TARCIZIO ALEXANDRE BINI

# APLICAÇÃO DO ALGORITMO DE KRUSKAL NA OTIMIZAÇÃO DE CONSULTAS COM MÚLTIPLAS JUNÇÕES RELACIONAIS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Marcos Sfair Sunye.

CURITIBA

2009

TARCIZIO ALEXANDRE BINI

# APLICAÇÃO DO ALGORITMO DE KRUSKAL NA OTIMIZAÇÃO DE CONSULTAS COM MÚLTIPLAS JUNÇÕES RELACIONAIS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Marcos Sfair Sunye.

CURITIBA

2009

TARCIZIO ALEXANDRE BINI

# APLICAÇÃO DO ALGORITMO DE KRUSKAL NA OTIMIZAÇÃO DE CONSULTAS COM MÚLTIPLAS JUNÇÕES RELACIONAIS

Dissertação aprovada como requisito parcial à obtenção do grau de Mestre no Programa de Pós-Graduação em Informática da Universidade Federal do Paraná, pela Comisso formada pelos professores:

Orientador: Prof. Dr. Marcos Sfair Sunye.
Departamento de Informática, UFPR

Prof. Dr. Fabiano Silva
Departamento de Informática, UFPR, membro interno.

Prof. Dr. Eduardo Cunha de Almeida
Université de Nantes, UN, França, membro externo.

Curitiba, 17 de abril de 2009

# AGRADECIMENTOS

Agradeço primeiramente a Deus, por ter me dado a oportunidade, forças e determinação para a realização e conclusão do curso de mestrado.

A todos os professores do Departamento de Informática da Universidade Federal do Paraná, que trabalham constantemente pela qualidade do ensino oferecido pela instituição. Especialmente agradeço ao meu orientador, professor Marcos Sfair Sunye, por acreditar em meu potencial e sempre estar disponível às orientações, dando dicas e sugestões para o melhor desenvolvimento do trabalho.

Aos familiares, de maneira especial meus pais, Angelo e Sofia, que me apoiaram nos momentos de maiores dificuldades. Palavras amigas, revigorantes ou simplesmente a torcida pelo meu sucesso, nunca faltaram. Sou imensamente grato.

Ao amigo Adriano Lange pela grande ajuda, paciência e também os diversos esclarecimentos e os conhecimentos repassados. Obrigado também pela configuração dos equipamentos utilizados e contribuições na correção deste documento.

A empresa DHL tratores, na pessoa do Sr. Carlos Cesar Alves, que proporcionou condições para que eu pudesse realizar o curso de mestrado. Em especial aos amigos Deivison e Reginaldo pelo bom-humor e momentos de descontração muito importantes para mim.

A colega Pryscila Barvick Guttoski, que na medida do possível procurou repassar seus conhecimentos a respeito dos assuntos pertinentes a esse trabalho, o qual foi iniciado por ela.

Aos integrantes do projeto GHI, que partilharam dia-a-dia, das dificuldades que encontrei e dos êxitos alcançados.

Enfim, agradeço a todos aqueles que de uma forma ou outra, muitas vezes sem saber, contribuíram para a realização deste trabalho.

# SUMÁRIO

# LISTA DE FIGURAS

# LISTA DE TABELAS

# RESUMO

A busca por planos ótimos de execução de consultas em SGDBs relacionais é certamente um problema da classe NP. Em virtude disso, a aplicação de algoritmos de programação dinâmica para esta finalidade fica restrita a certo limite de relações. Dessa forma, vários algoritmos foram propostos na tentativa de se encontrar planos aceitáveis em tempo hábil e com baixo consumo de recursos. Dentre estas soluções, podemos citar os algoritmos genéticos que apresentam a solução para o problema de otimização de consultas em tempo polinomial. Porém, por se tratar de um método aleatório, que exibe muitas variações em seus resultados, planos de execução impraticáveis podem ser escolhidos. Neste trabalho apresentaremos o algoritmo de Kruskal em conjunto com algumas regras de geração de sub-planos como alternativa para a geração de planos de execução de consultas. Tal algoritmo apresenta vantagens como código de implementação simples, tempo de execução polinomial e espaço de busca reduzido. Nós implementamos o algoritmo de Kruskal no SGBD PostgreSQL, o que permitiu confrontar seus resultados com o algoritmo de programação dinâmica em consultas simples ou algoritmos genéticos em consultas complexas. Para os testes de performance, nossa metodologia de avaliação, tomou por base os *benchmarks* TPC-H e TPC-E. Os resultados obtidos demonstram que o algoritmo de Kruskal aplicado a otimização de consultas é uma solução viável que exibe bons resultados em consultas que apresentam múltiplas junções.

# ABSTRACT

The search for optimal plans in order to execute queries in relational DBMS is certainly an NP-complete problem. In virtue of this, the applicability of dynamic programming algorithms to the search of the optimal plans is restricted to a certain limit of relations. Because of this problem, severals alternative algorithms were proposed in order to find satisfactory plans in an acceptable time and with low consumption of resources. Among these solutions, we can cite the genetic algorithms which that present the solution to the queries optimization problem in polynomial time. However, it is a random method, which displays many variations in their results, and impractical execution plans can be chosen. This document presents the Kruskal's algorithm together with some rules for generation of sub-plans as an alternative to the query plan generation. This algorithm has advantages as simple code implementation, polynomial run-time and reduced search space. We implemented this algorithm in PostgreSQL DBMS, which allowed compare their results with the dynamic programming in simple queries, or the genetic algorithm in complex queries. In performance tests, our evaluation method based on benchmarks TPC-H and TPC-E. The results show that Kruskal's algorithm applied to the queries optimization is a viable solution that shows good results for queries that have multiple joins.

# CAPÍTULO 1

# INTRODUÇÃO

Nas últimas décadas tornou-se fundamental o armazenamento, manipulação e recuperação de um número cada vez maior de informações. Em resposta a tais necessidades, surgiram os Sistemas Gerenciadores de Banco de Dados (SGBDs), os quais são aplicações que proporcionam mecanismos para gerir de forma eficiente e conveniente grande volume de informações.

Os primeiros modelos de banco de dados a surgirem foram o hierárquico e o de redes, nos quais os dados são representados por uma coleção de registros e os relacionamentos entre eles são representados por *links* (ponteiros) [24]. Atualmente o mercado é dominado por SGBDs que se baseiam no modelo relacional. Como característica marcante de tais aplicações, temos o armazenamento dos dados em tabelas de nomes únicos, também chamadas de relações.

As Linguagens de Manipulação de Dados (DMLs) viabilizam o acesso e manipulação dos dados, de acordo com o modelo de dados apropriado. As DMLs procedurais exigem que o usuário especifique quais dados são necessários e a forma de obtê-los. Logo as DMLs não procedurais permitem que o usuário especifique apenas quais dados são necessários.

Uma consulta é uma solicitação para recuperação de informações. A parte de uma DML responsável pela recuperação das informações solicitadas é chamada de linguagem de consultas (*query language*) [24].

A *Structured Query Language* (SQL) é um exemplo de DML não procedural empregada nos SGBDs relacionais. Além de ser classificada como "linguagem de consulta" o SQL fornece uma série de recursos como formas de definir e modificar a estrutura do banco de dados além de especificar restrições de segurança.

Devido ao seu alto grau de abstração, simplicidade e facilidade de uso, o SQL oculta diversos detalhes de baixo nível. Dessa forma, uma simples consulta, pode ser decomposta

em uma seqüência de operações físicas baseadas na álgebra relacional, onde cada seqüência apresenta variações de custos computacionais, como processamento e acesso a disco. Esta seqüência de operações pode ser representada em uma estrutura de árvore, chamada árvore de execução da consulta, a qual apresenta a ordem das operações a serem executadas.

Um plano de execução é uma seqüência de passos seguidos pelo Sistema Gerenciador de Banco de Dados para a execução de uma consulta, determinando assim como os dados armazenados nas relações serão acessados. Quanto maior o número de relações e junções referenciadas na consulta, maior a dificuldade da escolha do melhor plano de execução, devido à quantidade destes, aumentar exponencialmente. O processo de escolha do plano de execução mais eficiente para determinada consulta é chamado de otimização de consultas [21].

## 1.1 Motivação

Quando falamos do processo de otimização de consultas, devemos ter em mente que seu alvo principal é a operação de junção, a qual representa alto custo computacional. Os SGBDs podem se utilizar de diversos algoritmos para determinar qual a melhor ordem de se realizar as junções, referenciadas pela consulta. Entre eles podemos citar os algoritmos de programação dinâmica, heurísticos e aleatórios.

Os algoritmos de programação dinâmica são um dos métodos mais tradicionais de otimização. Sua técnica de buscas exaustivas gera todas as possíveis árvores de execução para determinada consulta. Baseado nas estimativas de custos, as árvores que apresentam maiores custos não são incluídas no plano de execução final. Como resultado, tem-se a árvore final de menor custo, considerada a ótima, sendo utilizada para recuperar as informações solicitadas pela *query*.

Os algoritmos de programação dinâmica irão apontar a solução ótima, ou seja, a árvore de execução que apresentar menor estimativa de custos. Porém, quando o número de relações envolvidas na consulta é elevado, o custo computacional necessário ao armazenamento dos diversos sub-planos e processamento dos mesmos, em busca do ótimo, pode tornar-se impraticável.

Atualmente é bastante comum em aplicações gerenciais, a utilização de consultas complexas às bases para a extração de informações estratégicas. Tais consultas certamente envolvem um grande número de relações. Sendo assim, torna-se interessante o estudo de algoritmos que permitam a otimização de consultas complexas, em um período de tempo viável e que indiquem como resultado ao menos uma solução próxima a ótima.

Pesquisas relacionadas a otimização de consultas que envolvem múltiplas junções estão recebendo atenção especial por parte de desenvolvedores de SGBDs e projetistas de banco de dados. Dentre elas podemos citar a incorporação do algoritmo de Kruskal na geração de planos de execução de consultas, proposta esta realizada por Pryscila Barvick Guttoski em [21]. O algoritmo de Kruskal é um algoritmo guloso utilizado na teoria de grafos para encontrar árvores geradoras mínimas, formadas pelo conjunto de arestas que contém todos os vértices do grafo e cujo peso total das arestas é minimizado.

A proposta de otimização de consultas que utiliza o algoritmo de Kruskal, foi implantada no SGBD PostgreSQL [20] devido a sua vasta documentação, código fonte aberto e capacidade para o tratamento de consultas complexas. Basicamente tal SGBD utiliza-se de duas técnicas para a geração dos planos de execução, sendo elas uma implementação de um algoritmo de programação dinâmica, neste documento chamado de Programação Dinâmica e o módulo de algoritmos genéticos GEQO (*Genetic Query Optimization*).

O módulo GEQO é acionado por padrão em consultas que envolvem mais de 11 relações na cláusula from, ou seja, o mesmo é empregado para a otimização de consultas que envolvem múltiplas junções. Porém, por se tratar de um método de otimização aleatório o mesmo pode escolher e manter planos de execução impraticáveis, por isso seu desempenho não é considerado suficiente.

Outra problemática referente a este otimizador diz respeito ao tempo de execução da consulta e apresentação dos resultados. Não existe regularidade no período de tempo necessário para o retorno das informações solicitadas pela consulta. Certa execução de determinada *query* pode apresentar resultados em um período de tempo baixo. Em outro caso, tratando-se da mesma consulta, em um período de tempo extremamente alto. Este problema torna-se mais crítico a medida que o número de relações referenciadas pelas

consultas aumentam.

Este trabalho descreve o processo de avaliação da performance do algoritmo de Kruskal empregado no processo de otimização de consultas. Os testes desenvolvidos levam em consideração *queries* simples que envolvem um número baixo de relações e também consultas complexas com múltiplos relacionamentos. Sendo assim, desenvolvemos uma metodologia própria para realização de tal avaliação, a qual se baseia em *benchmarks* fornecidos pelo TPC (*Transaction Processing Council*) [28]. O desenvolvimento de tal metodologia foi necessária devido a ausência, nos *benchmarks* tradicionais, de consultas que envolvem um número muito elevado de relações. Além disso, tais ferramentas buscam avaliar a performance do SGBD como um todo, não apresentando resultados característicos para análise dos algoritmos otimizadores, como tempo de execução e custos estimados para os plano de execução gerados.

## 1.2 Organização da Dissertação

Este documento está dividido em seis capítulos: No Capítulo 2 é apresentada uma revisão a respeito dos principais *benchmarks* relacionados à área de banco de dados, servindo de embasamento teórico para a melhor compreensão do leitor, a respeito das ferramentas de avaliação de desempenho utilizadas em nossos experimentos. No Capítulo 3 apresentamos os principais algoritmos para a realização de junções relacionais. O Capitulo 4 descreve o SGBD PostgreSQL, dando atenção especial aos algoritmos de otimização de consultas e também a implementação do algoritmo de Kruskal neste SGBD. O Capítulo 5 apresenta a descrição da metodologia para avaliação de desempenho dos algoritmos otimizadores e também descreve os resultados obtidos pelos experimentos. Finalmente o Capítulo 6 apresenta as conclusões obtidas com a elaboração deste trabalho.

# CAPÍTULO 2

# BENCHMARK

Existe uma grande quantidade de SGBDs disponíveis no mercado. Alguns são desenvolvidos para a execução de consultas complexas, outros para executar um grande número de transações simples por segundo. Para auxiliar na escolha e avaliação dos produtos, são desenvolvidos *benchmarks* de performance. Os mesmos são utilizados para mensurar, de acordo com padrões pré-estabelecidos, determinados produtos, ou realizar comparativos entre estes. Os *benchmarks* buscam ser portáveis, fáceis de serem entendidos e como resultados, apresentam na sua maioria, a performance ou a razão entre o custo do sistema avaliado e a performance do mesmo, levando em consideração a quantidade de transações executadas por unidade de tempo [22].

Na próxima seção será apresentado um histórico a respeito das principais ferramentas de *benchmark* aplicadas a banco de dados, assim como suas metodologias. Isso servirá de embasamento teórico para melhor compreensão desse trabalho, no que diz respeito a forma de avaliação da performance dos algoritmos de otimização de consultas empregados no SGBD PostgreSQL.

## 2.1 Histórico e Evolução dos *Benchmarks* de Banco de Dados

A avaliação de performance de SGBDs, é uma atividade que sempre gerou muitas controvérsias. Antes de surgirem *benchmarks* propostos por organizações reconhecidas, as ferramentas de avaliação e comparação de desempenho eram criadas e mantidas pelas próprias empresas desenvolvedoras de SGBDs, afetando assim, a credibilidade e a imparcialidade dos resultados.

Dessa forma, em 1983 foi apresentado por David Dewitt, acadêmico da Universidade de Wisconsin, o primeiro *benchmark* a adotar uma metodologia científica na avaliação de SGBDs, batizado de Wisconsin [4]. O mesmo tinha o objetivo de testar a performance dos

operadores relacionais presentes em consultas simples e também complexas. Tornou-se popular por se tratar do primeiro sistema de avaliação imparcial de produtos de banco de dados, identificando os mesmos pelos seus nomes. Porém, segundo [4], os resultados apresentados eram controversos e não condiziam com a realidade dos sistemas da época.

Em abril de 1985, no artigo "A Measure of Transaction Processing Power" [1], foi proposto o *benchmark* DebitCredit, o qual teve seu desenvolvimento motivado devido aos planos de um grande banco inserir suas mil agencias, dez mil caixas automáticos e dez milhões de clientes em um sistema *on-line* [5]. A estrutura do *benchmark* apresentava uma base de dados bastante simples, com apenas três relações: agencia, ATM's (caixas automáticos) e clientes. Também era composta por uma carga de trabalho que buscava simular as transações (depósito, saque) de um cliente em uma instituição bancária.

Após a publicação do DebitCredit teve inicio uma "guerra de *benchmarks*" provida pelos desenvolvedores de SGBDs. Isso ocorria quando um desenvolvedor A perdia em avaliação de desempenho para um desenvolvedor B. Mediante especialistas, o desempenho do produto do desenvolvedor A era melhorado, superando o outro produto e assim sucessivamente.

O termo "*Benchmarketing*" ficou bastante conhecido, sendo usado para designar o processo de divulgação de resultados de *benchmark* das empresas. Geralmente quem realizava esta atividade era o departamento de *marketing* ou empresa especializada. Na maioria das vezes apenas os resultados em que o sistema tinha melhor desempenho eram demonstrados e seus pontos negativos eram omitidos. Dessa forma, os resultados apresentados eram vistos como duvidosos, havendo a clara necessidade da criação de uma entidade independente e neutra para criação de *benchmarks* confiáveis.

Sendo assim, fundada por Omri Serlin, nasceu em Agosto de 1988 o TPC (*Transaction Processing Performance Council*) [28] uma organização sem fins lucrativos que objetiva definir o processamento de transações e *benchmarks* de bancos de dados. A descrição das metodologias e *benchmarks* desenvolvidos pelo TPC serão detalhados na próxima seção.

## 2.2 *Benchmark* TPC

O TPC oferece um conjunto de testes para avaliação de desempenho de produtos, baseando-se em quesitos como integridade, confiabilidade e consistência. Com exigências rigorosas, seus *benchmarks* levam em consideração aplicações reais com transações comuns ao mundo dos negócios. Seu objetivo não é a realização direta de testes, mas apontar padrões de medição de sistemas computacionais com resultados apresentados em quantidades de operações realizadas em minutos ou segundos. Empresas como Microsoft, IBM, Oracle, Sun, HP entre outras, tem seus produtos de hardware e software avaliados por *benchmarks*.

Na Tabela 2.1 são apresentados os principais *benchmarks* providos pelo TPC, assim como seus respectivos anos de desenvolvimento e aplicações. A classificação como obsoleto ou não é estipulada pelo desenvolvedor [28].

| **Benchmaks TPC Obsoletos** | **Ano de Desenvolv.** | **Aplicação** |
|---|---|---|
| TPC-A | 1989 | OLTP |
| TPC-B | 1990 | Carga de trabalho |
| TPC-D | 1995 | Suporte a Decisão |
| TPC-R | 1999 | Suporte a Decisão |
| TPC-W | 1999 | Transações WEB |
| **Benchmaks TPC Não Obsoletos** | **Ano de Desenvolv.** | **Aplicação** |
| TPC-APP | 2004 | Servidor Aplicações/WEB |
| TPC-C | 1992 | OLTP |
| TPC-H | 1999 | Suporte a Decisão |
| TPC-E | 2007 | OLTP |
| TPC-DS | Em desenvolvimento | Suporte a Decisão |

Tabela 2.1: *Benchmarks* providos pelo TPC

Nas próximas seções serão descritos os *benchmarks* TPC classificados como "não obsoletos". Terão foco especial os *benchmarks* TPC-E e TPC-H, uma vez que foram empregados nos experimentos descritos posteriormente neste documento.

### 2.2.1 TPC-APP

Criado em 2005, o *benchmark* TPC-App simula um ambiente de comércio eletrônico de uma livraria, no qual os clientes realizam a aquisição de livros e as encomendas são

entregues no endereço especificado. Para isso a ferramenta de testes conta com uma base de dados composta por oito relações, podendo ser verificadas na Figura 2.1.

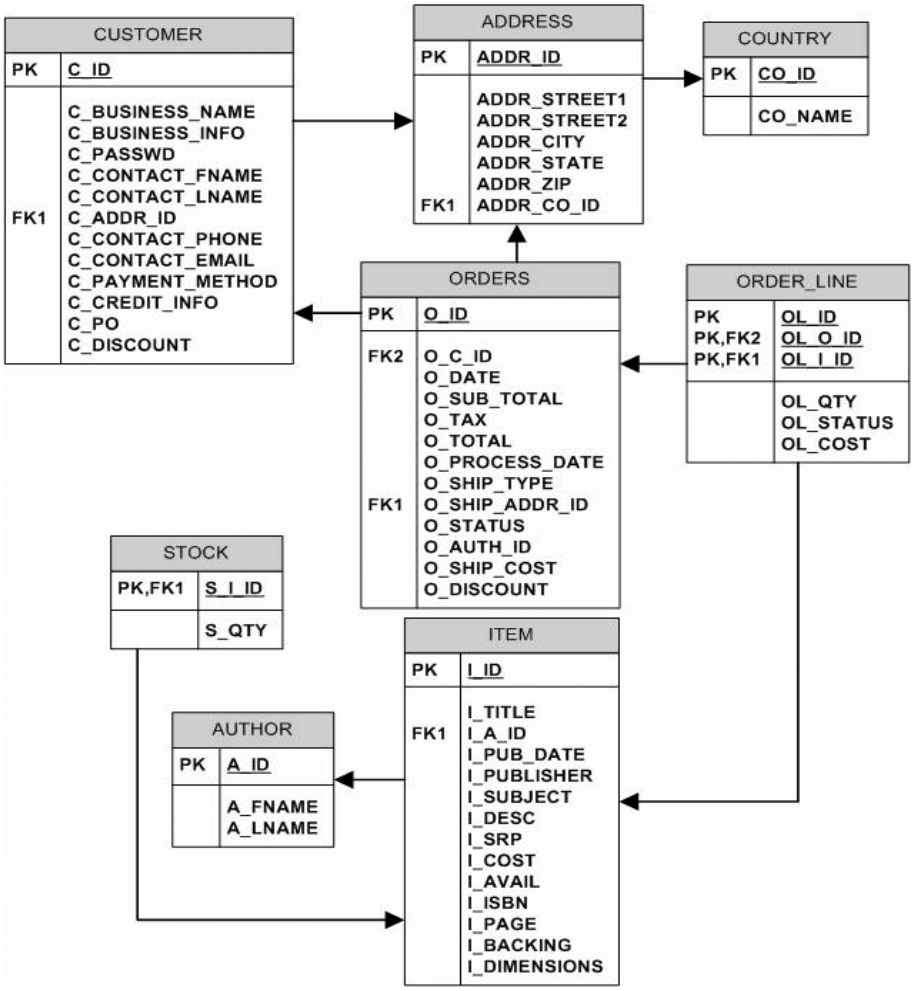


Figura 2.1: Esquema do banco de dados TPC-App [33].

O TPC-App é composto por um servidor de aplicações e um *benchmark* de serviços *web* com uma carga de trabalho que simula atividades de um servidor de transações *business-to-business* (transações eletrônicas realizadas entre empresas) em um ambiente 24X7, ou seja, com funcionamento 24 horas, 7 dias por semana [29].

O ambiente de comércio eletrônico proposto é composto por múltiplas transações *web* as quais possuem um tempo definido para retornar respostas. O TPC-App apresenta duas métricas como resultado. A primeira é a SIPS (*Web Service Interaction per Second / per Application Server System*), a qual mede a quantidade de transações de serviços *web* processadas por segundo por sistema. A segunda métrica "Total SIPS", apresenta o número total de SIPS de toda a entidade testada, associando isto ao preço por SIPS.

### 2.2.2 TPC-C

*Benchmark* composto por uma carga de trabalho, que tem por objetivo a simulação de um ambiente OLTP (*On-Line Transaction Processing*) no qual ocorrem muitas transações curtas que realizam consultas e também pequenas atualizações na base de dados [9]. O ambiente proposto procura simular as transações efetuadas por uma empresa atacadista que produz, comercializa e distribui os seus produtos. Esta empresa mostra-se distribuída geograficamente com suas vendas em distritos e armazéns. À medida que a mesma cresce, torna-se necessário a criação de novos armazéns, sendo que cada um mantém informações de cem mil produtos vendidos pela companhia. Baseado no número de armazéns é definido o tamanho da organização. Cada distrito atende a três mil clientes.

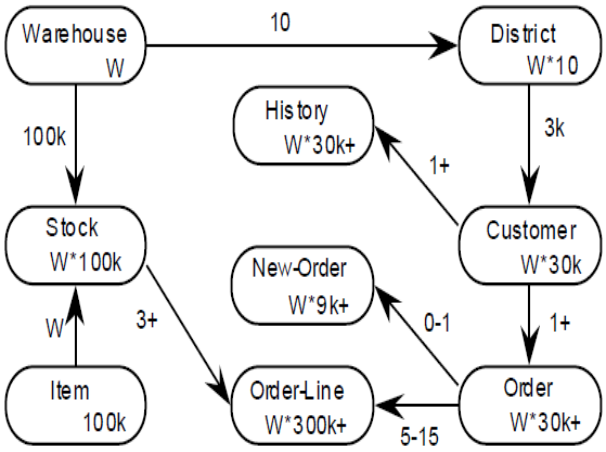


Figura 2.2: Esquema do banco de dados TPC-C [32].

