colloquium exactarum

ANÁLISE COMPARATIVA DOS ALGORITMOS DE OTIMIZAÇÃO DE CONSULTAS DO POSTGRESQL

COMPARATIVE ANALYSIS OF ALGORITHMS OPTIMIZATION POSTGRESQL QUERY

Lucas Bianchi Magalhães; Francisco Virginio Maracci; Aglaê Pereira Zaupa; Francisco Assis da Silva

Universidade do Oeste Paulista – UNOESTE, Faculdade de Informática. e-mail: aglae@unoeste.br

RESUMO – A utilização de banco de dados se tornou essencial para o armazenamento de informação. Os sistemas gerenciadores de banco de dados (SGBD) são responsáveis por facilitar a manipulação das informações por meio de consultas SQL, realizar buscas com o melhor desempenho e menor tempo requer uma otimização. O PostgreSQL utiliza dois algoritmos para efetuar a otimização das consultas (programação dinâmica e algoritmos genéticos). O que diferencia a utilização dos algoritmos são as quantidades de tabelas envolvidas no SQL. Este artigo tem como objetivo efetuar análise dos algoritmos em várias consultas com diferentes quantidades de tabelas, procurando identificar em qual situação a utilização de um determinado algoritmo é mais apropriada.
Palavras-chave: Banco de Dados; Otimização; PostgreSQL.

ABSTRACT – The use of databases to keep information turn an essential need. The databases management systems are responsible to realize SQL queries that manipulate the information in an easy way. The PostgreSQL have two different algorithm to optimization queries (dynamic programming and genetic algorithm). The amount of tables used in a query is what differentiates the algorithm to be used. This paper aim to make a analyze using the algorithms with different amount of tables in queries trying to verify in which situation the most appropriated algorithm.
Keywords: Database; Optimization; PostgreSQL.

Recebido em: 10/06/2014
Revisado em: 20/11/2014
Aprovado em: 25/05/2015

1 INTRODUÇÃO

A informatização dos processos empresariais vem aumentando de acordo com passar dos dias, e como consequência, muitas informações são geradas e armazenadas para sua futura reutilização. Estas informações são armazenadas, mantendo relações entre si, em um Banco de Dados (BD). Com o aumento dos dados armazenados, torna-se necessário a utilização de um Sistema Gerenciador de Banco de Dados (SGBD), cuja responsabilidade é armazenar, organizar, recuperar e fornecer toda segurança necessária aos dados. Em qualquer operação, o acesso aos dados e a resposta deve ser o melhor possível, para isso, o SGBD precisa otimizar as consultas SQL recebidas, a otimização de consulta é o processo de selecionar o plano de execução mais eficiente para obter a resposta solicitada. A eficiência da recuperação depende de fatores físicos (velocidade e capacidade do disco, quantidade de memória disponível e capacidade do processador) e lógicos (métodos e estrutura do banco de dados). Os SGBD's utilizam algoritmos determinísticos, aleatórios e genéticos para otimizar. Os algoritmos determinísticos constroem a solução utilizando busca exaustiva ou aplicando heurísticas, nos algoritmos aleatórios a solução final é encontrada por meio de um conjunto de transformações aleatórias. Os algoritmos genéticos tem seu método de otimização inspirado no processo biológico de seleção natural.

Considerando a importância da otimização de consultas, este artigo tem como objetivo principal detalhar e comparar os algoritmos utilizados na otimização do SGBD PostgreSQL, demonstrando os resultados em diversos comandos SQL. Os algoritmos avaliados serão: programação dinâmica (determinístico) e genético.

A escolha do PostgreSQL se motivou por ser um SGBD gratuito e de código aberto.

2 OBJETIVO

O propósito deste trabalho é apresentar um comparativo sobre os algoritmos de otimização do sistema de gerenciamento de banco de dados PostgreSQL e verificar se a otimização padrão do SGBD é ideal para algumas situações, com o uso de duas consultas empresariais. A quantidade de tabelas de uma consulta é o que determina o algoritmo utilizado, as configurações iniciais do PostgreSQL determinam que consultas com até onze tabelas serão otimizadas pela programação dinâmica, e a partir de doze tabelas serão otimizadas pelo algoritmo genético (THE POSTGRE GLOBAL DEVELOPMENT GROUP, 2014), o objetivo será atingido alterando a

quantidade de tabelas, que limita o uso dos algoritmos, na configuração do SGBD.

3 REVISÃO BIBLIOGRÁFICA

3.1 BANCO DE DADOS

Um sistema de banco de dados é uma coleção de dados que estão interligados possuindo significado implícito, a interpretação é dada por uma determinada aplicação (ELMASRI, 2010).

O banco de dados armazena dados para posteriormente o usuário recuperar ou atualizar as informações. Os bancos de dados envolvem quatro componentes básicos: dados, *hardware, software* e o usuário.

Segundo Martins Colares (2007), os dados são os registros realizados eletronicamente, podendo ser acessados por apenas um ou diversos usuários, devem ser integrados e compartilhados, ou seja, para ser integrado o banco de dados deve evitar a redundância, e para ser compartilhado, deve permitir o acesso ao mesmo dado por diversos usuários ao mesmo tempo.

O *Hardware* é toda parte física que participa na existência do banco de dados, responsáveis por manter os dados armazenados, e processadores e memória principal, responsáveis por dar suporte a execução do SGBD.

O *software* é toda parte lógica que acessam e manipulam os dados, os SGBD's, ficam na camada entre os dados físicos e os usuários, é responsável por fornecer um ambiente agradável e fácil para a manipulação dos dados.

O banco de dados tem como principais objetivos: reduzir a redundância (evitar armazenamento da mesma informação várias vezes); suportar transações (unidade responsável pelo controle de várias atualizações ao mesmo tempo); segurança dos dados (controlar o acesso à base de dados por meio de restrições); garantir a integridade (assegurar que os dados estão corretos de acordo com as consistências e coerências); e manter padrões impostos pelo administrador do banco, facilitando o compartilhamento e à compreensão dos dados (SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

3.2 SGBD

Com o grande crescimento do ambiente tecnológico, os dados aumentaram de tal forma que dificultou o seu manuseio e o seu arquivamento. Um sistema de gerenciamento de banco de dados (SGBD) é um conjunto de programas de computador responsáveis pelo gerenciamento de uma coleção de dados inter-relacionados, o seu principal objetivo é proporcionar um ambiente conveniente ao usuário e eficiente para o armazenamento e recuperação dos dados, tirando a responsabilidade da

aplicação cliente de gerenciar o acesso e a manipular os dados.

Sistemas de banco de dados foram projetados para administrar grandes volumes de dados, criando estruturas de armazenamento e definindo mecanismos de manipulação (LANGE, 2010).

De acordo com Ramarkrishnan e Gehrke (2011), as vantagens de um SGBD são:

- Independência dos dados: os aplicativos não precisam ser expostos aos detalhes de representação e armazenamento de dados.
- Acesso Eficiente aos Dados: a organização da apresentação dos dados minimizam a redundância, garantindo uma eficiente recuperação de informação.
- Integridade e Segurança dos Dados: o SGBD reforça a integridade dos dados utilizando restrições na manipulação dos dados, por exemplo, restringe a exclusão de um registro caso o mesmo tenha ligação com outro registro. O controle de acesso é utilizado para assegurar que cada nível de usuário irá acessar as informações liberadas pelo administrador.
- Administração de Dados: centralizar a administração aos profissionais experientes oferece melhorias significativas quando diversos usuários compartilham dados, minimizando as redundâncias, para garantir que o SGBD forneça uma eficiente recuperação.
- Acesso Concorrente e Recuperação de Falha: o SGBD realiza o acesso concorrente de tal forma que o usuário não percebe que o dado está sendo acessado por mais de um usuário, e protege o usuário dos efeitos de falhas do sistema.
- Tempo Reduzido de Desenvolvimento de Aplicativo: as funções suportadas são comuns a vários aplicativos que acessam os dados no SGBD, facilitando o desenvolvimento rápido de softwares.

3.3 SQL

A linguagem SQL (*Structured Query Language*) é a linguagem padrão para acesso e manipulação do banco de dados (COSTA, 2007).

Com o crescente uso dos sistemas gerenciadores de banco de dados, fez-se necessária a criação de um padrão para os acessos, o que possibilitou o sucesso dos Sistemas Gerenciadores Relacionais.

A linguagem SQL usa uma sintaxe simples, fácil de aprender e utilizar, de acordo com os estudos de Price (2008), os comandos são divididos em:

- Instruções de Consulta (*SELECT*);
- Instruções de DML (*Data Manipulation Language*): que modificam os dados (*INSERT, UPDATE e DELETE*);
- Instruções DDL (*Data Definition Language*): que permite definir estruturas de dados

(*CREATE, ALTER, DROP, RENAME* e *TRUNCATE*);

- Instruções TC (*Transaction Control*): registram permanentemente as alterações (*COMMIT, ROLLBACK e SAVEPOINT*);

- Instruções DCL (*Data Control Language*): alterando as permissões definidas no banco de dados (*GRANT e REVOKE).*

Algumas características principais do SQL é a capacidade de gerenciar índices (arquivo auxiliar associado a uma tabela com finalidade de acelerar o tempo de acesso aos registros) sem a necessidade de controle individualizado, a construção de visão (tabela virtual cujo conteúdo foi definido por uma consulta), facilitando a visualização dos registros em forma de tabelas, e a capacidade de aceitar e desfazer alterações.

3.4 OTIMIZAÇÃO DE CONSULTAS

Uma consulta é uma solicitação de informação a ser recuperada do banco de dados. Ao receber uma consulta SQL o sistema de gerenciamento de banco de dados precisa encontrar o melhor método para obter a resposta solicitada. O SGBD efetua um tratamento chamado de processamento de consultas (sequência de atividades que devem ser executadas). O SGBD precisa criar um plano de execução eficiente para recuperar o resultado, com o menor custo possível. A escolha da execução para o processamento da consulta é chamada de otimização de consulta (MORAES ARCANJO; NUNES DE LIMA, 2012).

