Artigo SQL Magazine 39 - Otimização de consultas no PostgreSQL

Artigo da Revista SQL Magazine - Edição 39.

Clique aqui para ler todos os artigos desta edição

Otimização de consultas no PostgreSQL

A todo o momento, quando se trabalha no desenvolvimento de aplicações e/ou administração de dados, busca-se sempre uma melhora de desempenho. Ao trabalhar com consultas em bancos de dados deve-se ter uma atenção maior com relação à sua eficiência, uma vez que consultas mal elaboradas podem degradar consideravelmente o desempenho do sistema como um todo.

Para que seja possível fazer melhorias relativas às consultas, é preciso entender como analisar seu desempenho, bem como os fatores que contribuem para sua melhoria.

Este artigo busca esclarecer estas questões e apresentar as ferramentas disponíveis no PostgreSQL que auxiliam nesta tarefa de otimização e, também, fornecer uma base de conhecimento para que seja possível criar consultas mais inteligentes, refinadas, objetivando o ganho de performance.

Entendendo o plano de execução

Quando se executa uma operação no banco de dados, seja ela um SELECT, INSERT, ou outra qualquer, o PostgreSQL, assim como outros SGBDs, possui um mecanismo interno chamado planejador (ou otimizador), que reescreve a consulta com a intenção de aperfeiçoar os resultados, gerando um Plano de Execução.

O Plano de Execução, ou de Consulta, é uma seqüência de passos que serão executados pelo SGBD para executar uma consulta, ou seja, quais os tipos de processamento que serão feitos diretamente nos registros ou em estruturas de índices, bem como informações como o tempo de entrada, o tempo de resposta e o total de registros percorridos. O planejador precisa então fazer uso de estatísticas como o número total de registros da tabela, o número de blocos de disco ocupados por cada tabela e se há a presença de índices ou não.

Criando um banco de dados

Para exemplificar os trabalhos do plano de execução, será criado o banco de dados ACADEMICO, utilizando a ferramenta de interface do PostgreSQL, pgAdmin III (ver **Figura 1**).

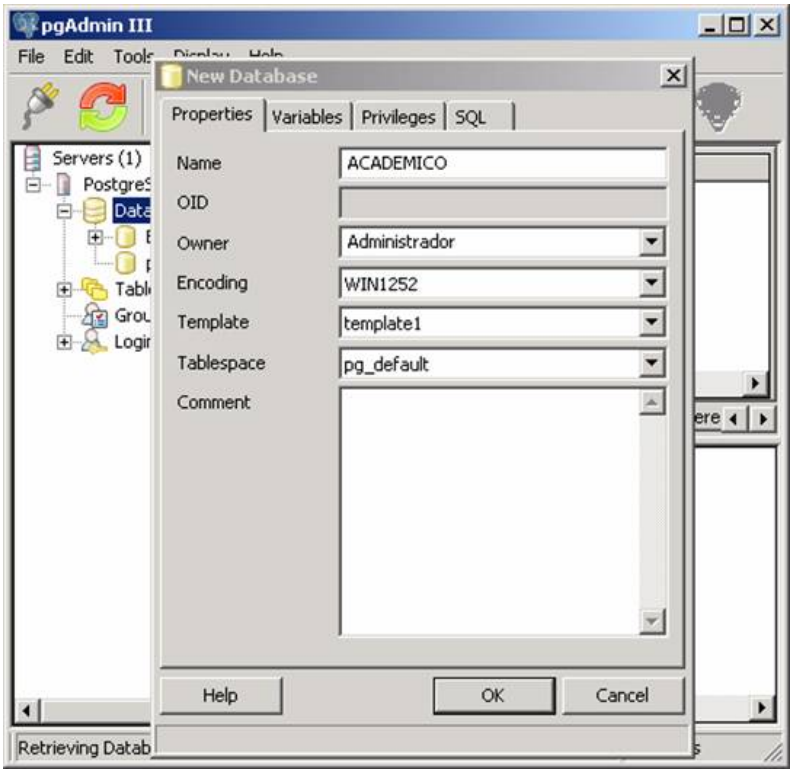

Figura 1. Criação do banco de dados Academico.

Para a criação deste banco de dados foi utilizado o usuário Administrador como proprietário, WIN1252 como tipo de caractere padrão e o template sendo o default. O PostgreSQL possui uma propriedade (tablespace) que permite ao proprietário do banco definir onde os objetos da base de dados (como tabelas e índices) irão residir. No nosso caso foi usado o caminho padrão.

Na **Figura 2**, pode-se visualizar, em linguagem SQL, o script de criação do banco de dados. A **Listagem 1** apresenta o script de criação das tabelas do banco. Para os exemplos deste artigo, foram inseridos alguns poucos registros em cada uma destas tabelas.

Figura 2. Script de criação do banco de dados.