A estrutura do banco de dados do *benchmark* TPC-C pode ser visualizada na Figura 2.2. A mesma é constituída por nove tabelas, as quais são carregadas com dados antes do início dos testes especificados. Os números presentes nas entidades representam a cardinalidade das tabelas, ou seja, a quantidade de registros das mesmas. Dependendo da entidade, este número deve ser multiplicado por "W" que é a quantidade de itens de dados presentes na tabela *WAREHOUSE*. Os números próximos aos relacionamentos indicam a cardinalidade dos mesmos.

São definidas cinco transações pelo *benchmark* TPC-C, as quais são submetidas a estrutura do banco de dados descrita anteriormente. Cada transação apresenta um conjunto de operações que devem ser executadas com sucesso para que um *commit* seja efetuado.

Dessa forma as transações podem ser vistas como unidades de trabalho, com operações realizadas de forma atômica [16]. A seguir são apresentadas as transações que compõem a carga de trabalho do TPC-C:

- **Transação *New Order*:** realiza o registro de uma operação de compra de um cliente. Esta transação é efetuada com alta freqüência, impondo carga média ao sistema através de operações de leitura e escrita. Trata-se de uma das principais transações do *benchmark*.

- **Transação *Payment*:** efetua o registro de pagamento de uma conta, onde é atualizado o credito do cliente, refletindo o pagamento nas tabelas *WAREHOUSE* e *DISTRICT*. Transação composta por operações de leitura-escrita de peso-leve e executada com alta freqüência durante a realização dos testes.

- **Transação *Order-Status*:** consiste da consulta de informações referentes à última compra realizada por um cliente. Essa transação faz acesso aos dados da tabela *CUSTOMER* por meio de uma chave não-primária. Trata-se de uma operação somente-leitura, com baixa freqüência de execução e que impõem uma carga moderada ao sistema.

- **Transação *Delivery*:** consiste no processamento em lote de 10 novos pedidos. Transação composta por operações de leitura e escrita na base de dados, possui baixa freqüência de execução, impondo carga moderada ao sistema.

- **Transação *Stock Level*:** esta transação determina entre os itens vendidos recentemente, quais estão com estoque abaixo de um determinado valor limite. Composta por operação somente-leitura, com baixa freqüência de execução, impõe ao sistema uma pesada carga de trabalho.

### 2.2.3 TPC-DS

O TPC-DS é o único *benchmark* provido pelo TPC que se encontra em fase de desenvolvimento. Esta ferramenta busca modelar um sistema de suporte a decisões composto

por consultas (99 *templates* de consultas distintas) e manutenção de informações (12 operações de manutenção de dados). As consultas apresentam variáveis de ligação permitindo que recebam valores diferentes, simulando uma carga de trabalho multiusuário e concorrente.

O ambiente propõem as seguintes características segundo [31]:

- Examinar grandes volumes de dados;
- Dar respostas a questões do mundo-real dos negócios;
- Executar consultas complexas que envolvem diversos requerimentos operacionais;
- Gerar atividade intensa no servidor de dados através de processamento e acesso a disco;
- Por meio de funções de manutenção, sincronizar-se com fonte de dados de sistemas OLTP.

A carga de trabalho do TPC-DS é dividida em três etapas: Carga de dados no banco; execução de consultas e manutenção de dados [14]. A fase de execução de consultas é realizada duas vezes intercaladas por uma etapa de manutenção de dados, conforme ilustra a Figura 2.3

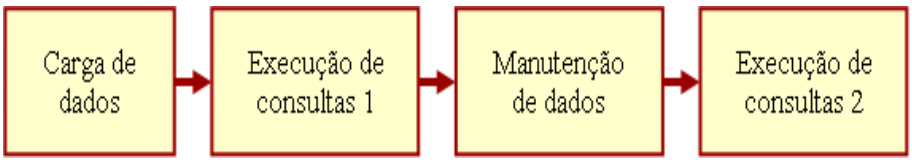


Figura 2.3: Etapas da carga de trabalho TPC-DS

O *benchmark* TPC-DS modela uma empresa que gerencia, vende e distribui seus produtos. Utiliza-se do modelo de negócio de uma grande companhia que possui múltiplas lojas em todo o país e também vende seus produtos através de catálogos e da Internet. De acordo com [31], a seguir estão alguns dos processos representados pelo TPC-DS:

- Armazenar registro de aquisições de clientes;
- Alterar preços de acordo com promoções;

- Criar páginas web dinâmicas;
- Manutenção do inventário;
- Manutenção de perfil de clientes.

### 2.2.4 TPC-E

Este *benchmark* é composto por um conjunto de transações *on-line* de múltiplos tipos (somente-leitura e intensas atualizações), que buscam simular as atividades de uma aplicação OLTP atual. Durante a escrita deste documento a versão corrente do TPC-E era a 1.6.0. A carga de trabalho e o esquema do banco de dados retratam as operações necessárias no contexto do ambiente de uma empresa de corretagem de ações, realizando interações entre seus clientes, o mercado de ações e a bolsa de valores [23], conforme ilustra a Figura 2.4.

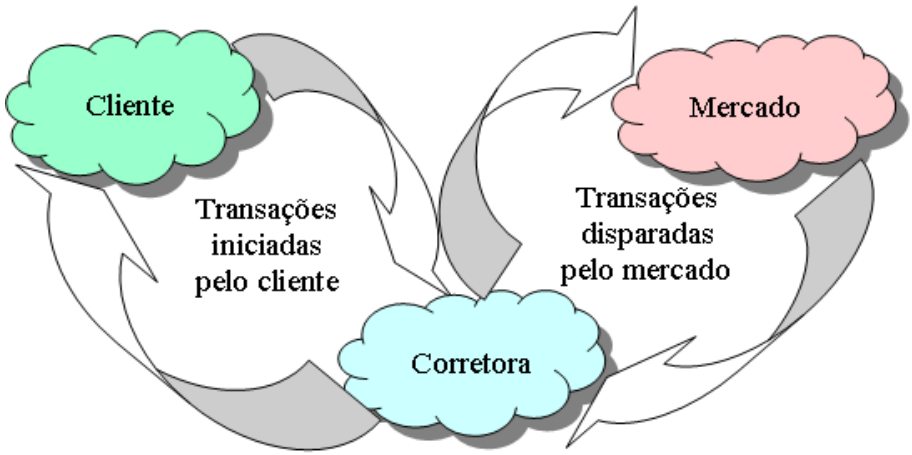


Figura 2.4: Modelo de fluxo de negócio TPC-E [34]

#### 2.2.4.1 Esquema do Banco de Dados TPC-E

O banco de dados sintético definido pelo TPC-E é composto por um conjunto de 33 relações, que são organizadas em quatro categorias descritas e visualizadas na Tabela 2.2. A geração de informações na base de dados é efetuada pelo componente *EgenLoader*, um executável binário que por padrão gera esta carga de dados no SGBD Microsoft SQL Server.

| Categoria | Tabela | Descrição |
|---|---|---|
| Cliente | ACCOUNT_PERMISSION<br>CUSTOMER<br>CUSTOMER_ACCOUNT<br>CUSTOMER_TAXRATE<br>HOLDING<br>HOLDING_HISTORY<br>HOLDING_SUMMARY<br>WATCH_ITEM<br>WATCH_LIST | Tabelas que contém informações relacionadas aos clientes. |
| Corretora | BROKER<br>CASH_TRANSACTION<br>CHARGE<br>COMMISSION_RATE<br>SETTLEMENT<br>TRADE<br>TRADE_HISTORY<br>TRADE_REQUEST<br>TRADE_TYPE | Tabelas que contém informações relacionadas à empresa corretora e aos corretores. |
| Mercado | COMPANY<br>COMPANY_COMPETITOR<br>DAILY_MARKET<br>EXCHANGE<br>FINANCIAL<br>INDUSTRY<br>LAST_TRADE<br>NEWS_ITEM<br>NEWS_XREF<br>SECTOR<br>SECURITY | Tabelas que contém informações relacionadas ao mercado financeiro como dados sobre títulos negociados, companhias entre outros. |
| Dimensão | ADDRESS<br>STATUS_TYPE<br>TAXRATE<br>ZIP_CODE | Tabelas que contém informações referenciadas pelas tabelas de fatos. |

Tabela 2.2: Tabelas do *benchmark* TPC-E

O tamanho do banco de dados como um todo assim como a vazão de transações executadas depende da cardinalidade da tabela CUSTOMER. Dessa forma as relações que compõem o banco de dados são classificadas em três conjuntos distintos conforme [34]:

- **Tabelas Fixas:** apresentam sempre as mesmas cardinalidades, independente do tamanho da base de dados e de sua vazão.
- **Tabelas Escalonadas:** tem suas cardinalidades relacionadas a cardinalidade da

tabela CUSTOMER. Durante a execução das transações estipuladas pelo *benchmark*, os dados armazenados nestas tabelas podem ser alterados, porém seus tamanhos permanecem constantes.

- **Tabelas Progressivas:** possuem suas cardinalidades iniciais relacionadas à tabela CUSTOMER, mas durante a execução das transações do *benchmark* aumentam seu tamanho.

#### 2.2.4.2 Transações TPC-E

O *benchmark* TPC-E realiza a execução de um conjunto de transações (doze no total), as quais efetuam inserções, atualizações e exclusões de informações na base de dados. Dez transações apresentam a carga de trabalho do *benchmark*, procurando manter o ambiente simples e fácil de ser executado. Logo a transação ***Data-Maintenance*** simula atualizações administrativas em tabelas que não são alteradas pelas dez transações já mencionadas. Por fim a transação ***Trade-Cleanup*** provê a "limpeza" de operações pendentes na base de dados. As transações definidas pelo *benchmark* TPC-E são descritas brevemente a seguir de acordo com [34].

- ***Broker-Volume***: tal transação busca emular o processamento efetuado pela corretora para apresentar o desempenho de vários corretores. Pode ser associada a geração de um relatório gerencial com informações de volume de corretores. Transação do tipo somente-leitura.

- ***Customer-Position***: esta transação obtém o perfil do cliente resumindo sua situação atual, tomando por base os valores correntes de mercado para seus ativos. Transação do tipo somente-leitura.

- ***Market-Feed***: emula o processo de monitoramento da atividade corrente do mercado, representando o processo efetuado pela corretora para disponibilizar o preço atualizado dos ativos. Trata-se de uma transação do tipo leitura/escrita.

- ***Market-Watch***: transação que emula o processo de monitoramento global do mercado, permitindo ao cliente verificar a tendência atual (alta ou baixa) dos ativos informados. Transação do tipo somente-leitura.

- ***Security-Detail***: busca emular o processo de acesso a informações detalhadas sobre um ativo em particular. Trata-se de uma transação do tipo somente-leitura.

- ***Trade-Lookup***: transação que emula o processo de recuperação de informações por um cliente ou corretor, que respondam a questões sobre um conjunto de operações. Como exemplo, a análise geral do mercado, a análise de desempenho de um ativo em determinado período de tempo, entre outros. Trata-se de uma transação do tipo somente-leitura.

- ***Trade-Order***: esta transação emula o processo de compra ou venda de um ativo por um cliente, corretor ou terceiro autorizado. Trata-se de uma transação do tipo leitura/escrita.

- ***Trade-Result***: transação que emula o processo de conclusão de uma operação no mercado de ações. As informações sobre as operações são armazenadas para eventuais consultas de histórico. Trata-se de uma transação do tipo leitura/escrita.

- ***Trade-Status***: efetua o processo de atualização do *status* de um conjunto particular de operações. Trata-se de uma operação do tipo somente-leitura.

- ***Trade-Update***: esta transação emula o processo para a realização de pequenas atualizações ou correções em um conjunto de operações, por exemplo, quando um cliente ou corretor, analisa um conjunto de operações que precisam ser editadas para correção. Trata-se de uma operação do tipo leitura/escrita.

- ***Data-Maintenance***: esta transação representa o processo de manutenção periódica de dados estatísticos, por exemplo, a atualização do *e-mail* de um cliente. Trata-se de uma transação do tipo leitura/escrita executada com baixa freqüência.

- ***Trade-Cleanup***: transação utilizada para cancelar operações pendentes na base de dados. Transação do tipo leitura/escrita, executada somente uma vez antes do início do *benchmark*.

#### 2.2.4.3 Métricas TPC-E

O *benchmark* TPC-E especifica as seguintes métricas primárias, publicadas nos resultados oficiais dos testes de performance:

- **Taxa de Vazão**: representa a quantidade de transações ***Trade-Result*** executadas por segundo durante a realização do teste. Tal métrica é expressa em tpsE (transações por segundo).
- **Preço/Desempenho**: esta métrica representa o custo do sistema que está sendo avaliado;
- **Data de Disponibilidade**: representa quando os produtos avaliados estarão disponíveis no mercado.

### 2.2.5 TPC-H

O TPC-H é uma metodologia de *benchmark* empregada na avaliação de sistemas de apoio a decisão. É composto por consultas de negócio onerosas e alterações simultâneas de dados, submetidas a um conjunto de dados pré-estabelecidos. Estas consultas têm características *ad-hoc*, uma vez que as variáveis que as compõem, têm seus valores alterados a cada execução, fazendo com que as consultas não se repitam. Esta metodologia não tem o objetivo de modelar um segmento de mercado em específico, mas apresentar atividades que sejam comuns a diversos [30].

#### 2.2.5.1 Esquema do Banco de Dados TPC-H

Até o inicio da escrita deste documento a versão corrente do TPC-H era a 2.7.0. A mesma apresenta uma estrutura de oito tabelas sendo divididas em: tabelas de fatos e

tabelas dimensionais. As tabelas dimensionais armazenam informações que descrevem os elementos do negócio, podendo ser descrições textuais que auxiliarão na tomada de decisões. As tabelas dimensionais são CUSTOMER, NATION, PART, PARTSUPP, REGION, SUPPLIER. As tabelas de fatos no TPCH são LINEITEM e ORDERS, as quais armazenam medições a respeito do negócio além de conter chaves para as tabelas de dimensões. Essas tabelas apresentam uma grande quantidade de registros comparados com as dimensionais.

A Figura 2.5 compreende a estrutura do banco de dados criada pelo *benchmark* TPC-H, apresentando suas tabelas e os relacionamentos entre os atributos.

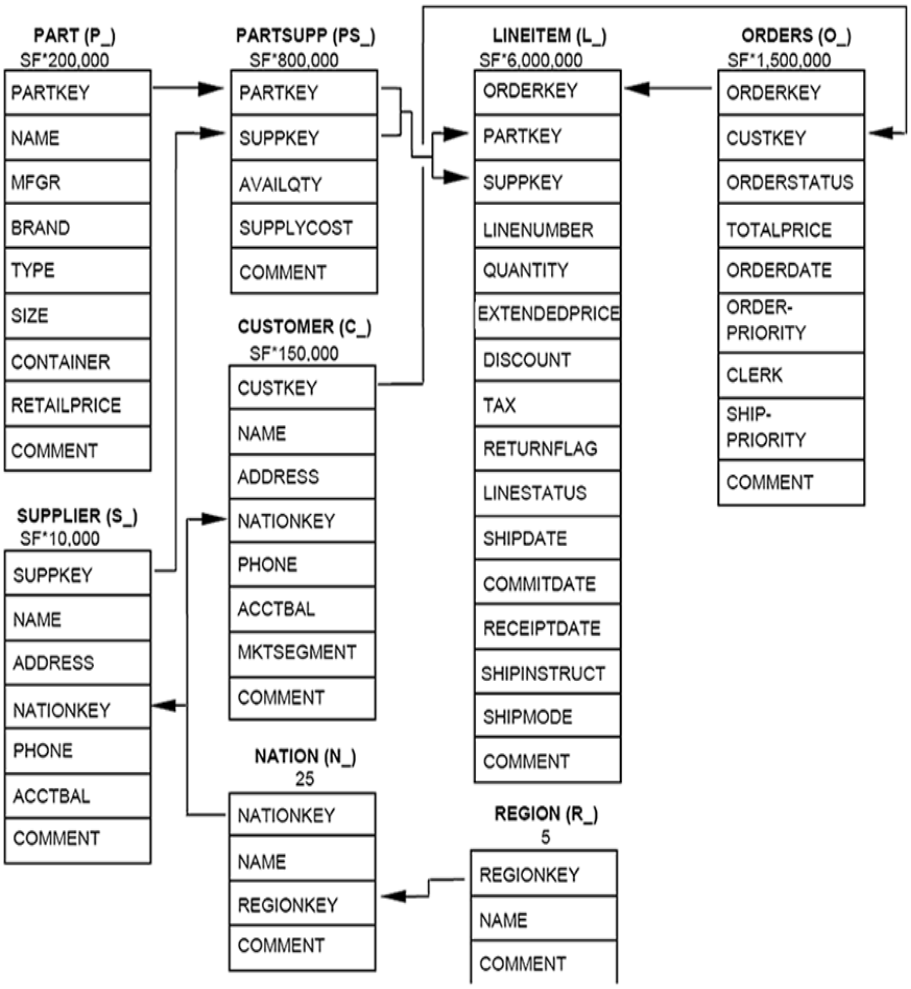


Figura 2.5: Esquema do banco de dados TPC-H 2.7.0 [35].

A geração dos dados para todas as tabelas pode ser realizada em arquivo texto ou diretamente no banco de dados, por meio da ferramenta dbgen (*database generator*) fornecida pelo TPC. O *Fator de Escala* (FS) estipulado, determinará o tamanho da base de dados

a ser gerada, devendo obedecer a um conjunto pré-definido de valores (1, 10, 30, 100, 300, 1.000, 3.000, 10.000, 30.000, 100.000). Ao definirmos este parâmetro como 1 (menor tamanho possível) será criada uma base de dados de 1 *gigabyte*, sendo isto equivalente para a escolha dos demais *Fatores de Escala* [35].

O TPC disponibiliza um aplicativo chamado qgen (*query generator*), o qual efetua a geração de vinte e duas consultas pré-estabelecidas e distintas. As consultas possuem natureza *ad-hoc*, uma vez que os parâmetros submetidos às mesmas são desconhecidos (se alteram durante as execuções) assim como a sua ordem de execução. Outra característica das consultas é o alto grau de complexidade e a capacidade de acesso a grande quantidade de dados armazenada no banco.

#### 2.2.5.2 Funções TPC-H

Depois de alimentada a base de dados e de posse das consultas, podem ser submetidas as funções de manipulação de dados e os testes que compõem o *benchmark* TPC-H, descritos abaixo, conforme [19]:

- **RF1 (*Refresh Function 1*):** função que realiza a inserção de registros nas maiores tabelas que compõem a estrutura do banco de dados gerado, ou seja, as tabelas ORDERS e LINEITEM. A quantidade de tuplas inseridas equivale a 0,1% da quantidade total de tuplas das tabelas.

- **RF2 (*Refresh Function 2*):** função que realiza a exclusão de registros nas tabelas ORDERS e LINEITEM. A quantidade de tuplas excluídas equivale a 0,1 % da quantidade total de tuplas das tabelas.

- ***Power:*** teste composto pela execução da função RF1 procedida do grupo das vinte e duas consultas mais a função RF2.

- ***Throughput:*** teste compreendido pela execução de várias sessões compostas pelas vinte e duas consultas. Essas sessões são chamadas de *query-streams* sendo que para cada uma, é necessária a execução de um par de funções de atualização (RF1 e

RF2). A seqüência de funções de atualização é chamada de *update-stream*. A quantidade de *query-streams* que devem ser submetidas depende diretamente do *Fator de Escala* estipulado no momento da criação da base de dados. Isto é especificado pelo *benchmark* conforme ilustra a Tabela 2.3.

- ***Performance:*** utiliza-se de informações dos testes *Power* e *Throughput*, apresentando aspectos referentes a capacidade de processamento de consultas.

| **Fator de Escala** | **Número de *Streams*** |
|---|---|
| 1 | 2 |
| 10 | 3 |
| 30 | 4 |
| 100 | 5 |
| 300 | 6 |
| 1000 | 7 |
| 3000 | 8 |
| 10000 | 9 |
| 30000 | 10 |
| 100000 | 11 |

Tabela 2.3: Número de Stream pelo Fator de Escala escolhido

Sempre que executada uma função de inserção de dados (RF1) deve ser executada a respectiva (RF2) para a exclusão de informações. Dessa forma garante-se que a quantidade de dados armazenadas no banco continuará a mesma. Os testes *Power* e *Throughput* podem ser mais bem compreendidos ao se verificar a Figura 2.6.

O teste *Power* fornece dados necessários ao cálculo do Power@Size que é obtido a partir da média geométrica do tempo decorrido para a execução de todas as vinte e duas consultas e das funções RFs [35], por meio da equação 2.1:

$$TPC\text{-}HPower@Size = \frac{3600 * SF}{\sqrt[24]{\prod_{i=1}^{i=22} QI(i,0) * \prod_{j=1}^{j=2} RI(j,0)}} \tag{2.1}$$

Na qual:

**SF:** é o valor do *Fator de Escala*;

**QI:** é o intervalo de tempo em segundos de cada consulta;

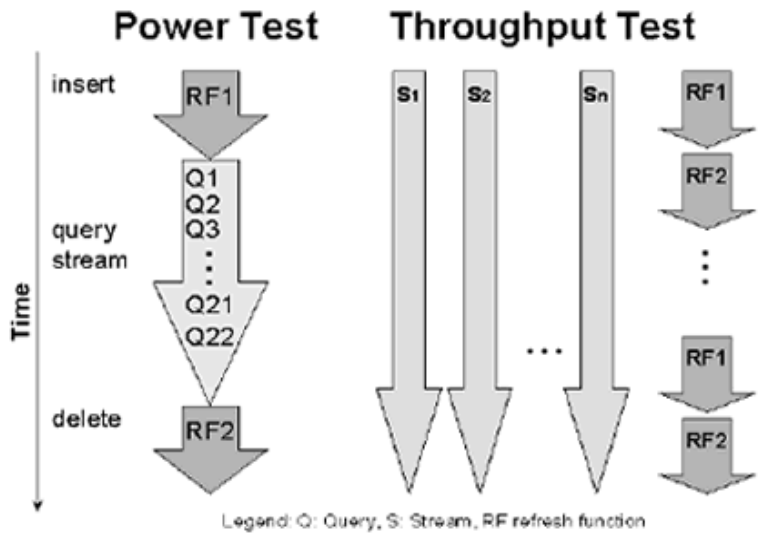


Figura 2.6: Métricas TPC-H [19]

**RI:** é o intervalo de tempo em segundos de cada função de atualização.

O teste *Throughput* fornece dados para o cálculo do Throughput@Size, que é definido como a razão do total de números de consultas pelo tempo decorrido durante a medição, sendo calculado pela equação 2.2:

$$TPC\text{-}HThroughput@Size = (S * 22 * 3600)/Ts * SF \tag{2.2}$$

Na qual:

**S:** é a quantidade de *query-streams* de consultas, valor ligado diretamente ao *Fator de Escala* descrito na Tabela 2.1;

**Ts:** é o intervalo de medição;

**SF:** é o valor do *Fator de Escala*.

### 2.2.5.3 Métricas TPC-H:

Os cálculos descritos anteriormente fornecerão dados utilizados para o cálculo das principais métricas que compõem o *benchmark* TPC-H, as quais serão descritas a seguir:

- **Metrica *Query-por-Hora*:** composta pela combinação das métricas *Power* e *Throughput* como mostra a equação 2.3:

$$QphH@Size = \sqrt{Power@Size * Throughput@Size} \quad (2.3)$$

- **Métrica Preço-desempenho:** é tido como a média do preço total do sistema, dividido pela métrica Query-por-Hora, como apresentado na equação 2.4:

$$TPC\text{-}HPrice\text{-}per\text{-}QphH@Size = \$/QphH@Size \quad (2.4)$$

- **Disponibilidade do sistema:** período de tempo em que o sistema está acessível e disponível aos clientes.

