Otimização de Semi-Junções: Aplicação do Nested Loop Join e Estratégias de Execução Eficientes

Matheus D. Souza[1], **Sérgio L. S. Mergen**[1]

[1]Centro de Tecnologia – Universidade Federal de Santa Maria (UFSM)
97105-900 -– Santa Maria -– RS — Brasil

{mdsouza,mergen}@inf.ufsm.br

Abstract. *This paper explores the use of the Nested Loop Join algorithm for executing semi-joins. Three variations of the algorithm are presented: a naive approach and two optimized versions. The experiments conducted analyzed the impact of these optimizations on two representative queries. Additionally, the generated execution plans were examined, highlighting how recognizing the semantics of a semi-join can enable significant optimizations, challenging the notion that subqueries are inherently more expensive than regular joins.*

Resumo. *Este artigo explora o uso do algoritmo Nested Loop Join para a execução de semi-junções. São apresentadas três variações do algoritmo: uma abordagem ingênua e duas versões otimizadas. Os experimentos realizados analisaram o impacto dessas otimizações em duas consultas representativas. Além disso, foram examinados os planos de execução gerados, destacando como a identificação da semântica de uma semi-junção pode permitir otimizações significativas, desmistificando a ideia de que subconsultas são mais custosas do que junções regulares.*

1. Introdução

O desempenho de sistemas de gerenciamento de bancos de dados (SGBDs) é frequentemente determinado pela eficiência com que operações de consulta são realizadas. Entre as operações mais custosas, a junção de tabelas desempenha um papel crucial, impactando diretamente o tempo de execução de consultas complexas. Nesse contexto, o algoritmo clássico de junção *Nested Loop Join*, apesar de simples, se destaca como solução amplamente adotada e otimizada ao longo dos anos.

A semi-junção surge como uma variação estratégica que, em muitos casos, permite reduzir significativamente o custo de processamento. Diferentemente da junção completa, a semi-junção visa apenas identificar e retornar as tuplas de **uma** relação que têm correspondência em outra relação. Essa particularidade torna a semi-junção uma operação especialmente vantajosa em cenários onde o objetivo é verificar a existência de conexões. Em SQL, a semi-junção pode ser alcançada por meio de uma junção regular seguida de uma etapa de remoção de duplicatas (com uso do **DISTINCT**). Entretanto, a forma apropriada de representar semi-junções é por meio de subconsultas, geralmente conectadas à consulta principal por meio dos operadores **IN** ou **EXISTS**.

Neste artigo, exploramos como o *Nested Loop Join* pode ser adaptado para implementar a semi-junção, destacando as modificações necessárias em suas estratégias de processamento, que serão apresentadas na forma de árvores de execução. Além disso,

realizamos experimentos comparativos para avaliar a eficiência dessas versões otimizadas em relação ao algoritmo original. A contribuição deste trabalho está em fornecer uma análise detalhada dos algoritmos envolvidos, investigar as modificações concretas para otimização da semi-junção e oferecer *insights* baseados em experimentos práticos que podem auxiliar na compreensão do potencial de otimização de consultas baseadas em semi-junções.

O restante deste artigo está estruturado nas seguintes seções. A seção 2 apresenta os trabalhos relacionados. A seção 3 apresenta o esquema de banco de dados e as consultas usadas ao longo do artigo. A seção 4 apresenta os operadores que podem ser usados em planos de execução para resolver as consultas propostas. A seção 5 explica como o algoritmo *Nested Loop Join* pode ser adaptado para resolver semi-junções. A seção 6 apresenta os experimentos realizados. A seção 7 apresenta as considerações finais.

2. Trabalhos Relacionados

A semi-junção é um operador fundamental no contexto de consultas SQL, pois sua semântica permite otimizações que reduzem a carga computacional dos algoritmos de junção, interrompendo a execução mais cedo. Corroborando essa ideia, o estudo recente de [Kossmann et al. 2022] evidencia que, em determinados cenários, semi-junções podem ser mais eficientes do que junções completas. Da mesma forma, [Mehta et al. 2018] discute os benefícios do uso de subconsultas no contexto de semi-junções, ressaltando os casos em que essa abordagem pode trazer vantagens.

A importância da semi-junção se torna ainda mais evidente em ambientes distribuídos e virtualizados, onde a eficiência na transferência de dados é um fator crítico. Ao eliminar tuplas irrelevantes antes da junção, a semi-junção reduz significativamente o volume de dados trafegados entre nós [Shankar et al. 2012, Lawrence 2014].

