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.

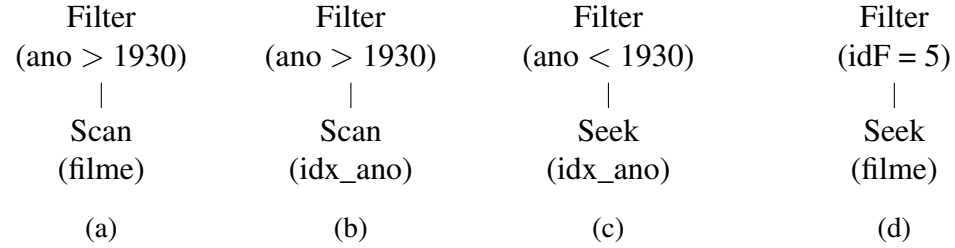


Figura 2.3: Exemplos de consultas com filtro

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.

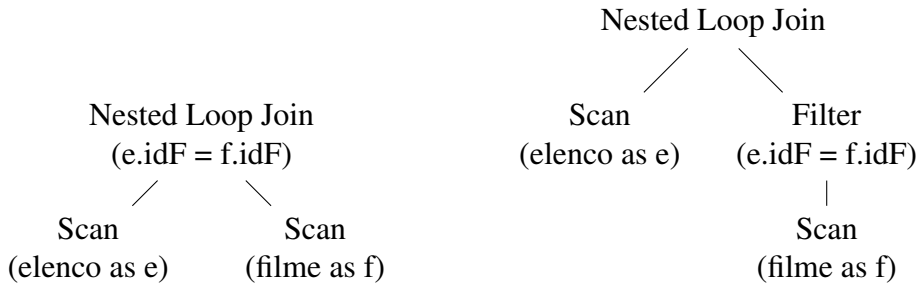
A escolha do algoritmo de junção pelo otimizador de consultas pode ter um impacto significativo no tempo de resposta. Portanto, ter um conhecimento sólido sobre os diferentes métodos de junção e seus comportamentos em diferentes cenários ajuda a escrever consultas mais eficientes e melhorar o desempenho 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

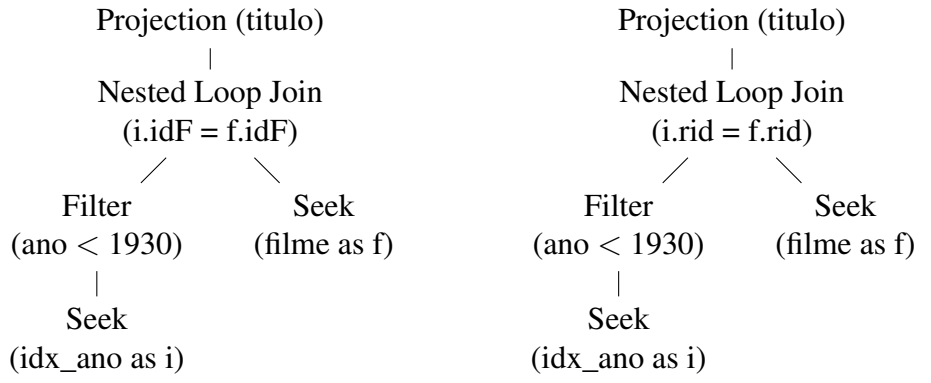
De modo geral, o *Nested Loop Join* é eficaz quando a tabela interna é muito pequena, mas pode se tornar inviável para tabelas maiores devido ao número excessivo de comparações necessárias. Uma variação mais eficiente deste algoritmo é o *Nested Loop Join* Indexado. Em vez de usar um *Scan* para percorrer 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

