

**INSTITUTO FEDERAL DE MINAS GERAIS  
CAMPUS SÃO JOÃO EVANGELISTA  
SISTEMAS DE INFORMAÇÃO**

CAMILA PARANHOS FERNANDES  
FILIPE RUBSON DE ALMEIDA SANTOS  
PATRÍCIA DA SILVA COSTA  
VALDIR DE SOUZA CARVALHO NETO

**BANCO DE DADOS II - SI 241  
PROJETO FINAL  
PARTE 3 - ARTEFATOS A & B**

São João Evangelista  
2025

## → Relatório Estatístico

Foram executados os seguintes comandos abaixo:

- **Contagem:** `SELECT count(*) FROM nome_da_tabela;`
- **Tamanho em Disco (Postgres):** `SELECT pg_size.pretty(pg_total_relation_size ('escola_idiomas.nome_da_tabela'));`

Obs: *nome\_da\_tabela* foi substituído pelo nome de cada tabela presente no banco para a geração dos dados abaixo.

Tabela	Contagem (MySQL)	Tamanho em Disco (PostgreSQL)
Alunos	2035	184 kB
Avaliações	2004	216 kB
Cursos	10	24 kB
Formas de Pagamento	5	24 kB
Idiomas	5	40 kB
Matrículas	2039	216 kB
Mensalidades	16280	1368 kB
Pessoas	2040	568 kB
Professores	5	24 kB
Salas	7	24 kB
Turmas	21	24 kB

## 1. Consulta 1: Desempenho dos Professores

### 1.1. Comparação: Original vs. Variação 1 (Filtro por ID)

#### Comparação de Custo e Tempo:

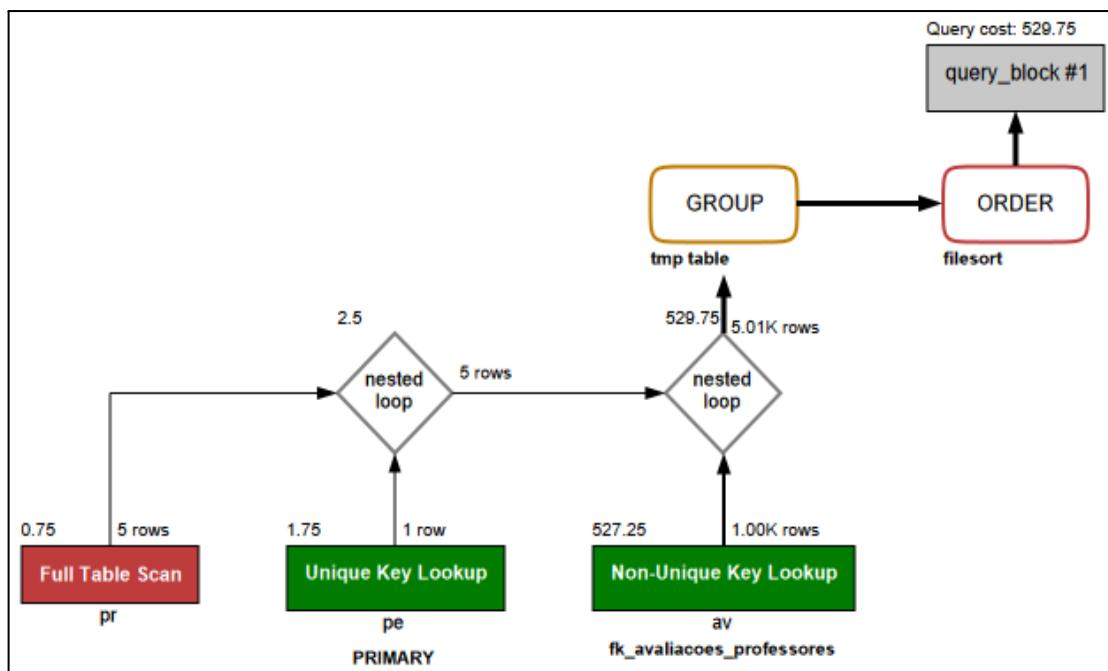
Ao executar a consulta no **PostgreSQL**, o custo estimado original foi de **272.31** e o tempo de execução foi de **105 ms**. Na Variação 1.A (com filtro por chave primária), o custo caiu drasticamente para **91.58**, representando uma melhoria de aproximadamente **66%**. Já no **MySQL**, o 'Query Cost' original foi de **529.75**, passando para **202.15** na versão otimizada.

