)

- **Uso**: Framework web para criação do dashboard interativo
- **Funcionalidades utilizadas**:
  - `st.set_page_config()`: Configuração da página
  - `st.cache_data()`: Cache de dados para otimização

  - `st.sidebar.*`: Filtros laterais
  - `st.plotly_chart()`: Exibição de gráficos
  - `st.metric()`: Indicadores (KPIs)
  - `st.dataframe()`: Tabelas de dados

Pandas (`pandas>=2.2.0`)

- **Uso**: Manipulação e análise de dados
- **Funcionalidades principais**:
  - `pd.read_parquet()`: Leitura de arquivos Parquet
  - `pd.concat()`: Concatenação de DataFrames
  - `pd.to_datetime()`: Conversão de datas
  - `pd.to_numeric()`: Conversão numérica
  - Operações de filtragem, agregação e transformação
  - `groupby()`: Agrupamento de dados

Plotly Express (`plotly>=5.15.0`)

- **Uso**: Criação de gráficos interativos
- **Tipos de gráficos utilizados**:
  - `px.line()`: Gráficos de linha (tendências temporais)
  - `px.bar()`: Gráficos de barras (distribuições)
  - `px.bar()` com `orientation='h'`: Barras horizontais
  - `barmode='stack'`: Barras empilhadas
  - `barmode='group'`: Barras agrupadas

Plotly Graph Objects (`plotly.graph_objects`)

- **Uso**: Customização avançada de gráficos
- **Funcionalidades**: Ajustes de layout, formatação de eixos

DuckDB (`duckdb>=0.9.0`) - Opcional

- **Uso**: Banco de dados analítico para consultas SQL em Parquet
- **Benefícios**:
  - Consultas SQL diretas em arquivos Parquet
  - Filtragem antes de carregar na memória
  - Agregações otimizadas
  - Redução de uso de memória em 70-90%

NumPy (`numpy>=2.3.0`)

- **Uso**: Operações numéricas e arrays
- **Utilizado indiretamente** através do Pandas

PyArrow (`pyarrow>=10.0.0`)

- **Uso**: Leitura/escrita de arquivos Parquet

- **Utilizado** pelo Pandas para processar arquivos Parquet

2.2 Bibliotecas Auxiliares

- **Pathlib**: Manipulação de caminhos de arquivos
- **Collections.Counter**: Contagem de elementos
- **Re**: Expressões regulares (usado em munic_dict_loader)

2.3 Bibliotecas Não Utilizadas (podem ser removidas)

- `dbfread`: Não utilizada no código atual
- `pathlib2`: Substituída por `pathlib` padrão
- `scipy`: Não utilizada
- `scikit-learn`: Não utilizada
- `seaborn`: Não utilizada
- `matplotlib`: Não utilizada (Plotly é usado)

---

3. Arquitetura do Sistema

3.1 Estrutura de Arquivos

```
SINAN-BIG-DATA-PYTHON/
  dashboard_sinan_real_data.py          # Aplicação principal Streamlit
  sinan_data_processor_comprehensive.py # Processador de dados (método tradicional)
  sinan_data_processor_duckdb.py        # Processador otimizado com DuckDB
  munic_dict_loader.py                  # Carregador de dicionários de municípios
  VIOLBR-PARQUET/                       # Dados de violência (2019-2024)
      VIOLBR19.parquet
      VIOLBR20.parquet
      VIOLBR21.parquet
      VIOLBR22.parquet
      VIOLBR23.parquet
      VIOLBR24.parquet
  TAB_SINANONLINE/                      # Dicionários de decodificação
      Munic*.cnv                        # Dicionários de municípios por UF
      [outros arquivos .cnv]            # Outros dicionários
```

3.2 Fluxo de Dados

```
1. Carregamento
     DuckDB (se disponível) → Consulta SQL direta em Parquet
     Pandas (fallback) → Carrega todos os arquivos

2. Filtragem
     Por idade (0-17 anos) → Códigos 4000-4017
     Por tipo de violência → VIOL_SEXU, VIOL_FISIC, VIOL_PSICO, VIOL_INFAN
```

```
3. Decodificação
      Aplicação de dicionários → Valores numéricos → Texto legível

4. Transformação
      Mapeamento de UF (códigos → nomes)
      Mapeamento de municípios (códigos → nomes)
      Criação de colunas derivadas (TIPO_VIOLENCIA, FAIXA_ETARIA, etc.)
      Correção de colunas (AUTOR_SEXO, GRAU_PARENTESCO)

5. Visualização
      Streamlit + Plotly → Gráficos interativos
```

---

4. Fluxo de Processamento de Dados

4.1 Função Principal: `load_sinan_data()`

Localização: `dashboard_sinan_real_data.py` (linhas 120-445)

