PRYSCILA BARVICK GUTTOSKI

OTIMIZAÇÃO DE CONSULTAS NO POSTGRESQL UTILIZANDO O ALGORITMO DE KRUSKAL

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

2006

PRYSCILA BARVICK GUTTOSKI

OTIMIZAÇÃO DE CONSULTAS NO POSTGRESQL UTILIZANDO O ALGORITMO DE KRUSKAL

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

2006

PRYSCILA BARVICK GUTTOSKI

OTIMIZAÇÃO DE CONSULTAS NO POSTGRESQL UTILIZANDO O ALGORITMO DE KRUSKAL

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

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

Prof. Dr. Antonio Miguel Vieira Monteiro
INPE, membro externo

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

Curitiba, 22 de junho de 2006

AGRADECIMENTOS

Ao meu orientador prof. Marcos Sfair Sunye por ter acreditado em minha capacidade e me incentivado, e pelas sugestões valiosas na construção deste trabalho.

Aos professores Marcos Alexandre Castilho e Fabiano Silva por suas grandes contribuições na definição do tema deste trabalho.

Aos professores do Departamento de Informática da UFPR que lutam para tornar cada vez melhor o curso de Mestrado em Ciência da Computação nesta universidade. Em especial ao prof. Alexandre Ibrahim Direne pela dedicação na coordenação do curso e atenção demonstrada com os alunos.

Ao meu querido Rafael pela extrema ajuda durante todo o curso, seu incentivo e participação no desenvolvimento da pesquisa, sua força nos momentos mais difíceis deste trabalho e pela paciência, companheirismo e amor. Muito obrigada.

Aos meus pais e irmã que sempre me apoiaram em todos os desafios e conquistas, e pela paciência em todos os momentos.

Ao meu amigo e diretor de Tecnologia da Informação da RPC Denilson Farias, por ter me encorajado a realizar o curso e pelo valioso apoio em diversos aspectos que me foi dado durante a toda realização do mestrado. Muito obrigada.

Ao meu amigo e coordenador de Tecnologia da Informação da RPC Martin Alain Kretschek que não mediu esforços para ajudar sempre que possível e pela paciência e disposição em realizar a correção deste texto. Muito obrigada.

Aos meus colegas de trabalho na compreensão das atividades do cotidiano e em especial ao Tiago de Oliveira que configurou o equipamento para a realização dos experimentos.

Aos meus colegas de curso pelo apoio e amizade, em especial a Andréia Aparecida Barbiero, Luciane Cristina Pinheiro, Adriana Zanella Martinhago e Regina de Cássia Nandi pela ajuda na elaboração do texto.

A todos meus amigos que se preocuparam e torceram, e assim participaram da elaboração deste trabalho, direta ou indiretamente.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

O papel da otimização de consultas é promover a recuperação dos dados desejados no menor tempo possível. Diversos algoritmos são utilizados nos atuais Sistemas Gerenciadores de Bancos de Dados (SGBD) para realizar a otimização de consultas e determinar a melhor ordem de execução das junções. Os algoritmos de programação dinâmica possuem um custo computacional elevado para determinar a solução ótima, enquanto os algoritmos heurísticos e aleatórios possuem custo computacional menor mas podem alcançar como resultado soluções que estão longe de serem ótimas. Encontrar um algoritmo que seja executado em um tempo razoável e que indique como resultado ao menos uma solução próxima da ótima significa obter um otimizador de consultas extremamente eficiente.

Este trabalho apresenta a implementação do algoritmo de Kruskal no processo de otimização de consultas do PostgreSQL. Esse é um algoritmo guloso *bottom-up* que sempre executa primeiro a junção com maior ganho para formar a árvore geradora mínima do grafo. Dentre os algoritmos existentes foi escolhido o algoritmo de Kruskal por sua simplicidade de implementação e possibilidade de adequação da sua definição ao processo de otimização de consultas.

O SGBD escolhido para a realização dos experimentos foi o PostgreSQL por possuir um conjunto complexo de recursos e por ser um *software* de código aberto. Na maioria dos experimentos realizados o algoritmo de Kruskal retornou os resultados esperados em tempos bastante próximos dos obtidos com os algoritmos implementados no PostgreSQL, com a vantagem da simplicidade de implementação do algoritmo de Kruskal e do seu espaço de busca reduzido enquanto apenas uma árvore de execução é gerada como solução ótima. Os resultados demonstram que o algoritmo de Kruskal é um algoritmo viável para a técnica de otimização de consultas, faltando no futuro definir quais conjuntos de consultas são beneficiados por este algoritmo.