#### Análise Teórica e Uso de Índices:

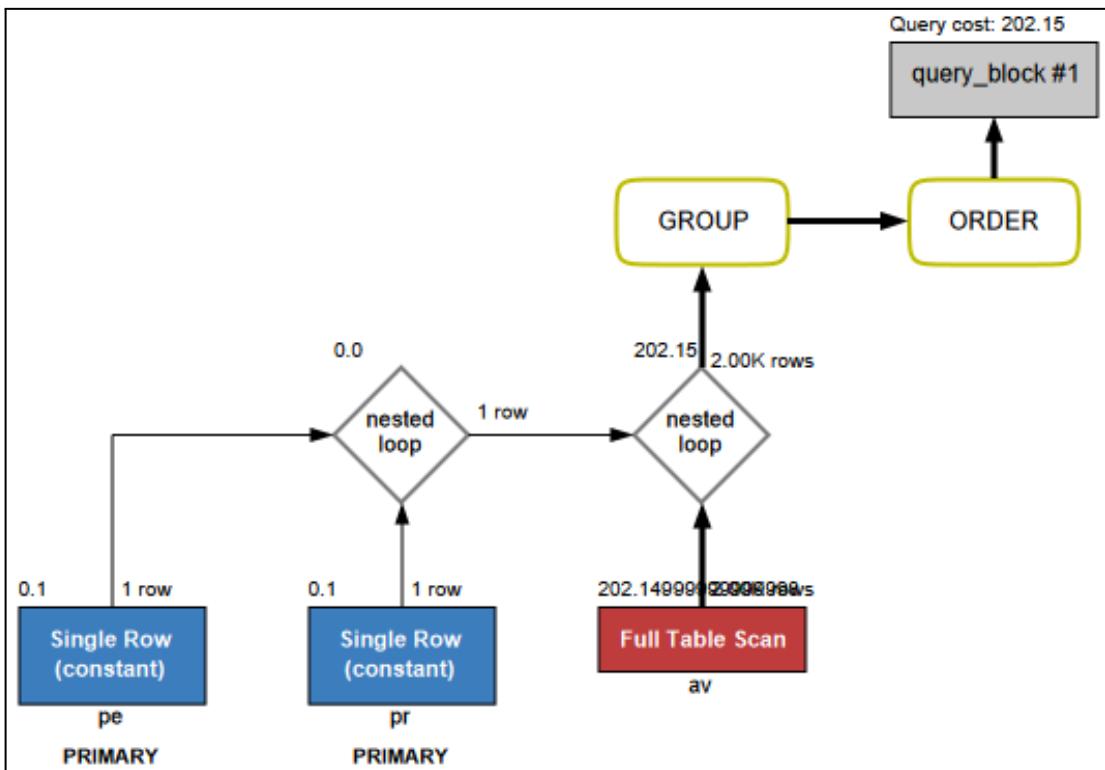
A redução de custo observada deve-se à mudança no Caminho de Acesso (Access Path). Enquanto a consulta original realiza um Seq Scan (leitura sequencial) em toda a tabela de professores para o agrupamento, a Variação 1 utiliza um Index Scan na chave primária (professor\_id). Segundo Elmasri e Navathe (2011), a busca em uma estrutura de índice baseada em árvore (B-Tree) reduz a complexidade da operação de linear  $O(n)$  para logarítmica  $O(\log n)$ . O otimizador identificou que buscar um único registro indexado é imensamente mais barato do que varrer a tabela inteira.

#### Evidência Visual:

Consulta Original:



Variação 1:



## 1.2. Comparação: Original vs. Variação 2 (Remoção de Ordenação)

### Comparação de Custo:

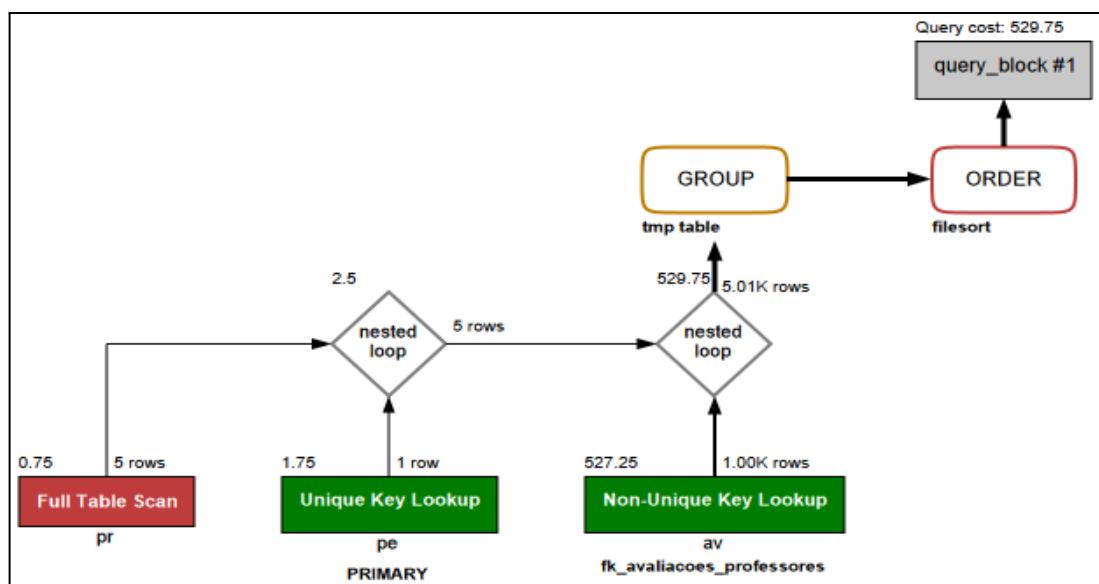
No MySQL, observamos que a remoção da cláusula ORDER BY reduziu o custo de **529.75** para **202.15**. No PostgreSQL, o tempo de execução baixou de **6.966 ms** para **0.956 ms**.