Fluxo detalhado:

Etapa 1: Decisão de Método de Carregamento

```python
if use_duckdb and DUCKDB_AVAILABLE:
    # Método otimizado com DuckDB
    with SINANDataProcessorDuckDB(...) as processor_duckdb:
        # Consulta SQL direta nos arquivos Parquet
        child_data_raw = processor_duckdb.load_filtered_violence_data(
            age_filter=True,
            violence_filter=True
        )
else:
    # Método tradicional com Pandas
    processor = SINANDataProcessorComprehensive(...)
    violence_data = processor.load_violence_data()
```

Explicação:

- DuckDB permite consultas SQL diretas nos arquivos Parquet sem carregar tudo na memória
- Pandas carrega todos os arquivos primeiro, depois filtra
- DuckDB é 5-10x mais rápido e usa 70-90% menos memória

Etapa 2: Filtragem por Idade (Método Pandas)

```python
# Códigos de idade: 4000 (menor de 1 ano) até 4017 (17 anos)
age_codes = ['4000'] + [f'400{i}' for i in range(1, 18)]
```

```
age_str = violence_data['NU_IDADE_N'].astype(str)
age_filter = age_str.isin(age_codes)
child_data_raw = violence_data[age_filter].copy(deep=False)
```

Explicação:

- O SINAN usa códigos específicos para idades: `4000` = menor de 1 ano, `4001` = 1 ano, ..., `4017` = 17 anos
- Filtragem feita ANTES de aplicar dicionários para reduzir memória
- `copy(deep=False)` cria shallow copy (economiza memória)

Etapa 3: Seleção de Colunas Essenciais

```
essential_columns = [
    'DT_NOTIFIC', 'NU_ANO', 'SG_UF_NOT', 'SG_UF', 'ID_MUNICIP', 'ID_MN_RESI',
    'NU_IDADE_N', 'CS_SEXO', 'VIOL_FISIC', 'VIOL_PSICO', 'VIOL_SEXU', 'VIOL_INFAN',
    'LOCAL_OCOR', 'AUTOR_SEXO', 'AUTOR_ALCO', 'CS_ESCOL_N', 'CS_RACA'
]
rel_cols = [col for col in child_data_raw.columns if col.startswith('REL_')]
columns_to_keep = list(set(available_essential + rel_cols + other_cols))
```

Explicação:

- Mantém apenas colunas necessárias para análise
- Inclui todas as colunas `REL_*` (relacionamento com agressor)
- Reduz drasticamente o uso de memória

Etapa 4: Aplicação de Dicionários

```
decoded_data = processor.apply_dictionaries(child_data_subset)
```

Processo interno (`sinan_data_processor_comprehensive.py`):

- Converte valores numéricos/códigos em texto legível
- Exemplo: `'1'` → `'Sim'`, `'4001'` → `'01 ano'`
- Usa `map()` e `fillna()` para manter valores não mapeados
- Processa apenas colunas necessárias

Etapa 5: Filtragem por Violência

```
child_data = processor.filter_comprehensive_violence(
    decoded_data,
    already_filtered_by_age=True
)
```

Lógica:

- Verifica colunas `VIOL_SEXU`, `VIOL_FISIC`, `VIOL_PSICO`, `VIOL_INFAN`
- Aceita valores: `'1'`, `'Sim'`, `'SIM'`, `'S'`, `'1.0'` ou numérico `1`
- Combina condições com OR (qualquer tipo de violência)

Etapa 6: Transformações e Mapeamentos **6.1 Mapeamento de UF (Estados)**

```
uf_dict = {
    '11': 'Rondônia', '12': 'Acre', ..., '53': 'Distrito Federal',
    11: 'Rondônia', 12: 'Acre', ..., 53: 'Distrito Federal'  # Suporta ambos
}
df['UF_NOTIFIC'] = df['SG_UF_NOT'].apply(map_uf)
```

6.2 Mapeamento de Municípios

```
municip_dict = load_municipality_dictionary()  # Carrega de arquivos .cnv
df['MUNICIPIO_NOTIFIC'] = df['ID_MUNICIP'].apply(
    lambda x: map_municipio(x, municip_dict)
)
```

6.3 Criação de Colunas Derivadas

- **TIPO_VIOLENCIA**: Combina tipos de violência em uma string
- **FAIXA_ETARIA**: Agrupa idades (0-1, 2-5, 6-9, 10-13, 14-17)
- **SEXO**: Normaliza valores de sexo
- **AUTOR_SEXO_CORRIGIDO**: Filtra valores inválidos de sexo do agressor
- **GRAU_PARENTESCO**: Extrai relacionamentos das colunas `REL_*`

---

5. Função e Responsabilidades de Cada Arquivo

5.1 `dashboard_sinan_real_data.py` - Aplicação Principal

Função Principal: Interface web interativa (dashboard) para visualização e análise de dados de violência infantil.