A literatura apresenta diversas abordagens para otimizar o processamento de consultas contendo semi-junções. O estudo de [Elhemali et al. 2007] propõe estratégias como a inversão da ordem das tabelas interna e externa, o uso de algoritmos como *merge join* e *hash join*, a promoção da subconsulta ao mesmo nível da consulta principal e a utilização de valores mínimos e máximos como atalhos de processamento. Em [Lee et al. 2016], os autores demonstram como índices de *bitmap* podem ser empregados para remover registros sem correspondência antes da junção, permitindo que uma semi-junção seja tratada como uma junção regular. Já o trabalho de [Płodzień and Subieta 2001] foca na eliminação de subconsultas que não contribuem para o resultado final da consulta. Além dos estudos puramente acadêmicos, há também as soluções que são efetivamente utilizadas em produtos comerciais. Por exemplo, o MySQL oferece diversas técnicas para lidar com semi-junções, como *Duplicate Weedout* e *First Match* [Krogh and Krogh 2020].

Apesar de ser uma ferramenta poderosa, as subconsultas de modo geral ainda são subutilizadas, seja por desconhecimento ou por receios infundados. A pesquisa de [Poulsen et al. 2020], realizada com estudantes de tecnologia, revelou que apenas uma pequena parcela dos participantes conseguiu construir consultas sintaticamente e semanticamente válidas. Além disso, os estudos de [Cagliero et al. 2018] e [Miedema et al. 2022] demonstram que erros de semântica são frequentes, especialmente no uso de subconsultas, devido a dificuldades na compreensão desse conceito. Como consequência, o mau uso desse recurso gera resistência à sua adoção e leva à sua subvalorização.

3. Esquema do Banco de Dados e Consultas

Nesta seção, apresentamos o esquema do banco de dados e as consultas que serão utilizadas ao longo deste artigo. A Figura 1 apresenta o esquema do banco e a quantidade de registros para cada tabela (o asterisco indica participação na chave primária). O esquema modela dados de elenco de filme, indicando para cada pessoa o nome do personagem que ela desempenhou no filme e sua ordem de aparição. Existe uma relação de n-1 de elenco com filme e pessoa.

`filme (*idF, titulo, ano)`	4803 registros
`pessoa (*idP, nome)`	104842 registros
`elenco (*idF, *idP, personagem, ordem)`	106084 registros
`fkElenPess (*idP, idF)`	106084 registros

Figura 1. Esquema do banco de dados utilizado no artigo.

As tabelas são armazenadas em árvores B+, com a chave de busca sendo a chave primária. Especificamente, `fkElenPess` não é precisamente uma tabela, mas sim um índice não clusterizado criado sobre o campo `idP` da tabela elenco. Esse índice permite realizar buscas mais eficientes na tabela `elenco`, especialmente quando buscamos por `idP` (id da pessoa). Essa estrutura de dados se assemelha à abordagem adotada pelo MySQL, que também indexa chaves estrangeiras utilizando índices não clusterizados.

As consultas são apresentadas na Figura 2. Ambas as consultas realizam semi-junções, ou seja, retornam o registro do lado externo caso se verifique que ele alguma correspondência com registros do lado interno. Em ambos os casos, o lado externo possui uma relação de 1-n com o lado interno. No entanto, há diferenças importantes entre elas. Na **Consulta 1**, o filtro `ano < 1930` é aplicado diretamente à tabela `filme`, que está no lado 1 da relação 1-n. Na **Consulta 2**, o filtro `e.ordem > 220` é aplicado na tabela `elenco`, ou seja, no lado n. Além disso, na **Consulta 1**, a junção entre `filme` e `elenco` é feita pela coluna `idF`. Essa coluna é um prefixo da chave primária da tabela `elenco`. Já na **Consulta 2**, a junção utiliza a coluna `idP`, que corresponde ao segundo nível da chave primária da tabela `elenco`.

1. Consulta 1

```sql
SELECT f.titulo
FROM filme f
WHERE EXISTS (
   SELECT 1
   FROM elenco e
   WHERE f.idF = e.idF
)
AND ano < 1930;
```

2. Consulta 2

```sql
SELECT p.nome
FROM pessoa p
WHERE EXISTS (
   SELECT 1
   FROM elenco e
   WHERE p.idP = e.idP AND
      e.ordem > 220
);
```

Figura 2. Consultas SQL utilizadas no artigo.