O principal passo da otimização é traduzir a consulta SQL para álgebra relacional, e após, otimizar o resultado. Para transformar em álgebra relacional, a consulta é decompostas em blocos, e cada bloco é transformado em uma expressão de álgebra relacional, assim, os blocos são otimizados separadamente. A otimização pode ser baseada em heurísticas ou em custo.

A otimização heurística é representada por árvore ou grafo, as árvores determinam a ordem de execução e o grafo indica as operações envolvidas, de acordo com o comando SQL, o SGBD gera a árvore algébrica e então a árvore é otimizada de acordo com as regras. Segundo Elmasri (2010) as regras são:

1. Quebrar quaisquer seleções com condições conjuntivas em uma cascata de seleções. Isso permite aumentar o grau de movimentação.
2. Mover cada operação de seleções o mais baixo possível na árvore de consulta, conforme for permitido pelos atributos da condição. Se a condição envolver uma tabela, a operação é movida até o nó da folha que representa essa tabela. Se a condição envolver duas tabelas, a condição é movida para um local abaixo da combinação das tabelas.

3. Reorganizar os nós folha da árvore. Posicionar as seleções mais restritivas (menor tamanho) com o nó folha e garantir que a ordenação não cause operações de produto cartesiano.

4. Combinar operação de produto cartesiano com operação de seleção subsequente na árvore para uma condição de junção.

5. Desmembrar e mover a lista de atributos de projeção para baixo da árvore ao máximo possível, criando novas junções.

6. Identificar subárvores que representam operações que podem ser executadas por um único algoritmo.

Aplicando as regras no SQL a seguir: a Figura 1 determina a árvore inicial e a Figura 2 a árvore otimizada.

```
SELECT unome
FROM funcionario, trabalha_em,
projeto
WHERE projnome = 'Aquarius'
AND projnumero = pnr
AND fcpf = cpf
AND datanasc>'31-12-1957'
```

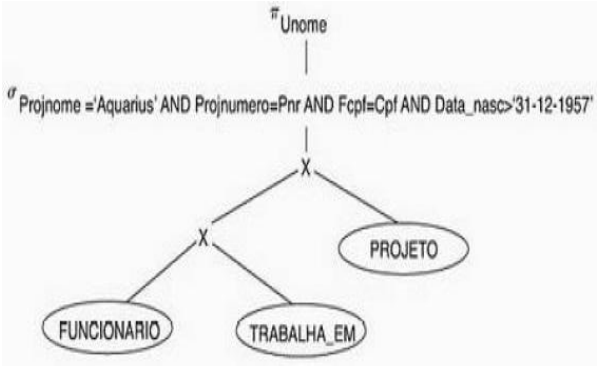


Figura 1. Árvore de consulta algébrica inicial.
Fonte: (Elmasri, 2010).

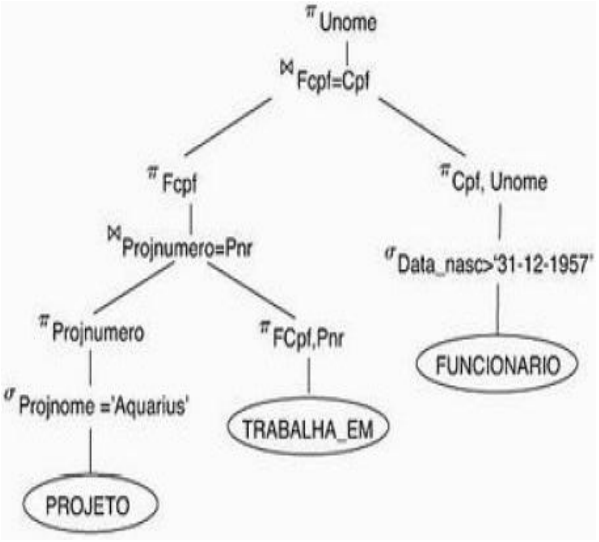


Figura 2. Árvore de consulta algébrica otimizada.
Fonte: (Elmasri, 2010).

Com a árvore de consulta otimizada, o plano de execução é definido utilizando informações como índices disponíveis, algoritmos e os métodos de acesso.

A otimização baseada em custo utiliza estatísticas sobre os números de tuplas, as chaves, o tamanho das tabelas, os índices, uso do CPU, acesso a armazenamento secundário, armazenamento de tabelas temporárias e uso de memória. Os custos dos planos de execução são comparados e o escolhido é o que possui menor estimativa de custo (GUTTOSKI, 2006).

Segundo Lange (2010), as classes de algoritmos utilizados no processo de otimização são:

- Algoritmos Determinísticos: apresentam comportamento previsível, possuem sequência finita de passos, e constroem a solução utilizando busca exaustiva ou aplicando heurísticas. Pertencem a essa

classe algoritmos programação dinâmica, seletividade mínima e algoritmos heurísticos. Na programação dinâmica as possíveis árvores de execução são construídas e avaliadas, a consulta é dividida e a melhor solução para o problema é dada utilizando as partes menos custosas. A seletividade mínima é composta pelos métodos gulosos, que busca a construção de uma árvore geradora mínima, nestes algoritmos o ganho em cada etapa é o máximo, sem verificar a consequência da escolha para a próxima etapa. Os algoritmos heurísticos encontram solução entre todas possíveis, mas não garantem que a solução seja a ótima, pois são algoritmos de aproximação e não de precisão.

- Algoritmos Aleatórios: a solução final é encontrada através de um conjunto de transformações aleatórias, a melhor solução encontrada é mantida e comparada com outra solução, assim, a melhor solução é encontrada ao final das possíveis transformações. Os algoritmos Melhoria Iterativa e Têmpera Simulada são exemplos de algoritmos aleatórios.

- Algoritmos Genéticos: formam outra classe de algoritmos não exaustivos, tem seu método de otimização inspirado no processo biológico de seleção natural, em vez de considerar apenas uma solução por vez, o algoritmo aplica várias regras de transformação para produzir novas soluções simultaneamente. São considerados algoritmos genéticos as árvores em profundidade à esquerda e árvores fechadas de junções.

4 POSTGRESQL

O PostgreSQL é um sistema de gerenciamento de banco de dados relacional e orientado a objetos, possui recursos em comuns a banco de dados de grande porte, é versátil, seguro, gratuito e tem código aberto, se tornando uma ótima opção para a comunidade acadêmica e empresas que querem alta confiabilidade e baixos custos de licenciamentos.

Segundo Souza, Amaral e Lizardo (2011), “Seus desenvolvedores são, em sua maioria, voluntariados espalhados pelo mundo que se comunicam pela internet”.

O SGBD possui suporte a *triggers*, *stored procedures*, suporte a transações, controle de integridade referencial, não há limite quanto ao número de usuários, controle de transações concorrentes e suporte nativo as quatro propriedades ACID, onde ACID significa Atomicidade, consistência, isolamento e durabilidade (CARVALHO SANTOS, s.d.).

Segundo The Postgre Global Development Group (2014), ele oferece características modernas como:

- Consultas completas: instruções *SELECT* dentro de outra instrução *SELECT*;

- Chaves estrangeiras: relacionamento entre tabelas distintas;
- Gatilhos (t*riggers*): comandos executados automaticamente;
- Exibições atualizáveis: Utilização de *View*;
- Integridade transacional: garante propagar atualização dos campos e a exclusão dos registros relacionados;
- Controle de concorrência multiversão: padrão ACID;

Ao executar a consulta SQL o SGBD gera o plano de execução, sequência de métodos necessária para obter o resultado, no PostgreSQL o plano é realizado por dois algoritmos, programação dinâmica, utilizado para consultas com até 11 tabelas, e algoritmo GEQO, genético utilizado para consultas com 12 ou mais tabelas.

- Programação Dinâmica: utiliza busca exaustiva, todos os planos possíveis para se fazer uma busca (*scan*) são gerados para cada relação (tabela). O plano de execução da busca sequencial (*sequential scan*) sempre é gerado, e para cada índice encontrado, é gerado um plano de busca (*index scan*) através desse índice (GUTTOSKI, 2006). Para consultas de apenas uma tabela, o de menor custo é escolhido, e para consultas com mais de uma tabela, o próximo passo é realizar a junção da tabela não utilizada com o resultado já obtido, utilizando métodos *nested-loop join, merge sort join e hash join*.

A junção entre o resultado e tabelas não verificadas repete até não existir mais tabelas sem verificação. No SQL: SELECT * FROM nf_eletronica n, nf_duplicatas d, clientes c; as junções realizadas são: {n, d}, {d,c}, {n,d,c}.

No método *nested-loop join*, os registros da primeira tabela são recuperados, e para cada registro recuperado, o acesso é realizado na segunda tabela; no *merge sort join* as duas relações são ordenadas e a busca é realizada em paralelo; e no método *hash join* a relação da direita é inserida em uma tabela hash e os valores da relação da esquerda são localizados na tabela hash.