Responsabilidades:

1. **Interface do Usuário (UI)**:
   - Renderiza a interface web usando Streamlit
   - Exibe KPIs (métricas principais) em tempo real
   - Cria 10 gráficos interativos respondendo às hipóteses de pesquisa
   - Gerencia filtros interativos (ano, UF, município, tipo de violência)
2. **Orquestração do Processamento**:
   - Chama funções de carregamento de dados
   - Decide entre usar DuckDB (otimizado) ou Pandas (fallback)
   - Coordena transformações de dados
   - Aplica filtros dinâmicos baseados na interação do usuário
3. **Transformações de Dados**:
   - Mapeia códigos de UF para nomes de estados

- Mapeia códigos de municípios para nomes
- Cria colunas derivadas (TIPO_VIOLENCIA, FAIXA_ETARIA, GRAU_PARENTESCO)
- Corrige dados inconsistentes (AUTOR_SEXO)
- Formata números no padrão brasileiro

4. **Visualizações**:

   - Gera gráficos usando Plotly Express
   - Formata eixos, rótulos e cores
   - Aplica formatação brasileira aos números
   - Otimiza visualizações para evitar sobreposição

Fluxo de Execução:

```
1. Carrega dados (com cache) → load_sinan_data()
2. Exibe título e informações
3. Cria filtros na sidebar
4. Aplica filtros ao DataFrame
5. Calcula e exibe KPIs
6. Gera e exibe 10 gráficos interativos
7. Exibe notas metodológicas e conformidade LGPD
```

Dependências:

- Importa `SINANDataProcessorComprehensive` (processador tradicional)
- Importa `SINANDataProcessorDuckDB` (processador otimizado, opcional)
- Importa `load_municipality_dict` (carregador de municípios)

Características Especiais:

- Cache de dados com TTL de 1 hora (`@st.cache_data(ttl=3600)`)
- Cache de dicionários de municípios com TTL de 24 horas
- Fallback automático se DuckDB não estiver disponível
- Tratamento robusto de erros com mensagens amigáveis

5.2 `sinan_data_processor_comprehensive.py` - Processador de Dados Tradicional

Função Principal: Classe responsável por carregar, decodificar e filtrar dados do SINAN usando Pandas.

Responsabilidades:

1. **Carregamento de Dados**:

   - Lista e carrega todos os arquivos `.parquet` da pasta `VIOLBR-PARQUET`
   - Concatena múltiplos arquivos em um único DataFrame
   - Trata erros de leitura de arquivos individuais

2. **Gerenciamento de Dicionários**:
    - Carrega dicionários hardcoded para decodificação de códigos
    - Mapeia códigos numéricos para valores legíveis
    - Exemplos:
        - `'1'` → `'Sim'` (para colunas de violência)
        - `'4001'` → `'01 ano'` (para idade)
        - `'01'` → `'Residência'` (para local de ocorrência)
3. **Aplicação de Dicionários**:
    - Aplica dicionários apenas nas colunas necessárias
    - Usa `copy(deep=False)` para economizar memória
    - Processa colunas `REL_*` (relacionamentos) limitadas
    - Mantém valores não mapeados com `fillna()`
4. **Filtragem de Dados**:
    - Filtra por idade (0-17 anos) usando códigos especiais
    - Filtra por tipo de violência (qualquer tipo)
    - Aceita múltiplos formatos de valores ('1', 'Sim', 1, etc.)
    - Trata divisão por zero em estatísticas
5. **Geração de Estatísticas**:
    - Gera estatísticas completas (temporal, demográfica, socioeconômica)
    - Cria resumos de análise
    - Calcula contagens e distribuições

Classe Principal: `SINANDataProcessorComprehensive`

Métodos Principais:

- `load_dictionaries()`: Carrega dicionários de decodificação
- `load_violence_data()`: Carrega arquivos Parquet
- `apply_dictionaries()`: Aplica dicionários aos dados
- `filter_comprehensive_violence()`: Filtra violência infantil
- `generate_comprehensive_statistics()`: Gera estatísticas
- `process_all_data()`: Pipeline completo de processamento

Características:

- Método tradicional (fallback quando DuckDB não está disponível)
- Processa dados em memória (requer mais RAM)
- Mais lento para grandes volumes, mas mais simples
- Não requer dependências adicionais além de Pandas

5.3 `sinan_data_processor_duckdb.py` - Processador Otimizado

Função Principal: Classe otimizada que usa DuckDB para consultas SQL diretas em arquivos Parquet, reduzindo drasticamente o uso de memória e tempo de processamento.

