Capítulo 12 – Engenharia de Chunking e Ingestão de Dados

Aléssio Miranda Jr.

Setembro - 2026/2

Capítulo em Revisão

Este conteúdo está sendo revisado. Algumas informações podem estar preliminares, conter erros de formatação ou sofrer alterações na versão final.

O que você verá neste capítulo

O Vector Store está pronto, mas os documentos do mundo real não. PDFs corporativos, contratos jurídicos, artigos científicos e manuais técnicos são textos longos e desestruturados que não cabem em um único vetor de Embedding. Neste capítulo, formalizaremos a **Engenharia de Chunking** — a ciência de fatiar textos longos em pedaços digeríveis preservando o contexto semântico —, e construiremos uma pipeline de ingestão de documentos completa: da extração de texto de PDFs até o povoamento do banco vetorial.

1 O Problema do Texto Longo

Os modelos de Embedding possuem um **limite de tokens** por entrada. O modelo `all-MiniLM-L6-v2` aceita no máximo 256 tokens (~200 palavras). Modelos mais recentes como o `text-embedding-3-large` da OpenAI aceitam até 8.192 tokens, mas mesmo assim, um livro de 300 páginas excede qualquer limite.

Além do limite técnico, há um problema semântico: quanto mais longo o texto, mais diluído fica o vetor resultante. Um Embedding de um livro inteiro capturará "um pouco de tudo", sem representar nenhum conceito específico com precisão. O resultado prático é que a busca semântica retornará o livro para *qualquer* consulta, sem nunca ser realmente relevante.

A solução é o **Chunking**: dividir o documento em pedaços (*chunks*) menores, cada um com foco semântico suficiente para gerar um Embedding preciso e discriminativo.

2 Estratégias de Chunking

Para que o banco vetorial funcione com alta precisão, precisamos fatiar o texto do mundo real em pedaços (*Chunks*) digeríveis. Mas como fatiar? A escolha da estratégia tem impacto direto na qualidade da busca:

- **Fatiamento por Caracteres Fixos:** Cortar a cada 1000 caracteres. É o método mais simples, mas o mais perigoso: pode cortar uma palavra no me…io, destruindo frases e quebrando o sentido.
- **Fatiamento por Sentenças (spaCy/NLTK):** Separar por pontos finais. Preserva a integridade gramatical, mas gera chunks de tamanhos muito variáveis.
- **Fatiamento Recursivo por Parágrafo:** O método mais robusto e adotado pela indústria. Tenta cortar em quebras de linha dupla (`\n\n`), preservando blocos lógicos. Se o bloco resultante ainda for grande demais, divide recursivamente em sentenças.

▶ Prática: title=RecursiveCharacterTextSplitter (LangChain)

```python
from langchain.text_splitter import (
    RecursiveCharacterTextSplitter
)

splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,      # Tamanho maximo do chunk
    chunk_overlap=200,    # Sobreposicao entre chunks
    separators=["\n\n", "\n", ". ", " ", ""]
)

chunks = splitter.split_text(texto_completo)
print(f"Documento dividido em {len(chunks)} chunks")
```

A lista de `separators` define a hierarquia de corte: tenta primeiro por parágrafo, depois por linha, depois por sentença, e só em último caso por caractere.

3 O Perigo da Fronteira (Chunk Overlap)

Imagine que um conceito crucial comece na última linha do Chunk 1 e termine na primeira linha do Chunk 2. Se cortarmos exatamente na fronteira,

quebramos o sentido da frase e o banco vetorial jamais encontrará a correlação completa.

A solução na engenharia de dados é o **Chunk Overlap** (Sobreposição). Se o nosso pedaço tem 1000 caracteres, fazemos com que os últimos 200 caracteres dele sejam **repetidos no início do próximo pedaço**. Essa técnica garante que o contexto faça uma "ponte" segura entre as quebras, impedindo a perda de raciocínio.