A Figura 3 é um exemplo da árvore binária das operações físicas realizadas pelo otimizador sobre o SQL:

```
EXPLAIN ANALYZE
SELECT n.nf_numero, c.cli_codigo
FROM nf_eletronica n, clientes c
WHERE c.cli_codigo = n.cli_codigo
   AND n.nf_numero in
 (select nf_numero from nf_duplicatas
where      dnf_datavencimento      >=
'01.01.2014')
ORDER BY n.nf_numero DESC
LIMIT 1;
```

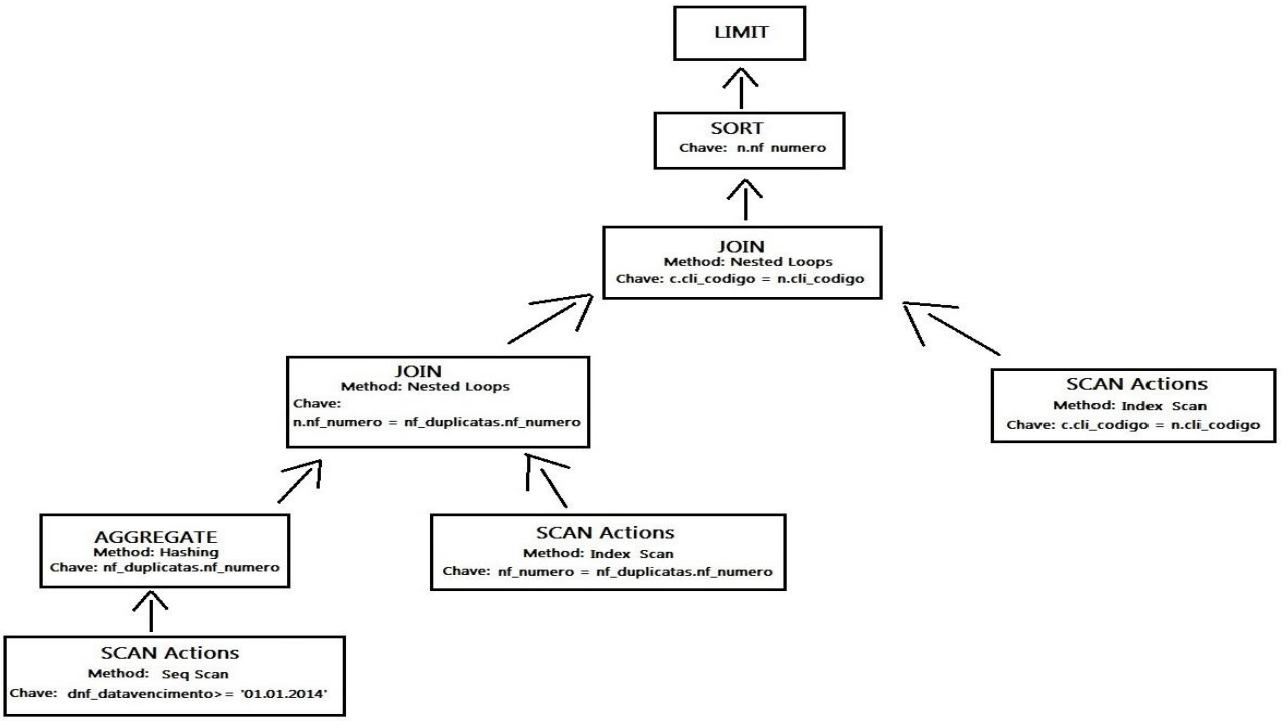


Figura 3. Plano de execução da consulta SQL gerado pelo PostgreSQL.

- Genético: *Genetic Query Optimization* (GEQO), utiliza buscas não exaustivas, determinísticas e aleatórias. Usa seleção natural, recombinação, novos indivíduos com melhores valores e mutação para gerar o plano de execução. O desempenho não é considerado como suficiente por ser um método aleatório. O método aborda o problema de otimização como se fosse o problema do Caxeiro Viajante, problema que consiste em encontrar o melhor caminho para passar em todas as cidades interligadas por rodovias. As consultas são decodificadas em inteiros, e cada sequência representa a ordem das junções, a Figura 4 representa a árvore gerada no SQL: SELECT * FROM nf_eletronica 1, nf_duplicatas 2, clientes 3, itens_nf_eletronica 4;

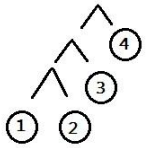


Figura 4. Árvore de junção da consulta SQL.

É possível que alguns elementos apresentem falhas de combinação entre as relações, em exemplo a ausência de predicados da junção, o algoritmo utiliza alternativa de combinações para utilizar o elemento, e quando um produto cartesiano é encontrado, uma combinação paralela é feita, para assim reutilizar quando existir um predicado da junção. Para demonstrar as falhas, a Figura 5 representa uma árvore com alternativa de combinações.

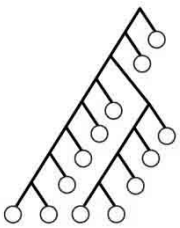


Figura 5. Exemplo de uma árvore alternativa para representar falhas em sua sequência normal de combinações.
Fonte: (Lange, 2010).

Ao iniciar o algoritmo de otimização por consulta genética gera algumas possíveis sequências de junções de forma aleatória. Para cada junção o custo da consulta é estimado e o algoritmo descarta os candidatos menos aptos em seguida novos candidatos são gerados aleatoriamente até que o número estabelecido nas configurações de junções a serem utilizadas seja atingido.

5 DESENVOLVIMENTO

5.1 COMANDO *EXPLAIN*

O comando *explain* é utilizado para visualizar e analisar o comportamento do otimizador sobre uma consulta SQL no SGBD. Com ele, é possível encontrar informações de como é feita a análise das tabelas, como é feito as junções e a ordem executadas, os custos dos métodos utilizados e o plano de execução (SAMPAIO; GOVEIA; MARQUES, 2011). O *explain* é um dos recursos mais utilizados por administradores de banco de dados (GOÉS PEDROZO, 2008). O comando tem a seguinte estrutura:

EXPLAIN [(option[, ...])] statement
EXPLAIN [ANALYZE] [VERBOSE] statement

onde *option* pode ser uma das seguintes:

- *ANALYZE*: Executa o comando e mostra os tempos reais de execução.
- *VERBOSE:* Exibe informações adicionais, como a lista de colunas de saída de cada nó da árvore gerada.
- *COSTS:* Inclui custo de início, custo total de cada nó, estimativa de linhas e comprimento.
- *BUFFERS:* Informações sobre o uso do *Buffer*.

Como resultado, o comando apresenta informações de custo (*cost*), quantidade de linhas retornadas (*rows*) e tamanho médio em bytes das linhas (*width*). A árvore gerada contém métodos ordenação (*sort*), repetição aninhada (*nested loop*), filtro de união (*join filter*), junções hash (hash *join*), sequência (*seq scan*), unidade de medição (*buckets*), verificação do índice (*index scan*), chave (*key*) e filtro (*filter*).

5.2 PROPRIEDADE GEQO

O PostgreSQL armazena suas principais configurações no arquivo postgresql.conf, armazenado por padrão dentro do subdiretório *data* de instalação do SGBD, há configurações sobre os métodos de planejamento, constantes sobre o custo do planejador, otimização genética e outras opções. As configurações sobre otimização

genética, *geqo_threshold* e *geqo_seed*, é de grande importância ao trabalho.

A configuração *threshold* determina a quantidade de tabelas necessárias em uma consulta SQL para que a otimização genética (GEQO) seja utilizada sendo o valor padrão estabelecido em doze tabelas. A mudança no valor desta propriedade fornece o controle do uso do algoritmo genético, podendo ser realizada diretamente no arquivo de configuração ou por SQL: set geqo_threshold = 12.

A propriedade *seed* controla o valor utilizado pelo GEQO para selecionar caminhos aleatórios por meio das alternativas existentes, o valor padrão é zero, podendo ser alterado apenas para um. Variando o seu valor, altera o conjunto de uniões percorridas, podendo resultar em um melhor ou pior caminho encontrado com o valor padrão (THE POSTGRE GLOBAL DEVELOPMENT GROUP, 2014). A alteração desta propriedade pode ser feita diretamente no arquivo de configuração ou por SQL: set geqo_seed = 0.

5.3 DESENVOLVIMENTO GERAL

Para o desenvolvimento dos testes, foi necessária a cópia de uma base de dados utilizada em uma empresa real, onde as inserções de dados foram realizadas de acordo com as necessidades do cotidiano empresarial, o foco da empresa é extração de produto para a venda e a revenda de produtos adquiridos de terceiros. A Figura 6 representa o modelo da base de dados utilizada.

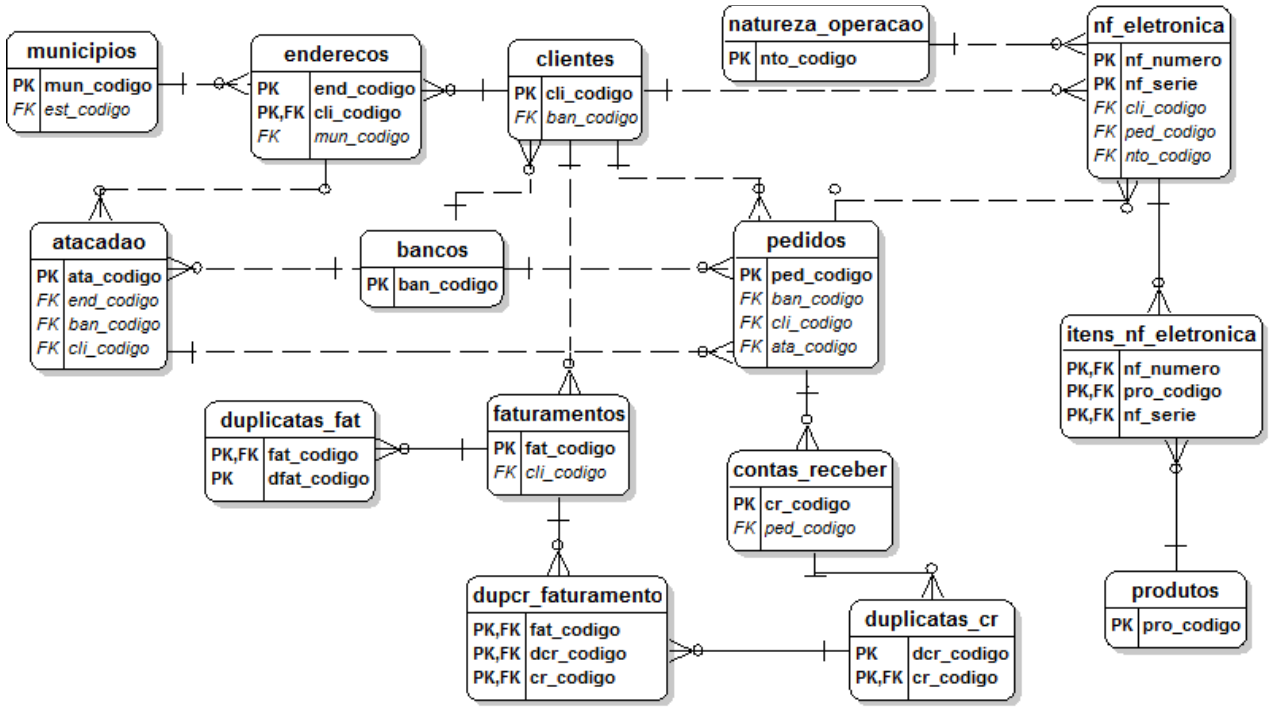


Figura 6. Modelo relacional do banco de dados.

