39º Simpósio Brasileiro de Bancos de Dados

39th Brazilian Symposium on Databases

Tópicos em Gerenciamento de Dados e Informações: Minicursos do SBBD 2024

Topics in Data Management and Information: Short courses of SBBD 2024

De 14 a 17 de outubro de 2024

October 14th to 17th, 2024

Execução

UNIVERSIDADE FEDERAL DE SANTA CATARINA

Realização

SBC Sociedade Brasileira de Computação

Cebd SBC

Organização do SBBD 2024

SBBD 2024 Organization

Program Chair

Eduardo Ogasawara (CEFET/RJ)

Short Papers Chair

Carlos Eduardo Santos Pires (UFCG, Brasil)

Demos and Applications Chair

Dimas Cassimiro Nascimento (UFAPE, Brasil)

WTDBD Chair

Maristela Holanda (UnB, Brasil)

CTDBD Chair

Karin Becker (UFRGS, Brasil)

Short Courses Chair

José Maria da Silva Monteiro Filho (UFC, Brasil)

Tutorials Chair

Fábio Porto (LNCC, Brasil)

Workshops Chair

Kelly Rosa Braghetto (IME/USP, Brasil)

WTAG Chair

Nadia P. Kozievitch (UTFPR, Brasil)

Proceedings Chair

Humberto Razente (UFU, Brasil)

Marathon Chair

Ana Carolina Brito de Almeida (UERJ, Brasil)

Execução

UNIVERSIDADE FEDERAL DE SANTA CATARINA

Realização

SBC Sociedade Brasileira de Computação

Cebd SBC

Organização Geral

General Organization

Chairs

Carina Friedrich Dorneles (UFSC, Brasil)

Ronaldo Dos Santos Mello (UFSC, Brasil)

Execução

UNIVERSIDADE FEDERAL DE SANTA CATARINA

Realização

SBC Sociedade Brasileira de Computação

Cebd SBC

Dados Internacionais de Catalogação na Publicação (CIP)

S612 Simpósio Brasileiro de Banco de Dados (39. : 14 – 17 out. 2024 : Florianópolis)
Minicursos do SBBD 2024 [recurso eletrônico] / organização: Humberto Luiz Razente ... [et al.]. Dados eletrônicos. – Porto Alegre: Sociedade Brasileira de Computação, 2024.
90 p. : il. : PDF ; 3,3 MB

Modo de acesso: World Wide Web.
Inclui bibliografia
ISBN 978-85-7669-606-3 (e-book)

1. Computação – Brasil – Simpósio. 2. Banco de Dados. I. Razente, Humberto Luiz. II. Ogasawara, Eduardo. III. Monteiro Filho, José Maria da Silva. IV. Sociedade Brasileira de Computação. V. Título.

CDU 004.6(063)

Ficha catalográfica elaborada por Annie Casali – CRB-10/2339

Biblioteca Digital da SBC – SBC OpenLib

Índices para catálogo sistemático:

1. Ciência e tecnologia dos computadores : Informática : Dados – Publicação de conferências, congressos e simpósios etc. ... 004.6(063)

O presente livro inclui três capítulos escritos pelos autores dos minicursos selecionados e apresentados durante o XXXIX Simpósio Brasileiro de Bancos de Dados (SBBD 2024), realizado de 14 a 17 de setembro de 2024. Os minicursos têm como objetivo apresentar temas relevantes relacionados à área de Banco de Dados e promover discussões sobre os fundamentos, tendências e desafios dos temas abordados. Cada minicurso tem quatro horas de duração e constitui uma excelente oportunidade de atualização para acadêmicos e profissionais que participam do evento. A qualidade dessa edição é devida essencialmente aos autores e revisores dos trabalhos submetidos. Expressamos nossos fortes agradecimentos pelas contribuições e discussões durante o SBBD 2024.

Os capítulos abordam conteúdos relacionados à Geração com Recuperação Aumentada (RAG), Exploração e Otimização de Consultas, além de Mineração de Padrões em Grafos. O comitê de programa de minicursos foi composto pelos professores José Maria da Silva Monteiro Filho (UFC), Humberto Razente (UFU) e Ronaldo dos Santos Mello (UFSC), sob coordenação do primeiro.

A qualidade dessa edição é devida essencialmente aos autores e revisores dos trabalhos submetidos. Expressamos nossos fortes agradecimentos pelas contribuições e discussões durante o SBBD 2024.

Para mais informações sobre o SBBD 2024, visite https://sbbd.org.br/2024, o site desta edição do evento.

This book includes three chapters written by the authors of the selected tutorials presented during the 39th Brazilian Symposium on Databases (SBBD 2024), held from September 14 to 17, 2024. They aim to present relevant topics related to Databases. Moreover, they promote discussions on the topics' fundamentals, trends, and challenges. Each short course lasts four hours and is an excellent opportunity to update academics and professionals participating in the event.

The chapters cover content related to Retrieval-Augmented Generation (RAG), Query Exploration and Optimization, as well as Graph Pattern Mining. The short course program committee was composed of José Maria da Silva Monteiro Filho (UFC), Humberto Razente (UFU), and Ronaldo dos Santos Mello (UFSC) under the coordination of the former.

The richness of this issue can be mainly credited to the authors and reviewers. We greatly thank them for their insightful contributions and discussions during SBBD 2024.

You can find more information about SBBD 2024, visiting the https://sbbd.org.br/2024 website of the event.

José Maria da Silva Monteiro Filho
(UFC)

Execução

UNIVERSIDADE FEDERAL DE SANTA CATARINA

Realização

SBC Sociedade Brasileira de Computação

Cebd SBC

MINI

sbbd:01

Chapter 1

Geração com Recuperação Aumentada (RAG) em Grafos de Conhecimento

Otávio Calaça Xavier, Anderson da Silva Soares

Abstract

This chapter addresses the integration of Knowledge Graphs with Retrieval-Augmented Generation (RAG) technology applied to Large Language Models (LLMs), exploring how this combination can enhance the natural language processing capabilities of AI systems. It discussed the fundamental structures of knowledge graphs, detailing the functioning and advantages of RAG technology, and presenting practical examples using tools such as LangChain and the Neo4j database. Moreover, advanced RAG techniques are explored, emphasizing the importance of effective pre- and post-processing strategies to maximize the relevance and accuracy of responses generated by modern AI systems.

Resumo

Este capítulo aborda a integração de Grafos de Conhecimento com a tecnologia de Geração com Recuperação Aumentada (RAG) aplicada a Modelos de Linguagem de Grande Escala (LLMs), explorando como essa combinação pode enriquecer a capacidade de processamento de linguagem natural dos sistemas de IA. Discutiu-se as estruturas fundamentais dos grafos de conhecimento, detalhando o funcionamento e as vantagens da tecnologia RAG, e apresentando exemplos práticos de uso de ferramentas como LangChain e o banco de dados Neo4j. Além disso, são exploradas técnicas avançadas de RAG, enfatizando a importância de estratégias eficazes de pré e pós-processamento para maximizar a relevância e a precisão das respostas geradas pelos sistemas de IA modernos.

1.1. Introdução aos Grafos de Conhecimento e Geração com Recuperação Aumentada (RAG)

Grafos de conhecimento são representações estruturadas que facilitam o armazenamento e a manipulação de informações complexas, onde entidades são mapeadas como nós e as relações entre elas como arestas [Hogan et al. 2021]. Essas estruturas são fundamentais

para modelar dados de forma que preservam suas interconexões semânticas, permitindo que sistemas computacionais interpretem as relações e propriedades das entidades de maneira contextualizada [Zou 2020]. O exemplo da Figura 1.1 apresenta a estrutura de um grafo de conhecimento simples. Nessa estrutura é possível observar dois tipos de nós: Pessoa e Interesse e dois tipos de arestas: INTERESSADA_EM e AMIGO_DE. Tanto as arestas quanto os nós podem conter propriedades.

Figure 1.1. Estrutura do grafo de conhecimento simples

Em um grafo de conhecimento, entidades são tipicamente objetos ou conceitos com existência física ou abstrata, tais como pessoas, locais, ou eventos, enquanto relações descrevem como essas entidades interagem ou estão conectadas [Suchanek et al. 2007]. As relações são explicitamente definidas, frequentemente como vértices dirigidos que ligam um par de nós, o que introduz uma dimensão de direcionalidade e semântica, crucial para inferências e análises subsequentes.

A aplicação de grafos de conhecimento na inteligência artificial (IA) oferece uma forma robusta de incorporar conhecimento contextual em sistemas automáticos, permitindo um raciocínio mais aprofundado e decisões baseadas em uma rica rede de informações interligadas [Paulheim 2017]. Grafos de conhecimento provam ser particularmente valiosos para enriquecer a inteligência artificial com capacidades de raciocínio complexo e adaptativo sobre vastos domínios de conhecimento [Zou 2020].

Utilizando grafos de conhecimento, sistemas de IA podem alcançar um entendimento contextual que é crítico para tarefas que requerem não apenas análise de dados isolados, mas uma interpretação holística das interações e relações. Por exemplo, na área de recomendações personalizadas, grafos de conhecimento permitem mapear preferências do usuário a produtos ou conteúdos de maneira dinâmica e interconectada, levando a recomendações mais precisas e personalizadas [Nickel et al. 2016].

A integração de grafos de conhecimento em modelos de aprendizado de máquina e deep learning proporciona uma base para a inclusão de conhecimento estruturado e semântico, o que é particularmente benéfico em tarefas de compreensão e geração de linguagem [Bordes et al. 2013]. Esta capacidade de incorporar e processar relações complexas dentro de um framework de IA permite que os sistemas não apenas respondam a consultas com maior precisão, mas também interajam de maneira mais natural e intuitiva com os usuários.

Modelos de Linguagem de Grande Escala (LLMs) como GPT-4[1], BERT[2] e Llama 3[3], são desenvolvidos para entender e gerar texto humano de forma que possam realizar tarefas variadas de processamento de linguagem natural, desde tradução até geração de conteúdo. Esses modelos são treinados em vastos conjuntos de dados textuais, absorvendo padrões linguísticos e nuances contextuais [Rogers et al. 2021]. No entanto, por serem primariamente treinados com dados históricos, eles frequentemente carecem de acesso a informações atualizadas ou específicas durante a geração de respostas.

A técnica de Geração com Recuperação Aumentada (RAG) surge como uma solução para a limitação dos LLMs em lidar com informações dinâmicas e específicas. Essencialmente, RAG é um método híbrido que combina as capacidades de geração de texto dos LLMs com sistemas de recuperação de informação. Isso permite que os modelos não apenas gerem texto baseados em seu treinamento prévio, mas também busquem e incorporem informações atualizadas de bases de dados externas durante a geração de texto [Lewis et al. 2020].

O funcionamento do RAG pode ser dividido em duas etapas principais: recuperação e geração. Primeiro, quando uma pergunta (*query*) é recebida, o componente de recuperação busca nos bancos de dados externos para encontrar as informações mais relevantes e atualizadas. Essas informações são geralmente estruturadas em formatos como grafos de conhecimento ou tabelas, que permitem rápida localização e extração de dados. Em seguida, esses dados são fornecidos ao modelo de linguagem, que integra as informações recuperadas com seu conhecimento pré-existente para gerar uma resposta coerente e contextualizada [Lewis et al. 2020].

A técnica RAG evoluiu rapidamente, apresentando paradigmas distintos como o *Naive RAG*, *Advanced RAG* e *Modular RAG* [Gao et al. 2023]. Cada um desses modelos traz melhorias incrementais sobre os seus predecessores, adaptando e refinando a interação entre os módulos de recuperação e geração para melhorar a qualidade e a relevância das respostas fornecidas pelos LLMs.

O Advanced RAG, em particular, aprimora significativamente o processo de recuperação ao introduzir estratégias de pré-recuperação e pós-recuperação, que refinam tanto a indexação quanto a incorporação de informações recuperadas antes da geração de texto. Essas inovações permitem que o RAG lide com tarefas mais complexas e conhecimento intensivo, superando assim as limitações do *Naive RAG* em contextos desafiadores [Gao et al. 2023].

Já o *Modular RAG* representa a mais recente inovação dentro deste campo, oferecendo uma abordagem altamente adaptável que permite a substituição e reconfiguração de módulos para enfrentar desafios específicos. Esta versatilidade torna o *Modular RAG* particularmente eficaz em ambientes dinâmicos e variados, proporcionando uma melhoria substancial sobre as abordagens anteriores [Gao et al. 2023].

[1]GPT é sigla para *Generative Pre-trained Transformer*. Mais detalhes sobre GPT-4 em: https://openai.com/index/gpt-4/

[2]BERT é sigla para *Bidirectional Encoder Representations from Transformers*[Devlin et al. 2018] e foi um dos primeiros modelos de linguagem baseados em *transformers*.

[3]Llama é sigla para *Large Language Model Meta AI*. Mais detalhes sobre Llama 3 em: https://llama.meta.com/

Integrando técnicas de RAG, os LLMs podem não apenas responder perguntas com maior precisão, mas também ajustar suas respostas com base em informações que são dinamicamente recuperadas e integradas. Isso não só aumenta a precisão das respostas geradas como também amplia significativamente a utilidade dos LLMs em aplicações do mundo real, marcando um avanço notável na interação entre recuperação de informações e geração de linguagem [Gao et al. 2023].

1.1.1. Introdução ao *framework* LangChain

O LangChain é um *framework* de código aberto em Python desenvolvida para a criação de aplicações que utilizam grandes modelos de linguagem (LLMs). Ele facilita todo o ciclo de vida de aplicativos baseados em LLM, desde o desenvolvimento até a produção e o lançamento [Pandya and Holia 2023].

Com o *framework* LangChain[4] é possível integrar LLMs com sistemas de recuperação de informações, facilitando o uso de dados estruturados durante a geração de texto. Desenvolvido para otimizar a interação entre LLMs e bases de dados, o LangChain suporta uma variedade de tecnologias de armazenamento de dados, permitindo consultas dinâmicas e a incorporação de informações externas em tempo real.

O LangChain apresenta uma arquitetura modular, que suporta a flexibilidade na configuração de conexões entre diversos LLMs e vários sistemas de armazenamento de dados, como bancos de dados SQL, NoSQL e sistemas baseados em grafos como o Neo4j. O *framework* oferece uma API que abstrai complexidades de consulta, permitindo que desenvolvedores e pesquisadores concentrem-se na lógica de aplicação sem a necessidade de detalhar as especificidades das linguagens de consulta de banco de dados.

No contexto de RAG, o LangChain facilita a execução de consultas complexas a partir de *prompts* de linguagem natural, traduzindo essas entradas para consultas estruturadas que são executadas em bases de dados externas. A informação recuperada é então sintetizada e incorporada pelo LLM para gerar respostas que são contextualmente relevantes. Esta capacidade é essencial para aplicações em que a atualidade e precisão dos dados são críticas, como em assistentes virtuais inteligentes e sistemas de suporte a decisões.

O LangChain suporta uma variedade de funcionalidades como *caching* de resultados, pré-processamento de dados e otimização de consultas, o que pode melhorar significativamente a eficiência das operações de recuperação de dados em aplicações de RAG. Além disso, o *framework* permite ajustes em tempo real dos parâmetros de consulta, o que aumenta a flexibilidade e adaptabilidade dos sistemas baseados em LLMs a mudanças em requisitos ou atualização constante dos dados.

Apesar de suas vantagens, a implementação de LangChain em sistemas de RAG apresenta desafios técnicos, incluindo a necessidade de garantir a segurança dos dados durante a transferência entre diferentes sistemas e a manutenção de latências baixas em consultas de alta complexidade. Não é escopo desde estudo focar no aprimoramento das capacidades de processamento paralelo do LangChain ou na otimização de algoritmos de indexação e recuperação para melhorar o desempenho e a escalabilidade de aplicações de

[4]Site oficial do LangChain: https://www.langchain.com/

RAG.

1.1.2. Introdução ao banco de dados Neo4j

Neo4j[5] é um banco de dados baseado em grafos, desenvolvido para facilitar consultas eficientes e a gestão de relações complexas entre dados. Diferentemente dos bancos de dados relacionais, que armazenam dados em tabelas, o Neo4j organiza dados em nós, arestas e propriedades, proporcionando uma representação mais intuitiva de redes de dados. Esta estrutura permite consultas que eficientemente percorrem grandes redes, explorando relações diretamente entre os objetos de interesse [Robinson et al. 2015].

A modelagem baseada em grafos do Neo4j oferece vantagens significativas, especialmente para aplicações que requerem intensa interação entre elementos conectados, como redes sociais, sistemas de recomendação e infraestruturas de segurança cibernética. As operações, como busca de caminhos mínimos, são otimizadas para serem rápidas e eficientes, o que não seria possível com a mesma eficiência em modelos relacionais ou hierárquicos devido ao custo associado a junções complexas e operações recursivas [Angles and Gutierrez 2008].

Neo4j utiliza a linguagem de consulta Cypher, base para o padrão ISO recentemente publicado GQL[6] (*Graph Query Language*), especificamente desenhada para facilitar a manipulação e recuperação de dados em grafos. Cypher é conhecida por sua expressividade e eficiência, permitindo que os usuários formulem consultas complexas de forma mais intuitiva. Com sintaxe declarativa, facilita significativamente o processo de descrever padrões nos grafos, extrair informações e modificar dados de grafos sem a necessidade de scripts complexos [Angles and Gutierrez 2008].

A flexibilidade do Neo4j para integrar-se com várias tecnologias modernas de análise de dados e inteligência artificial é uma de suas fortes vantagens. Ele pode ser utilizado junto com plataformas de processamento de dados *frameworks* de aprendizagem de máquina, permitindo a construção de sistemas inteligentes que adaptam e aprendem com a estrutura dos dados em grafos. Essa integração é crucial para desenvolver sistemas que não apenas reagem em tempo real, mas também evoluem com as mudanças nas relações de dados [Miller 2013].

Neo4j utiliza o modelo de grafos rotulados com propriedades (*labeled property graphs*), que enriquece os nós e arestas com rótulos e atributos, facilitando a representação detalhada de informações e a modelagem de relações complexas. Cada nó no Neo4j pode ter um ou mais rótulos, que definem seu tipo ou categoria, enquanto as arestas, que representam os relacionamentos, podem conter propriedades que detalham as características do relacionamento, como data, peso, direção ou qualquer outro metadado relevante. Esse modelo é altamente flexível e adaptável, adequado para representar estruturas de dados complexas, como os grafos de conhecimento [Angles et al. 2017].

Em contraste com os grafos rotulados com propriedades, outras tecnologias como o *Resource Description Framework* (RDF) e a *Web Ontology Language* (OWL) utilizam o conceito de triplas para armazenar dados. Uma tripla, no contexto de RDF, consiste em

[5]Site oficial do Neo4j: https://neo4j.com/

[6]ISO/IEC 39075:2024: Graph Query Language - GQL

um sujeito, um predicado e um objeto, e é usada para representar informações semânticas na web. OWL, por sua vez, utiliza RDF para criar ontologias que definem sistemas complexos de entidades e suas inter-relações com regras, normas e lógica, oferecendo uma camada adicional de formalismo e expressividade [Allemang and Hendler 2011].

Uma ontologia é um conjunto formal de conceitos dentro de um domínio e as relações entre esses conceitos. Ontologias desempenham um papel crucial na organização e na inferência de conhecimento em sistemas baseados em grafos. Ao definir um conjunto rigoroso de conceitos e categorias que descrevem um domínio específico, as ontologias permitem a aplicação de raciocínio lógico para deduzir novas informações e relações a partir dos dados existentes. Os grafos de conhecimento, como aqueles suportados pelo Neo4j, podem descrever ontologias, de forma a serem utilizadas para enriquecer a base de dados com metadados semânticos, melhorando a precisão das consultas e a relevância dos resultados [Allemang and Hendler 2011].

1.1.3. *Embeddings* e busca vetorial

Embeddings, ou representações vetoriais, são fundamentais no campo do processamento de linguagem natural (PLN). Algoritmos como Word2Vec [Mikolov et al. 2013], GloVe [Pennington et al. 2014], e BERT [Devlin et al. 2018] aprendem a codificar palavras e frases em vetores que capturam relações semânticas e sintáticas. Esses modelos são treinados em vastos corpora de texto, aprendendo a representar linguisticamente entidades como vetores em espaços dimensionais que refletem a proximidade semântica entre os conceitos. Modelos de linguagem mais recentes como Llama e GPT-4 também podem ser utilizados para geração de *embeddings*.

A busca vetorial utiliza medidas de similaridade, como a distância euclidiana ou a similaridade do cosseno, para identificar itens semanticamente próximos a partir de *embeddings*. No PLN, isso é aplicado em sistemas de recomendação e motores de busca para identificar conteúdo relacionado à consulta do usuário, mesmo sem correspondência direta de palavras-chave, demonstrando uma recuperação de informação mais flexível e contextualizada [Manning 2008].

Em RAG, *embeddings* facilitam a recuperação de conteúdos relevantes que informam a geração de texto subsequente. Ao converter consultas e conteúdos da base de dados em vetores de *embeddings*, sistemas de recuperação podem rapidamente comparar o a representação do texto informado na consulta com porções de texto disponíveis na base de dados. Assim, é possível identificar os segmentos mais pertinentes, enriquecendo a resposta gerada pela LLM com dados contextuais [Lewis et al. 2020].

A integração de técnicas de busca vetorial e *embeddings* em LLMs, como no RAG, melhora significativamente a capacidade dos modelos de responder perguntas complexas e realizar tarefas específicas de domínio. Por exemplo, em um cenário de assistente virtual, *embeddings* podem ajudar a refinar as respostas do assistente para que elas sejam mais personalizadas e informativas, baseando-se na análise vetorial de documentos de suporte em vez de apenas gerar respostas baseadas em treinamento prévio isolado [Henderson et al. 2017].

1.1.4. Configuração inicial do Neo4j e LangChain

O Neo4j é um banco de dados que pode ser instalado em máquina local, ou em um provedor de infraestrutura em nuvem. Para instalação local, o *download* do instalador pode ser realizado através do site oficial[7]. Para facilitar a configuração, será utilizado o Neo4j Aura[8], uma solução em nuvem gratuita para projetos iniciais que não requer instalação.

Como dito anteriormente, o LangChain é um framework Python. Logo, para sua instalação, pode-se utilizar o gerenciador de pacotes `pip`. Como ele será utilizado para conexão ao Neo4j, outros pacotes também são relevantes para instalação. A Figura 1.2 mostra o comando a ser executado para essa instalação. Visando facilitar a utilização do *framework* LangChain com Neo4j, será utilizado o Google Colab, um ambiente online onde é possível a execução de códigos Python sem a necessidade de instalação do interpretador da linguagem em ambiente local. Para isso, foi criado um arquivo *notebook*[9] com todos os códigos a serem executados e mais detalhes.

```
pip install langchain langchain-community neo4j
```

Figure 1.2. Instalação do LangChain e bibliotecas correlatas

Uma vez criado o banco de dados no Neo4j, será disponibilizada uma URI e senha. O valor padrão tanto para o banco de dados quanto para o usuário é `neo4j`. De posse dessas quatro informações (url, usuário, senha e banco de dados) é possível realizar a conexão ao Neo4j a partir do LangChain, conforme Figura 1.3.

```
from langchain_community.graphs import Neo4jGraph

NEO4J_URI = "<URI PARA SEU BANCO NEO4J>"
NEO4J_USERNAME = "neo4j"
NEO4J_PASSWORD = "<SENHA GERADA>"
NEO4J_DATABASE = "neo4j"

kg = Neo4jGraph(
    url=NEO4J_URI, username=NEO4J_USERNAME,
    password=NEO4J_PASSWORD, database=NEO4J_DATABASE
)
```

Figure 1.3. Conexão ao banco de dados Neo4j utilizando LangChain

[7]Site oficial para download do Neo4j: https://neo4j.com/download/

[8]Site oficial do Neo4j Aura: https://console.neo4j.io/?ref=aura-lp

[9]O *notebook* criado no Google Colab pode ser acessado em: https://colab.research.google.com/drive/1ZTdI5xWlMjjzPwrsx1q6SJxbOvm05H-j?usp=sharing

1.2. Introdução à linguagem Cypher

Cypher é a linguagem de consulta declarativa do Neo4j, projetada especificamente para trabalhar com dados em formato de grafo. Ela permite expressar o que recuperar do grafo, não como recuperar, o que facilita a focar na definição da lógica de negócios sem se preocupar com os detalhes de implementação subjacentes.

Uma consulta Cypher típica envolve especificar padrões em um grafo, que consistem em nós, arestas (relacionamentos) e propriedades. A sintaxe básica inclui:

- **Nós**: Representados por parênteses `()`, similar a um círculo no diagrama de grafo.
- **Relacionamentos**: Representados por setas `->` ou `<-`, conectando nós. Os colchetes com dois pontos `[:]` são utilizados para nomear o relacionamento, por exemplo: `-[:CONHECE]->`.
- **Propriedades**: Valores armazenados nos nós ou relacionamentos, acessados através de chaves `{}`.

Para criar nós, é utilizada a palavra-chave **`CREATE`**. A Figura 1.4 apresenta um exemplo de criação de um nó do tipo `Pessoa` com o atributo `nome` igual a "João". Já a Figura 1.5 apresenta um exemplo mais completo. Neste exemplo, inicialmente é criado um nó do tipo `Pessoa` (`alice`) e outro do tipo `Interesse`. Em seguida é criado um relacionamento `INTERESSADA_EM` entre esses dois nós.

```
CREATE (joao:Pessoa {nome:"João"})
RETURN joao
```

Figure 1.4. Adicionando um novo nó do tipo Pessoa

```
CREATE (alice:Pessoa {nome: "Alice", idade: 30})
CREATE (neo4j:Interesse {nome: "Neo4j"})
CREATE (alice)-[:INTERESSADA_EM]->(neo4j)
```

Figure 1.5. Adicionando dois nós do tipo pessoa e uma aresta entre eles

Para executar comandos Cypher em um banco de dados Neo4j utilizando o LangChain, utiliza-se o método `query` do objeto `Neo4jGraph`. A Figura 1.6 mostra um exemplo de execução do código Cypher apresentado na Figura 1.5 utilizando o objeto `kg` criado na Figura 1.3.

```
kg.query("""
CREATE (alice:Pessoa {nome: "Alice", idade: 30})
CREATE (neo4j:Interesse {nome: "Neo4j"})
CREATE (alice)-[:INTERESSADA_EM]->(neo4j)
""")
```

Figure 1.6. Conexão ao banco de dados Neo4j utilizando LangChain

Supondo um banco de dados de grafos com informações sobre pessoas e seus interesses. Uma consulta nesse grafo para encontrar interesses específicos de uma pessoa poderia ser realizada como apresentado na Figura 1.7. No código apresentado:

- **MATCH**: Define o padrão para a consulta. Aqui, é procurado um nó de pessoa (Pessoa) com nome "Alice" que tenha um relacionamento INTERESSADO_EM com qualquer nó de interesse (Interesse).
- **RETURN**: Especifica o que retornar. Neste caso, o nome da pessoa e o nome de seus interesses.

```
MATCH (p:Pessoa {nome: "Alice"})-[:INTERESSADA_EM]->(i:Interesse)
RETURN p.nome, i.nome
```

Figure 1.7. Consulta Cypher para requisitar todos os interesses de uma pessoa chamada Alice

Também existe a palavra-chave **WHERE** para filtragem dos resultados. No exemplo da Figura 1.8 é utilizada a palavra-chave **WHERE** objetivando o mesmo resultado da consulta realizada na Figura 1.7.

```
MATCH (p:Pessoa)-[:INTERESSADA_EM]->(i:Interesse)
WHERE p.nome = "Alice"
RETURN p.nome, i.nome
```

Figure 1.8. Consulta Cypher para requisitar todos os interesses de uma pessoa chamada Alice utilizando WHERE

Um outro exemplo interessante, apresentado na Figura 1.9, é a consulta de múltiplos nós. Neste exemplo, a busca objetiva encontrar pessoas que compartilham os mesmos interesses.

```
MATCH (p1:Pessoa)-[:INTERESSADA_EM]->(i:Interesse)<-[:INTERESSADA_EM]-
(p2:Pessoa)
RETURN p1.nome, p2.nome, i.nome LIMIT 10
```

Figure 1.9. Consulta Cypher para encontrar pessoas que compartilham os mesmos interesses

Para excluir relacionamentos ou nós, a palavra-chave **DELETE** é utilizada, como observado na Figura 1.10. A modificação de nós ou relacionamentos pode ser realizada utilizando a palavra-chave **MERGE**. No exemplo da Figura 1.11, inicialmente é realizada uma consulta com match para pegar dois nós e em seguida eles são atualizados para a inclusão de um novo relacionamento entre eles.

```
MATCH (p:Pessoa {nome:"João"})-[rel:INTERESSADA_EM]->(i:Interesse)
DELETE rel
```

Figure 1.10. Excluindo um relacionamento utilizando DELETE

```
MATCH (joao:Pessoa {nome:"João"}), (alice:Pessoa {nome:"Alice"})
MERGE (joao)-[rel:AMIGOS_DE]->(alice)
RETURN joao, rel, alice
```

Figure 1.11. Utilizando MERGE para adicionar um relacionamento entre nós existentes

1.3. Construção e Indexação de Grafos de Conhecimento para RAG

A primeira etapa na construção de grafos de conhecimento no Neo4j é a modelagem de dados adequada. Esta fase envolve definir como as entidades e as relações serão representadas no grafo. Os modelos de dados em grafos, diferentemente dos modelos relacionais, são projetados para otimizar a conectividade e a eficiência das consultas relacionais, aproveitando estruturas que refletem diretamente as interações e relações naturais entre os dados [Robinson et al. 2015]. Por exemplo, em um contexto de mídia social, usuários, posts, e interações como "curtidas" e comentários podem ser diretamente mapeados para nós e relações em um grafo.