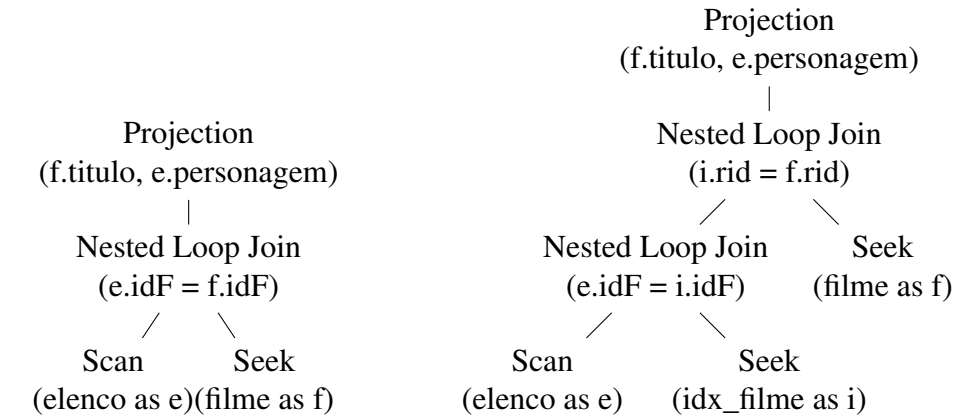
Figura 2.5: Exemplos de consultas com complementação de índice

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.

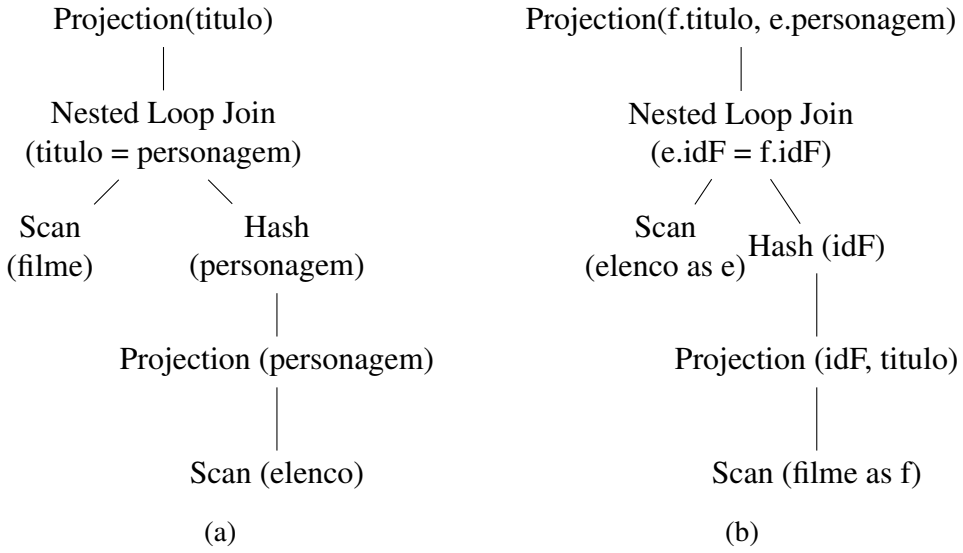


Figura 2.7: Exemplos de consultas com Hash Join

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.