As consultas SQL's utilizadas foram buscadas em relatórios gerenciais utilizados pela empresa, o comparativo será realizado sobre duas consultas diferentes, a primeira consulta tem o objetivo de mostrar os produtos que foram vendidos por municípios em um determinado período, e a segunda consulta tem o objetivo de identificar qual é a inadimplência atual sobre as vendas de acordo com o município e o banco responsável pela emissão da cobrança. As consultas tem grande importância para a interpretação gerencial do cotidiano empresarial. Em seguida, serão detalhados os sql's das consultas, os resultados obtidos e os caminhos percorridos pelo otimizador utilizando os dois tipos de algoritmos de otimização:

1)

```
Explain Analyze
select m.mun_codigo as CodMun,
mun_nome as Municipio, pro.pro_codigo
as CodPro, pro_descricao as Produto,
sum(inf_quantidade) as Vendido
from nf_eletronica nf inner join
itens_nf_eletronica inf on
(nf.nf_numero = inf.nf_numero and
nf.nf_serie = inf.nf_serie)
inner join produtos pro on
inf.pro_codigo = pro.pro_codigo
inner join natureza_operacao no on
nf.nto_codigo = no.nto_codigo
inner join pedidos p on nf.ped_codigo
= p.ped_codigo
inner join clientes c on
nf.cli_codigo = c.cli_codigo
inner join enderecos e on
(c.cli_codigo = e.cli_codigo and
nf_end_cod_entrega = e.end_codigo)
inner join atacadao a on p.ata_codigo
= a.ata_codigo
inner join municipios m on
e.mun_codigo = m.mun_codigo
where nf_dataemissao between
'01.01.2013' and '31.12.2013'
and nto_nomesaida = 'VENDAS'
group by mun_nome, m.mun_codigo,
pro_descricao, pro.pro_codigo
order by mun_nome, m.mun_codigo,
pro_descricao, pro.pro_codigo
```

Resultados obtidos:

Tabela 1. Retorno do SQL.

CodMun	Municipio	CodPro	Produto	Vendido
308	São Paulo	1	Prego	798
308	São Paulo	2	Rosca	1390
108	Brasília	1	Prego	4343

Com o uso do comando *Explain Analyze*, há estatísticas sobre qual o caminho percorrido pelo otimizador do SGBD e os custos utilizados para chegar ao resultado, a Figura 7 mostra o resultado do comando sobre a SQL.

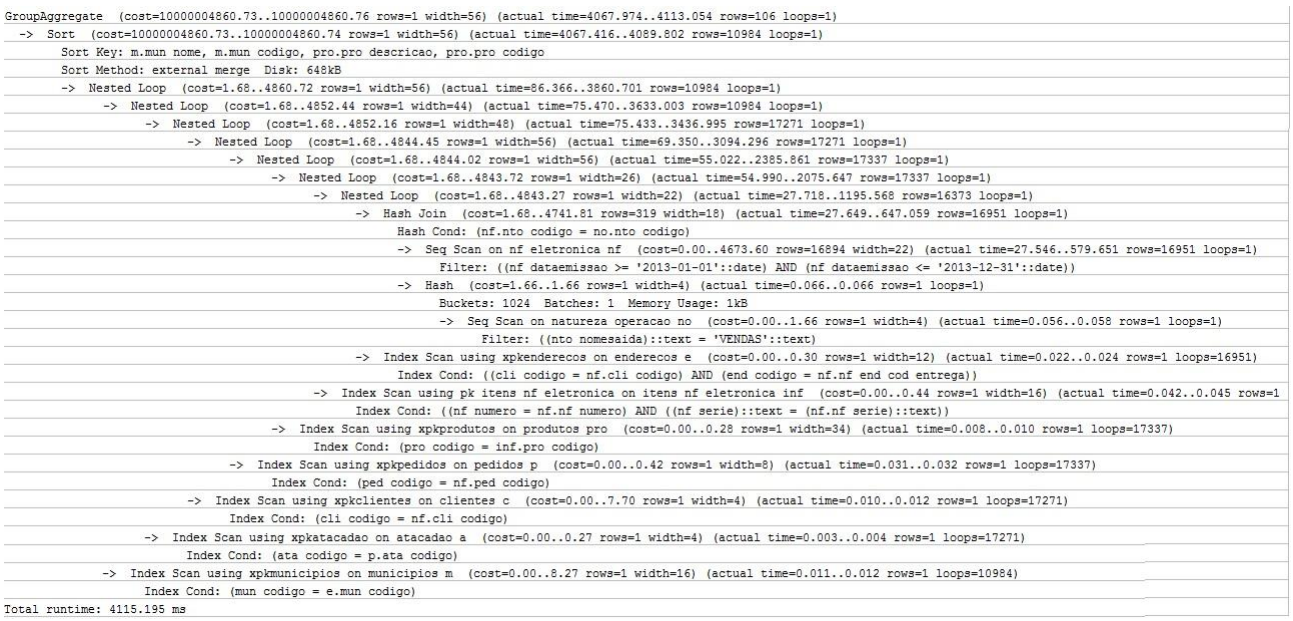


Figura 7. Caminho realizado pelo otimizador.

Nesta consulta o otimizador utilizou programação dinâmica, pois a quantidade de relações (tabelas) esta abaixo de doze (valor padrão), para forçar a utilização do algoritmo genético, é necessária a alteração da propriedade geqo_threshold para um valor abaixo da quantidade de relação utilizada no SQL: set geqo_threshold = 5; e para o SGBD encontrar caminhos alternativos, é necessária a alteração da propriedade seed: set geqo_seed = 0.

A Figura 8 mostra os resultados obtidos do comando Explain Analyze após as alterações para funcionar com algoritmo genético e encontrar caminhos alternativos.

```
GroupAggregate  (cost=10000004890.69..10000004890.72 rows=1 width=56) (actual time=1748.727..1794.081 rows=106 loops=1)
  ->  Sort  (cost=10000004890.69..10000004890.69 rows=1 width=56) (actual time=1748.203..1770.756 rows=10984 loops=1)
        Sort Key: m.mun_nome, m.mun_codigo, pro.pro_descricao, pro.pro_codigo
        Sort Method: external merge  Disk: 648kB
        ->  Nested Loop  (cost=1.68..4890.68 rows=1 width=56) (actual time=1.653..1540.856 rows=10984 loops=1)
              ->  Nested Loop  (cost=1.68..4882.40 rows=1 width=44) (actual time=1.637..1386.111 rows=10984 loops=1)
                    ->  Nested Loop  (cost=1.68..4874.12 rows=1 width=14) (actual time=1.623..1236.430 rows=10984 loops=1)
                          ->  Nested Loop  (cost=1.68..4865.83 rows=1 width=10) (actual time=1.604..1075.578 rows=10928 loops=1)
                                ->  Nested Loop  (cost=1.68..4859.26 rows=1 width=14) (actual time=1.589..887.602 rows=16313 loops=1)
                                      ->  Nested Loop  (cost=1.68..4850.98 rows=1 width=14) (actual time=1.571..651.477 rows=16373 loops=1)
                                            ->  Nested Loop  (cost=1.68..4843.27 rows=1 width=22) (actual time=1.555..422.655 rows=16373 loops=1)
                                                  ->  Hash Join  (cost=1.68..4741.81 rows=319 width=18) (actual time=1.529..166.147 rows=16951 loops=1)
                                                        Hash Cond: (nf.nto_codigo = no.nto_codigo)
                                                        ->  Seq Scan on nf_eletronica nf  (cost=0.00..4673.60 rows=16894 width=22) (actual time=1.466..99.106 rows=16951 loops=1)
                                                              Filter: ((nf_dataemissao >= '2013-01-01'::date) AND (nf_dataemissao <= '2013-12-31'::date))
                                                        ->  Hash  (cost=1.66..1.66 rows=1 width=4) (actual time=0.030..0.030 rows=1 loops=1)
                                                              Buckets: 1024  Batches: 1  Memory Usage: 1kB
                                                              ->  Seq Scan on natureza_operacao no  (cost=0.00..1.66 rows=1 width=4) (actual time=0.020..0.021 rows=1 loops=1)
                                                                    Filter: ((nto_nomesaida)::text = 'VENDAS'::text)
                                                  ->  Index Scan using xpkenderecos on enderecos e  (cost=0.00..0.30 rows=1 width=12) (actual time=0.005..0.007 rows=1 loops=16951)
                                                        Index Cond: ((cli_codigo = nf.cli_codigo) AND (end_codigo = nf.nf_end_cod_entrega))
                                            ->  Index Scan using xpkclientes on clientes c  (cost=0.00..7.70 rows=1 width=4) (actual time=0.004..0.006 rows=1 loops=16373)
                                                  Index Cond: (cli_codigo = nf.cli_codigo)
                                      ->  Index Scan using xpkpedidos on pedidos p  (cost=0.00..8.27 rows=1 width=8) (actual time=0.004..0.006 rows=1 loops=16373)
                                            Index Cond: (ped_codigo = nf.ped_codigo)
                                ->  Index Scan using xpkatacadao on atacadao a  (cost=0.00..6.55 rows=1 width=4) (actual time=0.003..0.004 rows=1 loops=16313)
                                      Index Cond: (ata_codigo = p.ata_codigo)
                          ->  Index Scan using pk_itens_nf_eletronica on itens_nf_eletronica inf  (cost=0.00..8.28 rows=1 width=16) (actual time=0.004..0.006 rows=1 loops=10928)
                                Index Cond: ((nf_numero = nf.nf_numero) AND ((nf_serie)::text = (nf.nf_serie)::text))
                    ->  Index Scan using xpkprodutos on produtos pro  (cost=0.00..8.27 rows=1 width=34) (actual time=0.004..0.005 rows=1 loops=10984)
                          Index Cond: (pro_codigo = inf.pro_codigo)
              ->  Index Scan using xpkmunicipios on municipios m  (cost=0.00..8.27 rows=1 width=16) (actual time=0.004..0.006 rows=1 loops=10984)
                    Index Cond: (mun_codigo = e.mun_codigo)
Total runtime: 1796.217 ms
```

Figura 8. Caminho realizado pelo otimizador.