### Análise Teórica:

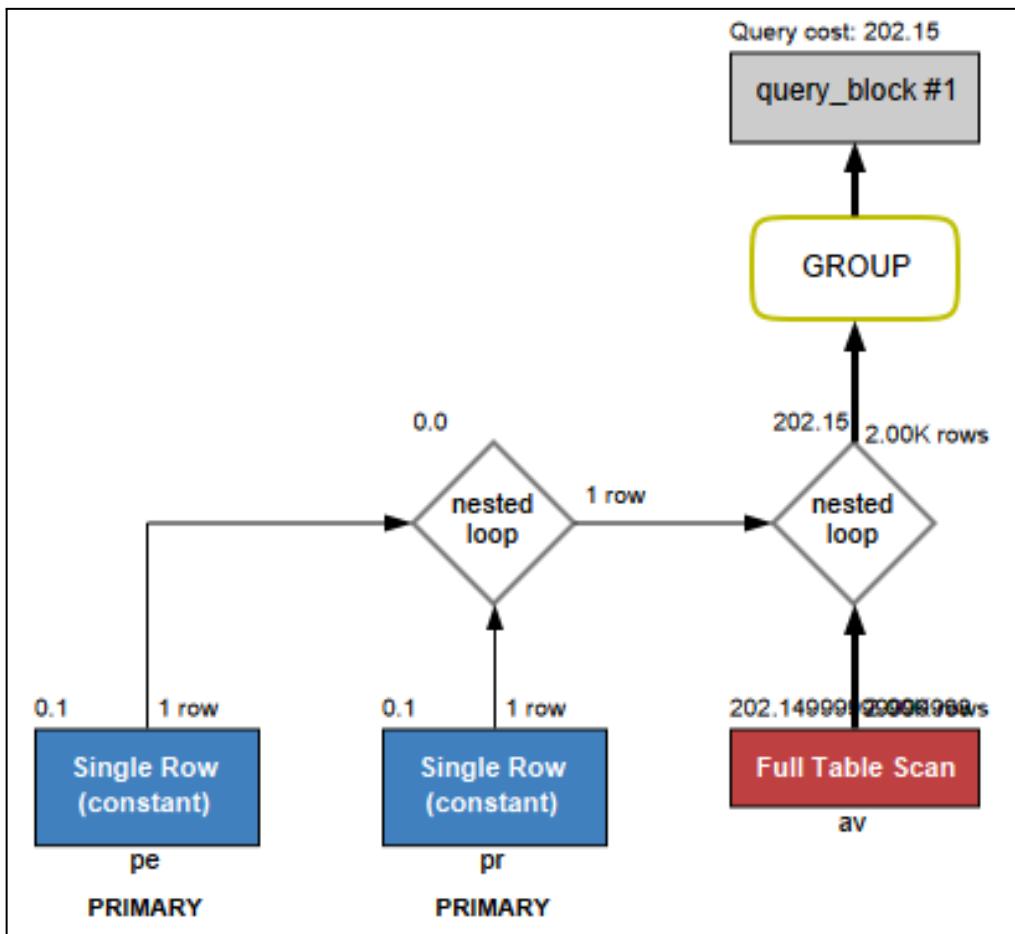
A variação evidenciou o custo computacional das operações de classificação (*Sorting*). Em bancos de dados relacionais, a ordenação exige algoritmos de *External Merge Sort* quando os dados excedem o buffer de memória. Ao remover essa etapa, eliminamos a operação de Sort Key (Postgres) ou filesort (MySQL) do plano, economizando ciclos de CPU.

### Evidência Visual:

Consulta Original:



Variação 2:



## 2. Consulta 2: Relatório de Inadimplência

### 2.1. Comparação: Original vs. Variação 1 (Filtro de Data)

#### Comparação de Custo e Índices:

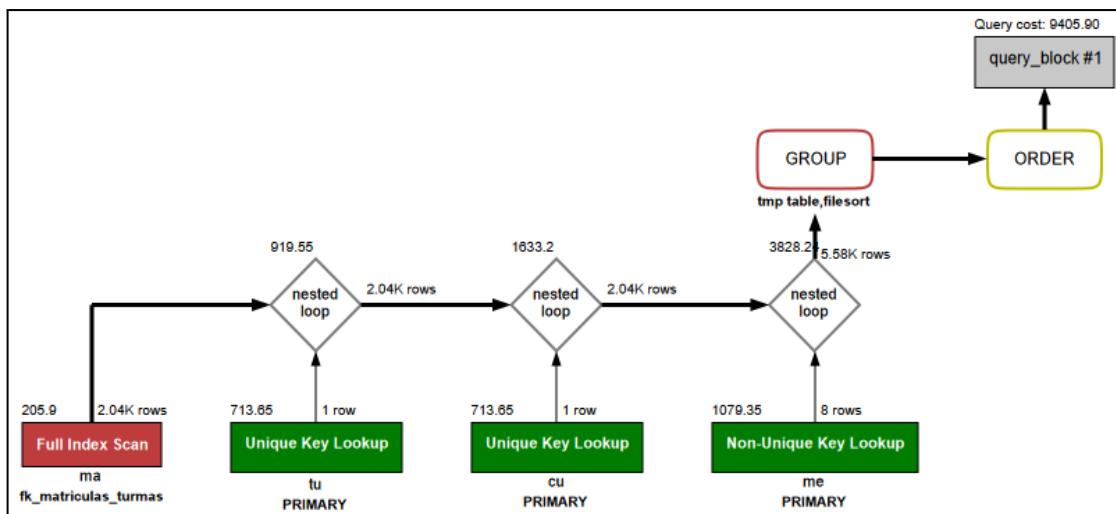
No PostgreSQL, a consulta original teve custo **448.98**. Na Variação 2.A (com filtro BETWEEN), observamos um fenômeno interessante: o custo subiu ligeiramente para **449.51**. Isso ocorre porque, sem um índice específico na coluna data\_vencimento, o SGBD precisou realizar a mesma leitura completa da tabela (Seq Scan) da consulta original, adicionando a sobrecarga de CPU para verificar a condição de data linha a linha. Isso comprova a teoria de que filtros (WHERE) só reduzem o custo de I/O se houver estruturas de acesso (índices) compatíveis; caso contrário, tornam-se apenas mais uma operação de processamento.