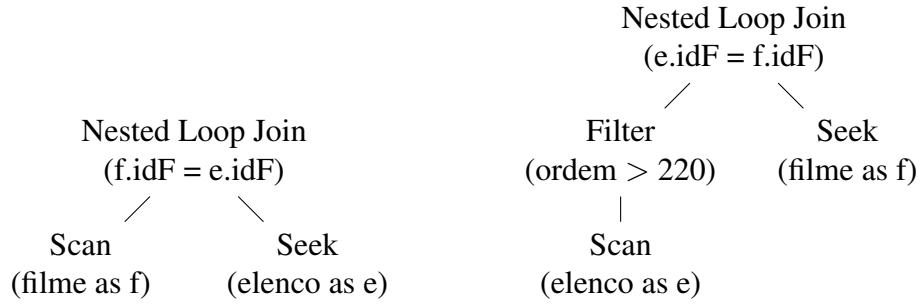
2.5.4. Custo das Junções

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

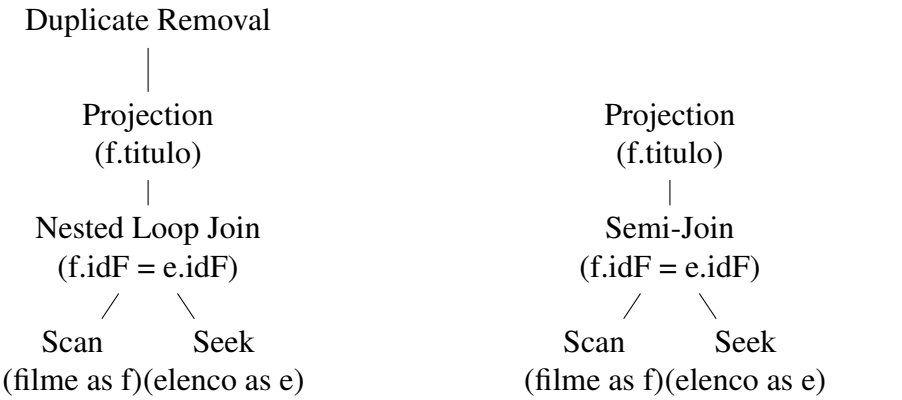
Um recurso muito utilizado em SQL para realizar consultas complexas e dinâmicas são as subconsultas. Elas são úteis para simplificar uma etapa de processamento que é exigida pela consulta principal. Uma das principais características de subconsultas é que seus resultados não podem ser usados pelas consultas 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

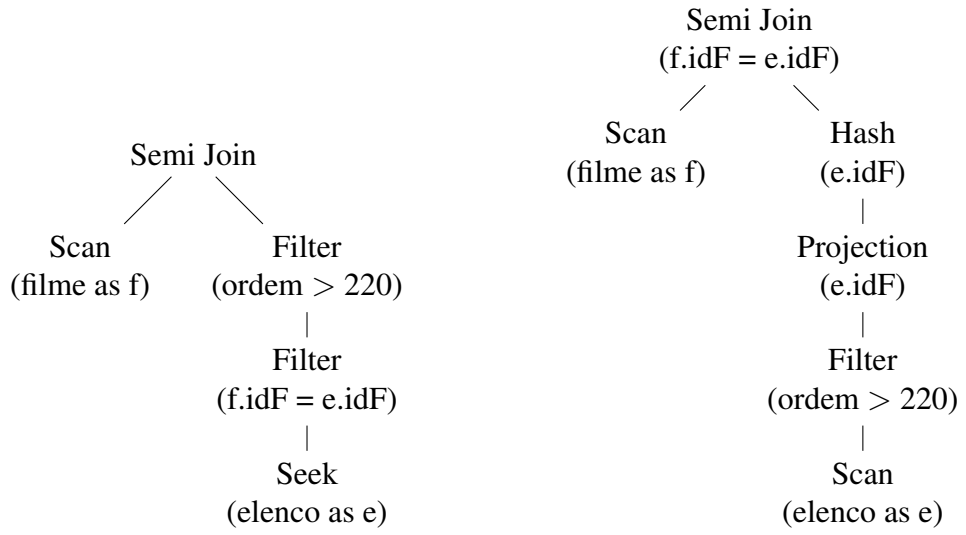
Além dessa característica principal, que estabelece que o retorno de uma consulta não pode conter dados vindos de uma subconsulta, as subconsultas também podem ser divididas em dois tipos principais:

- **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.

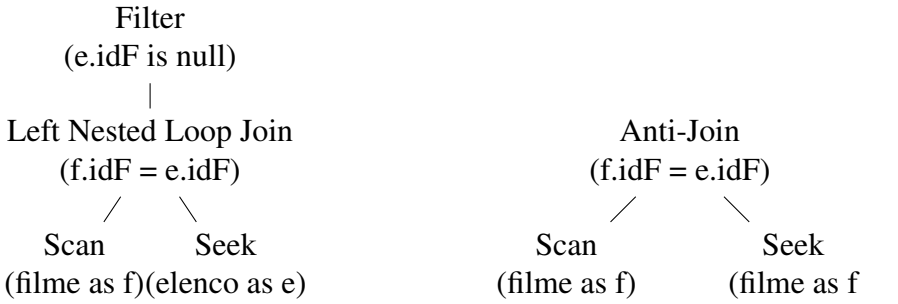
Uma das variações mais usadas é a junção externa esquerda (*left outer join*), que considera a tabela da esquerda da junção como sendo a principal. Por exemplo, uma junção externa 'filme left join elenco' retorna todos os registros da tabela à esquerda da junção (filme) e os registros correspondentes 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'.