Listagem 1. Script de criação das tabelas.

```
/*  Criação da tabela Aluno (Aluno)                                    */
CREATE TABLE Aluno (
      matricula_aluno          INTEGER       NOT NULL,
      nome_aluno               VARCHAR(30)   NOT NULL,
      end_logradouro           VARCHAR(30),
      end_numero               INTEGER,
      end_bairro               VARCHAR(20),
      telefone_residencial     VARCHAR(15),
      data_nascimento          DATE,
      cod_curso                INTEGER       NOT NULL);

/*  Criação da tabela Curso (Curso)                                    */
CREATE TABLE Curso (
      cod_curso                INTEGER       NOT NULL,
      desc_curso               VARCHAR(30)   NOT NULL,
      matricula_professor      INTEGER);

/*  Criação da tabela Professor (Professor)                            */
CREATE TABLE Professor (
      matricula_professor      INTEGER       NOT NULL,
      nome_professor           VARCHAR(30)   NOT NULL,
      titulacao_maxima         VARCHAR(10)   NOT NULL);
```

```
/*  Criação da tabela Curso_Professor (Curso_Professor)                    */
CREATE TABLE Curso_Professor (
      cod_curso                 INTEGER       NOT NULL,
      matricula_professor       INTEGER       NOT NULL);

/*  Criação da tabela Turma (Turma)                                        */
CREATE TABLE Turma (
      cod_curso                 INTEGER       NOT NULL,
      ano_turma                 INTEGER       NOT NULL,
      semestre_turma            INTEGER       NOT NULL,
      desc_turma                VARCHAR(10)   NOT NULL,
      matricula_professor       INTEGER       NOT NULL);

/*  Criação da tabela Turma_Aluno (Turma_Aluno)                            */
CREATE TABLE Turma_Aluno (
      matricula_aluno           INTEGER       NOT NULL,
      cod_curso                 INTEGER       NOT NULL,
      ano_turma                 INTEGER       NOT NULL,
      semestre_turma            INTEGER       NOT NULL,
      desc_turma                VARCHAR(10)   NOT NULL);

/*  Criação de chave primária PK_Aluno (PK_Aluno) da tabela Aluno (Aluno)   */
ALTER TABLE Aluno ADD CONSTRAINT PK_Aluno PRIMARY KEY(matricula_aluno);

/*  Criação de chave primária PK_Curso (PK_Curso) da tabela Curso (Curso)   */
ALTER TABLE Curso ADD CONSTRAINT PK_Curso PRIMARY KEY(cod_curso);

/*  Criação de chave primária PK_Professor (PK_Professor) da tabela Professor (Professor) */
ALTER TABLE Professor ADD CONSTRAINT PK_Professor PRIMARY KEY(matricula_professor);

/*  Criação de chave primária PK_Curso_Professor (PK_Curso_Professor) da tabela Curso_Professor
(Curso_Professor) */
ALTER TABLE Curso_Professor ADD CONSTRAINT PK_Curso_Professor PRIMARY
KEY(cod_curso,matricula_professor);

/*  Criação de chave primária PK_Turma (PK_Turma) da tabela Turma (Turma)   */
ALTER TABLE Turma ADD CONSTRAINT PK_Turma PRIMARY
KEY(cod_curso,ano_turma,semestre_turma,desc_turma);

/*  Criação de chave primária PK_Turma_Aluno (PK_Turma_Aluno) da tabela Turma_Aluno (Turma_Aluno) */
ALTER TABLE Turma_Aluno ADD CONSTRAINT PK_Turma_Aluno PRIMARY
KEY(matricula_aluno,cod_curso,ano_turma,semestre_turma,desc_turma);

/*  Criação das chaves estrangeiras da tabela Aluno                        */
```

```
/*  Criação da chave estrangeira FK_Aluno_01 (FK_Aluno_01)                    */
ALTER TABLE Aluno ADD CONSTRAINT FK_Aluno_01 FOREIGN KEY(cod_curso) REFERENCES Curso;

/*  Criação da chave estrangeira FK_Curso_01 (FK_Curso_01)                    */
ALTER TABLE Curso ADD CONSTRAINT FK_Curso_01 FOREIGN KEY(matricula_professor) REFERENCES
Professor;

/*  Criação das chaves estrangeiras da tabela Curso_Professor                 */
/*  Criação da chave estrangeira FK_Curso_Professor_01 (FK_Curso_Professor_01)      */
ALTER TABLE Curso_Professor ADD CONSTRAINT FK_Curso_Professor_01 FOREIGN KEY(cod_curso)
REFERENCES Curso;

/*  Criação da chave estrangeira FK_Curso_Professor_02(FK_Curso_Professor_Professor_02) */
ALTER TABLE Curso_Professor ADD CONSTRAINT FK_Curso_Professor_02 FOREIGN KEY(matricula_professor)
REFERENCES Professor ON DELETE CASCADE;

/*  Criação da chave estrangeira FK_Turma_02 (FK_Turma_02)                    */
ALTER TABLE Turma ADD CONSTRAINT FK_Turma_02 FOREIGN KEY(matricula_professor) REFERENCES
Professor;

/*  Criação da chave estrangeira FK_Turma_Aluno_01 (FK_Turma_Aluno_01)        */
ALTER TABLE Turma_Aluno ADD CONSTRAINT FK_Turma_Aluno_01 FOREIGN
KEY(cod_curso,ano_turma,semestre_turma,desc_turma) REFERENCES Turma;
```

Após a criação do banco de dados e da inserção de dados nas tabelas, serão apresentados conceitos e exemplos que interferem no desempenho de uma consulta. O desempenho dos comandos pode ser afetado por vários fatores, por exemplo, a atualização das tabelas internas utilizadas pelo planejador para formular os planos de consulta, a existência de índices e, também, a má elaboração de junções (JOINs) entre tabelas. Todos estes fatores podem ser manipulados pelos próprios usuários.

A escolha correta do planejador por um bom plano de consulta, visando o que possui menor custo, maior eficiência e melhor tempo de resposta; estruturar o comando SQL de forma planejada e; analisar as propriedades dos dados são de fundamentais para ganho de performance.

Utilizando o Plano de Execução

Aparentemente, uma simples consulta manipula um conjunto de registros. Entretanto, internamente o PostgreSQL cria um plano de execução para cada comando recebido. O plano de execução, gerado pelo SGBD, pode ser visualizado com a utilização do comando EXPLAIN. Este mostra o plano de consulta que o planejador do PostgreSQL gera para o comando fornecido, como foi feita a varredura pelas tabelas referenciadas, se houve ordenação e qual algoritmo de junção foi utilizado, no caso de junção de tabelas. A **Figura 3** apresenta um exemplo de utilização do Explain.

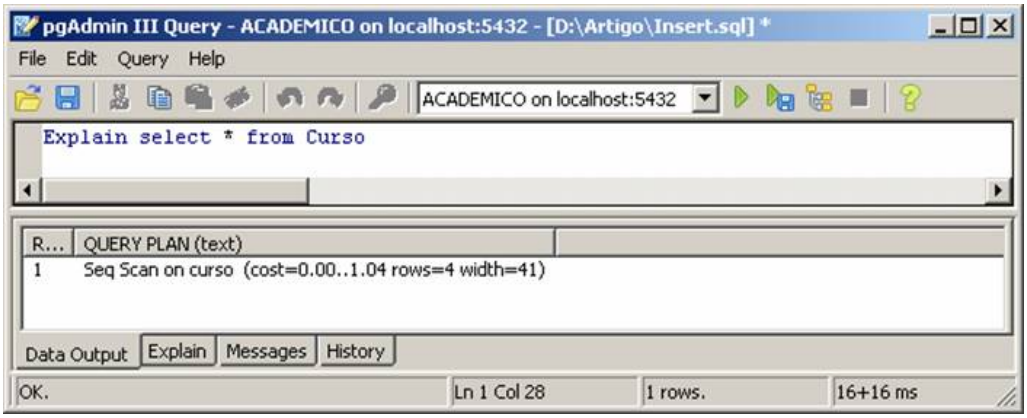

Figura 3. SELECT na tabela Curso utilizando o comando EXPLAIN.