ABSTRACT

The query optimization purpose is to retrieve the information as quickly as possible. Many different algorithms are implemented on the modern Database Management Systems (DBMS) aiming the query optimization and in order to define the optimizing join order. The optimization by Dynamic programming finds optimal solutions but has high costs of execution. On the other hand, heuristcs and randomized algorithms have lower execution costs, however there is no guarantee a good solution will be found using this methods. Therefore, developing an algorithm able to find a sub-optimal solution in a reasonable time means to have an extremely efficient query optimizer.

This work presents Kruskal's algorithm in query optimization process, which is a bottom-up greedy algorithm that always performs most profitable joins first to reduce query graph to its minimal spanning tree. Kruskal's algorithm has been chosen because of its easy implementation and because it is possible to apply its defition on query optimization process.

PostgreSQL DBMS has been chosen to perform the experiments because it offers full support for many sophisticated features, is an open source relational database system and performs its query optimization by using Dynamic programming and heuristics algorithms. These features allow a concrete comparison between our approach and the existing ones. On most of tests, Kruskal's algorithm got the expected results in almost the same time as the results reached by default PostgreSQL's optimization algorithms, with the advantage on the simplicity of Kruskal's algorithm implementation and its reduced search space necessary in order to generate only one optimal execution query tree. The results confirm us that Kruskal's algorithm is a viable method for query optimization while a future work is to perform a broad test to determine, more precisely, witch set of queries would be benefited by this algorithm.

CAPÍTULO 1

INTRODUÇÃO

O grande volume de informações geradas diariamente por empresas, instituições e pessoas demanda cada vez mais o armazenamento destes dados em sistemas computacionais. Um dos maiores objetivos do armazenamento em sistemas computacionais é a possibilidade de recuperação dessas informações de forma precisa, segura e rápida. Os SGBDs atuais atendem aos requisitos de precisão, segurança e velocidade na obtenção de dados e estão continuamente em evolução, motivando diversos estudos na área de Banco de Dados.

O tempo necessário para a recuperação de informações depende de diversos fatores físicos (*hadware*) e lógicos (*software*). Dentre os fatores físicos é possível mencionar a velocidade e a capacidade do disco no qual os dados estão armazenados, a quantidade de memória disponível e a capacidade de processamento do computador. Os fatores lógicos envolvem o método utilizado na recuperação dos dados e o projeto das complexas estruturas que representam os dados no banco de dados. Este estudo tem como foco os métodos de recuperação de dados.

A manipulação de dados nos bancos de dados envolve recuperação, inserção, modificação e eliminação de informações. A Linguagem de Manipulação de Dados (DML) é a linguagem que viabiliza o acesso e a manipulação dos dados. *DMLs procedurais* exigem que o usuário especifique quais dados deseja e a forma como esses dados serão obtidos. *DMLs não procedurais* exigem que o usuário informe quais dados deseja, sem especificar como obtê-los.

Uma *consulta* é uma solicitação para recuperação das informações armazenadas no banco de dados. A parte de uma DML responsável pela recuperação de informações é chamada de *linguagem de consultas* (*query language*).

Os primeiros modelos de bancos de dados foram o Modelo de Rede e o Modelo Hierárquico. Nesses modelos, os dados são representados por registros e os relaciona-

mentos entre os registros são representados por ponteiros (*links*), deixando a cargo do programador descrever quais ponteiros devem ser acessados para que os dados de que necessita sejam recuperados, ou seja, a forma como a consulta é executada é determinada pelo programador em uma DML procedural. Isto é implícito ao código do programa e à estrutura dessas classes de bancos de dados [32].

Nos SGBDs atuais, o Modelo Relacional é um dos modelos mais utilizados. Esse modelo difere dos modelos Hierárquico e de Rede por não possuir ponteiros, permitindo que o relacionamento entre os dados seja realizado baseado em fundamentos matemáticos. A álgebra relacional consiste em um conjunto de operações que podem ser efetuadas sobre uma ou duas relações, produzindo como resultado uma nova relação. A álgebra relacional é uma DML procedural.