A extração de dados é o próximo passo crítico, envolvendo tanto fontes estruturadas quanto não estruturadas. Fontes estruturadas incluem bancos de dados relacionais e planilhas, enquanto fontes não estruturadas abrangem textos, imagens, e vídeos. Ferramentas como processadores de linguagem natural podem ser utilizadas para extrair entidades e suas relações de documentos de texto, que são então traduzidas em nós e arestas no Neo4j [Manning 2008]. Essa etapa é crucial para garantir que os dados sejam mapeados corretamente para o modelo de grafo.

Após a extração, os dados são integrados no Neo4j usando uma variedade de técnicas, dependendo da fonte dos dados. Para dados estruturados, ferramentas de ETL (*Extract, Transform, Load*) podem ser empregadas para migrar dados de sistemas tradicionais para o Neo4j. Para dados não estruturados, analizadores especializados e algoritmos de aprendizado de máquina preparam e formatam os dados para inserção no grafo [Angles and Gutierrez 2008]. A combinação da linguagem Cypher com as características do Python e LangChain pode ser muito apropriada para essa etapa.

Uma vez que os dados estão no Neo4j, a indexação e outras otimizações são essenciais para melhorar a performance das consultas. O Neo4j permite a criação de índices em

propriedades de nós e relações, o que pode significativamente acelerar as consultas, especialmente em grafos grandes e complexos. Além disso, o Neo4j suporta procedimentos armazenados que podem realizar operações complexas diretamente no banco de dados, reduzindo a necessidade de processamento externo [Miller 2013].

O suporte para indexação vetorial é particularmente valioso para tarefas que envolvem a comparação de similaridade entre entidades, como na busca de documentos, recomendação de conteúdo, ou agrupamento de itens semelhantes. Por exemplo, no contexto de um banco de dados de filmes, podemos utilizar o campo textual com a sinopse de cada filme para gerar *embeddings* que capturam o conteúdo semântico das descrições. Estes *embeddings* podem então ser indexados e utilizados para encontrar filmes com descrições semelhantes.

1.3.1. Exemplo prático de criação de grafo de conhecimento a partir de dados tabulares

O TMDB 5000 Movie Dataset[10], derivado do The Movie Database (TMDb), é uma compilação abrangente de dados sobre aproximadamente 5.000 filmes, utilizada extensivamente para análise e modelagem em diversos campos de pesquisa, incluindo ciência de dados, aprendizado de máquina e sistemas de recomendação. Este dataset inclui variáveis multidimensionais, tais como orçamento, receita, popularidade, gêneros, e sinopses, proporcionando uma rica fonte de informações para explorar padrões de mercado, comportamento do consumidor e preferências cinematográficas. Além disso, a disponibilidade de detalhes sobre produtoras, países de produção, datas de lançamento, e métricas de votação, entre outros, permite análises profundas de tendências de produção e consumo dentro da indústria cinematográfica.

Os dados TMDB 5000 são dispostos de maneira tabular, tendo alguns campos multivalorados e compostos, em formato JSON, como apresentado na Figura 1.12. A Figura 1.14 apresenta o código para carregar o arquivo CSV do dataset em um DataFrame do Pandas. O primeiro passo para transformar esse dataset em um grafo de conhecimento é criar os nós que representam os filmes (*Movie*). Cada linha do dataset será um nó do grafo. Em seguida, alguns campos multivalorados podem ser extraidos, como Gênero (*genre* e Palavras-Chaves (*keywords*). Visando a busca vetorial, o campo com a sinopse (*overview*) pode ser utilizado para geração dos *embeddings*. O código para execução de todas essas atividades pode ser visto na Figura 1.13.

Figure 1.12. Dados tabulares com campos do tipo JSON no TMDB 5000

	id	title	genres	keywords	budget	revenue	overview	tagline
0	19995	Avatar	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	[{"id": 1463, "name": "culture clash"}, {"id":...	237000000	2787965087	In the 22nd century, a paraplegic Marine is di...	Enter the World of Pandora.
1	285	Pirates of the Caribbean: At World's End	[{"id": 12, "name": "Adventure"}, {"id": 14, "...	[{"id": 270, "name": "ocean"}, {"id": 726, "na...	300000000	961000000	Captain Barbossa, long believed to be dead, ha...	At the end of the world, the adventure begins.
2	206647	Spectre	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	[{"id": 470, "name": "spy"}, {"id": 818, "name...	245000000	880674609	A cryptic message from Bond's past sends him o...	A Plan No One Escapes
3	49026	The Dark Knight Rises	[{"id": 28, "name": "Action"}, {"id": 80, "nam...	[{"id": 849, "name": "dc comics"}, {"id": 853,...	250000000	1084939099	Following the death of District Attorney Harve...	The Legend Ends
4	49529	John Carter	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	[{"id": 818, "name": "based on novel"}, {"id":...	260000000	284139100	John Carter is a war-weary, former military ca...	Lost in our world, found in another.
...	...	...	...	...	...	...	...	...
4798	9367	El Mariachi	[{"id": 28, "name": "Action"}, {"id": 80, "nam...	[{"id": 5616, "name": "united states\u2013mexi...	220000	2040920	El Mariachi just wants to play his guitar and ...	He didn't come looking for trouble, but troubl...
4799	72766	Newlyweds	[{"id": 35, "name": "Comedy"}, {"id": 10749, "...	[]	9000	0	A newlywed couple's honeymoon is upended by th...	A newlywed couple's honeymoon is upended by th...
4800	231617	Signed, Sealed, Delivered	[{"id": 35, "name": "Comedy"}, {"id": 18, "nam...	[{"id": 248, "name": "date"}, {"id": 699, "nam...	0	0	"Signed, Sealed, Delivered" introduces a dedic...	NaN
4801	126186	Shanghai Calling	[]	[]	0	0	When ambitious New York attorney Sam is sent t...	A New Yorker in Shanghai
4802	25975	My Date with Drew	[{"id": 99, "name": "Documentary"}]	[{"id": 1523, "name": "obsession"}, {"id": 224...	0	0	Ever since the second grade when he first saw ...	NaN

4803 rows × 8 columns

[10]Detalhes sobre o TMDB 5000 Movie Dataset podem ser acessados em: https://www.kaggle.com/datasets/tmdb/tmdb-movie-metadata. O dataset dos filmes está disponível em: https://raw.githubusercontent.com/otaviocx/datasets/main/tmdb$_5$000$_m$ovies.csv

```python
from langchain_community.graphs import Neo4jGraph
from neo4j import GraphDatabase
import json

# Função para carregar dados do CSV para Neo4j
def load_data_to_neo4j(data, graph):
  # Criar nós e relações com base nos dados
  for index, row in data.iterrows():
    # Tratar campos JSON
    genres = json.loads(row['genres'].replace('\'', '"'))
    genre_names = [genre['name'] for genre in genres]
    keywords = json.loads(row['keywords'].replace('\'', '"'))
    keyword_names = [keyword['name'] for keyword in keywords]

    # Criar o nó do filme
    movie_query = """
    MERGE (movie:Movie {title: $title, id: $id})
    SET movie.budget = $budget, movie.revenue = $revenue,
        movie.overview = $overview, movie.tagline = $tagline
    """
    graph.query(movie_query, params={
      "title": row['title'],
      "id": row['id'],
      "budget": row['budget'],
      "revenue": row['revenue'],
      "overview": row['overview'],
      "tagline": row['tagline'],
    })

    # Criar nós de gêneros e relações
    for genre in genre_names:
      genre_query = """
      MATCH (movie:Movie {title: $movie_title})
      MERGE (genre:Genre {name: $genre_name})
      MERGE (movie)-[:HAS_GENRE]->(genre)
      """
      graph.query(genre_query, params={
        "genre_name": genre,
        "movie_title": row['title']
      })

    # Criar nós de palavras chaves e relações
    for keyword in keyword_names:
      keyword_query = """
      MATCH (movie:Movie {title: $movie_title})
      MERGE (keyword:Keyword {name: $keyword})
      MERGE (movie)-[:HAS_KEYWORD]->(keyword)
      """
      graph.query(keyword_query, params={
        "keyword": keyword,
        "movie_title": row['title']
      })

# Carregar dados
load_data_to_neo4j(data, kg)
```

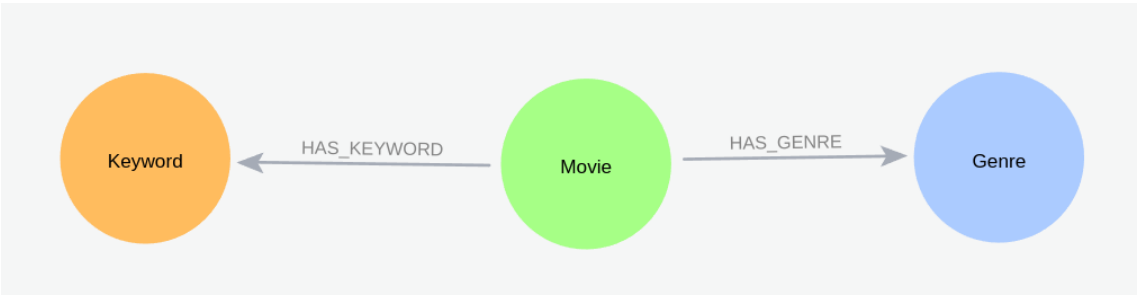
Figure 1.13. Criando Grafo de Conhecimento apartir de dados tabulares do dataset TMDB 5000

```python
import pandas as pd

# Carregando o arquivo CSV para examinar a estrutura
repo = 'https://raw.githubusercontent.com/otaviocx'
file_path = f'{repo}/datasets/main/tmdb_5000_movies.csv'
data = pd.read_csv(file_path)
data
```

Figure 1.14. Carregando o arquivo tabular (CSV) do dataset TMDB 5000

Ao final da execução do código apresentado na Figura 1.13, o grafo criado no Neo4j exibirá a estrutura apresentada na Figura 1.15. O próximo passo é gerar os embeddings e armazená-los nos nós do tipo `Movie`. Para tal, será utilizada a biblioteca GPT4All[11]. Desenvolvida para facilitar a implementação e a utilização de um dos mais avançados modelos de linguagem disponíveis, a GPT4All oferece uma interface simplificada para interagir com o GPT-4, permitindo aos desenvolvedores e pesquisadores incorporar capacidades de compreensão e geração de texto em uma variedade de aplicações. Com essa biblioteca, é possível realizar tarefas complexas de processamento de linguagem natural, como resposta a perguntas, geração de texto, e tradução automática, com relativa facilidade e sem a necessidade de infraestrutura computacional intensiva tipicamente requerida por modelos de grande escala. A GPT4All democratiza o acesso a tecnologias de ponta em PLN, promovendo uma inclusão mais ampla de capacidades de inteligência artificial em aplicações comerciais e acadêmicas, e facilitando a experimentação e inovação no campo da inteligência artificial [Brown et al. 2020].

Figure 1.15. Estrutura do grafo de conhecimento criado com base no dataset TMDB 5000



O GPT4All pode ser carregado a partir do LangChain, conforme apresentado no código da Figura 1.16. Neste código, inicialmente é carregado um modelo de linguagem utilizado para gerar os embeddings do campo `overview` de cada filme no dataset. Ao final da execução desse código, a variável `embeddings` possuirá uma lista de vetores numéricos representando cada um dos filmes. Em seguida, o código da Figura 1.17 é executado, ele criará um novo índice vetorial na propriedade `embedding` dos nós do tipo `Movie`. Por fim, o código da Figura 1.18 adiciona os embbedings aos nós correspondentes.

[11]Site oficial do GPT4All: https://www.nomic.ai/gpt4all

```python
from langchain_community.embeddings import GPT4AllEmbeddings

gpt4all_kwargs = {'allow_download': 'True'}
gpt4all_embd = GPT4AllEmbeddings(
    gpt4all_kwargs=gpt4all_kwargs
)

embeddings = gpt4all_embd.embed_documents(data['overview'].values)
embeddings
```

Figure 1.16. Gera os *embeddings* para o campo `overview` dos filmes.

```
CREATE VECTOR INDEX `movie_embedings` IF NOT EXISTS
FOR (m:Movie) ON (m.embedding)
OPTIONS { indexConfig: {
  `vector.dimensions`: 384,
  `vector.similarity_function`: "cosine"
}}
```

Figure 1.17. Adiciona os valores dos embeddings ao campo indexado `embedding` dos nós do tipo `Movie`

```python
# Função para carregar embeddings nos nós do Neo4j
def load_embeddings_to_neo4j(data, graph):
  for index, row in data.iterrows():
    # Atualizar o nó do filme com embeddings
    movie_query = """
    MATCH (movie:Movie {id: $id})
    CALL db.create.setNodeVectorProperty(movie, "embedding", $embeddings)
    """
    graph.query(movie_query, params={
      "id": row['id'],
      "embeddings": row['embeddings']
    })

data['embeddings'] = embeddings
load_embeddings_to_neo4j(data, kg)
```

Figure 1.18. Adiciona os valores dos embeddings ao campo indexado `embedding` dos nós do tipo `Movie`

Com os *embeddings* gerados e atribuidos aos respectivos nós, é possível fazer uma consulta vetorial conforme apresentado na Figura 1.19.

```
def neo4j_vector_search(question):
  question_embedding = gpt4all_embd.embed_query(question)

  vector_search_query = """
    CALL db.index.vector.queryNodes(
      "movie_embedings",
      $top_k, $question_embedding
    ) yield node, score
    RETURN score, node.title as title, node.overview AS overview
  """
  similar = kg.query(vector_search_query,
                     params={
                      'question_embedding': question_embedding,
                      'top_k': 10})
  return similar

neo4j_vector_search(
    'Which movies are about love and action?'
)
```

Figure 1.19. Realiza busca vetorial por similaridade no campo `embedding`

1.4. Aplicação de RAG em Grafos de Conhecimento

Ao passo que os `embeddings` foram gerados e a busca vetorial é possivel de ser realizada, a parte de indexação e recuperação da informação está concluida. O próximo passo é a geração, com base na recuperação realizada. Para tal, uma forma simples, e frequentemente utilizada, de implementação de RAG é o complemento do prompt com as informações recuperadas a partir da busca vetorial.

No exemplo da Figura 1.20, estamos carregando um LLM utilizando o GPT4All para gerar respostas com base em seu treino. Nesse código não há a utilização do RAG. Na Figura 1.21 é possível observar um possivel resultado dessa execução. A resposta é inteiramente formada com base no pré-treino do modelo, não há definição de escopo nem é possível uma atualização da lista de filmes conhecidos.

Já o código da Figura 1.22 adiciona uma função para implementação do RAG. Nesta função, inicialmente é utilizada a pergunta original para buscar nós do grafo que possam ser relevantes para a resposta. Em seguida, o prompt (pergunta a ser enviada para o LLM) é composto com os dados obtidos pela busca vetorial. Assim, gerando uma resposta contextualizada com base nos dados que estão no grafo de conhecimento, concluindo a implementação do RAG. O resultado é apresentado na Figura 1.23. Nela é possível observar uma resposta mais direta e correlata com os filmes existentes no dataset.

```python
from langchain.chains import LLMChain
from langchain_community.llms import GPT4All
from langchain_core.prompts import PromptTemplate

template = """
{question}
"""

prompt = PromptTemplate.from_template(template)

llm = GPT4All(
  model="Meta-Llama-3-8B-Instruct.Q4_0.gguf",
  backend="gptj", verbose=True
)

llm_chain = LLMChain(prompt=prompt, llm=llm)

question = "Which movies are about love and action?"

llm_chain.run(question)
```

Figure 1.20. Consulta utilizando LLM sem RAG

```
There are plenty of movies that blend love and action. Here are a few
notable ones:

"Mr. & Mrs. Smith" (2005)
"True Lies" (1994)
"The Princess Bride" (1987)
"Romeo Must Die" (2000)
"The Adjustment Bureau" (2011)
"Kill Bill: Volume 1 & 2" (2003-2004)
```

Figure 1.21. Resultado da execução do LLM sem RAG conforme código da Figura 1.20

```python
def print_answer(answer):
  response = ""
  for item in answer:
    response += f"""Score: {item["score"]}
Title: {item["title"]}
Overview: {item["overview"]}

"""
  return response

def ask_with_rag(question):
  retrieved_info = print_answer(neo4j_vector_search(question))
  template = f"""
  Based on the following context:
  {retrieved_info}
  Please answer (be objective, avoid introduction and
  conclusive texts): {{question}}
  """

  prompt = PromptTemplate.from_template(template)

  llm_chain = LLMChain(prompt=prompt, llm=llm)

  llm_chain.run(question)

ask_with_rag("Which movies are about love and action?")
```

Figure 1.22. Consulta utilizando LLM com RAG

```
The Lovers: Epic romance and time travel adventure with love
and action across different periods and continents.

Maleficent: Romance and action with a focus on Maleficent's journey
of betrayal and redemption.

Pirates of the Caribbean: On Stranger Tides: Romance and action
involving Jack Sparrow and a past love.

Spider-Man 2: Romance and action with Peter Parker struggling with his
superhero identity and feelings for Mary Jane Watson.
```

Figure 1.23. Resultado da execução do LLM com RAG conforme código da Figura 1.22

1.5. Técnicas Avançadas de Interligação e Consulta em Grafos de Conhecimento para *Advanced RAG*

Como visto, a tecnologia de Geração com Recuperação Aumentada (RAG), introduzida por [Lewis et al. 2020], é um paradigma inovador no campo do processamento de linguagem natural que combina técnicas de recuperação de informação com geração de linguagem baseada em modelos de linguagem. A abordagem avançada, conhecida como

Advanced RAG, procura ampliar este conceito integrando métodos mais sofisticados de recuperação e contextualização dos dados recuperados antes da geração de respostas. Isso pode inclui o uso de índices vetoriais construídos a partir de *embeddings* densos, que permitem consultas semânticas mais profundas e precisas, e a aplicação de mecanismos de atenção que direcionam o modelo de geração para considerar os segmentos mais relevantes do conteúdo recuperado. Essas inovações visam melhorar a qualidade e a relevância das respostas geradas, tornando o RAG uma ferramenta ainda mais poderosa para aplicações que vão desde assistentes virtuais até sistemas avançados de suporte à decisão.

O *Advanced RAG* incorpora técnicas de aprendizado de máquina para ajustar dinamicamente os métodos de recuperação baseados em feedback do usuário ou em análises de eficácia em tempo real. Por exemplo, estratégias de re-ranking baseadas em aprendizado de reforço podem ser utilizadas para ajustar a ordem dos nós recuperados conforme a interação do sistema com os usuários, melhorando assim a capacidade de adaptação e personalização do sistema. Este nível de personalização e interatividade abre novas possibilidades para a criação de sistemas de resposta a perguntas e de geração de conteúdo que são sensíveis ao contexto específico de suas aplicações, oferecendo respostas que são não apenas corretas, mas também contextualmente apropriadas e informativas.

Para implementar o *Advanced RAG*, necessariamente são necessarias etapas de pré e pós-processamento. Limpeza de dados, concatenação de atributos dos nós, filtragem com base da estrutura do grafo são exemplos de atividades que podem ser realizadas na etapa de pre-processamento. Já na etapa de pós-processamento, pode-se incluir calculo dos embeddings da resposta para comparação com os embeddings dos nós e da pergunta, limpeza da resposta pare remoção de texto desnecessário, entre outros.

Um exemplo um pouco mais avançado é apresentado na Figura 1.24. Inicialmente é gerado um template de prompt mais complexo, cujo foco é a construção de consultas diretamentas ao banco de dados Neo4j. Para tal, é utilizado o schema (estrutura) do banco de dados. A Figura 1.25 apresenta um exemplo de utilização dessa abordagem. Nota-se que a consulta retornada funciona perfeitamente no banco de dados. A Figura 1.26 já apresenta a execução de uma pergunta em linguagem natural diretamente no banco de dados. Observe que a consulta envolve 3 tipos de nós e 2 tipos de relacionamento. Essa execução direta deve ser realizada com cautela. Aconselha-se a adção de outros mecanismos de pós-processamento para evitar falhas de segurança.

```
def generate_cypher(question):
  kg.refresh_schema()

  CYPHER_GENERATION_TEMPLATE = f"""Task:Generate Cypher statement to
  query a graph database.

  Instructions:
  Use only the provided relationship types and properties in the
  schema. Do not use any other relationship types or properties
  that are not provided.

  Schema:
  {kg.schema}
  Note: Do not include any explanations or apologies in
  your responses. Do not respond to any questions that
  might ask anything else than for you to construct a Cypher
  statement.
  Do not include any text except the generated Cypher statement.

  Examples: Here are a few examples of generated Cypher
  statements for particular questions:

  # Top 10 movies with biggest loss
  MATCH (m:Movie)
  WHERE m.budget > 0 AND m.revenue > 0
  WITH m, (m.budget - m.revenue) AS loss
  WHERE loss > 0
  RETURN m.title, m.budget, m.revenue, loss
  ORDER BY loss DESC
  LIMIT 10

  The question is:
  {{question}}"""

  prompt = PromptTemplate.from_template(CYPHER_GENERATION_TEMPLATE)

  llm_chain = LLMChain(prompt=prompt, llm=llm)

  llm_chain.run(question)
```

Figure 1.24. Função para geração de consultas Cypher a partir de linguagem natural

```
generate_cypher("What are the Top 3 most expensive Action movies?")

# Saida retornada:
# MATCH (m:Movie)-[:HAS_GENRE]->(:Genre {name: "Action"})
# RETURN m.title, m.budget
# ORDER BY m.budget DESC
# LIMIT 3
```

Figure 1.25. Geração de consultas Cypher a partir de linguagem natural

```
kg.query(generate_cypher("""What are the Top 3 cheaper Adventure movies
which have the keyword ocean?"""))

# Saida Esperada:
# [{'m.title': 'Waterworld', 'm.budget': 175000000},
# {'m.title': "Pirates of the Caribbean: At World's End",
#  'm.budget': 300000000}]

# Consulta:
# MATCH (m:Movie)-[:HAS_GENRE]->(:Genre {name: "Adventure"})
# MATCH (m)-[:HAS_KEYWORD]->(:Keyword {name: "ocean"})
# RETURN m.title, m.budget
# ORDER BY m.budget ASC
# LIMIT 3

```

Figure 1.26. Consultando diretamente no banco de dados com linguagem natural

1.6. Conclusão

Este capítulo explorou conceitos fundamentais e aplicações práticas dos Grafos de Conhecimento e da tecnologia de Geração com Recuperação Aumentada (RAG) no contexto de Modelos de Linguagem de Grande Escala (LLMs). Iniciou-se com uma introdução aos Grafos de Conhecimento, destacando como essas estruturas facilitam a representação semântica e interconectada de informações, crucial para tarefas de IA que exigem um entendimento profundo das relações e contextos.

Em seguida foi detalhado como a tecnologia RAG melhora significativamente a funcionalidade dos LLMs, permitindo que esses modelos não só gerem textos com base em seu treinamento prévio, mas também busquem e integrem informações em tempo real durante a geração de respostas. Essa capacidade de atualização dinâmica é vital para aplicações que necessitam de informações precisas e contextualmente relevantes.

Adicionalmente, discutimos o uso prático do framework LangChain e do banco de dados Neo4j para implementar essas tecnologias. LangChain serve como uma ferramenta facilitadora para a integração de LLMs com bases de dados, enquanto Neo4j permite a manipulação eficiente e intuitiva de grafos de conhecimento, utilizando sua poderosa linguagem de consulta, Cypher.

O capítulo também abordou técnicas avançadas como o Advanced RAG, que introduz métodos refinados de pré e pós-recuperação para melhorar a precisão e a relevância das interações de LLMs. Estas técnicas representam a vanguarda do processamento de linguagem natural, combinando recuperação de informação com geração de linguagem de maneira inovadora.

1.6.1. Tópicos para Estudo Posterior

Para futuros estudos, seria benéfico explorar as seguintes áreas:

- Desenvolvimento de Modelos RAG Personalizados: Investigar como modelos personalizados de RAG podem ser desenvolvidos para atender necessidades específicas

de diferentes domínios de aplicação.

- Otimização de Performance em Grafos de Conhecimento: Estudar métodos para melhorar a eficiência de consultas em grafos de conhecimento, especialmente em bases de dados de grande escala.
- Segurança e Privacidade em RAG: Examinar as implicações de segurança e privacidade ao integrar RAG com LLMs, especialmente quando manipulando dados sensíveis.
- Integração de Novas Fontes de Dados em Grafos: Avaliar técnicas para integrar continuamente novas fontes de dados aos grafos de conhecimento, mantendo a consistência e a precisão das informações.
- Avaliação Comparativa de Frameworks e Ferramentas: Realizar estudos comparativos entre diferentes ferramentas e frameworks utilizados na implementação de RAG e grafos de conhecimento, identificando suas forças e limitações.

Aprofundar nestes tópicos permitirá não apenas uma melhor compreensão das tecnologias discutidas, mas também impulsionará inovações que podem transformar a maneira como interagimos e utilizamos sistemas baseados em IA e LLMs.

1.7. Sobre os autores

Otávio Calaça Xavier

Professor nas instituições UFG e IFG, cursando atualmente doutorado em Ciência da Computação, com foco em *Graph Neural Networks* e *Retrieval-Augmented Generation (RAG)*. Possui mestrado em Ciência da Computação pela UFG, obtido em 2011, com ênfase em Web Semântica e *Linked Data*. Com 19 anos de experiência em Desenvolvimento de Aplicações Web e Administração de Bancos de Dados, além de 15 anos em Arquitetura de Software e Liderança de Equipes. Desde 2007, atuou como palestrante em mais de 100 eventos de tecnologia em cidades como Goiânia, Brasília, São Paulo, Foz do Iguaçu, Rio de Janeiro e Porto Alegre. Leciona há mais de dez anos em cursos de graduação, pós-graduação e capacitação. Atua também como consultor em Aprendizado de Máquina, Ciência de Dados, Engenharia e Arquitetura de Software. Otávio é o autor que irá ministrar o mini-curso.

Anderson da Silva Soares

Professor do Instituto de Informática da Universidade Federal de Goiás, onde é membro permanente dos programas de mestrado e doutorado em Ciência da Computação. Foi vice-coordenador do programa de doutorado entre 2015 e 2016 e atuou como editor associado do *Journal of Computer Science* de 2015 a 2018. É um renomado pesquisador nas áreas de aprendizado de máquina, deep learning e otimização com heurísticas. Fundador do laboratório Deep Learning Brasil, Anderson também é presidente da comissão de criação e atual

coordenador do Bacharelado em Inteligência Artificial na UFG. Representante brasileiro na *Global Partnership on Artificial Intelligence (GPAI)*, realizou captações significativas de mais de 200 milhões de reais em P&D, proporcionando bolsas para alunos em projetos com empresas como Data-H, Copel Distribuição, iFood, entre outros. Foi fundador e diretor-geral do Centro de Excelência em Inteligência Artificial de Goiás (unidade Embrapii), onde atualmente atua como coordenador científico. Várias de suas iniciativas de P&D geraram startups *spin-offs*. Como atividade de extensão, coordenou a Olimpíada Brasileira de Robótica em Goiás e no Distrito Federal, e continua atuando como voluntário na organização. Além disso, é entusiasta e mantenedor do núcleo de robótica Pequi Mecânico, focado no nível universitário.

References

[Allemang and Hendler 2011] Allemang, D. and Hendler, J. (2011). *Semantic web for the working ontologist: effective modeling in RDFS and OWL*. Elsevier.

[Angles et al. 2017] Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., and Vrgoč, D. (2017). Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)*, 50(5):1–40.

[Angles and Gutierrez 2008] Angles, R. and Gutierrez, C. (2008). Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39.

[Bordes et al. 2013] Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., and Yakhnenko, O. (2013). Translating embeddings for modeling multi-relational data. *Advances in neural information processing systems*, 26.

[Brown et al. 2020] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

[Devlin et al. 2018] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

[Gao et al. 2023] Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., and Wang, H. (2023). Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*.

[Henderson et al. 2017] Henderson, M., Al-Rfou, R., Strope, B., Sung, Y.-H., Lukács, L., Guo, R., Kumar, S., Miklos, B., and Kurzweil, R. (2017). Efficient natural language response suggestion for smart reply. *arXiv preprint arXiv:1705.00652*.

[Hogan et al. 2021] Hogan, A., Blomqvist, E., Cochez, M., D'amato, C., Melo, G. D., Gutierrez, C., Kirrane, S., Gayo, J. E. L., Navigli, R., Neumaier, S., Ngomo, A.-C. N., Polleres, A., Rashid, S. M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S., and Zimmermann, A. (2021). Knowledge graphs. *ACM Comput. Surv.*, 54(4).

[Lewis et al. 2020] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.

[Manning 2008] Manning, C. D. (2008). *Introduction to information retrieval*. Syngress Publishing,.

[Mikolov et al. 2013] Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

[Miller 2013] Miller, J. J. (2013). Graph database applications and concepts with neo4j. In *Proceedings of the southern association for information systems conference, Atlanta, GA, USA*, volume 2324, pages 141–147.

[Nickel et al. 2016] Nickel, M., Murphy, K., Tresp, V., and Gabrilovich, E. (2016). A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33.

[Pandya and Holia 2023] Pandya, K. and Holia, M. (2023). Automating customer service using langchain: Building custom open-source gpt chatbot for organizations. *arXiv preprint arXiv:2310.05421*.

[Paulheim 2017] Paulheim, H. (2017). Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic web*, 8(3):489–508.

[Pennington et al. 2014] Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

[Robinson et al. 2015] Robinson, I., Webber, J., and Eifrem, E. (2015). *Graph databases: new opportunities for connected data*. " O'Reilly Media, Inc.".

[Rogers et al. 2021] Rogers, A., Kovaleva, O., and Rumshisky, A. (2021). A primer in bertology: What we know about how bert works. *Transactions of the Association for Computational Linguistics*, 8:842–866.