#### Diferença entre SGBDs:

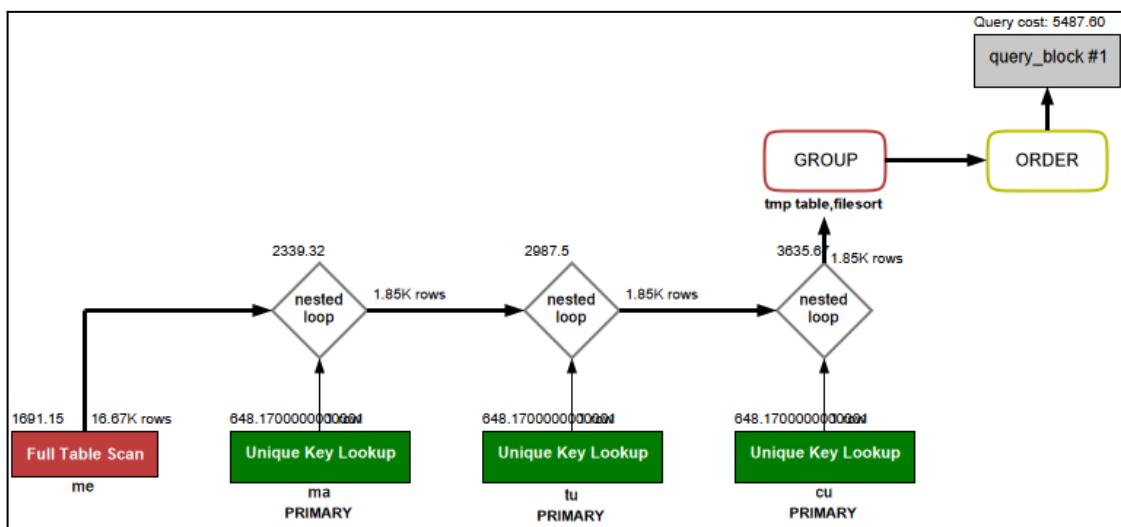
Curiosamente, para esta consulta filtrada, o PostgreSQL preferiu manter um **Hash Join** entre mensalidades e matriculas, enquanto o MySQL alterou sua estratégia para **Nested Loop**, indicando que, com menos linhas retornadas pelo filtro de data, o custo de laços aninhados tornou-se aceitável para o otimizador da Oracle.

#### Evidência Visual:

Consulta Original:



Variação 1:



## 2.2. Comparação: Original vs. Variação 2 (Eliminação de Join)

### Comparação de Custo:

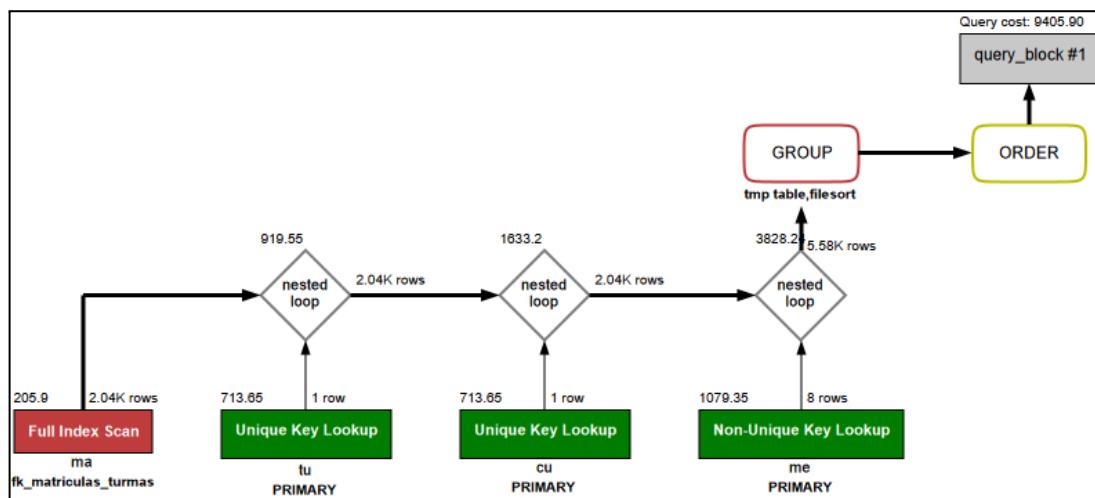
Esta foi a otimização mais impactante. O custo no PostgreSQL despencou de **448.98** (com 4 tabelas) para apenas **323.68** (tabela única). No MySQL, o custo foi de **9405.90** para **1691.15**.

### Análise Teórica:

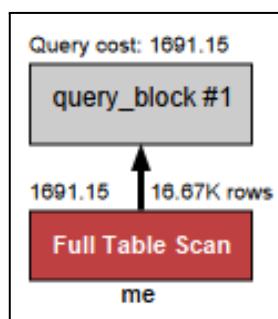
A Variação 2.B valida a teoria de processamento de junções (*Join Processing*). Ao removermos as tabelas matriculas, turmas e cursos, eliminamos a necessidade de algoritmos de correspondência de tuplas e reduzimos o espaço de busca do otimizador. Isso demonstra o *trade-off* clássico citado por Date (2004): a normalização garante integridade, mas consultas desnormalizadas (tabela única) oferecem desempenho superior para agregações simples.