Traduzindo o resultado apresentado:

- Seq Scan on curso: indica que foi feita uma busca seqüencial simples pela tabela curso;
- Cost = 0.00.. 1.04: indica os custos inicial e final estimados;
- Rows = 4: indica a quantidade de linhas que o plano espera encontrar;
- Width = 41: indica o tamanho médio estimado (em bytes) das linhas retornadas na consulta.

O custo é uma estimativa feita pelo planejador de quanto tempo será gasto para executar o comando. Para calcular este tempo é analisada a quantidade de páginas de disco acessadas somada à quantidade de linhas percorridas (o número total de linhas percorridas varia de acordo com fatores estipulados, denominados constantes de custo), número que também depende do tipo de varredura.

Há ainda o custo de partida que é o tempo gasto para começar a varrer a saída, ou seja, o tempo para fazer a primeira classificação, e o custo total, que estima o valor se todas as linhas fossem recuperadas. Geralmente, o tempo total terá maior importância, mas haverá casos em que o tempo inicial, ao invés do final, será enfatizado, como é o caso de uma consulta com a cláusula EXIST, já que a execução pára após ter obtido uma linha que atenda a esta condição.

Para conhecer o número de páginas em que a tabela está distribuída, utiliza-se a tabela pg_class, que é uma das tabelas de sistema do PostgreSQL. Ela possui diversas informações, como nome de todas as tabelas, número de páginas ocupadas pelas mesmas, índices e visões (ver **Figura 4**).

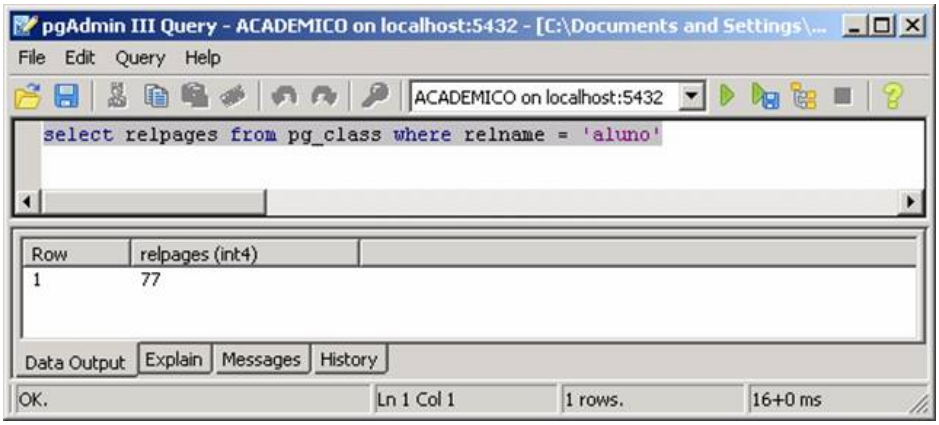

Figura 4. Número de páginas de disco ocupadas pela tabela Aluno.

O PostgreSQL possui uma ferramenta gráfica que possibilita visualizar a execução do comando EXPLAIN. Para executá-la, é necessário criar a consulta e selecionar a aba EXPLAIN, como pode ser observado na **Figura 5**.

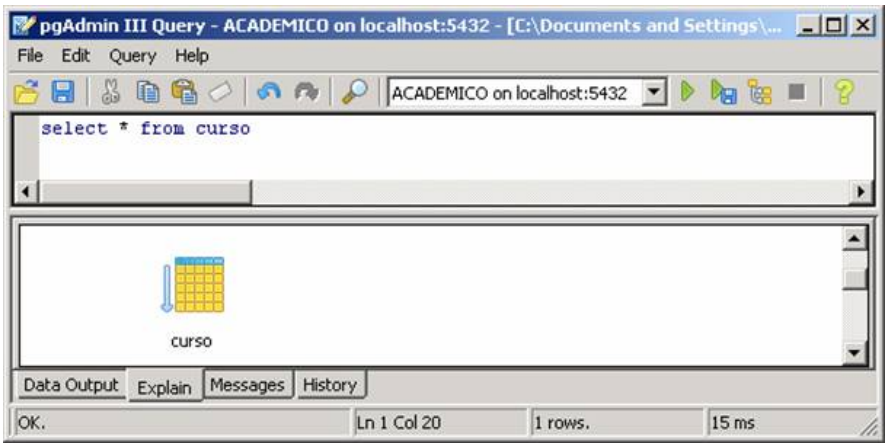

Figura 5. Representação gráfica do comando EXPLAIN.

Muitas vezes são utilizadas restrições nas consultas, geralmente fazendo uso da cláusula WHERE, com o intuito de filtrar linhas. Em termos de desempenho, utilizar este tipo de recurso precisa ser cuidadosamente avaliado.

A utilização destas operações muitas vezes não diminui o custo de execução, ao contrário, ainda aumentam, pois além de ter que ler todas as linhas sequencialmente, gasta-se um tempo a mais na verificação da cláusula WHERE, como pode ser verificado no resultado da **Figura 3** (sem a cláusula WHERE) em comparação com a **Figura 6** (com a cláusula WHERE).