[Suchanek et al. 2007] Suchanek, F. M., Kasneci, G., and Weikum, G. (2007). Yago: a core of semantic knowledge. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, page 697–706, New York, NY, USA. Association for Computing Machinery.

[Zou 2020] Zou, X. (2020). A survey on application of knowledge graph. In *Journal of Physics: Conference Series*, volume 1487, page 012016. IOP Publishing.

sbbd:02

Capítulo 2

Desvendando Planos de Execução: Uma Abordagem Visual para Exploração e Otimização de Consultas

Sergio Mergen

Resumo

SQL é uma linguagem central para interação com bancos de dados relacionais, amplamente adotada graças à sua padronização e versatilidade. No entanto, o uso ineficiente dessa linguagem pode acarretar problemas de desempenho, mesmo em consultas funcionalmente corretas. A falta de conhecimento sobre os planos de execução, que determinam como as consultas são processadas pelos Sistemas de Gerenciamento de Banco de Dados (SGBDs), é um fator chave nesses problemas. Este capítulo explora como os planos de execução influenciam o desempenho das consultas e a complexidade envolvida na escolha do plano mais eficiente, oferecendo insights valiosos para a otimização de consultas.

Abstract

SQL is a central language for interaction with relational databases, widely adopted due to its standardization and versatility. However, inefficient use of this language can lead to performance issues, even in functionally correct queries. The lack of knowledge about execution plans, which determine how queries are processed by Database Management Systems (DBMS), is a key factor in these problems. This chapter explores how execution plans influence query performance and the complexity involved in choosing the most efficient plan, offering valuable insights for query optimization.

2.1. Introdução

A *Structured Query Language* (SQL) é a linguagem utilizada para interagir com bancos de dados relacionais. Devido a sua versatilidade, tornou-se essencial para a manipulação e consulta de dados em sistemas de gerenciamento de bancos de dados (SGBDs). Sua padronização pelo *American National Standards Institute* (ANSI) e pela *International*

Organization colaborou para consolidar sua ampla adoção e uso consistente em diversas plataformas de banco de dados.

Embora seja uma linguagem poderosa, o uso ineficiente do SQL pode resultar em sérios problemas de desempenho e escalabilidade. É comum que desenvolvedores escrevam consultas que são funcionalmente corretas, mas apresentem desempenho insatisfatório. Esses problemas ocorrem quando as consultas retornam os resultados esperados de maneira ineficiente. Além disso, também são frequentes erros de lógica, resultando em consultas incorretas [Taipalus 2020]. Um dos principais motivos para essas dificuldades é o desconhecimento sobre o processamento interno das consultas pelos Sistemas de Gerenciamento de Banco de Dados (SGBDs). Para otimizar o desempenho, é crucial entender como os planos de execução funcionam, pois eles determinam o caminho que a consulta percorre até o resultado final.

Neste capítulo, vamos explorar detalhadamente os planos de execução em bancos de dados relacionais, examinando como eles influenciam o desempenho das consultas. Além disso, examinaremos a complexidade envolvida na escolha de planos de execução, dado que diversos fatores devem ser considerados, como o tamanho dos dados, a seletividade dos filtros e a disponibilidade de índices, para determinar o caminho mais eficiente. Essa análise é crucial, pois mesmo consultas aparentemente simples podem ser executadas de maneiras muito diferentes, com variações significativas no desempenho.

Ao longo do capítulo, entenderemos que a existência de múltiplos caminhos possíveis para a execução de uma consulta se deve à flexibilidade e poder expressivo da SQL. No entanto, essa flexibilidade também traz a complexidade de determinar o plano de execução mais eficiente. A compreensão desses conceitos permitirá aos leitores desenvolverem habilidades que podem ser úteis para escrever consultas melhores.

Por fim, apresentaremos uma linguagem de consulta alternativa, que oferece maior controle sobre a execução das consultas. Discutiremos suas vantagens, como a possibilidade de manipulação direta dos planos de execução e de criação de operadores de transformação customizados, ressaltando como essa ferramenta pode auxiliar o desenvolvedor na escrita de consultas.

2.2. Planos de Execução

Um plano de execução é uma representação detalhada da estratégia escolhida pelo otimizador de consultas para executar uma consulta. Ele inclui informações como a ordem das operações, seu custo, métodos de acesso aos dados e uso de índices [Lan et al. 2021].

Os SGBDs modernos oferecem recursos para visualizar e analisar esses planos de execução. No entanto, não existe um padrão universalmente aceito. Cada sistema tem sua própria forma de visualizar e detalhar os planos. Para superar essa limitação, adotaremos um formato de representação genérico ao longo deste capítulo. Esse formato é baseado no modelo Volcano, que utiliza uma estrutura de operadores relacionais organizados em forma de árvore para descrever o fluxo de dados [Graefe 1994]. Essa abordagem permitirá a compreensão dos conceitos fundamentais, independentemente do SGBD.

Os planos de execução podem ser compreendidos como árvores, cujos nós funcionam como operadores responsáveis por algum tipo de transformação de dados. Pensando

numa representação genérica, esses nós podem ser categorizados em três tipos principais, dependendo do número de entradas que recebem e da função que desempenham. Os tipos são:

- Operadores Unários: Esses operadores processam dados recebendo entrada de um único operador precedente.
- Operadores Binários: Operadores que necessitam de duas entradas para realizar suas operações.
- Nós fontes: Estes são os pontos de entrada na árvore de execução, onde os dados são acessados na sua forma original. Não recebem dados de outros operadores.

Os operadores combinados formam uma árvore que representa um plano de execução. Os nós folha são as fontes de dados, enquanto a árvore possui um único nó raiz que simboliza o resultado da consulta. Na representação que utilizaremos, o nó raiz será posicionado na parte superior da árvore. Nesse caso, as informações fluem de baixo para cima, partindo dos nós folha em direção ao nó raiz. Ao adotar esse formato genérico de estruturação e representação, seremos capazes de discutir e ilustrar as principais estratégias de execução de consultas de maneira clara e consistente. Essa abordagem facilitará a identificação de padrões e práticas recomendadas, independentemente das idiossincrasias de um SGBD específico.

Ao longo do capítulo, abordaremos vários operadores que compõem os planos de execução. Esses operadores serão explicados à medida que forem aparecendo, proporcionando uma compreensão gradual e prática de como eles funcionam. Serão usados nomes em inglês para se referir aos operadores. Essa escolha foi feita porque a maioria das ferramentas, documentação e interfaces de bancos de dados usa terminologia em inglês. Assim, facilitamos a consulta e compreensão das referências técnicas e da literatura especializada.

Nos exemplos apresentados, utilizaremos o mesmo esquema de dados, composto por filmes, membros do elenco e membros da equipe de produção. O esquema, em uma versão textual simplificada, está descrito na Figura 2.1. Chaves primárias são indicadas pelos atributos sublinhados, enquanto as chaves estrangeiras estão representadas pela palavra 'referencia'.

filme(<u>idF</u>, titulo, ano)
pessoa(<u>idP</u>, nome)
elenco(<u>idF, idP</u>, personagem, ordem)
    idF referencia filme, idP referencia pessoa
producao(<u>idF,idP</u>, funcao)
    idF referencia filme, idP referencia pessoa

Figura 2.1: Modelo simplificado do esquema do banco relacional de filmes

2.3. Organização dos Registros

Para estudar planos de execução, é crucial entender como os registros estão fisicamente organizados na memória secundária, pois isso é um fator determinante para o custo de acessá-los. Este tema é tão importante que é frequentemente abordado em livros sobre fundamentos de bancos de dados. Obras de referência exploram a relação entre a organização física dos dados e o desempenho das consultas, destacando como a escolha de estruturas de armazenamento afeta o custo computacional ao acessar grandes volumes de dados [Sumathi and Esakkirajan 2007].

De modo geral, o custo para acessar um conjunto de registros é menor se esses registros estiverem próximos no disco. Isso ocorre devido à forma como o sistema operacional, a controladora do disco e o gerenciamento de memória lidam com a transferência de dados.

Em primeiro lugar, existe o custo de localização dos registros. Em discos magnéticos, a localização envolve movimentos mecânicos dos cabeçotes de leitura/escrita para encontrar os setores corretos. Quanto mais distantes estiverem os registros de interesse uns dos outros, maior será o tempo necessário para reposicionar os cabeçotes e localizar os dados. Essa busca é um dos fatores mais lentos no acesso a dados em discos magnéticos. Portanto, quando os registros estão fisicamente próximos, o tempo de busca é reduzido, resultando em um acesso mais rápido.

Além disso, existe o custo de transferência dos registros para a memória. As transferências estão intimamente ligadas ao tamanho dos *clusters* do sistema de arquivos e às páginas de memória controladas pelo sistema de gerenciamento de memória. Quando uma aplicação solicita ao sistema operacional dados armazenados na memória secundária, normalmente o *cluster* inteiro é carregado, a fim de otimizar a transferência e o uso da memória. Dessa forma, registros presentes no mesmo *cluster* são carregados em uma única operação de E/S. Esse é um dos motivos pelos quais alguns SGBDs adotam o conceito de página de banco de dados, que é uma unidade de armazenamento e transferência de dados geralmente alinhada aos *clusters* do sistema de arquivos. Ao alinhar as páginas do banco de dados com os *clusters*, os SGBDs podem maximizar a eficiência das operações de leitura e escrita, carregando múltiplos registros em uma única operação.

Outra técnica relevante nesse contexto é a leitura antecipada (*read-ahead*). A leitura antecipada é uma técnica que pode ser implementada em diversos níveis, como na controladora de disco e no sistema operacional. Ela antecipa a necessidade de leitura de blocos de dados sequenciais. Quando um bloco de dados é lido, os blocos adjacentes também são carregados e armazenados em um *buffer*, para que possam ser acessados rapidamente se forem necessários posteriormente. Esse mecanismo reforça a ideia de que o custo de acesso aos registros é menor quando os registros estão fisicamente próximos, uma vez que a leitura antecipada pode efetivamente diminuir o tempo de espera para acessar dados sequenciais.

A partir da noção fundamental de custo no acesso, podemos discutir como os registros são fisicamente organizados no disco e como isso influencia o custo de acesso. Duas das principais formas de organizar fisicamente os registros são: organização *Heap* (*Heap File*) e organização ordenada (*Sorted File*) [Hammer and Schneider 2018].

Na organização *Heap*, os registros são armazenados no arquivo na ordem em que são inseridos. À medida que ocorrem remoções, espaços vazios (buracos) são criados no arquivo e são preenchidos pelos registros inseridos posteriormente. Consequentemente, conforme a tabela é modificada, registros com chaves primárias próximas podem se distanciar fisicamente no disco. Esse afastamento pode prejudicar o acesso sequencial aos registros, especialmente quando são necessárias leituras em ordem de chave primária.

Por outro lado, na organização ordenada, os registros são fisicamente ordenados pela chave primária. Isso significa que registros com chaves primárias similares estão próximos uns dos outros no disco. Essa proximidade física reduz o custo de acesso em leituras sequenciais, pois o sistema pode ler blocos de dados contíguos de forma eficiente. Em sistemas de bancos de dados relacionais, a organização ordenada geralmente é implementada utilizando árvores B+.

2.3.1. Árvores B+

As árvores B+ são estruturas de dados usadas para armazenar pares <chave,valor>, sendo os pares ordenados pela chave. Uma das principais características dessas árvores é que todos os valores são armazenados nos nós folha. Além disso, as folhas são encadeadas entre si. Esse encadeamento permite percorrer todos os valores simplesmente acessando e navegando pelos nós do último nível, a partir do nó mais à esquerda [Bertino et al. 2012].

As árvores B+ dispõem de mecanismos de busca eficientes para localizar valores armazenados com base em uma chave de referência, seja para uma busca por igualdade ou intervalo. Em vez de percorrer todas as folhas, o algoritmo pode iniciar na raiz e navegar pelos nós intermediários até alcançar os nós folha que contêm os valores desejados.

O número de filhos que cada nó pode ter depende de um parâmetro conhecido como ordem da árvore. Quanto maior for a ordem da árvore, mais filhos cada nó pode ter. Isso resulta em uma estrutura com menor profundidade, o que reduz o número de acessos necessários para chegar a um nó folha a partir da raiz.

As árvores B+ são comumente utilizadas em sistemas de banco de dados para a organização física dos registros. Quando utilizadas para essa finalidade, as chaves correspondem às chaves primárias, enquanto os valores correspondem aos registros completos. O encadeamento do nível folha permite que os registros sejam lidas de forma sequencial. Além dessa finalidade, as árvores B+ também são usadas em SGBDs para fins de indexação, como será apresentado na próxima seção.

2.3.2. Índices Primários e Secundários

Os índices são fundamentais para melhorar a eficiência de acesso aos registros. Eles permitem que as consultas sejam executadas mais rapidamente ao fornecer caminhos alternativos para encontrar os registros desejados sem a necessidade de acessar todos os registros da tabela. Índices são construídos sobre uma chave de busca, que é um conjunto de colunas da tabela, e aceleram a localização de registros que tenham valores específicos para esse conjunto de colunas.

Existem dois tipos principais de índices: índices primários e índices secundários. Em um **índice primário**, a chave de busca é a chave primária da tabela. Já em um **índice**

secundário, a chave de busca é qualquer outro conjunto de colunas da tabela. Por exemplo, na tabela filme, haveria um índice primário sobre a coluna 'idF'. Por outro lado, poderiam ser criados índices secundários sobre a coluna 'titulo' e 'ano', respectivamente.

Em ambos os tipos de índice, costuma-se usar árvores B+ como estrutura de indexação, e os índices podem ser classificados como índices clusterizados e não clusterizados [Manolopoulos et al. 2000].

- **Índice Clusterizado:** Um índice clusterizado é usado para organizar fisicamente os registros no arquivo. Ele não é apenas um índice lógico sobre a chave de ordenação; ele impõe uma reordenação física dos registros no armazenamento para corresponder à chave do índice.
- **Índice Não Clusterizado:** Um índice não clusterizado não tem função de organização. Ele é usado apenas para localizar registros com base em comparações sobre a chave de busca. Nesse caso, o nível folha da árvore B+ guarda um ponteiro que leva até o registro que possui a chave de busca.

Nas tabelas ordenadas mantidas por um índice B+, os índices primários são clusterizados. A própria árvore já armazena os registros por ordem de chave primária. Ou seja, a mesma estrutura de dados é usada para representar tanto o índice quanto os dados. Por exemplo, o componente denominado 'Filme' daria acesso ao índice primário sobre a tabela 'filme'. Nesse índice, cada entrada no nível folha teria o seguinte formato: <idF, registro completo>. Por outro lado, os índices secundários são estruturas a parte, que usam como ponteiro a chave primária do registro. Por exemplo, 'idx_ano' poderia ser o nome do componente que daria acesso ao índice secundário sobre a coluna 'ano'. Nesse índice, cada entrada no nível folha teria o seguinte formato: <ano, idF>. Para acessar o registro de filme, deve-se primeiro acessar o índice secundário 'idx_ano' para obter o ponteiro (a chave primária 'idF'). Então, deve-se acessar o índice primário 'Filme' usando a chave primária obtida. Essa abordagem permite que os registros sejam deslocados no arquivo conforme ocorrem inserções e remoções, sem que os índices secundários precisem ser atualizados frequentemente. Contudo, geram uma sobrecarga na localização do registro, como veremos mais adiante.

Nas tabelas *Heap*, as árvores B+ não tem função de organização dos registros. Ou seja, nenhum índice é clusterizado. Como os registros permanecem no mesmo lugar, os ponteiros dos índices guardam a localização absoluta do registro no arquivo. Vamos chamar essa localização de *Record Id*(RID). Por exemplo, no índice sobre ano, cada entrada no nível folha teria o seguinte formato: <ano, rid>. O índice primário também precisa ser representado por uma estrutura a parte. Por exemplo, enquanto o componente 'Filme' dá acesso aos registros organizados por uma Heap, o componente 'idx_filme' daria acesso ao índice primário da tabela filme, com as entradas no nível folha <idF, rid> ordenadas por idF;

Em síntese, para tabelas ordenadas, cujos índices secundários usam a chave primária como ponteiro, é necessário uma busca adicional sobre o índice primário para localizar o registro completo. Por outro lado, em tabelas *Heap*, cujos índices secundários usam o RID como ponteiro, o acesso ao registro pode ser realizado de forma direta.

A diferença no gerenciamento dos ponteiros entre as tabelas *Heap* e as tabelas ordenadas pode ter um impacto significativo no desempenho das consultas. Por exemplo, existem cenários onde a ordenação física dos registros nas tabelas ordenadas pode trazer vantagens, como em consultas que se beneficiam de leituras ordenadas. No entanto, quando se usa índices secundários, a necessidade de um passo adicional para acessar o registro completo pode tornar algumas operações mais lentas, especialmente quando lidamos com grandes volumes de dados.

Esses dois tipos de organização são usadas na prática opor diferentes SGBDs. Por exemplo, o MySQL(*engine* InnoDB) utiliza tabelas ordenadas, com índices clusterizados para chaves primárias. Já o PostgreSQL usa tabelas *Heap*, sem índices clusterizados. Esse aspecto é fundamental para entender porque SGBDS muitas vezes usam caminhos diferentes para chegar até a mesma resposta. Em seções posteriores, exploraremos como a organização dos registros afeta a geração de planos de execução e como a indexação pode ser usada para seja possível gerar planos mais eficientes.

2.4. Acesso aos dados

Como vimos na seção anterior, os registros podem estar armazenados em estruturas *Heap* ou árvores B+, enquanto as entradas de um índice são armazenadas em árvores B+. Uma abstração aceitável é considerar que ambos os componentes provem acesso a coleções de registros. Uma estrutura *Heap* dá acesso direto aos registros de uma tabela. Já uma árvore B+ dá acesso a registros compostos por <chave, valor>.

Essa visão unificada permite definir duas formas básicas de acesso que são válidas independente da forma de organização. Isso facilita a compreensão de como os dados são acessados durante a execução de uma consulta. As duas formas são: *Scan* (varredura) ou *Seek* (busca).

O *Scan* percorre todos os registros da coleção. Em uma árvore B+, essa varredura acessa as entradas/registros de todos os nós folha da árvore, a partir do nó folha mais à esquerda. Isso assegura que as entradas do índice sejam lidas por ordem de chave de busca. Em uma estrutura *Heap*, o *Scan* acessa todos os registros conforme a sua posição no arquivo. Nesse caso, os registros não necessariamente seguem uma ordem específica.

Já o *Seek* tem por objetivo localizar registros que satisfaçam algum predicado de filtragem. Se o *Seek* for feito sobre uma árvore B+, usando um critério de filtragem suportado pelo índice, a busca é resolvida de forma eficiente, através de navegação pelos nós da árvore B+. Caso contrário, a busca é resolvida percorrendo todos os nós do nível folha. Se o *Seek* for feito diretamente sobre uma estrutura *Heap*, todos os registros são varridos na busca por correspondências.

A escolha entre *Scan* e *Seek* é crucial para a otimização do desempenho das consultas, e entender as situações em que cada método é vantajoso pode ajudar a tomar decisões informadas sobre a criação e uso de índices no banco de dados. Para exemplificar, vamos demonstrar duas situações simples em que *Scans* e *Seeks* podem ser usados: projeções e filtros. As projeções são úteis quando queremos escolher que colunas devem ser retornadas. Eles equivalem a um corte vertical na coleção de registros acessada. Os filtros, por sua vez, são úteis quando queremos escolher que registros devem ser retornados.

Eles equivalem a um corte horizontal na coleção.

A Figura 2.2 mostra um exemplo de projeção, onde uma única coluna interessa: título. O operador *Projection* delimita a coluna de interesse. No plano a), o título é obtido por meio de um *Scan* sobre a tabela filme. Já no plano b), o título é obtido por meio de um *Scan* sobre um índice criado sobre essa coluna. Podemos chamar essas duas formas de acesso de *Table Scan* e *Index Scan*, respectivamente. Como os registros presentes no índice são menores do que os registros presentes na tabela, o *Index Scan* é o caminho mais eficiente, pois exige menor esforço para acesso. Aliás, quando uma consulta é resolvida usando apenas o índice, sem necessidade de acessar a tabela, diz-se que o índice é de cobertura (*covering index*).

Figura 2.2: Exemplos de consultas com projeção

Apesar de serem úteis para diversas finalidades, os índices são mais importantes para consultas contendo filtros. A Figura 2.3 mostra exemplos envolvendo um operador de filtragem. Os planos exibidos na figura são simples, mas ajudam a entender as formas como os operadores *Scan* e *Seek* lidam como essa questão, e como os filtros são resolvidos em cenários mais complexos.

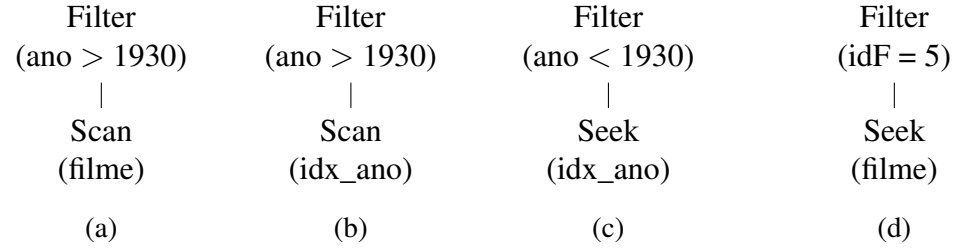


Figura 2.3: Exemplos de consultas com filtro

No plano a), é usado um *Table Scan* para encontrar filmes que satisfaçam o critério de filtro. A varredura pode ser vantajosa em tabelas pequenas, onde o custo de ler todos os registros é baixo, ou quando uma grande proporção dos registros na tabela satisfizer o filtro (baixa seletividade). No exemplo, como o filtro é pouco seletivo (filmes a partir de 1930), possivelmente o *Scan* seja uma boa opção.

Por sua vez, o plano b) utiliza um *Index Scan*. Como o custo para ler de um índice é menor do que o custo para ler de uma tabela, a varredura de índice é considerada mais adequada. No entanto, é importante destacar que os dois planos são distintos. O plano a) fornece acesso a todas as colunas da tabela, enquanto o plano b) só permite acessar as colunas que fazem parte do índice. Caso as colunas do índice sejam suficientes para a consulta, temos um *covering index*, e o plano b) é considerado mais eficiente.

No plano c), um *Seek* é utilizado no índice sobre a coluna ano para localizar diretamente o registro desejado, sem a necessidade de ler toda a tabela. Este método é especialmente eficiente em tabelas grandes, onde varrer todos os registros seria muito custoso, ou quando a consulta é seletiva. No exemplo, como o filtro é seletivo (filmes anteriores a 1930), possivelmente o *Seek* seja efetivamente a melhor opção. No entanto, assim como no plano b), apenas as colunas presentes no índice estão disponíveis para uso. Caso outras colunas sejam relevantes (como título, por exemplo), deve-se obtê-las por um método de complementação. Porém, existe um custo associado a essa complementação, como será demonstrado na Seção 2.5.

Por fim, o plano d) mostra um filtro de igualdade sobre a chave primária da tabela filme, resolvido por meio de uma *Seek*. Neste exemplo, considera-se que a tabela seja organizada por uma árvore B+. Ou seja, o *Seek* consegue localizar o registro desejado de forma eficiente. Caso a tabela de dados utilizasse *Heap*, a busca deveria ser realizada sobre o índice primário (ex. 'idx_filme'). Essa busca retornaria o endereço físico do registro (RID), o que exigiria um método de complementação para obter o registro propriamente dito.

De modo geral, tanto o operador de *Scan* quanto de *Seek* podem se beneficiar da presença de índices. No entanto, isso não significa que índices devam ser criados sempre que houver possibilidade de uso. Em primeiro lugar, índices ocupam espaço e precisam ser atualizados sempre que os registros são atualizados. A existência de muitos índices pode levar a custos de espaço e de manutenção proibitivos. Além disso, nem sempre os otimizadores utilizam índices. A seletividade de um filtro, a presença de índices melhores, a necessidade de complementação, a estratégia usada para organização de arquivos ou mesmo a própria estrutura da consulta pode levar um índice a ser desprezado. As próximas seções exibem cenários que ajudam a explorar essa questão mais a fundo.

2.5. Junções

As junções são um dos elementos mais essenciais da linguagem SQL. Elas combinam dados relacionados de diferentes tabelas com base em condições especificas. Isso é particularmente vital em bancos de dados relacionais, onde os dados são distribuídos entre diversas tabelas e conectados através de chaves estrangeiras.

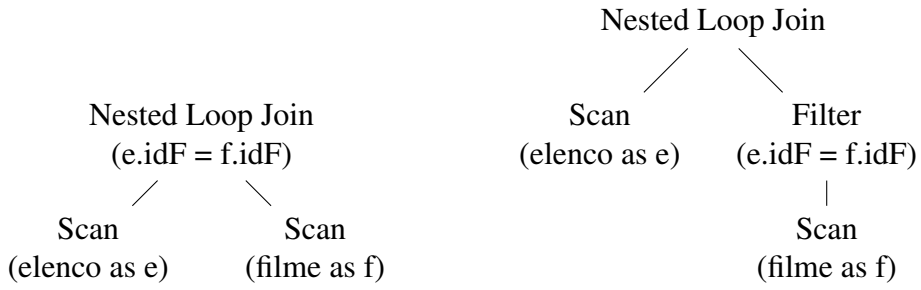
A escolha do algoritmo de junção pelo otimizador de consultas pode ter um impacto significativo no tempo de resposta. Portanto, ter um conhecimento sólido sobre os diferentes métodos de junção e seus comportamentos em diferentes cenários ajuda a escrever consultas mais eficientes e melhorar o desempenho geral do banco de dados. Ao longo deste capítulo, abordaremos três dos algoritmos de junção mais comumente utilizados: *Nested Loop Join*, *Hash Join* e *Merge Join*.

2.5.1. Nested Loop Join

O algoritmo clássico para processamento de junções é o 'Nested Loop Join'. As duas tabelas que participam da junção são chamadas de tabela interna e tabela externa. O algoritmo funciona combinando cada linha da tabela externa com cada linha da tabela interna e verificando se a condição de junção é satisfeita.

A Figura 2.4 apresenta um exemplo de como representar esse tipo de junção em um plano de execução. O objetivo é encontrar filmes e seus elencos. No exemplo, a tabela externa e a interna são elenco e filme, respectivamente. Em uma árvore construída de baixo para cima, o lado externo e interno geralmente são representados à esquerda e à direita da junção, respectivamente.

No exemplo, 'e.idF = f.idF' é o predicado de junção. No plano a), o predicado faz parte do algoritmo de junção. Isso significa que, dos registros de filme que chegam até a junção, apenas os que satisfizerem o predicado geram correspondências. No plano b), não há predicado na junção. Isso significa que será gerado um produto cartesiano: cada registro do lado externo será combinado com cada registro do lado interno. No entanto, um filtro do lado interno se encarrega de fazer com que apenas os registros correspondentes cheguem até a junção. Ou seja, o predicado de junção foi empurrado para baixo da árvore (*push down*). Neste exemplo, as duas opções são equivalentes em termos de desempenho, mas pode haver situações em que o *push down* compense, especialmente em casos em que o filtro está muito distante da tabela cujos registros devem ser filtrados.



(a) Com critério de junção (b) Sem critério de junção

Figura 2.4: Exemplos de consultas com Nested Loop Join

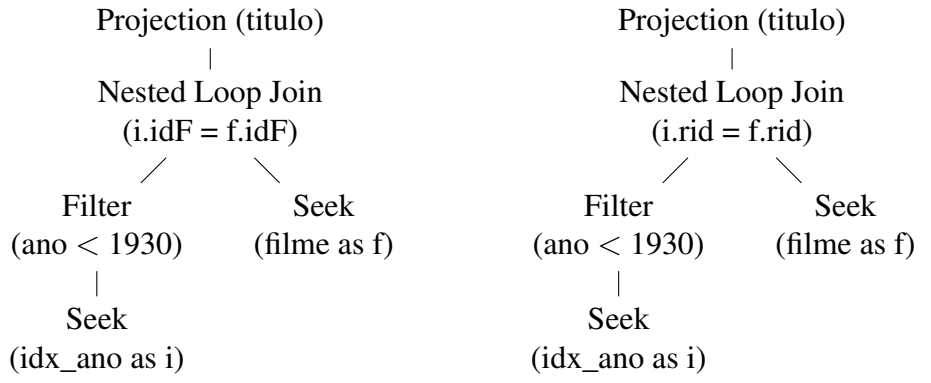
De modo geral, o *Nested Loop Join* é eficaz quando a tabela interna é muito pequena, mas pode se tornar inviável para tabelas maiores devido ao número excessivo de comparações necessárias. Uma variação mais eficiente deste algoritmo é o *Nested Loop Join* Indexado. Em vez de usar um *Scan* para percorrer toda a tabela interna, utiliza-se um *Seek*. Se houver um índice disponível no lado interno que atenda ao critério de junção, este algoritmo é preferível ao *Nested Loop Join* simples.

2.5.2. Complementação de índices

Na seção anterior vimos que, quando uma consulta pede de uma tabela mais colunas do que as presentes no índice a ser usado, isso exige que as entradas recuperadas do índice sejam complementadas com as colunas que faltam. Os SGBDs têm diferentes maneiras de realizar e representar essa complementação. Neste capítulo, utilizaremos o algoritmo de *Nested Loop Join* para isso. Esse pode não ser o algoritmo que é realmente empregado internamente pelos SGBDs para realizar a complementação, mas dá uma ideia bastante aproximada do custo envolvido nessa etapa.