Essas distinções mostram como as semi-junções podem ser aplicadas de maneira flexível para atender a diferentes cenários de consulta. Além disso, destacam o impacto

da posição dos filtros e das colunas utilizadas na junção sobre a eficiência e o propósito das operações.

4. Operadores Utilizados nos Planos de Execução

Em bancos de dados, consultas SQL são transformadas em planos de execução de consulta, que indicam os passos necessários para alcançar o resultado final. Neste artigo, um plano de execução é representado por meio de uma árvore composta por operadores, que podem ser unários ou binários, cujo papel é acessar tuplas de entrada e transformá-las em tuplas de saída.

Os operadores que serão utilizados no transcorrer do artigo são *Filter*, para filtragem de tuplas, *Distinct* e *Hash Distinct*, para remoção de duplicatas, *Projection*, para a filtragem de colunas, *Nested Loop Join* e *Nested Loop Semi Join*, para junção, e *Scan* e *Seek* para acesso aos dados.

Duas das principais formas de medir o custo dos operadores é a quantidade de páginas acessadas e o consumo de memória. Dos operadores apresentados, os únicos que acarretam em acesso a páginas são os operadores *Scan* e *Seek*. O *Scan* percorre todos os registros armazenados em uma estrutura de árvore B+. Por sua vez, o *Seek* navega pelos nós de uma árvore B+ para localizar tuplas que satisfaçam um predicado de seleção. Já no quesito de consumo de memória, apenas o *Hash Distinct* leva a um consumo adicional. Enquanto o *Distinct* remove duplicatas apenas de valores consecutivos (pressupondo que os dados estejam ordenados), o *Hash Distinct* precisa materializar as tuplas em uma estrutura de *hash* para localizar e remover duplicatas de maneira eficiente, mesmo com dados desordenados.

É importante destacar que o formato dos planos de execução varia de SGBD para SGBD. Tanto os planos de execução quanto os operadores definidos e usados neste artigo são apenas uma forma de representação que permite compreender as etapas envolvidas no processamento de uma consulta, independente do SGBD.

5. Algoritmos de *Nested Loop Join*

O *Nested Loop Join* é um dos algoritmos de junção amplamente utilizado em SGBDs devido à sua flexibilidade e simplicidade de implementação. Este algoritmo funciona percorrendo todas as tuplas de uma tabela (chamada de tabela externa) e, para cada uma delas, varrendo todas as tuplas da outra tabela (chamada de tabela interna) para verificar combinações que atendam à condição de junção. Normalmente, o custo do algoritmo é reduzido mantendo a menor tabela no lado externo. Nesta seção apresentaremos três variações deste algoritmo que podem ser usadas para lidar com semi-junções.

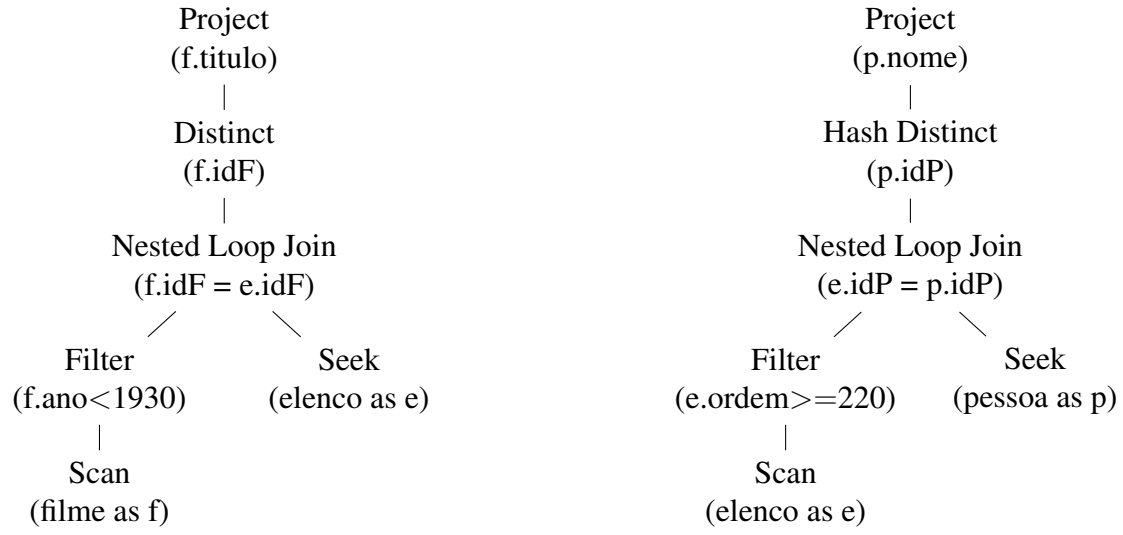
5.1. Nested Loop Join Clássico

O *Nested Loop Join* clássico, aqui chamado de **NL1**, recorre à remoção de duplicatas para garantir que cada tupla da consulta externa apareça apenas uma vez no resultado. Porém, o algoritmo não se preocupa em resolver a remoção de duplicatas antes da junção. Ou seja, a etapa de remoção ocorre somente após a junção.