Responsabilidades:

1. **Consultas SQL em Parquet**:
   - Executa consultas SQL diretamente nos arquivos Parquet
   - Não precisa carregar todos os dados na memória primeiro
   - Filtra dados ANTES de carregar (pushdown de filtros)
2. **Filtragem Otimizada**:
   - Aplica filtros de idade diretamente no SQL
   - Carrega apenas colunas necessárias
   - Combina múltiplos arquivos com `UNION ALL`
3. **Agregações Eficientes**:
   - Realiza agregações diretamente no DuckDB
   - Muito mais rápido que agregações em Pandas
   - Reduz transferência de dados

Classe Principal: `SINANDataProcessorDuckDB`

Características Especiais:

- **Context Manager**: Implementa `__enter__` e `__exit__` para gerenciar conexão
- **Herança Funcional**: Usa `SINANDataProcessorComprehensive` para dicionários
- **Opcional**: Se DuckDB não estiver instalado, o sistema usa fallback

Métodos Principais:

- `query_with_filters()`: Executa consulta SQL com filtros
- `load_filtered_violence_data()`: Carrega dados já filtrados
- `aggregate_by_filters()`: Agrega dados diretamente no DuckDB

Vantagens sobre Método Tradicional:

- **70-90% menos uso de memória**: Filtra antes de carregar
- **5-10x mais rápido**: Consultas SQL otimizadas
- **Escalável**: Funciona bem com milhões de registros
- **Eficiente**: Carrega apenas dados necessários

Exemplo de Query Gerada:

```
SELECT DT_NOTIFIC, NU_ANO, SG_UF_NOT, ...
FROM read_parquet('VIOLBR19.parquet')
WHERE NU_IDADE_N IN ('4000', '4001', ..., '4017')
```

```
UNION ALL
SELECT DT_NOTIFIC, NU_ANO, SG_UF_NOT, ...
FROM read_parquet('VIOLBR20.parquet')
WHERE NU_IDADE_N IN ('4000', '4001', ..., '4017')
...
```

Quando Usar:

- Quando DuckDB está instalado (`pip install duckdb`)
- Para grandes volumes de dados (> 1 milhão de registros)
- Quando há limitações de memória
- Para melhor performance geral

5.4 `munic_dict_loader.py` - Carregador de Dicionários de Municípios

Função Principal: Módulo utilitário que lê arquivos `.cnv` e cria um dicionário mapeando códigos de municípios (6 dígitos) para seus nomes completos.

Responsabilidades:

1. **Leitura de Arquivos .cnv**:
   - Busca arquivos `Munic*.cnv` e `munic*.cnv` na pasta `TAB_SINANONLINE`
   - Lê arquivos com encoding `latin-1` (compatível com caracteres especiais)
   - Trata erros de encoding com `errors='ignore'`
2. **Parsing de Linhas**:
   - Ignora linhas de comentário (começam com `;`)
   - Ignora linhas vazias
   - Extrai código de município (6 dígitos) e nome
   - Formato esperado: `"1  210530 Imperatriz  210530"`
3. **Criação do Dicionário**:
   - Mapeia código → nome: `{'210530': 'Imperatriz'}`
   - Filtra municípios ignorados
   - Retorna dicionário completo

Função Principal: `load_municipality_dict(cnv_path="TAB_SINANONLINE")`

Processo de Parsing:

```
# Linha de exemplo: "1  210530 Imperatriz  210530"
parts = line.strip().split()  # ['1', '210530', 'Imperatriz', '210530']

# Encontra código de 6 dígitos
for i, part in enumerate(parts):
    if part.isdigit() and len(part) == 6:
```

```
        codigo = part  # '210530'
        nome_parts = parts[i+1:]  # ['Imperatriz', '210530']
        # Remove código repetido no final
        if nome_parts[-1].isdigit() and len(nome_parts[-1]) == 6:
            nome_parts = nome_parts[:-1]
        nome = ' '.join(nome_parts)  # 'Imperatriz'
        break
```

Uso no Sistema:

- Chamado uma vez durante o carregamento de dados
- Resultado é cacheado por 24 horas
- Usado para mapear `ID_MUNICIP` → `MUNICIPIO_NOTIFIC`

Características:

- **Robusto**: Trata erros de encoding e arquivos malformados
- **Eficiente**: Processa apenas uma vez e cacheia
- **Flexível**: Funciona com diferentes formatos de arquivo .cnv

Exemplo de Uso:

```
municip_dict = load_municipality_dict("TAB_SINANONLINE")
# Resultado: {'210530': 'Imperatriz', '355030': 'São Paulo', ...}

# Aplicação no DataFrame
df['MUNICIPIO_NOTIFIC'] = df['ID_MUNICIP'].apply(
    lambda x: municip_dict.get(str(x), str(x))
)
```

