Modelagem e Armazenamento de Mensagens de uma Rede Social Banco de Dados Cassandra e DataStax Astra

Guilherme César Athayde

Novembro de 2025

1 Introdução

Este relatório descreve a implementação de um sistema de armazenamento de mensagens de uma rede social para a comunidade universitária da UFSCar, utilizando o DataStax Astra DB (Cassandra as a Service) e a biblioteca `astrapy` em Python.

O sistema foi projetado para atender aos requisitos de uma rede social moderna, com foco em escalabilidade, performance e flexibilidade na modelagem de dados.

2 Scripts CQL e Implementação

2.1 Tarefa 1: Criar a Coleção (Tabela)

No DataStax Astra DB, utilizamos a abordagem de *collections* que é equivalente a tabelas no Cassandra tradicional, mas com maior flexibilidade por ser schemaless. A estrutura dos documentos foi definida da seguinte forma:

```
collection_name = "social_messages"

# Dropar colecao se ja existir (para testes)
db.drop_collection(collection_name)

# Criar a colecao com estrutura:
# {
#   "_id": "message_id",
#   "user_id": "user_xxx",
#   "user_age": 25,
#   "topic": "tecnologia",
#   "message_text": "Texto da mensagem",
#   "timestamp": "2025-11-20T10:30:00"
# }

collection = db.create_collection(collection_name)
```

Listing 1: Criação da coleção

Equivalente CQL Cassandra:

```
CREATE TABLE IF NOT EXISTS social_messages (
    message_id TEXT,
    user_id TEXT,
    user_age INT,
    topic TEXT,
    message_text TEXT,
    timestamp TIMESTAMP,
    PRIMARY KEY (user_id, timestamp, message_id)
) WITH CLUSTERING ORDER BY (timestamp DESC);

CREATE INDEX IF NOT EXISTS idx_topic
ON social_messages (topic);
```

Listing 2: CQL equivalente para Cassandra tradicional

2.2 Tarefa 2: Inserir Dados (20+ mensagens)

Foram inseridas 25 mensagens com distribuição variada entre 10 usuários diferentes e 6 tópicos distintos:

```
users = [f"user_{str(i).zfill(3)}"
         for i in range(1, 11)]  # user_001 a user_010
user_ages = {user: random.randint(18, 65)
             for user in users}
topics = ["politica", "saude", "tecnologia",
          "educacao", "esportes", "entretenimento"]

# Gerar 25 mensagens com diferentes usuarios,
# topicos e datas
messages = []
base_date = datetime.now() - timedelta(days=30)

for i in range(25):
    user = random.choice(users)
    topic = random.choice(topics)
    timestamp = base_date + timedelta(
        days=random.randint(0, 30),
        hours=random.randint(0, 23),
        minutes=random.randint(0, 59))

    message = {
        "_id": f"msg_{str(i+1).zfill(3)}",
        "user_id": user,
        "user_age": user_ages[user],
        "topic": topic,
        "message_text": f"{message_text}
                          (Tema: {topic})",
        "timestamp": timestamp.isoformat()
    }
    messages.append(message)
```

```
31
32 result = collection.insert_many(messages)
```

Listing 3: Inserção de mensagens

Equivalente CQL:

```
INSERT INTO social_messages
  (message_id, user_id, user_age, topic,
   message_text, timestamp)
VALUES
  ('msg_001', 'user_001', 25, 'tecnologia',
   'Mensagem sobre tecnologia',
   '2025-11-01T10:30:00');
-- Repetir para todas as 25 mensagens...
```

Listing 4: CQL para inserção de dados

2.3 Tarefa 3: Recuperar Mensagem Específica

Implementação da consulta para recuperar mensagens de um usuário específico:

```
# Buscar uma mensagem do user_001
target_user = "user_001"
user_message = collection.find_one(
    {"user_id": target_user}
)

# Buscar todas as mensagens de um usuario especifico
all_user_messages = list(collection.find(
    {"user_id": target_user}
))
```

Listing 5: Consulta de mensagens por usuário

Equivalente CQL:

```
-- Buscar uma mensagem especifica
SELECT * FROM social_messages
WHERE user_id = 'user_001'
LIMIT 1;

-- Buscar todas as mensagens de um usuario
SELECT * FROM social_messages
WHERE user_id = 'user_001';
```