Conforme descrito em [8] o SGBD PostgreSQL apresenta deficiência na execução de algumas consultas fornecidas pelo TPC-H, acarretando em demora excessiva para apresentação de resultados. Como a reescrita de consultas é proibida pelo TPC-H, tem-se a alternativa de empregar o *benchmark* DBT3 apresentado em seções adiante.

## 2.3 *Benchmarks* DBT

A Linux Foundation (www.linux-foundation.org) é um consórcio sem fins lucrativos, que tem como objetivo auxiliar a padronização e proteção da plataforma Linux. Fundada em Janeiro de 2007 a partir da fusão da OSDL (*Open Source Development Labs*) com o FSG (*Free Standards Group*), a Linux Foundation oferece uma suíte de testes baseada nos *benchmarks* TPC. Tais ferramentas buscam automatizar a realização dos testes, auxiliando na criação e população da base de dados e geração de relatórios de desempenho exibidos nos resultados.

Os *benchmarks* providos pela Linux Foundation, também chamados de DBT, não devem ter ser resultados comparados aos obtidos pelo TPC, uma vez que estes devem cumprir a normas de auditoria rigorosas para ter seus resultados publicados. A seguir são descritas brevemente as cargas de trabalho DBT:

O *Toolkit Database Test* 2 (DBT-2) simula uma carga de trabalho baseada no *benchmark* TPC-C. Caracteriza-se por um conjunto de transações curtas que realizam alterações

ou consultas em uma quantidade pequena de tuplas. Tal ferramenta pode ser aplicada a SGBDs como PostgreSQL, MySQL ou SAP-DB.

O *kit* de testes DBT-3 baseia-se no *benchmark* TPC-H, buscando ser mais flexível e fácil de ser utilizado. Abaixo, os principais diferenciais do DBT3 em relação ao TPC-H:

- Não apresenta a métrica Preço-desempenho;
- Pode-se estipular *Fatores de Escala* diferentes dos pré-estabelecidos pelo TPC-H;
- Permite a reescrita de consultas, o que é proibido no *benchmark* TPC;
- Não é realizado o teste de ACID (Atomicidade, Consistência, Isolamento e Durabilidade);

Uma característica importante para a não comparação de resultados obtidos pelo *benchmark* TPC-H e a ferramenta DBT3, deve-se ao fato do TPC divulgar valores (preços) dos produtos utilizados nas avaliações, o que não é feito pelo DBT3.

O *Toolkit Database Test* 4 (DBT-4) baseia-se no *benchmark* TPC-App versão 1.1, o qual procura simular um ambiente transacional *business-to-business* composto por um servidor de aplicações e uma carga de trabalho de serviços *web*.

Por fim, o *Toolkit Database Test* 5 (DBT-5) simula o ambiente de uma empresa de corretagem de ações, com atividades de uma aplicação OLTP atual. Sua carga de trabalho é semelhante ao *benchmark* TPC-E. Desenvolvido pelo mestrando Rilson Oscar do Nascimento, em parceria com a Linux Foundation, o kit de testes teve sua apresentação publicada no artigo *Dbt-5: A fair usage open-source tpc-e implementation for performance evaluation of computer systems* [17]. Tal ferramenta pode ser executada apenas na aplicação PostgreSQL, porém pode ter seu funcionamento estendido para outros SGBDs.

# CAPÍTULO 3

# ALGORITMOS/ESTRATÉGIAS DE JUNÇÕES RELACIONAIS

Proporcionar respostas rápidas aos comandos estipulados pelos usuários é uma tarefa crítica desempenhada pelo SGDB. Este fato está diretamente relacionado à otimização de consultas, que é a atividade de escolha do plano mais eficiente para a execução de determinada *query* em bancos de dados relacionais.

Segundo [21], a definição de uma boa estratégia de avaliação de expressões com junções, pode ser realizada por meio de algoritmos eficientes para a execução de tais junções e de algoritmos que determinem a ordem em que as junções devem ser realizadas. Consultas SQL que fazem referência a um elevado número de relações, resultam em diversas formas de realizar as junções entre as mesmas. Avaliar todas as soluções possíveis em busca da ótima, ou seja, determinar a seqüência de junções com menor custo é uma tarefa que exige grande recurso computacional.

Neste capítulo serão abordadas diversas classes de algoritmos que apresentam soluções para o problema de otimização de consultas em banco de dados relacionais, apontando suas vantagens e desvantagens.

## 3.1 Algoritmos Determinísticos

Os algoritmos determinísticos apresentam como característica marcante a construção da solução para problemas através de uma sequência finita de passos. Para isso, pode-se utilizar de buscas exaustivas ou da aplicação de determinada heurística [15]. Seu tempo de execução é fixo e apresentará resultados idênticos quando aplicado diversas vezes sobre a mesma entrada de dados. A seguir será apresentado um algoritmo determinístico clássico que se utiliza da tecnica de programação dinâmica, onde discutiremos aspectos relacionados a sua utilização na otimização de consultas em bases de dados relacionais.

### 3.1.1 Programação Dinâmica

A técnica de programação dinâmica geralmente é aplicada a problemas de otimização que apresentam muitas soluções possíveis. Sua metodologia procura dividir o problema apresentando diversas soluções para os subproblemas (método de dividir e conquistar). O objetivo é encontrar uma solução ótima, a qual é composta por um valor ótimo (máximo ou mínimo). Cada subproblema é resolvido apenas uma vez pelo algoritmo, sendo sua resposta gravada em uma tabela, para evitar recalcular a solução, cada vez que o mesmo for encontrado [3].

Proposto pela primeira vez como estratégia de busca na otimização de consultas no SYSTEM-R por Sellinger et al. [10], a programação dinâmica atualmente é empregada em vários SGBDs. Sua execução inicia-se com a geração do plano de acesso, no qual estão contidos todas as formas de acesso às relações mencionadas na consulta, como por exemplo, varredura seqüencial ou por meio de índices presentes nas relações. Em uma segunda fase, de posse do plano de acesso, o algoritmo procura todas as formas de unir duas relações, onde o plano de menor custo é escolhido. A mesma fase é repetida, com base em seus resultados, buscando agora, a melhor forma de efetuar a junção entre três relações. Esta rotina é executada até que todas as relações mencionadas na consulta sejam incluídas no plano de execução. Os planos que apresentam custos elevados não são incluídos no plano ótimo. O mais barato é escolhido como o plano a ser executado.

Um algoritmo de programação dinâmica certamente irá apontar a solução ótima quando empregado no processo de otimização de consultas. Porém apresenta alto custo computacional ao deparar-se com *queries* que envolvam um número elevado de relações, acima de 10 a 15 mencionadas na consulta, segundo [3]. Como resultado, a utilização de memória torna-se elevada, devido à necessidade do armazenamento de soluções parciais. O processamento requerido para criação e avaliação de todos os planos possíveis aumenta exponencialmente de acordo com o número de relações existentes na consulta, podendo tornar-se impraticável. De acordo com [25], a escolha de planos para otimização de performance de consultas em banco de dados relacional é um problema NP-completo. Assim, a técnica de programação dinâmica é restrita a determinado limite de relações quando

utilizada para otimizar consultas.

Devido a estas características, motivou-se o desenvolvimento de algoritmos de otimização como forma de indicar uma solução próxima a ótima em tempo polinomial. Como exemplo, podemos citar os algoritmos heurísticos, gulosos e aleatórios, os quais são utilizados para otimizar consultas que apresentam um grande número de relações encontrando soluções sub-ótimas em um espaço de tempo aceitável.

### 3.1.2 Algoritmo de Kruskal

O algoritmo de Kruskal foi apresentado em 1956 pelo pesquisador Joseph B. Kruskal [13] como solução ao problema da árvore de custos total mínima, a qual deve ser formada pelo conjunto de arestas de peso mínimo de um grafo, selecionadas sucessivamente a partir de seus pesos.

Tido com algoritmo guloso, a cada iteração busca selecionar a aresta que possuir menor peso, adicionando-a em um subgrafo. Caso sejam encontradas arestas de mesmo custo, a escolha é feita aleatoriamente. Como restrição imposta, tem-se a escolha de arestas que não formem um ciclo entre os vértices do grafo. O algoritmo se encerra quando todos os vértices que compõem o grafo foram adicionados ao subgrafo, ou seja, todos os vértices do grafo estão presentes na árvore geradora mínima [3].

O algoritmo de Kruskal pode ser empregado na solução aproximada do problema do caixeiro viajante, no qual se busca encontrar a trajeto mais curto para a visita de todas as cidades. Na mesma linha de raciocínio, o algoritmo de Kruskal pode ser aplicado em diversas áreas como transporte, telecomunicações, processamento de imagens, entre outras, buscando a redução de custos diversos.

Este algoritmo pode ser aplicado à otimização de consultas, conforme [2], uma vez que o plano de execução ótimo pode ser representado por uma árvore geradora mínima de um grafo. As relações mencionadas na consulta são representadas como vértices do grafo. As arestas representam possíveis junções entre as relações, sendo que seus pesos são baseados nos custos para realização de tais junções. O cálculo dos custos leva em consideração atributos como cardinalidade e seletividade, por exemplo, obtidos estatisticamente pelo

SGBD. O algoritmo de Kruskal realizará primeiramente as junções entre relações que possuem menor custo, reduzindo desta forma o custo das junções seguintes que envolvem relações mais custosas e maiores.

## 3.2 Algoritmos Genéticos

Os algoritmos genéticos (GA - *Genétic Algorithm*) foram propostos por John Holland e acadêmicos da Universidade de Michigan como um método de otimização heurística de buscas, baseado no processo de seleção natural e genética [7]. Uma característica marcante dessa classe de algoritmos é que os mesmos não trabalham com uma única solução, mas com um conjunto que é chamado de população. Tais soluções são representadas por *strings* (cromossomos) compostas por caracteres (genes) que podem possuir uma série de valores [15]. A representação de um cromossomo é realizada por codificação seqüencial, numérica ou binária, sendo esta a mais utilizada.

O algoritmo genético se inicia gerando aleatoriamente a primeira população de indivíduos, com um tamanho fixo de cromossomos, os quais podem ser decodificados para uma solução do problema. De acordo com o *fitness*, que é o grau de adaptação de um individuo ao ambiente, os cromossomos são escolhidos (Seleção) a partir da população para se tornarem "pais". A reprodução ocorre entre pares de cromossomos havendo a recombinação dos mesmos (*Crossover*) produzindo descendentes. Certa fração da população pode ser escolhida aleatoriamente, para ter um gene ou um pequeno conjunto desses alterados (Mutação). A nova população criada torna-se a nova geração e o processo é repetido [12]. Essas iterações são realizadas até que sejam observadas melhorias na qualidade da população, ou o número de gerações pretendidas seja alcançado ou quando a solução desejada for encontrada. As técnicas descritas anteriormente permitem a criação de indivíduos com um grau de *fitness* maior, ou seja, com melhor adaptação que seus ancestrais [26].

Como alternativa às buscas exaustivas, os algoritmos genéticos podem representar uma solução para o problema de otimização de consultas em tempo polinomial. Porém, por se tratar de um método aleatório existe a possibilidade do algoritmo apresentar como

resultado a escolha de planos de execução impraticáveis. Além disso, outra desvantagem dos algoritmos genéticos refere-se à necessidade de realizar muitas avaliações sobre os valores de *fitness* para escolha dos cromossomos que irão se reproduzir.

## 3.3 Algoritmos Aleatórios

Os algoritmos aleatórios também chamados de randômicos, atualmente são empregados em diversas áreas como criptografias, verificação de impressões digitais entre outras. As primeiras publicações sobre sua utilização datam da década de 70, usados para a identificação de números primos.

Uma característica marcante dos algoritmos aleatórios é apresentar variações nos seus resultados, como por exemplo, o seu tempo de processamento durante as execuções, mesmo que sejam fornecidos valores de entrada idênticos. Isso ocorre porque o algoritmo realiza escolhas aleatórias, podendo estas, serem guiadas por um gerador de números aleatórios [11].

Consultas SQL que envolvem um grande número de relações resultando em um espaço de busca com tamanho significativo, podem ser otimizadas por algoritmos aleatórios. Estes algoritmos apresentam soluções bastante próximas à ótima, escolhendo em média bons planos de execução. Segundo [15] a classe de algoritmos aleatórios vê as soluções para o problema como pontos no espaço de buscas. Esses pontos são conectados por meio de arestas que representam as transformações ocorridas em uma solução, originando outra solução. As transformações ocorrem aleatoriamente segundo regras previamente estabelecidas. O algoritmo se encerra quando não é mais possível realizar transformações ou quando um tempo limite pré-definido é atingido [21].

# CAPÍTULO 4

# POSTGRESQL

Neste capítulo será apresentado o SGBD PostgreSQL, onde concentraremos nosso estudo em seu módulo de otimização de consultas, responsável pela geração de planos de execução. Tal SGBD foi escolhido por apresentar seu código fonte disponível e aberto, permitindo alterações e inclusões de funcionalidades. Outro fator determinante para esta escolha, deve-se ao fato de que neste trabalho realizaremos o aprimoramento e avaliação do algoritmo de Kruskal como otimizador de consultas simples e complexas. Dessa forma será dado continuidade aos trabalhos propostos por Pryscila Barvick Guttoski [21] que se utilizaram do SGBD PostgreSQL.

## 4.1 Histórico

O PostgreSQL é um SGBD objeto relacional de código fonte aberto desenvolvido pela Universidade da Califórnia [20]. Surgiu em 1977 com o nome de Ingres, sofrendo diversas alterações até o ano de 1985 quando se iniciou o projeto Pos-Ingres, devido à necessidade de criação de um novo SGBD. Em 1986 sua equipe de desenvolvimento lançou a documentação base do sistema Postgres, sendo que no ano seguinte uma versão operacional de demonstração já estava disponível. Em 1989 um grupo de usuários já fazia utilização de sua primeira versão.

Em 1994 com o nome Postgres95, o mesmo recebeu o interpretador da linguagem SQL e teve sua versão distribuída na Internet. Em 1996 recebeu o nome de PostgresSQL para manter o padrão com suas versões anteriores sendo apresentada a 6.0.

Tido como o maior SGBD de código-fonte aberto, disponibiliza recursos como controle de concorrência, *stored procedures*, visões (*views*), chaves primárias (*primary keys*), chaves estrangeiras (*foreign keys*), gatilhos (*triggers*) entre outros. Quando se executam transações de leitura e escrita, busca-se resguardar as propriedades ACID dando maior

confiabilidade à aplicação e informações armazenadas. Outra característica marcante é a sua portabilidade, uma vez que existem versões disponíveis para a plataforma Windows, Linux, Solaris entre outras [26].

O PostgresSQL implementa os padrões SQL ANSI, 92, 96 e 99, sendo que as declarações desta linguagem são divididas em duas categorias de acordo com suas funcionalidades: DDL(Linguagem de Definição de Dados) e DML (Linguagem de Manipulação de Dados) [18]. A DDL tem por objetivo a criação e modificação da estrutura do banco de dados compreendendo comandos como CREATE TABLE, ALTER USER, DROP INDEX. Logo a DML busca fazer a manipulação das informações armazenadas no SGBD utilizando para isso comandos como SELECT, UPDATE, DELETE e INSERT.

## 4.2 Processamento de Consultas

O processamento de consultas no PostgreSQL é composto pelas etapas detalhadas a seguir, conforme [26, 8, 27] e ilustradas na Figura 4.1.

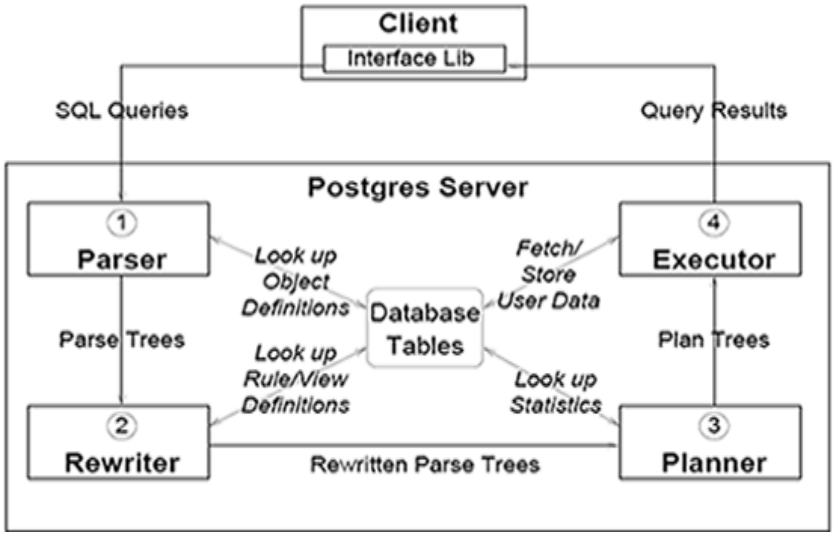


Figura 4.1: Processamento de consultas no PostgreSQL [27]

- Primeiramente é estabelecida a conexão entre a aplicação (*front-end*) e o servidor PostgreSQL (*back-end*). Em seguida a aplicação poderá transmitir comandos ao servidor ficando no aguardo dos resultados.

- É realizada uma verificação sintática dos comandos transmitidos ao servidor pelo analisador (*parse*), gerando como saída a árvore de análise (*parse tree*), utilizando-se apenas de regras fixas da estrutura sintática do SQL. Nesta etapa não há possibilidade da compreensão da estrutura semântica das operações.

- Logo após é realizada a escrita da árvore de análise gerada na etapa anterior, onde por exemplo, consultas que envolvem *views* passam a acessar as tabelas base por meio da cláusula *from*. Também são feitas verificações para se constatar se as relações, seus atributos e funções mencionadas na consulta estão presentes no banco de dados. Como resultado desta etapa, tem-se a árvore de consultas (*query tree*).

- Tomando por base a árvore de consultas, o planejador/otimizador do PostgreSQL cria um plano de execução para a consulta. Um comando SQL pode ser executado de diversas formas para a obtenção do mesmo resultado, cabendo ao otimizador de comandos a escolha do plano de execução que aparenta ser o mais rápido. O otimizador utiliza-se de estruturas chamadas *paths* as quais indicam formas para realizar as buscas nas tabelas ou em junções entre as mesmas. Depois de determinado o caminho, tido como de menor custo pelo otimizador, é gerada a árvore de plano (*plan tree*) com informações para a execução da consulta.

- A *plan tree* será processada recursivamente pelo *Executor* de modo a recuperar o conjunto de linhas solicitado pela consulta, retornando os resultados à aplicação.

## 4.3 Otimização de Consultas no PostgreSQL

O SGBD PostgreSQL utiliza-se do algoritmo de Programação Dinâmica e do módulo de algoritmo genético GEQO (*Genetic Query Optimization*) para a geração de planos de execução (levando-se em consideração o menos custoso). Dessa forma, baseando-se em um modelo de custos, o PostgreSQL define para cada operação a ser realizada, um valor numérico o qual indica a quantidade de recursos necessários para a sua execução. Tal valor numérico é expresso em unidades arbitrárias de acesso a disco e processamento [26], por exemplo.

Por padrão, o algoritmo de Programação Dinâmica é empregado em consultas que envolvem no máximo 11 relações, visto que acima desse número a sua execução torna-se inviável devido à grande quantidade de soluções possíveis e ao excessivo consumo de memória. Sendo assim, consultas onde o número de tabelas na cláusula *from* for igual ou superior a 12, acionarão automaticamente o módulo de algoritmo genético GEQO [26]. O algoritmo de Programação Dinâmica e o módulo GEQO serão apresentados de forma detalhada em seções adiante.

### 4.3.1 Programação Dinâmica no PostgreSQL

Baseado na técnica de buscas exaustivas, o algoritmo de Programação Dinâmica procura gerar todas as possíveis árvores de execução para uma determinada consulta SQL, levando em consideração a árvore geradora de menor custo.

Durante a execução do algoritmo de Programação Dinâmica primeiramente procura-se determinar quais as melhores maneiras de se acessar as relações mencionadas na consulta, elaborando-se planos individuais para cada relação. O plano de execução utilizando varredura seqüencial (*sequencial scan*) sempre é gerado para determinada relação. Quando possível o plano de execução utilizando-se de varredura por índice (*index scan*) é gerado. Sendo assim, nas consultas que se referenciam a apenas uma relação é feita uma comparação entre os planos e escolhido o qual apresentar menor custo.

Nas consultas que apresentam mais de uma relação, as junções são realizadas primeiramente entre duas relações do bloco total da consulta. Logo depois, é verificada a melhor maneira de unir grupos de três relações e assim sucessivamente, até que todas as relações mencionadas na consulta estejam presentes no plano de execução. Tomando-se por base junções entre duas tabelas, segundo [26], o PostgreSQL disponibiliza três métodos de junção os quais são apresentados a seguir:

- ***Nested loop join:*** ou *loop* aninhado, onde uma varredura completa é realizada na relação da direita para cada linha da relação da esquerda. Esta estratégia pode demandar tempo dependendo da quantidade de registros presentes nas relações e da

necessidade da varredura de dados ser realizada mais de uma vez em determinada relação.

- ***Merge join:*** antes do início das junções, ambas as relações são ordenadas pelo atributo de junção. A varredura das relações é realizada em paralelo e as linhas com correspondência formam o resultado das junções. Esta estratégia é mais atraente visto que é preciso varrer somente uma vez cada relação e geralmente é utilizada caso já existam dados ordenados anteriormente.

- ***Hash Join:*** a relação da direita é percorrida e carregada em uma tabela *hash*, na qual os atributos de junção são usados como chave da mesma. Então a relação da esquerda é percorrida, os valores apropriados são utilizados como chave de *hash* na busca de linhas com correspondência.

O algoritmo de Programação Dinâmica descartará as árvores que apresentam custo elevado, utilizando a melhor solução para realizar a busca das informações. Porém conforme o número de relações mencionadas nas consultas aumenta, o número de planos de execução possíveis também aumenta.

| **Relações (n)** | **Árvores de Processamento** | **Soluções ( Árvores . n!)** |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 2 | 12 |
| 4 | 5 | 120 |
| 5 | 14 | 1.680 |
| 6 | 42 | 30.240 |
| 7 | 132 | 665.280 |
| 8 | 429 | 17.297.280 |
| 9 | 1430 | 518.918.400 |
| 10 | 4862 | 17.643.225.600 |
| 11 | 16796 | 670.442.572.800 |
| 12 | 58786 | 28.158.588.057.600 |
| ... | ... | ... |

Tabela 4.1: Possíveis soluções para o número de relações mencionadas na consulta

Isto é ilustrado na Tabela 4.1 conforme [21], sendo o principal fator que torna o algoritmo de Programação Dinâmica inviável para consultas que apresentam buscas e

relacionamentos entre várias tabelas, devido a sua execução demandar tempo e consumir grandes quantidades de memória.

### 4.3.2 Algoritmos Genéticos no PotsgreSQL

Quando a consulta submetida ao PostgreSQL faz referência a um elevado número de tabelas (doze - valor padrão, porém alterável) é acionado o módulo de algoritmos genéticos GEQO, o qual foi implantado na versão 6.1 do SGBD em 1997 por Martin Utesch. Este módulo é composto pela combinação de seis algoritmos sendo que sua execução no PostgreSQL se processa da seguinte forma:

1. Gera-se a população, ou seja, um conjunto de possíveis junções entre as relações (*paths*) de forma randômica.

2. Escolhe-se aleatoriamente um par de *paths* da população gerada fazendo a recombinação dos mesmos. Seu resultado é inserido novamente na população. Subentende-se que este *path* gerado é melhor (plano de menor custo) que os seus geradores. O pior *path* detectado na população é excluído.

3. Por fim o melhor plano verificado na população é escolhido.

Como alternativa as buscas exaustivas, o módulo GEQO propõem a solução para o problema da otimização de consultas, buscando representar os planos de execução por meio de *strings* compostas por valores inteiros que identificam cada relação referenciada na consulta [26]. Nas *strings* encontra-se também a ordem em que as junções entre as tabelas são realizadas. Por exemplo, se tivermos a árvore representada pela Figura 4.2:

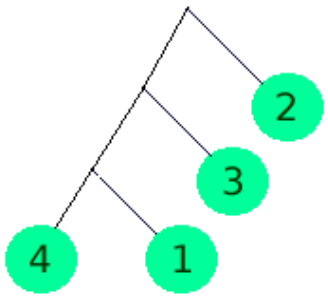


Figura 4.2: Árvore de consulta suposta

A *string* de inteiros que será codificada será: *'4-1-3-2'* onde os números 1, 2, 3, 4 são identificadores das relações.