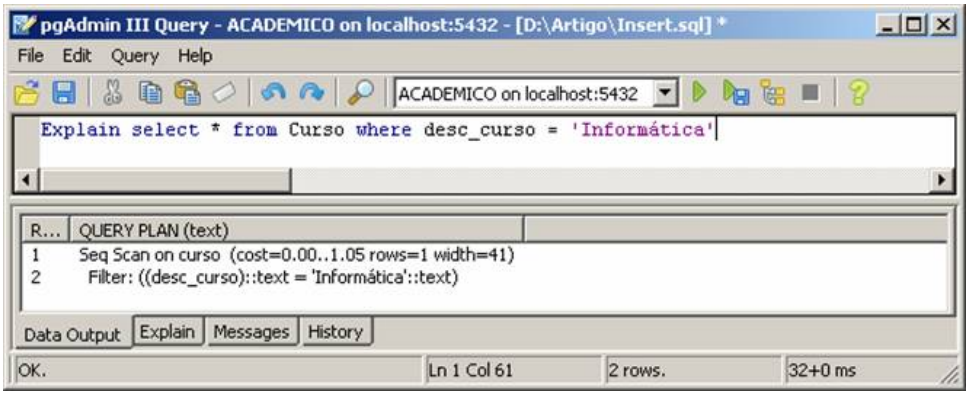

Figura 6. Exemplo de consulta com a utilização da cláusula WHERE.

Observa-se que apesar de diminuir a estimativa de linhas a serem retornadas e manter a largura média, o custo de execução teve uma relativa mudança. Isso ocorreu porque a varredura ainda precisa percorrer todas as linhas da tabela e fazer o filtro pela cláusula WHERE, aumentando o custo, refletindo no tempo maior gasto pela CPU verificando as condições da cláusula WHERE. Esta diferença pode ser significativamente maior em função da quantidade de linhas a serem recuperadas e filtradas.

Utilizando índices

Internamente, o banco de dados é armazenado em blocos de disco e, a cada consulta, todas as páginas que armazenam dados da tabela são varridas. Nestas páginas estão todas as linhas que o PostgreSQL irá ler para retornar o resultado. Esta varredura pelas páginas pode ser feita de maneira seqüencial, onde todas as linhas são lidas, ou com a utilização de índices.

Os índices são usados somente quando são bastante seletivos, ou seja, o resultado trazido deve possuir no máximo 5% do total de linhas armazenadas. Toda essa análise é feita internamente pelo SGBD para que se obtenha um ganho no custo estimado pela execução do comando.

Para melhor entendimento, será criado um índice na tabela Aluno (ver **Figura 7**).

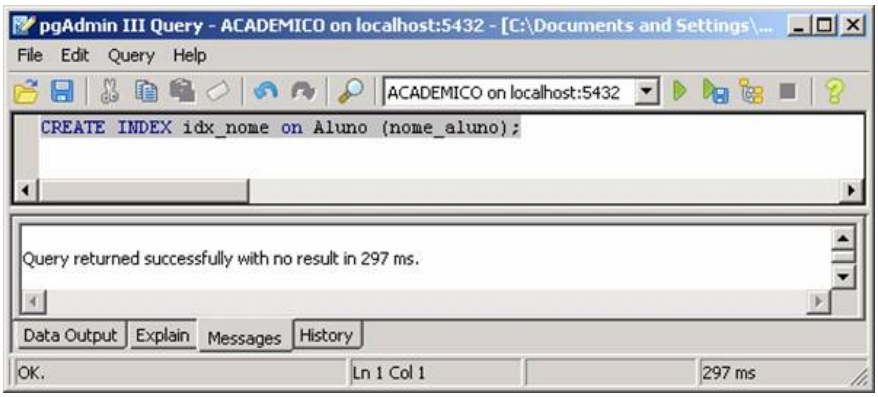

Figura 7. Criação do índice idx_nome na tabela Aluno.

Será observado que após a criação do índice e da utilização de uma cláusula WHERE bastante seletiva, o planejador decide que a varredura por índice tem um custo menor e a executa, obtendo um ganho de custo considerável (ver **Figura 8**). A mesma operação pode ser visualizada em modo gráfico na aba Explain.

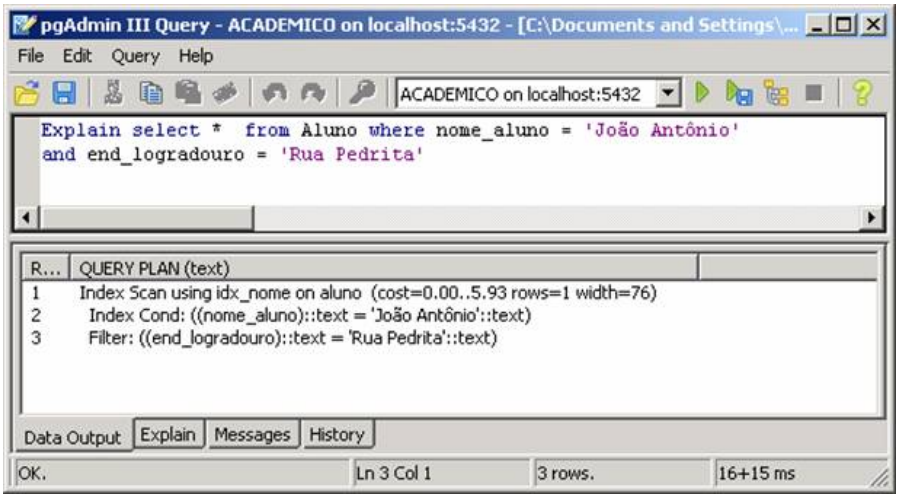

Figura 8. Varredura utilizando índice.

Observa-se também que apesar de haver duas condições na cláusula WHERE, o campo NOME_ALUNO é utilizado como condição de índice e o outro, END_LOGRADOURO, como filtro.

O fato de uma consulta ser altamente seletiva, ou seja, fazer uso de índices quando estes existem, não significa que a mesma obterá um ganho de custo. Em muitos casos, ao tentar cercar todas alternativas, esquece-se de avaliar o número de linhas retornadas, resultando na não diminuição do custo.

Utilizando junções

Em consultas onde várias tabelas são envolvidas, há a necessidade de utilizar junções para relacionar umas com as outras. Para evitar que o planejador faça uma busca exaustiva à procura da melhor solução, pode-se forçá-lo a fazer uma escolha por meio da sintaxe da junção. Essa técnica é útil tanto para reduzir o tempo de planejamento quanto para direcionar o planejador para um bom plano de execução.