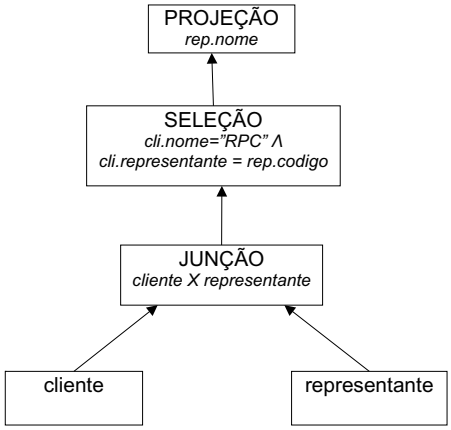
Buscando a facilidade de uso e aproveitando o alto nível de abstração encontrado nas DML não procedurais, as consultas nos SGBDs relacionais são descritas pelos usuários através desse tipo de linguagem, indicando apenas o que se deseja como resultado e não a maneira como os dados devem ser recuperados. Essa independência dos dados foi introduzida por Codd em [7] para os modelos relacionais.

A *Structured Query Language* (SQL) [6] utilizada nos SGBDs relacionais é um exemplo de DML não procedural. Além de linguagem de consulta, possui recursos para definição da estrutura de dados, modificações de dados e especificações de restrições de segurança.

O grande problema do alto nível de abstração é que uma simples consulta SQL pode ser traduzida para várias seqüências diferentes de operações de baixo-nível, que são as operações físicas baseadas na álgebra relacional executadas pelo processador, e existe uma grande variação entre os custos computacionais dessas seqüências. A sequência de operações físicas normalmente é representada como uma árvore de operações chamada *árvore de execução de consultas*, pois esta estrutura permite representar a ordem em que as operações são executadas.

Considerando as operações relacionais projeção (π), seleção (σ) e junção (X). Para a consulta SQL abaixo, as duas expressões em álgebra relacional da figura 1.1 forneceriam o resultado correto para a consulta.

1) π (rep.nome) (σ (cli.nome="RPC" Λ cli.representante = rep.codigo) (cliente X representante))



2) π (rep.nome) (σ (cli.representante = rep.codigo) (σ ((cli.nome="RPC") (cliente)) X representante)

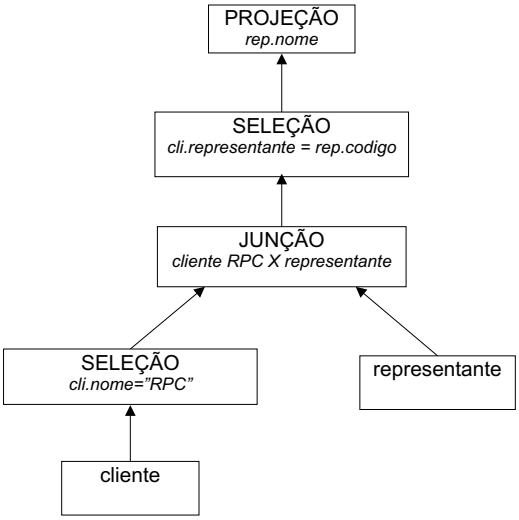


Figura 1.1: Expressões em álgebra relacional e suas respectivas árvores de execução.

```
SELECT rep.nome
FROM cliente cli, representante rep
WHERE cli.nome = "RPC" AND cli.representante = rep.codigo
```

Considerando a relação Cliente com 10.000 registros e a relação Representante com 100 registros. Na primeira expressão, a junção entre as duas relações é um produto cartesiano que gera um resultado de 1.000.000 tuplas. Na segunda expressão, as tuplas recuperadas da relação Cliente são limitadas àquelas nas quais o Cliente é "RPC"para então realizar a junção desse conjunto com a relação Representante. Desta forma, o processamento da segunda expressão envolve um número menor de tuplas do que o processamento da primeira expressão.

Pela simples reescrita da expressão que será executada é possível evitar um número grande de acessos e comparações de dados e por isso a escolha do melhor plano de execução é tão importante. O processo de selecionar o plano de execução mais eficiente para uma consulta é chamado *otimização de consultas*.