2) Explain Analyze

```
select m.mun_codigo as CodMun, mun_nome as Municipio, b.ban_codigo as CodBan, ban_razao as Banco, sum(nf_valortotal) as Vlr_Atraso
from nf_eletronica nf inner join natureza_operacao no on nf.nto_codigo = no.nto_codigo
inner join pedidos p on nf.ped_codigo = p.ped_codigo
inner join bancos b on p.ban_codigo = b.ban_codigo
inner join clientes c on nf.cli_codigo = c.cli_codigo
inner join enderecos e on (c.cli_codigo = e.cli_codigo and nf_end_cod_entrega = e.end_codigo)
inner join municipios m on e.mun_codigo = m.mun_codigo
inner join contas_receber cr on p.ped_codigo = cr.ped_codigo
inner join duplicatas_cr dcr on cr.cr_codigo = dcr.cr_codigo
inner join dupcr_faturamento dcrf on (cr.cr_codigo = dcrf.cr_codigo and dcr.cr_codigo = dcrf.cr_codigo and dcr.dcr_codigo = dcrf.dcr_codigo)
inner join faturamentos f on dcrf.fat_codigo = f.fat_codigo
inner join duplicatas_fat df on f.fat_codigo = df.fat_codigo
where dfat_datarecebida is null
and dfat_datavencimento < CURRENT_DATE
and nto_nomesaida = 'VENDAS'
group by mun_nome, m.mun_codigo, ban_razao, b.ban_codigo
order by mun_nome, m.mun_codigo, ban_razao, b.ban_codigo
```

Resultados obtidos:

Tabela 2. Retorno do SQL.

CodMun	Municipio	CodBan	Banco	Vlr_Atraso
308	São Paulo	33	Brasil	1983,00
308	São Paulo	341	Itaú	236,00
108	Brasília	341	Itaú	4343,00

Mantendo as configurações alteradas, o SGBD utiliza algoritmos genéticos no segundo SQL e a Figura 9 mostra o resultado.

```
Sort  (cost=10000005983.38..10000005983.39 rows=1 width=44) (actual time=4218.720..4218.950 rows=177 loops=1)
  Sort Key: m.mun nome, m.mun codigo, b.ban razao, b.ban codigo
  Sort Method: quicksort  Memory: 34kB
  ->  HashAggregate  (cost=5983.36..5983.37 rows=1 width=44) (actual time=4217.068..4217.352 rows=177 loops=1)
        ->  Nested Loop  (cost=5507.67..5983.35 rows=1 width=44) (actual time=2790.549..4194.023 rows=6947 loops=1)
              ->  Nested Loop  (cost=5507.67..5983.04 rows=1 width=52) (actual time=2717.721..3743.851 rows=24292 loops=1)
                    ->  Nested Loop  (cost=5507.67..5982.74 rows=1 width=48) (actual time=2717.705..3275.435 rows=24292 loops=1)
                          ->  Hash Join  (cost=5507.67..5974.45 rows=1 width=60) (actual time=2717.672..2887.060 rows=24292 loops=1)
                                Hash Cond: (dcrf.cr codigo = cr.cr codigo)
                                ->  Seq Scan on dupcr faturamento dcrf  (cost=0.00..375.47 rows=24347 width=12) (actual time=0.014..43.911 rows=24347 loops=1)
                                ->  Hash  (cost=5507.66..5507.66 rows=1 width=48) (actual time=2714.618..2714.618 rows=30363 loops=1)
                                      Buckets: 1024  Batches: 4 (originally 1)  Memory Usage: 1025kB
                                      ->  Nested Loop  (cost=4650.36..5507.66 rows=1 width=48) (actual time=265.073..2637.894 rows=30363 loops=1)
                                            ->  Nested Loop  (cost=4650.36..5507.35 rows=1 width=36) (actual time=265.055..2216.207 rows=30363 loops=1)
                                                  ->  Nested Loop  (cost=4650.36..5500.41 rows=1 width=44) (actual time=265.040..1799.211 rows=30363 loops=1)
                                                        ->  Nested Loop  (cost=4650.36..5500.13 rows=1 width=26) (actual time=265.024..1407.098 rows=30363 loops=1)
                                                              ->  Nested Loop  (cost=4650.36..5499.75 rows=1 width=30) (actual time=264.970..964.258 rows=30363 loops=1)
                                                                    ->  Hash Join  (cost=4650.36..5331.05 rows=555 width=26) (actual time=264.938..510.948 rows=30440 loops=1)
                                                                          Hash Cond: (cr.ped codigo = nf.ped codigo)
                                                                          ->  Seq Scan on contas receber cr  (cost=0.00..560.83 rows=30483 width=8) (actual time=0.008..96.201 rows=30483 loops=1)
                                                                          ->  Hash  (cost=4643.24..4643.24 rows=569 width=18) (actual time=263.508..263.508 rows=29121 loops=1)
                                                                                Buckets: 1024  Batches: 2 (originally 1)  Memory Usage: 1025kB
                                                                                ->  Hash Join  (cost=1.68..4643.24 rows=569 width=18) (actual time=1.239..209.460 rows=30173 loops=1)
                                                                                      Hash Cond: (nf.nto codigo = no.nto codigo)
                                                                                      ->  Seq Scan on nf eletronica nf  (cost=0.00..4522.73 rows=30173 width=22) (actual time=1.173..95.523 rows=30173 loops=1)
                                                                                      ->  Hash  (cost=1.66..1.66 rows=1 width=4) (actual time=0.034..0.034 rows=1 loops=1)
                                                                                            Buckets: 1024  Batches: 1  Memory Usage: 1kB
                                                                                            ->  Seq Scan on natureza operacao no  (cost=0.00..1.66 rows=1 width=4) (actual time=0.024..0.026 rows=1 loops=1)
                                                                                                  Filter: ((nto nomesaida)::text = 'VENDAS'::text)
                                                                    ->  Index Scan using xpkenderecos on enderecos e  (cost=0.00..0.29 rows=1 width=12) (actual time=0.005..0.007 rows=1 loops=30440)
                                                                          Index Cond: ((cli codigo = nf.cli codigo) AND (end codigo = nf.nf end cod entrega))
                                                              ->  Index Scan using xpkpedidos on pedidos p  (cost=0.00..0.36 rows=1 width=8) (actual time=0.005..0.006 rows=1 loops=30363)
                                                                    Index Cond: (ped codigo = nf.ped codigo)
                                                        ->  Index Scan using xpkbancos on bancos b  (cost=0.00..0.27 rows=1 width=22) (actual time=0.003..0.005 rows=1 loops=30363)
                                                              Index Cond: (ban codigo = p.ban codigo)
                                                  ->  Index Scan using xpkclientes on clientes c  (cost=0.00..6.93 rows=1 width=4) (actual time=0.004..0.006 rows=1 loops=30363)
                                                        Index Cond: (cli codigo = nf.cli codigo)
                                            ->  Index Scan using xpkmunicipios on municipios m  (cost=0.00..0.29 rows=1 width=16) (actual time=0.004..0.006 rows=1 loops=30363)
                                                  Index Cond: (mun codigo = e.mun codigo)
                                ->  Index Scan using pk duplicatas cr on duplicatas cr dcr  (cost=0.00..8.27 rows=1 width=8) (actual time=0.006..0.008 rows=1 loops=24292)
                                      Index Cond: ((dcr codigo = dcrf.dcr codigo) AND (cr codigo = cr.cr codigo))
                          ->  Index Scan using pk faturamentos on faturamentos f  (cost=0.00..0.29 rows=1 width=4) (actual time=0.009..0.011 rows=1 loops=24292)
                                Index Cond: (fat codigo = dcrf.fat codigo)
                    ->  Index Scan using pk duplicatas fat on duplicatas fat df  (cost=0.00..0.30 rows=1 width=4) (actual time=0.012..0.012 rows=0 loops=24292)
                          Index Cond: (fat codigo = dcrf.fat codigo)
                          Filter: ((dfat datarecebida IS NULL) AND (dfat datavencimento < ('now'::text)::date))
Total runtime: 4220.839 ms
```

Figura 9. Caminho realizado pelo otimizador.