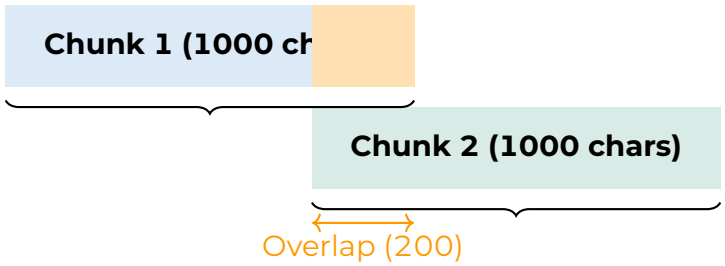


Figura 1: Visualização do Chunk Overlap: os últimos 200 caracteres do Chunk 1 são repetidos no início do Chunk 2, criando uma "ponte" de contexto.

△ Importante: title=Overlap Excessivo

Um overlap muito grande (ex: 500 de 1000) gera redundância massiva no banco vetorial: os mesmos trechos aparecem em múltiplos chunks, inflando o custo de armazenamento e confundindo o ranking. A regra prática da indústria é manter o overlap entre **10% e 20%** do tamanho do chunk.

4 Bibliotecas de Extração de Texto

Antes de fatiar, precisamos *extrair* o texto dos formatos do mundo real. PDFs não são arquivos de texto amigáveis — eles são essencialmente instruções de **pintura de tela** (coordenadas de onde desenhar cada glifo). Extrair texto de PDFs corporativos exige bibliotecas específicas:

- **PyPDF2 / pypdf:** Biblioteca leve e clássica para PDFs limpos e nativos (gerados digitalmente). Rápida, mas falha em layouts complexos.
- **pdfplumber:** Mais robusta, capaz de lidar com formatações complexas, colunas múltiplas e tabelas. Muito usada em auditoria e compliance.
- **Unstructured:** Framework que unifica a extração de PDFs, DOCXs, HTMLs, e-mails e imagens em uma pipeline padronizada.
- **Tesseract (OCR):** Necessário quando o PDF é, na verdade, uma imagem escaneada. O OCR (*Optical Character Recognition*) converte a imagem em texto reconhecível.

5 A Pipeline Completa de Ingestão

A combinação de extração, chunking e inserção no Vector Store forma a **Pipeline de Ingestão**:

1. **Extração:** Ler o PDF com `pypdf` ou `pdfplumber` → texto bruto.
2. **Limpeza:** Remover cabeçalhos/rodapés repetidos, números de página, e caracteres de controle.
3. **Chunking:** Fatiar com `RecursiveCharacterTextSplitter` com overlap.
4. **Embedding:** Gerar o vetor de cada chunk via modelo de Embedding.
5. **Inserção:** Armazenar vetor + texto + metadados (nome do arquivo, página, data) no ChromaDB.

Nesta aula, montaremos o motor responsável por ler o mundo real, mastigar em blocos e povoar nossa infraestrutura para preparar o terreno para a Geração Aumentada (RAG).

✔ Takeaways

- Documentos longos (PDFs, livros) ultrapassam o limite de tokens e geram vetores diluídos.
- **Chunking** é a técnica de fatiar o texto preservando a integridade semântica de cada pedaço.
- **Chunk Overlap** é vital: repetir o final de um chunk no início do próximo impede a quebra de raciocínio. Um overlap de 10-20% é o padrão.
- **RecursiveCharacterTextSplitter** tenta cortar grandes textos priorizando quebras naturais (parágrafos, depois sentenças, depois palavras).
- A **Pipeline de Ingestão** envolve: Extração → Limpeza → Chunking → Embedding → Inserção no Vector DB.

✎ Questões: Autoavaliação

1 **[Direta]**Por que gerar um único Embedding para um documento de 20 páginas resulta em péssimo desempenho na busca semântica?

2 **[Direta]**Explique qual é a principal vantagem do particionamento recursivo (ex: `RecursiveCharacterTextSplitter`) em comparação ao fatiamento de comprimento fixo (ex: cortar a cada 1000 caracteres sem critério).

3 **[Direta]**O que é o Chunk Overlap e qual problema prático ele previne durante a fase de busca no Vector DB?

4 **[Reflexiva]**Se você usar um *overlap* excessivo (ex: chunks de 500 caracteres com overlap de 400), quais seriam as consequências diretas na infraestrutura e nos resultados do banco vetorial?

5 **[Reflexiva]**Ao processar contratos jurídicos digitalizados (onde o texto em si é uma imagem escaneada), a simples biblioteca `pypdf` falha. Como a pipeline de ingestão precisaria ser adaptada para resolver isso e quais custos adicionais (tempo/processamento) seriam inseridos?