Listing 6: CQL para consulta por usuário

2.4 Tarefa 4: Índice e Frequência de Tópicos

Para consultar a frequência de tipos de mensagens por usuário:

```
# Recuperar todas as mensagens
all_messages = list(collection.find({}))
```

```python

# Agrupar por usuario e contar frequencia de topicos
user_topic_frequency = {}

for msg in all_messages:
    user = msg['user_id']
    topic = msg['topic']

    if user not in user_topic_frequency:
        user_topic_frequency[user] = {}

    if topic not in user_topic_frequency[user]:
        user_topic_frequency[user][topic] = 0

    user_topic_frequency[user][topic] += 1

# Exibir resultados
for user, topics in sorted(
    user_topic_frequency.items()
):
    print(f"\n{user}:")
    for topic, count in sorted(
        topics.items(),
        key=lambda x: x[1],
        reverse=True
    ):
        print(f"  - {topic}: {count} mensagem(ns)")
```

Listing 7: Consulta de frequência de tópicos por usuário

Equivalente CQL (agregação):

```cql
-- No Cassandra, a agregacao deve ser feita
-- em nivel de aplicacao
-- ou usando Spark/DataStax Analytics

SELECT user_id, topic, COUNT(*) as frequency
FROM social_messages
GROUP BY user_id, topic;

-- Alternativa com indice:
CREATE MATERIALIZED VIEW messages_by_user_topic AS
  SELECT user_id, topic, message_id, timestamp
  FROM social_messages
  WHERE user_id IS NOT NULL
    AND topic IS NOT NULL
  PRIMARY KEY ((user_id, topic), timestamp,
               message_id)
  WITH CLUSTERING ORDER BY (timestamp DESC);
```

Listing 8: CQL para agregação de frequências

3 Discussão sobre a Modelagem

3.1 Escolhas de Modelagem

A modelagem escolhida seguiu os princípios do Cassandra/DataStax Astra:

1. **Desnormalização**: Ao contrário de bancos relacionais, optamos por armazenar todos os dados da mensagem em um único documento, incluindo idade do usuário, que poderia estar em uma tabela separada em um modelo relacional.
2. **Query-First Design**: A estrutura foi projetada considerando as consultas mais frequentes:
   - Busca por usuário
   - Busca por data/timestamp
   - Busca por tópico
3. **Primary Key**: Em uma implementação Cassandra tradicional, a chave primária seria `(user_id, timestamp, message_id)`, onde:
   - `user_id`: Partition key (distribui dados entre nós)
   - `timestamp`: Clustering key (ordena dados dentro da partição)
   - `message_id`: Garante unicidade
4. **Schemaless com Astra**: O uso de collections no Astra DB oferece flexibilidade schemaless, permitindo evolução do schema sem migração de dados.

3.2 Estratégia de Indexação

3.2.1 Índices Automáticos do Astra DB

O DataStax Astra DB gerencia índices automaticamente para otimizar consultas. Para campos frequentemente consultados como `user_id` e `topic`, o Astra cria índices secundários dinamicamente.

3.2.2 Índices Secundários (Cassandra Tradicional)

Em um ambiente Cassandra tradicional, implementaríamos:

- **Índice em topic**: Para permitir consultas por tipo de mensagem
- **Materialized View**: Para agregações pré-computadas de frequência por usuário e tópico

3.2.3 Trade-offs

Vantagens:

- Consultas por usuário são extremamente rápidas (partition key lookup)
- Mensagens ordenadas por timestamp dentro de cada partição

- Escalabilidade horizontal natural do Cassandra
- Índices secundários permitem consultas flexíveis por tópico

Desvantagens:

- Índices secundários podem impactar performance de escrita
- Agregações (contagem de frequências) são mais eficientes quando processadas na aplicação
- Consultas cross-partition (todos os usuários) são menos eficientes

3.3 Consultas e Análise de Resultados

3.3.1 Performance de Consultas

1. **Consulta por usuário específico**: Operação O(1) usando partition key, extremamente eficiente mesmo com milhões de mensagens.
2. **Consulta de frequências**: Requer full table scan, mas com processamento eficiente em memória. Para produção, recomenda-se:
   - Usar Materialized Views para pré-agregação
   - Implementar cache (Redis) para resultados frequentes
   - Usar Apache Spark para análises complexas
3. **Consulta por tópico**: Índice secundário permite busca eficiente, mas menos otimizada que partition key lookup.

3.3.2 Resultados Obtidos

A implementação demonstrou:

- Inserção bem-sucedida de 25 mensagens distribuídas entre 10 usuários
- Recuperação instantânea de mensagens por usuário específico
- Cálculo correto de frequências de tópicos por usuário
- Distribuição equilibrada entre os 6 tópicos disponíveis

4 Conclusões

A implementação atendeu todos os requisitos funcionais propostos:

1. ✓ Cada mensagem possui ID único
2. ✓ Armazenamento de user_id e idade
3. ✓ Classificação por tema/tópico
4. ✓ Registro de timestamp
5. ✓ Consultas rápidas por usuário implementadas

4.1 Código-Fonte

O código completo deste projeto está disponível no GitHub:

`https://github.com/guiathayde/cassandra-rede-social`