1.1 Motivação

A operação de junção é o alvo principal no processo de otimização de consultas, devido ao seu alto custo de execução. Diversos algoritmos são utilizados nos atuais Sistemas Gerenciadores de Bancos de Dados (SGBD) para realizar a otimização de consultas e indicar a melhor ordem para execução das junções, entre eles algoritmos de programação dinâmica, algoritmos heurísticos e algoritmos aleatórios. Os algoritmos de programação dinâmica possuem um custo computacional elevado para determinar a solução ótima, enquanto os algoritmos heurísticos e aleatórios possuem custos computacionais menores, mas podem alcançar como resultado soluções que estão longe de serem ótimas. Encontrar um algoritmo que seja executado em um tempo razoável e que indique como resultado ao menos uma solução próxima da ótima significa obter um otimizador de consultas extremamente eficiente.

Um dos algoritmos mais utilizados é o algoritmo de programação dinâmica, com uma técnica de buscas exaustivas para gerar todas as árvores de execução possíveis relacionadas a uma determinada consulta SQL. O algoritmo é guiado por uma técnica baseada no cálculo da estimativa de custo das árvores. As árvores que apresentam custos altos são eliminadas, até que permaneça a árvore com o menor custo. Esta árvore é considerada ótima e é utilizada para recuperar as informações solicitadas.

O algoritmo de programação dinâmica certamente indicará a árvore de execução ótima pois todos os possíveis planos de execução serão gerados. O problema é que a quantidade de alternativas de planos aumenta exponencialmente com o aumento do número de relações existentes na consulta, ilustrado na tabela 1.1, e o tempo e espaço necessários para criação de todos os possíveis planos utilizando o algoritmo de programação dinâmica tornam-se impraticáveis.

♯ 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	1.430	518.918.400
10	4.862	17.643.225.600
11	16.796	670.442.572.800
12	58.786	28.158.588.057.600
...	...	...

Tabela 1.1: Possíveis sequências de junções obtidas na avaliação de uma consulta

O problema de otimização de consultas é NP-completo, conforme provado por Ibaraki e Kameda em [36]. Sendo assim, para otimizar consultas formadas por um número elevado de relações são utilizadas outras classes de algoritmos, como os algoritmos heurísticos e os algoritmos aleatórios, que indicam uma solução próxima da solução ótima dentro de um tempo aceitável.

Este trabalho apresenta o funcionamento do algoritmo de Kruskal no processo de otimização de consultas e avalia a sua utilização. Dentre os algoritmos existentes foi escolhido o algoritmo de Kruskal por sua simplicidade de implementação e possibilidade de adequação da sua definição ao processo de otimização de consultas, além da sua referência na literatura em [5, 10, 3, 20].

A implementação de um SGBD é uma atividade extremamente complexa e inviável em uma pesquisa de curto prazo. O movimento dos *softwares* de código aberto (*open source software*) foi fundamental no desenvolvimento deste trabalho, pois com a disponibilidade do código fonte do SGBD `PostgreSQL` [1] em um ambiente de desenvolvimento também de código livre envolvendo o GNU/Linux e C/C++, foi possível fazer a implementação do algoritmo proposto e observar o seu comportamento em um ambiente real.

1.2 Trabalhos Relacionados

O primeiro projeto que apresentou avanços significativos na otimização de consultas de SGBDs relacionais foi o projeto System-R [12]. Esse projeto utiliza programação dinâmica com busca exaustiva na geração dos planos de execução de consultas. Foi incorporado em diversos sistemas comerciais e até hoje seu núcleo é utilizado por otimizadores de SGBDs, geralmente com algumas variações ou extensões, como no `PostgreSQL`.

O espaço de busca do otimizador System-R é formado por árvores de execução de consultas que correpondem a sequências lineares de operações de junção. Essas sequências são equivalentes devido às propriedades de associatividade e comutatividade que a operação de junção apresenta. Cada operação é representada por um nó da árvore. As operações primárias são as buscas nas tabelas, que podem ser sequenciais (*sequencial scan*) ou através de índices (*index scan*), e a avaliação de predicados, que são os filtros das cláusulas *where*. A avaliação de predicados é prioritária e ocorre assim que possível na árvore de execução de consultas. A junção pode ser implementada através de *nested loop* ou *merge sort*.

O modelo de custo indica o custo estimado para cada plano que se encontra no espaço de busca. A estimativa do custo é calculada com base em dados estatísticos sobre as tabelas, seus índices e atributos. Esses dados são usados em fórmulas que calculam a seletividade dos predicados e a partir disso são obtidas as estimativas de custos de processamento e acesso a disco para retornar um conjunto de dados. Uma técnica de corte (*prunning*) que elimina os planos de execução com custo elevado é utilizada a cada passo do processo de otimização. Sem a técnica de corte, o otimizador tornaria-se extremamente lento.