### Evidência Visual:

Consulta Original:



Variação 2:



### 3. Consulta 3: Lotação de Turmas

#### 3.1. Comparação: Original vs. Variação 1 (WHERE vs HAVING)

##### Comparação de Custo:

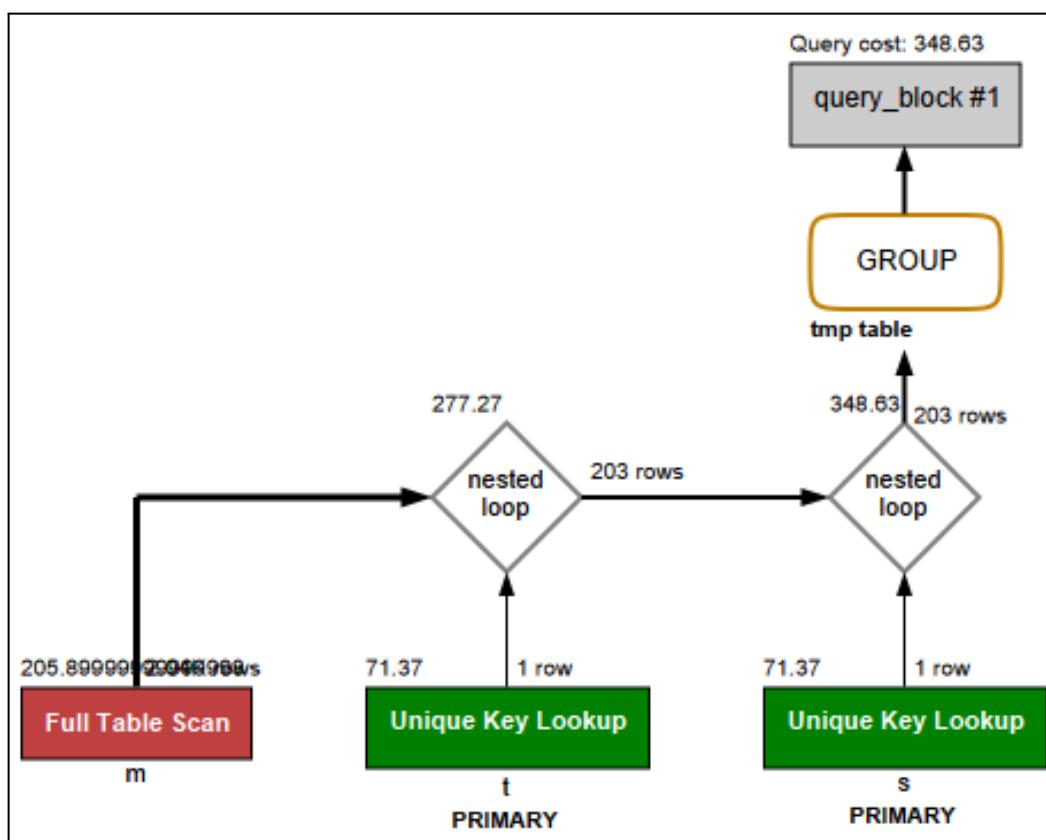
Na consulta original, o custo do PostgreSQL foi **132.86**. Ao aplicarmos a técnica de 'Selection Pushdown' (movendo o filtro para o WHERE), o custo caiu para **60.38**.

##### Análise Teórica:

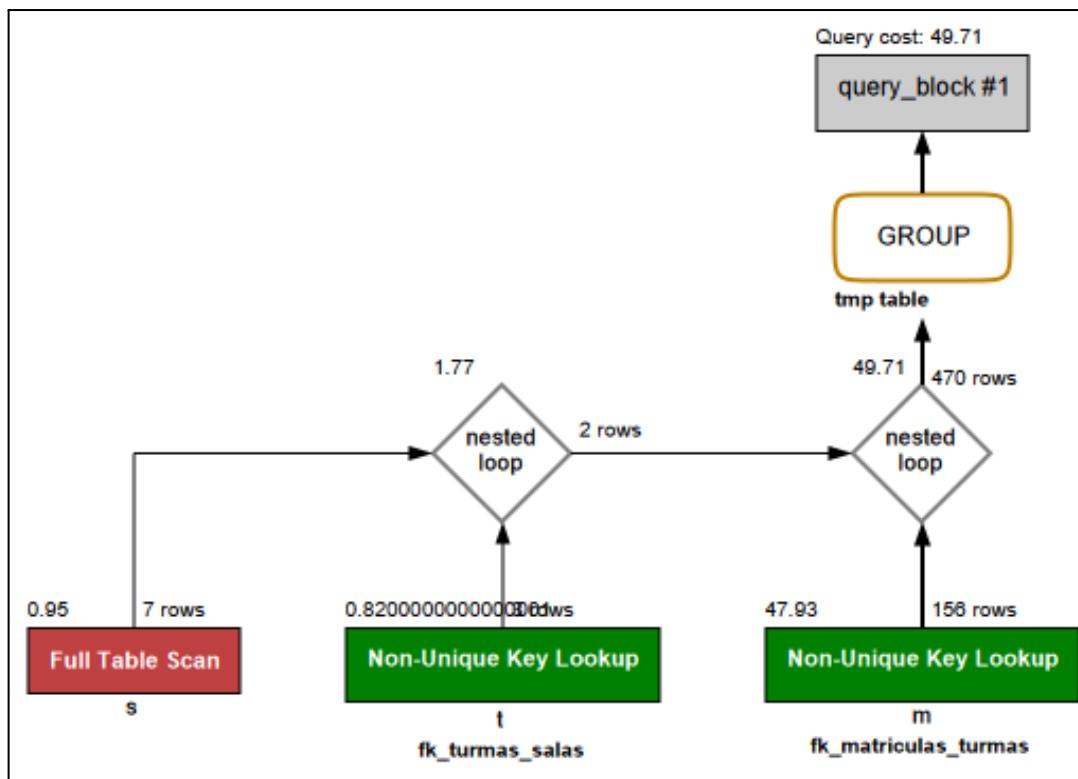
A comparação demonstra a eficiência da **Seleção Antecipada** na Álgebra Relacional. O plano de execução mostrou que, na Variação 1, o filtro tipo\_sala = 'Laboratório' foi aplicado **antes** da junção e do agrupamento. Isso reduziu a cardinalidade das tabelas intermediárias, fazendo com que a operação pesada de GROUP BY tivesse que processar apenas **5** turmas em vez de **50**, economizando memória e CPU.