Para obter os resultados sobre a programação dinâmica, é necessária a alteração da propriedade geqo_threshold: set geqo_threshold = 15. A Figura 10 mostra o resultado.

```
Sort  (cost=10000005983.38..10000005983.39 rows=1 width=44) (actual time=3779.180..3779.407 rows=177 loops=1)
  Sort Key: m.mun nome, m.mun codigo, b.ban razao, b.ban codigo
  Sort Method: quicksort  Memory: 34kB
  ->  HashAggregate  (cost=5983.36..5983.37 rows=1 width=44) (actual time=3777.537..3777.828 rows=177 loops=1)
        ->  Nested Loop  (cost=5507.67..5983.35 rows=1 width=44) (actual time=2633.655..3754.927 rows=6947 loops=1)
              ->  Nested Loop  (cost=5507.67..5983.04 rows=1 width=52) (actual time=2630.053..3467.024 rows=24292 loops=1)
                    ->  Nested Loop  (cost=5507.67..5982.74 rows=1 width=48) (actual time=2630.038..3136.296 rows=24292 loops=1)
                          ->  Hash Join  (cost=5507.67..5974.45 rows=1 width=60) (actual time=2630.004..2787.186 rows=24292 loops=1)
                                Hash Cond: (dcrf.cr codigo = cr.cr codigo)
                                ->  Seq Scan on dupcr faturamento dcrf  (cost=0.00..375.47 rows=24347 width=12) (actual time=0.012..33.970 rows=24347 loops=1)
                                ->  Hash  (cost=5507.66..5507.66 rows=1 width=48) (actual time=2626.898..2626.898 rows=30363 loops=1)
                                      Buckets: 1024  Batches: 4 (originally 1)  Memory Usage: 1025kB
                                      ->  Nested Loop  (cost=4650.36..5507.66 rows=1 width=48) (actual time=263.201..2551.007 rows=30363 loops=1)
                                            ->  Nested Loop  (cost=4650.36..5507.35 rows=1 width=36) (actual time=263.184..2132.640 rows=30363 loops=1)
                                                  ->  Nested Loop  (cost=4650.36..5500.41 rows=1 width=44) (actual time=263.169..1718.103 rows=30363 loops=1)
                                                        ->  Nested Loop  (cost=4650.36..5500.13 rows=1 width=26) (actual time=263.155..1326.654 rows=30363 loops=1)
                                                              ->  Nested Loop  (cost=4650.36..5499.75 rows=1 width=30) (actual time=263.138..906.321 rows=30363 loops=1)
                                                                    ->  Hash Join  (cost=4650.36..5331.05 rows=555 width=26) (actual time=263.107..455.884 rows=30440 loops=1)
                                                                          Hash Cond: (cr.ped codigo = nf.ped codigo)
                                                                          ->  Seq Scan on contas receber cr  (cost=0.00..560.83 rows=30483 width=8) (actual time=0.007..44.005 rows=30483 loops=1)
                                                                          ->  Hash  (cost=4643.24..4643.24 rows=569 width=18) (actual time=262.473..262.473 rows=29121 loops=1)
                                                                                Buckets: 1024  Batches: 2 (originally 1)  Memory Usage: 1025kB
                                                                                ->  Hash Join  (cost=1.68..4643.24 rows=569 width=18) (actual time=1.284..209.236 rows=30173 loops=1)
                                                                                      Hash Cond: (nf.nto codigo = no.nto codigo)
                                                                                      ->  Seq Scan on nf eletronica nf  (cost=0.00..4522.73 rows=30173 width=22) (actual time=1.222..97.024 rows=30173 loops=1)
                                                                                      ->  Hash  (cost=1.66..1.66 rows=1 width=4) (actual time=0.033..0.033 rows=1 loops=1)
                                                                                            Buckets: 1024  Batches: 1  Memory Usage: 1kB
                                                                                            ->  Seq Scan on natureza operacao no  (cost=0.00..1.66 rows=1 width=4) (actual time=0.023..0.025 rows=1 loops=1)
                                                                                                  Filter: ((nto nomesaida)::text = 'VENDAS'::text)
                                                                    ->  Index Scan using xpkenderecos on enderecos e  (cost=0.00..0.29 rows=1 width=12) (actual time=0.005..0.007 rows=1 loops=30440)
                                                                          Index Cond: ((cli codigo = nf.cli codigo) AND (end codigo = nf.nf end cod entrega))
                                                              ->  Index Scan using xpkpedidos on pedidos p  (cost=0.00..0.36 rows=1 width=8) (actual time=0.004..0.006 rows=1 loops=30363)
                                                                    Index Cond: (ped codigo = nf.ped codigo)
                                                        ->  Index Scan using xpkbancos on bancos b  (cost=0.00..0.27 rows=1 width=22) (actual time=0.003..0.005 rows=1 loops=30363)
                                                              Index Cond: (ban codigo = p.ban codigo)
                                                  ->  Index Scan using xpkclientes on clientes c  (cost=0.00..6.93 rows=1 width=4) (actual time=0.004..0.006 rows=1 loops=30363)
                                                        Index Cond: (cli codigo = nf.cli codigo)
                                            ->  Index Scan using xpkmunicipios on municipios m  (cost=0.00..0.29 rows=1 width=16) (actual time=0.004..0.006 rows=1 loops=30363)
                                                  Index Cond: (mun codigo = e.mun codigo)
                          ->  Index Scan using pk duplicatas cr on duplicatas cr dcr  (cost=0.00..8.27 rows=1 width=8) (actual time=0.005..0.006 rows=1 loops=24292)
                                Index Cond: ((dcr codigo = dcrf.dcr codigo) AND (cr codigo = cr.cr codigo))
                    ->  Index Scan using pk faturamentos on faturamentos f  (cost=0.00..0.29 rows=1 width=4) (actual time=0.004..0.005 rows=1 loops=24292)
                          Index Cond: (fat codigo = dcrf.fat codigo)
              ->  Index Scan using pk duplicatas fat on duplicatas fat df  (cost=0.00..0.30 rows=1 width=4) (actual time=0.005..0.006 rows=0 loops=24292)
                    Index Cond: (fat codigo = dcrf.fat codigo)
                    Filter: ((dfat datarecebida IS NULL) AND (dfat datavencimento < ('now'::text)::date))
Total runtime: 3780.343 ms
```

Figura 10. Caminho realizado pelo otimizador.

Para completar os dados para teste, os SQL's serão reescritos de três tipos diferentes: T1 – definição de atributos e junções utilizando *inner join* (conforme já citado); T2 - com definição de atributos e junções na clausula *WHERE*; T3 - sem definição de atributos e junções na clausula *WHERE*. O melhor caso de escrita das consultas é definindo os atributos e utilizando *inner join*, o pior caso é não definindo os atributos e utilizando junções na clausula *WHERE*. A Figura 11 mostra o caminho percorrido pelo otimizador utilizando o algoritmo de programação dinâmica no SQL 1 escrito no pior caso.

```
Nested Loop  (cost=1.68..4852.75 rows=1 width=4221) (actual time=1.626..1887.570 rows=10984 loops=1)
  ->  Nested Loop  (cost=1.68..4852.44 rows=1 width=4180) (actual time=1.608..1719.920 rows=10984 loops=1)
        ->  Nested Loop  (cost=1.68..4852.16 rows=1 width=2919) (actual time=1.590..1504.479 rows=17271 loops=1)
              ->  Nested Loop  (cost=1.68..4844.45 rows=1 width=2715) (actual time=1.571..1234.161 rows=17271 loops=1)
                    ->  Nested Loop  (cost=1.68..4844.02 rows=1 width=2579) (actual time=1.552..967.425 rows=17337 loops=1)
                          ->  Nested Loop  (cost=1.68..4843.72 rows=1 width=2458) (actual time=1.536..710.429 rows=17337 loops=1)
                                ->  Nested Loop  (cost=1.68..4843.27 rows=1 width=2360) (actual time=1.514..449.217 rows=16373 loops=1)
                                      ->  Hash Join  (cost=1.68..4741.81 rows=319 width=2279) (actual time=1.487..181.251 rows=16951 loops=1)
                                            Hash Cond: (nf.nto codigo = no.nto codigo)
                                            ->  Seq Scan on nf eletronica nf  (cost=0.00..4673.60 rows=16894 width=1003) (actual time=1.417..101.131 rows=16951 loops=1)
                                                  Filter: ((nf dataemissao >= '2013-01-01'::date) AND (nf dataemissao <= '2013-12-31'::date))
                                            ->  Hash  (cost=1.66..1.66 rows=1 width=1276) (actual time=0.030..0.030 rows=1 loops=1)
                                                  Buckets: 1024  Batches: 1  Memory Usage: 1kB
                                                  ->  Seq Scan on natureza operacao no  (cost=0.00..1.66 rows=1 width=1276) (actual time=0.020..0.022 rows=1 loops=1)
                                                        Filter: ((nto nomesaida)::text = 'VENDAS'::text)
                                      ->  Index Scan using xpkenderecos on enderecos e  (cost=0.00..0.30 rows=1 width=81) (actual time=0.005..0.007 rows=1 loops=16951)
                                            Index Cond: ((cli codigo = nf.cli codigo) AND (end codigo = nf.nf end cod entrega))
                                ->  Index Scan using pk itens nf eletronica on itens nf eletronica inf  (cost=0.00..0.44 rows=1 width=98) (actual time=0.004..0.006 rows=1 loops=16373)
                                      Index Cond: ((nf numero = nf.nf numero) AND ((nf serie)::text = (nf.nf serie)::text))
                          ->  Index Scan using xpkprodutos on produtos pro  (cost=0.00..0.28 rows=1 width=121) (actual time=0.004..0.005 rows=1 loops=17337)
                                Index Cond: (pro codigo = inf.pro codigo)
                    ->  Index Scan using xpkpedidos on pedidos p  (cost=0.00..0.42 rows=1 width=136) (actual time=0.004..0.006 rows=1 loops=17337)
                          Index Cond: (ped codigo = nf.ped codigo)
              ->  Index Scan using xpkclientes on clientes c  (cost=0.00..7.70 rows=1 width=204) (actual time=0.004..0.006 rows=1 loops=17271)
                    Index Cond: (cli codigo = nf.cli codigo)
        ->  Index Scan using xpkatacadao on atacadao a  (cost=0.00..0.27 rows=1 width=1261) (actual time=0.003..0.004 rows=1 loops=17271)
              Index Cond: (ata codigo = p.ata codigo)
  ->  Index Scan using xpkmunicipios on municipios m  (cost=0.00..0.29 rows=1 width=41) (actual time=0.004..0.006 rows=1 loops=10984)
        Index Cond: (mun codigo = e.mun codigo)
Total runtime: 1903.298 ms
```

Figura 11. Caminho realizado pelo otimizador.

