Lab. 11: Banco de Dados Vetorial com ChromaDB

Aléssio Miranda Júnior

Setembro - 2026/2

Lab. 11: Banco de Dados Vetorial com ChromaDB

⋆ Objetivo do Laboratório

No Lab 10, seu buscador semântico usava um laço `for` que varria N documentos: $O(N \times D)$. Para 500 milhões de parágrafos, isso levaria **minutos** por query. Nesta atividade, você abandonará o laço artesanal e adotará a infraestrutura profissional **ChromaDB**. Usando o algoritmo HNSW, a busca cai para $O(\log N)$ — milissegundos ao invés de minutos. Você inicializará um Banco de Dados Vetorial persistente, injetará um corpus de 20 fatos históricos com metadados filtráveis, e realizará buscas semânticas com Top-K e filtros SQL-like.

1 Parte 1: O Problema K — Estilo Maratona

Problema K: A Memória Enciclopédica da HistoriAI

Contexto: A *HistoriAI*, uma edtech de ensino de História, quer construir um "chatbot enciclopédico" que responda perguntas sobre fatos históricos. O problema: o corpus tem 20 fatos espalhados por 4 categorias (Ciência, Tecnologia, Política, Cultura) e séculos (XVI ao XXI). A busca por keywords falha miseravelmente — "descobertas que salvaram vidas" não contém a palavra "penicilina".
O CTO exige que a solução use um **Banco de Dados Vetorial profissional** (ChromaDB) com:

1. Busca semântica Top-K sem nenhum laço `for`
2. Filtros por metadados (categoria, século) — estilo SQL
3. Busca combinada: semântica + filtro categórico

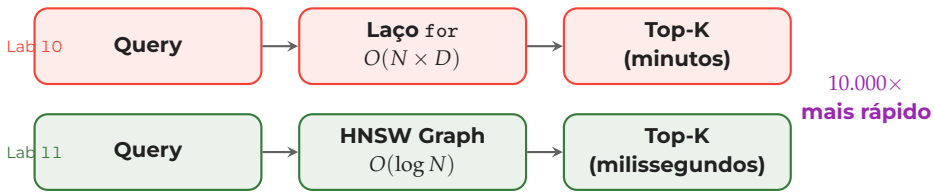
Critérios de Aprovação:

1. Coleção com exatamente 20 documentos inseridos
2. Query Top-2 retorna exatamente 2 resultados
3. Query "animais domésticos" retorna doc com "gato" ou "cachorro" no Top-1

2 Parte 2: Entendendo o Salto — Do Laço For ao HNSW

No Lab 10, seu buscador varreva o corpus inteiro para cada query. Funciona para 10 frases, mas imagine um banco com **500 milhões de documentos** (a escala real do Google ou da Wikipedia). O laço `for` levaria minutos por query — inviável para um produto real.

Os **Bancos de Dados Vetoriais** resolvem isso com o algoritmo **HNSW** (Hierarchical Navigable Small World): ao invés de comparar todos os vetores, ele constrói um grafo de vizinhança e "navega" por atalhos hierárquicos, chegando ao vizinho mais próximo em $O(\log N)$.



O ChromaDB é um banco vetorial open-source que encapsula o HNSW, gera embeddings automaticamente, e persiste os dados em disco — tudo isso em menos de 10 linhas de Python.

3 Parte 3: Instalação e Inicialização (5 min)

A. Instalando o ChromaDB

Instale o ChromaDB. Ele já inclui seu próprio motor de embeddings embarcado (`all-MiniLM-L6-v2`), não precisaremos mais do spaCy:

```
pip install chromadb
```

O modelo `all-MiniLM-L6-v2` é um Transformer leve (22M parâmetros) que gera vetores de 384 dimensões. Ele é baixado automaticamente na primeira execução e fica em cache local.