A Figura 2.5 ilustra um exemplo em que o objetivo é obter títulos de filmes anteriores a 1930. O operador *Projection* se encarrega de recortar apenas essa coluna do

resultado da junção. Para encontrar os filmes que satisfazem o critério de filtragem, foi utilizado um *Seek* sobre o índice secundário 'ano'. Esse índice localiza registros compostos por <ano, ponteiro>. Como a consulta requer a coluna título, é necessário complementar os registros localizados com a informação faltante. Isso é realizado pelo algoritmo de junção. Para cada registro de índice localizado no lado externo, ocorre um *Seek* no lado interno.



(a) Usando índice clusterizado

(b) Usando índice não clusterizado

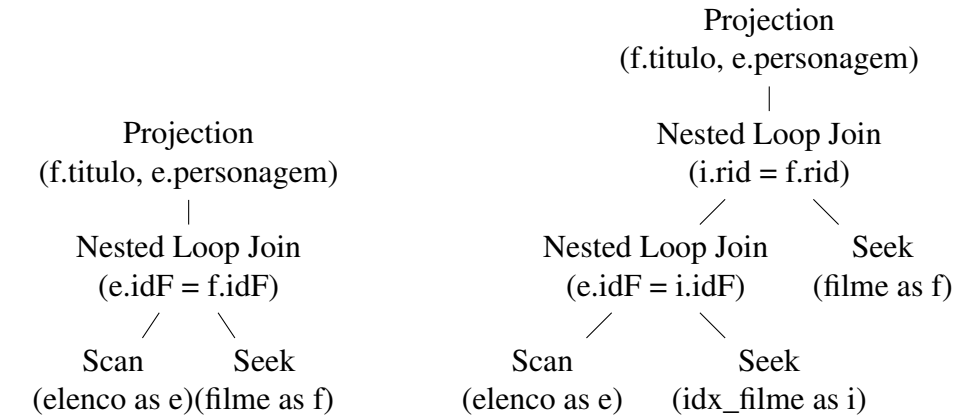
Figura 2.5: Exemplos de consultas com complementação de índice

O *Seek* depende do ponteiro obtido pelo lado externo. No plano a), considera-se que a tabela de filmes seja organizada por um índice clusterizado. Nesse caso, o ponteiro é composto pela chave primária, e o *Seek* interno realiza uma busca no índice primário. No plano b), considera-se que os dados sejam organizados em uma *heap*. Nesse caso, o ponteiro é composto pela localização física do registro (RID), e o *Seek* interno acessa o registro de forma mais direta, através da posição absoluta do registro no arquivo.

O custo dessa etapa de complementação não pode ser desprezado. Dependendo da seletividade do filtro, pode ser mais vantajoso realizar uma varredura sequencial no arquivo de filmes, em vez de acessar os registros de forma não sequencial, por meio do índice sobre ano.

A Figura 2.6 ilustra outro exemplo onde a complementação é necessária. Agora, o objetivo é obter títulos de filmes juntamente com os nomes de seus personagens. Para isso, é necessária uma junção entre as tabelas 'filme' e 'elenco'. Nos dois planos exibidos na figura, a tabela 'elenco' é usada no lado externo da junção. O que muda é a forma como ocorre a complementação. No plano a), considera-se que os dados sejam organizados por índices clusterizados. Nesse caso, o *Seek* realiza uma busca no índice primário 'filme'. Esse índice, sendo clusterizado, já disponibiliza a coluna 'titulo', requisitada pela projeção. No plano b), considera-se que os dados sejam organizados em uma *heap*. Nesse caso, o índice primário 'idx_filme' não disponibiliza a coluna 'titulo'. Dessa forma, duas junções são necessárias. A primeira, feita sobre o índice primário, recupera o RID. A segunda usa o RID para recuperar o título.

Como esse exemplo sugere, o *Nested Loop Join* indexado tende a ser mais barato quando a busca emprega um índice clusterizado, ou quando o índice já incluir a coluna a



(a) Usando índice clusterizado

(b) Usando índice não clusterizado

Figura 2.6: Exemplos de complementação de índices clusterizados e não clusterizados

ser buscada (titulo). Outro fator a considerar é que, quando os dados são organizados em uma *heap*, não existe garantia de que filmes com ids parecidos estejam próximos entre si no disco. Mesmo que os registros de elenco estivesse ordenados pelo id do filme, essa ordenação não traria benefícios à junção, pois a recuperação de um id diferente do anterior pode resultar em muitos acessos aleatórios no disco, e consequentemente, o carregamento de um número maior de páginas. É por esse motivo que bancos que optam por tabelas *heap* muitas vezes empregam um algoritmo de junção diferente, conforme apresentado na seção seguinte.

2.5.3. Hash Join

Juntamente com o *Nested Loop Join*, o *Hash Join* é um dos algoritmos de junção mais utilizados. Assim como o *Nested Loop Join* indexado, ele encontra correspondências por meio de buscas indexadas. No entanto, em vez de empregar índices preexistentes, este algoritmo realiza a busca a partir de uma tabela *Hash* temporária mantida em memória. A tabela mapeia registros em *buckets* pelo valor de uma chave de busca, de modo a permitir que essa chave de busca seja usada para recuperar os registros mapeados em tempo constante.

A Figura 2.7 apresenta dois exemplos de uso. Em a), o objetivo é recuperar filmes cujo título seja igual ao nome de algum personagem, mesmo que esse personagem não tenha relação com o filme. Em b), o objetivo é recuperar títulos de filme e nomes de seus personagens. Nos dois casos, no lado interno usa uma tabela *Hash* construída sobre a coluna da junção. Como em a) o nome do personagem não é único, o operador de *Hash* se encarrega de remover as eventuais duplicatas. Ainda, o hash é construído sobre o resultado de uma projeção que mantém apenas as colunas de interesse, de modo a reduzir o custo em espaço para a sua construção.

Observe que o operador de junção usado é o *Nested Loop Join*, pois a lógica de processamento é idêntica: para cada registro do lado externo, deve-se encontrar correspondências no lado interno. O que muda é como essa correspondência é encontrada. Vamos analisar as três opções. No *Nested Loop Join* puro, todo o lado interno precisa ser

percorrido. Essa é a alternativa que é geralmente menos eficaz. Nas outras duas opções, ocorre uma busca indexada. Com *Nested Loop Join* indexado, um índice permanente é usado para encontrar correspondências. Com *Hash Join*, uma tabela *Hash* temporária cumpre esse propósito.

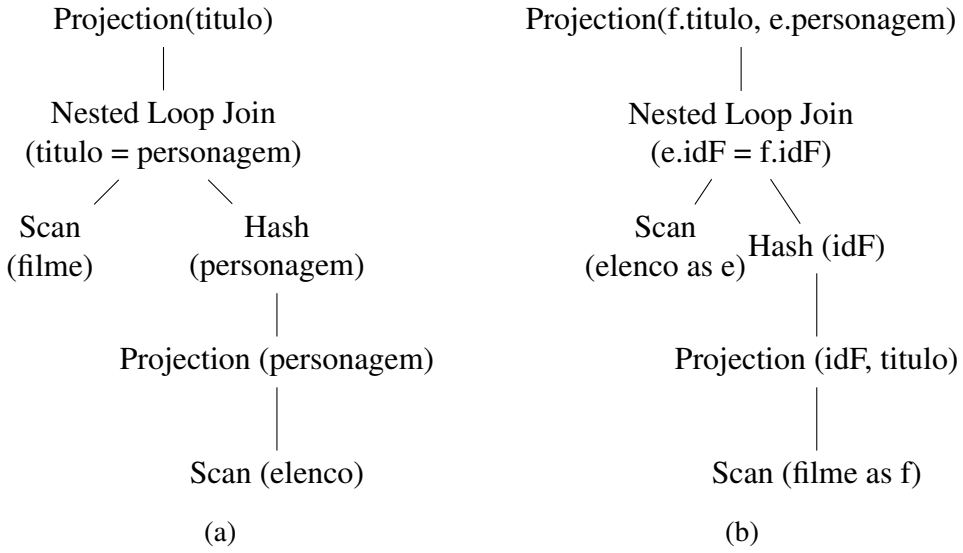


Figura 2.7: Exemplos de consultas com Hash Join

O *Hash Join* é uma alternativa eficaz quando não existem índices compatíveis. É o caso, por exemplo, da consulta a). Apesar do tempo gasto para construir a tabela, a busca indexada ocorre em tempo constante. O *Hash Join* também surge como solução mesmo quando existem índices compatíveis, mas o uso desses índices é considerado dispendioso. Como vimos, a etapa de complementação de um índice pode ter um custo relativamente alto. Dependendo da quantidade de registros a complementar, os acessos aleatórios a disco podem elevar consideravelmente o custo. O PostgreSQL, por exemplo, utiliza frequentemente o *Hash Join*, mesmo quando existem índices disponíveis. No entanto, é importante estar atento ao consumo de memória exigido para manter as tabelas *Hash*. O uso excessivo desse algoritmo pode levar a necessidade de recorrer ao *swap*, fazendo com que o desempenho caia.

Convém destacar que existem algoritmos de *Hash Join* mais complexos, como o *Grace Hash Join*, que exige que os dois lados da junção sejam transformados em tabelas *hash* [Jahangiri et al. 2022]. Essa estratégia é necessária quando não há espaço disponível para manter a tabela *Hash* em memória. Isso exige o uso de um operador de junção diferenciado, pois não se pode se valer da lógica de percorrimento de registros do *Nested Loop Join*. Devido a falta de espaço, essas variações não serão abordadas neste capítulo.

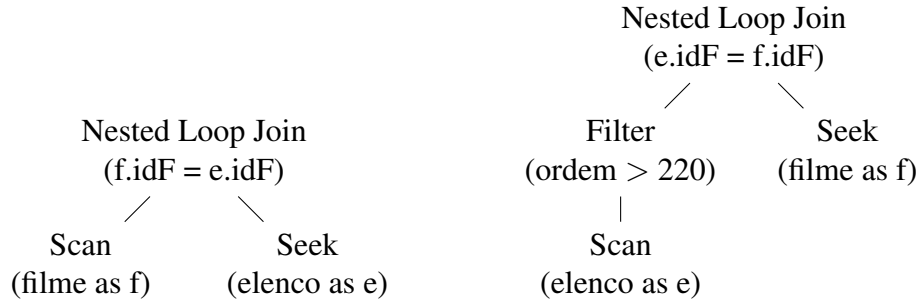
2.5.4. Custo das Junções

O custo das junções é um dos principais fatores que impactam o processamento de consultas. Por isso, os otimizadores de consulta dão especial atenção a esse aspecto. As decisões tomadas pelos otimizadores são relacionadas ao posicionamento das tabelas e aos algoritmos a serem utilizados.

O primeiro fator a analisar é a possibilidade de recorrer a um índice. Por exemplo, considere o seguinte critério de junção ('f.titulo = e.personagem'). Caso alguma dessas colunas possua algum índice, a tabela respectiva ficará do lado interno. Isso agiliza a busca por correspondências, livrando o algoritmo de fazer uma varredura completa. Caso não haja índices disponíveis, a melhor opção possivelmente é a criação de um índice temporário, por exemplo, por meio da estratégia de *Hash Join*, como indicado na Figura 2.7 a).

Nas situações em que ambos os lados estão indexados, a decisão depende do algoritmo de junção a ser usado. Para o caso do *Nested Loop Join*, os otimizadores tendem a deixar do lado externo a tabela que possui menos registros, pois isso reduz a quantidade de Seeks a serem realizados.

A Figura 2.8 apresenta dois exemplos de junção envolvendo uma chave primária (f.idF) e uma chave estrangeira(e.idF). O plano a) não emprega nenhum filtro. Na ausência de filtros, a menor tabela é aquela que contém a chave primária, justamente pela relação de multiplicidade que existe entre as tabelas (ex. um filme, n artistas). Dessa forma, a tabela 'filme' é usada no lado externo da junção. No exemplo b) um filtro é aplicado sobre a tabela 'elenco.' Como se trata de um filtro bastante seletivo, torna-se interessante posicioná-lo no lado externo da junção, pois isso implica em um número reduzido de Seeks.



(a) Junção com Filtro (b) Junção sem filtro

Figura 2.8: Impacto da presença de filtros em junções

Caso o *Hash Join* seja usado, nem sempre o otimizado mantém a menor tabela do lado externo. O motivo tem relação com a montagem da tabela *Hash*. Ao deixar a menor tabela do lado interno, em vez do externo, a tabela *Hash* ocupa menos espaço, o que pode ser uma decisão interessante caso a quantidade de memória disponível seja limitada. Para o plano a), ainda existe outro motivo. Usando filme do lado interno, além da tabela *Hash* ocupar menos espaço, a chave da função *Hash* seria um valor único por registro, uma vez que 'idF' não se repete nessa tabela. Isso aumenta a uniformidade de distribuição dos registros pelos *buckets*, reduzindo o custo no acesso.

Para o plano b), possivelmente o *Hash Join* não seja uma boa opção, independente da posição dos operadores. Mantendo a menor tabela no lado interno(elenco), a tabela *Hash* seria bem reduzida. No entanto, o número de *Seeks* seria grande, um para cada filme. Mantendo a menor tabela do lado externo, o número de *Seeks* seria pequeno. No

entanto, teríamos que criar uma tabela *Hash* para todos os filmes, independente de quantos tiverem mais do que 220 personagens.

Para saber que a estratégia baseada em *Hash* é desvantajosa, ou que é preferível deixar a tabela filtrada no lado externo na junção, o SGBD precisa saber o quão seletivo o filtro é. Sem estatísticas apuradas a respeito da distribuição dos valores para a coluna 'ordem', o otimizador pode tomar a decisão equivocada. É nesses momentos que o administrador do banco de dados pode intervir, encorajando o otimizador a seguir um outro caminho. Cada SGBD possui seus próprios recursos para isso. No problema mencionado, poderíamos usar a cláusula 'STRAIGHT_JOIN', para encorajar o MySQL a deixar elenco do lado externo. Já no caso no PostgreSQL, podemos usar 'set enablehashjoin = false' para que o algoritmo de *Hash Join* não seja usado.

Em planos de execução que envolvem mais de duas tabelas, é comum colocar nós fonte(tabelas ou índices) no lado interno de uma junção, em vez do resultado de uma junção intermediária. Isso ocorre para reduzir o esforço da junção, já que o lado interno é processado uma vez para cada registro do lado externo. Se uma junção intermediária fosse colocada no lado interno, essa junção teria que ser repetida para cada registro do lado externo, o que aumentaria significativamente o custo de processamento. Além disso, posicionar tabelas/índices no lado interno favorece que sejam realizadas buscas eficientes por meio de *Seeks*.

Com essa disposição das tabelas, os planos de execução assumem um aspecto inclinado para a esquerda. O fato de os planos serem inclinados para a esquerda também simplifica o posicionamento das tabelas na árvore de junções. Como o otimizador não precisa se preocupar com outros formatos de árvore, ele pode se concentrar em decidir apenas qual será a próxima tabela a ser incluída. Isso reduz consideravelmente o espaço de busca por soluções.

Naturalmente, existem exceções. As árvores podem assumir um outro formato. Isso normalmente está associado a situações em que uma parte da árvore é materializada, por exemplo, por meio de um operador de *Hash*. Nesse caso, seja de um lado ou de outro, essa parte do plano é executada uma única vez. Entre os diversos cenários de utilidade, a materialização pode ser usada em situações envolvendo subconsultas, onde a consulta interna pode ser tratada como uma sub-árvore independente. A próxima seção descreve esse cenário mais a fundo.

2.5.5. Subconsultas

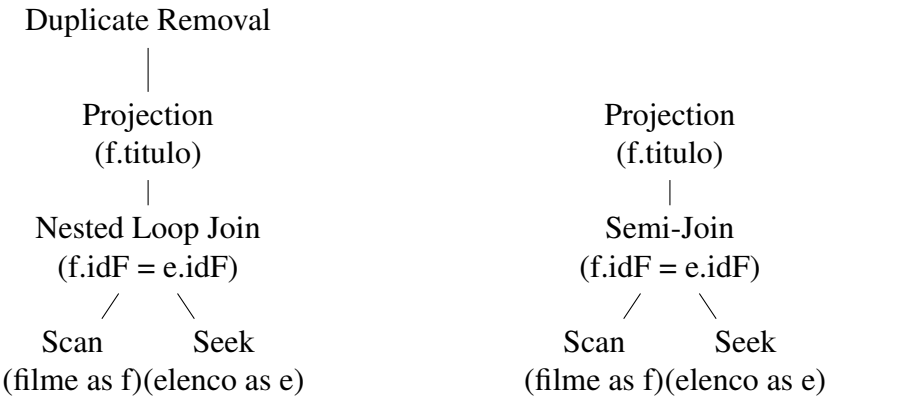
Um recurso muito utilizado em SQL para realizar consultas complexas e dinâmicas são as subconsultas. Elas são úteis para simplificar uma etapa de processamento que é exigida pela consulta principal. Uma das principais características de subconsultas é que seus resultados não podem ser usados pelas consultas de níveis superiores, exceto para fins de comparação. Ou seja, não é possível retornar colunas que sejam geradas por uma subconsulta.

Em um plano de execução, essa condição é representada por meio do operador de semi-junção, em que o resultado da junção inclui apenas os registros do lado externo que possuem correspondências com algum registro do lado interno. Diferentemente da junção

regular, a semi-junção não retorna nada do lado interno, apenas verifica a existência de correspondências.

A Figura 2.9 apresenta um exemplo de uso de uma semi-junção. O objetivo é encontrar títulos de filmes que possuam elenco. O plano da esquerda cumpre esse objetivo por meio de uma junção regular entre filme e elenco, finalizado por uma etapa de remoção de duplicatas. Já o plano da direita emprega o operador *Semi-Join*, que dispensa a remoção de duplicatas. O plano a) precisa recuperar todas as correspondências de um filme, para logo em seguida remover todas, com exceção de uma. Quanto maior for a capilaridade de um filme, mais trabalhosa será a recuperação dos membros do elenco e sua posterior remoção. Já o plano b) em nenhum momento recupera correspondências. Ele apenas verifica se elas existem. Isso naturalmente faz com que essa estratégia seja mais eficiente.

Aqui está um cuidado importante que muitas vezes não é observado: sempre que se percebe que a consulta realiza uma semi-junção, é recomendável usar o operador especificamente projetado para essa finalidade. Outra otimização possível envolve ajustar a cláusula 'SELECT', retirando todas as colunas de um dos lados da junção. Isso permite que a semi-junção seja usada. Naturalmente, deve-se investigar se essa alteração não elimina colunas cujo acesso seja imprescindível.



(a) Usando uma junção regular (b) Usando o operador de semi-junção

Figura 2.9: Alternativas para resolver uma semi-junção

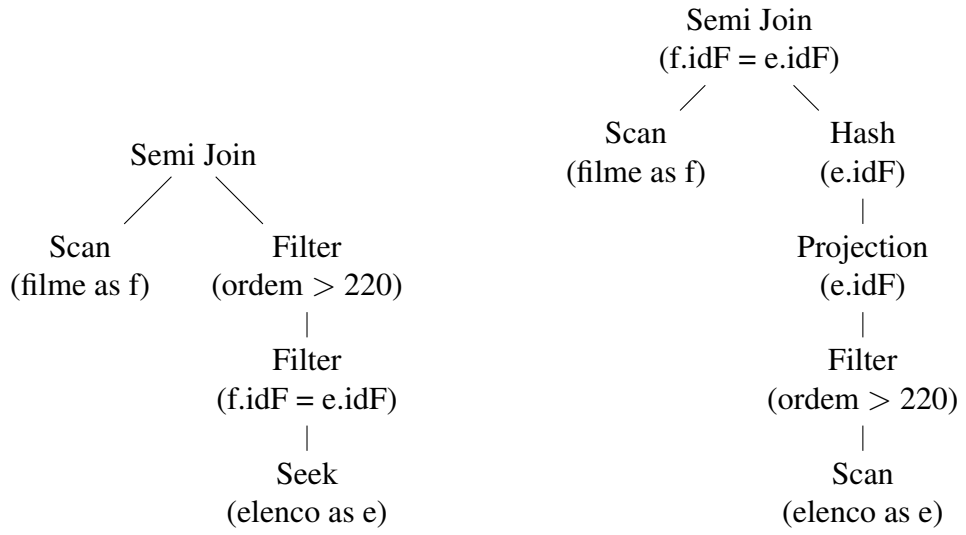
Além dessa característica principal, que estabelece que o retorno de uma consulta não pode conter dados vindos de uma subconsulta, as subconsultas também podem ser divididas em dois tipos principais:

- **Subconsultas correlacionadas:** Dependem de cada registro gerado pela consulta externa e são executadas repetidamente, uma vez para cada registro externo. A dependência é indicado pela presença de colunas da consulta externa como critério de filtragem na subconsulta. São tipicamente representadas em SQL pela cláusula 'EXISTS'.

- **Subconsultas não correlacionadas:** Não dependem de cada registro da consulta externa. Isso permite que elas sejam executadas apenas uma vez, gerando uma

tabela temporária, que pode então ser usada pela consulta externa. São tipicamente representadas em SQL pela cláusula 'IN'.

A Figura 2.10 ilustra um exemplo onde o objetivo é descobrir filmes que possuam mais do que 220 membros do elenco. Os dois planos usam semi-junção. Porém, em a) a subconsulta é correlacionada. O lado interno é executado para cada filme, o que envolve uma busca indexada (f.idF = e.idF) e um filtro(ordem $>$ 220). Em b) a subconsulta é não correlacionada. O lado interno é executado uma única vez, gerando uma tabela *Hash* que armazena os ids de filmes que possuem mais do que 220 membros. Para cada filme, o critério de correspondência é verificado contra a tabela *Hash*. Observe que, como a tabela *Hash* é usada apenas para verificação, ela não precisa contemplar todas as colunas de elenco. Dessa forma, a projeção anterior ao operador de *Hash* preserva apenas a coluna que será a chave da tabela *Hash*(e.idF). Isso faz com que a tabela ocupe menos espaço.



(a) Plano baseado na cláusula EXISTS

(b) Plano baseado na cláusula IN

Figura 2.10: Representação das cláusulas SQL IN e e EXISTS em planos de consulta

A primeira alternativa é indicada para casos em que poucos registros devem ser processados do lado externo, enquanto a segunda alternativa é mais indicada quando a subconsulta retorna poucos registros. Muitos bancos de dados, como o MySQL e o PostgreSQL, conseguem otimizar uma consulta independente da forma como ela foi escrita, seja com 'IN' ou com 'EXISTS'. No entanto, sempre podem existir situações mais complexas (ex. aninhamentos de subconsultas, condições de filtragem diferenciados, presença de campos nulos) onde as possibilidades de otimização automatizada são reduzidas. Para esses casos, é importante saber as implicações de cada estratégia de acesso, pois isso pode ser útil na concepção de um caminho alternativo mais eficiente.

Por fim, subconsultas podem ser conectadas com o seu nível superior por meio de outros operadores, como 'SOME' e 'ALL', por exemplo. Dentre todas as vertentes, uma das variações mais comuns é a anti-junção. Seu propósito é inverso ao da semi-junção, ou seja, ela inclui apenas os registros do lado externo que não possuem correspondências com o lado interno. Em SQL, elas equivalem ao 'NOT IN'/'NOT EXISTS'.

Apesar de a semi-junção e a anti-junção serem operadores com semântica diferente, mantém-se a premissa de que os resultados de uma sub-consulta não são exportados para o nível superior. Esse requisito faz com que a verificação possa ser realizada por um caminho mais eficiente do que por meio de junções regulares. A próxima seção discute um caso em que uma anti-junção (sub-consulta conectada por 'NOT IN'/'NOT EXISTS') é preferível à alternativas baseadas em junções regulares.

2.5.6. Junções Externas

A junção externa, ou *outer join*, é uma operação de junção entre duas tabelas que retorna todos os registros de uma tabela e os registros correspondentes da outra tabela. Se não houver correspondência, a junção ainda retorna todos os registros de uma das tabelas (a principal), preenchendo com valores nulos os campos da outra tabela (a secundária) onde não houver correspondência. Essa operação é útil principalmente quando precisamos garantir que todos os registros de uma tabela estejam presentes no resultado, independentemente de haver correspondências do outro lado.

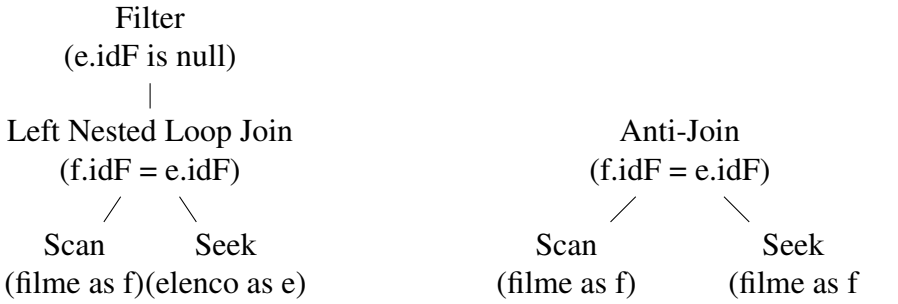
Uma das variações mais usadas é a junção externa esquerda (*left outer join*), que considera a tabela da esquerda da junção como sendo a principal. Por exemplo, uma junção externa 'filme left join elenco' retorna todos os registros da tabela à esquerda da junção (filme) e os registros correspondentes da tabela à direita (elenco). Filmes sem correspondência têm os campos da tabela elenco preenchidos com valores nulos.

Devido a sua utilidade prática, esse recurso é muito utilizado. Na verdade, seu uso é tão disseminado que por vezes ele é usado de forma equivocada. Por isso, é importante assegurar que realmente a sua semântica é a desejada. No caso do exemplo acima, deve-se analisar se os filmes sem elenco são relevantes para o resultado, ou se ao menos existem filmes nessas condições. Caso contrário, é recomendável escrever a consulta usando junções convencionais.

Para entender, vamos considerar o plano que seria gerado para o exemplo citado. Neste plano, o algoritmo de junção precisa saber quais registros da tabela principal não possuem correspondências, para complementá-los com nulos. Esse controle por si só faz com que a junção externa seja mais custosa do que uma junção regular. Além disso, alguns SGBDs, como o MySQL, exigem que a tabela principal de uma junção externa seja colocado no lado externo, pois isso simplifica a obtenção de todos os registros de interesse. Ou seja, a presença de junções externas em uma consulta já implica em uma ordenação prévia das junções, retirando parte da liberdade do otimizador em escolher ordens diferentes.

Outra questão pertinente está relacionada ao uso da junção externa para realizar anti-junções. Um exemplo é apresentado na Figura 2.11. O plano a) emprega o algoritmo de junção externa para obter todos os filmes. Em seguida, aplica um filtro para que apenas os filmes sem correspondência sejam mantidos no resultado. Por sua vez, o plano b) usa o operador de anti-junção, que verifica se um filme possui correspondências. Caso contrário, o filme é mantido no resultado.

Comparando os planos, percebe-se que a estratégia que utiliza junção externa realiza um trabalho desnecessário. Ela precisa encontrar todas as correspondências de filmes,



(a) Usando uma junção externa (b) Usando o operador de anti-junção

Figura 2.11: Alternativas para resolver uma anti-junção

para então eliminá-las. Por outro lado, a anti-junção sequer recupera as correspondências, sendo, por isso, mais eficiente. Como já mencionado, mesmo sendo possível representar anti-junções(ou semi-junções) sem recorrer aos operadores especificamente projetados para isso, é recomendável que esses operadores sejam usados. Como eles foram desenvolvidos com essa finalidade, é provável que seu desempenho seja superior.

2.6. Dados Ordenados

Um dos operadores presentes em planos de execução é a ordenação. Ele é essencial quando as consultas exigem que os dados sejam ordenados por algum critério específico ou quando algum dos operadores internos do plano de execução precisa que os dados sejam lidos em uma ordem predeterminada.

O custo depende da quantidade de registros a ordenar. Caso a quantidade seja muito alta, a ponto de não ser possível ordenar tudo em memória, o SGBD pode recorrer à ordenação externa, onde dados parcialmente ordenados são salvos temporariamente em memória secundária. Devido à necessidade de armazenar dados no disco, essa solução tem um elevado custo de E/S, tanto para ler quanto para gravar os dados.

Se a coluna a ser ordenada possui um índice, é possível acessar os dados ordenados por meio desse índice, em vez de recorrer à ordenação externa. No entanto, o uso de índices nem sempre compensa. Para entender isso, considere o exemplo em que se deseja obter o título e o ano de filmes, ordenados por ano. A Figura 2.12 apresenta dois planos que encontram a resposta desejada. No plano a), os registros de filme devem ser explicitamente ordenados pelo operador de ordenação (*Sorter*). Já o plano b) recorre a um índice sobre a coluna 'ano'. Como o índice já organiza os anos em ordem, isso poupa o motor de execução da tarefa de ordenação explícita. No entanto, o plano b) precisa de uma etapa de complementação para obter o título.

Os acessos ao disco necessários para realizar a ordenação no plano a) são feitos sequencialmente (por meio de um *Scan*). Já a complementação de índice no plano b) é caracterizada pelos acessos aleatórios (*Seeks*). Assim, é necessário analisar o quão custosas são as operações de E/S da ordenação externa em comparação com os acessos aleatórios a partir do índice. Um dos fatores que afeta o custo é o tamanho do registro a ordenar. No exemplo, os registros a ordenar são compostos por apenas duas colunas

(a) Usando operador de ordenação (b) Usando índice sobre a coluna de ordenação

Figura 2.12: Alternativas para resolver uma ordenação

(título e ano). Registros pequenos tornam a ordenação externa mais eficiente, reduzindo a quantidade de vezes que os dados precisam ser lidos/gravados do disco. Por outro lado, se muitas colunas precisarem ser recuperadas (ex. todas as colunas de filme), o esforço do algoritmo de ordenação externa seria maior, podendo tornar o uso do índice mais atrativo.