A **Figura 9** apresenta uma junção entre duas tabelas. Neste exemplo, pode-se avaliar se existe algum ganho em fazer as operações de filtro (c.matricula_professor = 100) e junção (c.matricula_professor = cp.matricula_professor) numa mesma consulta ou fazê-las separadamente. A **Figura 10** mostra a mesma consulta em modo gráfico.

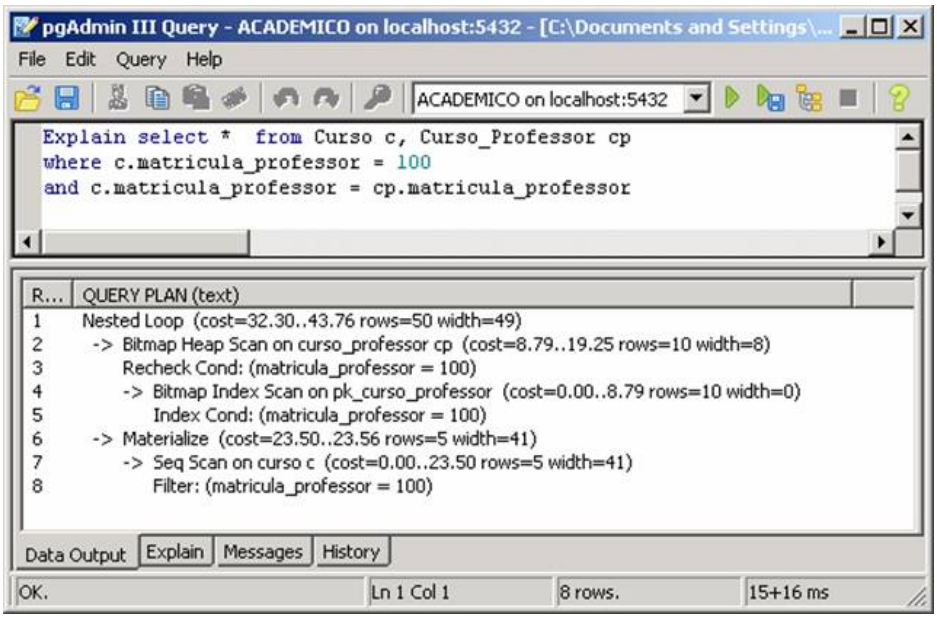

Figura 9. Exemplo do comando EXPLAIN em junções.

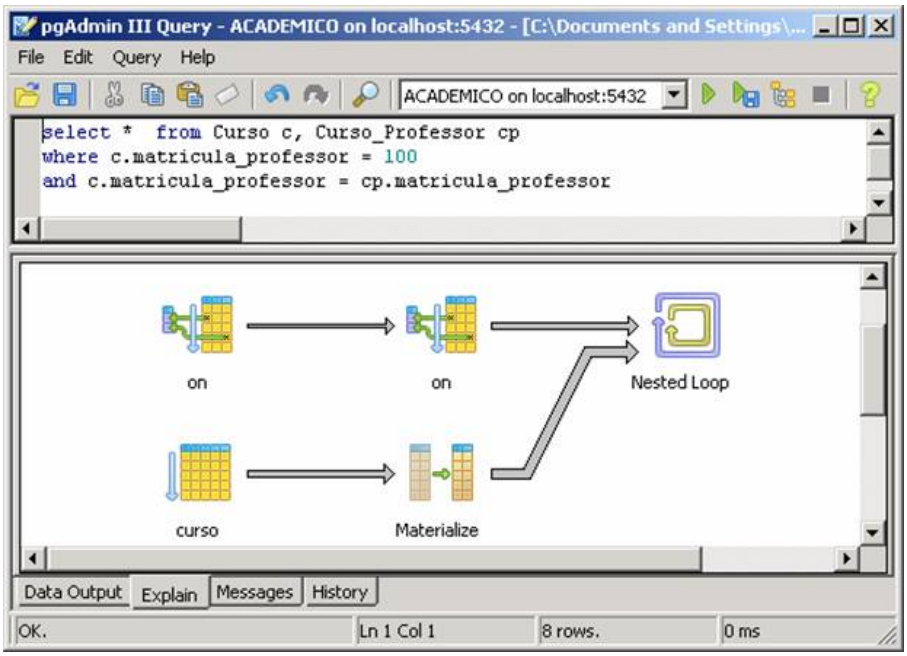

Figura 10. Exemplo do comando EXPLAIN em junções, em modo gráfico.

Uma técnica de induzir o planejador para um bom plano é explicitando quais JOINs devem ser feitos primeiramente, em consultas envolvendo várias tabelas. Pois, se o planejador respeitar a ordem de JOINs, as consultas seguintes podem levar menos tempo para serem planejadas. Esta situação é muito interessante, embora possa não ser notado um ganho de desempenho quando utilizada com poucas tabelas ou tabelas com poucos registros, mas faz grande diferença quando se tem muitas tabelas envolvidas.

Observa-se nas **Figuras 11** e **12** que enquanto uma consulta comum pode levar 683 ms para ser executada, a mesma consulta melhor elaborada obtém um tempo total de execução de apenas 278 ms. Esta diferença de 405 ms é bastante considerável.