O módulo GEQO muitas vezes compromete o desempenho do SGBD PostgreSQL pela escolha de um plano de execução impraticável. Isto se deve ao fato da escolha dos planos ser realizada por *fitness* que não realize de forma eficiente o planejamento das junções necessárias.

### 4.3.3 Algoritmo de Kruskal no PotsgreSQL

O algoritmo de Kruskal como otimizador de consultas foi implementado no SGBD PostgreSQL, utilizando-se da linguagem C por questões de compatibilidade. A função que realiza a chamada ao algoritmo de Kruskal foi incluída na mesma função que realiza a chamada aos otimizadores algoritmo de Programação Dinâmica e módulo GEQO do PostgreSQL.

Conforme descrito em [2], o *RelOptInfo* é uma estrutura de dados do PostgreSQL que representa as relações base e também as relações consideradas no processo de planejamento de execução das consultas. Para cada par de *RelOptinfo* são gerados todos os caminhos (*paths*), que representem os possíveis métodos de junção entre as relações mencionadas na consulta. Se o par não possuir cláusula de junção entre as relações, um caminho que represente o produto cartesiano da junção é gerado. A estimativa de custos para cada caminho é calculado levando-se em consideração todas as junções possíveis entre duas relações. Se existirem apenas duas relações, o caminho de menor custo é escolhido e retornado à função inicial do PostgreSQL. Caso contrário o algoritmo de Kruskal irá entrar em ação.

#### 4.3.3.1 A Função Kruskal

A função Kruskal inicia-se com a criação de um vetor no qual são ordenadas as estimativas de custos dos pares de relações. As junções de menor custo são avaliadas a cada iteração, e caso não formem ciclo, são adicionadas à solução. Caso contrário, tal junção é descartada e uma nova iteração é iniciada. Esse processo se repete até que todas as

relações estejam presentes na árvore final de execução.

Tanto o algoritmo de Kruskal com o algoritmo de Prim podem ser usados para encontrar árvores geradoras mínimas. A principal diferença entre eles é que, ao contrário do algoritmo de Prim, o algoritmo de Kruskal pode criar várias sub-árvores geradoras durante sua execução. Dessa forma, a função Kruskal pode construir *left-deep trees*, *right-deep trees* e *bushy trees*.

Toda a árvore de execução é gerada na lista *subplan*. Nesta lista são armazenados todos os planos intermediários até que se obtenha um único plano completo. O algoritmo inicialmente carrega esta lista como todos os planos individuais de cada relação presente na consulta. Neste momento, cada relação é um vértice raiz de sua própria sub-árvore geradora. Em seguida, a primeira junção *(i,j)* de relações é escolhida a partir da lista de arestas previamente ordenada. O algoritmo então percorre toda a lista de arestas e verifica se cada junção pode ser adicionada ao plano. A cada *join* realizado pela função Kruskal, um dos vértices é escolhido para ser a raiz do plano intermediário e armazenar o *RelOptInfo* da junção em seu respectivo *subplan index*. Uma possível junção entre duas relações é descartada se os sub-planos raízes de ambas as relações forem o mesmo.

O algoritmo se encerra retornando à função inicial do PostgreSQL a *RelOptInfo*, com a árvore de execução final que contém todas as relações mencionadas pela consulta presentes. O pseudocódigo implementando o algoritmo de Kruskal é apresentado na Figura 4.3.

#### 4.3.3.2 O Gerador de Arestas da Função Kruskal

O algoritmo de Kruskal é um algoritmo guloso que tem por objetivo encontrar uma árvore geradora de custo mínimo. Na implementação do algoritmo, utilizou-se a lista de possíveis junções entre pares de relações da consulta como arestas do grafo. Para cada aresta do grafo, era atribuído como peso o custo estimado da junção entre as relações envolvidas. Em seguida, a geração do plano de execução era feita seguindo a mesma ordem da construção da árvore geradora mínima.

A primeira versão do otimizador desenvolvida por Pryscila Barvick Guttoski em [2], foi utilizado um grafo completo para representar todas as possíveis junções entre as relações.

```
finaltree KRUSKAL()
{
   EDGE_LIST = GENERATE_EDGES();
   sort EDGE_LIST by join estimated cost;
   LAST_JOIN = null;
   For each EDGE(i,j) in EDGE_LIST do
   {
      If ROOT(i) is equal ROOT(j)
      Then
         Discarding the join(i,j);
      Else
         If IS_COMPOSED_PLAN( SUBPLAN[ROOT(i)] ) and
            IS_COMPOSED_PLAN( SUBPLAN[ROOT(j)] )
         Then
            SUBPLAN[ROOT(i)] = MAKE_BUSHY_JOIN(
               SUBPLAN[ROOT(i)], SUBPLAN[ROOT(j)] );
            SET_PARENT(j,i);
            LAST_JOIN = ROOT(i);
         Else If IS_COMPOSED_PLAN( SUBPLAN[ROOT(i)] )
         Then
            SUBPLAN[ROOT(i)] = MAKE_RIGHT_DEEP_JOIN(
               SUBPLAN[ROOT(i)], SUBPLAN[j] );
            SET_PARENT(j,i);
            LAST_JOIN = ROOT(i);
         Else If IS_COMPOSED_PLAN( SUBPLAN[ROOT(j)] )
         Then
            SUBPLAN[ROOT(j)] = MAKE_LEFT_DEEP_JOIN(
               SUBPLAN[i], SUBPLAN[ROOT(j)] );
            SET_PARENT(i,j);
            LAST_JOIN = ROOT[j];
         Else
            SUBPLAN[i] = MAKE_SINGLE_JOIN(
               SUBPLAN[i], SUBPLAN[j] );
            SET_PARENT(j,i);
            LAST_JOIN = i;
         End If
      End If
   }

   Return SUBPLAN[ LAST_JOIN ];
}
```

Figura 4.3: Pseudocódigo do algoritmo de Kruskal implementado.

Esta forma de geração de arestas foi então utilizada em nossos testes iniciais. Nós observamos que em alguns grafos existem arestas com valores de pesos muito bons. Entretanto

estas arestas não são boas escolhas para o plano completo.

O problema observado em nossos experimentos com certeza é característico da classe dos algoritmos gulosos. Dessa forma nós procuramos explorar uma forma de evitar estas más escolhas. Como alternativa, consideramos somente as junções que apresentam a cláusula restritiva entre suas relações. A Figura 4.4 apresenta um comparativo entre as duas formas de geração de arestas para uma mesma consulta. O grafo a) apresenta a primeira versão do gerador de arestas. O grafo b) apresenta a mesma consulta com o número reduzido de arestas. As arestas mais fortes representam as utilizadas na árvore geradora mínima.

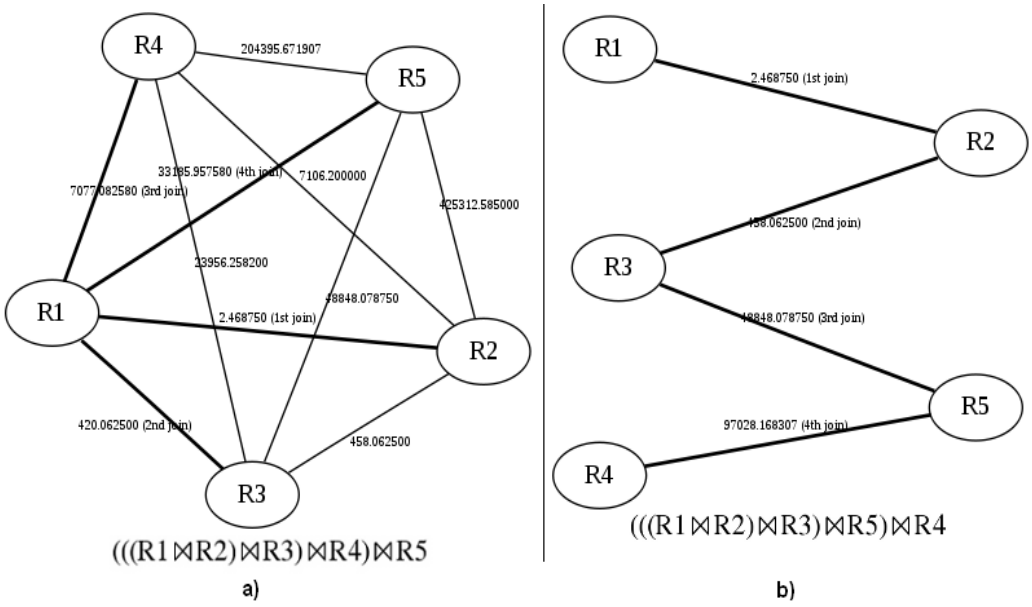


Figura 4.4: Duas formas de geração de arestas para a mesma consulta.

Foram comparados os planos e os grafos definidos pelos dois geradores de arestas. Além da redução do espaço de busca, o novo gerador de arestas apresentou planos melhores em todos os casos de teste realizados. Os planos gerados nos casos dos grafos a) e b) apresentam como custos finais 563,96 e 92,51 respectivamente. Isto mostra que uma boa definição na escolha dos planos iniciais permite que o algoritmo de Kruskal escolha bons planos.

Outro fator que motivou o desenvolvimento de uma nova versão do otimizador Kruskal foi a apresentação da mensagem de erro, exibida na Figura 4.5 e que ocorria nos seguintes casos:

- Na execução da consulta Q8 do *benchmark* TPC-H;
- Ao executar o comando *initdb*, o qual dá origem a estrutura de diretórios onde o banco de dados irá residir, e cria as tabelas de catálogo compartilhadas que pertencem ao agrupamento do banco de dados como um todo [26];
- Quando executado na aplicação psql o comando **\d**, o qual apresenta uma lista de componentes do banco de dados em questão como tabelas, *views*, índices entre outros;
- Quando executado na aplicação psql o comando **\l**, o qual exibe a lista de bases de dados presentes no SGBD PostgreSQL.

```
LOG:  server process (PID 8032) was terminated by signal 11:
 Segmentation fault

LOG:  terminating any other active server processes

LOG:  all server processes terminated; reinitializing

LOG:  database system was interrupted; last known up at
 2008-07-12 11:14:44 BRT

LOG:  database system was not properly shut down; automatic
 recovery in progress

FATAL:  the database system is in recovery mode

LOG:  record with zero length at 1/E265C670

LOG:  redo is not required
```

Figura 4.5: Log de erro apresentado na aplicação servidor PostgreSQL

Este problema era causado pelo acesso incorreto à posições de memória no momento da geração dos planos de execução. Porém o mesmo foi solucionado através da alteração das estruturas de dados que armazenam os planos parciais da consulta.

# CAPÍTULO 5

# METODOLOGIA EMPREGADA NOS EXPERIMENTOS E ANÁLISE DE RESULTADOS OBTIDOS

Este capítulo irá descrever a metodologia utilizada para a avaliação dos algoritmos de otimização de consultas juntamente com o detalhamento do ambiente experimental empregado. Também serão apresentados em detalhes a avaliação dos algoritmos e a análise dos resultados obtidos.

## 5.1 Ambiente Experimental

Nas próximas subseções daremos atenção a aspectos do ambiente utilizado para realizar os testes nos algoritmos otimizadores, tais como especificação de hardware, adaptações e configurações efetuadas nas ferramentas e metodologias de *benchmark* e também o SGBD utilizado.

### 5.1.1 Hardware e Plataforma Operacional

Tanto para a implementação quanto para a realização dos testes de performance do algoritmo de Kruskal, foi utilizado um computador com processador Intel Xeon Quad-Core 2 GHz, de 64 bits. O mesmo possuía 12 MB de memória cache L2, um total de 2 GB de memória RAM 667 MHz e dois discos rígidos SCSI de 250 GB cada, configurados em Raid 0. O sistema operacional instalado era o Linux Ubuntu 8.04 *Server Edition* com kernel 2.6.24 X86_64.

### 5.1.2 SGBD PostgreSQL

O otimizador de consultas que se utiliza do algoritmo de Kruskal foi implementado no SGBD PostgreSQL versão 8.3.1, o qual foi obtido diretamente do site do desenvolvedor.

Seu código-fonte é modularizado, o que facilita o seu entendimento e a incorporação de novas funcionalidades, visto que uma quantidade pequena de arquivos-fonte precisaram ser alterados.

A iteração com as bases de dados fornecidas pelos *benchmarks* foi realizada por meio da ferramenta psql provida pelo PostgreSQL, a qual permitiu a submissão de consultas SQL e a visualização de seus resultados. Foi habilitado o comando **\timing** do psql o qual demonstra o tempo em milissegundos (ms) necessário para que a consulta submetida a base de dados apresente os seus resultados ao cliente solicitante. Neste intervalo estão contabilizados, entre outros, o tempo para a geração do plano de execução e o *delay* da rede.

### 5.1.3 Ferramentas para Avaliação de Performance

Como foram realizados testes avaliando o comportamento de algoritmos otimizadores de consultas, surgiu a necessidade de *benchmarks* que forneçam um conjunto diversificado de *queries*, ou seja, que façam acesso a um conjunto variado de relações e conseqüentemente realizem diversas junções relacionais. Dessa forma não somente o otimizador padrão do PostgreSQL (algoritmo de Programação Dinâmica) será acionado, mas também o módulo GEQO possibilitando o confronto de seus resultados com os obtidos a partir do otimizador Kruskal.

Com esse intuito, adaptamos as metodologias de *benchmark* TPC-E e TPC-H, para que as mesmas se enquadrassem às necessidades exigidas por nossos experimentos. As bases de dados de tais *benchmarks* foram implementadas respectivamente pelos *toolkits* DBT-5 e DBT-3. Dessa forma teremos duas metodologias próprias para realização dos testes de performance. Uma que toma por base a metodologia TPC-E e outra a TPC-H, as quais serão detalhadas nas próximas seções.

#### 5.1.3.1 Metodologia de Testes Baseada no *Benchmark* TPC-E

A metodologia TPC-E foi adaptada em nossos experimentos, visto que a TPC-H fornece um banco de dados composto por apenas 8 tabelas, o qual não dá suporte a

quantidade mínima de relações requisitadas para o acionamento do módulo GEQO. Outro importante fator para esta escolha, deve-se ao *toolkit* DBT-5 efetuar a carga de dados e iterações diretamente com a aplicação PostgreSQL. Por fim, utilizando-se da base de dados provida pelo DBT-5 e das consultas SQL disponibilizadas, os testes descritos neste documento poderão ser recriados por outros pesquisadores.

Como já apresentado no Capítulo 2, o *benchmark* TPC-E executa uma série de transações as quais realizam a manutenção da base de dados. Como estas transações não apresentam relevância no contexto dos testes realizados (nossa necessidade são *queries* SQL complexas), as mesmas foram descartadas. Outra característica do *benchmark* TPC-E é a falta de consultas que façam referência a um número elevado de relações para ativar o módulo GEQO do PostgreSQL. Porém esse problema foi solucionado com a criação de *queries* próprias, uma vez que a base de dados do TPC-E apresenta um conjunto de 33 relações.

As 15 consultas desenvolvidas encontram-se no Apêndice B deste documento e fazem referência a um conjunto que varia de 5 até 19 relações na cláusula *from* conforme demonstrado na Tabela 5.2. Basicamente, tais *queries* realizam a extração de diversas informações da base de dados do *benchmark* TPC-E, como dados a respeito da empresa corretora de ações e taxas de impostos aplicadas às negociações. Também são apresentados o nome dos indivíduos que são clientes de determinada corretora de ações e o nome das empresas que possuem títulos a serem negociados, assim como seu respectivo setor ou área de atuação.

O tamanho da base de dados gerada pelo DBT-5 depende diretamente da cardinalidade da tabela CUSTOMER. Em nossos testes, foi escolhido o fator de 1000 registros inseridos inicialmente nesta tabela. Este é o valor mínimo gerado pelo DBT-5, uma vez que nosso objetivo não é gerar testes em bases de dados de larga escala.

#### 5.1.3.2 Metodologia de Testes Baseada no *Benchmark* TPC-H

A metodologia TPC-H também foi escolhida para ser adaptada em nossos experimentos uma vez que apresenta um conjunto diversificado de consultas SQL complexas, que

fazem acesso de forma variada, a uma grande quantidade de informações armazenadas na base de dados. Porém, do total de 22 consultas fornecidas pelo *benchmark*, apenas 11 foram utilizadas. As demais fazem acesso a apenas 1 ou 2 relações, o que resulta no não acionamento do módulo do algoritmo de Kruskal, conforme descrito no Capítulo 4. Na Tabela 5.1 são apresentadas as consultas fornecidas pelo *benchmark* TPC-H que foram utilizadas em nossos testes, assim como suas respectivas quantidades de tabelas a que fazem referência. Tais consultas estão presentes no Apêndice C deste documento.

| **Consultas TPC-H** | **Número de Tabelas** |
|---|---|
| Q2 | 5 |
| Q3 | 3 |
| Q5 | 6 |
| Q7 | 6 |
| Q8 | 8 |
| Q9 | 6 |
| Q10 | 4 |
| Q11 | 3 |
| Q18 | 4 |
| Q20 | 3 |
| Q21 | 4 |

Tabela 5.1: Consultas do *benchmark* TPC-H utilizadas nos experimentos

Conforme demonstrado em [8] o PostgreSQL não consegue resolver de forma eficaz algumas consultas providas pelo TPC-H. Sendo assim, foi utilizado o *toolkit* DBT-3, o qual permite a reescrita de consultas, o que é proibida pelo TPC. As métricas que compõem o *benchmark* TPC-H não foram aplicadas em nossos experimentos, pois não pretendemos avaliar o custo total do sistema ou sua disponibilidade aos usuários.

A população da base de dados foi efetuada utilizando a escala de 1 GB (a menor possível), uma vez que o nosso objetivo é testar o comportamento dos otimizadores em consultas complexas e não em bases de dados de larga escala.

Essas foram as adaptações e configurações realizadas na metodologia de *benchmark* TPC-H, para que a mesma atendesse ao intuito de nossos experimentos. Nas próximas seções, serão exibidos e discutidos os resultados obtidos nos testes de performance.

## 5.2 Resultado dos Experimentos

Depois das bases de dados dos *benchmarks* TPC-E e TPC-H devidamente populadas, iniciou-se a fase de análise dos algoritmos otimizadores. Durante a execução dos testes, as consultas foram aleatoriamente submetidas às suas respectivas bases de dados durante trinta vezes, utilizando os três algoritmos otimizadores. Antes da execução de cada consulta, o serviço do SGDB PostgresSQL era interrompido. Logo após, a limpeza da memória *cache* do computador era realizada. Em seqüência o SGBD reiniciado. Por meio desses procedimentos, buscou-se evitar que informações armazenadas em *cache* pudessem influenciar os resultados. Tais práticas foram aplicadas para a extração das médias apresentadas como resultado, referentes ao tempo de execução das consultas, custo dos planos de execução gerados e tempo de execução do algoritmo de otimização, sendo estes, os três testes realizados.

Nas próximas subseções serão apresentadas em detalhes, as avaliações efetuadas juntamente com seus resultados. Tal demonstração será dividida em duas partes, de acordo com a metodologia de *benchmark* a qual nos baseamos, ou seja, serão apresentados e discutidos os resultados obtidos através do *benchmark* TPC-E e TPC-H, devidamente adaptados aos nossos testes de performance.

### 5.2.1 Resultados Obtidos pela Metodologia de Testes Baseada no *Benchmark* TPC-E

Por padrão, nos testes realizados, apresentamos como resultado a média aritmética de um conjunto de trinta execuções de cada consulta submetida à base de dados. Porém, como poderemos verificar, o módulo de otimização GEQO do PostgreSQL, apresenta grandes variações em seus resultados, tanto de valores referentes ao tempo de execução das consultas, quanto de custos de geração dos planos de execução. Dessa forma, do total de valores referentes a trinta execuções, utilizando somente a otimização pelo módulo GEQO, retiramos do cálculo da média final destes dois testes, os cinco melhores resultados e os cinco piores, buscando apresentar ao leitor uma média real mais refinada dos dados

exibidos pelo algoritmo genético.

Nas tabelas que demonstram os resultados dos testes realizados, os dados apresentados em negrito foram obtidos através do otimizador Kruskal e de acordo com a quantidade de relações referenciadas na consulta, pelo algoritmo de Programação Dinâmica ou módulo GEQO. Os dados que não estão em negrito foram obtidos alterando-se as configurações padrão do PostgreSQL, ou seja, acionando-se o otimizador que utiliza o algoritmo de Programação Dinâmica para consultas com mais de 11 relações na cláusula *from*, e acionando-se o módulo de algoritmos genéticos para consultas que apresentam menos de 12 relações.

#### 5.2.1.1 Tempo de Execução das Consultas

Os primeiros testes realizados utilizando a metodologia baseada no *benchmark* TPC-E, têm seus resultados apresentados na Tabela 5.2, os quais se referem a média de tempo, em milissegundos, necessária para que o SGBD possa apresentar as informações solicitadas pelas consultas. Ao observá-los, verificamos que o algoritmo de Programação Dinâmica retornou as informações em um período médio menor, em todos os casos, quando comparado às médias obtidas pelos otimizadores Kruskal e módulo GEQO. Isto ocorreu porque o algoritmo de Programação Dinâmica apresenta a solução ótima na escolha do plano de execução.

Ao verificarmos alguns dos valores de tempos de execução das consultas, que irão compor a média aritmética das consultas Q5 e Q12, constatamos grande regularidade entre os mesmos, quando a otimização é realizada pelos algoritmos de Programação Dinâmica e Kruskal. Estes resultados são característicos da classe dos algoritmos determinísticos da qual ambos fazem parte. Devido a esta regularidade, o algoritmo de Kruskal consegue desempenho melhor em consultas que possuem mais de 15 relações em comparação ao módulo GEQO do PostgreSQL.

A maior variação apresentada pelo módulo GEQO nas diversas execução das consultas Q5 e Q12 se deve a natureza randômica do algoritmo genético que realiza a escolha dos planos de execução utilizando-se de valores aleatórios gerados como *fitness*.

| Consultas Próprias | Tempo médio de execução da consulta (ms) | | | Número de relações |
|---|---|---|---|---|
| | Alg. Prog. Dinâm. | Módulo GEQO | Alg. de Kruskal | |
| Q1 | **203,996** | 225,771 | **211,942** | 5 |
| Q2 | **2.187,642** | 2.278,518 | **2.277,281** | 6 |
| Q3 | **2.318,941** | 2.386,785 | **4.200,989** | 7 |
| Q4 | **2.330,352** | 2.422,287 | **4.226,590** | 8 |
| Q5 | ***2.322,429*** | *2.493,105* | ***4.214,129*** | |
| | ***2.311,989*** | *2.602,428* | ***4.252,543*** | |
| | ***2.335,659*** | *2.451,495* | ***4.229,923*** | |
| | ***2.351,356*** | *2.391,759* | ***4.191,479*** | |
| | ***2.329,372*** | *2.408,093* | ***4.229,267*** | |
| | . | . | . | |
| | . | . | . | |
| | . | . | . | |
| Média | **2.320,054** | 2.422,877 | **4.222,099** | 9 |
| Q6 | **2.365,507** | 2.485,630 | **4.432,240** | 10 |
| Q7 | **2.430,236** | 2.791,915 | **6.765,130** | 11 |
| Q8 | 2.480,835 | **3.367,651** | **6.788,037** | 12 |
| Q9 | 2.523,089 | **3.630,714** | **7.002,197** | 13 |
| Q10 | 2.430,236 | **7.468,124** | **10.950,735** | 14 |
| Q11 | 4.906,784 | **8.998,196** | **11.074,331** | 15 |
| Q12 | *5.052,805* | ***10.788,089*** | ***10.863,256*** | |
| | *5.077,703* | ***18.874,020*** | ***10.784,473*** | |
| | *5.043,014* | ***7.208,869*** | ***10.780,561*** | |
| | *5.048,870* | ***18.067,826*** | ***10690,991*** | |
| | *5.078,427* | ***16.729,860*** | ***10.724,963*** | |
| | . | . | . | |
| | . | . | . | |
| | . | . | . | |
| Média | 5.077,196 | **13.125,131** | **10.823,223** | 16 |
| Q13 | 5.421,417 | **17.394,801** | **11.103,035** | 17 |
| Q14 | 10.635,684 | **40.672,559** | **14.626,561** | 18 |
| Q15 | 12.169,261 | **62.909,249** | **14.894,753** | 19 |

Tabela 5.2: Tempo médio de execução das consultas - Base de dados do *benchmark* TPC-E

As observações e conclusões obtidas a partir dos dados presentes na Tabela 5.2, podem ser mais bem compreendidas através do gráfico presente na Figura 5.1.