Lab. 12: Pipeline de Ingestão de PDFs

Capítulo em Revisão

Este conteúdo está sendo revisado. Algumas informações podem estar preliminares, conter erros de formatação ou sofrer alterações na versão final.

⋆ Objetivo do Laboratório

No Lab 11, seu banco vetorial recebeu frases curtas digitadas à mão. Na vida real, o conhecimento corporativo está trancado em **PDFs de centenas de páginas**. Nesta atividade, você construirá um **pipeline ETL completo** (Extract, Transform, Load) que: extrai texto de um PDF normativo, aplica uma estratégia de *Chunking* com sobreposição para preservar contexto entre fatias, injeta os fragmentos no ChromaDB com metadados de página, e realiza buscas semânticas sobre o documento fatiado.

6 Parte 1: O Problema L — Estilo Maratona

Problema L: O Engenheiro de Dados da DocuMind

Contexto: A *DocuMind*, uma legaltech que automatiza due diligence, recebeu um desafio do cliente: ingerir o **Regimento de Estágio da universidade** (PDF com 10+ páginas) no banco vetorial e permitir que o chatbot responda perguntas como "qual a carga horária mínima de estágio?" sem que o texto completo seja enviado ao LLM.
O pipeline precisa resolver dois problemas críticos:

1. **Extrair** texto de PDF (com tratamento de páginas vazias)
2. **Fatiar** o texto em chunks que preservem contexto semântico (overlap)

Modelo de Entrada: Um PDF com ≥ 5 páginas.
Modelo de Saída: Coleção ChromaDB com $N > 1$ chunks indexados.

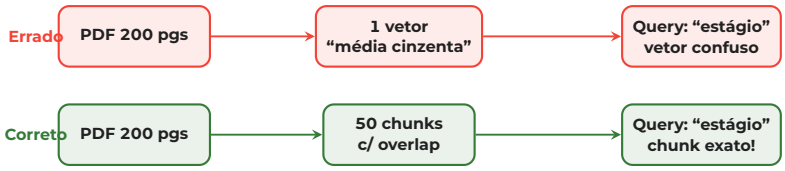
Critérios de Aprovação:

1. Texto extraído do PDF com > 100 caracteres
2. Chunking produz $N > 1$ fragmentos
3. Todos os chunks injetados no ChromaDB sem erro
4. Busca semântica retorna chunk relevante à query

7 Parte 2: Entendendo o Problema — Por que Não Vetorizar o PDF Inteiro?

Antes de codar qualquer coisa, é fundamental entender **por que** este laboratório existe. No Lab 11, você inseriu frases curtas no ChromaDB e tudo funcionou perfeitamente. Mas o que acontece quando o documento tem **200 páginas**?

Imagine que o Regimento de Estágio fale de "carga horária mínima" na página 3, "avaliação do orientador" na página 12 e "desligamento por reprovação" na página 18. Se você vetorizar o documento inteiro como um único vetor de 384 dimensões, o resultado será uma **média cinzenta** — um vetor que não aponta para nenhum desses temas com precisão. É como misturar todas as tintas de um estojo e obter uma pasta marrom sem identidade.



• Teoria: As Duas Restrições Fundamentais

1. **Janela de Contexto (Tokens):** Quando o RAG encontrar o trecho mais relevante, ele enviará esse trecho ao LLM para gerar a resposta. Se o trecho for o PDF inteiro, a API do ChatGPT negará a requisição por exceder o limite de tokens — ou cobrará dezenas de dólares por uma única pergunta.
2. **Perda de Resolução Semântica:** Um vetor de 384 dimensões tem capacidade limitada de representação. Se o documento aborda culinária, direito e medicina em capítulos diferentes, o vetor resultante não apontará para nenhum desses temas com precisão. A “direção” se perde.

A solução é o **Chunking**: fatiar o documento longo em pedaços menores, onde cada pedaço se torna uma entidade independente no banco vetorial. A query “carga horária de estágio” encontrará exatamente o chunk que fala sobre isso, não o documento inteiro.