A Figura 3 apresenta as árvores de execução para as consultas 1 e 2. Em ambas, a tabela com o filtro foi posicionada no lado esquerdo da junção. Essa escolha é vantajosa

quando o filtro é seletivo, pois, como já mencionado, o *Nested Loop Join* opera de forma mais eficiente quando o lado esquerdo contém poucos registros. Observe que há um *Seek* do lado direito da junção. Esse operador explora o índice em forma de árvore B+ para localizar as correspondências de forma eficiente.

Como se trata de uma semi-junção, é necessário garantir que o lado *1* apareça apenas uma vez no resultado. Na **árvore 1**, como a tabela `filme` está no lado esquerdo da junção, os registros resultantes já estão ordenados por *filme*, permitindo o uso da versão mais eficiente do *Distinct*, que elimina duplicatas de valores consecutivos. Na **árvore 2**, no entanto, é necessário utilizar a versão *Hash* do *Distinct*, que consome mais memória, pois a tabela `elenco`, posicionada no lado esquerdo, não está ordenada pelo `idP` da pessoa. Isso inviabiliza a aplicação do *Distinct* baseado em ordenação.



(a) Plano de execução da consulta 1

(b) Plano de execução da consulta 2

Figura 3. Exemplos de consultas com Nested Loop Join

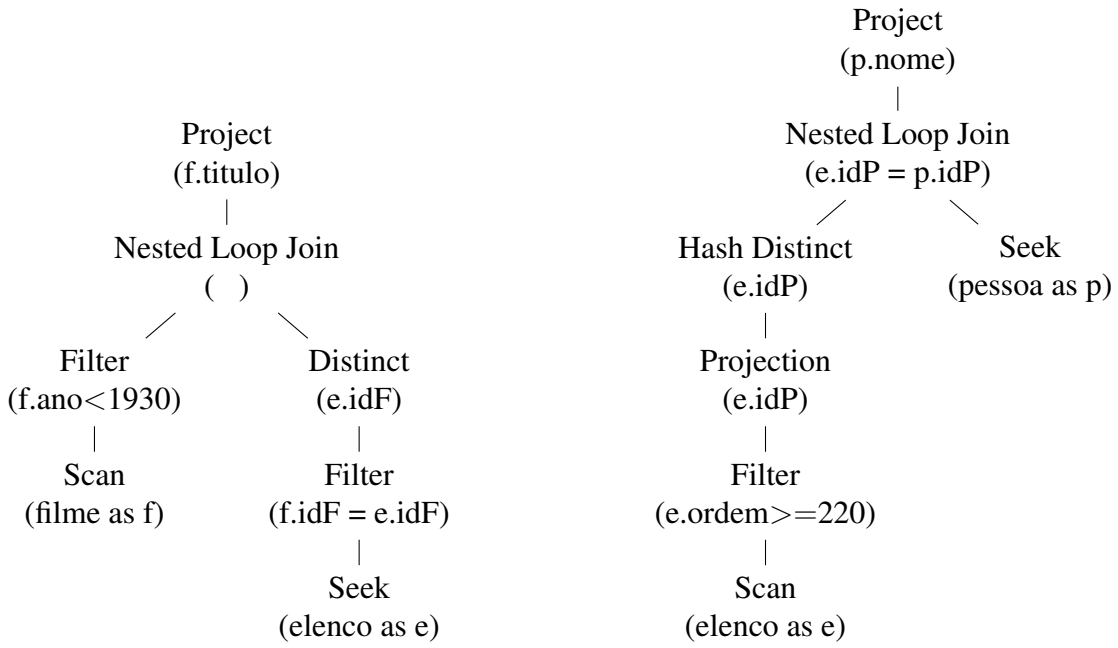
5.2. Nested Loop Join Otimizado

Na **árvore 2** da Figura 3, o custo em memória da operação *Hash Distinct* é proporcional ao tamanho das tuplas **após** a junção. Uma otimização possível no caso de uma semi-junção é antecipar a remoção de duplicatas. O algoritmo de *Nested Loop Join* otimizado, aqui chamado de **NL2**, caracteriza-se por realizar a remoção de duplicatas antes da junção, como ilustrado na Figura 4.