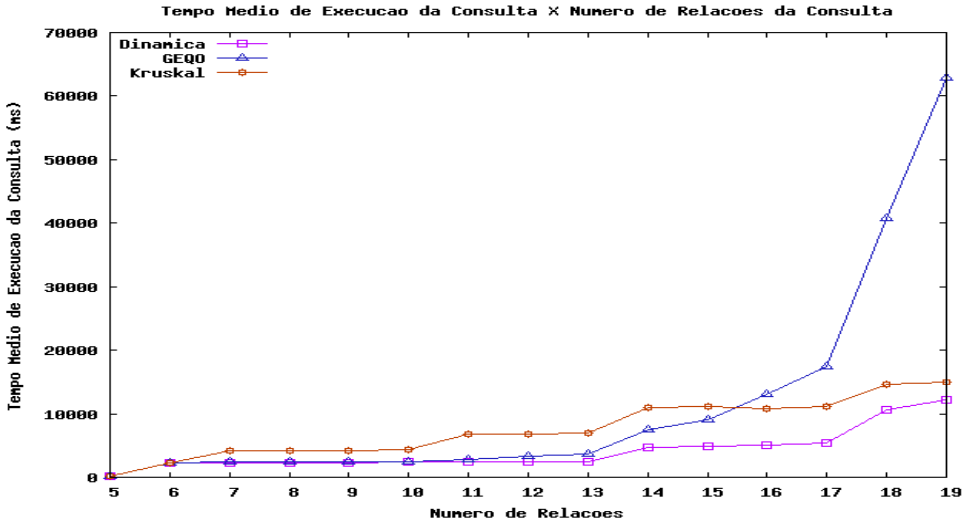


Figura 5.1: Tempo médio de execução das consultas X número de relações

A imprevisibilidade na apresentação dos resultados pelo módulo GEQO torna-se mais evidente à medida que se aumenta o número de relações referenciadas pela consulta, sendo estas características ilustradas pelos gráficos apresentados nas Figuras 5.2 e 5.3. As ilustrações apresentam informações sobre 30 execuções das consultas Q5 e Q12 que fazem referência a respectivamente 9 e 16 relações na cláusula *from*.

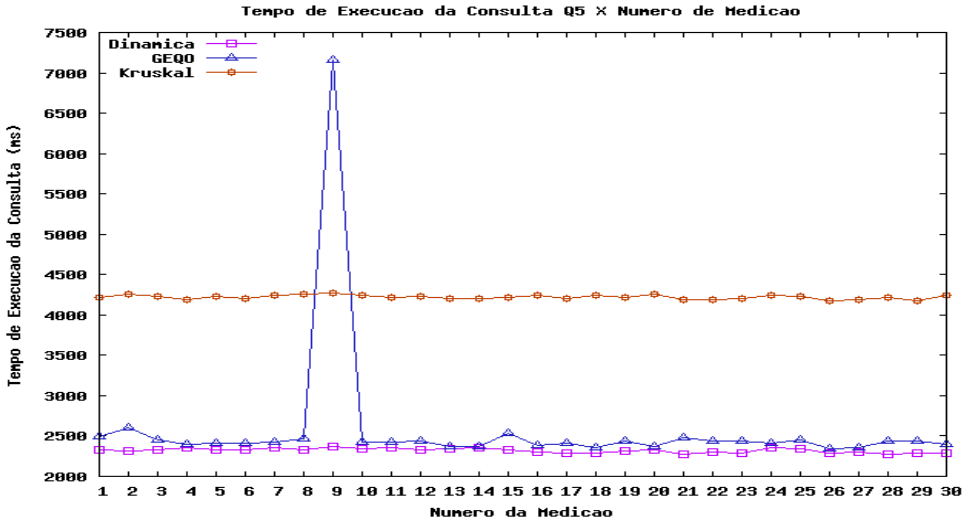


Figura 5.2: Tempos de execução da consulta Q5 X número da medição

Como o algoritmo genético trabalha com um número fixo de amostras em relação ao espaço total de buscas, talvez estas informações não sejam suficientes para avaliar com

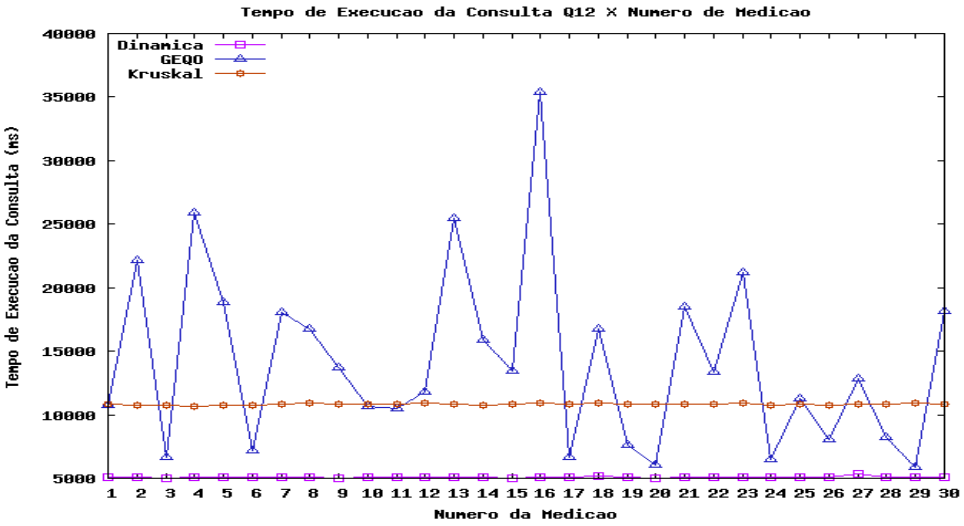


Figura 5.3: Tempos de execução da consulta Q12 X número da medição

maior eficiência os possíveis planos de execução que podem ser gerados. Isto se torna mais problemático à medida que se aumenta o número de relações na cláusula *from*. Como resultado, os planos escolhidos podem ficar bastante distantes do ótimo, ocasionando menor desempenho para retornar as informações solicitadas pela consulta.

| Consultas Próprias | Tempos retirados GEQO (ms) | |
|---|---|---|
| | Máximo | Mínimo |
| Q1 | 239,572 | 214,157 |
| Q2 | 2.607,313 | 2.210,176 |
| Q3 | 2.696,496 | 2.266,438 |
| Q4 | 2.735,535 | 2.314,741 |
| Q5 | 7.152,235 | 2.339,853 |
| Q6 | 5.246,017 | 2.331,370 |
| Q7 | 7.684,273 | 2.438,536 |
| Q8 | 7.623,416 | 2.455,537 |
| Q9 | 8.165,441 | 2.690,058 |
| Q10 | 15.718,557 | 4.762,482 |
| Q11 | 17.944,153 | 4.882,425 |
| Q12 | 35.357,617 | 5.913,021 |
| Q13 | 62.504,944 | 5.126,520 |
| Q14 | 95.811,289 | 11.434,756 |
| Q15 | 495.422,963 | 12.134,301 |

Tabela 5.3: Valores máximo e mínimo obtidos pelo Módulo GEQO para o tempo de execucao das consultas - Base de dados do *benchmark* TPC-E

Finalizaremos nossa análise a respeito do tempo de execução das consultas de acordo com os otimizadores empregados, apresentando na Tabela 5.3, o valor máximo e mínimo de

tempo de execução exibidos pelo módulo GEQO para cada *query* executada. Tais valores não foram contabilizados na média do tempo de execução das consultas otimizadas pelos algoritmos genéticos, conforme já descrito, objetivando dessa forma a apresentação de valores médios mais refinados.

O gráfico apresentado na Figura 5.4 ilustra os valores máximos de tempo de execução das consultas, encontrados durante a bateria de trinta execuções de cada consulta com os três algoritmos otimizadores. Mais uma vez fica claro que o módulo GEQO do PostgreSQL pode apresentar valores irregulares, extremamente altos de tempos de execução das consultas, quando estas apresentam diversas junções relacionais.

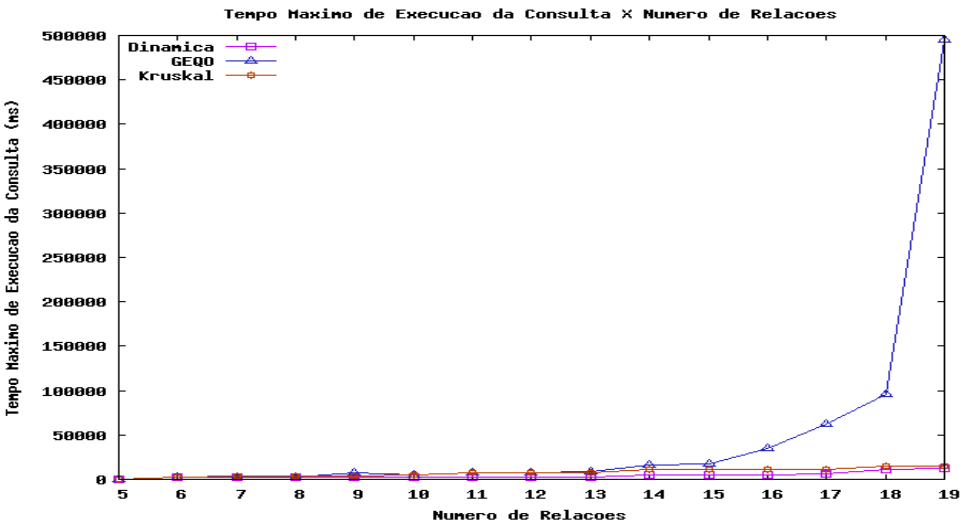


Figura 5.4: Valores máximos de tempo de execução das consultas de acordo com o otimizador empregado

### 5.2.1.2 Tempo de Execução dos Algoritmos Otimizadores

Nos próximos testes que tem seus resultados exibidos na Tabela 5.4, nossa atenção foi voltada ao tempo de execução dos algoritmos otimizadores. Estas informações foram calculadas por meio de funções que desenvolvemos para este intuito, apresentando seus resultados em milissegundos. Observa-se, que o algoritmo de Kruskal teve, em todos os testes, sua média de tempo de execução menor em relação ao otimizador que se utiliza do algoritmo de programação dinâmica e também ao módulo GEQO. Ao verificarmos a Tabela 5.4, fica claro que quanto maior a quantidade de relações na cláusula *from*, melhor

é o desempenho, relacionado ao tempo de execução, do otimizador Kruskal, comparado ao algoritmo Programação Dinâmica. Basta analisarmos a consulta Q7 que envolve 11 relações, onde o otimizador padrão do PostgreSQL teve sua média de tempo de execução quase 17 vezes mais lenta comparada ao Kruskal.

| Consultas Próprias | Tempo de execução do algoritmo (ms) | | |
|---|---|---|---|
| | Alg. Prog. Dinâm. | Módulo GEQO | Alg. de Kruskal |
| Q1 | **0,727** | 5,022 | **0,526** |
| Q2 | **1,365** | 13,492 | **0,672** |
| Q3 | **2,287** | 34,993 | **0,773** |
| Q4 | **3,350** | 43,525 | **0,837** |
| Q5 | ***4,701*** | *52,549* | ***0,930*** |
| | ***4,692*** | *53,547* | ***0,931*** |
| | ***4,680*** | *54,004* | ***0,937*** |
| | ***4,671*** | *51,635* | ***0,934*** |
| | ***4,685*** | *53,758* | ***0,934*** |
| | . | . | . |
| | . | . | . |
| | . | . | . |
| Média | **4,698** | 52,786 | **0,927** |
| Q6 | **9,079** | 65,482 | **1,024** |
| Q7 | **18,806** | 74,926 | **1,115** |
| Q8 | 50,768 | **92,380** | **1,279** |
| Q9 | 92,490 | **108,711** | **1,357** |
| Q10 | 214,160 | **122,118** | **1,509** |
| Q11 | 377,702 | **141,836** | **1,588** |
| Q12 | *600,083* | ***153,746*** | ***1,700*** |
| | *602,494* | ***155,615*** | ***1,699*** |
| | *600,933* | ***160,186*** | ***1,704*** |
| | *599,711* | ***159,917*** | ***1,700*** |
| | *600,714* | ***158,438*** | ***1,702*** |
| | . | . | . |
| | . | . | . |
| | . | . | . |
| Média | 601,195 | **157,106** | **1,701** |
| Q13 | 886,265 | **173,861** | **1,883** |
| Q14 | 1.991,894 | **189,758** | **2,045** |
| Q15 | 3.396,845 | **213,632** | **2,163** |

Tabela 5.4: Tempo médio para execução dos algoritmos otimizadores - Base de dados do *benchmark* TPC-E

Procuramos apresentar na Tabela 5.4 os valores individuais do tempo de execução dos algoritmos otimizadores utilizando as consultas Q5 e Q12. Como podemos observar, não

ocorreram variações bruscas entre os resultados apresentados pelas diversas execuções, tratando-se da mesma consulta e utilizando qualquer um dos algoritmos otimizadores. Porém a medida que se aumenta o número de relações referenciadas na cláusula *from*, o algoritmo de Programação Dinâmica exibe um crescimento vertiginoso relacionado ao seu tempo de execução. Tal fato ocorre uma vez que o mesmo efetua a geração de diversas árvores de execução, e então às compara para a escolha da solução final. Dessa forma, quanto maior a quantidade de relações que a consulta apresentar, mais lento será a construção do plano ótimo em relação ao algoritmo de Kruskal, que gera apenas uma árvore de execução, sendo está a final. Nossa análise referente aos tempos de execução dos algoritmos otimizadores pode ser mais bem observada através do gráfico exibido na Figura 5.5.

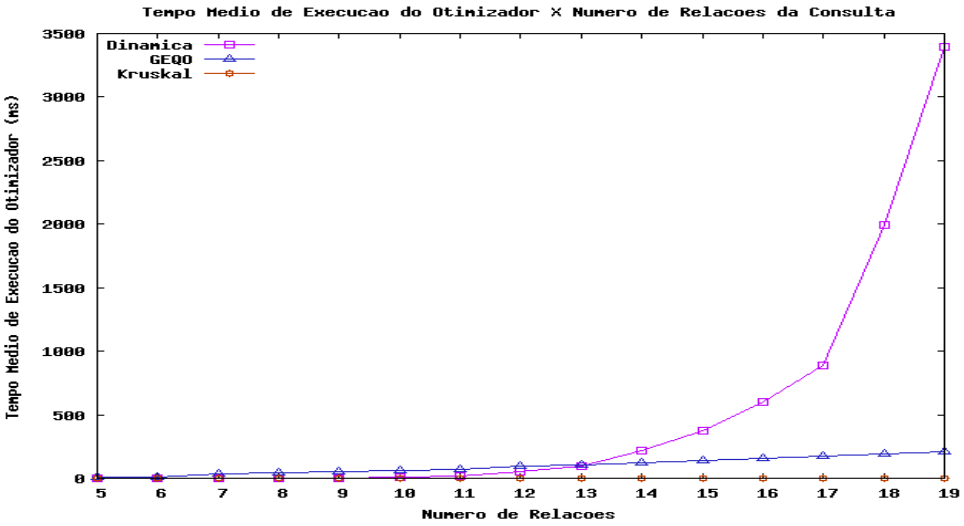


Figura 5.5: Tempo médio de execucao dos otimizadores X número de relações

Levando-se em consideração os algoritmos genéticos e de Kruskal, observa-se que este teve seu tempo de execução sempre inferior comparado ao módulo GEQO, mesmo aumentando-se a quantidade de tabelas mencionadas nas consultas. Isto ocorre por diversos fatores, entre eles podemos citar que o Kruskal é um algoritmo classificado como guloso e para resolver o problema de otimização, efetua escolhas que lhe parecem melhores em determinado momento, não realizando a verificação de todas as possíveis soluções. Além disso, o algoritmo de Kruskal possui um código de implementação simples e adaptável ao conceito de otimização de consultas. Os algoritmos genéticos por sua vez, precisam

realizar muitas avaliações sobre os valores de *fitness*, necessários para a escolha de quais planos de execução serão recombinados, sendo este um dos principais fatores que afetam seu desempenho.

#### 5.2.1.3 Custo dos Planos de Execução Gerados pelos Otimizadores

Os últimos testes utilizando a metodologia baseada no *benchmark* TPC-E, tratam das estimativas de custos para a geração dos planos de execução, considerando os três algoritmos avaliados. Estas informações podem ser obtidas através do plano de execução da consulta, apresentado pelo comando **EXPLAIN ANALYZE** do psql. A Figura 5.6 demonstra o plano estimado para a consulta Q1, otimizada pelo algoritmo de Programação Dinâmica. O valor 15.59 representa o "Custo Total" estimado para retornar todas as linhas solicitadas pela consulta, mesmo valor este, apresentado na Tabela 5.5 para a consulta Q1.

```
                          QUERY PLAN
------------------------------------------------------------------------
 Hash Join (cost=4.59..15.59 rows=240 width=73)
   Hash Cond: (commission_rate.cr_tt_id = trade_type.tt_id)
   ->  Hash Join (cost=3.48..11.18 rows=240 width=64)
        Hash Cond: (commission_rate.cr_ex_id = exchange.ex_id)
        ->Seq Scan on commission_rate (cost=0.00..4.40 rows=240 width=11)
        ->Hash  (cost=3.43..3.43 rows=4 width=67)
               -> Merge Join  (cost=1.08..3.43 rows=4 width=67)
```

Figura 5.6: Custo do plano de execução estimado para a consulta Q1

Os planos de execução apresentados pelo algoritmo de Kruskal em nenhum dos casos foram iguais aos exibidos pelo algoritmo de Programação Dinâmica ou módulo GEQO, de acordo com o número de relações referenciadas nas *queries*

Se compararmos as médias dos custos estimados, com as referentes médias de tempo de execução das consultas, observaremos certa equivalência. Porém estes dados não devem ser analisados diretamente, pois os tempos de execução são medidos em milissegundos de

| Consultas Próprias | Custo médio de geração dos planos de execução | | |
|---|---|---|---|
| | Alg. Prog. Dinâm. | Módulo GEQO | Alg. de Kruskal |
| Q1 | **15,59** | 20,38 | **185,25** |
| Q2 | **32.909,59** | 32.910,23 | **33.079,25** |
| Q3 | **34.162,09** | 34.135,74 | **85.091,75** |
| Q4 | **34.232,06** | 34.207,08 | **85.161,73** |
| Q5 | . . . ***34.330,51*** . . . . | *34.484,58* *34.298,93* *34.402,98* *35.022,01* *34.311,36* . . . | . . . ***85.162,98*** . |
| Média | **34.233,31** | 34.381,01 | **85.162,98** |
| Q6 | **34.350,56** | 35.218,52 | **85.280,23** |
| Q7 | **35.078,01** | 42.040,56 | **123.096,23** |
| Q8 | 35.017,11 | **54.562,52** | **123.129,06** |
| Q9 | 35.052,78 | **57.347,52** | **123.164,73** |
| Q10 | 66.825,84 | **139.169,87** | **205.937,23** |
| Q11 | 66.918,48 | **163.638,29** | **206.033,57** |
| Q12 | . . . *66.928,65* . . . . | ***231.823,61*** ***524.349,01*** ***145.624,95*** ***592.468,66*** ***344.354,71*** . . . | . . . ***206.043,74*** . |
| Média | 66.928,65 | **254.091,73** | **206.043,74** |
| Q13 | 66.931,32 | **330.982,75** | **206.046,41** |
| Q14 | 131.638,73 | **696.836,39** | **295.843,07** |
| Q15 | 131.677,43 | **1.062.655,61** | **295.881,77** |

Tabela 5.5: Custo médio de geração dos planos de execução - Base de dados do *benchmark* TPC-E

tempo real, e as estimativas de custo são expressas em unidades arbitrárias de acesso a disco [26].

Como as informações utilizadas para a geração dos planos de execução são obtidas por meio de estimativas, podem ocorrer casos em que determinados valores de custo apresentados pelos otimizadores sejam ligeiramente menores aos apresentados pelo otimizador que se utilizado do algoritmo de Programação Dinâmica. Foi o que ocorreu nas consul-

tas Q3 e Q4 na Tabela 5.5. Nos demais casos, o algoritmo de Programação Dinâmica foi capaz de gerar planos de execução com custos mais reduzidos em relação aos outros otimizadores. O gráfico apresentado na Figura 5.7 procura auxiliar na compreensão das informações exibidas pela Tabela 5.5.

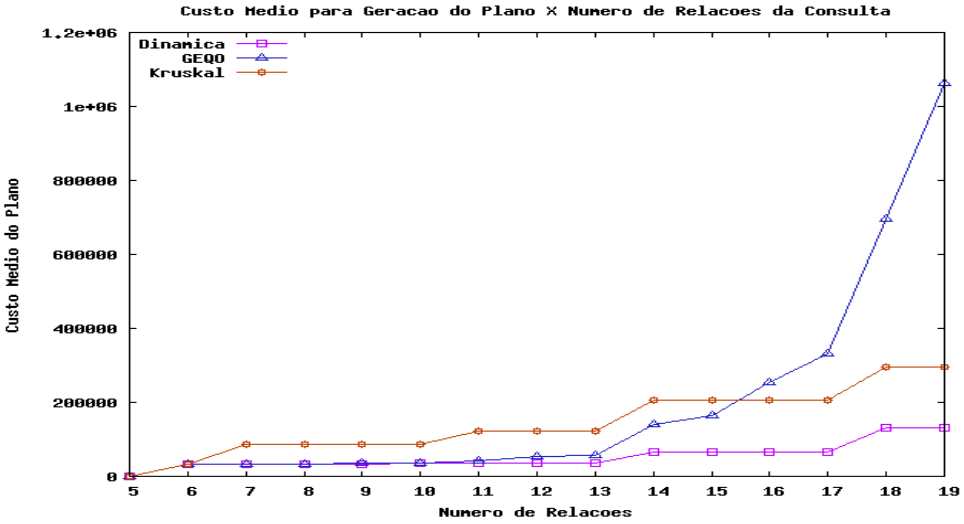


Figura 5.7: Custo médio estimado dos planos para as consultas X número de relações

Na Tabela 5.5 constatamos novamente a regularidade nos valores individuais dos custos apresentados pelas consultas Q5 e Q12, quando a otimização é realizada pelos algoritmos determinísticos. Tanto o algoritmo de Kruskal quanto o Programação Dinâmica, exibem o mesmo valor de custo durante as diversas execuções de determinada consulta, sendo esta uma característica marcante da classe dos algoritmos determinísticos.

Ainda tratando dos resultados individuais dos custos para as consultas Q5 e Q12, verificamos que a otimização realizada pelo módulo GEQO do PostgreSQL, apresenta variações significativas em seus resultados. Dessa forma, retiramos do cálculo da média de custos, os cinco valores de custo mais altos e os cinco mais baixos, os quais têm seus picos apresentados na Tabela 5.6.