B. Criando o Cliente e a Coleção

Crie o arquivo `src/banco_vetorial.py`. A primeira etapa é instanciar um cliente **persistente** — ou seja, os dados ficam salvos em disco e sobrevivem ao reinício do script:

```
import chromadb

# 1. Instancia um Cliente Persistente do ChromaDB
# (Cria uma pasta local com os binarios do banco)
cliente = chromadb.PersistentClient(path="./chroma_db")

# 2. Cria uma Colecao (semelhante a uma Tabela SQL)
# Configuracao: metrica cosseno + espaco HNSW
colecao = cliente.get_or_create_collection(
    name="fatos_historicos",
    metadata={"hnsw:space": "cosine"}
)

print(f"Colecao criada: '{colecao.name}'")
print(f"Documentos existentes: {colecao.count()}")
```

O conceito de **Collection** é análogo a uma tabela SQL: é onde os documentos serão armazenados e indexados. O parâmetro `hnsw:space: cosine` diz ao ChromaDB para usar a mesma Similaridade de Cosseno que implementamos manualmente no Lab 10.

• Teoria: O que Acontece por Baixo do Capô

Ao especificar `hnsw:space: cosine`, o ChromaDB faz três coisas automaticamente:

1. **Normaliza** cada vetor para comprimento unitário (L2) durante o INSERT — assim, o cosseno vira um simples dot product.
2. **Constrói o grafo HNSW** com $M = 16$ conexões por nó — uma rede de "atalhos" para navegação rápida.
3. **Armazena** metadados em SQLite e vetores em binários otimizados no diretório `chroma_db/`.

A busca posterior é uma travessia no grafo: $O(\log N)$ ao invés do $O(N)$ do seu laço `for`.

4 Parte 4: Injetando o Corpus com Metadados (10 min)

A. Preparando os Dados

Agora vamos popular o banco com 20 fatos históricos organizados por **categoria** e **século**. Esses metadados permitirão filtros SQL-like na busca:

```
# 3. Corpus de fatos historicos com metadados categoricos
fatos = [
    # --- CIENCIA ---
    ("A penicilina foi descoberta por Fleming em 1928.",
     {"categoria": "ciencia", "seculo": "XX"}),
    ("Einstein publicou a relatividade geral em 1915.",
     {"categoria": "ciencia", "seculo": "XX"}),
    ("Watson e Crick descreveram o DNA em 1953.",
     {"categoria": "ciencia", "seculo": "XX"}),
    ("Galileu observou as luas de Jupiter em 1610.",
     {"categoria": "ciencia", "seculo": "XVII"}),
    ("A vacina contra variola foi criada por Jenner em 1796.",
     {"categoria": "ciencia", "seculo": "XVIII"}),
    # --- TECNOLOGIA ---
    ("O primeiro computador ENIAC foi ligado em 1945.",
     {"categoria": "tecnologia", "seculo": "XX"}),
    ("A internet comercial chegou ao Brasil em 1995.",
     {"categoria": "tecnologia", "seculo": "XX"}),
    ("O iPhone foi lancado pela Apple em 2007.",
     {"categoria": "tecnologia", "seculo": "XXI"}),
    ("O ChatGPT foi lancado pela OpenAI em 2022.",
     {"categoria": "tecnologia", "seculo": "XXI"}),
    ("Alan Turing criou a maquina universal em 1936.",
```

```
        {"categoria": "tecnologia", "seculo": "XX"}),
    # --- POLITICA ---
    ("A Revolucao Francesa ocorreu em 1789.",
     {"categoria": "politica", "seculo": "XVIII"}),
    ("O Muro de Berlim caiu em novembro de 1989.",
     {"categoria": "politica", "seculo": "XX"}),
    ("A Declaracao de Independencia dos EUA e de 1776.",
     {"categoria": "politica", "seculo": "XVIII"}),
    ("O Brasil se tornou republica em 1889.",
     {"categoria": "politica", "seculo": "XIX"}),
    ("A ONU foi fundada em 1945 apos a Segunda Guerra.",
     {"categoria": "politica", "seculo": "XX"}),
    # --- CULTURA ---
    ("Leonardo da Vinci pintou a Mona Lisa em 1503.",
     {"categoria": "cultura", "seculo": "XVI"}),
    ("Beethoven compos a Nona Sinfonia em 1824.",
     {"categoria": "cultura", "seculo": "XIX"}),
    ("Os Jogos Olimpicos modernos comecaram em 1896.",
     {"categoria": "cultura", "seculo": "XIX"}),
    ("Shakespeare escreveu Hamlet por volta de 1600.",
     {"categoria": "cultura", "seculo": "XVII"}),
    ("A Copa do Mundo de futebol comecou em 1930.",
     {"categoria": "cultura", "seculo": "XX"}),
]
```