---

6. Interação Entre Arquivos

6.1 Fluxo de Dependências

```
dashboard_sinan_real_data.py (Aplicação Principal)

    → sinan_data_processor_comprehensive.py (Processador Tradicional)
        → Usado para: carregar dados, aplicar dicionários, filtrar

    → sinan_data_processor_duckdb.py (Processador Otimizado - Opcional)
        → Usa: sinan_data_processor_comprehensive.py (para dicionários)
        → Usado para: consultas SQL otimizadas

    → munic_dict_loader.py (Carregador de Municípios)
        → Usado para: mapear códigos de municípios para nomes
```

6.2 Decisão de Qual Processador Usar

```python
# Em dashboard_sinan_real_data.py
if use_duckdb and DUCKDB_AVAILABLE:
    # Usa DuckDB (otimizado)
    with SINANDataProcessorDuckDB(...) as processor_duckdb:
        # Carrega dados filtrados diretamente
        child_data_raw = processor_duckdb.load_filtered_violence_data(...)
        # Usa processador tradicional apenas para dicionários
        processor = processor_duckdb.processor
        decoded_data = processor.apply_dictionaries(child_data_raw)
else:
    # Usa Pandas (fallback)
    processor = SINANDataProcessorComprehensive(...)
    violence_data = processor.load_violence_data()
    # Filtra e processa em memória
```

6.3 Pipeline Completo de Processamento

```
1. dashboard_sinan_real_data.py inicia

2. Tenta usar DuckDB (se disponível)
        → sinan_data_processor_duckdb.py
                → Consulta SQL em Parquet
                → Retorna DataFrame filtrado

3. OU usa Pandas (fallback)
        → sinan_data_processor_comprehensive.py
                → Carrega todos os Parquet
                → Filtra em memória

4. Aplica dicionários
        → sinan_data_processor_comprehensive.py.apply_dictionaries()

5. Carrega dicionário de municípios
        → munic_dict_loader.py.load_municipality_dict()

6. Transforma dados
        → Mapeia UF, municípios, cria colunas derivadas

7. Aplica filtros do usuário

8. Gera visualizações
```

---

7. Análise Detalhada dos Arquivos

7.1 `dashboard_sinan_real_data.py` (Arquivo Principal)

Tamanho: ~900 linhas

Estrutura:

Seção 1: Imports e Configuração (linhas 1-117)

- Imports de bibliotecas
- Configuração do Streamlit
- CSS personalizado
- Funções auxiliares
- Dicionários de relacionamento

Função `formatar_numero_br()` (linhas 78-89):

```
def formatar_numero_br(numero):
    """Formata número no padrão brasileiro (ponto para milhares)"""
    # Converte: 1000 → "1.000" (padrão brasileiro)
    # Converte: 1000.5 → "1.000,50"
```

Seção 2: Carregamento de Dados (linhas 119-445)

- Função `load_sinan_data()` com cache
- Lógica de DuckDB vs Pandas
- Transformações de dados

Seção 3: Interface do Dashboard (linhas 447-908)

- Título e informações
- Filtros interativos (sidebar)
- KPIs (métricas principais)
- 10 gráficos interativos

Gráficos implementados:

1. Tendência temporal (linha)
2. Composição por tipo de violência (barras empilhadas)
3. Distribuição por faixa etária e sexo (barras agrupadas)
4. Distribuição geográfica (barras)
5. Local de ocorrência (barras horizontais)
6. Perfil do agressor - sexo (barras)
7. Relacionamento com agressor (barras horizontais)
8. Distribuição por raça/cor (barras)
9. Evolução mensal (linha)
10. Comparação regional (barras)

5.2 `sinan_data_processor_comprehensive.py`

Classe: `SINANDataProcessorComprehensive`

Métodos principais:

`load_dictionaries()` (linhas 29-78)

- Carrega dicionários hardcoded para decodificação
- Mapeia códigos numéricos para valores legíveis
- Exemplo: `'1'` → `'Sim'`, `'4001'` → `'01 ano'`

`load_violence_data()` (linhas 80-105)

- Lista arquivos `.parquet` na pasta `VIOLBR-PARQUET`
- Carrega cada arquivo com `pd.read_parquet()`
- Concatena todos em um único DataFrame
- Retorna DataFrame combinado

`apply_dictionaries()` (linhas 107-161)

- Aplica dicionários apenas nas colunas necessárias
- Usa `copy(deep=False)` para economizar memória
- Processa colunas `REL_*` limitadas (primeiras 5)
- Mantém valores não mapeados com `fillna()`

`filter_comprehensive_violence()` (linhas 163-255)