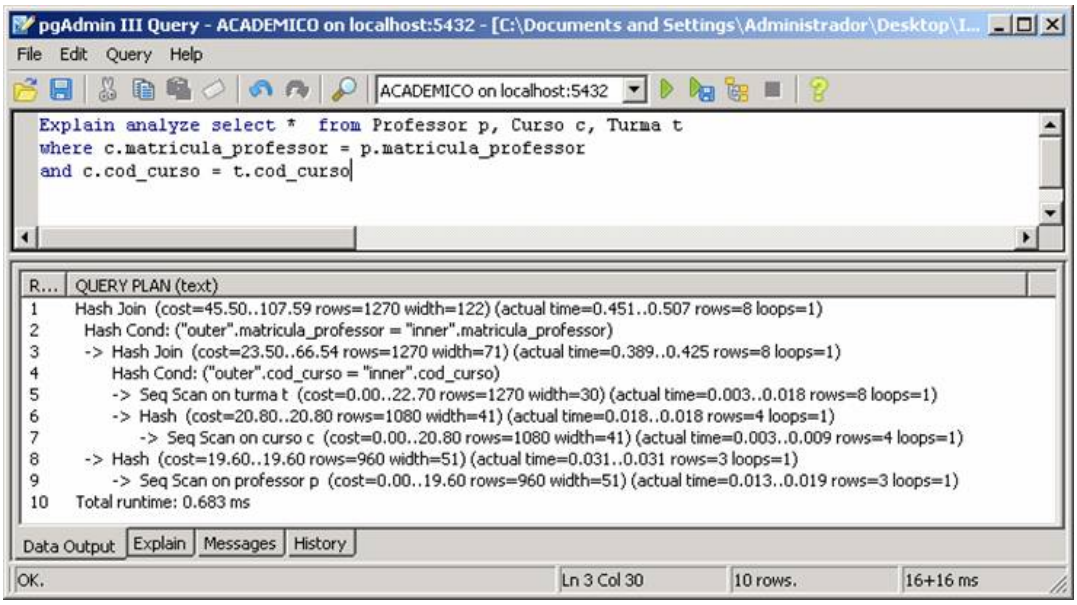

Figura 11. Consulta com junções normalmente utilizadas.

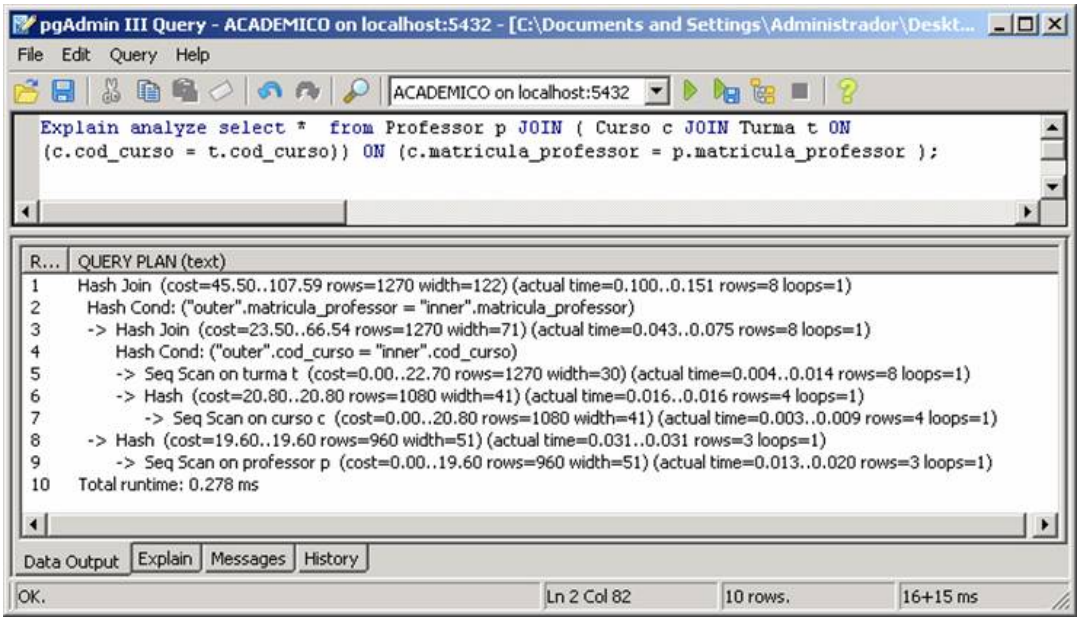

Figura 12. Consulta com junções elaboradas.

Características do comando EXPLAIN

Há diferentes opções para a utilização do comando EXPLAIN, com diferentes funções:

EXPLAIN [ANALYZE] [VERBOSE]

O comando EXPLAIN ANALYZE tem uma grande diferença do comando EXPLAIN simples, que apenas apresenta qual plano será utilizado pelo planejador. Esta sintaxe realmente executa o comando SQL, como observado na **Figura 13**.

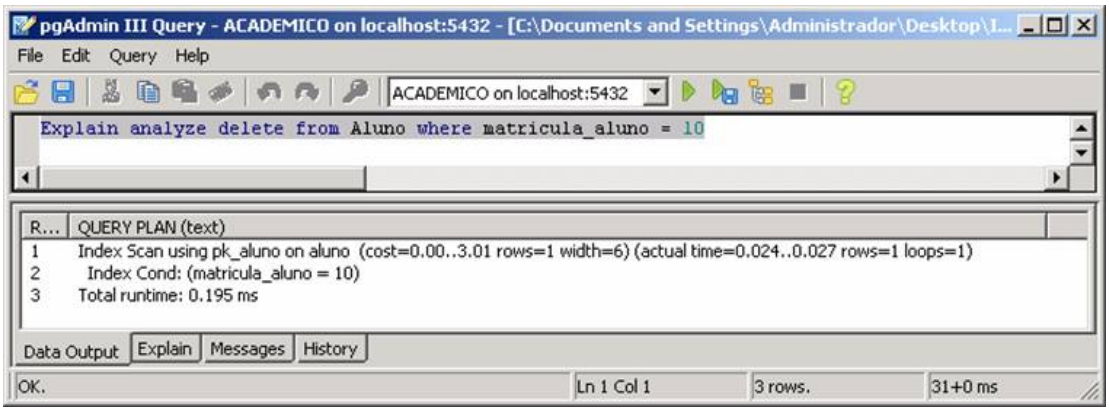

Figura 13. Execução do comando Explain Analyze para um comando Delete.

Assim, o uso do ANALYZE possibilita uma verificação mais precisa dos resultados, pois além de apresentar os custos estimados, o tempo inicial e o tempo total que o comando EXPLAIN simples mostraria, apresenta o tempo real de cada etapa do plano (ver **Figura 14**).