No sistema de banco de dados relacional distribuído MICROBE [16] foram utilizadas regras de transformações algébricas. O objetivo do otimizador baseado em regras MICROBE era minimizar o número de operadores e a quantidade de dados carregada entre os operadores. Um conjunto de regras de transformação foi formulado e provado para garantir um resultado determinístico, independente da atual sequência de transformações. O otimizador MICROBE utiliza no máximo O(N log N) passos, onde N é o número de operadores da consulta.

O projeto Starburst da IBM [15] consite de dois subsistemas baseados em regras, os quais são conectados por uma representação estrutural da consulta SQL que é utilizada em todo o ciclo de otimização. Essa representação é chamada de *Query Graph Model* (QGM), na qual as caixas representam um bloco da consulta e as arestas entre as caixas representam as ligações entre os blocos. Cada caixa contém informações sobre os predicados e sobre como o conjunto de dados será ordenado.

O primeiro subsistema contém a fase de reescrita da consulta. Nesta fase, a estrutura QGM é transformada em outra equivalente, a partir de regras de transformações. Uma máquina de regras coordena a aplicação dessas regras, que podem ser agrupadas em classes, e ajusta a ordem de avaliação das mesmas. Como a aplicação de regras resulta em uma QGM válida, qualquer conjunto de regras aplicadas garante uma consulta equivalente. Na fase de reescrita de consultas não existe informação sobre o custo das operações e QGMs, obrigando a utilização de heurística.

A segunda fase é chamada de otimização de plano. Nesta fase, um plano de execução é escolhido a partir de uma QGM. No projeto Starburst, os operadores físicos são chamados de LOLEPOPs e podem ser combinados de diversas formas. A realização das operações de alto-nível são expressadas por derivações dos operadores físicos. Na realização das derivações, planos que são equivalentes nas propriedades físicas e lógicas mas que contêm um alto custo são eliminados. Cada plano contém (1) a descrição relacional correspondente à expressão algébrica que ele representa, (2) a estimativa de custo e (3) as propriedades físicas, como ordem (*sort order*). Os planos são construídos no esquema *bottom-up*, assim como no projeto System-R, e dessa maneira as propriedades físicas são propagadas.

Outro trabalho importante na otimização de consultas envolve os projetos EXODUS [18], Volcano [19] e Cascades [13]. O projeto EXODUS foi o primeiro desenvolvido e deu origem aos outros dois projetos. Nesses sistemas, regras são intensivamente utilizadas para popular o espaço de busca. Existem as regras de transformação, que mapeiam as expressões algébricas em outras expressões algébricas, e as regras de implementação, que mapeiam as expressões algébricas em árvores de operações físicas. As regras podem ter condições para sua aplicação.

No projeto EXODUS, regras de transformações algébricas são traduzidas dentro de um otimizador de consultas executável, o qual transforma as árvores de consulta e seleciona métodos para executar as operações de acordo com as funções de custo associadas aos métodos. A estratégia de busca evita utilizar busca exaustiva e se automodifica, procurando tirar vantagem das experiências passadas. Uma visão geral da arquitetura do EXODUS pode ser encontrada em [23]. A técnica de corte para reduzir o número de planos considerados envolve regras, como utilizar sempre que possível o operador de junção e evitar produto cartesiano seguido por seleção, e envolve heurísticas, como executar as seleções antes das junções. O resultado mais importante desse projeto foi demonstrar a possibilidade de separar a estratégia de busca do modelo de dados, permitindo a implementação de otimizador e algoritmo de busca genéricos, compatíveis a vários modelos de dados. Isso foi possível porque as regras de transformação e implementação são independentes umas das outras. Em comparação com os resultados produzidos através de programação dinâmica com busca exaustiva, o EXODUS apresentou resultados semelhantes.