- Filtra por idade (0-17 anos) se necessário
- Filtra por tipo de violência (qualquer tipo)
- Aceita múltiplos formatos de valores ('1', 'Sim', 1, etc.)
- Tratamento de erro para divisão por zero

5.3 `sinan_data_processor_duckdb.py`

Classe: `SINANDataProcessorDuckDB`

Características:

- Context manager (`__enter__`, `__exit__`)
- Usa DuckDB para consultas SQL em Parquet
- Herda funcionalidades do processador tradicional

`query_with_filters()` (linhas 40-82)

```
def query_with_filters(self, filters=None, columns=None):
    # Constrói query SQL para cada arquivo Parquet
    query = f"SELECT {cols_str} FROM read_parquet('{file_path}')"
    if filters:
        query += " WHERE " + " AND ".join(where_clauses)
```

```
    # Combina com UNION ALL
    full_query = " UNION ALL ".join(queries)
    return self.conn.execute(full_query).df()
```

Vantagens:

- Filtra ANTES de carregar na memória
- Carrega apenas colunas necessárias
- Muito mais rápido para grandes volumes

`load_filtered_violence_data()` (linhas 84-157)

- Descobre colunas disponíveis fazendo query de amostra
- Aplica filtro de idade no SQL
- Aplica filtro de violência em Pandas (após carregar)
- Retorna DataFrame já filtrado

5.4 `munic_dict_loader.py`

Função: `load_municipality_dict()`

Processo:

1. Lista arquivos `Munic*.cnv` na pasta `TAB_SINANONLINE`
2. Para cada arquivo:
   - Lê com encoding `latin-1`
   - Ignora linhas de comentário (`;`)
   - Extrai código (6 dígitos) e nome do município
   - Formato: `"1  210530 Imperatriz  210530"`
3. Retorna dicionário `{codigo: nome}`

Exemplo de linha processada:

```
"1  210530 Imperatriz  210530"
→ Extrai: codigo='210530', nome='Imperatriz'
→ Adiciona ao dict: {'210530': 'Imperatriz'}
```

---

6. Otimizações de Performance

6.1 Otimizações de Memória

1. Filtragem Precoce

```
# ANTES: Carrega tudo, depois filtra
violence_data = load_all_data()  # 2.8M registros
filtered = violence_data[age_filter]  # 500K registros

# DEPOIS: Filtra durante carregamento (DuckDB)
filtered = query_with_filters(age_filter=True)  # 500K registros direto
```

2. Shallow Copies

```
# Usa copy(deep=False) em vez de copy(deep=True)
# Economiza memória ao não duplicar dados
decoded_data = data.copy(deep=False)
```

3. Seleção de Colunas

```
# Mantém apenas colunas essenciais
columns_to_keep = essential_columns + rel_cols
df_subset = df[columns_to_keep].copy(deep=False)
```

4. Liberação Explícita de Memória

```
del violence_data  # Libera memória explicitamente
del child_data_raw
```

6.2 Otimizações de Processamento

1. Cache do Streamlit

```
@st.cache_data(ttl=3600)  # Cache por 1 hora
def load_sinan_data():
    # Dados são carregados apenas uma vez
```

2. Cache de Dicionários

```
@st.cache_data(ttl=86400)  # Cache por 24 horas
def load_municipality_dictionary():
    # Dicionários não mudam, cache longo
```

3. Processamento Lazy

- Dados são processados apenas quando necessário
- Filtros aplicados dinamicamente
- Gráficos gerados sob demanda

6.3 Otimizações de Visualização

1. Limitação de Ticks no Eixo Y

```
# Evita sobreposição de rótulos
num_ticks = min(8, len(df_tendencia))
tick_values = [min_value + i * step for i in range(num_ticks)]
```

2. Limitação de Dados Exibidos

```
# Top 10, Top 20, etc.
df_regional = df_regional.sort_values('Contagem', ascending=False).head(10)
```

3. Formatação Brasileira

- Números formatados com ponto para milhares
- Melhora legibilidade

---

7. Estrutura de Dados

7.1 Formato dos Dados de Entrada (Parquet)

Arquivos: `VIOLBR19.parquet` a `VIOLBR24.parquet`

Colunas principais:

Coluna	Tipo	Descrição	Exemplo
`DT_NOTIFIC`	string/int	Data de notificação	`20240115` ou `2024-01-15`
`NU_ANO`	int/string	Ano da notificação	`2024`
`SG_UF_NOT`	string/int	Código UF (2 dígitos)	`21` ou `'21'`
`ID_MUNICIP`	string	Código município (6 dígitos)	`'210530'`
`NU_IDADE_N`	string	Código de idade	`'4001'` (1 ano)
`CS_SEXO`	string/int	Sexo da vítima	`'1'` (Masculino)
`VIOL_FISIC`	string/int	Violência física	`'1'` (Sim)
`VIOL_PSICO`	string/int	Violência psicológica	`'1'` (Sim)
`VIOL_SEXU`	string/int	Violência sexual	`'1'` (Sim)
`VIOL_INFAN`	string/int	Violência infantil	`'1'` (Sim)
`LOCAL_OCOR`	string	Local de ocorrência	`'01'` (Residência)
`AUTOR_SEXO`	string	Sexo do agressor	`'1'` (Masculino)
`REL_PAI`	string/int	Relação: Pai	`'1'` (Sim)
`REL_MAE`	string/int	Relação: Mãe	`'1'` (Sim)
`REL_*`	string/int	Outros relacionamentos	`'1'` (Sim)
`CS_RACA`	string	Raça/cor	`'1'` (Branca)
`CS_ESCOL_N`	string	Escolaridade	`'01'` (Analfabeto)

7.2 Códigos de Idade (NU_IDADE_N)

Código	Idade
`4000`	Menor de 1 ano
`4001`	1 ano
`4002`	2 anos
…	…
`4017`	17 anos

7.3 Códigos de UF (SG_UF_NOT)

Código	Estado
11	Rondônia
12	Acre
13	Amazonas
...	...
53	Distrito Federal

7.4 Transformações Aplicadas

Colunas criadas durante processamento:

1. `ANO_NOTIFIC`: Extraído de `DT_NOTIFIC` ou `NU_ANO`
2. `UF_NOTIFIC`: Nome do estado (mapeado de código)
3. `MUNICIPIO_NOTIFIC`: Nome do município (mapeado de código)
4. `TIPO_VIOLENCIA`: String combinando tipos (ex: "Física, Sexual")
5. `FAIXA_ETARIA`: Grupos (0-1, 2-5, 6-9, 10-13, 14-17)
6. `SEXO`: Normalizado (Masculino/Feminino)
7. `AUTOR_SEXO_CORRIGIDO`: Sexo do agressor corrigido
8. `GRAU_PARENTESCO`: Relacionamentos extraídos das colunas `REL_*`

---

8. Explicação de Trechos Importantes

8.1 Filtragem por Idade (Códigos Especiais)

```
age_codes = ['4000'] + [f'400{i}' for i in range(1, 18)]
# Gera: ['4000', '4001', '4002', ..., '4017']
```

Por que códigos especiais?

- O SINAN usa códigos específicos para idades de crianças
- `4000` = menor de 1 ano (não é `0`)
- `4001` a `4017` = 1 a 17 anos
- Permite filtragem eficiente antes de decodificar

8.2 Correção de AUTOR_SEXO

```
def map_autor_sexo(val):
    val_str = str(val).upper().strip()
    if val_str in ['1', 'M', 'MASCULINO', 'MAS']:
        return 'Masculino'
    elif val_str in ['2', 'F', 'FEMININO', 'FEM']:
        return 'Feminino'
    else:
        return 'Não informado'  # Filtra valores inválidos
```

Problema resolvido:

- A coluna `AUTOR_SEXO` tinha valores misturados (parentescos, etc.)
- Agora filtra apenas valores válidos de sexo
- Valores inválidos viram "Não informado"

8.3 Extração de Grau de Parentesco

```
def get_relacionamento(row):
    relacionamentos = []
    for col in rel_cols:  # Todas as colunas REL_*
        if col in row.index and pd.notna(row[col]):
            val = str(row[col]).upper().strip()
            if val in ['1', 'SIM', 'S', '1.0']:
                if col in RELACIONAMENTO_DICT:
                    relacionamento = RELACIONAMENTO_DICT[col]
                relacionamentos.append(relacionamento)
    return ', '.join(relacionamentos)
```

Lógica:

- Itera sobre todas as colunas `REL_*` (REL_PAI, REL_MAE, etc.)
- Se valor = '1' (Sim), adiciona o relacionamento
- Combina múltiplos relacionamentos em uma string
- Usa dicionário para nomes completos

8.4 Formatação Brasileira de Números

```
def formatar_numero_br(numero):
    num = float(numero)
    if num == int(num):
        return f"{int(num):,}".replace(",", ".")  # 1000 → "1.000"
    else:
        return f"{num:,.2f}".replace(",", "X").replace(".", ",").replace("X", ".")
        # 1000.5 → "1.000,50"
```

Técnica:

- Usa formatação padrão (vírgula para milhares)
- Substitui vírgula por ponto (padrão brasileiro)
- Para decimais, troca vírgula e ponto