Na **árvore 1** da Figura 4, a movimentação do *Distinct* para o lado direito da junção exigiu a criação de uma operação de filtragem conectada ao *Seek* na tabela `elenco`. Para cada filme, o filtro recupera as participações correspondentes em `elenco`, e, em seguida, o *Distinct* mantém apenas uma dessas participações. Como o critério de junção passou a ser implementado por um filtro, o *Nested Loop Join* ficou sem critérios explícitos, comportando-se como um produto cartesiano.

Já na **árvore 2** da Figura 4, a operação *Hash Distinct* foi deslocada para o lado esquerdo da junção. Com esse deslocamento, o custo em memória é reduzido, pois as tuplas de `elenco` ainda não foram complementadas com os dados de `filme`. A projeção

de `idP` reduz ainda mais o consumo de memória, mantendo na tabela *Hash* a única coluna necessária para as etapas seguintes no plano. Além disso, a junção se torna mais eficiente em comparação com o plano da Figura 3, pois o número de tuplas processadas no lado esquerdo é menor (apenas uma por filme).



(a) Plano de execução da consulta 1 (b) Plano de execução da consulta 2

Figura 4. Exemplos de consultas com Nested Loop Join

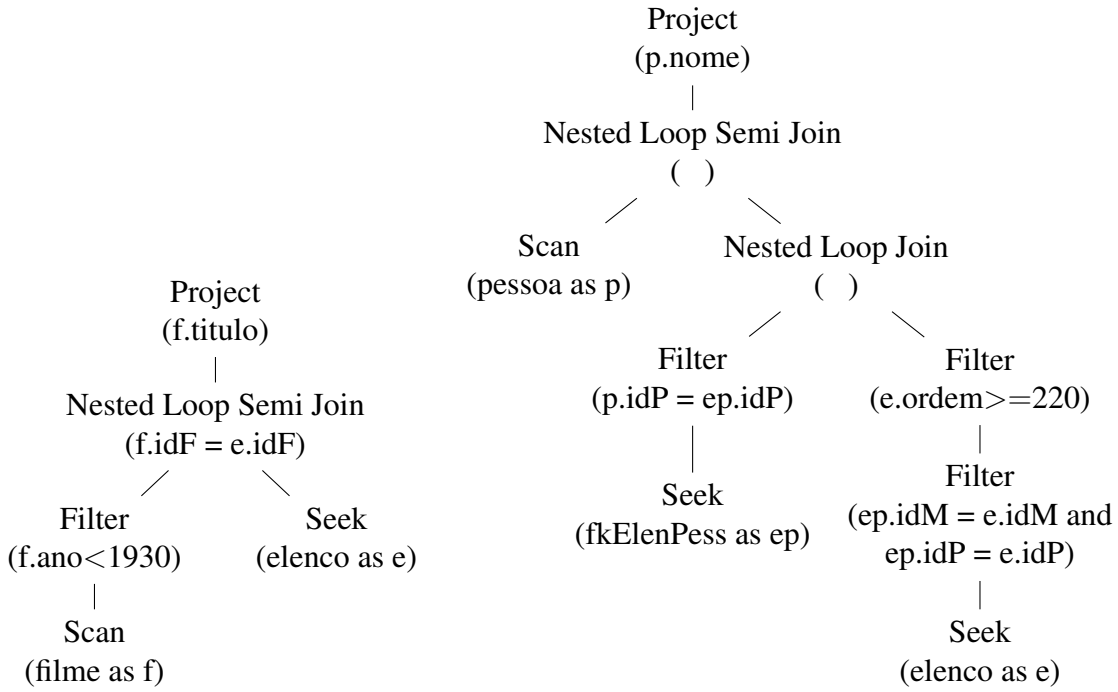
5.3. Nested Loop Semi Join

Os dois exemplos anteriores ilustraram diferentes formas de utilizar a operação *Nested Loop Join*. Esta seção demonstra o uso do algoritmo *Nested Loop Semi Join*, aqui chamado de **NL3**, projetado especificamente para semi-junções. Durante a iteração sobre as tuplas da tabela interna, o algoritmo interrompe a varredura assim que encontra uma correspondência para a tupla da tabela externa. Isso elimina a necessidade da operação de remoção de duplicatas.