Para tornar os índices ainda mais atrativos, os SGBDs usam técnicas que visam organizar os acessos ao disco, visando reduzir a aleatoriedade. Outra técnica válida envolve adicionar colunas no índice, transformando-o em um *covering index*. No exemplo acima, isso equivaleria a adicionar a coluna 'título' como segundo nível do índice sobre 'ano'.

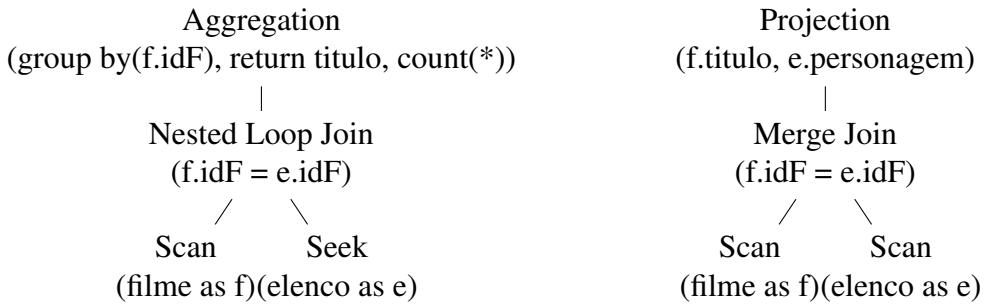
Essas soluções, apesar de reduzirem o custo da ordenação, trazem efeitos colaterais. Por exemplo, organizar os acessos ao disco demanda memória. Já o incremento de um índice aumenta o custo de manutenção desse índice. Ou seja, as soluções apresentadas acabam deixando rastros que podem afetar o desempenho de outras consultas que sejam executadas de forma concomitante.

Uma forma mais adequada de lidar com esse problema é analisar a possibilidade de adicionar filtros à consulta. A presença de filtros seletivos pode diminuir consideravelmente a quantidade de filmes a processar, tornando o processo de ordenação bem menos custoso, a ponto de sequer exigir ordenação externa.

Além dos casos em que os dados devem ser retornados em ordem, operadores internos podem se beneficiar ou até exigir que os dados estejam ordenados. Alguns exemplos são agregação, remoção de duplicatas, operações com conjuntos e o algoritmo de junção *Merge Join*. Por exemplo, para a agregação, o objetivo é formar grupos de registros que partilhem o mesmo valor para algum conjunto de colunas e realizar alguma função de agregação sobre cada grupo formado. Caso os dados estejam ordenados por esse conjunto de colunas, os registros que compõe cada grupo são retornados de forma consecutiva. Isso simplifica a etapa de cálculo da função de agregação. De forma similar, a remoção de duplicatas é simplificada se os registros estiverem ordenados, uma vez que as duplicatas serão retornadas de forma consecutiva. Já o *Merge Join* é um algoritmo de junção que pode ser usado quando os dois lados estão ordenados pelas colunas de junção. Ao contrário do *Nested Loop Join*, que fixa um registro do lado externo para buscar correspondências do lado interno, o *Merge Join*, devido a ordenação, encontra as

correspondências por meio de varreduras unidirecionais nos dois lados, sem que registros já acessados precisem ser verificados novamente. A mesma lógica de percorrimento se aplica para as operações que trabalham com conjuntos (união, diferença e interseção).

A Figura 2.13 apresenta dois exemplos de operadores que se valem de dados ordenados. O plano a) usa agregação para retornar o título de cada filme juntamente com a contagem de artistas. Já o plano b) usa *Merge Join* para encontrar o elenco de cada filme. Considere que, em ambos os casos, as tabelas são armazenadas em índices clusterizados, ou seja, o *Scan* sobre essas tabelas já retorna os dados na ordem de chave primária. Essa propriedade garante que as operações de agregação e de *Merge Join* funcionem sem depender de uma ordenação explícita. Convém ressaltar que os planos exibidos são idealizações, que exploram ao máximo as características dos dados acessados para evitar ordenações explícita. No entanto, é possível que os SGBDs não disponham de mecanismos que permitem encontrar esses planos otimizados.



(a) Usando agregação

(b) Usando Merge Join

Figura 2.13: Planos contendo operadores que se beneficiam de dados ordenados

Assim como discutido no exemplo da Figura 2.12, mesmo que índices possam ser usados, deve-se considerar se vale a pena usá-los. Caso haja algum filtro que reduza a quantidade de registros, possivelmente não compense recorrer a algum índice. A ordenação explícita poderia ser feita em memória, usando apenas os registros relevantes. Já o acesso ao índice recuperaria todos os registros, quando uma pequena parte deles seria suficiente.

Alguns operadores, como agregação e remoção de duplicatas, também podem trabalhar com técnicas baseadas em *Hash* em vez de dados ordenados. Ou seja, em vez de formar grupos por meio de registros adjacentes, a técnica de *hash* mantém os registros de um grupo dentro do mesmo *bucket*. Para casos em que é possível usar as duas estratégias, cabe ao SGBD decidir qual delas é mais vantajosa.

Para grandes volumes de registros, a solução baseada em *Hash* tende a ser mais eficiente do que a ordenação externa. No entanto, para volumes menores de dados, o custo constante associado ao gerenciamento da tabela *hash* pode tornar essa estratégia menos vantajosa do que a ordenação em memória. Além disso, o tamanho do registro também influencia: a ordenação de registros grandes é mais custosa devido à movimentação dos registros no vetor de ordenação e ao uso ineficiente do cache da CPU. Em contraste, em tabelas *hash*, a movimentação desses registros é geralmente menos custosa, especialmente

se os dados forem distribuídos uniformemente nos *buckets*.

A Figura 2.14 ilustra um exemplo. O objetivo é retornar todos os dados de filmes e seus participantes, sejam eles membros da equipe de atuação ou de produção. Como os membros dessas duas equipes são registrados em tabelas separadas, o operador de união pode concatená-los, garantindo a remoção de duplicatas, uma vez que uma mesma pessoa pode atuar em ambas as equipes. A concatenação pode ser realizada por meio de *hash* ou ordenação. No exemplo, uma solução baseada em *hash* é mostrada, pois os registros a serem retornados contêm muitas colunas e é necessário retornar todos os filmes, o que torna a construção de uma tabela *hash* mais viável.

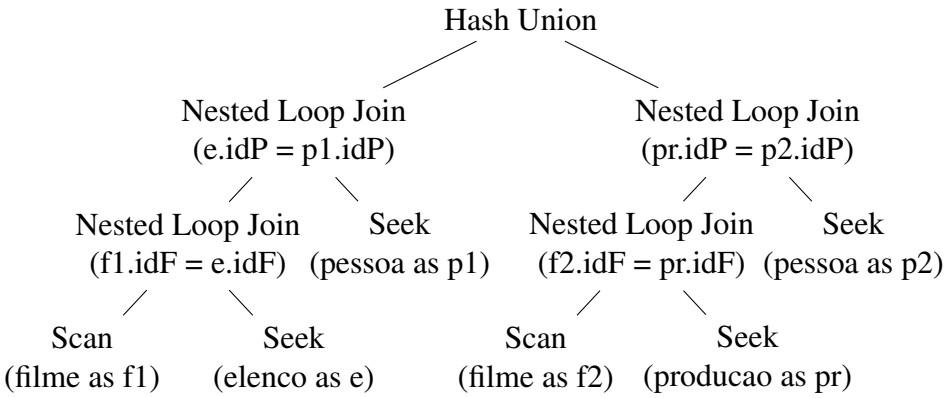


Figura 2.14: Uso de Hash em vez de ordenação para resolver uma operação de união

Existem diversas formas de otimizar essa consulta. Uma delas é reestruturar a união para ocorrer diretamente entre as tabelas de elenco e produção. Como essas tabelas são organizadas pelo id do filme, a união pode utilizar o método de ordenação, já que os dados estão em uma ordem consistente. O resultado da união seria então usado para realizar a junção com as tabelas de pessoa e filme. No entanto, para consultas com um certo grau de complexidade(como essa), o otimizador pode ter dificuldade em encontrar o caminho mais eficiente, e cabe ao desenvolvedor proceder com a reestruturação da consulta em SQL. Mesmo após o ajuste, não há garantias de que o otimizador use as melhores estratégias devido a limitações na identificação das propriedades dos registros retornados pelas partes mais internas do plano de execução.

De modo geral, a ordenação, seja por necessidade explícita da consulta ou por uma estratégia interna de agrupamento, implica em um custo significativo. Se for difícil obter um plano de execução satisfatório, é importante considerar a redução do tamanho dos registros (removendo colunas) ou a quantidade de registros a serem retornados (aplicando filtros seletivos). Outra técnica útil para reduzir a quantidade de registros é a paginação, que será apresentada na próxima seção.

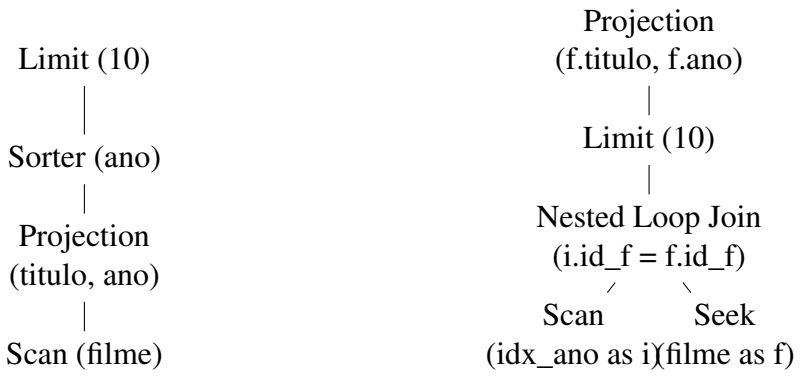
2.7. Paginação

A paginação em SQL é um recurso essencial para gerenciar a exibição de grandes volumes de dados de maneira eficiente e acessível. Em vez de retornar todos os registros de uma consulta de uma só vez, a paginação permite dividir os resultados em páginas menores, facilitando a navegação e o processamento dos dados.

A paginação é frequentemente implementada usando cláusulas SQL como 'LIMIT', para definir a quantidade de registros a retornar, e 'OFFSET', para definir a partir de qual registro o retorno deve ocorrer. A paginação normalmente está associada ao uso da ordenação, para que o retorno seja sempre consistente e indique os registros de maior relevância.

Da perspectiva do processamento de uma consulta, esse recurso é bastante valioso. Consultas que retornam grandes conjuntos de resultados podem ser extremamente lentas e consumir muitos recursos do servidor. A paginação melhora o desempenho ao limitar a quantidade de dados processados. Além disso, transferir grandes quantidades de dados de uma vez pode sobrecarregar a memória do sistema tanto no servidor quanto no cliente. A paginação ajuda a gerenciar melhor o uso de memória, carregando apenas os dados necessários para cada página. Por fim, ao limitar a quantidade de dados transferidos em cada consulta, a paginação reduz a latência de resposta, tornando as aplicações mais rápidas e responsivas.

A Figura 2.15 apresenta dois exemplos em que a paginação é aplicada, utilizando o operador *Limit* para retornar os 10 filmes mais antigos. No plano a), foi necessário ordenar explicitamente os registros pela coluna 'ano'. Nesse caso, a ordenação não precisa ser total; devido à presença do operador *Limit*, o operador de ordenação sabe que apenas 10 registros são relevantes. Dessa forma, a ordenação pode ser otimizada descartando registros que não se classifiquem entre os 10 primeiros. No plano b), foi utilizado um índice sobre a coluna 'ano'. Nesse caso, basta acessar os 10 primeiros registros presentes no índice.



(a) Usando ordenação explícita (b) Usando índice sobre a coluna de ordenação

Figura 2.15: Alternativas para limitar a quantidade de registros retornados

A diferença fundamental entre os dois planos de execução reside na maneira como os dados são ordenados e acessados. No plano a), a ordenação é feita diretamente sobre os registros da tabela. A projeção das duas colunas a retornar e o uso do operador *Limit* reduzem o custo da ordenação, mas ainda assim o custo pode ser alto se a tabela for grande. O plano b) permite acessar diretamente os registros já ordenados. Como apenas as 10 primeiras entradas precisam ser complementadas, este plano tende a ser mais eficiente.

Como vimos, o operador *Limit* pode agilizar consultas ao reduzir o número de registros retornados, evitando a necessidade de processar a tabela inteira. No entanto,

pode enfrentar dificuldades quando usado com operadores materializados, que exigem que todos os registros sejam acessados antes de prosseguir. Por exemplo, por definição, o operador *Hash* gera uma tabela *hash* contendo todos os registros acessíveis. Ele é agnóstico com relação às exigências feitas por operadores que o usam. Por esse motivo, quando o limite é informado, e ele for restritivo, geralmente o algoritmo de *Hash Join* é preterido por outra opção que não exija a leitura de toda a tabela.

Existem situações em que a ordenação é necessária, mas o limite não é estabelecido sobre o critério de ordenação. Isso exige que toda a tabela seja acessada. Um exemplo é descrito na Figura 2.16. O objetivo é retornar os 10 artistas que tiveram o maior valor referente a ordem de aparição em filmes. Uma agregação é necessária para encontrar a maior ordem de cada pessoa, e essa agregação foi facilitada por um operador de ordenação. Nesse caso, a ordenação precisa ser total, uma vez que o critério de ordenação do agrupamento (idP) é diferente do critério de ordenação do limite (ordem máxima). Já a ordenação reversa pela ordem é influenciada pelo limite definido, podendo se restringir a encontrar os 10 maiores valores. Por fim, a junção com a tabela pessoa é usada para recuperar o nome da pessoa classificada. É interessante observar que a agregação está desvinculada da junção com a tabela 'pessoa'. Muito embora ela não seja afetada pelo operador de limite, o seu custo é reduzido por não ser preciso estabelecer o relacionamento com 'pessoa'. Como a junção é realizada apenas após a ordenação reversa pela ordem, isso assegura que apenas os 10 ids de pessoas selecionadas precisem ser complementados com o nome.

A construção desse plano exige que o cálculo da agregação seja desvinculado da junção já na consulta SQL. Do contrário, o otimizador possivelmente realize a junção antes da agregação. Ou seja, em algumas situações, cabe ao desenvolvedor identificar planos de execução mais eficientes. No entanto, nem sempre é óbvio como escrever consultas SQL para que esses planos sejam escolhidos, ainda mais para casos mais complexos. Por isso, é importante compreender a relação entre uma consulta SQL e seu plano de execução, permitindo que o usuário busque alternativas mais eficientes. Muitas vezes, é difícil fazer essa conexão apenas visualizando os planos gerados, sem poder manipulá-los diretamente. A próxima seção apresenta uma ferramenta que permite explorar planos de execução, ajudando o usuário a entender melhor as conexões entre SQL e planos de execução, o que pode ajudar na escrita de consultas melhores.

2.8. Uma Linguagem Alternativa para a Criação de Planos de Execução

Quando Edgar F. Codd introduziu o conceito de banco de dados relacional em 1970, ele inicialmente propôs linguagens procedurais baseadas em conceitos formais denominados Cálculo Relacional e Álgebra Relacional. No entanto, a linguagem SQL, desenvolvida posteriormente por pesquisadores da IBM, adotou uma abordagem declarativa, onde os usuários especificam "o que"querem obter sem detalhar "como"o sistema deve executar a operação [Chamberlin 2012]. O modelo relacional se tornou o paradigma dominante, e a SQL se estabeleceu como a linguagem padrão para a interação com bancos de dados relacionais.

Devido à natureza declarativa da SQL, os usuários não precisam saber detalhes sobre como os dados são armazenados ou os algoritmos usados para acessá-los. Cabe ao

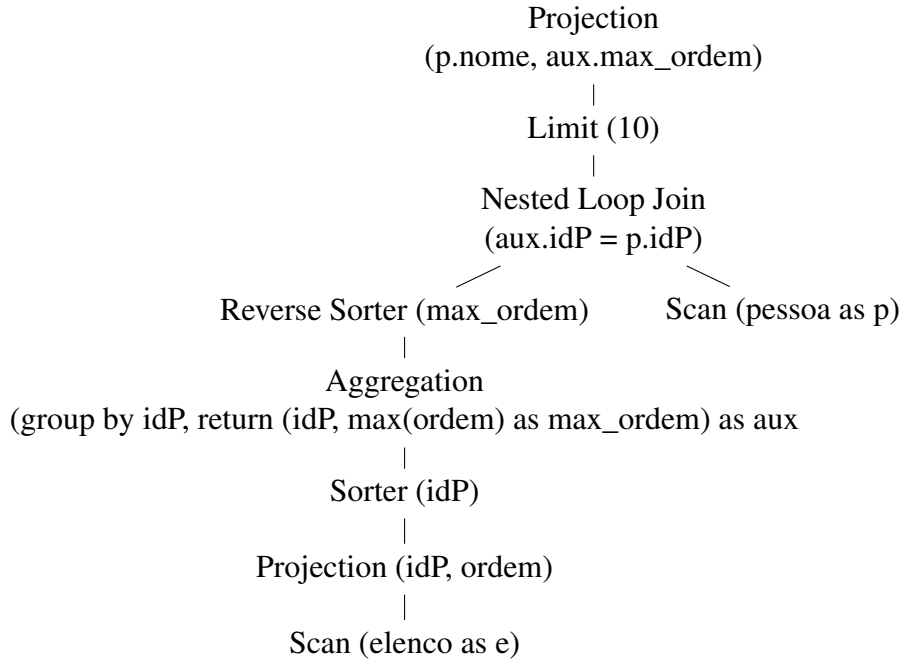


Figura 2.16: Aplicando limite sobre o resultado de uma agregação

SGBD a responsabilidade de otimizar a consulta, escolhendo o plano de execução mais eficiente. Embora a linguagem ofereça simplicidade e abstração, permitindo ao usuário focar no resultado desejado sem se preocupar com os detalhes de implementação, essa característica traz algumas limitações na forma como o usuário interage com o banco de dados.

Primeiramente, os planos de execução têm um caráter apenas informativo. Isso significa que, embora seja possível visualizá-los e analisá-los, não se pode modificá-los de forma interativa. Para modificar um plano de execução, é necessário alterar a consulta SQL ou ajustar a estrutura do banco de dados (ex. adicionando índices), e então verificar se essas alterações resultam em uma mudança no plano de execução. Como o usuário não tem controle direto sobre o plano de execução que o SGBD escolhe, isso pode ser problemático em situações onde o conhecimento especializado do usuário poderia levar a uma execução mais eficiente.

Além disso, os planos de execução não permitem a execução de operações internas de forma isolada para verificar o que essas partes retornam. Essa limitação torna o processo de otimização mais difícil e menos intuitivo, pois os desenvolvedores não conseguem testar partes específicas do plano diretamente. Eles precisam modificar a consulta ou os índices e reexecutar a consulta inteira para ver o efeito dessas mudanças.

Essas limitações dificultam não apenas a otimização das consultas, mas também a compreensão detalhada do que acontece durante a execução. Sem a capacidade de modificar diretamente os planos ou executar partes específicas, os desenvolvedores precisam recorrer a um processo de tentativa e erro, o que pode ser demorado e ineficiente.

Ao longo dos anos, as limitações da abordagem declarativa foram abordadas com

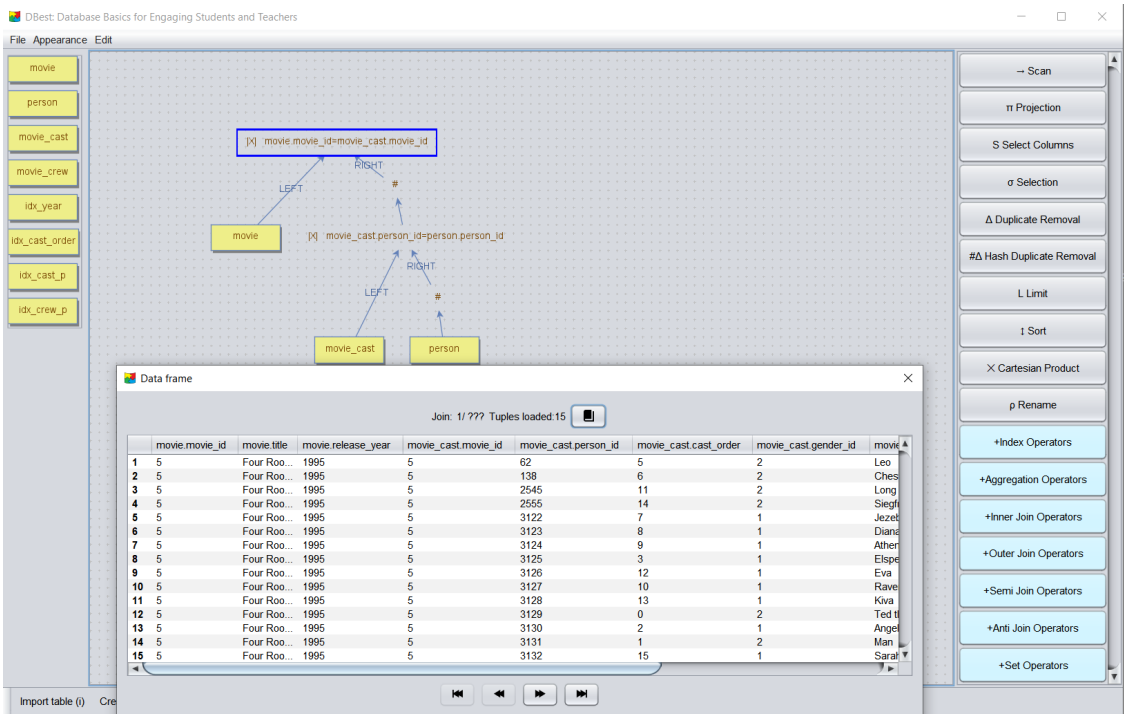


Figura 2.17: Interface gráfica para a escrita de consultas

várias extensões e melhorias nos SGBDs, incluindo dicas de otimização, ferramentas de análise de planos de execução e a integração de linguagens procedurais para operações mais complexas (como PL/SQL no Oracle, T-SQL no SQL Server, etc.) [Zhang 2017]. Isso proporcionou um equilíbrio entre a simplicidade da SQL declarativa e o controle detalhado quando necessário.

Ironicamente, a proposta original de Codd foi suplantada por uma linguagem mais simplificada. Contudo, as limitações encontradas levaram a linguagem a se tornar gradativamente mais sofisticada, mesclando aspectos procedurais e declarativos, permitindo a escrita de consultas mais complexas do que as possíveis com as linguagens propostas por Codd.

Nesse contexto, apresentamos uma proposta de linguagem de representação de consultas alternativa. Essa linguagem é imperativa, ou seja, o usuário indica de que forma o resultado deve ser gerado. Isso envolve determinar quais operadores devem ser usados e como eles devem ser estruturados.

Uma das formas de usar a linguagem é por meio de uma interface gráfica, conforme apresentado na Figura 2.17. Na tela, pode-se ver parte da árvore de execução, construída por meio de recursos de *drag and drop*. Em primeiro plano, aparece uma janela de navegação pelos resultados obtidos. A tela também apresenta as tabelas importadas que podem ser usadas como fontes de dados (à esquerda) e alguns dos operadores suportados (à direita).

A montagem da árvore é realizada de forma visual, o que envolve a seleção dos operadores e a edição de suas configurações. Com a árvore configurada, pode-se executá-la e conferir os resultados gerados. Diferentemente dos planos de execução convencionais

```
// Definicao das tabelas
Operation movie = new IndexScan("m", movieTable);
Operation cast = new IndexScan("c", castTable);

// Condicao de filtro
Operation condition = new FilterByValue("m.ano", EQUAL, 1930);
Filter filter = new Filter(movie, condition);

// Juncao
JoinPredicate jp = new JoinPredicate();
jp.addTerm("m.movie_id", "c.movie_id");
Operation join = new NestedLoopJoin(filter, cast, jp);

// Projecao simples
String[] columns = {"f.titulo", "e.personagem"};
Operation projection = new SimpleProjection(join, columns);

// Execucao da consulta
ResultSet result = projection.execute();
```

Figura 2.18: Trecho de código usando o *framework* para a escrita de consultas

gerados por SGBDs, essa ferramenta permite processar a consulta a partir de qualquer nó. Isso possibilita ao usuário perceber o que está acontecendo em cada etapa da consulta. Essa funcionalidade, aliada à presença de múltiplos indicadores de custo, permite que o usuário compreenda como o fluxo definido impacta no tempo de execução, levando-o a refletir sobre a construção de planos de execução mais eficientes.

Outra funcionalidade permite comparar os indicadores de custo de diferentes árvores de execução, conforme os registros referentes às arvores vão sendo gerados. Esse recurso é útil para compreender porque duas árvores diferentes possuem tempos de resposta distintos, mesmo quando levam aos mesmos resultados.

Consultas também podem ser construídas diretamente em código-fonte, por meio de um *framework* desenvolvido em Java. A Figura 2.18 apresenta um trecho de código descrevendo uma consulta contendo operadores de filtro e de junção.

Os operadores de um plano se comunicam por meio de tuplas, que representam um conjunto de valores de diferentes tipos de dados, como inteiros, strings e datas. As tuplas são a unidade de transferência de dados entre os operadores, permitindo que eles processem e manipulem os dados de forma estruturada e consistente.

Uma das características essenciais do *framework* é sua extensibilidade. Todos os operadores implementam uma interface comum, estabelecendo um contrato que define como eles se comunicam entre si. Novos operadores podem ser adicionados desde que respeitem esse contrato, o que possibilita a inclusão de funcionalidades diversas e a realização de testes com algoritmos diferentes.

Atualmente, a arquitetura suporta dois tipos de fontes de dados: arquivos CSV e um formato de tabela relacional customizado. A principal distinção entre esses formatos reside na organização e acessibilidade dos dados. Enquanto os arquivos CSV são apenas

coleções de registros organizados sequencialmente, o formato de tabela relacional utiliza árvores B+ para organizar e consultar dados tabulares de maneira eficiente.

A API possibilita a integração de novas fontes de dados, desde que os dados possam ser representados na forma de tuplas. Isso engloba a inclusão de fontes heterogêneas como diferentes bancos de dados, APIs web e arquivos textuais formatados. A capacidade de definir novos conectores e adaptadores simplifica a criação de soluções que unem dados provenientes de múltiplas fontes de maneira integrada e coesa.

Opcionalmente, a consulta poderia ser expressa por meio de uma linguagem específica de domínio (DSL) que represente operadores unários e binários. Essencialmente, isso envolve a descrição de árvores binárias, onde cada nó representa um operador e pode ser complementado com parâmetros específicos.

A Figura 2.19 descreve uma consulta expressa seguindo esses preceitos. No exemplo, o objetivo é retornar filmes de 2023 e seus elencos. Além de especificar as conexões entre os operadores, a linguagem também define quais algoritmos são utilizados. A descrição aninhada dos operadores é bastante similar à programação funcional, pois ambas utilizam a composição de funções (ou operadores) para construir resultados de maneira declarativa e modular [Gabbrielli and Martini 2023].

```
1 NESTED_LOOP_JOIN(
2     FILTER(
3         ano = 2023,
4         SCAN(filme) AS f
5     )
6     SEEK(elenco) AS e,
7     ON f.idF = e.idF
8 )
```

Figura 2.19: Exemplo de estrutura textual para a descrição de consultas imperativas

Embora tenha algumas semelhanças com SQL, essa abordagem de construção de consultas é significativamente diferente. Uma alternativa menos disruptiva seria estender a linguagem SQL para manter a estrutura básica das cláusulas SQL. Por exemplo, poderiam ser adicionadas anotações com instruções imperativas (ex. informando qual algoritmo de junção a usar) ou permitir que as cláusulas existentes funcionem como funções (por exemplo, fazer com que 'ORDER BY' receba como parâmetro o resultado de uma operação interna).

O exemplo descrito na Figura 2.19 não foi efetivamente implementado. Sua menção visa provocar uma discussão referente à adoção de alternativas mais ricas em detalhes referentes à execução. É importante destacar que não pretendemos suplantar a linguagem SQL, que já é amplamente adotada, compreendida e eficaz para uma vasta gama de consultas e operações em bancos de dados relacionais. Além disso, SQL oferece uma interface declarativa que é intuitiva e prática para a maioria dos usuários. Nosso intuito é fornecer uma alternativa que permita explorar os planos de execução, testar novos algoritmos e integrar fontes de dados heterogêneas, sem substituir a robustez e a familiaridade que SQL já proporciona.

2.9. Considerações Finais

Neste capítulo, destacamos a importância dos planos de execução no contexto de bancos de dados relacionais, explorando como esses planos influenciam a performance das consultas. Observamos que frequentemente cabe ao desenvolvedor identificar e selecionar planos de execução mais eficientes, reestruturando consultas para obter um melhor desempenho.

Encerramos o capítulo introduzindo uma linguagem de consulta imperativa que complementa o SQL. Esta linguagem oferece um controle mais refinado sobre a execução das consultas, permitindo aos usuários explorar e manipular diretamente os planos de execução. Tal abordagem não apenas aumenta a familiaridade dos usuários com as conexões entre planos e desempenho, mas também possibilita a experimentação com novos operadores, teste de algoritmos e integração de fontes de dados heterogêneas, aproveitando a extensibilidade e flexibilidade oferecidas.

Por fim, enfatizamos a importância de compreender os detalhes internos da execução de consultas. Nesse contexto, uma linguagem imperativa pode servir como um ambiente experimental valioso para aprendizado. Isso permite que tanto iniciantes quanto usuários experientes explorem diferentes abordagens, testem consultas complexas e compreendam melhor o comportamento do SQL em cenários específicos, facilitando o aprendizado prático e a resolução de problemas reais. Afinal, conhecer o caminho é a maneira mais eficiente de se chegar ao destino.

Referências

[Bertino et al. 2012] Bertino, E., Ooi, B. C., Sacks-Davis, R., Tan, K.-L., Zobel, J., Shidlovsky, B., and Andronico, D. (2012). *Indexing techniques for advanced database systems*, volume 8. Springer Science & Business Media.