B. Inserindo com Upsert

O `upsert()` insere se o ID não existe, ou atualiza se já existe — isso garante **idempotência**: você pode rodar o script 10 vezes sem duplicar dados.

```
# 4. Inserindo no ChromaDB com upsert (idempotente)
print("Injetando dados na base...")
colecao.upsert(
    documents=[f[0] for f in fatos],
    metadatas=[f[1] for f in fatos],
    ids=[f"fato_{i:02d}" for i in range(len(fatos))]
)
print(f"Total de documentos na base: {colecao.count()}")
```

Note que você **não precisou gerar embeddings manualmente**. O ChromaDB chama internamente o modelo `all-MiniLM-L6-v2`, vetoriza cada frase, e armazena o vetor junto com o texto e os metadados. Toda a complexidade do Lab 09 (spaCy, vetores, cosseno) foi encapsulada em uma única chamada.

△ Importante: upsert vs. add

Use `upsert()` ao invés de `add()`: se você rodar o script pela segunda vez, `add()` lança erro de ID duplicado, enquanto `upsert()` atualiza silenciosamente. Isso é essencial para scripts reprodutíveis e para o CI/CD.

5 Parte 5: Busca Semântica com Top-K (12 min)

A. Função de Busca Reutilizável

Agora implemente uma função de busca genérica que aceita query e filtros opcionais:

```
# 5. Funcao de busca reutilizavel
def buscar(query, top_k=3, filtro=None):
    """Busca semantica no ChromaDB com filtro opcional."""
    params = {
        "query_texts": [query],
        "n_results": top_k,
    }
    if filtro:
        params["where"] = filtro
    return colecao.query(**params)
```

Observe que não há nenhum laço `for`, nenhum cálculo de cosseno, nenhuma ordenação manual. O ChromaDB faz tudo internamente usando o grafo HNSW.

B. Testando com Queries sem Keywords

```
# 6. Queries de teste (sem keywords em comum!)
queries = [
    "Descobertas medicas que salvaram vidas",
    "Marcos da computacao e inteligencia artificial",
    "Revolucoes e mudancas de regime politico",
    "Obras de arte e musica classica",
]

if __name__ == "__main__":
    for q in queries:
        print(f"\n{'='*60}")
        print(f"QUERY: '{q}'")
        print(f"{'='*60}")
        resultados = buscar(q)
        for i, (doc, dist) in enumerate(
                zip(resultados['documents'][0],
                    resultados['distances'][0])):
            print(f"  #{i+1} [Dist: {dist:.4f}] {doc}")
```

O campo `distances` retorna a distância cosseno (quanto menor, mais similar). Verifique se “descobertas médicas” retorna penicilina e vacina como Top-2.

6 Parte 6: Filtros SQL-like sobre Metadados (13 min)

O grande diferencial de um banco vetorial profissional é combinar **busca por significado** com **filtros estruturados**. É como fazer um "WHERE" do SQL, mas sobre uma busca semântica:

A. Filtro por Categoria

```
if __name__ == "__main__":
    # 7. Busca APENAS em documentos de ciencia
    print("\n=== BUSCA COM FILTRO: Ciencia ===")
    resultados = buscar(
        "Inovacoes que mudaram o mundo",
        top_k=3,
        filtro={"categoria": "ciencia"}
    )
    for doc in resultados['documents'][0]:
        print(f"  > {doc}")
```

Sem o filtro, "inovações que mudaram o mundo" poderia retornar o iPhone ou a Revolução Francesa. Com o filtro `categoria: ciencia`, o resultado fica restrito a penicilina, Einstein e DNA. O filtro é aplicado **antes** da busca vetorial, reduzindo o espaço de busca.