O gráfico ilustrado pela Figura 5.8 mostra que assim como para valores referentes ao tempo de execução das consultas, os algoritmos genéticos apresentam valores extremamente altos de estimativa de custos para a geração de planos. Esta característica torna-se mais evidente quando o número de relações referenciadas pelas consultas é superior a 14, apresentando valores maiores de custos que os exibidos pelo algoritmo de Kruskal.

| Consultas Próprias | Custos retirados GEQO (ms) | |
|---|---|---|
| | Máximo | Mínimo |
| Q1 | 38,29 | 14,29 |
| Q2 | 32.932,29 | 32.908,29 |
| Q3 | 34.168,94 | 33.872,79 |
| Q4 | 34.732,94 | 33.942,77 |
| Q5 | 57.566,26 | 33.944,30 |
| Q6 | 57.981,06 | 34.196,85 |
| Q7 | 71.864,65 | 35.079,86 |
| Q8 | 81.464,98 | 35.512,97 |
| Q9 | 82.938,15 | 37.067,24 |
| Q10 | 241.449,32 | 71.722,86 |
| Q11 | 307.920,84 | 72.095,60 |
| Q12 | 613.169,32 | 120.444,99 |
| Q13 | 1.047.327,97 | 73.634,59 |
| Q14 | 1.488.718,86 | 271.852,43 |
| Q15 | 3.129.034,15 | 299.369,63 |

Tabela 5.6: Custos máximo e mínimo obtidos pelo Módulo GEQO para as consultas - Base de dados do *benchmark* TPC-E

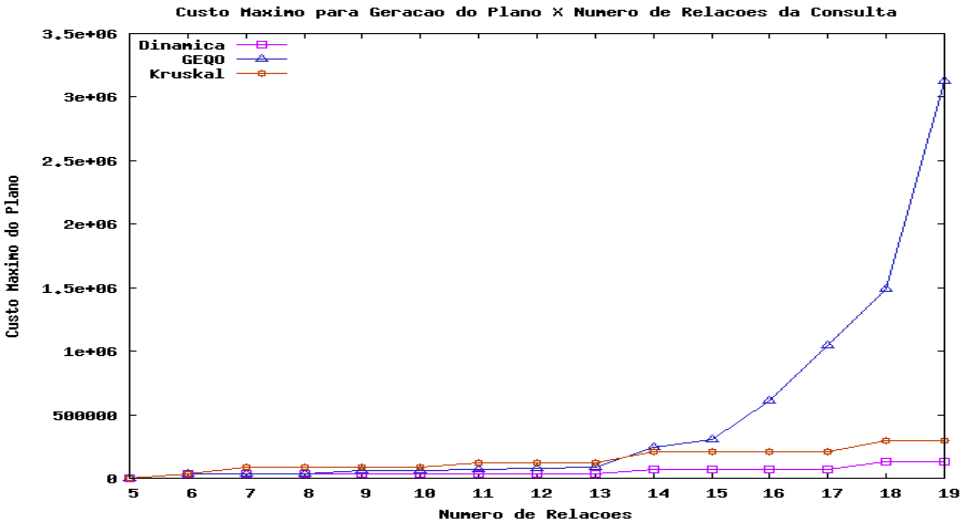


Figura 5.8: Valores máximos de custos estimados para planos de execução de acordo com o otimizador empregado

### 5.2.2 Resultados Obtidos pela Metodologia de Testes Baseada no *Benchmark* TPC-H

A adaptação do *benchmark* TPC-H aos nossos experimentos é justificada pelos seguintes aspectos:

- Avaliar o comportamento dos otimizadores em consultas complexas que fazem acesso

a diversas informações da base de dados;

- Refazer os testes de performance realizados por Pryscila Barvick Guttoski em [21], a partir de uma nova metodologia de testes e das alterações realizadas no algoritmo de Kruskal;

- Realizar testes de performance utilizando-se de consultas fornecidas por uma ferramenta de *benchmark* consolidada;

Nas próximas subseções serão detalhados os testes que envolvem o *benchmark* TPC-H adaptado aos experimentos.

#### 5.2.2.1 Tempo de Execução das Consultas

Na etapa inicial dos testes, analisamos a média de tempo, em milissegundos, necessária para que o SGBD possa retornar as informações solicitadas pelas consultas, de acordo com o algoritmo de otimização empregado. Como podemos observar na Tabela 5.7, em diversos casos o algoritmo de Kruskal é capaz retornar as informações solicitadas em tempos próximos ao algoritmo padrão do PostgreSQL. Isso ocorreu em seis consultas (Q3, Q10, Q11, Q18, Q20 e Q21), nas quais, o Kruskal apresentou soluções bastante próximas à ótima, que é exibida pelo algoritmo de Programação Dinâmica. Como característica marcante, tais *queries* mencionam uma pequena quantidade de relações na cláusula *from* da SQL. Nas consultas Q2 e Q8 que referenciam uma quantidade maior de relações, (respectivamente cinco e oito), também apresentaram os resultados solicitados em tempo próximo ao algoritmo padrão do PostgreSQL.

Porém nas consultas Q5, Q7 e Q9, as quais fazem referência a um total de seis relações, a heurística empregada pelo algoritmo de Kruskal efetuou a seleção de planos de execução distantes dos escolhidos pelo algoritmo padrão do PostgreSQL. A consulta Q5, por exemplo, quando otimizada pelo algoritmo de Programação Dinâmica apresentou em média os resultados solicitados em 13 segundos. A mesma consulta submetida ao otimizador Kruskal demorou, em média, cerca de 4 minutos para exibir resultados. Isso ocorreu

| Consultas TPC-H | Tempo médio de execução da consulta (ms) | | Qtde de relações na consulta |
|---|---|---|---|
| | Alg. Prog. Dinâm. | Alg. de Kruskal | |
| Q2 | 3.633,171 | 3.773,563 | 5 |
| Q3 | 14.706,317 | 15.203,847 | 3 |
| Q5 | 13.277,888 | 241.219,434 | 6 |
| Q7 | 12.214,222 | 20.301,878 | 6 |
| Q8 | 247.393,070 | 247.626,412 | 8 |
| Q9 | 1.039.492,953 | 1.423.284,610 | 6 |
| Q10 | *13.362,432*<br>*13.476,712*<br>*13.100,503*<br>*13.451,194*<br>*13.231.669*<br>.<br>.<br>. | *15.123,461*<br>*14.663,121*<br>*14.554,433*<br>*14.336,304*<br>*15.322,936*<br>.<br>.<br>. | 4 |
| média | 13.262,600 | 15.063,718 | |
| Q11 | 83.394,229 | 85.673,846 | 3 |
| Q18 | 39.265,047 | 41.155,851 | 4 |
| Q20 | 444.396,380 | 444.830,228 | 3 |
| Q21 | 879.418,268 | 878.567,981 | 4 |

Tabela 5.7: Tempo médio de execução das consultas TPC-H pelos algoritmos de Programação Dinâmica e Kruskal

porque o Kruskal apresentou planos de execução com custos maiores que o algoritmo padrão do PostgreSQL conforme pode ser visualizado na Tabela 5.9.

#### 5.2.2.2 Tempo de Execução dos Algoritmos Otimizadores

Nos próximos testes, buscamos verificar e confrontar o tempo médio de execução dos algoritmos otimizadores. Os valores demonstrados na Tabela 5.8 revelam o tempo médio em milissegundos, gasto pelos otimizadores para a criação e escolha dos planos de execução das consultas fornecidas. Neste caso observamos que o algoritmo de Kruskal teve seu tempo médio de execução menor em todos os casos comparado às médias do Programação Dinâmica.

Também é possível verificar que quanto maior a quantidade de relações referenciadas pela cláusula *from*, melhor é o desempenho do algoritmo de Kruskal em comparação ao otimizador padrão do PostgreSQL. Isto se deve ao fato do Kruskal não gerar tantas

| Consultas TPC-H | Tempo de execução do algoritmo (ms) | |
|---|---|---|
| | Alg. Prog. Dinâm. | Alg. de Kruskal |
| Q2 | 1,230 | 0,396 |
| Q3 | 0,324 | 0,282 |
| Q5 | 4,167 | 1,215 |
| Q7 | 2,447 | 0,567 |
| Q8 | 3,391 | 0,731 |
| Q9 | 3,421 | 0,930 |
| Q10 | *0,839*<br>*0,840*<br>*0,844*<br>*0,842*<br>*0,838*<br>.<br>.<br>. | *0,752*<br>*0,746*<br>*0,759*<br>*0,752*<br>*0,753*<br>.<br>.<br>. |
| média | 0,840 | 0,745 |
| Q11 | 0,499 | 0,458 |
| Q18 | 0,660 | 0,410 |
| Q20 | 0,311 | 0,289 |
| Q21 | 0,459 | 0,358 |

Tabela 5.8: Tempo médio para execução dos algoritmos de Programação Dinâmica e Kruskal nas consultas TPC-H

soluções no espaço de busca quanto o Programação Dinâmica, resultando em menor tempo para ser executado.

#### 5.2.2.3 Custo dos Planos de Execução Gerados pelos Otimizadores

Nos últimos testes realizados utilizando a metodologia baseada no *benchmark* TPC-H, buscou-se verificar o custo médio dos planos de execução gerados por ambos os algoritmo de otimização. Como já comentamos, existe certa equivalência entre os valores de custo estimado pelos algoritmos e a média de tempo necessária para exibir os dados solicitados pela consulta. Tal relação pode ser constatada ao verificarmos as informações da consulta Q7. Um alto custo para geração do plano de execução apresentado pelo algoritmo de Kruskal, o qual é visualizado na Tabela 5.9, resultou em maior demora para apresentação de resultados pela consulta em comparação ao algoritmo de Programação

Dinâmica, demonstrado na Tabela 5.7. Em outros casos o algoritmo de Kruskal retornou valores de custos estimados bastante próximos aos exibidos pelo algoritmo de Programação Dinâmica como nas consultas Q2, Q8 e Q20.

| Consultas TPC-H | Custo dos planos de execução | | Planos de Execução Iguais |
|---|---|---|---|
| | Alg. Prog. Dinâm. | Alg. de Kruskal | |
| Q2 | 92.597,34 | 92.668,10 | Não |
| Q3 | 295.561,99 | 295.561,99 | Sim |
| Q5 | 283.109,74 | 5.870.303,23 | Não |
| Q7 | 277.199,04 | 322.208,50 | Não |
| Q8 | 182.530,71 | 183.575,57 | Não |
| Q9 | 6.625,11 | 7.111,81 | Não |
| Q10 | 256.780,10 | 260.687,79 | Não |
| Q11 | 52.415,68 | 52.415,68 | Sim |
| Q18 | 613.907,55 | 664.297,22 | Não |
| Q20 | 13.494,27 | 13.731,98 | Não |
| Q21 | 895.137,15 | 895.137,15 | Sim |

Tabela 5.9: Média dos custos dos planos de execução gerados pelos algoritmos de Programação Dinâmica e Kruskal para consultas TPC-H

Outra informação interessante apresentada na Tabela 5.9 diz respeito à igualdade ou não dos planos de execução apresentados por ambos os algoritmos para determinada consulta SQL. Tais informações são observadas ao habilitarmos os comandos **EXPLAIN ANALYZE** do psql. Planos de execução idênticos apresentados pelos dois algoritmos para a mesma consulta resultarão em custos iguais para geração dos mesmos.

Nas Tabelas 5.7 e 5.8, foram apresentados valores individuais do tempo necessário para a execução da consulta Q10 e também o tempo para a execução do algoritmo otimizador aplicado à mesma *query*. Tais dados irão compor a média apresentada como resultado. Como podemos observar, em ambos os algoritmos otimizadores, não ocorreram variações bruscas nos valores do tempo de execução da consulta e do otimizador. Esta é uma característica marcante já descrita, da classe dos algoritmos determinísticos, na qual o Programação Dinâmica e o Kruskal fazem parte.

Quanto aos valores dos custos dos planos de execução gerados, os mesmos são idênticos, utilizando o mesmo algoritmo otimizador e a mesma consulta é aplica diversas vezes à base de dados. Isso retrata a premissa de que, ao aplicarmos diversas vezes a mesma

entrada de dados a um algoritmo determinístico, será exibida sempre a mesma saída de dados.

### 5.2.3 Discussão sobre os Resultados

Ao final da análise dos resultados, podemos constatar que o algoritmo de Kruskal apresentou bons resultados referentes ao tempo de execução e estimativa de custos, levando-se em consideração consultas que mencionam três ou quatro relações na cláusula *from*. Esta conclusão é obtida quando observamos os resultados apresentados pela metodologia de testes baseada no *benchmark* TPC-H. Porém tais vantagens, não justificam sua escolha em relação ao algoritmo de Programação Dinâmica, uma vez que os planos de execução podem apresentar soluções bastante distantes da ótima. Isto torna inviável nestes casos, a utilização do algoritmo de Kruskal em comparação ao algoritmo de Programação Dinâmica, devido a possível demora para a apresentação dos resultados solicitados na consulta. Ainda levando-se em consideração consultas que envolvem poucas relações, verificamos que o tempo de execução do algoritmo de Programação Dinâmica que é exponencial, não chega a causar influência considerável no tempo de execução da consulta como um todo.

Quando consideramos *queries* que envolvem um número elevado de relações, (acima de 15 verificado em nossos testes), vemos grande vantagem na utilização do algoritmo de Kruskal em relação ao módulo genético do PostgreSQL. Não somente pela execução da consulta em um período médio de tempo menor, mas principalmente pela regularidade de tempo na apresentação dos resultados. Como podemos ver na consulta Q15, o algoritmo genético foi capaz de retornar as informações solicitadas em apenas 12 segundos no melhor caso, logo no pior em mais de 8 minutos. O algoritmo de Kruskal por sua vez retornou os dados com maior regularidade em um período médio de 14 segundos.

Outra vantagem do algoritmo de Kruskal refere-se ao seu tempo de execução, que é polinomial gerando o plano de execução da consultas de forma muito mais rápida em comparação ao módulo de algoritmos genéticos e Programação Dinâmica.

# CAPÍTULO 6

# CONCLUSÃO E TRABALHOS FUTUROS

Não é exagero afirmar que o otimizador de consultas é um dos principais responsáveis pelo bom desempenho do SGBD. Como foi possível observar nos resultados dos nossos experimentos, o processo de otimização de consultas é fundamental para garantir a eficiência na apresentação dos resultados solicitados pelas aplicações, através das *queries*.

Apesar de diversos métodos de otimização de consultas complexas terem sido propostos nos últimos anos, por universidades e SGBDs comerciais, nenhum conseguiu atender satisfatoriamente a todas as classes de consultas. Caso seja optado por um algoritmo otimizador que realize a escolha do plano de execução ótimo, teremos o rendimento do SGBD afetado devido ao seu tempo de execução, principalmente quando aplicado a consultas que envolvem múltiplas relações. Neste caso, o estudo de um método que apresente planos de execução ao menos próximos ao ótimo, em um tempo de execução viável, com certeza resultará em um otimizador de consultas bastante eficiente.

O aumento da disseminação de softwares-livres e de código-fonte aberto, permite que diversos usuários possam contribuir para a melhoria das aplicações a qual fazem uso, sendo o SGBD PostgreSQL, um exemplo bastante interessante. A facilidade na obtenção do código-fonte, aliado a sua modularidade e documentação bastante ampla, permite a incorporação de novas funcionalidades, além da correção de possíveis erros de implementação.

O algoritmo de Kruskal apresenta diversas vantagens quando aplicado a otimização de consultas, como código de implementação simples e tempo de execução polinomial. Devido ao seu espaço de busca ser reduzido a apenas uma árvore de consulta, seu processo de construção do plano de execução é mais rápido em comparação ao algoritmo de Programação Dinâmica.

Porém a maior vantagem na utilização do algoritmo de Kruskal é a sua regularidade

referente ao tempo de execução das consultas, a qual se mantém constante à medida que o número de relações mencionadas nas *queries* também aumenta. Esta característica não é observada pelos algoritmos genéticos no PostgreSQL.

Ao finalizarmos este trabalho é importante deixar claro ao leitor que o nosso objetivo com a realização dos diversos testes, não foi apontar a melhor metodologia de otimização, mas sim apresentar o algoritmo de Kruskal como uma solução viável, de fácil implementação e com baixo custo computacional para otimização de consultas complexas. Os resultados obtidos pela análise dos otimizadores podem sofrer variações, principalmente tratando-se do módulo GEQO. Isto pode ser constatado, nos resultados apresentados pelos testes, devido as buscas deste algoritmo serem realizadas de forma aleatória. É absolutamente normal que o módulo GEQO apresente aleatoriamente, planos de execução mais eficientes que os gerados pelo algoritmo de Kruskal.

**Trabalhos Futuros**: Os *benchmarks* disponíveis no mercado permitem a análise de desempenho dos SGBDs como um todo, levando em consideração, a quantidade de transações executadas por unidade de tempo. Porém durante o desenvolvimento deste trabalho, constatou-se a necessidade de uma ferramenta que realizasse a avaliação de aspectos relacionados ao algoritmo otimizador. Dados peculiares, como tempo de execução do algoritmo ou estimativa de custos para geração do plano, são vitais para a análise de performance dos otimizadores. O desenvolvimento de uma ferramenta que automatize o processo de avaliação e permita a apresentação clara dos resultados ao usuário é um trabalho a ser realizado.

Verificar a eficiência dos cálculos de estimativas de custo para geração dos planos de execução no SGBD PostgreSQL é outro trabalho que pode ser realizado. Isso pode resultar na escolha de planos de execução mais eficientes fazendo com que as informações sejam retornadas mais rapidamente ao solicitante.

Como o resultado de nossos experimentos apontaram, apesar de inadequado à otimização de consultas com múltiplas junções relacionais, algumas melhorias podem ser aplicadas ao GEQO, como forma de aumentar sua previsibilidade em relação à geração dos planos. Dong e Yiwen [6] consideram algumas técnicas de heurística como forma de

geração de planos iniciais para um algoritmo genético aplicado ao problema de múltiplas junções. Neste trabalho, os autores observam algumas melhorias dos planos em certas classes de consultas. Heurísticas semelhantes poderiam ser aplicadas como forma de diminuir a variação de planos gerados.

Assim como o algoritmo de Kruskal apresentou bons resultados, é interessante o estudo, implementação e testes de outros algoritmos determinísticos para a otimização de consultas que envolvem múltiplas junções.