[Chamberlin 2012] Chamberlin, D. D. (2012). Early history of sql. *IEEE Annals of the History of Computing*, 34(4):78–82.

[Gabbrielli and Martini 2023] Gabbrielli, M. and Martini, S. (2023). Functional programming paradigm. In *Programming Languages: Principles and Paradigms*, pages 335–368. Springer.

[Graefe 1994] Graefe, G. (1994). Volcano/spl minus/an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135.

[Hammer and Schneider 2018] Hammer, J. and Schneider, M. (2018). Data structures for databases. In *Handbook of Data Structures and Applications*, pages 967–981. Chapman and Hall/CRC.

[Jahangiri et al. 2022] Jahangiri, S., Carey, M., and Freytag, C. (2022). Design trade-offs for a robust dynamic hybrid hash join. *Proceedings of the VLDB Endowment*, 15(10).

[Lan et al. 2021] Lan, H., Bao, Z., and Peng, Y. (2021). A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration. *Data Science and Engineering*, 6:86–101.

[Manolopoulos et al. 2000] Manolopoulos, Y., Theodoridis, Y., Tsotras, V. J., Manolopoulos, Y., Theodoridis, Y., and Tsotras, V. J. (2000). Fundamental access methods. *Advanced Database Indexing*, pages 37–59.

[Sumathi and Esakkirajan 2007] Sumathi, S. and Esakkirajan, S. (2007). *Fundamentals of relational database management systems*, volume 47. Springer Science & Business Media.

[Taipalus 2020] Taipalus, T. (2020). The effects of database complexity on sql query formulation. *Journal of Systems and Software*, 165:110576.

[Zhang 2017] Zhang, P. (2017). *Practical Guide for Oracle SQL, T-SQL and MySQL*. Crc press.

sbbd:03

Chapter 3

Practical Graph Pattern Mining: Systems, Applications, and Challenges

Vinícius Dias, Samuel Ferraz

Abstract

Graph Pattern Mining (GPM) refers to algorithms that extract and process subgraphs from larger graphs. This course covers the GPM fundamentals, an overview of GPM systems, and practical use case applications for students and professionals with basic graph and programming skills. It features Fractal and DuMato systems, two in-house GPM systems offering good experience and performance with such algorithms. Fractal, on Spark, provides an efficient API for algorithm design, while DuMato, on CUDA, enables efficient GPM on GPUs. The challenges of GPM algorithms and performance issues are discussed, providing a solid grasp of graph processing acceleration for efficient solutions that are valuable for practitioners and the database community.

Resumo

Mineração de Padrões em Grafos (sigla em inglês: GPM) refere-se a algoritmos que extraem e processam subgrafos de grafos maiores. Este curso aborda os fundamentos de GPM, uma visão geral dos sistemas de GPM e aplicações práticas para estudantes e profissionais com habilidades básicas em grafos e programação. Ele apresenta os sistemas Fractal e DuMato, dois sistemas GPM que oferecem uma boa experiência e desempenho com tais algoritmos. O Fractal, baseado em Spark, oferece uma API eficiente para o projeto de algoritmos, enquanto o DuMato, em CUDA, possibilita uma mineração eficiente em GPUs. Os principais desafios dos algoritmos de GPM e questões de desempenho são discutidos, fornecendo uma compreensão sólida da aceleração do processamento de grafos para soluções eficientes, valioso tanto para praticantes quanto para a comunidade de banco de dados.

3.1. Introduction

Graphs are widely used to model problems in various areas, including web applications, social media, biological networks, brain networks, conceptual graphs, among others. With

the popularization of large-scale systems and the easy access to large volumes of data, the demand for graph processing has been substantial [Fan 2022]. The scope of this course proposal is on a particular class of graph algorithms, recently referred to as Graph Pattern Mining (GPM). Figure 3.1 depicts the key aspects of GPM processing. Given an input graph, GPM algorithms visit its subgraphs of interest through a combinatorial procedure (subgraph enumeration) and, using an user-defined processing on the visited subgraphs, filter the search-space of subgraph candidates and generate the result. The relevance of GPM computation is multidisciplinary, including applications such as motif extraction from biological networks [Agrawal et al. 2018], frequent subgraph mining [Elseidy et al. 2014], subgraph searching over semantic data [Elbassuoni and Blanco 2011], social media network characterization [Ugander et al. 2013], community discovery [Benson et al. 2016], periodic community discovery [Qin et al. 2019], temporal hotspot identification [Yang et al. 2016], identification of dense subgraphs in social networks [Hooi et al. 2020], link spam detection [Leon-Suematsu et al. 2011], financial fraud detection [Hoffman and Krasle 2015], recommendation systems [Zhao et al. 2019], graph learning [Meng et al. 2018], to cite a few. Since GPM algorithms are often complex to develop (graph theory involved) and also expensive in terms of systems performance (combinatorics and high memory consumption), many general-purpose GPM systems emerged in the last decade to improve the user experience in those aspects. These systems provide implementations of subgraph enumeration and facilitate the creation of user-defined processing on subgraphs.

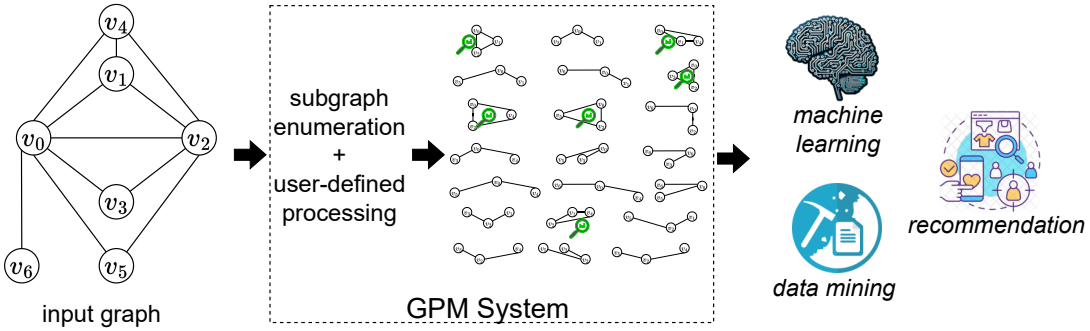


Figure 3.1. Overview of GPM processing over graph data.

In this course we are going to give an overview of GPM fundamentals, existing GPM systems, and use case scenarios that may benefit from such processing. We also discuss the main challenges that impact the design choices in state-of-the-art GPM systems. This course is intended for students and professionals interested in expanding their knowledge on algorithms and systems for mining graph data. This includes data scientists, database community students, researchers on graph processing and big-data in general, among others. Since this course is going to address graph algorithms, basic knowledge on graphs and basic programming experience is required.

3.2. Background on Graph Pattern Mining

We adopt in this course an input graph model with vertices[1] and non-directed edges. Vertices and edges also may contain a label, which captures application-specific semantics

[1]In this work, the terms "vertex" and "node" are used interchangeably

in real data. Formally, a **graph** G is represented by three sets, $V(G)$, $E(G)$ and $L(G)$ which are the sets of vertices, edges, labels of G and one map function f_L. Each edge $e = (v,u) \in E(G)$ connects a pair of vertices v and $u \in V(G)$. The edges are not directed and there are no self-loops in G. The labels of a vertex or an edge are defined according the function $f_L : V(G) \cup E(G) \rightarrow L(G)$. Figure 3.1 shows an example of input graph considered, in this case the graph is undirected and unlabeled, although colors could be used to represent vertex/edge labels.

The most fundamental routine in GPM algorithms concerns the processing of subgraphs extracted from a larger input graph G. A **subgraph** (a.k.a. general subgraphs) S is represented by a set of vertices and edges embedded in the input graph G. In this work, we are interested in *connected subgraphs*: there must be a path between any pair of nodes in $V(S)$ comprising edges in $E(S)$. If not otherwise specified, when we mention the word "subgraph" in this course we actually mean *connected subgraph*. Hence, a *subgraph* refers to a subgraph *instance* in an input graph G. Some application-specific semantics may require a more strict definition of a subgraph, namely a *connected and induced subgraph*. A subgraph S is **induced** in case it can be obtained from the input graph G from a set of vertices, and consequently its edges comprises all existing edges from G among those vertices. Note that a result of these definitions is that general not induced subgraphs are more fine-grained than induced subgraphs, since many connected not induced subgraphs may be extracted from an induced subgraph. Consider, for instance, a complete induced subgraph with 4 vertices (a.k.a. a clique) – if any of its edges is removed, we would have produced a different still connected subgraph and many of these exist given the same 4 vertices.

Subgraphs can be mapped to a naive representation of its structural and labeling information, referred simply as **pattern**. Note that patterns represent multiple subgraphs, thus they discard the identification of individual vertices and edges from the input graph. Figure 3.2 provides an example of these concepts. Colors represent vertex labels and numbers represent vertex unique identifiers.

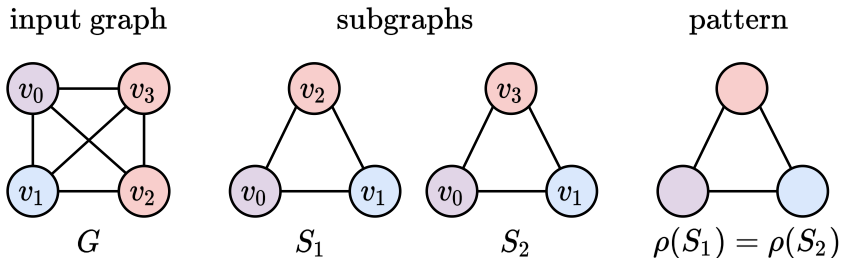


Figure 3.2. Example of input graph, subgraphs, and pattern [dos Santos Dias 2023]. Vertex colors denote labels. A single edge label is illustrated in this Figure: solid black lines.

Different patterns extracted from G may exhibit the same structural template and labeling information. We say that such subgraphs belong to the same equivalence class and that they are *isomorphic* to each other. **Graph isomorphism** is the problem of verifying whether two (sub)graphs have an identical structure (topology), being fundamental to a variety of GPM applications in graph data mining and machine learning. Although the

definition of pattern is useful to visit subgraphs with specific properties, there are different ways to represent the same pattern. For example, the triangle pattern in Figure 3.2 can be represented by a list of its edges, $[(0,1),(1,2),(2,0)]$, considering indexes starting from 0. However, an alternative representation to the same pattern can be $[(0,1),(0,2),(1,2)]$. Such ambiguity can be a problem when comparing patterns: two representation are indeed different patterns or they are two alternatives for defining the same pattern?

To handle the issues that arise with this question, we define **canonical pattern** to be a common and unique representation for each pattern. Canonical labeling algorithms exists with the purpose of mapping a subgraph to its canonical pattern, so they can be compared by equality operators (string comparison, for example) [Kuramochi and Karypis 2005, Huan et al. 2003, Yan and Han 2002, Borgelt 2007, Junttila and Kaski 2007]. It is out of the scope of this work to dive into these algorithms, but it is important to notice its importance within GPM systems, as it allows handling isomorphism issues that may arise from pattern and or subgraph extraction.

The visitation of subgraphs from a graph is also related to the concept of isomorphism. Specifically, subgraphs are built by the visitation of connected vertices and edges from G in a specific order, incrementally. Therefore, any permutation of vertices and edges represents a visitation ordered code for the *same* subgraph instance in G (Figure 3.3). We refer to these codes as subgraph codes.

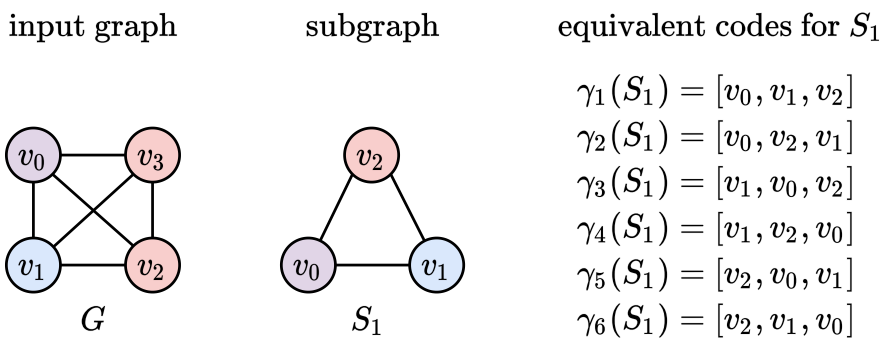


Figure 3.3. Example of input graph, subgraph, and equivalent codes [dos Santos Dias 2023]. In the provided example it is sufficient to specify vertex ordering, as each vertex in the ordering induces new edges connecting it to previous vertices in the ordering.

We say that these equivalent orders representing the same subgraph are automorphic to each other – i.e. they represent isomorphisms from the subgraph to itself. To prevent redundancy in computation and information, GPM systems usually strict themselves to visiting only a single **canonical representative code**[2] for each subgraph to prevent redundant and unnecessary work. Again, the details about enumeration algorithms is out of the scope of this course, more details can be found in [Dias et al. 2019]. Instead, in this text we are going to highlight the two main subgraph exploration paradigms used in subgraph enumeration and adopted by most GPM systems. In a **pattern-oblivious** subgraph enumeration, subgraphs (induced or not) of no particular pattern are extracted from the

[2]not to be confused with *canonical pattern*

input graph and hence, the subgraph codes produced can be of two types: a sequence of vertices in case of induced subgraphs or a sequence of edges in case of general subgraphs. Figures 3.4b and 3.4c illustrate two enumeration trees from the input graph of Figure 3.4a. The first represents the enumeration of *induced subgraphs* with 3 vertices and hence, each root-to-leaf path is a subgraph code used to represent a particular subgraph in the input graph. The second tree represents the enumeration of *general subgraphs* and hence, the subgraph codes depicted in the figure are sequences of edges instead of vertices. Nevertheless, both enumeration trees are pattern-oblivious since the subgraphs obtained are of possibly different patterns. In a **pattern-aware** subgraph enumeration, only subgraphs of a particular given pattern are extracted. This process is also known as *pattern/subgraph matching*. Figure 3.4d shows an enumeration tree conditioned by a 4-clique pattern, i.e., each path represent an enumerated subgraph that in the input graph has the structure of a complete subgraph with 4 vertices.

3.2.1. GPM systems: efficient abstractions for GPM

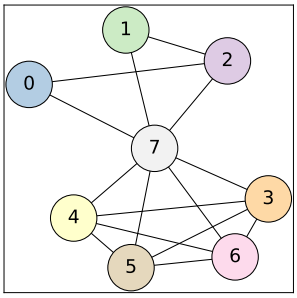
The main goal of a GPM system is to provide abstractions that properly navigate the trade-off between performance and programming experience concerning the modeling and development of subgraph enumeration and user-defined processing of visited subgraphs. The performance in this context is central because often GPM tasks include combinatorial algorithms that require efficient system optimizations to scale in parallel and/or distributed architectures. The programming experience is also important since they guarantee the applicability of the system in real-world data analysis pipelines, such as machine learning and data mining. Thus, important features of GPM systems include:

1. High-level programming abstractions to cope with the complexity of GPM theory (e.g. isomorphism, enumeration algorithms, canonical definitions and so on);
2. General-purpose programming design to enable handling user-defined application semantics;
3. System optimization strategies capable of scaling general-purpose applications in parallel/distributed architectures.

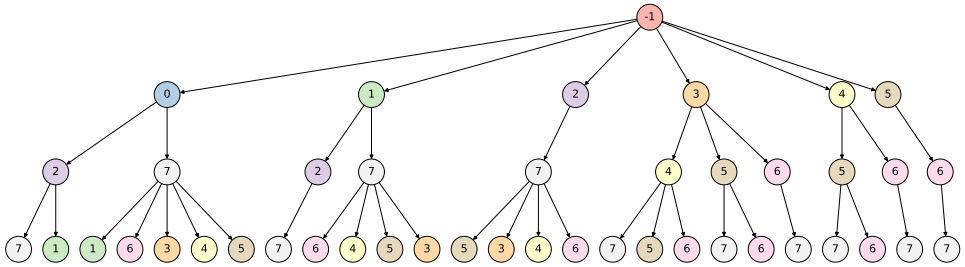
Properly providing all these wanted features simultaneously is not an easy task since systems performance and abstractions are often handled as a compromise. Next we review the literature of GPM systems and give an overview of how seminal works approach this trade-off.

3.3. GPM tools and Related work

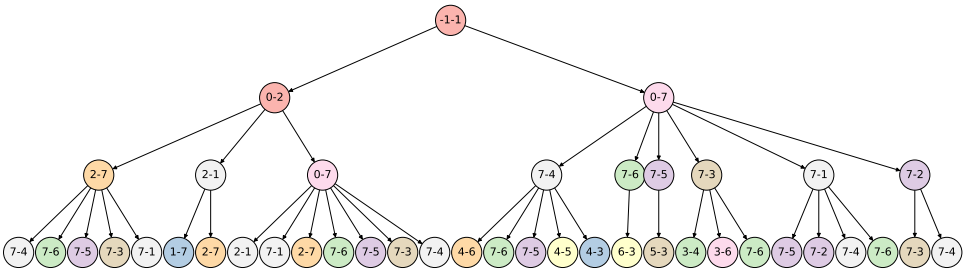
Table 3.1 briefly reviews the most seminal GPM systems in the literature. We classify the existing systems according to the important features established in Section 3.2.1. Concerning column "Productivity", a GPM system gets a HIGH in this criterion if, and only, if it can be programmed with a handful of lines and also allows easy integration with other systems; we consider a GPM system to have a FAIR productivity when it supports concise programming but fails in providing integration with other systems; finally, LOW productivity systems are both challenging to deploy and to integrate with other systems. Two



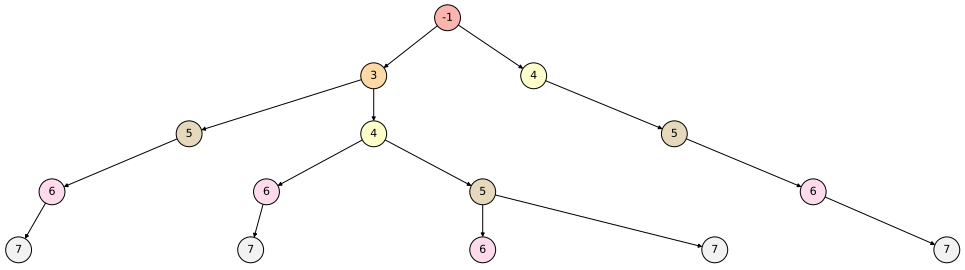
(a) Input graph example: vertex colors represent individual vertices.



(b) Induced subgraphs with 3 vertices via pattern-oblivious exploration.



(c) Subgraphs with 3 edges via pattern-oblivious exploration (entire tree has 182 leaves, and each node represents an edge).



(d) Pattern-aware exploration given a 4-clique pattern.

Figure 3.4. Enumeration trees given different target subgraph types. Each node in this tree represent one item in the subgraph code (vertex or edge), each edge in this tree indicates the order in which the subgraph code is extracted, hence each *root*→*leaf* path represent one unique subgraph from the graph of Figure 3.4a.

systems, Fractal [Dias et al. 2019] (CPU) and DuMato [Ferraz et al. 2024] (GPU), are selected to give in this chapter a practical overview of applications and system challenges.

Next we detail the related work of GPM systems.

Table 3.1. Selected related works on GPM systems: Fractal and DuMato are used as tools in this work to for practical examples.

	Proc. Unit	Parallel	Distrib.	Productivity
Fractal [Dias et al. 2019]	CPU	YES	YES	HIGH
DuMato [Ferraz et al. 2024]	GPU	YES	NO	LOW
Peregrine [Jamshidi et al. 2020]	CPU	YES	NO	FAIR
Pangolin [Chen and Arvind 2022]	GPU	YES	NO	LOW
G2Miner [Chen and Arvind 2022]	GPU	YES	NO	LOW
Tesseract [Bindschaedler et al. 2021]	CPU	YES	YES	FAIR
Arabesque [Teixeira et al. 2015]	CPU	YES	YES	FAIR
RStream [Wang et al. 2018]	CPU	YES	NO	FAIR
AutoMine [Mawhirter and Wu 2019]	CPU	YES	NO	FAIR
GraphZero [Mawhirter et al. 2021]	CPU	YES	NO	FAIR

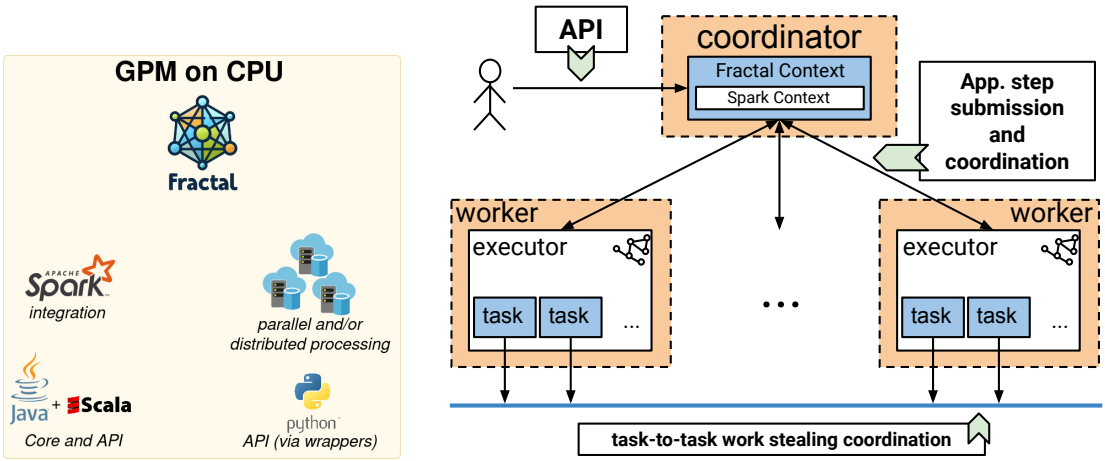
GPM systems for CPU. *Arabesque* [Teixeira et al. 2015] is one of the first GPM systems targeting distributed memory machines. The enumeration engine of Arabesque is known as pattern-oblivious, as the subgraphs are visited with no pattern information, i.e., vertex- and edge-induced only. Arabesque also proposes a data structure to compress subgraphs in-memory and to mitigate the memory demands of the BFS-style exploration while it also employs load balancing. *RStream* [Wang et al. 2018] is a relational GPM system that relies on expensive join operations to perform subgraph enumeration. It presents limitations caused by high memory consumption as the length of enumerated subgraphs increases. *Fractal* [Dias et al. 2019] is a distributed memory CPU-based GPM system that uses a DFS exploration strategy to reduce memory demands. Fractal proposes and implements a hierarchical work-stealing mechanism to mitigate load imbalance. *AutoMine* [Mawhirter and Wu 2019] proposes an automated code generation for GPM algorithms on CPU. It employs efficient scheduling of intersect/subtract operations to automate code generation for custom patterns. *Peregrine* [Jamshidi et al. 2020] is a parallel GPM system designed for shared-memory CPU machines. GraphZero [Mawhirter et al. 2021] is a compilation-based GPM system which improves AutoMine's schedule generation and symmetry breaking. Both Peregrine, AutoMine and GraphZero use an exploration strategy known as pattern-aware, where canonical pattern representatives are used to guide the subgraph enumeration by leveraging specialized execution plans (i.e. pattern-induced subgraphs are produced). Although pattern-aware exploration is efficient for enumerating small subgraphs, it has limitations whenever the application searches for a large number of canonical representatives (e.g., counting large motifs). In general, CPU based GPM systems offer a limited scaling capability since the number of execution units is reduced but, on the other hand, can be more easily integrated into data analysis pipeline due to simplified memory management and programming interfaces.

GPM systems for GPU. *Pangolin* [Chen et al. 2020] is a GPM system designed for GPU and follows the pattern-oblivious enumeration using the BFS exploration strategy. Pangolin's design enables execution optimizations by pruning the search-space of subgraphs and by reducing the amount of isomorphism tests required. Materialized intermediate states generated by the BFS exploration facilitate the runtime to leverage BSP (*Bulk Synchronous Parallel*) load balancing schemes. However, the BFS high memory demand limits its applicability to enumerate small subgraphs. Besides, Pangolin does not leverage optimizations to handle irregularity of parallel GPM algorithms on GPU and relies on CPU frameworks to perform isomorphism tests. *G2Miner* [Chen and Arvind 2022] improves Pangolin by supporting multi-GPU executions for speedup and a DFS pattern-aware subgraph enumeration engine. *DuMato* [Ferraz et al. 2024] is a pattern-oblivious GPM system with a warp-centric execution model and an automated work redistribution scheme, allowing improved memory access on GPU and more balanced executions. In general, GPU systems have a greater potential for scaling GPM tasks due to its massive number of processing units. However, this potential is challenging to fully leverage since real-world data is often skewed, which implies in very poor resource utilization if not employed together with coalesced memory access and load balancing mechanisms.

3.4. Fractal system: GPM processing on Big-Data stack

Fractal [Dias et al. 2019] is a high-performance and high-productivity system implemented over Spark [Zaharia et al. 2012] that provides a compositional API that facilitates the design of GPM algorithms while ensuring competitive parallel performance. The system (both design and implementation) is publicly available as an open-source software. Fractal core implements the main features, such as subgraph enumeration and aggregation engine and Spark integration (Figure 3.5a). Programming in Fractal may be with Scala API (core implementation) or with Python wrappers, which expose some of the main features via Py4J[3] interfaces. Figure 3.5b shows the architecture of the system. The user interacts with a local coordinator via an API that abstracts GPM processing and facilitates the application design. This program is automatically parallelized and distributed among worker nodes, each responsible for some portion of the original work. Each task in this architecture represents such unit of work that can be done in parallel. Besides this transparent parallelization provided by Spark, Fractal also extend the existing design to enable task-to-task communication for load balancing. We highlight that this load balancing protocol is central to the scalability of the system, as GPM workload is often very skewed (scale-free networks). Therefore, two points summarize the main contributions of Fractal: (1) programming abstractions that improve user experience with complex GPM algorithms and that allows them to be easily integrated into existing data analysis pipelines; and (2) extension of Spark's execution model to accommodate dynamic load balancing via work-stealing during parallel subgraph enumeration and aggregation routines. Although we do not further discuss runtime performance matters within Fractal, in Section 3.5 we address some of the main challenges in implementing efficient GPM systems.

[3] https://www.py4j.org/



(a) Fractal features (b) Fractal architecture

Figure 3.5. Fractal: a high productivity GPM system for parallel/distributed GPM.

3.4.1. Fractal installation and execution environment initialization

Fractal is essentially an extension over Spark Core JVM (*Java Virtual Machine*) programming interface and thus, Fractal programs are executed as Spark applications. This means that, provided that we have Fractal's package (`.jar` in this case) and dependencies, a Fractal application can be submitted via any of the following methods: (1) via Spark's `spark-submit` tool for batch executions; or (2) via some interactive engine such as Spark's `spark-shell` and other programming notebooks. Due to the practical approach of this material, we are going to address only how to compile the project and use the interactive alternatives for interfacing with Fractal applications. Nevertheless we point the curious reader to the official page of the project[4], which contains detailed information for deploying Fractal for batched executions as well.

The most straightforward approach to setup an interactive Fractal application is through the use of Python notebooks, for instance, Google Colab[5] or Jupyter[6]. The following Python notebook command is the only requirement to setup a local running environment for Fractal applications. This set of commands are expected to (1) download Fractal and Spark dependencies, and (2) compile and install any dependencies, including Spark pre-built library.

```
!git clone -b sbbd2024_course https://github.com/dccspeed/fractal.git
!cd fractal/python/ && make && make install
!pip install -q networkx tabulate matplotlib pygraphviz
```

To confirm that the installation were successful, the following python libraries should be available for importing:

```
from pyspark.sql import SparkSession  # Spark entrypoint for apps
import pyfractal                      # Fractal python wrapper
```

[4]Fractal project page: https://github.com/dccspeed/fractal
[5]Google Colab Cloud-hosted Notebooks: https://colab.research.google.com/
[6]Jupyter notebooks: https://jupyter.org/

```
import networkx as nx                    # Graph visualization
import torch                             # Tensor API and PyG models
```

A Spark session is the entrypoint in a Spark application that abstract all the underlying parallel and distributed computation. Fractal wrapper provides a default configuration builder for Spark, including important configurations such as dependencies and default parameters. At this point that the underlying execution environment of Spark can be configured to make use of parallelism and/or a cluster of machines. Details on how to create a Spark session on a distributed environment are available on the official Spark documentation[7], and are out of the scope of this practical review. From a Spark session, a Fractal context can be created – as we may see, a Fractal context is the basis for designing and executing GPM applications.

```
builder = pyfractal.DefaultSparkBuilder
builder = builder.master("local[8]")                    # 8 local threads
builder = builder.config("spark.driver.memory", "2g") # 2GB memory
builder = builder.appName("FractalQuickstartApp")
spark = spark_builder.getOrCreate()                     # spark session
fc = FractalContext(spark)                              # fractal context
```

3.4.2. Preparing the graph data

A Fractal context object allows the creation of graph objects that can be used as input to GPM tasks. These graph objects, as we may see, are built from a directory containing the graph data. The format supported for graph data is illustrated in Figure 3.6 for a toy example. The graph data must include the following mandatory files: `metadata`, containing the number of vertices and edges of the graph; `adjlists`, containing the adjacency lists of the graph. Additionally, vertex and/or edge labeling information can be included via files `vlabels` and `elabels`, respectively. In this course, due to brevity, we only address unlabeled and vertex labeled graphs.

The following code snippet can be used to create a Fractal graphs from the directory path for the toy example in Figure 3.6. Note that the same graph data can be loaded as a vertex labeled graph (Figure 3.6d) or unlabeled graph (Figure 3.6c) – in the latter the system just ignores file `vlabels` and assigns a default single label to each vertex.