Para utilizar o algoritmo genético a propriedade geqo_threshold precisa ser alterada: set geqo_threshold = 5. A Figura 12 mostra o novo caminho percorrido pelo otimizador após a nova configuração da propriedade geqo_threshold na execução da consulta.

```
Nested Loop  (cost=1.68..4891.82 rows=1 width=4221) (actual time=1.468..1738.365 rows=10984 loops=1)
  ->  Nested Loop  (cost=1.68..4883.54 rows=1 width=4100) (actual time=1.451..1569.345 rows=10984 loops=1)
        ->  Nested Loop  (cost=1.68..4875.25 rows=1 width=4002) (actual time=1.430..1391.380 rows=10928 loops=1)
              ->  Nested Loop  (cost=1.68..4866.97 rows=1 width=2741) (actual time=1.412..1186.295 rows=16313 loops=1)
                    ->  Nested Loop  (cost=1.68..4858.69 rows=1 width=2605) (actual time=1.394..934.814 rows=16373 loops=1)
                          ->  Nested Loop  (cost=1.68..4850.98 rows=1 width=2401) (actual time=1.375..685.615 rows=16373 loops=1)
                                ->  Nested Loop  (cost=1.68..4843.27 rows=1 width=2360) (actual time=1.358..446.806 rows=16373 loops=1)
                                      ->  Hash Join  (cost=1.68..4741.81 rows=319 width=2279) (actual time=1.331..179.186 rows=16951 loops=1)
                                            Hash Cond: (nf.nto codigo = no.nto codigo)
                                            ->  Seq Scan on nf eletronica nf  (cost=0.00..4673.60 rows=16894 width=1003) (actual time=1.269..98.836 rows=16951 loops=1)
                                                  Filter: ((nf dataemissao >= '2013-01-01'::date) AND (nf dataemissao <= '2013-12-31'::date))
                                            ->  Hash  (cost=1.66..1.66 rows=1 width=1276) (actual time=0.029..0.029 rows=1 loops=1)
                                                  Buckets: 1024  Batches: 1  Memory Usage: 1kB
                                                  ->  Seq Scan on natureza operacao no  (cost=0.00..1.66 rows=1 width=1276) (actual time=0.020..0.021 rows=1 loops=1)
                                                        Filter: ((nto nomesaida)::text = 'VENDAS'::text)
                                      ->  Index Scan using xpkenderecos on enderecos e  (cost=0.00..0.30 rows=1 width=81) (actual time=0.005..0.007 rows=1 loops=16951)
                                            Index Cond: ((cli codigo = nf.cli codigo) AND (end codigo = nf.nf end cod entrega))
                                ->  Index Scan using xpkmunicipios on municipios m  (cost=0.00..7.70 rows=1 width=41) (actual time=0.004..0.006 rows=1 loops=16373)
                                      Index Cond: (mun codigo = e.mun codigo)
                          ->  Index Scan using xpkclientes on clientes c  (cost=0.00..7.70 rows=1 width=204) (actual time=0.004..0.006 rows=1 loops=16373)
                                Index Cond: (cli codigo = nf.cli codigo)
                    ->  Index Scan using xpkpedidos on pedidos p  (cost=0.00..8.27 rows=1 width=136) (actual time=0.004..0.006 rows=1 loops=16373)
                          Index Cond: (ped codigo = nf.ped codigo)
              ->  Index Scan using xpkatacadao on atacadao a  (cost=0.00..8.27 rows=1 width=1261) (actual time=0.003..0.004 rows=1 loops=16313)
                    Index Cond: (ata codigo = p.ata codigo)
        ->  Index Scan using pk itens nf eletronica on itens nf eletronica inf  (cost=0.00..8.28 rows=1 width=98) (actual time=0.004..0.006 rows=1 loops=10928)
              Index Cond: ((nf numero = nf.nf numero) AND ((nf serie)::text = (nf.nf serie)::text))
  ->  Index Scan using xpkprodutos on produtos pro  (cost=0.00..8.27 rows=1 width=121) (actual time=0.004..0.005 rows=1 loops=10984)
        Index Cond: (pro codigo = inf.pro codigo)
Total runtime: 1754.040 ms
```

Figura 12. Caminho realizado pelo otimizador.

Mantendo as configurações alteradas, a Figura 13 mostra o caminho utilizado pelo otimizador na SQL 2 com algoritmo genético.

```
Nested Loop  (cost=4811.36..6537.03 rows=1 width=3045) (actual time=597.391..43926.994 rows=6947 loops=1)
  ->  Nested Loop  (cost=4811.36..6530.33 rows=1 width=2999) (actual time=597.372..43796.650 rows=6947 loops=1)
        ->  Nested Loop  (cost=4811.36..6522.05 rows=1 width=2905) (actual time=597.353..43669.583 rows=6947 loops=1)
              ->  Nested Loop  (cost=4811.36..6513.77 rows=1 width=2701) (actual time=597.334..43525.059 rows=6947 loops=1)
                    ->  Nested Loop  (cost=4811.36..6505.48 rows=1 width=2625) (actual time=488.338..43039.346 rows=24292 loops=1)
                          ->  Nested Loop  (cost=4811.36..6322.57 rows=1 width=2613) (actual time=455.690..16982.287 rows=30363 loops=1)
                                ->  Nested Loop  (cost=4811.36..6314.29 rows=1 width=2572) (actual time=455.672..16344.849 rows=30363 loops=1)
                                      Join Filter: (b.ban codigo = p.ban codigo)
                                      ->  Nested Loop  (cost=4811.36..6308.73 rows=1 width=2530) (actual time=455.630..2351.160 rows=30363 loops=1)
                                            ->  Nested Loop  (cost=4811.36..6300.45 rows=1 width=2394) (actual time=455.609..1738.569 rows=30363 loops=1)
                                                  ->  Hash Join  (cost=4811.36..6131.05 rows=555 width=2313) (actual time=455.574..1035.660 rows=30440 loops=1)
                                                        Hash Cond: (cr.ped codigo = nf.ped codigo)
                                                        ->  Seq Scan on contas receber cr  (cost=0.00..560.83 rows=30483 width=34) (actual time=0.014..43.224 rows=30483 loops=1)
                                                        ->  Hash  (cost=4643.24..4643.24 rows=569 width=2279) (actual time=447.727..447.727 rows=29121 loops=1)
                                                              Buckets: 1024  Batches: 32 (originally 2)  Memory Usage: 1025kB
                                                              ->  Hash Join  (cost=1.68..4643.24 rows=569 width=2279) (actual time=1.389..237.353 rows=30173 loops=1)
                                                                    Hash Cond: (nf.nto codigo = no.nto codigo)
                                                                    ->  Seq Scan on nf eletronica nf  (cost=0.00..4522.73 rows=30173 width=1003) (actual time=1.313..100.967 rows=30173 loops=1)
                                                                    ->  Hash  (cost=1.66..1.66 rows=1 width=1276) (actual time=0.037..0.037 rows=1 loops=1)
                                                                          Buckets: 1024  Batches: 1  Memory Usage: 1kB
                                                                          ->  Seq Scan on natureza operacao no  (cost=0.00..1.66 rows=1 width=1276) (actual time=0.027..0.029 rows=1 loops=1)
                                                                                Filter: ((nto nomesaida)::text = 'VENDAS'::text)
                                                  ->  Index Scan using xpkenderecos on enderecos e  (cost=0.00..0.29 rows=1 width=81) (actual time=0.010..0.012 rows=1 loops=30440)
                                                        Index Cond: ((cli codigo = nf.cli codigo) AND (end codigo = nf.nf end cod entrega))
                                            ->  Index Scan using xpkpedidos on pedidos p  (cost=0.00..8.27 rows=1 width=136) (actual time=0.007..0.009 rows=1 loops=30363)
                                                  Index Cond: (ped codigo = nf.ped codigo)
                                      ->  Seq Scan on bancos b  (cost=0.00..3.58 rows=158 width=42) (actual time=0.007..0.219 rows=158 loops=30363)
                                ->  Index Scan using xpkmunicipios on municipios m  (cost=0.00..8.27 rows=1 width=41) (actual time=0.008..0.010 rows=1 loops=30363)
                                      Index Cond: (mun codigo = e.mun codigo)
                          ->  Index Scan using pk dupcr faturamento on dupcr faturamento dcrf  (cost=0.00..182.90 rows=1 width=12) (actual time=0.512..0.848 rows=1 loops=30363)
                                Index Cond: (cr codigo = cr.cr codigo)
                    ->  Index Scan using pk duplicatas fat on duplicatas fat df  (cost=0.00..8.28 rows=1 width=76) (actual time=0.011..0.012 rows=0 loops=24292)
                          Index Cond: (fat codigo = dcrf.fat codigo)
                          Filter: ((dfat datarecebida IS NULL) AND (dfat datavencimento < ('now'::text)::date))
              ->  Index Scan using xpkclientes on clientes c  (cost=0.00..8.27 rows=1 width=204) (actual time=0.007..0.009 rows=1 loops=6947)
                    Index Cond: (cli codigo = nf.cli codigo)
        ->  Index Scan using pk faturamentos on faturamentos f  (cost=0.00..8.27 rows=1 width=94) (actual time=0.006..0.007 rows=1 loops=6947)
              Index Cond: (fat codigo = dcrf.fat codigo)
  ->  Index Scan using pk duplicatas cr on duplicatas cr dcr  (cost=0.00..6.69 rows=1 width=46) (actual time=0.006..0.008 rows=1 loops=6947)
        Index Cond: ((dcr codigo = dcrf.dcr codigo) AND (cr codigo = cr.cr codigo))
Total runtime: 43940.106 ms
```

Figura 13. Caminho realizado pelo otimizador.

Alterando o valor da propriedade geqo_threshold para quinze, o otimizador utilizará programação dinâmica. A Figura 14 mostra o caminho percorrido após realizar a mudança do parâmetro mencionado.