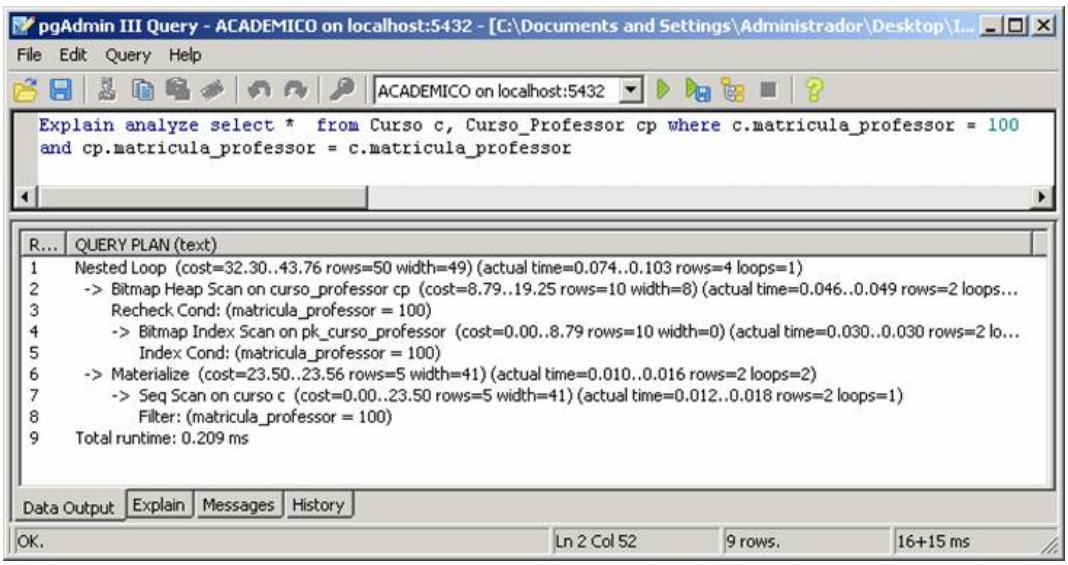

Figura 14. Exemplo do comando Explain Analyze.

Observa-se que os valores "actual time" são em milessegundos de tempo real, enquanto as estimativas de custo (cost) são expressas em unidades de busca em disco.

O tempo total de execução (Total runtime) apresentado pelo ANALYZE inclui os tempos de inicialização, de finalização e de processamento do resultado, não considerando o tempo de reescrita e nem análise da consulta.

Para os comandos INSERT, UPDATE e DELETE, o tempo de execução pode ser um pouco maior do que para um comando SELECT, uma vez que para executar um destes comandos, normalmente antes deve ser feita uma consulta para encontrar os dados a serem modificados (ver **Figura 13**).

O comando ANALYZE deve ser sempre executado para que as tabelas responsáveis por armazenar informações que serão utilizadas pelo planejador do PostgreSQL sejam atualizadas. Este comando, quando executado, registra as estatísticas sobre a distribuição dos dados na tabela pg_class. Quando isso não é feito ou se a distribuição dos dados dentro da tabela selecionada mudar significativamente, os custos podem ser calculados de maneira equivocada, gerando planos menos eficientes.

Por sua vez, o comando EXPLAIN VERBOSE mostra a representação interna completa da árvore do plano de consulta, em vez de apenas um resumo. Geralmente, esta opção é útil para finalidades especiais de depuração. A **Figura 15** apresenta a saída produzida utilizando a opção VERBOSE.

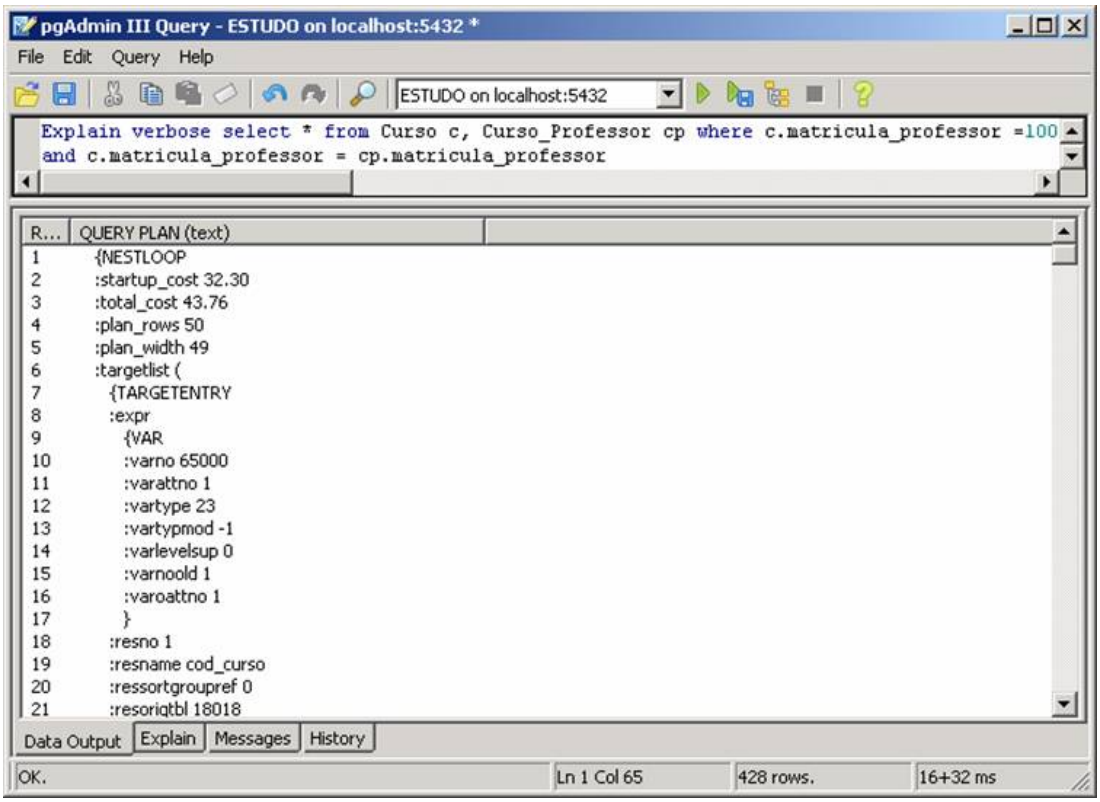

Figura 15. Exemplo do comando EXPLAIN VERBOSE.

Manipulando consultas

Existem algumas maneiras de se manipular o planejador para que ele execute diferentes planos de consulta para uma mesma operação, forçando-o a não considerar a estratégia que sairia vencedora. Isto é feito habilitando-se e desabilitando-se sinalizadores de cada tipo de plano.