##### Evidência Visual:

Consulta Original:



Variação 1:



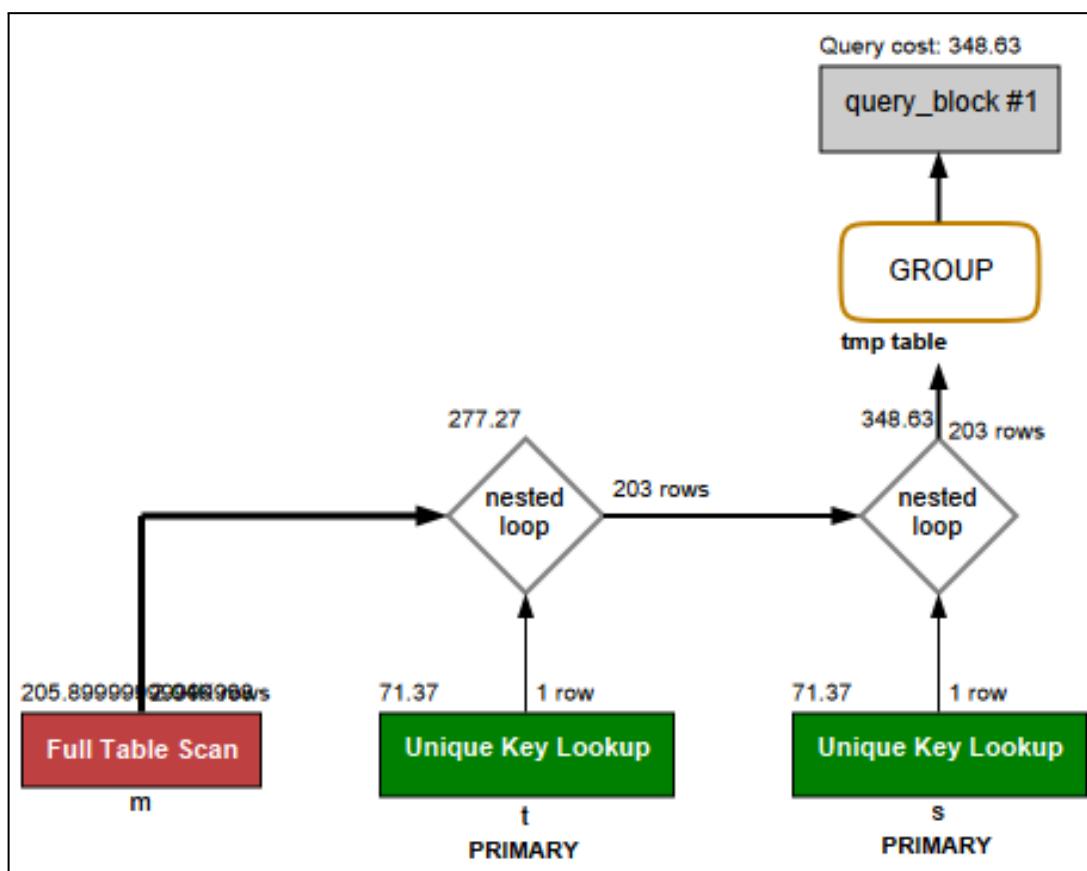
### 3.2. Análise Teórica (Variação 2 - Remoção de Colunas)

#### Análise de Desempenho (Rede/IO):

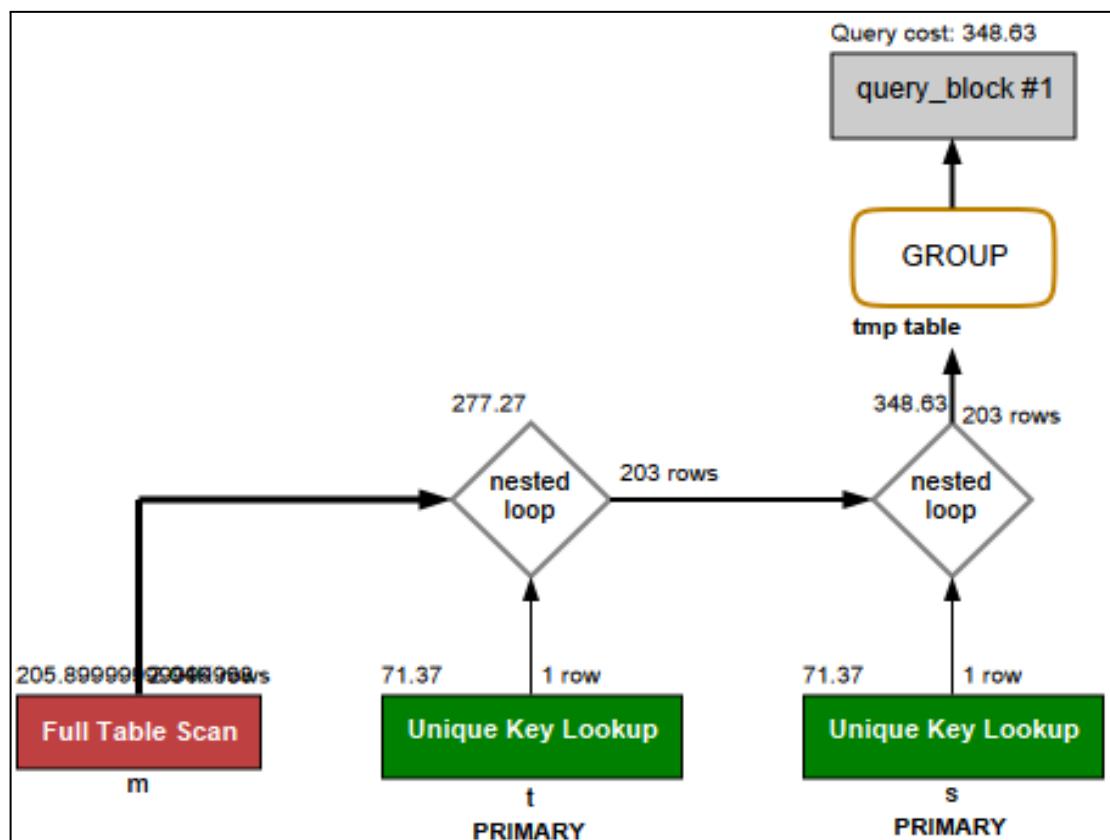
Embora o custo computacional no EXPLAIN tenha permanecido similar no PostgreSQL (**132.86** vs **127.76**), a análise teórica de Silberschatz, Korth e Sudarshan (2012) sobre I/O se aplica aqui. Ao removermos colunas textuais do SELECT, reduzimos o tamanho da tupla (*Tuple Size*). Em um cenário de carga massiva, isso resultaria em menor tráfego de rede e menor uso de *buffer* de saída, embora o plano de acesso aos dados (Joins) permaneça inalterado.