B. Filtro por Século

```
    # 8. Busca APENAS no seculo XX
    print("\n=== BUSCA COM FILTRO: Seculo XX ===")
    resultados = buscar(
        "Acontecimentos marcantes",
        top_k=5,
        filtro={"seculo": "XX"}
    )
    for doc in resultados['documents'][0]:
        print(f"  > {doc}")
```

C. Filtro Combinado (AND)

Para filtros compostos, use o operador `$and`:

```
    # 9. Busca combinada: Tecnologia + Seculo XXI
    print("\n=== FILTRO: Tecnologia + Seculo XXI ===")
    resultados = buscar(
        "Inovacoes digitais recentes",
        top_k=3,
        filtro={
```

```
            "$and": [
                {"categoria": "tecnologia"},
                {"seculo": "XXI"}
            ]
        }
    )
    for doc in resultados['documents'][0]:
        print(f"  > {doc}")
```

Apenas iPhone e ChatGPT devem aparecer, pois são os únicos fatos de tecnologia do século XXI.

▷ **Exemplo: title=Referência: Operadores de Filtro do ChromaDB**

Operador	Uso
$eq (igual)	{"categoria": "ciencia"}
$ne (diferente)	{"categoria": {"$ne": "politica"}}
$in (lista)	{"seculo": {"$in": ["XX", "XXI"]}}
$and, $or	Combinações lógicas entre filtros

7 Parte 7: Garantia de Qualidade — Testes CI/CD (5 min)

Os testes usam um banco **in-memory** (sem persistência) para isolamento no CI. Crie `tests/test_banco.py`:

```
import chromadb

# Setup: banco in-memory para CI (sem persistencia)
cliente = chromadb.Client()
col = cliente.get_or_create_collection(
    name="test_col",
    metadata={"hnsw:space": "cosine"}
)
col.upsert(
    documents=[
        "O gato dormiu no sofa.",
        "Python e a linguagem mais usada em IA.",
        "O cachorro brincou no jardim.",
    ],
    ids=["d1", "d2", "d3"]
)

def test_contagem_documentos():
    """Verifica que 3 documentos foram inseridos."""
    assert col.count() == 3
```

```
def test_busca_retorna_top_k():
    """Query deve retornar exatamente n_results documentos."""
    r = col.query(query_texts=["animais"], n_results=2)
    assert len(r['documents'][0]) == 2

def test_busca_semantica_relevancia():
    """Query sobre animais deve retornar docs de animais,
    nao de programacao."""
    r = col.query(query_texts=["animais domesticos"],
                  n_results=1)
    doc_top1 = r['documents'][0][0]
    assert "gato" in doc_top1 or "cachorro" in doc_top1, \
        f"Top-1 inesperado: {doc_top1}"
```

Note o `chromadb.Client()` (sem `Persistent`): cria um banco temporário em memória que desaparece após o teste. Isso evita conflitos com o banco real do script principal. Execute `pytest tests/test_banco.py -v` e confirme os **3 passed**.

8 Parte 8: Commit e Push

A pasta `chroma_db/` contém binários pesados que **não devem** ser commitados:

```
echo "chroma_db/" >> .gitignore
git add src/banco_vetorial.py tests/test_banco.py .gitignore
git commit -m "Lab 11: ChromaDB + HNSW + filtros + CI/CD"
git push origin main
```

Conclusão: O que Você Construiu Hoje

✔ Takeaways

- Você saiu do laço `for` artesanal (Lab 10) para a infraestrutura profissional **ChromaDB** com HNSW — $O(\log N)$ ao invés de $O(N)$.
- Criou uma **Collection** persistente com métrica cosseno e injetou 20 documentos com metadados categóricos (categoria e século).
- Executou **buscas semânticas Top-K** que encontram documentos por significado, não por keywords.
- Usou **filtros SQL-like** (`where`) para restringir buscas por categoria e século — combinando semântica com metadados estruturados.
- Os 3 testes CI/CD garantem: inserção correta, Top-K funcional e relevância semântica.

◇ **Informação**

Gancho para a Próxima Aula: Seu banco vetorial funciona perfeitamente com frases curtas. Mas e se o documento for um **PDF de 200 páginas**? Inserir o texto inteiro como um único vetor destruiria a granularidade semântica. Na próxima aula, enfrentaremos a **Engenharia de Chunking** — a arte de fatiar documentos longos em pedaços digeríveis que preservam o contexto. Essa técnica é o que separa um RAG amador de um RAG profissional.