# APÊNDICE A

# OTIMIZADOR KRUSKAL IMPLEMENTADO NO SGBD POSTGRESQL

```
1   /*
     * kruskal.c
     *        solution to the query optimization problem
     *        by means of Kruskal Algorithm
     *
     * Portions Copyright (c) 1996-2009, PostgreSQL Global Development Group
     * Portions Copyright (c) 1994, Regents of the University of California
     *
     * contributed by:
     *=========================================================================*
     *  Pryscila Barvick Guttoski      *  C3SL - Centro de Computação    *
     *  prycabg@gmail.com              *  Científica e Software Livre /  *
     *  ----------------------------   *  Departamento de Informática /  *
     *  Adriano Lange                  *  Universidade Federal do Paraná *
     *  adriano@c3sl.ufpr.br           *  Curitiba, Brasil               *
     *  ----------------------------   *                                 *
     *  Tarcizio Bini                  *  http://www.c3sl.ufpr.br        *
     *  tarcizioab@c3sl.ufpr.br        *                                 *
     *=========================================================================*
     */


    #include "postgres.h"

    #include "optimizer/kruskal.h"
    #include "optimizer/kruskal_2wayplan.h"
    #include "optimizer/pathnode.h"
    #include "optimizer/paths.h"
    #include "parser/parsetree.h"

    bool kruskal_log     = DEFAULT_KRUSKAL_LOG;
    bool kruskal_dynedge = DEFAULT_KRUSKAL_DYNEDGE;

    /**
     * edgeType
     *    Structure used to compare the edges costs.
37   */
    typedef struct edgeType {
            RelOptInfo *rel; // join plan with 2 relations
            double weight;   // join cost (heuristic for the Kruskal's algorithm)
            int tail_idx,    // (head and tail) relation index in initial_rels
                head_idx;
            int tail_relid,  // (head and tail) relation id
                head_relid;
            bool used;       // debug only. Used in print_kruskal_graph()
    } edgeType;

    static RelOptInfo *kruskal(PlannerInfo *root, List **initialrels,
                    edgeType *edge_list, int numEdges, int numVertices);

    static int find_index_by_relid( List *joinitems, int relid );

    static inline int find_root(int vertex, int *parent_list);

    static void join_trees(int *root1, int *root2, int *weight, int *parent,
                    int *numTrees);
```

```
static int qsort_edge_comparer(const void* xin, const void* yin);

static const char *get_relation_name(PlannerInfo *root, int relid);

static void print_kruskal_graph( PlannerInfo *root, edgeType *edges,
                int numEdges );

#define elog_kruskal \
        if( kruskal_log ) \
                elog_start(__FILE__, __LINE__, PG_FUNCNAME_MACRO); \
        if( kruskal_log ) \
                elog_finish

/**
 * kruskal_joins_search
 *    Query tree optimization by Kruskal's Algorithm.
 */
RelOptInfo *
kruskal_joins_search(PlannerInfo *root, int levels_needed, List *initial_rels)
{
        List       **joinitems;
        RelOptInfo  *rel;
        ListCell    *x;
        int          numEdges=0,
                     numEdge=0;
        edgeType    *edge_list;

        /*
         * Applying the two first levels of the query plan.
         * joinitems[0] not used.
         * joinitems[1] all first order plans (no join).
         * joinitems[2] all 2-relation joins used in kruskal's algorithm.
         */
        joinitems = (List **) palloc0( 3 * sizeof(List *) );
        /*
         * joinitems[1] stores the RelOptInfos of base relations
         */
        joinitems[1] = initial_rels;
        elog_kruskal(NOTICE, "  kruskal_joins_search():   levels_needed: %d",
                        levels_needed);
        elog_kruskal(NOTICE, "  kruskal_joins_search():   joinitems[1]: %d",
                        list_length(joinitems[1]));
        /*
         * joinitems[2] stores the RelOptInfos of the joins between
         * relations pairs in the query.
         * Use for this, either dynamic programming function or
         * kruskal_make_rels_by_joins() in kruskal_2wayplan.c.
         */
        if( kruskal_dynedge ) {
                elog_kruskal(NOTICE,
                        "  kruskal_joins_search():   dynamic programming to 2-way joins" );
                joinitems[2] = join_search_one_level(root, 2, joinitems);
        } else {
                elog_kruskal(NOTICE,
                        "  kruskal_joins_search():   kruskal_make_rels_by_joins() to 2-way joins" );
                joinitems[2] = kruskal_make_rels_by_joins(root, joinitems);
        }
        elog_kruskal(NOTICE, "  kruskal_joins_search():   joinitems[2]: %d",
                        list_length(joinitems[2]));

        /*
         * If the number of relations in query is 2, then return the cheapest
         * plan found out in joinitems[2]. Kruskal is not needed.
         */
        if( levels_needed == 2 ){
                foreach(x, joinitems[2])
                {
                        rel = (RelOptInfo *) lfirst(x);
                        set_cheapest(rel);
                }
                rel = (RelOptInfo *) linitial(joinitems[2]);

                return rel;
```

```
        }

        /*
         * else, build a graph using each item of the joinitems[2] as edge.
         * Allocating the memory epace...
         */
        numEdges = list_length(joinitems[2]);
        edge_list = (edgeType*) palloc0(numEdges * sizeof(edgeType));
        /*
         * Building graph's edges:
         *   Use the join cost as edge weight (heuristic)
         */
        foreach(x, joinitems[2])
        {
                Bitmapset *aux;
                int        relid;

148             rel = (RelOptInfo *) lfirst(x);
                set_cheapest(rel);
                aux = bms_copy( rel->relids ); // copy relations' ids

                edge_list[numEdge].rel = rel; // store the subplan of this edge
                /*
                 * Identify the relation index of each relation evolved in this
                 * join. Needed to do left-deep joins in kruskal() function.
                 */
                relid = bms_first_member(aux);
                edge_list[numEdge].tail_relid = relid;
                edge_list[numEdge].tail_idx = find_index_by_relid(joinitems[1], relid);

                relid = bms_first_member(aux);
                edge_list[numEdge].head_relid = relid;
                edge_list[numEdge].head_idx = find_index_by_relid(joinitems[1], relid);
                /*
                 * Store the join cost (heuristic). Define the order of planner
                 * execution. (further, this heuristic may be improved)
                 */
                edge_list[numEdge].weight = rel->cheapest_startup_path->total_cost;

                numEdge++; // edge_list count
        }

        /*
         * Perform a search for a sub-optimal plan using the minimum spanning tree
         * by Kruskal's algorithm.
         */
        rel = kruskal(root, joinitems, edge_list, numEdges,
                      levels_needed);

        return rel;
}

/**
 * kruskal
185  *    Generate approximation of the optimal plan using the minimum spanning
 *    tree by Kruskal's algorithm.
 *    This function is called only if the query have 3 or more relations.
 * @param initialrels Store the two first levels of optimization.
 * @param edge_list   List of edges generated by kruskal_join_search()
 * @param numEdges    Count(edge_list or initialrels[2])
 * @param numVertices Count(Relations or initialrels[1])
 */
static RelOptInfo *
kruskal(PlannerInfo *root, List **initialrels,
                edgeType *edge_list, int numEdges, int numVertices)
{
        RelOptInfo **subplan_list; /* partial pans of each tree */
        int          i,
                     numTrees, /* number of trees */
                    *parent, /* parent list. Used in kruskal for tree detection */
                    *weight, /* weight list. Used to decide the new root in join
                              * trees */
                     last_join = -1; /* finally, point the index of final plan in
```

```
                                * subplan_list */

elog_kruskal(NOTICE, "  kruskal(): > alocando memória");
parent = (int*) palloc((numVertices) * sizeof(int));
weight = (int*) palloc((numVertices) * sizeof(int));
subplan_list = (RelOptInfo**) palloc((numVertices) * sizeof(RelOptInfo*));
/*
 * Initializing values...
 */
numTrees = numVertices;
for (i=0; i < numVertices; i++)
{
        parent[i] = i;       // todos os vértices são raízes de árvores
        weight[i] = 1;       // todas as árvores têm 1 vértice
        subplan_list[i] = NULL; // ainda não existem novos planos processados
}
elog_kruskal(NOTICE, "  kruskal(): < alocando memória");

222     elog_kruskal(NOTICE, "  kruskal(): > qsort");
/*
 * Sort the edge_list by qsort_edge_comparer().
 * Currently using weight value.
 */
qsort(edge_list, numEdges, sizeof(edgeType), qsort_edge_comparer);
elog_kruskal(NOTICE, "  kruskal(): < qsort");
if( kruskal_log ){
        for (i=0; i < numEdges; i++){
                elog_kruskal(NOTICE, "  kruskal():   edge(%d,%d) weight:%lf",
                                edge_list[i].head_idx, edge_list[i].tail_idx,
                                edge_list[i].weight);
        }
}
elog_kruskal(NOTICE, "  kruskal(): > for numEdges");
/*
 * For each edge or while exists more that 1 sub-tree.
 * Verify whether the edge belong to minimal spanning tree.
 */
for (i=0; i < numEdges && numTrees > 1; i++) // edge-by-edge loop
{
        int root1, root2;

        /*
         * Test the root of each relation in selected edge.
         */
        root1 = find_root(edge_list[i].tail_idx, parent);
        root2 = find_root(edge_list[i].head_idx, parent);
        /*
         * If both roots is not the same, the edge belong to the minimal
         * spanning tree. Join the trees in parent[] and the execution plan
         * in subplan_list[].
         */
        if (root1 != root2)
        {
                int other_root;

259             /*
                 * Debug only. Used in print_kruskal_graph()
                 */
                edge_list[i].used = true;

                /*
                 * Join two trees. root1 is the root of new tree.
                 */
                join_trees(&root1, &root2, weight, parent, &numTrees);
                last_join = root1;
                other_root = root2;

                elog_kruskal(NOTICE,
                                "  kruskal():   edge(%d,%d), last_join:%d, other_root:%d",
                                edge_list[i].tail_idx,edge_list[i].head_idx,
                                last_join,other_root);

                /*
```

```
                         * Juntando planos de execução:
                         */
                        if( ! subplan_list[last_join] ){
                                /*
                                 * First join of tree.
                                 */
                                elog_kruskal(NOTICE, "  kruskal(): > First plan");
                                subplan_list[last_join] = edge_list[i].rel;
                                elog_kruskal(NOTICE, "  kruskal(): < First plan");
                        } else if( ! subplan_list[other_root] ) {
                                /*
                                 * Left-deep join.
                                 * Join one relation to a composed plan.
                                 */
                                RelOptInfo *new_rel = NULL;
                                RelOptInfo *other_rel;
                                elog_kruskal(NOTICE, "  kruskal(): > Left-deep join");

                                other_rel = (RelOptInfo*) list_nth(initialrels[1],other_root);
296                             elog_kruskal(NOTICE, "  kruskal():   > make_join_rel");
                                new_rel = make_join_rel(root, subplan_list[last_join], other_rel);
                                elog_kruskal(NOTICE, "  kruskal():   < make_join_rel");
                                if( new_rel ){
                                        subplan_list[last_join] = new_rel;
                                } else {
                                        elog(ERROR, "Não foi possível fazer left-deep join.");
                                }
                                set_cheapest(subplan_list[last_join]);
                                elog_kruskal(NOTICE, "  kruskal(): < Left-deep join");
                        } else {
                                /*
                                 * Bushy-join.
                                 * Join two composed plans.
                                 */
                                RelOptInfo *new_rel = NULL;
                                elog_kruskal(NOTICE, "  kruskal(): > Bushy join");
                                elog_kruskal(NOTICE, "  kruskal():   > make_join_rel");
                                new_rel = make_join_rel(root, subplan_list[last_join],
                                                                subplan_list[other_root]);
                                elog_kruskal(NOTICE, "  kruskal():   < make_join_rel");
                                if( new_rel ){
                                        subplan_list[last_join] = new_rel;
                                } else {
                                        elog(ERROR, "Não foi possível fazer bushy-join.");
                                }
                                set_cheapest(subplan_list[last_join]);
                                elog_kruskal(NOTICE, "  kruskal(): < Bushy join");
                        }
                }
        }
        elog_kruskal(NOTICE, "  kruskal(): < for numEdges");

        if( last_join == -1 ){ // exception
                elog(ERROR,"Não foi possível gerar o plano.");
                return NULL;
        }
333
        if( kruskal_log )
                print_kruskal_graph( root, edge_list, numEdges );

        return subplan_list[last_join];
}

/**
 * find_index_by_relid
 *   Used in kruskal_join_search.
 *   Search the joinitems index where relid is member of relids.
 */
static int
find_index_by_relid( List *joinitems, int relid )
{
        int count = 0;
        RelOptInfo *rel;
```

```
        ListCell   *x;
        foreach( x, joinitems ){
                rel = (RelOptInfo *) lfirst(x);

                if( bms_is_member(relid,rel->relids))
                        return count;

                count++;
        }
        return -1;
}

/**
 * find_root
 *   Used in kruskal() to search the root of a tree in parent[].
 */
static inline int
find_root(int vertex, int *parent_list)
{
        while( parent_list[vertex] != vertex )
                vertex = parent_list[vertex];

        return vertex;
}

/**
 * join_trees
 *   Join two trees and decrease numTrees.
 *   root1 will be the root of the new tree.
 */
static void
join_trees(int *root1, int *root2, int *weight, int *parent, int *numTrees)
{
        if( weight[*root2]>weight[*root1] ){
                int aux = *root1;
                *root1 = *root2;
                *root2 = aux;
        }
        weight[*root1] += weight[*root2];
        parent[*root2] = parent[*root1];
        (*numTrees)--;
}
/**
 * qsort_edge_comparer
 *   Used in qsort().
 */
static int
qsort_edge_comparer(const void* xin, const void* yin)
{
        edgeType *x,*y;

        x=(edgeType*) xin;
        y=(edgeType*) yin;

        if (x->weight > y->weight)
                return 1;
        else if (x->weight < y->weight)
                return -1;

        return 0;
}

////////////////////////////// log functions ////////////////////////////////

static const char *
get_relation_name( PlannerInfo *root, int relid )
{
        RangeTblEntry *rte;

        Assert(relid <= list_length(root->parse->rtable));
        rte = (RangeTblEntry *) rt_fetch(relid, root->parse->rtable);
        return rte->eref->aliasname;
}
```

```
static void
print_kruskal_graph( PlannerInfo *root, edgeType *edges, int numEdges )
{
        int i, count=1;
        elog(NOTICE, "graph G {" );
        elog(NOTICE, "\tedge [ len = 3.0 fontsize = 7.0 labelfloat = true ];" );
        for( i=0; i<numEdges; i++ ){
                if( edges[i].used ) {
                        elog(NOTICE,
                                        "\t\"%s\" -- \"%s\" [ label=\"%lf (%d)\" style=\"bold\" ];",
                                        get_relation_name(root,edges[i].head_relid),
                                        get_relation_name(root,edges[i].tail_relid),
                                        edges[i].weight, count++ );
                } else {
                        elog(NOTICE,
                                        "\t\"%s\" -- \"%s\" [ label=\"%lf\" ];",
                                        get_relation_name(root,edges[i].head_relid),
                                        get_relation_name(root,edges[i].tail_relid),
                                        edges[i].weight);
                }
444     }
        elog(NOTICE, "}" );
}
```

# APÊNDICE B

# CONSULTAS DESENVOLVIDAS E SUBMETIDAS A BASE DE DADOS DO BENCHMARK TPC-E

```
1
    -- Consulta Q1

    SELECT ZIP_CODE.ZC_TOWN, ADDRESS.AD_LINE1, EXCHANGE.EX_NAME,
    TRADE_TYPE.TT_NAME
    FROM EXCHANGE, ZIP_CODE, ADDRESS, COMMISSION_RATE,TRADE_TYPE
    WHERE ZIP_CODE.ZC_CODE=ADDRESS.AD_ZC_CODE
    AND ADDRESS.AD_ID=EXCHANGE.EX_AD_ID
    AND EXCHANGE.EX_ID=COMMISSION_RATE.CR_EX_ID
    AND COMMISSION_RATE.CR_TT_ID=TRADE_TYPE.TT_ID;

    -- Consulta Q2

    SELECT ZIP_CODE.ZC_TOWN, ADDRESS.AD_LINE1, EXCHANGE.EX_NAME,
    TRADE_TYPE.TT_NAME, T_TRADE_PRICE
    FROM EXCHANGE, ZIP_CODE, ADDRESS, COMMISSION_RATE,TRADE_TYPE,
    TRADE
    WHERE ZIP_CODE.ZC_CODE=ADDRESS.AD_ZC_CODE
    AND ADDRESS.AD_ID=EXCHANGE.EX_AD_ID
    AND EXCHANGE.EX_ID=COMMISSION_RATE.CR_EX_ID
    AND COMMISSION_RATE.CR_TT_ID=TRADE_TYPE.TT_ID
    AND TRADE_TYPE.TT_ID=TRADE.T_TT_ID;


    -- Consulta Q3

    SELECT ZIP_CODE.ZC_TOWN, ADDRESS.AD_LINE1, EXCHANGE.EX_NAME,
    TRADE_TYPE.TT_NAME, T_TRADE_PRICE
    FROM EXCHANGE, ZIP_CODE, ADDRESS, COMMISSION_RATE,TRADE_TYPE,
    TRADE, CUSTOMER_ACCOUNT
    WHERE ZIP_CODE.ZC_CODE=ADDRESS.AD_ZC_CODE
    AND ADDRESS.AD_ID=EXCHANGE.EX_AD_ID
    AND EXCHANGE.EX_ID=COMMISSION_RATE.CR_EX_ID
    AND COMMISSION_RATE.CR_TT_ID=TRADE_TYPE.TT_ID
    AND TRADE_TYPE.TT_ID=TRADE.T_TT_ID
    AND TRADE.T_CA_ID=CUSTOMER_ACCOUNT.CA_ID;
37
    -- Consulta Q4

    SELECT ZIP_CODE.ZC_TOWN, ADDRESS.AD_LINE1, EXCHANGE.EX_NAME,
    TRADE_TYPE.TT_NAME, T_TRADE_PRICE
    FROM EXCHANGE, ZIP_CODE, ADDRESS, COMMISSION_RATE,TRADE_TYPE,
    TRADE, CUSTOMER_ACCOUNT, BROKER
    WHERE ZIP_CODE.ZC_CODE=ADDRESS.AD_ZC_CODE
    AND ADDRESS.AD_ID=EXCHANGE.EX_AD_ID
    AND EXCHANGE.EX_ID=COMMISSION_RATE.CR_EX_ID
    AND COMMISSION_RATE.CR_TT_ID=TRADE_TYPE.TT_ID
    AND TRADE_TYPE.TT_ID=TRADE.T_TT_ID
    AND TRADE.T_CA_ID=CUSTOMER_ACCOUNT.CA_ID
    AND CUSTOMER_ACCOUNT.CA_B_ID=BROKER.B_ID;

    -- Consulta Q5

    SELECT ZIP_CODE.ZC_TOWN, ADDRESS.AD_LINE1, EXCHANGE.EX_NAME,
    TRADE_TYPE.TT_NAME, T_TRADE_PRICE
    FROM EXCHANGE, ZIP_CODE, ADDRESS, COMMISSION_RATE,TRADE_TYPE,
    TRADE,CUSTOMER_ACCOUNT, BROKER, STATUS_TYPE
    WHERE ZIP_CODE.ZC_CODE=ADDRESS.AD_ZC_CODE
```

```
AND ADDRESS.AD_ID=EXCHANGE.EX_AD_ID
AND EXCHANGE.EX_ID=COMMISSION_RATE.CR_EX_ID
AND COMMISSION_RATE.CR_TT_ID=TRADE_TYPE.TT_ID
AND TRADE_TYPE.TT_ID=TRADE.T_TT_ID
AND TRADE.T_CA_ID=CUSTOMER_ACCOUNT.CA_ID
AND CUSTOMER_ACCOUNT.CA_B_ID=BROKER.B_ID
AND BROKER.B_ST_ID=STATUS_TYPE.ST_ID;

-- Consulta Q6

SELECT ZIP_CODE.ZC_TOWN, ADDRESS.AD_LINE1, EXCHANGE.EX_NAME,
 TRADE_TYPE.TT_NAME, T_TRADE_PRICE, CUSTOMER.C_F_NAME
FROM EXCHANGE, ZIP_CODE, ADDRESS, COMMISSION_RATE,TRADE_TYPE,
TRADE, CUSTOMER_ACCOUNT, BROKER, STATUS_TYPE, CUSTOMER
WHERE ZIP_CODE.ZC_CODE=ADDRESS.AD_ZC_CODE
AND ADDRESS.AD_ID=EXCHANGE.EX_AD_ID
AND EXCHANGE.EX_ID=COMMISSION_RATE.CR_EX_ID
AND COMMISSION_RATE.CR_TT_ID=TRADE_TYPE.TT_ID
AND TRADE_TYPE.TT_ID=TRADE.T_TT_ID
AND TRADE.T_CA_ID=CUSTOMER_ACCOUNT.CA_ID
AND CUSTOMER_ACCOUNT.CA_B_ID=BROKER.B_ID
AND BROKER.B_ST_ID=STATUS_TYPE.ST_ID
AND CUSTOMER_ACCOUNT.CA_C_ID=CUSTOMER.C_ID;

-- Consulta Q7

SELECT ZIP_CODE.ZC_TOWN, ADDRESS.AD_LINE1, EXCHANGE.EX_NAME,
TRADE_TYPE.TT_NAME, T_TRADE_PRICE, CUSTOMER.C_F_NAME
FROM EXCHANGE, ZIP_CODE, ADDRESS, COMMISSION_RATE,TRADE_TYPE,
TRADE, CUSTOMER_ACCOUNT, BROKER, STATUS_TYPE, CUSTOMER, SECURITY
WHERE ZIP_CODE.ZC_CODE=ADDRESS.AD_ZC_CODE
AND ADDRESS.AD_ID=EXCHANGE.EX_AD_ID
AND EXCHANGE.EX_ID=COMMISSION_RATE.CR_EX_ID
AND COMMISSION_RATE.CR_TT_ID=TRADE_TYPE.TT_ID
AND TRADE_TYPE.TT_ID=TRADE.T_TT_ID
AND TRADE.T_CA_ID=CUSTOMER_ACCOUNT.CA_ID
AND CUSTOMER_ACCOUNT.CA_B_ID=BROKER.B_ID
AND BROKER.B_ST_ID=STATUS_TYPE.ST_ID
AND CUSTOMER_ACCOUNT.CA_C_ID=CUSTOMER.C_ID
AND TRADE.T_S_SYMB=SECURITY.S_SYMB;

-- Consulta Q8

SELECT ZIP_CODE.ZC_TOWN, ADDRESS.AD_LINE1, EXCHANGE.EX_NAME,
TRADE_TYPE.TT_NAME, T_TRADE_PRICE, CUSTOMER.C_F_NAME
FROM EXCHANGE, ZIP_CODE, ADDRESS, COMMISSION_RATE,TRADE_TYPE,
TRADE, CUSTOMER_ACCOUNT, BROKER, STATUS_TYPE, CUSTOMER, SECURITY,
LAST_TRADE
WHERE ZIP_CODE.ZC_CODE=ADDRESS.AD_ZC_CODE
AND ADDRESS.AD_ID=EXCHANGE.EX_AD_ID
AND EXCHANGE.EX_ID=COMMISSION_RATE.CR_EX_ID
AND COMMISSION_RATE.CR_TT_ID=TRADE_TYPE.TT_ID
AND TRADE_TYPE.TT_ID=TRADE.T_TT_ID
AND TRADE.T_CA_ID=CUSTOMER_ACCOUNT.CA_ID
AND CUSTOMER_ACCOUNT.CA_B_ID=BROKER.B_ID
AND BROKER.B_ST_ID=STATUS_TYPE.ST_ID
AND CUSTOMER_ACCOUNT.CA_C_ID=CUSTOMER.C_ID
AND TRADE.T_S_SYMB=SECURITY.S_SYMB
AND LAST_TRADE.LT_S_SYMB=SECURITY.S_SYMB;

-- Consulta Q9

SELECT ZIP_CODE.ZC_TOWN, ADDRESS.AD_LINE1, EXCHANGE.EX_NAME,
TRADE_TYPE.TT_NAME, T_TRADE_PRICE, CUSTOMER.C_F_NAME, COMPANY.CO_NAME
FROM EXCHANGE, ZIP_CODE, ADDRESS, COMMISSION_RATE,TRADE_TYPE, TRADE,
CUSTOMER_ACCOUNT, BROKER, STATUS_TYPE, CUSTOMER, SECURITY, LAST_TRADE,
COMPANY
WHERE ZIP_CODE.ZC_CODE=ADDRESS.AD_ZC_CODE
AND ADDRESS.AD_ID=EXCHANGE.EX_AD_ID
AND EXCHANGE.EX_ID=COMMISSION_RATE.CR_EX_ID
AND COMMISSION_RATE.CR_TT_ID=TRADE_TYPE.TT_ID
AND TRADE_TYPE.TT_ID=TRADE.T_TT_ID
AND TRADE.T_CA_ID=CUSTOMER_ACCOUNT.CA_ID
```

```
AND CUSTOMER_ACCOUNT.CA_B_ID=BROKER.B_ID
AND BROKER.B_ST_ID=STATUS_TYPE.ST_ID
AND CUSTOMER_ACCOUNT.CA_C_ID=CUSTOMER.C_ID
AND TRADE.T_S_SYMB=SECURITY.S_SYMB
AND LAST_TRADE.LT_S_SYMB=SECURITY.S_SYMB
AND SECURITY.S_CO_ID=COMPANY.CO_ID;

-- Consulta Q10

SELECT ZIP_CODE.ZC_TOWN, ADDRESS.AD_LINE1, EXCHANGE.EX_NAME,
TRADE_TYPE.TT_NAME, T_TRADE_PRICE, CUSTOMER.C_F_NAME, COMPANY.CO_NAME
FROM EXCHANGE, ZIP_CODE, ADDRESS, COMMISSION_RATE,TRADE_TYPE, TRADE,
CUSTOMER_ACCOUNT, BROKER, STATUS_TYPE, CUSTOMER, SECURITY, LAST_TRADE,
COMPANY, NEWS_XREF
WHERE ZIP_CODE.ZC_CODE=ADDRESS.AD_ZC_CODE
AND ADDRESS.AD_ID=EXCHANGE.EX_AD_ID
AND EXCHANGE.EX_ID=COMMISSION_RATE.CR_EX_ID
AND COMMISSION_RATE.CR_TT_ID=TRADE_TYPE.TT_ID
AND TRADE_TYPE.TT_ID=TRADE.T_TT_ID
AND TRADE.T_CA_ID=CUSTOMER_ACCOUNT.CA_ID
AND CUSTOMER_ACCOUNT.CA_B_ID=BROKER.B_ID
AND BROKER.B_ST_ID=STATUS_TYPE.ST_ID
AND CUSTOMER_ACCOUNT.CA_C_ID=CUSTOMER.C_ID
AND TRADE.T_S_SYMB=SECURITY.S_SYMB
AND LAST_TRADE.LT_S_SYMB=SECURITY.S_SYMB
AND SECURITY.S_CO_ID=COMPANY.CO_ID
AND NEWS_XREF.NX_CO_ID=COMPANY.CO_ID;

-- Consulta Q11

SELECT ZIP_CODE.ZC_TOWN, ADDRESS.AD_LINE1, EXCHANGE.EX_NAME,
TRADE_TYPE.TT_NAME, T_TRADE_PRICE, CUSTOMER.C_F_NAME, COMPANY.CO_NAME
FROM EXCHANGE, ZIP_CODE, ADDRESS, COMMISSION_RATE,TRADE_TYPE, TRADE,
CUSTOMER_ACCOUNT, BROKER, STATUS_TYPE, CUSTOMER, SECURITY, LAST_TRADE, COMPANY,
NEWS_XREF, NEWS_ITEM
WHERE ZIP_CODE.ZC_CODE=ADDRESS.AD_ZC_CODE
AND ADDRESS.AD_ID=EXCHANGE.EX_AD_ID
AND EXCHANGE.EX_ID=COMMISSION_RATE.CR_EX_ID
AND COMMISSION_RATE.CR_TT_ID=TRADE_TYPE.TT_ID
AND TRADE_TYPE.TT_ID=TRADE.T_TT_ID
AND TRADE.T_CA_ID=CUSTOMER_ACCOUNT.CA_ID
AND CUSTOMER_ACCOUNT.CA_B_ID=BROKER.B_ID
AND BROKER.B_ST_ID=STATUS_TYPE.ST_ID
AND CUSTOMER_ACCOUNT.CA_C_ID=CUSTOMER.C_ID
AND TRADE.T_S_SYMB=SECURITY.S_SYMB
AND LAST_TRADE.LT_S_SYMB=SECURITY.S_SYMB
AND SECURITY.S_CO_ID=COMPANY.CO_ID
AND NEWS_XREF.NX_CO_ID=COMPANY.CO_ID
AND NEWS_XREF.NX_NI_ID=NEWS_ITEM.NI_ID;

-- Consulta Q12

SELECT ZIP_CODE.ZC_TOWN, ADDRESS.AD_LINE1, EXCHANGE.EX_NAME,
TRADE_TYPE.TT_NAME, T_TRADE_PRICE, CUSTOMER.C_F_NAME, COMPANY.CO_NAME
FROM EXCHANGE, ZIP_CODE, ADDRESS, COMMISSION_RATE,TRADE_TYPE, TRADE,
CUSTOMER_ACCOUNT, BROKER, STATUS_TYPE, CUSTOMER, SECURITY, LAST_TRADE,
COMPANY, NEWS_XREF, NEWS_ITEM, INDUSTRY
WHERE ZIP_CODE.ZC_CODE=ADDRESS.AD_ZC_CODE
AND ADDRESS.AD_ID=EXCHANGE.EX_AD_ID
AND EXCHANGE.EX_ID=COMMISSION_RATE.CR_EX_ID
AND COMMISSION_RATE.CR_TT_ID=TRADE_TYPE.TT_ID
AND TRADE_TYPE.TT_ID=TRADE.T_TT_ID
AND TRADE.T_CA_ID=CUSTOMER_ACCOUNT.CA_ID
AND CUSTOMER_ACCOUNT.CA_B_ID=BROKER.B_ID
AND BROKER.B_ST_ID=STATUS_TYPE.ST_ID
AND CUSTOMER_ACCOUNT.CA_C_ID=CUSTOMER.C_ID
AND TRADE.T_S_SYMB=SECURITY.S_SYMB
AND LAST_TRADE.LT_S_SYMB=SECURITY.S_SYMB
AND SECURITY.S_CO_ID=COMPANY.CO_ID
AND NEWS_XREF.NX_CO_ID=COMPANY.CO_ID
AND NEWS_XREF.NX_NI_ID=NEWS_ITEM.NI_ID
AND COMPANY.CO_IN_ID=INDUSTRY.IN_ID;
```

```
-- Consulta Q13

SELECT ZIP_CODE.ZC_TOWN, ADDRESS.AD_LINE1, EXCHANGE.EX_NAME,
TRADE_TYPE.TT_NAME, T_TRADE_PRICE, CUSTOMER.C_F_NAME, COMPANY.CO_NAME,
SECTOR.SC_NAME
FROM EXCHANGE, ZIP_CODE, ADDRESS, COMMISSION_RATE,TRADE_TYPE, TRADE,
CUSTOMER_ACCOUNT, BROKER, STATUS_TYPE, CUSTOMER, SECURITY, LAST_TRADE,
COMPANY, NEWS_XREF, NEWS_ITEM, INDUSTRY, SECTOR
WHERE ZIP_CODE.ZC_CODE=ADDRESS.AD_ZC_CODE
AND ADDRESS.AD_ID=EXCHANGE.EX_AD_ID
AND EXCHANGE.EX_ID=COMMISSION_RATE.CR_EX_ID
AND COMMISSION_RATE.CR_TT_ID=TRADE_TYPE.TT_ID
AND TRADE_TYPE.TT_ID=TRADE.T_TT_ID
AND TRADE.T_CA_ID=CUSTOMER_ACCOUNT.CA_ID
AND CUSTOMER_ACCOUNT.CA_B_ID=BROKER.B_ID
AND BROKER.B_ST_ID=STATUS_TYPE.ST_ID
AND CUSTOMER_ACCOUNT.CA_C_ID=CUSTOMER.C_ID
AND TRADE.T_S_SYMB=SECURITY.S_SYMB
AND LAST_TRADE.LT_S_SYMB=SECURITY.S_SYMB
AND SECURITY.S_CO_ID=COMPANY.CO_ID
AND NEWS_XREF.NX_CO_ID=COMPANY.CO_ID
AND NEWS_XREF.NX_NI_ID=NEWS_ITEM.NI_ID
AND COMPANY.CO_IN_ID=INDUSTRY.IN_ID
AND INDUSTRY.IN_SC_ID=SECTOR.SC_ID;

-- Consulta Q14

SELECT ZIP_CODE.ZC_TOWN, ADDRESS.AD_LINE1, EXCHANGE.EX_NAME,
TRADE_TYPE.TT_NAME, T_TRADE_PRICE, CUSTOMER.C_F_NAME, COMPANY.CO_NAME,
SECTOR.SC_NAME
FROM EXCHANGE, ZIP_CODE, ADDRESS, COMMISSION_RATE,TRADE_TYPE, TRADE,
CUSTOMER_ACCOUNT, BROKER, STATUS_TYPE, CUSTOMER, SECURITY, LAST_TRADE,
COMPANY, NEWS_XREF, NEWS_ITEM, INDUSTRY, SECTOR, CUSTOMER_TAXRATE
WHERE ZIP_CODE.ZC_CODE=ADDRESS.AD_ZC_CODE
AND ADDRESS.AD_ID=EXCHANGE.EX_AD_ID
AND EXCHANGE.EX_ID=COMMISSION_RATE.CR_EX_ID
AND COMMISSION_RATE.CR_TT_ID=TRADE_TYPE.TT_ID
AND TRADE_TYPE.TT_ID=TRADE.T_TT_ID
AND TRADE.T_CA_ID=CUSTOMER_ACCOUNT.CA_ID
AND CUSTOMER_ACCOUNT.CA_B_ID=BROKER.B_ID
AND BROKER.B_ST_ID=STATUS_TYPE.ST_ID
AND CUSTOMER_ACCOUNT.CA_C_ID=CUSTOMER.C_ID
AND TRADE.T_S_SYMB=SECURITY.S_SYMB
AND LAST_TRADE.LT_S_SYMB=SECURITY.S_SYMB
AND SECURITY.S_CO_ID=COMPANY.CO_ID
AND NEWS_XREF.NX_CO_ID=COMPANY.CO_ID
AND NEWS_XREF.NX_NI_ID=NEWS_ITEM.NI_ID
AND COMPANY.CO_IN_ID=INDUSTRY.IN_ID
AND INDUSTRY.IN_SC_ID=SECTOR.SC_ID
AND CUSTOMER_TAXRATE.CX_C_ID=CUSTOMER.C_ID;

-- Consulta Q15

SELECT ZIP_CODE.ZC_TOWN, ADDRESS.AD_LINE1, EXCHANGE.EX_NAME,
TRADE_TYPE.TT_NAME, T_TRADE_PRICE, CUSTOMER.C_F_NAME, COMPANY.CO_NAME,
SECTOR.SC_NAME, TAXRATE.TX_NAME
FROM EXCHANGE, ZIP_CODE, ADDRESS, COMMISSION_RATE,TRADE_TYPE, TRADE,
CUSTOMER_ACCOUNT, BROKER, STATUS_TYPE, CUSTOMER, SECURITY, LAST_TRADE,
COMPANY, NEWS_XREF, NEWS_ITEM, INDUSTRY, SECTOR, CUSTOMER_TAXRATE, TAXRATE
WHERE ZIP_CODE.ZC_CODE=ADDRESS.AD_ZC_CODE
AND ADDRESS.AD_ID=EXCHANGE.EX_AD_ID
AND EXCHANGE.EX_ID=COMMISSION_RATE.CR_EX_ID
AND COMMISSION_RATE.CR_TT_ID=TRADE_TYPE.TT_ID
AND TRADE_TYPE.TT_ID=TRADE.T_TT_ID
AND TRADE.T_CA_ID=CUSTOMER_ACCOUNT.CA_ID
AND CUSTOMER_ACCOUNT.CA_B_ID=BROKER.B_ID
AND BROKER.B_ST_ID=STATUS_TYPE.ST_ID
AND CUSTOMER_ACCOUNT.CA_C_ID=CUSTOMER.C_ID
AND TRADE.T_S_SYMB=SECURITY.S_SYMB
AND LAST_TRADE.LT_S_SYMB=SECURITY.S_SYMB
AND SECURITY.S_CO_ID=COMPANY.CO_ID
AND NEWS_XREF.NX_CO_ID=COMPANY.CO_ID
AND NEWS_XREF.NX_NI_ID=NEWS_ITEM.NI_ID
```

```
AND COMPANY.CO_IN_ID=INDUSTRY.IN_ID
AND INDUSTRY.IN_SC_ID=SECTOR.SC_ID
AND CUSTOMER_TAXRATE.CX_C_ID=CUSTOMER.C_ID
AND CUSTOMER_TAXRATE.CX_TX_ID=TAXRATE.TX_ID;
```