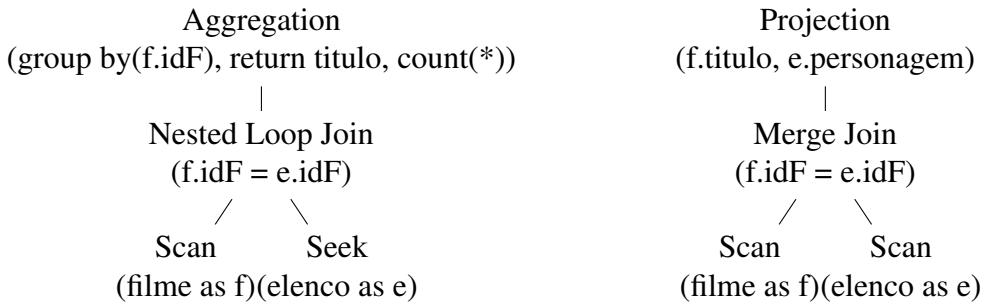
Essas soluções, apesar de reduzirem o custo da ordenação, trazem efeitos colaterais. Por exemplo, organizar os acessos ao disco demanda memória. Já o incremento de um índice aumenta o custo de manutenção desse índice. Ou seja, as soluções apresentadas acabam deixando rastros que podem afetar o desempenho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

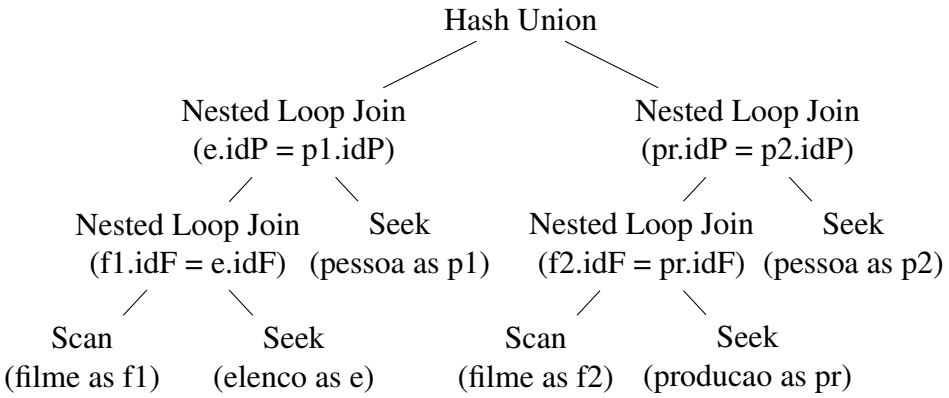


Figura 2.14: Uso de Hash em vez de ordenação para resolver uma operação de união

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.