#### Evidência Visual:

Consulta Original:



Variação 2:



## 4. Consulta 4: Alunos Críticos

### 4.1. Comparação: Original vs. Variação 1 (IN vs EXISTS)

#### Comparação de Custo:

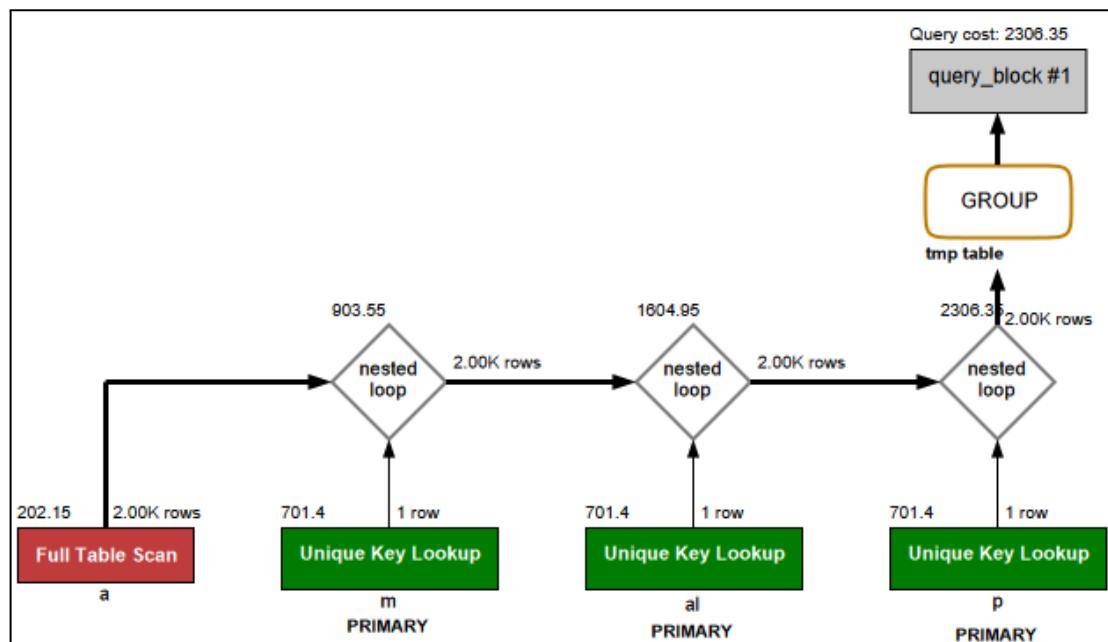
No PostgreSQL, a consulta original usando IN teve custo **668.68**. A Variação 4.A (com EXISTS) baixou o custo para **470.89**, pois evitou a materialização completa da subconsulta. Em contrapartida, no MySQL, houve uma divergência na estimativa do otimizador: o custo original de **2306.35** subiu para **4769.13** na variação.

#### Análise Teórica:

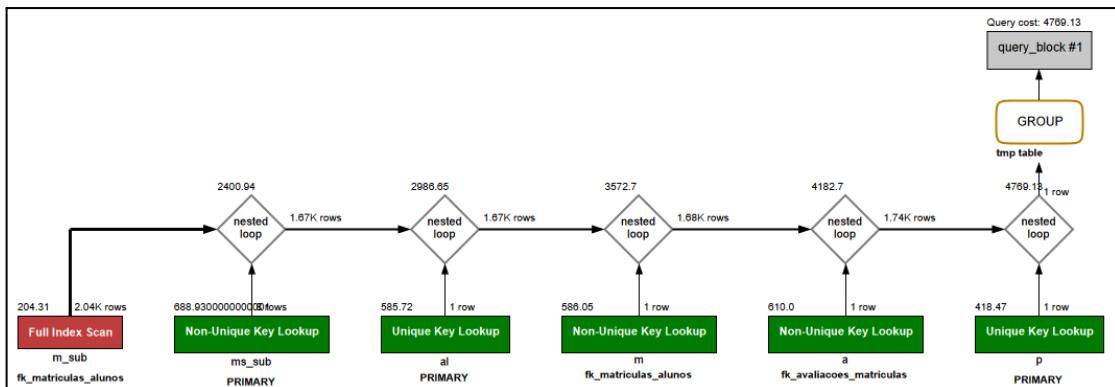
A substituição do operador IN pelo EXISTS explora o conceito de *Semi-Join* e a avaliação de curto-circuito (*Short-Circuit Evaluation*). Teoricamente, o SGBD deveria interromper a busca assim que encontrasse o primeiro registro correspondente. No entanto, a elevação do custo no MySQL ocorre porque o otimizador classificou a operação como uma *Dependent Subquery*. Diferente do PostgreSQL, o otimizador do MySQL estimou que o custo de executar a subconsulta repetidamente (para cada linha da tabela externa) seria superior à varredura da lista do IN, resultando em um custo estimado maior, embora a execução real possa ser beneficiada pelo curto-circuito.