A Figura 5 apresenta os planos de execução que utilizam esse algoritmo. Um aspecto importante a se destacar é que ele garante a preservação de apenas um registro do lado esquerdo da junção. Isso implica que, na **Consulta 2**, as tabelas precisam ser invertidas, posicionando `pessoa` no lado esquerdo. Dessa forma, mesmo que o filtro sobre `elenco` seja seletivo, não é possível mantê-lo do lado esquerdo da junção. Ainda, como `idP` não é prefixo da chave primária em elenco, foi necessário utilizar a estrutura `fkElenPess`, que é indexada pela coluna `idP`, e que provê acesso à coluna `idM`. O *Nested Loop Join* mais interno mostra como as entradas localizadas desse índice são complementadas com registros de elenco, para que todas as colunas de elenco se tornem acessíveis.

Outro ponto relevante é que agora existem duas operações de filtragem sobre `elenco`. A primeira corresponde ao filtro da junção, que mantém apenas as tuplas de

elenco que possuem correspondência para cada pessoa. A segunda corresponde ao filtro sobre a coluna ordem. A posição desses filtros pode ser ajustada caso se perceba que a seletividade do filtro sobre a coluna ordem é maior. Observe que a impossibilidade de manter a tabela filtrada no lado esquerdo gerou um plano mais complexo, com mais estruturas de dados envolvidas e com menos espaço para otimizações.



(a) Plano de execução da consulta 1 (b) Plano de execução da consulta 2

Figura 5. Exemplos de consultas com Nested Loop Join

6. Experimentos

Nesta seção analisaremos o desempenho dos três algoritmos apresentados, considerando as duas consultas usadas como exemplo. Para cada consulta, foram testados diversos valores na condição de filtragem, abrangendo tanto consultas pouco seletivas quanto altamente seletivas. Para fins de comparação, também destacaremos as escolhas feitas pelos bancos de dados comerciais MySQL e PostgreSQL.

O desempenho é medido pela quantidade de páginas acessadas e a quantidade de memória utilizada. A métrica de páginas acessadas refere-se ao número total de acessos realizados a uma página, independentemente de ela já estar em memória ou não. Dessa forma, elimina-se o viés que poderia ser introduzido pelo algoritmo de substituição de páginas utilizado. Já a métrica de consumo de memória corresponde à quantidade de dados materializados em memória para o *Hash Distinct*.

Para os testes foi utilizada a ferramenta DBest[1]. A ferramenta é um motor de execução de consultas que permite gerar árvores B+ para as tabelas/índices, criar os planos de execução usando os operadores descritos no artigo, executar os planos e analisar os indicadores de desempenho resultantes.

[1]https://github.com/mergen-sergio/DBest/blob/main/README.md

A Figura 6 apresenta a quantidade de páginas acessadas para a execução da **Consulta 1**. No eixo x, é representado o valor utilizado como critério de filtragem (`ano < valor`). Valores maiores tornam o filtro menos seletivo, resultando em um conjunto de resultados mais amplo. Nesta consulta, os algoritmos **NL1** e **NL2** exibem o mesmo comportamento em termos de páginas acessadas. Em contrapartida, o algoritmo **NL3** requer um menor número de acessos, especialmente à medida que a consulta se torna menos seletiva. Esse comportamento ocorre porque, para filtros pouco seletivos, os algoritmos **NL1** e **NL2** precisam acessar filmes que possuem um grande número de membros no elenco, aumentando a quantidade de páginas necessárias para recuperar todas as associações. Já o **NL3** otimiza esse processo ao interromper a busca assim que o primeiro membro do elenco correspondente é encontrado, reduzindo significativamente o número total de acessos às páginas.

Cabe salientar que nenhum dos três algoritmos precisou recorrer a dados materializados para a remoção de duplicatas de filme. Isso ocorre porque a tabela filme orientou o processo de iteração, ficando à esquerda da junção, o que assegura que as tuplas geradas pela junção estejam ordenadas por filme. Isso permitiu o uso de um algoritmo de remoção mais eficiente.

Quanto às soluções usadas em produtos comerciais, tanto MySQL quanto o PostgreSQL usam o algoritmo de semi-junção para resolver a consulta, criando um plano praticamente idêntico ao apresentado ao **NL3**. Isso mostra que essa realmente é uma escolha promissora para o caso apresentado.