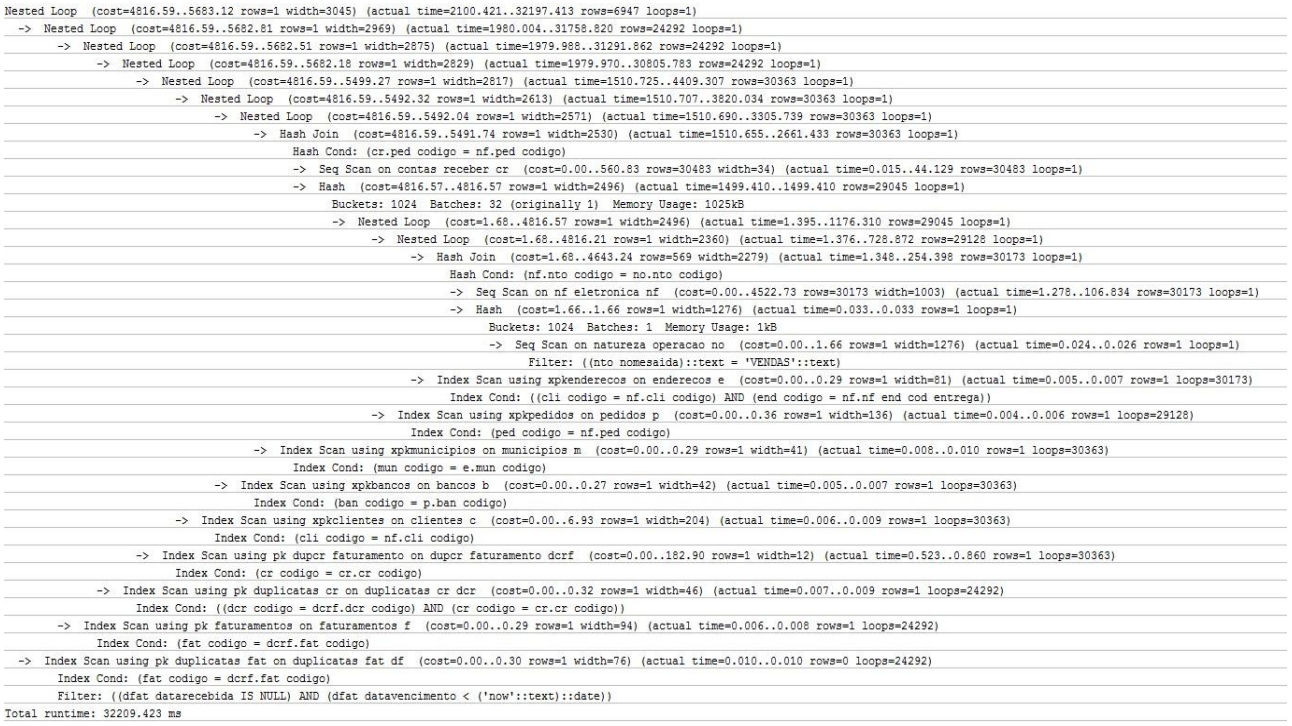


Figura 14. Caminho realizado pelo otimizador.

Para os resultados, os SQL's forão testados três vezes.

5.4 RESULTADOS

Os resultados apresentados nesta seção foram obtidos utilizando as informações geradas e exibidas pelo próprio POSTGRESQL usando o método *Explain Analyze.* Durante a execução dos comandos SQL mencionados na seção anterior foi possível, como pode ser verificado no topo das Figuras 7 a 14, coletar as informações sobre: total de linhas retornadas pela consulta, tempo em milissegundos durante cada execução realizada.

Tabela 3. Informações do *Explain Analyze* para o SQL T1.

SQL	Qtde Tabela	Valor THRESHOLD	Algoritmo Utilizado	Linhas Retornadas	Tempo (1) Milissegundos	Tempo (2) Milissegundos	Tempo (3) Milissegundos
1	9	12	Prog. Dinâmica	106	2002,590	2030,474	2006,916
1	9	5	Genético	106	1645,502	1701,102	1668,291
2	12	12	Genético	177	45487,737	43286,379	43221,685
2	12	15	Prog. Dinâmica	177	29978,350	30305,806	30170,172

De acordo com os resultados obtidos do SQL T1 (Tabela 3), os resultados são opostos ao padrão utilizado no PostgreSQL. No SQL 1 o algoritmo genético realizou a

otimização de forma mais rápida que a programação dinâmica. No SQL 2 o algoritmo de programação dinâmica otimizou com melhor performance do que o algoritmo genético. Em ambos os casos o comportamento inverso era esperado.

Tabela 4. Informações do *Explain Analyze* para o SQL T2.

SQL	Qtde Tabela	Valor THRESHOLD	Algoritmo Utilizado	Linhas Retornadas	Tempo (1) Milissegundos	Tempo (2) Milissegundos	Tempo (3) Milissegundos
1	9	12	Prog. Dinâmica	10984	1914,018	1946,426	1910,447
1	9	5	Genético	10984	1757,459	1781,514	1757,772
2	12	12	Genético	6943	44090,640	43981,270	44217,926
2	12	15	Prog. Dinâmica	6943	31812,234	31714,400	32759,552

Na situação demonstrada pela Tabela 4 o resultado é idêntico ao SQL T1, sendo opostos ao padrão utilizado no PostgreSQL. No SQL 1, o algoritmo genético otimizou em menor tempo, no SQL 2 a programação dinâmica otimizou em menor tempo.

Tabela 5. Informações do *Explain Analyze* para o SQL T3.

SQL	Qtde Tabela	Valor THRESHOLD	Algoritmo Utilizado	Linhas Retornadas	Tempo (1) Milissegundos	Tempo (2) Milissegundos	Tempo (3) Milissegundos
1	9	12	Prog. Dinâmica	106	2080,651	2050,515	2056,344
1	9	5	Genético	106	1807,696	1858,763	1801,759
2	12	12	Genético	177	3809,791	3798,437	3796,488
2	12	15	Prog. Dinâmica	177	3790,234	3798,577	3807,362

Segundo os resultados apresentados na Tabela 5 no SQL 1 o algoritmo genético otimizou em menor tempo, e no SQL 2 a otimização obteve uma diferença insignificante, em alguns momentos o genético otimizou mais rápido e em outros momentos a programação dinâmica otimizou com mais eficiência.

6 CONSIDERAÇÕES FINAIS

O presente artigo apresentou uma análise dos algoritmos de otimização de consultas do sistema de gerenciamento de banco de dados PostgreSQL, detalhando o uso dos algoritmos genéticos e de programação dinâmica. Os algoritmos genéticos são utilizados pelo SGBD quando a consulta SQL possui doze ou mais tabelas, caso contrário, o SGBD utiliza a programação dinâmica para otimizar a consulta. Com as informações adquiridas com o método *Explain Analyze,* utilizando um banco de dados real e SQL's do cotidiano empresarial, é possível concluir que para nas consultas com nove tabelas o algoritmo genético otimiza com mais eficiência, já nas consultas com doze ou mais tabelas a programação dinâmica otimiza com mais eficiência (com exceção da consulta utilizando *inner join*, pois

obteve uma diferença insignificante). Sendo assim, o padrão de uso dos algoritmos de otimização estabelecidos pelo PostgreSQL não demonstrou nesta situação de teste ser o ideal.

Para obter melhores resultados, como trabalhos futuros, o comparativo pode ser realizado com base no uso do processamento de memória em cada otimização, e no aumento da diversidade e na complexidade de consultas analisadas.

REFERÊNCIAS

CARVALHO SANTOS, R; Estudo Comparativo dos Sistemas Gerenciadores de Bancos de Dados: Oracle, SQL Server e PostgreSQL. 15 f.

COSTA, R. L. C. SQL: Guia Prático, 2. ed. Rio de Janeiro: Brasport, 2007.

ELMASRI, R.; Sistemas de banco de dados / Ramez Thatyana e Shamkant B. Navathe, 6. Ed. São Paulo: Person Addison Wesley, 2010.

GÓES PEDROZO, W. Arquitetura para Seleção de Índice no SGBD PostgreSQL, utilizando abordagem baseada em custos do Otimizador. 2008. 77 f. Dissertação (Mestrado em Ciência Exatas) Universidade Federal do Paraná, Curitiba.

GUTTOSKI, P. B. Otimização de Consulta no Postgresql Utilizando o Algoritmo de Kruskal. 2006. 95 f. Dissertação (Mestrado). Unidade Federal do Paraná. Curitiba.

LANGE, A. Uma Avaliação de Algoritmos Não Exaustivos para a Otimização de Junções. 2010. 98 f. Dissertação (Mestrado) – Universidade Federal do Paraná, Curitiba.

MARTINS COLARES, F. Análise Comparativa de Banco de Dados Gratuitos. 2007. 74 f. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) – Faculdade Lourenço Filho, Fortaleza.

MORAES ARCANJO, E. A.; NUNES DE LIMA, I. Otimizador de Consultas em Banco de Dados Estudo de Caso: Microsoft SQL Server 2008 R2. Disponível em: <http://blog.newtonpaiva.br/pos/e5i18-otimizador-de-consultas-em-banco-de-dados-estudo-de-caso-microsoft-sql-server-2008-r2/>. Acesso em: 13 jun. 2012.

PRICE, J. Oracle Database 11g SQL, 1. ed. Porto Alegre: ARTMED, 2008.

RAMARKRISHNAN, R.;GEHRKE, J. Sistemas de gerenciamento de banco de dados, 3. ed. Porto Alegre: AMGH, 2011.

SAMPAIO, B.; GOVEIA, J.; MARQUES, P. Relatório de comparação de SGBDs. 2011. 58 f. Dissertação (Mestrado em Engenharia Informática). Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Lisboa.

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. Sistema de Banco de Dados. 5. ed. Campus, 2006.

SOUZA, A. C.; AMARAL, H. R.; LIZARD, L. E. O. PostgreSQL: uma Alternativa para Sistemas Gerenciadores de Banco de Dados de Código Aberto. Universidade Federal de Minas Gerais. Belo Horizonte, 2011.

The Postgre Global Development Group. Documentação do PostgreSQL 9.3. Disponível em: <http://www.postgresql.org/docs/9.3/interactive/index.html>. Acesso em: 01 maio 2014.