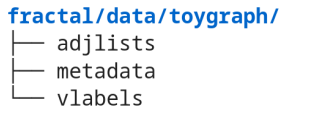
```
unlabeled_graph = fc.unlabeled_graph("fractal/data/toygraph/")
vlabeled_graph = fc.vertex_labeled_graph("fractal/data/toygraph/")
```

The Fractal graph object is the entry point for building GPM tasks. We highlight also that in case the system is deployed in a distributed setting, the graph data is loaded on each worker machine independently (i.e. the graph data is replicated).

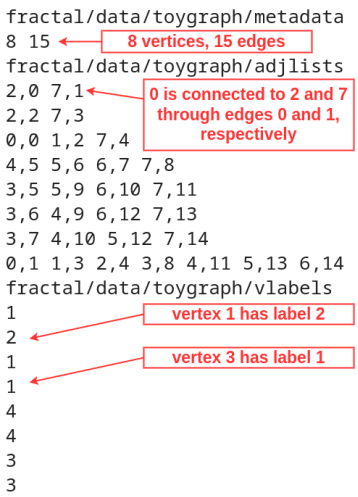
3.4.3. Fractal programming model and library

Table 3.2 summarizes the Fractal functions discussed in this course. These are the basic calls that allows the practitioner to build customized GPM applications or to access existing optimized implementations.

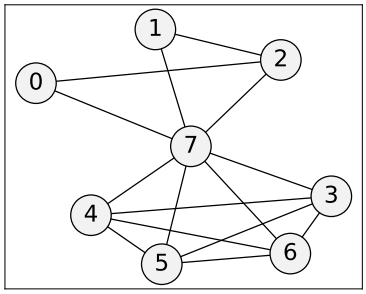
[7]Spark deploy: https://spark.apache.org/docs/3.5.0/cluster-overview.html



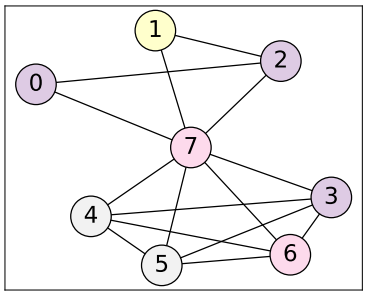
(a) Directory tree



(b) Files' contents



(c) ToyGraph read as unlabeled `unlabeled_graph`



(d) ToyGraph read as vertex-labeled `vlabeled_graph`

Figure 3.6. ToyGraph: Fractal default graph input format. Colors in this figures represent vertex labels.

Fractal low-level API for custom applications

Fractal low-level API allows building custom GPM applications, with user-defined semantics. Next we drilldown into the main operators of Fractal API, accessed via the Python Wrapper. As we may see, these building blocks allow the enumeration of different types of subgraphs/paradigms [VI,EI,PI], the filtering of the search-space [EX,FI], and the efficient output retrieval via Spark's RDDs [SN]. Spark RDD stands for *Resilient Distributed Datasets*, and allows efficient, parallelized, distributed and fault-tolerant processing of datasets, especially useful in settings where the amount of data is massive (BigData). The subgraph enumeration settings are configured by three alternative function calls: `vfractoid()` indicates that enumeration must produce induced subgraphs via a pattern-oblivious paradigm, such as depicted in Figure 3.4b; `efractoid()` indicates that enumeration must produce not induced (general) subgraphs via a pattern-oblivious paradigm, such as depicted in Figure 3.4c; `pfractoid(pattern, induced)` indicates that enumeration must produce subgraphs of a particular `pattern`, and subgraphs are induced or not depending on the flag `induced`.

[VI,EX,FI,SN] Induced subgraphs via a pattern-oblivious exploration: Suppose a GPM routine whose goal is to extract induced subgraphs from the input graph that contain

Table 3.2. Fractal Python Wrapper API and built-in library. Only the functions addressed in this course are listed in this table, please refer to the official documentation for a complete view.

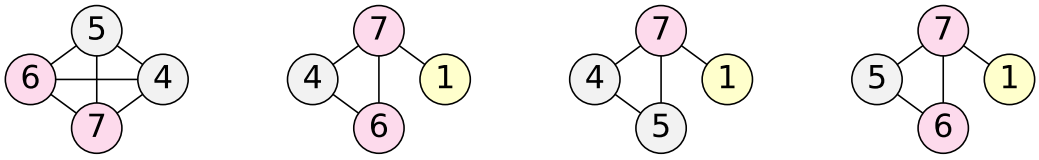
Low-level API (Purpose: to build custom applications)
[VI]:`vfractoid()` [EI]:`efractoid()` [PI]:`pfractoid(pattern, induced)`
[EX]:`extend(k)` [FI]:`filter(subgraph_filter)` [SN]:`subgraphs_networkx()`

Built-in Library (Purpose: to leverage optimized built-in applications)
[MC]:`motif_counting(k)` [PQ]:`pattern_querying(pattern, induced)`
[CC]:`cliques(k)` [QC]:`quasi_cliques(k, min_density)`
[KH]:`khop_induced_subgraphs(k)` [GV]:`graphlet_degree_vectors(k)`
[FSM]:`frequent_subgraph_mining(k, min_support)`

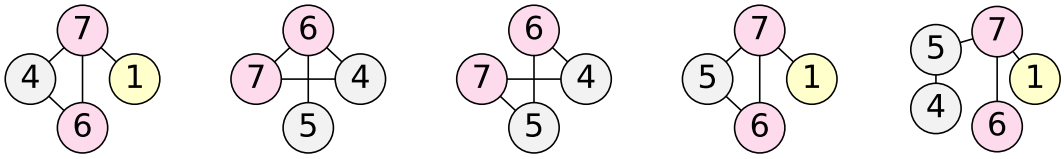
a certain set of vertex labels. The following example depicts an application that extracts induced subgraphs with 4 vertices, but only those in which each vertex have some label of interest. In this example, this predicate condition is enforced through the user-defined function `filter_func`, which returns True iff each vertex has label in the set $\{2,4,3\}$. Note also that the application is build sequentially from the input graph: first the intention to mine vertex induced subgraphs, followed by the configuration or size it is wanted for the subgraphs, and finally a label filtering function is applied to the set of subgraphs. The output is returned as a Spark RDD of NetworkX graphs, so they can be used for downstream processing tasks.

```
labels_of_interest = set([2,4,3])

# check whether a subgraph contains only labels of interest
def filter_func(s):
    for vlabel in nx.get_node_attributes(s, 'label').values():
        if vlabel not in labels_of_interest: return False
    return True

subgraphs_app = vlabeled_graph.vfractoid()          # induced subgraphs
subgraphs_app = subgraphs_app.extend(4)             # add 4 vertices
subgraphs_app = subgraphs_app.filter(filter_func) # user-def filter
subgraphs = subgraphs_app.subgraphs_networkx()     # output: nx graphs
```

Next we illustrate a few outputs of this code. Notice how every induced subgraph has 4 vertices and none contains vertex with labels in $\{1\}$ (represented as purple in Figure 3.6).

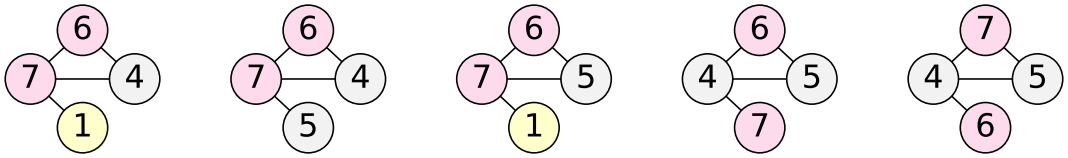


[EI] Not induced subgraphs via pattern-oblivious exploration: Also, the following output would be the result if `efractoid()` were used instead of `vfractoid()`. In this case, only 4 edges are allowed, which produces general subgraphs with at least 4 vertices and at most 5 vertices.



[PI] Subgraphs via pattern-aware paradigm: Fractal API also allows exploring the search space of subgraph of a particular predetermined pattern of interest, in this case `pfractoid()` must be set before adding vertices. The following code snippet generates only subgraphs whose pattern is a not induced "tailed triangle".

```
tailed_triangle = nx.from_edgelist([(0,1),(0,2),(0,3),(1,2)])
subgraphs_app = vlabeled_graph.pfractoid(tailed_triangle, induced=False)
# ... the rest is exactly the same as the first example above
```



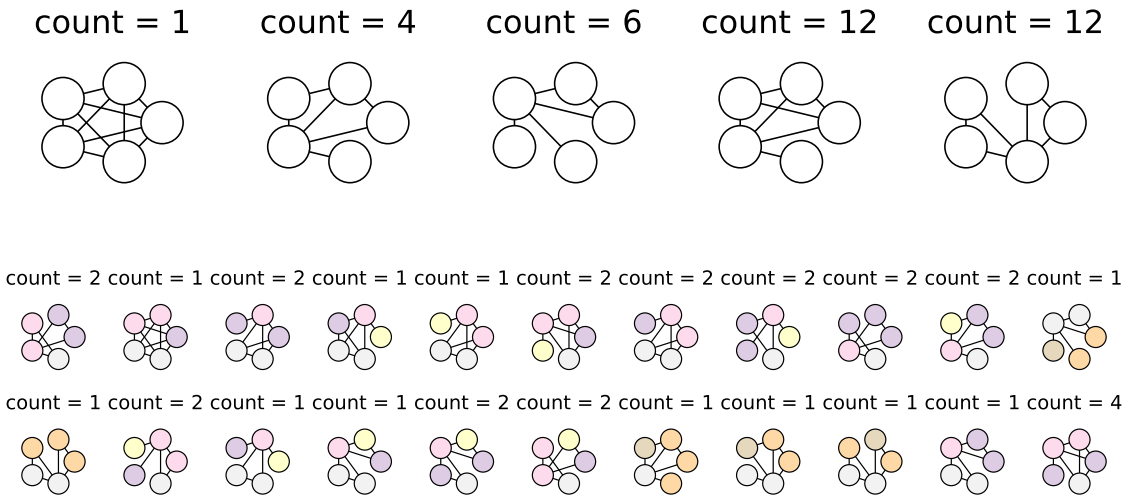
Fractal built-in library

For the traditional GPM tasks, Fractal offers off the shelf standard implementations. In this section we cover some of these applications, which can be accessed directly from a Fractal graph object.

[MC] Motif Counting (parameters: integer k): Given a number of vertices k and an input graph G, the goal of Motif counting is to output how many *unique* induced subgraphs of each different pattern exist in G. Motif counting is often used to characterize

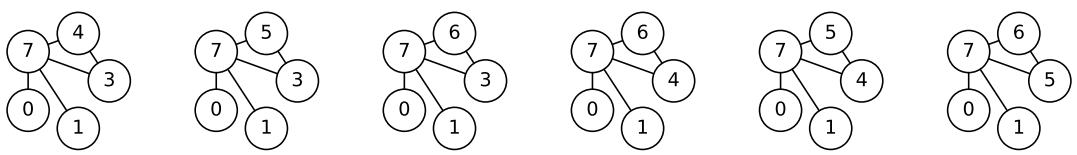
complex networks [Ribeiro et al. 2021] and the its computation allows the extraction of *Graph Degree Vectors* (a.k.a. GDV vectors) to be used as node features [Pržulj et al. 2004] in graph learning tasks. Next we show the code and output of counting motifs with 5 vertices in both the unalabeled and vertex-labeled version of the input graph (Fig. 3.6).

```
motif_count_unlabeled = unlabeled_graph.motif_counting(5)
motif_count_vlabeled = vlabeled_graph.motif_counting(5)
```



[PQ] Pattern Querying (parameters: pattern p): Pattern querying (a.k.a. pattern matching or subgraph matching) receives as input a pattern p and must list all the unique subgraphs (induced or not, depending on the goal) of such pattern in G. This computation is somewhat related to Neoj's[8] `MATCH` clause, however subgraph uniqueness during enumeration is not explicitly enforced in these transactional database systems. GPM systems, on the other hand, are more specialized and prepared to deal with such isomorphism-related issues. In the following example we see examples of subgraphs of a specific pattern (triangle with two tails). Notice also that induced flag is set to true, which indicates that induced subgraphs are to be considered (i.e. exactly the same 6 subgraphs counted in the motif counting example above). In case of flag induced set to false, the number of subgraphs of this particular pattern would be much larger, as this pattern is contained in other patterns, for example, a clique with 5 vertices.

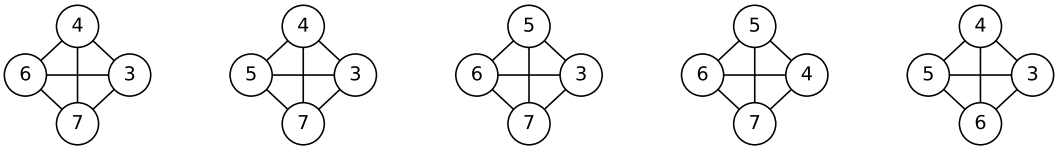
```
pattern_5 = nx.from_edgelist([(0,1),(0,2),(0,3),(0,4),(1,2)])
subgraphs = unlabeled_graph.pattern_querying(pattern_5, induced=True)
```



[8] https://neo4j.com/

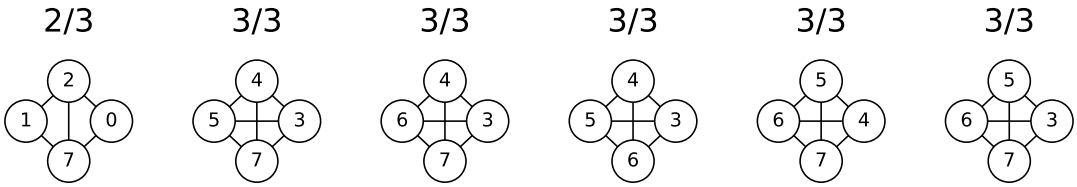
[CC] Clique Listing (parameters: integer k): Clique listing is a problem whose goal is to count/list all cliques with k vertices in the graph G. A k-clique is a complete subgraph of G with exactly k vertices. If the density condition is relaxed, more diverse output can be generated through *quasi clique listing*. Next a small example of clique enumeration, code followed by output.

```
cliques = unlabeled_graph.cliques(4)
```



[QC] Quasi-clique Listing (parameters: integer k, density threshold α): In Quasi clique listing the goal is to find in G induced subgraphs satisfying a minimum density measure threshold $0 < \alpha < 1$. Many density measures exist and for the purpose of this work, a subgraph S is considered dense iff for each one of its vertices u, its degree in the subgraph is at least $\lceil \alpha * (k-1) \rceil$. Dense subgraphs are used to extract communities [Dourisboure et al. 2009], to identify surprising groups in complex networks [Hooi et al. 2020], to improve graph compression [Buehrer and Chellapilla 2008], among others. Next we show the code and the output (with densities) for a quasi clique finding call requesting subgraphs with 4 vertices and minimum density of $2/3$, i.e., each subgraph vertex must be connected to at least other two in the subgraph.

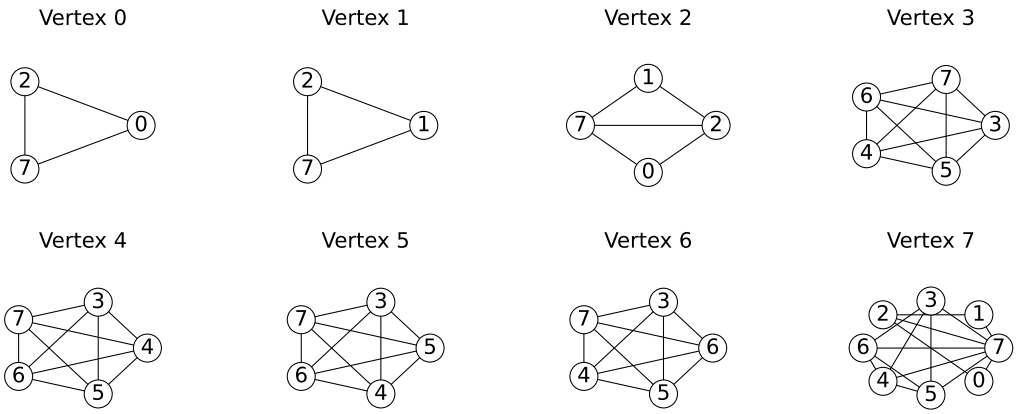
```
quasi_cliques = unlabeled_graph.quasi_cliques(4, 2/3)
```



[KH] K-hop subgraph (a.k.a. ego network) extraction (parameters: integer k): The goal of this procedure is to extract k-hop induced subgraphs around each vertex of graph G. This task is often used in machine learning pipelines as a mean to extract local node features., i.e., the surroundings of a node in a graph is expected to provide important insights on its role in the graph. Ego-Networks are widely used in graph learning tasks, for instance, k-hop can be used to map node-level tasks (e.g. node classification) into graph-level tasks (e.g. graph classification of k-hop subgraphs extracted from a graph) [Sun et al. 2023]. Also, reachability queries may benefit from such computation, limiting the search space that needs to be considered [Jin et al. 2009]. The following code and Figure shows the 2-hop induced subgraphs of the graph in Figure 3.6. Notice how ego networks

have an interesting capability of distinguishing similar node neighborhoods, fact that we explore in more details for a real-world graph in Section 3.4.5.

```
khop_subgraphs = unlabeled_graph.khop_induced_subgraphs(2)
```



[GV] Graphlet-Degree Vectors (parameters: integer k): For machine learning tasks on graphs in which the structure of the input graph is relevant to the target goal, capturing such information as node-features may substantially improve the overall quality of the models. One approach for doing this is through *Graphlet-Degree Vectors* extraction. This feature extraction strategy consists of building a counting vector per vertex/node in the graph. Each vector contains the frequency counts of all possible motif/graphlet orbits up to a pattern size k. In general terms, graphlet orbits represent non-equivalent positions in a set of motifs up to size k. Figure 3.7 shows an example of the GDV of vertex u in a graph – notice how four distinct positions exists in motifs up to 3 vertices and hence, the GDV is composed of these four dimensions. This GPM task is implemented by enumerating all induced subgraphs of each possible motif and counting each subgraph position individually according to the unique positions determined as graphlet orbits. The output is one vector per vertex in the graph. Below we illustrate the API call and result for the unlabeled ToyGraph. In this case, graphlet orbits are extracted from motifs of up to 4 vertices, resulting a 15-dimension feature vector for each one of the 8 vertices in the graph.

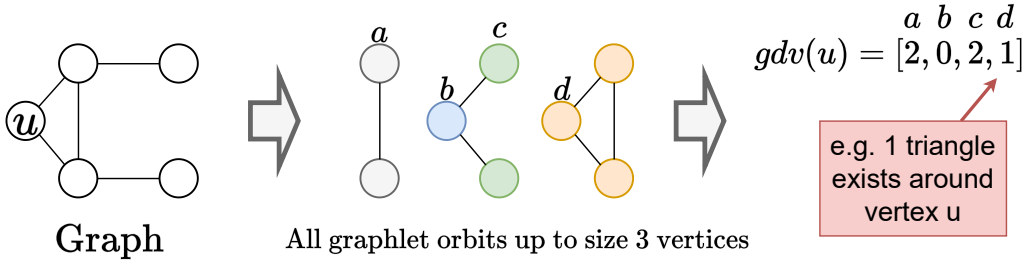


Figure 3.7. Graphlet-Degree Vector example: extracting node-features via GPM tasks.

```
gdvs = unlabeled_graph.graphlet_degree_vectors(4)
```

```
tensor([[ 2.,  8.,  1.,  1., 20.,  4., 16.,  0.,  1.,  6.,  0.,  3.,  1.,  0.,  0.],
        [ 2.,  6.,  1.,  1., 18.,  4., 10.,  0.,  1.,  6.,  0.,  3.,  1.,  0.,  0.],
        [ 1.,  6.,  1.,  1., 20.,  5.,  6.,  1.,  1.,  5.,  1.,  3.,  0.,  1.,  0.],
        [ 4., 12.,  1.,  6., 24.,  4., 12.,  1., 12., 15.,  2.,  2.,  6.,  2.,  4.],
        [ 3.,  9.,  1.,  3.,  4.,  4.,  6.,  1.,  3.,  2.,  2.,  2., 11.,  1.,  1.],
        [ 2.,  6.,  1.,  1., 12.,  4.,  2.,  1.,  3.,  2.,  2.,  2.,  5.,  1.,  1.],
        [ 1.,  3.,  1.,  1.,  6.,  4.,  2.,  1.,  3.,  6.,  6.,  4.,  2.,  2.,  1.],
        [ 1.,  3.,  1.,  1.,  6.,  4.,  2.,  1.,  3.,  6., 15.,  1.,  2.,  1.,  1.]])
```

GDV of vertex 0

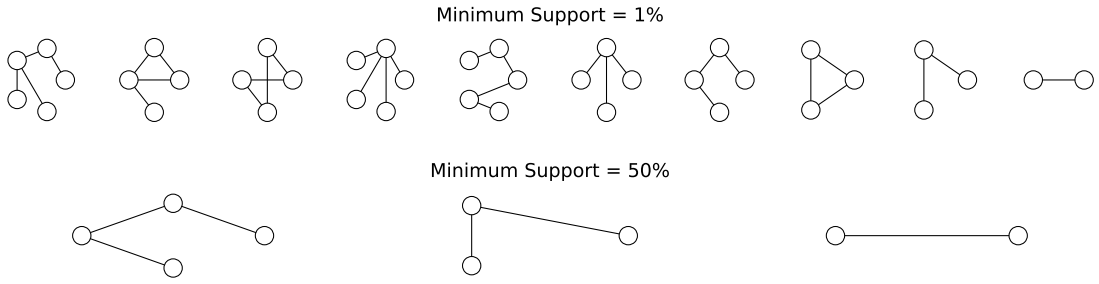
GDV of vertex 3

...

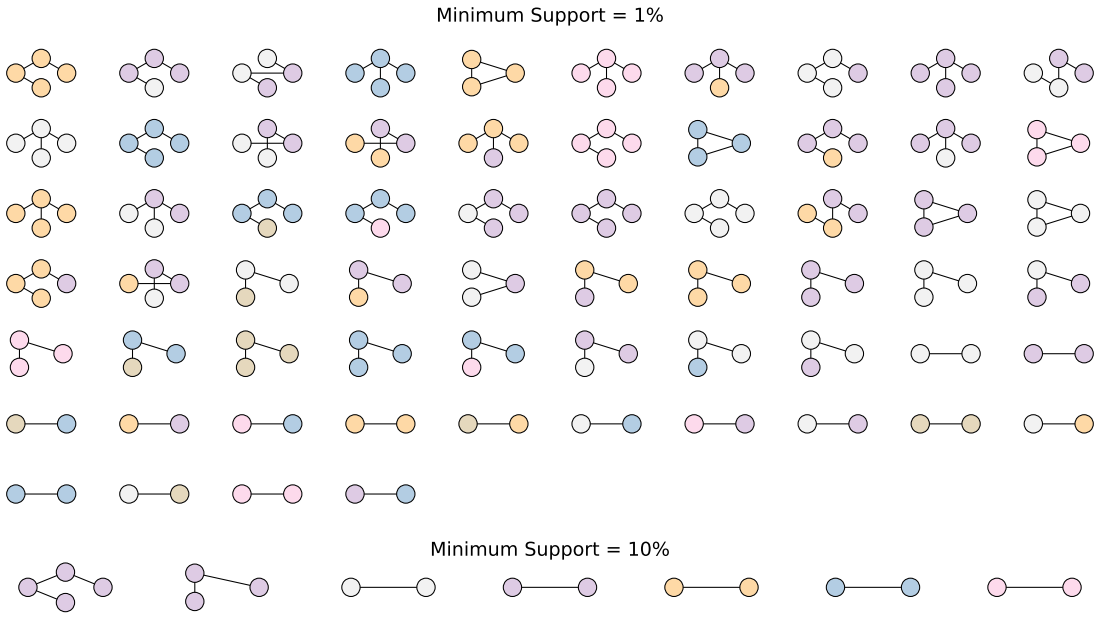
[FSM] Frequent Subgraph Mining (parameters: integer k, frequency threshold α): Frequent Subgraph Mining (FSM) leverages some frequency measure to extract from G those patterns with k edges meeting a minimum frequency threshold α. This frequency measure usually is defined to enforce the anti-monotonic property, i.e., every sub-pattern of a frequent pattern must also be frequent. Thus, if the input is a set of graphs, the frequency of a pattern can be simply the proportion of graphs containing the pattern [Yan and Han 2002]. For the single large graph setting, however, such counting frequency is not anti-monotonic. For this reason, for a single large graph, the frequency of a pattern is determined by other measures such as the *minimum image support* [Bringmann and Nijssen 2008]. Next we show an example of the usage of FSM algorithm over the real-world graph Citeseer [Elseidy et al. 2014], composed of Computer Science articles (total of 3312 nodes) and citations among them (total of 4732 edges). Vertex labels indicate the related area of the article (total of 6). In this case, the frequency is represented in terms of a percentage over the number of vertices in the graph. Thus, 1% minimum support indicates that only patterns whose minimum image support of at least "1% of the number of nodes in the graph" are considered frequent.

```
citeseer = fc.unlabeled_graph("fractal/data/citeseer")
frequent_patterns_by_support = dict()
for min_support in [0.01,0.5]:
    frequent_patterns = citeseer.frequent_subgraph_mining(4, min_support)
    frequent_patterns_by_support[min_support] = frequent_patterns
```

Below we see the result of this code for different support thresholds – larger minimum supports are more selective to the space of patterns. Also in this illustration we show the results for the unlabeled version of Citeseer graph.



If the graph is loaded as vertex-labeled, more candidate patterns are possible and consequently, it becomes more difficult for a pattern to be considered frequent. In the output below we show the frequent patterns for two other minimum support thresholds – in this case, a minimum support of 50% returns no patterns (not shown).



3.4.4. Use Case: Extracting node-level features for graph learning tasks

GPM can be used in the context of graph learning, enabling the extraction of meaningful structural features for machine learning. Node-level graph learning tasks is an example that can benefit from GPM computation. In a graph-level classification task, a database of graphs is given as input. Each graph has an associated class and the goal is to train a model that is competent in guessing the class of graphs given their structural information.

In MUTAG dataset, derived from a study of chemical compounds [Kriege and Mutzel 2012], each compound is a graph, and vertices represent atoms and edges indicate chemical bonds between a pair of atoms. Overall, MUTAG dataset consists of 188 chemical compounds divided into two classes according to their mutagenic effect on a bacterium. In this use case we are going to consider the following graph-level learning task over this dataset: to predict the class of a chemical compound given its structure. One possible approach to accomplishing this is to generated Graphlet Degree Vector (GDV) features for the vertices in the dataset and forward this modified dataset to a classification model of choice.

PyTorch Geometric (PyG)[9] is a graph learning framework focused in deep models on graphs (e.g. Graph Neural Networks and Shallow Embedding). PyG also facilitates the access to public datasets such as MUTAG and hence, in this use case we show how to generate GDV feature vectors for the MUTAG dataset and use this modified dataset in a PyG classification model for the task described above. Figure 3.8 illustrates the workflow of how to integrate Fractal output into Machine Learning Systems such as PyG.

The following code implements method `get_dataset_with_gdv` from Figure 3.8. The overall operation is to merge the whole dataset as a single Fractal graph and compute the GDV features for each vertex using Fractal built-in API [GV].The method returns a copy of the dataset with GDV as vertex features, represented in PyG as attribute `data.x`.