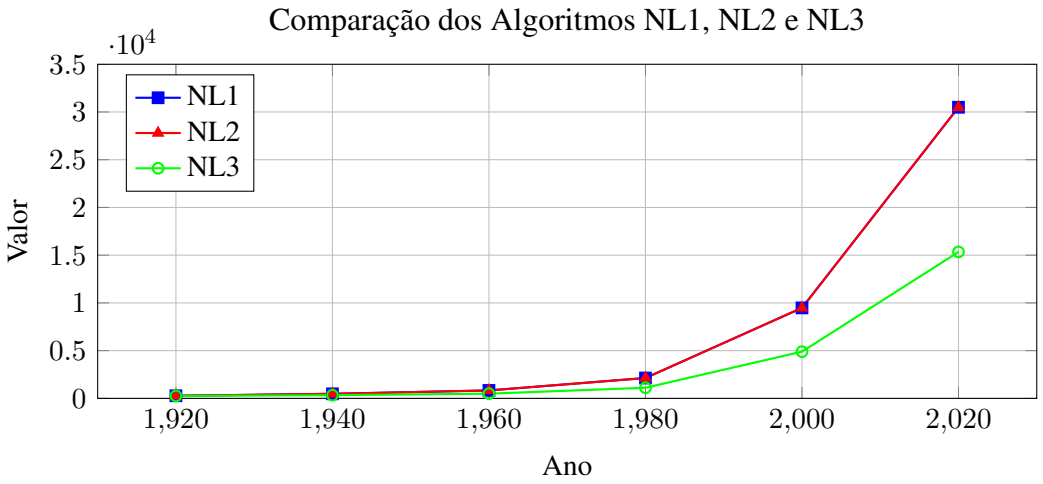


Figura 6. Uso dos Algoritmos NL1, NL2 e NL3 para execução da Consulta 1

A Figura 7 apresenta o desempenho dos três algoritmos para a **Consulta 2**. Em relação à quantidade de páginas lidas (gráfico da esquerda), o algoritmo de semi-junção (**NL3**) mostrou-se inadequado, pois não permitiu que a tabela contendo o critério de filtragem fosse posicionada no lado esquerdo da junção. Por outro lado, os algoritmos **NL1** e **NL2** conseguiram manter a tabela filtrada (`elenco`) no lado externo da junção. Entre eles, o **NL2** demonstrou melhor desempenho devido ao menor número de tuplas processadas no lado externo da junção.

No que diz respeito ao consumo de memória (gráfico do lado direito), o **NL2**

também se mostrou superior ao **NL1**. Embora ambos utilizem *Hash Distinct* para a remoção de duplicatas, o **NL2** lida com tuplas menores, pois realiza a remoção antes da junção. Nesse cenário, o **NL3**, por se basear em um algoritmo de semi-junção, apresentou consumo de memória zerado.

Tanto MySQL quanto o PostgreSQL usaram um plano de execução praticamente idêntico ao **NL2**. Ou seja, ambos descartaram o uso do algoritmo de semi-junção. Por outro lado, cabe salientar que é importante que o SGBD saiba que a operação solicitada na consulta é uma semi-junção, pois isso dá subsídios para que seja feita a escolha mais acertada. Por exemplo, caso a consulta fosse expressa sem subconsultas, recorrendo ao **DISTINCT** para manter apenas um registro de pessoa por cruzamento, tanto o MySQL quanto o PostgreSQL seguiriam o caminho ineficiente descrito por **NL1**.