Por exemplo, o PostgreSQL possui comandos como SET ENABLE_SEQSCAN, SET ENABLE_INDEXSCAN e SET ENABLE_NESTLOOP, os quais colocados em OFF desabilitam uma futura consulta do planejador utilizando consulta seqüencial, consulta com índices ou com laço aninhado, respectivamente, como pode ser visto na **Figura 16**.

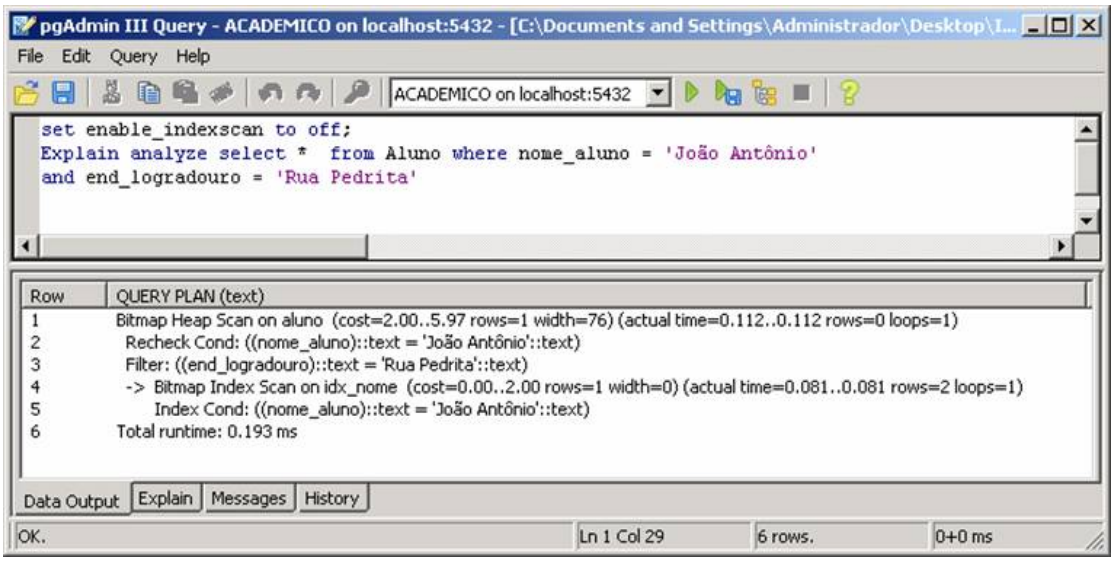

Figura 16. Desabilita varredura por índice.

Neste caso, a consulta feita anteriormente na **Figura 8** que utilizava índices, utilizou-se de outros métodos para fazer a varredura.

Um exemplo em que desabilitar uma operação pode tornar uma consulta muito mais rápida é o caso de uma tabela que ocupa uma única página de disco, e que possui colunas com índices.

O usuário sabendo desta estrutura inibe a execução dos índices, como no exemplo anterior, para forçar o planejador a executar uma busca seqüencial ao invés de fazer a leitura por índices. A consulta, neste caso, se torna mais eficiente uma vez que evitaria a leitura de páginas adicionais na procura por índices.

Finalizando, pode-se observar na **Tabela 1** um resumo das principais operações do PostgreSQL utilizadas em planos de consulta.

Tabela 1. Descrição das principais operações encontradas nos planos de consulta.

Operação	**Descrição**
Seq Scan	É a operação mais básica e representa uma leitura seqüencial da tabela. Quando ocorre uma Seq Scan, significa que a tabela foi lida da primeira à ultima linha.
Index Scan	Ocorre quando há uma leitura em alguma estrutura de índice a fim de se evitar uma leitura inteira da tabela, ou quando o planejador deseja aproveitar a ordem do índice para evitar uma operação de Sort.
Sort	Ocorre principalmente para satisfazer a cláusulas de ORDER BY, mas também é utilizada quando outra operação necessita de conjunto de dados ordenado para ser executada.
Unique	Essa operação é utilizada principalmente para satisfazer a cláusulas de DISTINCT, eliminando repetições.
Nested Loop	É utilizada para realizar JOIN entre duas tabelas. Ocorre principalmente em INNER JOINs, LEFT JOINs e UNIONs. Não processa completamente a tabela mais interna.
Hash e Hash Join	

	As operações de Hash e Hash Join trabalham juntas para fazer JOINs entre tabelas como INNER JOINs, LEFT JOINs, e UNIONs. Nas operações em que ocorrem Hash Join as colunas que fazem a junção não precisam estar ordenadas.

Conclusão

Há muitas outras maneiras de se manipular consultas para encontrar um resultado satisfatório. Neste artigo foram enfatizados importantes recursos do PostgreSQL que podem ser muito úteis em grandes e pequenas aplicações.

Para melhor compreensão destes recursos, torna-se interessante conhecer as tabelas de sistema e os benefícios que podem ser retirados delas, além de conhecer outros comandos que podem proporcionar a criação de aplicações com melhor desempenho.

Leonardo Barbosa Lima Corrêa Pires (lbpires@gmail.com) é graduando do Centro de Ensino Superior de Juiz de Fora - CES/JF no curso de Bacharelado em Sistemas de Informação. Atualmente trabalha no desenvolvimento de software para arquitetura e paisagismo na AuE Soluções em Juiz de Fora/MG.

Marco Antônio Pereira Araújo (maraujo@cesjf.br) é Doutorando e Mestre em Engenharia de Sistemas e Computação pela COPPE/UFRJ, Especialista em Métodos Estatísticos Computacionais e Bacharel em Matemática com Habilitação em Informática pela UFJF, Professor do Curso de Bacharelado em Sistemas de Informação do Centro de Ensino Superior de Juiz de Fora, além de Analista de Sistemas da Prefeitura de Juiz de Fora.

Lucas Rezende Ricardo

é estudante do Curso de Bacharelado de Sistemas de Informação no Centro de Ensino Superior de Juiz de Fora. Atua como programador e membro da equipe de administração de dados da Zeus Rio Solutions (www.zeusrio.com.br). Trabalha co [...]

Publicado em 1899