#### Evidência Visual:

##### Consulta Original:



Variação 1:



## 4.2. Comparação: Original vs. Variação 2 (Aumentar Retorno)

### Comparação de Custo e Linhas:

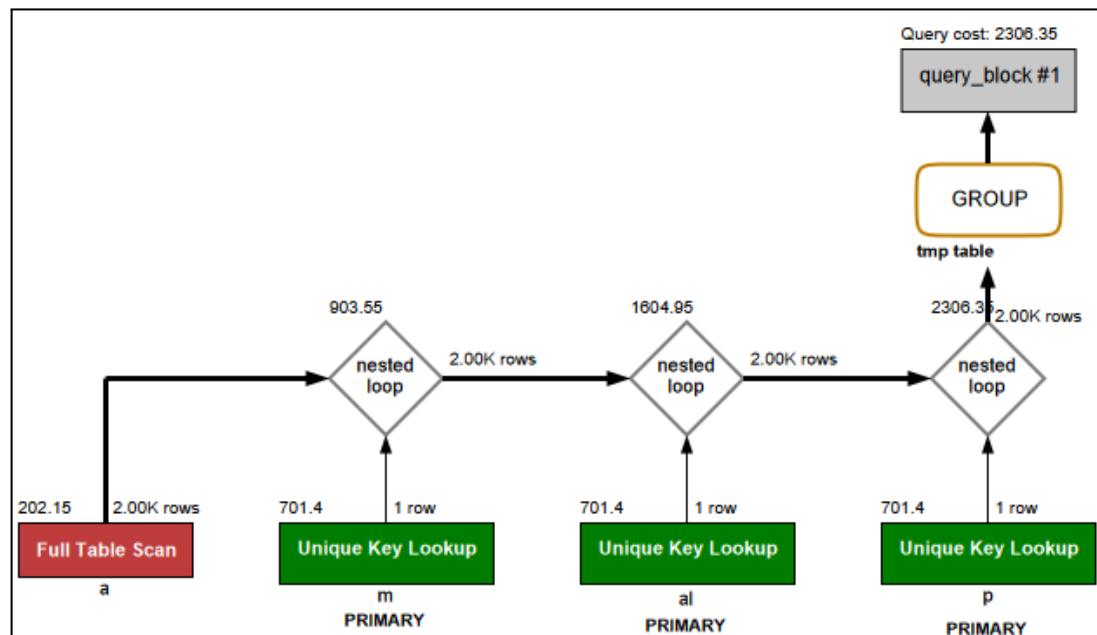
Ao remover o filtro HAVING avg < 7.0, o número de linhas retornadas subiu, mas o custo estimado de processamento permaneceu idêntico ao da variação anterior em ambos os SGBDs: manteve-se em **470.89** no PostgreSQL e em **4769.13** no MySQL.

### Análise Teórica:

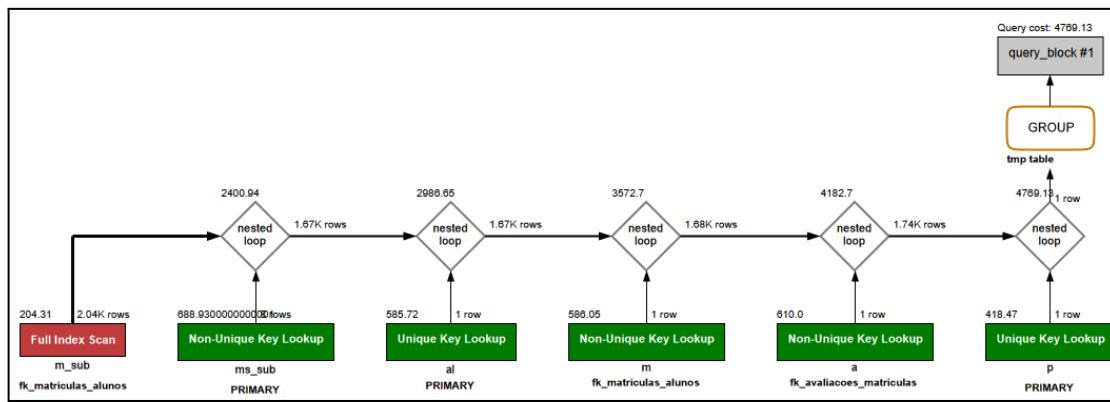
Este comportamento se justifica porque a cláusula HAVING é um filtro lógico aplicado estritamente **após** a etapa de agregação (GROUP BY). Para o otimizador, o esforço computacional para realizar os JOINs e calcular as médias de todos os alunos é o mesmo, independentemente de exibirmos o resultado final filtrado ou completo. A diferença real de desempenho neste caso não está no custo de processamento (CPU/Disco) medido pelo *Explain*, mas sim no tráfego de rede (*Network I/O*) necessário para transferir o volume maior de dados ao cliente.

### Evidência Visual:

Consulta Original:



Variação 2:



## **5. Referências**

- DATE, C. J. Introdução a Sistemas de Bancos de Dados. 8. ed. Rio de Janeiro: Elsevier, 2004.
- ELMASRI, Ramez; NAVATHE, Shamkant B. Sistemas de Banco de Dados. 6. ed. São Paulo: Pearson Addison Wesley, 2011.
- SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S. Sistema de Banco de Dados. 6. ed. Rio de Janeiro: Campus, 2012.