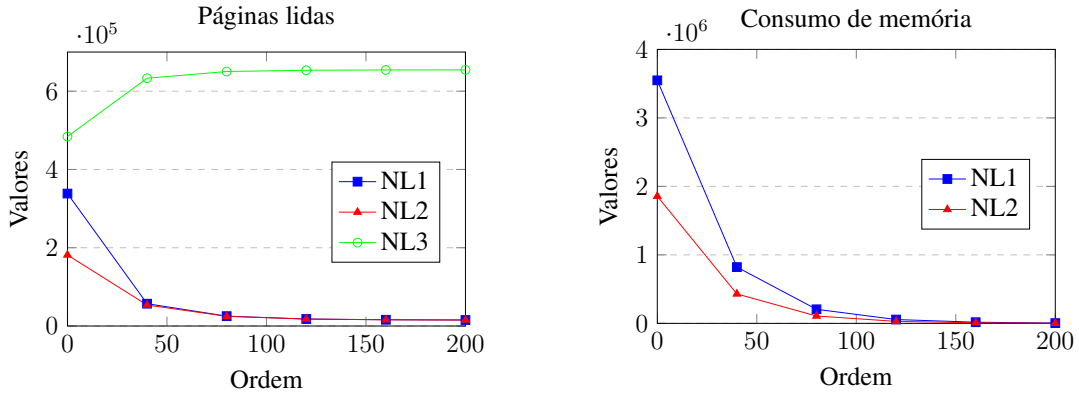


Figura 7. Uso dos Algoritmos NL1, NL2 e NL3 para execução da Consulta 2

7. Considerações Finais

Este artigo apresentou como o algoritmo *Nested Loop Join* pode ser utilizado para resolver semi-junções. Foram exploradas três variações do algoritmo: uma abordagem ingênua e duas versões otimizadas, que incorporam melhorias no processamento. Os experimentos realizados permitiram avaliar o desempenho dessas variações em duas consultas representativas. Os resultados mostraram que a estratégia mais eficiente foi a mesma adotada pelos bancos de dados comerciais MySQL e PostgreSQL, reforçando a relevância das otimizações discutidas.

Além disso, foram analisados os planos de execução gerados para essas consultas, destacando como a detecção da semântica de uma semi-junção pode abrir possibilidades de otimização. Essa análise contribui para desmistificar a ideia de que subconsultas são inerentemente mais custosas do que junções regulares—ainda que essa preocupação possa ter sido válida no passado, os mecanismos modernos de execução já incorporam estratégias eficientes para lidar com esses cenários. As otimizações baseadas em *Nested Loop Join* apresentadas neste artigo são um exemplo disso.

Bancos de dados comerciais também empregam outras estratégias avançadas. Por exemplo, alguns otimizadores podem reescrever subconsultas como junções convencionais quando o lado n está na parte externa da consulta. Além disso, técnicas baseadas em *Hash Join* permitem manter a menor tabela no lado externo da junção, mesmo que apenas os registros da tabela maior sejam desejados. O impacto dessas variações é um

tema para trabalhos futuros, aprofundando ainda mais a compreensão sobre a importância das semi-junções na construção de planos de execução eficientes.

Referências

Cagliero, L., De Russis, L., Farinetti, L., and Montanaro, T. (2018). Improving the effectiveness of sql learning practice: A data-driven approach. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 01, pages 980–989.

Elhemali, M., Galindo-Legaria, C. A., Grabs, T., and Joshi, M. M. (2007). Execution strategies for sql subqueries. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 993–1004.

Kossmann, J., Papenbrock, T., and Naumann, F. (2022). Data dependencies for query optimization: a survey. *The VLDB Journal*, 31(1):1–22.

Krogh, J. W. and Krogh, J. W. (2020). The query optimizer. *MySQL 8 Query Performance Tuning: A Systematic Method for Improving Execution Speeds*, pages 417–485.

Lawrence, R. (2014). Integration and virtualization of relational sql and nosql systems including mysql and mongodb. In *2014 International Conference on Computational Science and Computational Intelligence*, volume 1, pages 285–290. IEEE.

Lee, K., König, A. C., Narasayya, V., Ding, B., Chaudhuri, S., Ellwein, B., Eksarevskiy, A., Kohli, M., Wyant, J., Prakash, P., et al. (2016). Operator and query progress estimation in microsoft sql server live query statistics. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1753–1764.

Mehta, S., Kaur, P., Lodhi, P., and Mishra, O. (2018). Empirical evidence of heuristic and cost based query optimizations in relational databases. In *2018 Eleventh International Conference on Contemporary Computing (IC3)*, pages 1–3.

Miedema, D., Fletcher, G., and Aivaloglou, E. (2022). Expert perspectives on student errors in sql. *ACM Trans. Comput. Educ.*, 23(1).

Płodzień, J. and Subieta, K. (2001). Query optimization through removing dead subqueries. In *East European Conference on Advances in Databases and Information Systems*, pages 27–40. Springer.

Poulsen, S., Butler, L., Alawini, A., and Herman, G. L. (2020). Insights from student solutions to sql homework problems. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '20, page 404–410, New York, NY, USA. Association for Computing Machinery.

Shankar, S., Nehme, R., Aguilar-Saborit, J., Chung, A., Elhemali, M., Halverson, A., Robinson, E., Subramanian, M. S., DeWitt, D., and Galindo-Legaria, C. (2012). Query optimization in microsoft sql server pdw. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 767–776.