[9] https://pytorch-geometric.readthedocs.io/en/latest/index.html

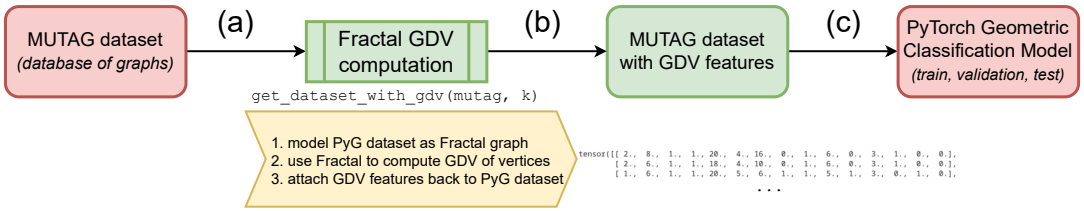


Figure 3.8. Workflow of use-case: Using Fractal to extract GPM features to enrich graph data for classification. (a) and (b) PyG MUTAG dataset is used as input for Fractal's computation of GDV features; (c) the modified dataset is forwarded to a machine learning pipeline of training/validation/testing.

```
def get_dataset_with_gdv(dataset, k):
    # compute graphlet degree vectors
    loader = DataLoader(dataset, batch_size=len(dataset), shuffle=False)
    full_batched_data = next(iter(loader))
    fg = fc.unlabeled_graph_from_pyg_data(full_batched_data)
    full_batched_data_x = fg.graphlet_degree_vectors(k)

    # apply GDV features to a copy of the dataset
    dataset_with_gdv_features = []
    for i in range(len(dataset)):
        data = dataset[i].clone()
        from_idx = full_batched_data.ptr[i]
        to_idx = full_batched_data.ptr[i + 1]
        x = full_batched_data_x[from_idx:to_idx]
        data.x = x
        dataset_with_gdv_features.append(data)
    return dataset_with_gdv_features
```

We omit the whole code since details on machine learning pipelines on graphs is out of the scope of this course and because our focus is on showing how GPM algorithms may be useful as preprocessing steps in such scenarios. Nevertheless, below we show the output of this workflow for a PyG Classification Model composed of three Graph Convolution Network [Kipf and Welling 2017] layers followed by a pooling mechanism for graph classification. The output is obtained from random dataset splits. While these results depend on specific hyperparameter configurations and may vary, stills holds that GPM features can be useful for graph learning tasks and hence, this integration is beneficial. Another important observation is that GDV features include *only structural information* concerning the surroundings of the vertices, which alone is enough to provide high accuracy results.

```
== Model with default features (atom attributes) ==
[01] Train Loss: 0.50 Validation Accuracy: 0.79 Test Accuracy: 0.68
[02] Train Loss: 0.49 Validation Accuracy: 0.79 Test Accuracy: 0.74
[03] Train Loss: 0.52 Validation Accuracy: 0.74 Test Accuracy: 0.68
[04] Train Loss: 0.54 Validation Accuracy: 0.84 Test Accuracy: 0.68
[05] Train Loss: 0.56 Validation Accuracy: 0.74 Test Accuracy: 0.79
Average Test Accuracy: 0.79

== Model with Graphlet Degree Vector Features ==
```

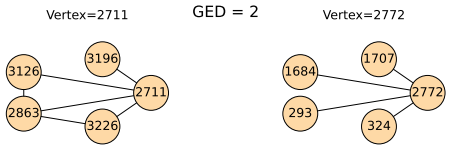
```
[01] Train Loss: 0.23 Validation Accuracy: 0.84 Test Accuracy: 0.79
[02] Train Loss: 0.28 Validation Accuracy: 0.84 Test Accuracy: 0.95
[03] Train Loss: 0.30 Validation Accuracy: 0.63 Test Accuracy: 0.79
[04] Train Loss: 0.32 Validation Accuracy: 0.74 Test Accuracy: 0.74
[05] Train Loss: 0.33 Validation Accuracy: 0.89 Test Accuracy: 0.89
Average Test Accuracy: 0.89
```

3.4.5. Use Case: Node similarity via Ego-Networks

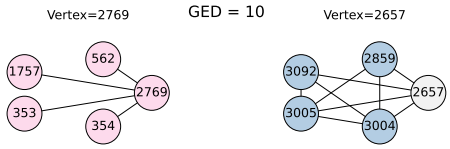
In this use case we illustrate how to leverage k-hop subgraphs (a.k.a. ego networks) to compare nodes in a labeled graph. We consider the real-world graph Citeseer [Elseidy et al. 2014], the same graph used in the FSM example above. An alternative to compute the similarity between nodes in a networks is as follows: (1) generate k-hop networks for each node; (2) compare nodes' networks by Graph Edit Distance (GED); (3) the output measure is an estimate on how similar two nodes are with respect to structure and labeling. The GED of a pair of graphs (G_1, G_2) is defined as the number of graph operations required to transform G_1 into G_2. The graph operations include adding/deleting nodes/edges and changing node labels. Therefore, a GED distance of 0 indicates that both graphs are exactly the same, while a GED of k indicate that this amount of graph operations is required to both be equal. This measure is included in the NetworkX package and thus, we can generate ego-nets using Fractal and compute pairwise similarities. The following code accomplishes this workflow – notice how we choose $k = 2$ to be the radius of the ego networks and it is defined that two nodes are equal whenever they share the same label.

```
citeseer = fc.vertex_labeled_graph('fractal/data/citeseer')
ego_nets = citeseer.khop_induced_subgraphs(2)

def node_match(n1, n2): return n1['label'] == n2['label']

for en1 in ego_nets:
    for en2 in ego_nets:
        ged = nx.graph_edit_distance(en1, en2, node_match=node_match)
        yield en1, en2, ged
```

From the output of the code above, we may select two pairs of graphs to exemplify this similarity computation. In the left side of the Figure below we see the most similar non-equal ego networks with 5 nodes in the Citeseer graph. These networks correspond to nodes 2711 and 2772, respectively, with a GED of 2 operations. Notice how both graphs share the same unique label for each node, so the 2 operations necessary are just edge additions to the first one. In the right side of the Figure below, we see the opposite: the least similar pair of ego networks with 5 nodes. In this case, the GED is equal to 10 because 5 label transformations and 5 edge additions are necessary to make both graphs equal.



Ego-net pair with minimum GED



Ego-net pair with maximum GED

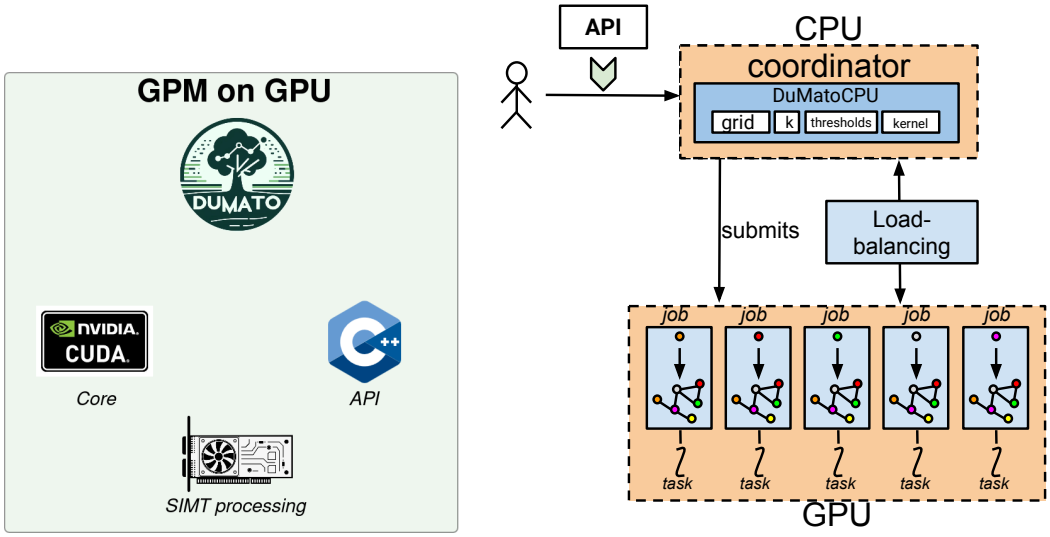
3.5. DuMato system: massively parallel GPM on GPUs

The CUDA architecture is the market-leading GPU architecture proposed by NVIDIA [Corporation 2024], and other GPU vendors such as AMD follow the same architectural concepts but using a distinct nomenclature. In this course we will adopt the CUDA nomenclature to define the standard GPU concepts.

A GPU is a massively parallel device that follows the SIMT (Single-Instruction Multiple-Thread) computing scheme. In this scheme, a parallel task is executed by the GPU by a group of 32 cores, and this execution unit is called a *warp*. Warps execute independently, but threads belonging to the same warp are supposed to execute the same instruction in lockstep to achieve maximum parallel efficiency. Besides, a warp shall follow a regular memory access pattern to reduce the amount of memory transactions needed to answer memory requests. When threads belonging to the same warp need to execute different instructions, we say there is a *divergence* and the parallel performance deteriorates. In the same sense, when threads belonging to the same warp follow irregular memory access patterns, we say there is *memory uncoalescence* and memory performance deteriorates. The modern GPUs provide distinct physical groups of cores (multiple of warp size) to execute a pool of warps, and these groups are named *Streaming Multiprocessors* (SM).

DuMato [Ferraz et al. 2022, Ferraz et al. 2024] is a graph pattern mining system implemented over the CUDA architecture, and supports efficient implementations of GPM algorithms on GPU through a high-level API and an well-defined execution workflow. It is also publicly available as a open-source software, and requires C/C++ 11 and CUDA 12. DuMato's core implements pattern-oblivious subgraph enumeration and provides high-level programming interfaces to allow an user to create custom routines to visit only specific induced subgraphs and generate the desired output of the GPM algorithm. DuMato uses a graph traversal named *DFS-wide*, which allows a predictable memory consumption and improves memory locality. One may develop a GPM algorithm with DuMato using C/C++ on host-side and CUDA C++ on device-side (Figure 3.9a).

The design and implementation of Graph Pattern Mining in any parallel computing environment must deal with two challenges: **irregularity** and **load imbalance**. The *irregularity* is related to the dynamism and unpredictability of the access pattern regarding the data structures (mainly the adjacency lists) during the visitation of subgraphs, generating overheads concerning memory uncoalescence and caching. Assuming different threads process different regions of the input graph, those threads tend to visit different amounts of subgraphs, incurring a *load imbalance* that may deteriorate parallel performance. Take Figure 3.4b for example, 7 induced subgraphs with 3 vertices starting at vertex 0 exist, whereas only 1 exists starting at vertex 5. This imbalance can only be exacerbated for large graphs, leading to poor resource utilization and runtime performance. In Section 3.5.2 we show real data to support the importance of a well-balanced GPM



(a) DuMato features (b) DuMato architecture

Figure 3.9. DuMato system for GPU-accelerated GPM.

execution and the performance benefits in incorporating such optimization into the runtime, meanwhile we provide further details on DuMato's design choices and programming overview.

3.5.1. DuMato architecture and programming overview

Figure 3.9b shows the architecture of the system. The user creates a *DuMatoCPU* object on CPU to define the grid size to be launched on GPU, the size k of the subgraphs visited, load-balancing thresholds and a function pointer to the GPU kernel for the desired GPM algorithm (implemented using DuMato API). The *DuMatoCPU* object submits GPM jobs to be executed by parallel tasks on the GPU. Given an input graph G and a subgraph S of G, a GPM job is responsible for visiting the desired subgraphs with k vertices of the GPM algorithm starting from S. In Figure 3.9b, each parallel task received one job that visits all desired subgraphs of the GPM algorithms starting from a specific vertex of the input graph. A parallel task may receive more than one job, depending on the load-balancing thresholds.

After submitting the job on GPU, the *DuMatoCPU* object starts the load-balancing layer on CPU, depicted by Figure 3.10. Each task on GPU carries a flag indicating whether its enumeration is active, and the load-balancing layer reads all GPU flags asynchronously. Once the load-balancing layer detects that the amount of idle GPU tasks is higher than a threshold, it stops the kernel execution on GPU, reads the current enumeration data, redistributes the jobs among the tasks in the grid, and resubmits the job to the GPU. The load-balancing layer also uses another parameter to indicate the amount of jobs each parallel task is supposed to receive after redistribution.

A DuMato parallel task depicted in Figure 3.9b may have two different granularities: *thread-level* and *warp-level*, named as *DM_DFS* and *DM_WC* is DuMato's specification, respectively. In the thread-level granularity, each GPM job is allocated to a single

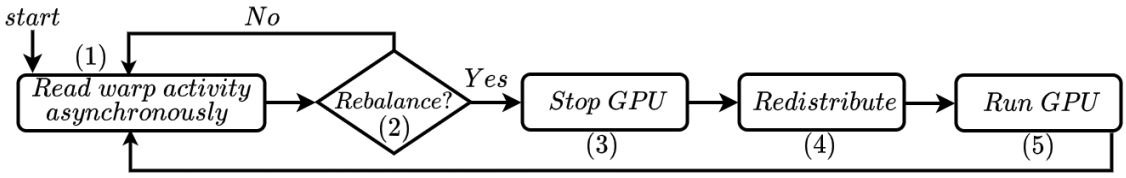


Figure 3.10. Load-balancing layer proposed by DuMato [Aquino 2023].

GPU thread. On the other hand, in the warp-level granularity each GPM job is allocated to a single GPU warp. The warp-level granularity was implemented using the warp-centric programming model [Hong et al. 2011], and is an optimization proposed by DuMato to improve GPU's parallel efficiency. This optimization allowed all threads within a warp to execute in lockstep to minimize GPU divergences (reducing the amount of individual instructions executed by GPU), as well as to facilitate memory coalescence and improve memory locality (reducing the amount of memory transactions needed to answer parallel memory requests). DuMato also provides a version named *DM_WCLB*, which extends the version *DM_WC* with the load-balancing performed by the CPU.

Table 3.3 shows DuMato's API used to build GPM algorithms on GPU. The function *control* is responsible for receiving load-balancing information from CPU, and function *move* goes forward and backward in the enumeration tree. Function *extend* creates new subgraphs from the current subgraph, and function *filter* eliminates subgraphs according to user-defined criteria defined by the function pointer parameter *P*. The aggregate functions are used to generate outputs for the algorithms, which can be a total counting (*aggregate_counter*), a counting per pattern (*aggregate_pattern*) and a buffer containing the subgraph codes of all visited subgraphs (*aggregate_store*). DuMato provides an skeleton code that can be used as the starting point to implement any GPM algorithm using the API.

Table 3.3. DuMato API. [Aquino 2023]

Functions	Scope
`[CT]` *control(TE)*	Algorithm-independent
`[MV]` *move(TE, genedges)*	
`[EX]` *extend(TE, begin, size)*	Algorithm-specific
`[FL]` *filter(TE, P, args)*	
`[A1]` *aggregate_counter(TE)*	
`[A2]` *aggregate_pattern(TE)*	
`[A3]` *aggregate_store(TE)*	

In the next section we present results of DuMato execution to understand the performance challenges associated with the parallel design and implementation of GPM algorithms on GPU.

3.5.2. Performance Challenges of GPM on GPU.

The experiments executed for this section were performed using Google Colab. Our goal is to demonstrate empirically the importance of load balancing in GPM systems, and in particular within GPU. Once a GPU environment is chosen, CUDA Toolkit is available and you may type the following commands to download and compile DuMato:

```
!git clone https://github.com/samuelbferraz/DuMato.git
!cd DuMato && make sm=75   # compile all built-in applications
```

All executable files will be available under the `DuMato/exec` folder after the compilation process. We executed the motif counting application provided by DuMato to evaluate the performance impacts of parallel granularity. The command lines executed are the following:

```
!./motifs_DFS ../datasets/citeseer.edgelist 3 102400 256
!./motifs_DFS ../datasets/citeseer.edgelist 4 102400 256
!./motifs_DFS ../datasets/citeseer.edgelist 5 102400 256
!./motifs_DFS ../datasets/citeseer.edgelist 6 102400 256
!./motifs_HAND_WC ../datasets/citeseer.edgelist 3 102400 256
!./motifs_HAND_WC ../datasets/citeseer.edgelist 4 102400 256
!./motifs_HAND_WC ../datasets/citeseer.edgelist 5 102400 256
!./motifs_HAND_WC ../datasets/citeseer.edgelist 6 102400 256
```

The executable `motifs_DFS` contains the *thread-level* implementation, while the executable `motifs_HAND_WC` contains the *warp-level* implementation. These two versions receive the same parameters, which are: the input graph (using the standard edge list format), the size of the visited subgraphs, the number of parallel threads and the block size. All executions were performed using the same amount of threads (102400) and the same block size (256), whose values are the default adopted in DuMato.

Table 3.4 shows the execution time of DuMato using the *thread-level* and *warp-level* granularity. For the small subgraph sizes (3 and 4), the executions are fast and the performance difference between the versions are not clear. However, starting from subgraph size 5, we clearly see that the version with the warp-level granularity provides speedups from 11x to 21x, showing that warp-level lockstep execution associated with good memory locality/coalescence are essential to keep performance as the size of visited subgraphs increases.

Table 3.4. Execution time (seconds) varying the parallel granularity.

Granularity	***Subgraph size (k)***			
	3	***4***	***5***	***6***
Thread-level	0.04	0.63	10.95	339.08
Warp-level	0.01	0.08	0.96	16.15

We also executed the motif counting application along with the load-balancing layer turned on, in order to evaluate the performance impacts of load-balancing. The command lines executed are the following:

```
!./motifs_DM_WCLB ../datasets/citeseer.edgelist 3 102400 256 8 30
!./motifs_DM_WCLB ../datasets/citeseer.edgelist 4 102400 256 8 30
!./motifs_DM_WCLB ../datasets/citeseer.edgelist 5 102400 256 8 30
!./motifs_DM_WCLB ../datasets/citeseer.edgelist 6 102400 256 8 30
```

The executable `motifs_DM_WCLB` implements the warp-level granularity along with the load-balancing layer on CPU. This executable receives two additional parameters: the amount of jobs assigned per task during the load-balancing phase (8, for all executions), and the upper-bound percentage of threads allowed to be idle without the need of calling the load-balancing layer (30%, for all executions). Table 3.5 compares the result of warp-level granularity without and with the load-balancing layer.

Table 3.5. Execution time (seconds) using the load-balancing layer.

Granularity	***Subgraph size (k)***			
	3	***4***	***5***	***6***
Warp-level	0.01	0.08	0.96	16.15
Warp-level with load-balancing	0.11	0.12	0.35	0.91

For the small subgraph sizes (up to 4), the version with load-balancing is a little worse than the standard warp-level version. This happens because the inclusion of the load-balancing layer generates an overhead that is not worth paying for short executions. The importance of load-balancing becomes clear after subgraph size 5, when we start seeing an increasing speedup. This happens because, as the size of the visited subgraphs increases, some regions of the input graph tend to concentrate most of the computation, and when the load-balancing is enabled, this data skewness is mitigated. This shows the importance of effective load-balancing on GPUs as we increase the size of visited subgraphs for any GPM algorithm, as GPUs are massive parallel environments which rely on active parallel threads throughout the entire execution to obtain valuable speedups.

Figure 3.11 shows the average SM occupancy (number of active warps per SM divided by the maximum amount of active warps per SM) through the execution of motif counting application for the Citeseer dataset, using subgraph size 8 and load-balancing threshold 30%. In other words, when the average SM occupancy is lower than 30%, the load-balancing layer is called to perform job redistribution. This behavior is depicted by the red line in the Figure, and every time we see a peak in the plot, it means the load-balancing layer was called. Although all SMs are active through the execution (green plot), the amount of active warps per SM varies. The choice of the appropriate load-balancing threshold is challenging, as the load imbalance varies depending on the dataset and the algorithm. An extensive evaluation of the load-balancing is beyond of the scope of this course, but you may check additional references that explain this trade-off deeply [Ferraz et al. 2024, Ferraz et al. 2022, Aquino 2023].

3.6. Conclusion

Graph Pattern Mining (GPM) refers to the processing of graph data that involves the extraction of subgraphs, being of especial interest to the data mining and machine learning

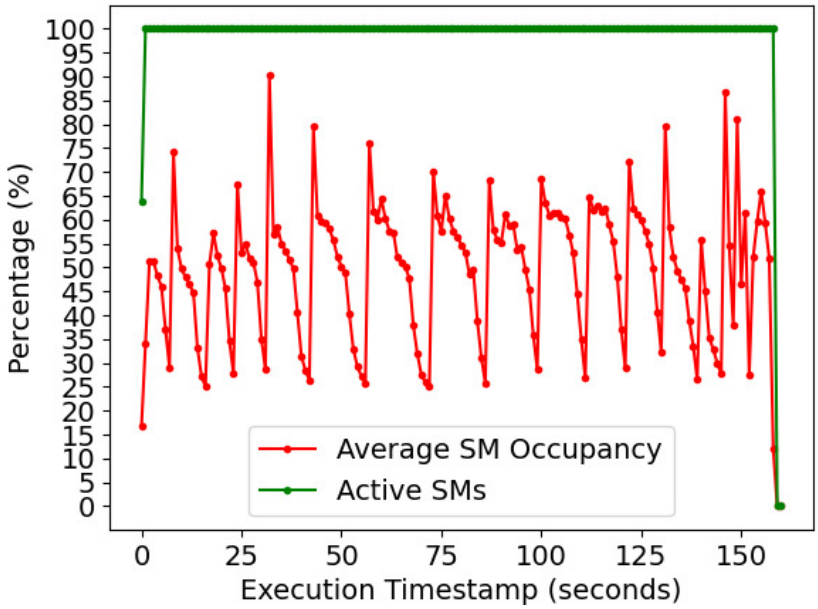


Figure 3.11. Load-balancing rounds of DuMato with warp-level parallelism and load balancing activated. In case, load balancing was turned off, the occupancy would progressively decay and hurt the overall performance.

communities. GPM systems emerged in the last decade as a solution to improve the user experience with GPM algorithms and also their runtime performance. Thus, GPM systems offer strong abstractions for programming that hide much of the complexity of algorithm design while ensuring performance scalability via system optimizations tailored for graph data processing. In general, GPM algorithms are often used as a preprocessing step to extract relevant knowledge from graph data. In this chapter, we walked through the main concepts, applications and challenges concerning GPM systems. Our practical approach leverages two recent GPM systems published as scientific work in the area: Fractal and DuMato, both open-source and public available.

Through the use of Fractal via Python Wrappers we are able to illustrate the main concepts and paradigms for enumerating and processing subgraphs. We give an overview of the programming interface which allows, with a few lines of code and inexpensive effort, to build custom user-defined applications and also to leverage existing built-in optimized application implementations. We also show use case scenarios where Fractal processing is integrated with other data analysis frameworks via Spark's resilient datasets (RDD) and, making complex data pipelines relying on GPM processing more natural and straightforward to implement and deploy. Through the use of DuMato system, we demonstrate how GPM processing can be accelerated using GPUs. Our examples offer a general picture on the main challenges in ensuring a proper parallel performance of GPM processing on GPUs: irregularity of real-world graph data and constant load imbalance. We take an experimental approach to understanding these challenges and show runtime performance that highlight the importance of the proposed optimizations for dealing with the aforementioned challenges.

Overall we believe this chapter gives a straightforward and concise overview of the topic, being useful for students, researchers, and other practitioners interested in learning interesting and efficient alternatives for extracting knowledge from graph data.

References

[Agrawal et al. 2018] Agrawal, M., Zitnik, M., and Leskovec, J. (2018). Large-scale analysis of disease pathways in the human interactome. *Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing*, 23:111–122.

[Aquino 2023] Aquino, S. B. F. (2023). *Strategies for efficient subgraph enumeration on GPUs*. Phd thesis, Federal University of Minas Gerais. Available at http://hdl.handle.net/1843/62443.

[Benson et al. 2016] Benson, A. R., Gleich, D. F., and Leskovec, J. (2016). Higher-order organization of complex networks. *Science*.

[Bindschaedler et al. 2021] Bindschaedler, L., Malicevic, J., Lepers, B., Goel, A., and Zwaenepoel, W. (2021). *Tesseract: Distributed, General Graph Pattern Mining on Evolving Graphs*, page 458–473. Association for Computing Machinery, New York, NY, USA.

[Borgelt 2007] Borgelt, C. (2007). Canonical forms for frequent graph mining. In Decker, R. and Lenz, H. J., editors, *Advances in Data Analysis*, pages 337–349, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Bringmann and Nijssen 2008] Bringmann, B. and Nijssen, S. (2008). What is frequent in a single graph? In *Proceedings of the 12th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, PAKDD'08, pages 858–863, Berlin, Heidelberg. Springer-Verlag.

[Buehrer and Chellapilla 2008] Buehrer, G. and Chellapilla, K. (2008). A scalable pattern mining approach to web graph compression with communities. In *Proceedings of the 2008 International Conference on Web Search and Data Mining*, WSDM '08, page 95–106, New York, NY, USA. Association for Computing Machinery.

[Chen and Arvind 2022] Chen, X. and Arvind (2022). Efficient and scalable graph pattern mining on GPUs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 857–877, Carlsbad, CA. USENIX Association.

[Chen et al. 2020] Chen, X., Dathathri, R., Gill, G., and Pingali, K. (2020). Pangolin: An efficient and flexible graph mining system on cpu and gpu. *Proc. VLDB Endow.*, 13(8):1190–1205.

[Corporation 2024] Corporation, N. (2024). NVIDIA Website. https://www.nvidia.com/. [Online; accessed 5-August-2024].

[Dias et al. 2019] Dias, V., Teixeira, C. H. C., Guedes, D., Meira Jr., W., and Parthasarathy, S. (2019). Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*.

[dos Santos Dias 2023] dos Santos Dias, V. V. (2023). *Graph pattern mining: consolidating models, systems, and abstractions*. Phd thesis, Federal University of Minas Gerais. Available at http://hdl.handle.net/1843/51806.

[Dourisboure et al. 2009] Dourisboure, Y., Geraci, F., and Pellegrini, M. (2009). Extraction and classification of dense implicit communities in the web graph. *ACM Trans. Web*, 3(2).

[Elbassuoni and Blanco 2011] Elbassuoni, S. and Blanco, R. (2011). Keyword search over rdf graphs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 237–242, New York, NY, USA. ACM.

[Elseidy et al. 2014] Elseidy, M., Abdelhamid, E., Skiadopoulos, S., and Kalnis, P. (2014). Grami: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.*, 7(7):517–528.

[Fan 2022] Fan, W. (2022). Big graphs: Challenges and opportunities. *Proc. VLDB Endow.*, 15(12):3782–3797.

[Ferraz et al. 2024] Ferraz, S., Dias, V., Teixeira, C. H., Parthasarathy, S., Teodoro, G., and Meira, W. (2024). Dumato: An efficient warp-centric subgraph enumeration system for gpu. *Journal of Parallel and Distributed Computing*, 191:104903.

[Ferraz et al. 2022] Ferraz, S., Dias, V., Teixeira, C. H., Teodoro, G., and Meira, W. (2022). Efficient strategies for graph pattern mining algorithms on gpus. In *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 110–119.

[Hoffman and Krasle 2015] Hoffman, F. and Krasle, D. (2015). Fraud detection using network analysis. Patent No. EP2884418A1, Filed September 1st., 2014, Issued June 17th., 2015.

[Hong et al. 2011] Hong, S., Kim, S. K., Oguntebi, T., and Olukotun, K. (2011). Accelerating cuda graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, page 267–276, New York, NY, USA. Association for Computing Machinery.

[Hooi et al. 2020] Hooi, B., Shin, K., Lamba, H., and Faloutsos, C. (2020). Telltail: Fast scoring and detection of dense subgraphs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):4150–4157.

[Huan et al. 2003] Huan, J., Wang, W., and Prins, J. (2003). Efficient mining of frequent subgraphs in the presence of isomorphism. In *Proceedings of the Third IEEE International Conference on Data Mining*, ICDM '03, pages 549–, Washington, DC, USA. IEEE Computer Society.

[Jamshidi et al. 2020] Jamshidi, K., Mahadasa, R., and Vora, K. (2020). Peregrine: A pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA. Association for Computing Machinery.

[Jin et al. 2009] Jin, R., Xiang, Y., Ruan, N., and Fuhry, D. (2009). 3-hop: a high-compression indexing scheme for reachability query. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, page 813–826, New York, NY, USA. Association for Computing Machinery.

[Junttila and Kaski 2007] Junttila, T. and Kaski, P. (2007). Engineering an efficient canonical labeling tool for large and sparse graphs. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 135–149, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

[Kipf and Welling 2017] Kipf, T. N. and Welling, M. (2017). Semi-supervised classification with graph convolutional networks.

[Kriege and Mutzel 2012] Kriege, N. and Mutzel, P. (2012). Subgraph matching kernels for attributed graphs. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, ICML'12, page 291–298, Madison, WI, USA. Omnipress.

[Kuramochi and Karypis 2005] Kuramochi, M. and Karypis, G. (2005). Finding frequent patterns in a large sparse graph*. *Data Min. Knowl. Discov.*, 11(3):243–271.

[Leon-Suematsu et al. 2011] Leon-Suematsu, Y. I., Inui, K., Kurohashi, S., and Kidawara, Y. (2011). Web Spam Detection by Exploring Densely Connected Subgraphs. In *2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, volume 1, pages 124–129.

[Mawhirter et al. 2021] Mawhirter, D., Reinehr, S., Holmes, C., Liu, T., and Wu, B. (2021). Graphzero: A high-performance subgraph matching system. *SIGOPS Oper. Syst. Rev.*, 55(1):21–37.

[Mawhirter and Wu 2019] Mawhirter, D. and Wu, B. (2019). Automine: Harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 509–523, New York, NY, USA. ACM.

[Meng et al. 2018] Meng, C., Mouli, S. C., Ribeiro, B., and Neville, J. (2018). Subgraph pattern neural networks for high-order graph evolution prediction.

[Pržulj et al. 2004] Pržulj, N., Corneil, D. G., and Jurisica, I. (2004). Modeling interactome: scale-free or geometric? *Bioinformatics*, 20(18):3508–3515.

[Qin et al. 2019] Qin, H., Li, R.-H., Wang, G., Qin, L., Cheng, Y., and Yuan, Y. (2019). Mining periodic cliques in temporal networks. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1130–1141.

[Ribeiro et al. 2021] Ribeiro, P., Paredes, P., Silva, M. E. P., Aparicio, D., and Silva, F. (2021). A survey on subgraph counting: Concepts, algorithms, and applications to network motifs and graphlets. *ACM Comput. Surv.*, 54(2).

[Sun et al. 2023] Sun, X., Cheng, H., Li, J., Liu, B., and Guan, J. (2023). All in one: Multi-task prompting for graph neural networks. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, page 2120–2131, New York, NY, USA. Association for Computing Machinery.

[Teixeira et al. 2015] Teixeira, C. H. C., Fonseca, A. J., Serafini, M., Siganos, G., Zaki, M. J., and Aboulnaga, A. (2015). Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15. ACM.

[Ugander et al. 2013] Ugander, J., Backstrom, L., and Kleinberg, J. (2013). Subgraph frequencies: mapping the empirical and extremal geography of large graph collections. In *WWW*.

[Wang et al. 2018] Wang, K., Zuo, Z., Thorpe, J., Nguyen, T. Q., and Xu, G. H. (2018). Rstream: Marrying relational algebra with streaming for efficient graph mining on a single machine. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 763–782, Berkeley, CA, USA. USENIX Association.

[Yan and Han 2002] Yan, X. and Han, J. (2002). gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, pages 721–, Washington, DC, USA. IEEE Computer Society.

[Yang et al. 2016] Yang, Y., Yan, D., Wu, H., Cheng, J., Zhou, S., and Lui, J. C. (2016). Diversified temporal subgraph pattern mining. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 1965–1974, New York, NY, USA. Association for Computing Machinery.

[Zaharia et al. 2012] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*.

[Zhao et al. 2019] Zhao, H., Zhou, Y., Song, Y., and Lee, D. L. (2019). Motif enhanced recommendation over heterogeneous information network. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, CIKM '19, pages 2189–2192, New York, NY, USA. ACM.