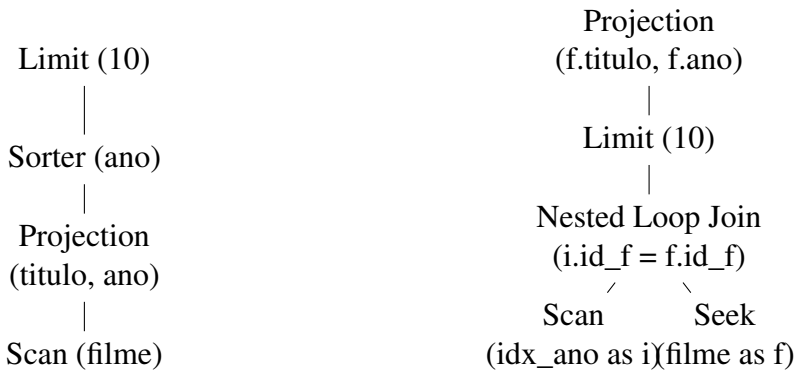
2.7. Paginação

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

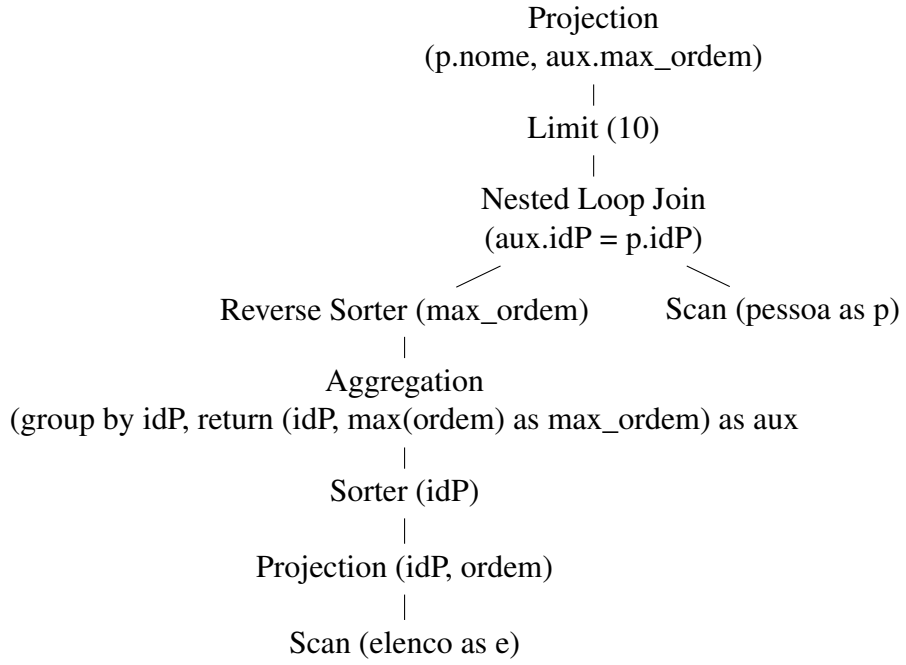


Figura 2.16: Aplicando limite sobre o resultado de uma agregação

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

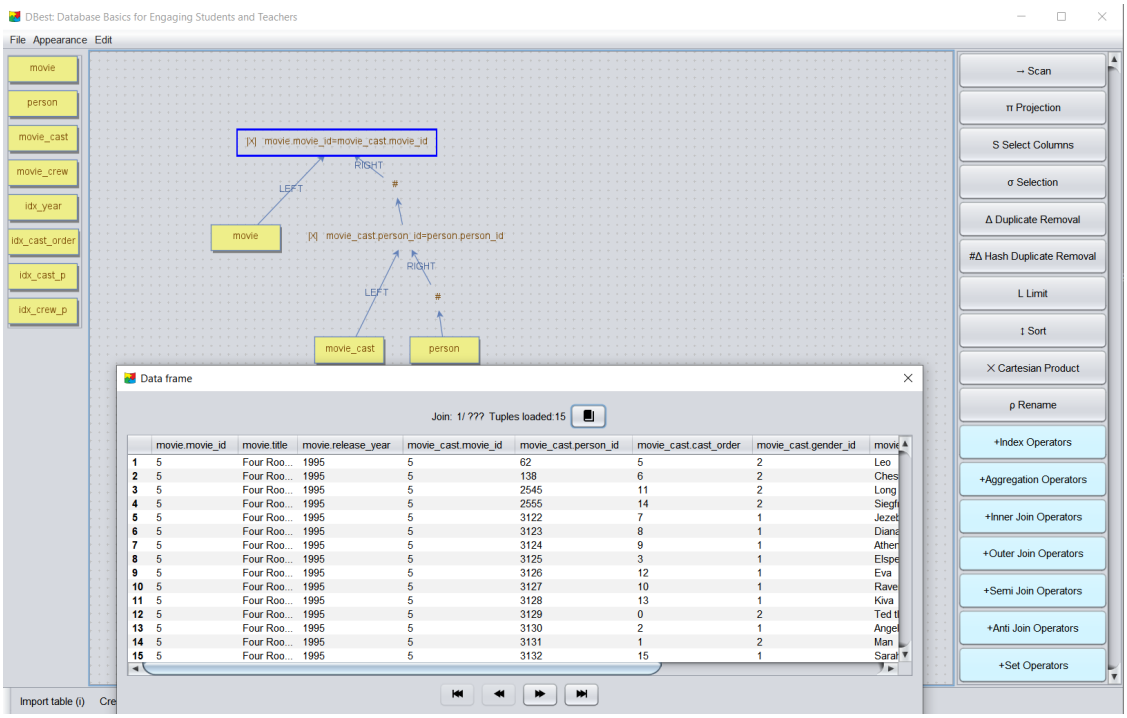


Figura 2.17: Interface gráfica para a escrita de consultas

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.