# APÊNDICE C

# CONSULTAS DO BENCHMARK TPC-H SELECIONADAS PARA OS EXPERIMENTOS REALIZADOS

Consulta TPC-H -Q2

```
-- using 916101011 as a seed to the RNG
-- @(#)2.sql    2.1.8.2
-- TPC-H/TPC-R Minimum Cost Supplier Query (Q2)
-- Functional Query Definition
-- Approved February 1998

select
        s_acctbal,
        s_name,
        n_name,
        p_partkey,
        p_mfgr,
        s_address,
        s_phone,
        s_comment
from
        part,
        supplier,
        partsupp,
        nation,
        region
where
        p_partkey = ps_partkey
        and s_suppkey = ps_suppkey
        and p_size = 29
        and p_type like '%BRASS'
        and s_nationkey = n_nationkey
        and n_regionkey = r_regionkey
        and r_name = 'MIDDLE EAST'
        and ps_supplycost = (
                select
                        min(ps_supplycost)
                from
                        partsupp,
                        supplier,
                        nation,
                        region
                where
                        p_partkey = ps_partkey
                        and s_suppkey = ps_suppkey
                        and s_nationkey = n_nationkey
                        and n_regionkey = r_regionkey
                        and r_name = 'MIDDLE EAST'
        )
order by
        s_acctbal desc,
        n_name,
        s_name,
        p_partkey
LIMIT 100;
```

Consulta TPC-H -Q3

```
-- using 916101011 as a seed to the RNG
-- @(#)3.sql    2.1.8.1
```

```
-- TPC-H/TPC-R Shipping Priority Query (Q3)
-- Functional Query Definition
-- Approved February 1998

select
        l_orderkey,
        sum(l_extendedprice * (1 - l_discount)) as revenue,
        o_orderdate,
        o_shippriority
from
        customer,
        orders,
        lineitem
where
        c_mktsegment = 'FURNITURE'
74      and c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and o_orderdate < date '1995-03-26'
        and l_shipdate > date '1995-03-26'
group by
        l_orderkey,
        o_orderdate,
        o_shippriority
order by
        revenue desc,
        o_orderdate
LIMIT 10;


Consulta TPC-H -Q5

-- using 916101011 as a seed to the RNG
-- @(#)5.sql    2.1.8.1
-- TPC-H/TPC-R Local Supplier Volume Query (Q5)
-- Functional Query Definition
-- Approved February 1998

select
        n_name,
        sum(l_extendedprice * (1 - l_discount)) as revenue
from
        customer,
        orders,
        lineitem,
        supplier,
        nation,
        region
where
        c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and l_suppkey = s_suppkey
        and c_nationkey = s_nationkey
111     and s_nationkey = n_nationkey
        and n_regionkey = r_regionkey
        and r_name = 'MIDDLE EAST'
        and o_orderdate >= date '1995-01-01'
        and o_orderdate < date '1995-01-01' + interval '1 year'
group by
        n_name
order by
        revenue desc;


Consulta TPC-H -Q7

-- using 916101011 as a seed to the RNG
-- @(#)7.sql    2.1.8.1
-- TPC-H/TPC-R Volume Shipping Query (Q7)
-- Functional Query Definition
-- Approved February 1998

select
        supp_nation,
```

```
        cust_nation,
        l_year,
        sum(volume) as revenue
from
        (
                select
                        n1.n_name as supp_nation,
                        n2.n_name as cust_nation,
                        extract(year from l_shipdate) as l_year,
                        l_extendedprice * (1 - l_discount) as volume
                from
                        supplier,
                        lineitem,
                        orders,
                        customer,
                        nation n1,
148                     nation n2
                where
                        s_suppkey = l_suppkey
                        and o_orderkey = l_orderkey
                        and c_custkey = o_custkey
                        and s_nationkey = n1.n_nationkey
                        and c_nationkey = n2.n_nationkey
                        and (
                                (n1.n_name = 'IRAQ' and n2.n_name = 'CANADA')
                                or (n1.n_name = 'CANADA' and n2.n_name = 'IRAQ')
                        )
                        and l_shipdate between date '1995-01-01' and date '1996-12-31'
        ) as shipping
group by
        supp_nation,
        cust_nation,
        l_year
order by
        supp_nation,
        cust_nation,
        l_year;


Consulta TPC-H -Q8

-- using 916101011 as a seed to the RNG
-- @(#)8.sql    2.1.8.1
-- TPC-H/TPC-R National Market Share Query (Q8)
-- Functional Query Definition
-- Approved February 1998

select
        o_year,
        sum(case
                when nation = 'CANADA' then volume
                else 0
        end) / sum(volume) as mkt_share
185 from
        (
                select
                        extract(year from o_orderdate) as o_year,
                        l_extendedprice * (1 - l_discount) as volume,
                        n2.n_name as nation
                from
                        part,
                        supplier,
                        lineitem,
                        orders,
                        customer,
                        nation n1,
                        nation n2,
                        region
                where
                        p_partkey = l_partkey
                        and s_suppkey = l_suppkey
                        and l_orderkey = o_orderkey
                        and o_custkey = c_custkey
```

```
                        and c_nationkey = n1.n_nationkey
                        and n1.n_regionkey = r_regionkey
                        and r_name = 'AMERICA'
                        and s_nationkey = n2.n_nationkey
                        and o_orderdate between date '1995-01-01' and date '1996-12-31'
                        and p_type = 'PROMO POLISHED NICKEL'
        ) as all_nations
group by
        o_year
order by
        o_year;


Consulta TPC-H -Q9

-- using 916101011 as a seed to the RNG
-- @(#)9.sql    2.1.8.1
-- TPC-H/TPC-R Product Type Profit Measure Query (Q9)
-- Functional Query Definition
-- Approved February 1998

select
        nation,
        o_year,
        sum(amount) as sum_profit
from
        (
                select
                        n_name as nation,
                        extract(year from o_orderdate) as o_year,
                        l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount
                from
                        part,
                        supplier,
                        lineitem,
                        partsupp,
                        orders,
                        nation
                where
                        s_suppkey = l_suppkey
                        and ps_suppkey = l_suppkey
                        and ps_partkey = l_partkey
                        and p_partkey = l_partkey
                        and o_orderkey = l_orderkey
                        and s_nationkey = n_nationkey
                        and p_name like '%turquoise%'
        ) as profit
group by
        nation,
        o_year
order by
        nation,
        o_year desc;


Consulta TPC-H -Q10

-- using 916101011 as a seed to the RNG
-- @(#)10.sql   2.1.8.1
-- TPC-H/TPC-R Returned Item Reporting Query (Q10)
-- Functional Query Definition
-- Approved February 1998

select
        c_custkey,
        c_name,
        sum(l_extendedprice * (1 - l_discount)) as revenue,
        c_acctbal,
        n_name,
        c_address,
        c_phone,
        c_comment
from
```

```
        customer,
        orders,
        lineitem,
        nation
where
        c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and o_orderdate >= date '1994-11-01'
        and o_orderdate < date '1994-11-01' + interval '3 month'
        and l_returnflag = 'R'
        and c_nationkey = n_nationkey
group by
        c_custkey,
        c_name,
        c_acctbal,
        c_phone,
        n_name,
        c_address,
296     c_comment
order by
        revenue desc
LIMIT 20;


Consulta TPC-H -Q11

-- using 916101011 as a seed to the RNG
-- @(#)11.sql   2.1.8.1
-- TPC-H/TPC-R Important Stock Identification Query (Q11)
-- Functional Query Definition
-- Approved February 1998

\timing
Explain analyze
select
        ps_partkey,
        sum(ps_supplycost * ps_availqty) as value
from
        partsupp,
        supplier,
        nation
where
        ps_suppkey = s_suppkey
        and s_nationkey = n_nationkey
        and n_name = 'KENYA'
group by
        ps_partkey having
                sum(ps_supplycost * ps_availqty) > (
                        select
                                sum(ps_supplycost * ps_availqty) * 0.0001000000
                        from
                                partsupp,
                                supplier,
                                nation
                        where
333                             ps_suppkey = s_suppkey
                                and s_nationkey = n_nationkey
                                and n_name = 'KENYA'
                )
order by
        value desc;


Consulta TPC-H -Q18

-- using 916101011 as a seed to the RNG
-- @(#)18.sql   2.1.8.1
-- TPC-H/TPC-R Large Volume Customer Query (Q18)
-- Function Query Definition
-- Approved February 1998

select
        c_name,
```

```
        c_custkey,
        o_orderkey,
        o_orderdate,
        o_totalprice,
        sum(l_quantity)
from
        customer,
        orders,
        lineitem
where
        o_orderkey in (
                select
                        l_orderkey
                from
                        lineitem
                group by
                        l_orderkey having
                                sum(l_quantity) > 313
        )
370     and c_custkey = o_custkey
        and o_orderkey = l_orderkey
group by
        c_name,
        c_custkey,
        o_orderkey,
        o_orderdate,
        o_totalprice
order by
        o_totalprice desc,
        o_orderdate
LIMIT 100;


Consulta TPC-H -Q20

-- using 916101011 as a seed to the RNG
-- @(#)20.sql   2.1.8.1
-- TPC-H/TPC-R Potential Part Promotion Query (Q20)
-- Function Query Definition
-- Approved February 1998

select
        s_name,
        s_address
from
        supplier,
        nation
where
        s_suppkey in (
                select
                        distinct (ps_suppkey)
                from
                        partsupp,
                        part
                where
                        ps_partkey=p_partkey
407                     and p_name like 'drab%'
                        and ps_availqty > (
                                select
                                        0.5 * sum(l_quantity)
                                from
                                        lineitem
                                where
                                        l_partkey = ps_partkey
                                        and l_suppkey = ps_suppkey
                                        and l_shipdate >= '1993-01-01'
                                        and l_shipdate < date '1993-01-01' + interval '1 year'
                        )
        )
        and s_nationkey = n_nationkey
        and n_name = 'UNITED STATES'
order by
        s_name;
```

Consulta TPC-H -Q21

```
-- using 916101011 as a seed to the RNG
-- @(#)21.sql   2.1.8.1
-- TPC-H/TPC-R Suppliers Who Kept Orders Waiting Query (Q21)
-- Functional Query Definition
-- Approved February 1998

select
        s_name,
        count(*) as numwait
from
        supplier,
        lineitem l1,
        orders,
        nation
where
        s_suppkey = l1.l_suppkey
444     and o_orderkey = l1.l_orderkey
        and o_orderstatus = 'F'
        and l1.l_receiptdate > l1.l_commitdate
        and exists (
                select
                        *
                from
                        lineitem l2
                where
                        l2.l_orderkey = l1.l_orderkey
                        and l2.l_suppkey <> l1.l_suppkey
        )
        and not exists (
                select
                        *
                from
                        lineitem l3
                where
                        l3.l_orderkey = l1.l_orderkey
                        and l3.l_suppkey <> l1.l_suppkey
                        and l3.l_receiptdate > l3.l_commitdate
        )
        and s_nationkey = n_nationkey
        and n_name = 'CANADA'
group by
        s_name
order by
        numwait desc,
        s_name
LIMIT 100;
```

# REFERÊNCIAS BIBLIOGRÁFICAS

[1] A measure of transaction processing power. *Datamation*, 31(7):112–118, 1985.

[2] Guttoski P. B., Sunye M. S., e Silva F. Kruskal's algorithm for query tree optimization. *IDEAS '07: Proceedings of the 11th International Database Engineering and Applications Symposium*, páginas 296–302, Washington, DC, USA, 2007. IEEE Computer Society.

[3] Cormen T. H. and Leiserson C. E. and Rivest R. L. and Stein C. . *Algoritmos: Teoria e Prática.* Campus, 2ª edition, 2002.

[4] DeWitt D.J. The wisconsin benchmark: Past, present, and future. *The Benchmark Handbook*. 1993.

[5] DeWitt D.J. e Levine C. Not just correct, but correct and fast: a look at one of jim gray's contributions to database system performance. *SIGMOD Rec.*, 37(2):45–49, 2008.

[6] Hongbin Dong e Yiwen Liang. Genetic algorithms for large join query optimization. *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, páginas 1211–1218, New York, NY, USA, 2007. ACM.

[7] Goldberg D. E. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Company, Alabama, 1989.

[8] Eduardo Cunha de Almeida. Estudo de Viabilidade de uma Plataforma de Baixo Custo para Data Warehouse. Dissertação de mestrado, Departamento de Informática, UFPR, Junho de 2004. http://dspace.c3sl.ufpr.br/dspace/bitstream/1884/662/1/EduardoCunhadeAlmeida.pdf.

[9] Eduardo Maria Terra Morelli. Recriação Automática de Índices em um SGBD Relacional. Dissertação de mestrado, Pontifícia Universidade Católica do Rio de Janeiro,

Setembro de 2006. http://www2.dbd.puc-rio.br/pergamum/tesesabertas/0410827-06-pretextual.pdf.

[10] Selinger P. G., Astrahan M. M., Chamberlin D. D., Lorie D. D., e Price T. G. Access path selection in a relational database management system. *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, páginas 23–34, New York, NY, USA, 1979. ACM.

[11] Horácio A. Braga Fernandes. Teoria de Linguagens - Máquinas de Turing Probabilísticas. Relatório técnico, Universidade Federal de Minas Gerais - Departamento de Ciência da Computação, 2005.

[12] Bennett K., Ferris M. C., e Ioannidis Y. E. A genetic algorithm for database query optimization. *In Proceedings of the fourth International Conference on Genetic Algorithms*, páginas 400–407. Morgan Kaufmann Publishers, 1991.

[13] Kruskal J. B. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. Relatório técnico, Proc. Amer. Math. Soc, 1956.

[14] Poess M., Nambiar R. O., e Walrath D. Why you should run tpc-ds: a workload analysis. *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, páginas 1138–1149. VLDB Endowment, 2007.

[15] Steinbrunn M., Moerkotte G, e Kemper A. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, 1997.

[16] Murilo Rodrigues de Lima. Execução Distribuída de Benchmarks em Sistemas de Bancos de Dados Relacionais . Dissertação de mestrado, Departamento de Informática, UFPR, Agosto de 2008. http:verificar.

[17] Nascimento R. O. and Wong M. and Maciel P. R. M. DBT-5: A Fair Usage Open-Source TPC-E Implementation for Performance Evaluation of Computer Systems. páginas 9, 2008.

[18] Neto A. P. *PostgreSQL: Tecnicas Avançadas: Versões Open Source 7.x e 8.x: Soluções para Desenvolvedores e Administradores de Bancos de Dados.* Érica, 4ª edition, 2007.

[19] Poess M. and Chris Floyd. New TPC Benchmarks for Decision Support and Web Commerce. páginas 64–71, Dezembro de 2000.

[20] PostgreSQL: The World1's most advanced open source database. `http://www.postgresql.org/`, acessado em 15/02/2008.

[21] Pryscila Barvick Guttoski. Otimização de Consultas no PostgreSQL Utilizando o Algoritmo de Kruskal. Dissertação de mestrado, Departamento de Informática, UFPR, Junho de 2006. http://dspace.c3sl.ufpr.br/dspace/bitstream/1884/7868/1/dissertacao.pdf.

[22] Ramakrishnan R. and Gehrke J. *Database Management Systems.* McGraw-Hill, 3rd edition, 2003.

[23] Rilson Oscar do Nascimento. DBT-5: Uma Implementação de Código Aberto do TPC-E para Avaliação de Desempenho de Sistemas de Processamento Transacional . Dissertação de mestrado, Centro de Informática da Universidade Federal de Pernambuco, UFPE, Agosto de 2008. http:verificar.

[24] Silberschatz A. and Korth H.F. and Sudarshan H.F. *Sistema de Banco de Dados.* Makron Books, 3ª edition, 1999.

[25] Ibaraki T. e Kameda T. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems*, 9(3):482–502, 1984.

[26] The PostgreSQL Global Development Group. PostgreSQL 8.3.1 Documentation. Relatório técnico, PostgreSQL Global Development Group, 2008.

[27] A Tour of PostgreSQL Internals. Open Source Database conference in October 2000. `http://www.postgresql.org/files/developer/tour.pdf`, acessado em 15/06/2008.

[28] Transaction Processing Performance Council. http://www.tpc.org/, acessado em 11/03/2008.

[29] TCP-App-Application Server. http://www.tpc.org/tpc_app/default.asp, acessado em 01/07/2008.

[30] TCP-H. http://www.tpc.org/tpch/default.asp, acessado em 20/03/2008.

[31] Transaction Processing Performance Council. TPC Benchmark DS - Decision Support. Relatório técnico, Transaction Processing Performance Council, 2005.

[32] Transaction Processing Performance Council. TPC Benchmark C. Relatório técnico, Transaction Processing Performance Council, 2007.

[33] Transaction Processing Performance Council. TPC Benchmark App - Application Server. Relatório técnico, Transaction Processing Performance Council, 2008.

[34] Transaction Processing Performance Council. TPC BENCHMARK E - Standard Specification - Version 1.6.0. Relatório técnico, Transaction Processing Performance Council, 2008.

[35] Transaction Processing Performance Council. TPC Benchmark H (Decision Support) Standard Specification - Revision 2.7.0. Relatório técnico, Transaction Processing Performance Council, 2008.

TARCIZIO ALEXANDRE BINI

# APLICAÇÃO DO ALGORITMO DE KRUSKAL NA OTIMIZAÇÃO DE CONSULTAS COM MÚLTIPLAS JUNÇÕES RELACIONAIS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Marcos Sfair Sunye.

CURITIBA

2009