8 Parte 3: Instalação do Ambiente (5 min)

Para este laboratório, precisaremos de duas bibliotecas: a **pypdf** para extrair texto de PDFs, e o **chromadb** que já conhecemos do Lab 11. Abra o terminal e instale:

```
pip install pypdf chromadb
```

Além disso, coloque na pasta do projeto um arquivo `documento.pdf` com pelo menos 5 páginas. Pode ser o Regimento de Estágio da sua universidade, uma apostila, ou qualquer documento normativo que você encontre no site institucional. O importante é que seja um PDF com texto extraível (não um PDF escaneado como imagem).

9 Parte 4: Extraindo Texto do PDF (10 min)

A. Entendendo a Extração

A biblioteca `pypdf` abre o arquivo PDF e percorre cada página, extraindo o texto como uma string Python. Nem toda página terá texto — páginas com apenas imagens ou gráficos retornarão `None`. Por isso, precisamos verificar se `extract_text()` retornou algo antes de concatenar.

Crie o arquivo `src/extrator_pdf.py` e comece pela etapa de extração:

```python
from pypdf import PdfReader
import chromadb

# 1. Leitura do PDF --- abrindo o documento pagina a pagina
print("Extraindo texto do PDF...")
leitor = PdfReader("documento.pdf")
texto_total = ""

for i, pagina in enumerate(leitor.pages):
    texto_pagina = pagina.extract_text()
    if texto_pagina:  # Ignora paginas sem texto (imagens)
        texto_total += texto_pagina + " "
        print(f"  Pagina {i+1}: {len(texto_pagina)} caracteres")

print(f"\nTotal extraido: {len(texto_total)} caracteres")
print(f"Total de paginas: {len(leitor.pages)}")
```

Rode o script e observe a saída. Cada página contribui com uma quantidade diferente de caracteres. Se alguma página aparecer com 0 caracteres, provavelmente é uma capa com apenas logotipos — isso é normal.

B. O que Pode Dar Errado?

△ Importante: PDFs Escaneados vs. Digitais

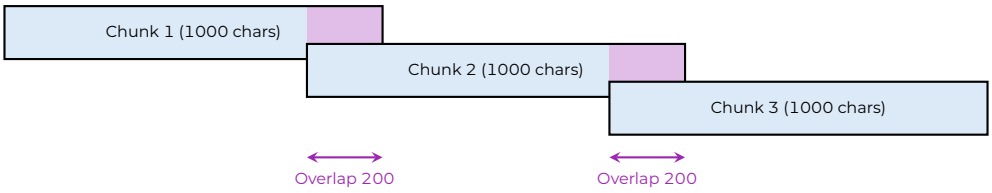
Se o seu PDF for uma **imagem escaneada** (como uma foto de um documento antigo), o `extract_text()` retornará string vazia. Para esses casos, seria necessário OCR (ex: Tesseract) — mas isso está fora do escopo deste lab. Use um PDF digital com texto selecionável.

10 Parte 5: A Engenharia do Chunking (15 min)

A. O Conceito de Overlap (Sobreposição)

Agora vem a parte mais crítica do pipeline: decidir **como fatiar** o texto. A técnica mais simples é o *Fixed-Size Chunking* — cortar a cada N caracteres. Mas um corte cego pode separar "infarto agudo do" (Chunk 1) de "miocárdio grave" (Chunk 2), destruindo o significado da frase.

A solução é a **sobreposição (overlap)**: cada chunk "invade" os 200 caracteres finais do chunk anterior. Assim, se o corte acontecer no meio de uma frase, ela aparecerá completa no chunk seguinte.



A região roxa na figura acima mostra o trecho que é repetido entre chunks adjacentes. Essa redundância controlada é o preço que pagamos para garantir que nenhuma informação seja "cortada ao meio".

B. Implementando a Função de Chunking

Adicione a função ao seu `src/extrator_pdf.py`:

```python
# 2. Funcao de Chunking com Overlap
def quebrar_texto(texto_completo, tamanho_chunk=1000, overlap=200):
    """Fatia texto em chunks com sobreposição.

    Argumentos:
        texto_completo: string com todo o texto extraido
        tamanho_chunk: tamanho maximo de cada fatia (em chars)
        overlap: quantos chars do final de um chunk se repetem
                 no inicio do proximo (evita corte no meio de frase)
    """
    chunks = []
    inicio = 0
    while inicio < len(texto_completo):
        fim = inicio + tamanho_chunk
        pedaco = texto_completo[inicio:fim]
        chunks.append(pedaco)
        # Avanca o ponteiro, subtraindo o overlap
        inicio = fim - overlap
    return chunks
```

Observe a lógica do ponteiro: ao invés de avançar `tamanho_chunk` posições a cada iteração, avançamos `tamanho_chunk − overlap`. Isso garante que os últimos 200 caracteres do Chunk k reapareçam nos primeiros 200 caracteres do Chunk $k+1$.