8.5 Consulta SQL com DuckDB

```
query = f"SELECT {cols_str} FROM read_parquet('{file_path}')"
if filters:
    query += " WHERE NU_IDADE_N IN ('4000', '4001', ...)"
full_query = " UNION ALL ".join(queries)  # Combina todos os arquivos
result = self.conn.execute(full_query).df()
```

Vantagens:

- Filtra no nível do arquivo (não carrega tudo)
- Combina múltiplos arquivos com UNION ALL
- Retorna apenas dados necessários

---

9. Estratégias de Tratamento de Erros

9.1 Tratamento de Dados Faltantes

```
if pd.isna(val):
    return 'Não informado'
```

9.2 Tratamento de Tipos Múltiplos

```
# Aceita string e numérico
if val_str in ['1', 'SIM', 'S', '1.0']:
    # String
if df[col].dtype in ['int64', 'float64']:
    col_num = (df[col] == 1)  # Numérico
```

9.3 Divisão por Zero

```
if original_count > 0:
    percentage = (filtered_count/original_count*100)
else:
    print("Dados já filtrados anteriormente")
```

9.4 Fallback de Métodos

```
try:
    # Tenta DuckDB
    if DUCKDB_AVAILABLE:
        # Usa DuckDB
except:
    # Fallback para Pandas
    processor = SINANDataProcessorComprehensive(...)
```

---

10. Métricas e KPIs

10.1 KPIs Calculados

1. **Total de Notificações**: `len(df_filtrado)`
2. **Média Anual**: `df_filtrado.groupby('ANO_NOTIFIC').size().mean()`
3. **Tipo Mais Frequente**: `df_filtrado['TIPO_VIOLENCIA'].mode()[0]`

4. **Sexo Mais Frequente**: `df_filtrado['SEXO'].value_counts().index[0]`

10.2 Agregações por Gráfico

- **Tendência Temporal**: `groupby('ANO_NOTIFIC').size()`
- **Composição**: `groupby(['ANO_NOTIFIC', 'TIPO_VIOLENCIA']).size()`
- **Demografia**: `groupby(['FAIXA_ETARIA', 'SEXO']).size()`
- **Geográfica**: `groupby('UF_NOTIFIC').size()`

---

11. Conformidade LGPD

11.1 Anonimização

- Dados são **agregados** (não individuais)
- Nenhum dado pessoal identificável é exibido
- Apenas estatísticas e contagens

11.2 Dados Sensíveis

- Informações sobre violência são sensíveis
- Dashboard usa apenas dados agregados
- Não expõe informações individuais

---

12. Performance e Escalabilidade

12.1 Volumes de Dados

- **Total de registros**: ~2.8 milhões (2019-2024)
- **Após filtragem**: ~500K-1M (depende dos dados)
- **Memória necessária**: ~2-4 GB (sem DuckDB), ~500MB-1GB (com DuckDB)

12.2 Tempo de Processamento

- **Carregamento inicial**: 30-60 segundos (sem DuckDB), 5-10 segundos (com DuckDB)
- **Aplicação de filtros**: < 1 segundo
- **Geração de gráficos**: < 2 segundos cada

12.3 Otimizações Aplicadas

1. Filtragem precoce (idade e violência)
2. Seleção de colunas essenciais
3. Shallow copies
4. Cache do Streamlit

5. DuckDB para consultas SQL
6. Liberação explícita de memória

---

13. Limitações e Considerações

13.1 Limitações Conhecidas

1. **Memória**: Requer pelo menos 4GB RAM (sem DuckDB)
2. **Tempo de carregamento**: Primeira carga pode ser lenta
3. **Filtros de município**: Limitados a 100 para performance
4. **Dicionários**: Alguns valores podem não estar mapeados

13.2 Melhorias Futuras Possíveis

1. Implementar paginação para grandes volumes
2. Adicionar mais dicionários de decodificação
3. Implementar cache de resultados de gráficos
4. Adicionar exportação de dados filtrados

---

14. Conclusão

O dashboard SINAN é uma aplicação robusta e otimizada para análise de grandes volumes de dados de violência contra crianças e adolescentes. Utiliza técnicas avançadas de processamento de dados, otimizações de memória e performance, e uma interface interativa moderna.

Principais conquistas:

- Processa milhões de registros eficientemente
- Interface interativa e responsiva
- 10 hipóteses de pesquisa respondidas
- Conformidade com LGPD
- Otimizações de performance (DuckDB)
- Formatação brasileira de dados
- Tratamento robusto de erros

Tecnologias-chave:

- Streamlit para interface
- Pandas para manipulação de dados
- Plotly para visualizações
- DuckDB para otimização (opcional)
- Parquet para armazenamento eficiente

Documento gerado em: 2024 **Versão do código analisado**: Final (com todas as otimizações)