C. Aplicando o Chunking ao Texto Extraído

```python
# 3. Aplicando o fatiamento
print("\nAplicando Chunking...")
lista_de_chunks = quebrar_texto(texto_total)
print(f"Documento fatiado em {len(lista_de_chunks)} fragmentos")
print(f"Tamanho medio por chunk: "
      f"{sum(len(c) for c in lista_de_chunks)//len(lista_de_chunks)}
         chars")

```

```
# Visualizando os 3 primeiros chunks (preview)
for i, chunk in enumerate(lista_de_chunks[:3]):
    preview = chunk[:80].replace('\n', ' ')
    print(f"\n  Chunk {i}: \"{preview}...\"")
```

Rode o script novamente. Você deve ver algo como "Documento fatiado em 15 fragmentos". Se o número for 1, seu PDF provavelmente tem menos de 1000 caracteres — use um documento maior. Se for muito alto (> 100), considere aumentar o `tamanho_chunk`.

• Teoria: Como Escolher o Tamanho do Chunk?

Não existe um valor mágico, mas a tabela abaixo resume as boas práticas do mercado:

Tamanho	Overlap	Trade-off
200 chars	50	Alta granularidade, *muitos* chunks
1000 chars	**200**	**Equilíbrio (recomendado para RAG)**
3000 chars	500	Poucos chunks, perde detalhes finos

O valor ideal depende do domínio: textos jurídicos densos pedem chunks menores; artigos científicos com parágrafos longos podem usar chunks maiores.

11 Parte 6: Ingestão no ChromaDB (10 min)

A. Criando a Coleção e Injetando os Chunks

Com os chunks prontos, vamos injetá-los no ChromaDB. Cada chunk se torna um documento independente no banco vetorial, com metadados que registram sua posição no texto original:

```
# 4. Ingestao no VectorDB
print("\nInjetando no ChromaDB...")
cliente = chromadb.PersistentClient(path="./chroma_pdf")
colecao = cliente.get_or_create_collection(
    name="pdf_rag",
    metadata={"hnsw:space": "cosine"}
)

# Metadados: indice do chunk e posicao aproximada no texto
metadatas = [{"chunk_idx": i, "posicao": i * 800}
             for i in range(len(lista_de_chunks))]

colecao.upsert(
    documents=lista_de_chunks,
    metadatas=metadatas,
    ids=[f"chunk_{i:03d}" for i in range(len(lista_de_chunks))]
```

```
)

print(f"Total de chunks na base: {colecao.count()}")
print("Processo de ETL concluido com sucesso!")
```

O `upsert()` garante idempotência: se você rodar o script duas vezes, os chunks são atualizados, não duplicados. Note que o ChromaDB gera automaticamente os embeddings de cada chunk usando o modelo `all-MiniLM-L6-v2` embutido — você não precisa chamar o spaCy.

B. Verificando a Ingestão

Após rodar o script, a pasta `chroma_pdf/` aparecerá no seu diretório de trabalho contendo os binários do banco. Essa pasta **não deve** ser commitada no Git (adicionaremos ao `.gitignore` na Parte 8).

12 Parte 7: Busca Semântica no PDF Fatiado (5 min)

Agora vem o momento da verdade: o buscador encontra o chunk correto mesmo sem keywords?

```
# 5. Busca semantica nos chunks do PDF
def buscar_pdf(query, top_k=3):
    """Busca os top_k chunks mais relevantes para a query."""
    return colecao.query(
        query_texts=[query], n_results=top_k)

if __name__ == "__main__":
    queries = [
        "Qual a carga horaria minima de estagio?",
        "Quais sao os direitos do estagiario?",
        "Como funciona a avaliacao do estagio?",
    ]
    for q in queries:
        print(f"\n{'='*60}")
        print(f"QUERY: '{q}'")
        resultados = buscar_pdf(q, top_k=2)
        for i, (doc, dist) in enumerate(
                zip(resultados['documents'][0],
                    resultados['distances'][0])):
            preview = doc[:120].replace('\n', ' ')
            print(f"  #{i+1} [Dist: {dist:.4f}] {preview}...")
```

Observe os resultados: a query “carga horária mínima de estágio” deve retornar o chunk que contém essa informação no Regimento, mesmo que a redação use sinônimos como “período obrigatório” ou “horas-aula”. Essa é a

magia da busca semântica que construímos nos Labs 10 e 11, agora aplicada a documentos reais.

▷ Exemplo: title=Adaptando para Seu PDF

Se você usou um PDF diferente do Regimento de Estágio, modifique as queries de teste para perguntas relevantes ao conteúdo do seu documento. O importante é verificar que o buscador retorna o trecho correto.

13 Parte 8: Garantia de Qualidade — Testes CI/CD (5 min)

Os testes verificam cada etapa do pipeline ETL. Crie o arquivo `tests/test_extrator_pdf.py`:

```python
from src.extrator_pdf import (
    quebrar_texto, texto_total, lista_de_chunks, colecao)

def test_texto_extraido():
    """PDF deve ter gerado texto com > 100 caracteres."""
    assert len(texto_total) > 100, \
        f"Texto muito curto: {len(texto_total)} chars"

def test_chunking_produz_multiplos():
    """Chunking deve gerar mais de 1 fragmento."""
    assert len(lista_de_chunks) > 1, \
        f"Apenas {len(lista_de_chunks)} chunk(s)"

def test_chunks_injetados():
    """ChromaDB deve conter todos os chunks."""
    assert colecao.count() == len(lista_de_chunks)

def test_busca_retorna_resultado():
    """Busca semantica deve retornar pelo menos 1 resultado."""
    r = colecao.query(query_texts=["estagio"], n_results=1)
    assert len(r['documents'][0]) == 1
```

Cada teste valida uma etapa diferente: extração (>100 chars), chunking (>1 pedaço), ingestão (contagem correta), e busca (resultado não vazio). Rode com `pytest tests/test_extrator_pdf.py -v`. Os **4 testes** devem aparecer em verde.

14 Parte 9: Commit e Push (5 min)

Antes de enviar, adicione a pasta do banco ao `.gitignore` para não commitar binários pesados:

```
echo "chroma_pdf/" >> .gitignore
git add src/extrator_pdf.py tests/test_extrator_pdf.py documento.pdf
    .gitignore
git commit -m "Lab 12: Pipeline ETL PDF + Chunking + ChromaDB + CI/CD
    "
git push origin main
```

O GitLab CI interceptará seu push e executará o `pytest` automaticamente. O **Selo Verde** confirma sua nota.

Conclusão: O que Você Construiu Hoje

✔ Takeaways

- Você construiu um **pipeline ETL completo** (Extract → Transform → Load) que transforma um PDF de múltiplas páginas em chunks pesquisáveis no ChromaDB.
- Entendeu por que **vetorizar o documento inteiro é uma péssima ideia**: o vetor perde resolução semântica e estoura a janela de contexto do LLM.
- Implementou **Chunking com sobreposição** (200 chars de overlap) que preserva o contexto semântico entre fatias adjacentes — a técnica que separa um RAG amador de um RAG profissional.
- Realizou **buscas semânticas sobre um PDF real**, encontrando trechos relevantes sobre estágio sem digitar nenhuma keyword.
- Os 4 testes CI/CD validam cada etapa do pipeline: extração, chunking, ingestão e busca.

◇ **Informação**

Gancho para a Próxima Aula: Seu pipeline de ingestão está pronto — o "lado esquerdo" do RAG (retrieval). Mas quem processa a query do "lado direito"? Na próxima aula, abriremos o capô do **modelo Transformer** e visualizaremos o mecanismo de **Self-Attention** — a peça matemática que permite ao modelo entender que, na frase "o cachorro não atravessou a rua porque *ele* estava cansado", a palavra "ele" se refere ao "cachorro", não à "rua". Essa é a revolução que gerou o GPT, o BERT e todos os LLMs modernos.