O projeto Volcano apresenta uma máquina de busca mais extensivel e poderosa do que a desenvolvida no projeto EXODUS, fornecendo suporte efetivo a modelos de custo não-triviais e a propriedades físicas, como ordem. Outras diferenças são (1) a utilização da programação dinâmica baseada nas propriedades físicas em uma busca direcionada a um objetivo, (2) a técnica de corte *branch-and-bound*, na qual após formado um plano de execução, nenhum outro plano de execução gerado e que tenha um custo maior é mantido e (3) a utilização de heurística. Esse novo algoritmo de busca foi denominado *direct dynamic programming*. O otimizador Volcano em comparação com o EXODUS demonstrou melhor desempenho.

O projeto Cascades foi desenvolvido para complementar a implementação do otimizador dos projetos Volcano e EXODUS, oferecendo algumas vantagens, sem abandonar a modularidade, extensibilidade, programação dinâmica e exploração da memorização presentes nos projetos anteriores. Os operadores passam a ser tanto lógicos quanto físicos, fazendo parte da consulta, que é a entrada do otimizador, ou da saída, que é o plano de

execução.

Em [30] é proposta uma alteração no otimizador do projeto Cascades, através do conceito de otimização de consultas baseada em regiões. Embora definir o plano de execução ótimo seja benéfico para a maioria das consultas, o tempo de otimização para determinados tipos de consultas cresce inaceitavelmente e por isso é importante limitar o número de alternativas de planos considerados pelo otimizador para deteminados tipos de consultas através do uso de heurísticas apropriadas, no qual diferentes tipos de consultas são otimizadas utilizando diferentes estratégias de busca em cada região. O otimizador de consultas baseado em regiões apresentou desempenho superior ao otimizador de consultas por buscas exaustivas, tanto nos tempos de otimização e de execução quanto na qualidade dos planos gerados.

Os *frameworks* Volcano e Cascade diferem do Starburst na técnica de enumeração de planos. Enquanto o Starburst possui duas fases distintas de otimização, todas as transformações no Volcano e no Cascades são algébricas e baseadas no custo, ocorrendo em um único passo.

1.3 Organização da Dissertação

Esta dissertação está organizada em capítulos da seguinte forma. O capítulo 2 descreve brevemente os métodos que podem ser aplicados na otimização de consultas para geração de planos de execução e determinação do plano de consulta ótimo. O capítulo 3 descreve em detalhes o processamento de consultas do SGBD PostgreSQL. O capítulo 4 descreve os métodos de otimização de cosultas estudados para serem implementados em um SGBD real, apresentando os motivos pelos quais alguns deles foram descartados neste trabalho, e apresentando o algoritmo de Kruskal como método escolhido para aplicação no PostgreSQL. O capítulo 5 descreve a implementação do algoritmo de Kruskal no otimizador do PostgreSQL, com os experimentos realizados e a avaliação deste método na otimização de consultas. O capítulo 6 apresenta as conclusões sobre o trabalho aqui descrito.

CAPÍTULO 2

MÉTODOS DE OTIMIZAÇÃO DE CONSULTA

O trabalho na definição de uma boa estratégia de avaliação para as expressões com junções pode ser dividido em desenvolvimento de algoritmos eficientes para executar as junções e desenvolvimento de algoritmos para determinar a ordem em que as junções devem ser efetuadas. Este capítulo apresenta uma descrição das diversas classes de algoritmos utilizados no processo de otimização de consultas para determinar a sequência de junções com o menor custo.

Em problemas que apresentam um número muito grande de soluções, como nos exponenciais ou fatoriais, é inviável criar ou examinar todas as soluções possíveis para definir qual é a melhor delas, pois o tempo de computação é em termos de anos até mesmo nos modernos computadores paralelos. Neste caso são usados algoritmos que encontram uma solução aceitável, mesmo sem a garantia de que essa seja a solução ótima. Estes algoritmos utilizam alguns critérios para definir qual o melhor caminho a seguir, como custo e ganho de cada opção.

2.1 Definição de Termos

O problema de otimização recebe como entrada um *grafo de consulta* (*query graph*), no qual os nodos são as relações da consulta e as arestas são as junções entre as relações. As arestas são nomeadas com o predicado e a seletividade da junção. O predicado da junção mapeia quais as tuplas do produto cartesiano entre as relações que devem ser incluídas no resultado. A seletividade da junção é a fração *número de tuplas no resultado/número de tuplas no produto cartesiano*. O produto cartesiano pode ser considerado como uma junção de seletividade 1.

O *espaço de busca* é o conjunto com todos os planos de execução que computam um mesmo resultado para uma consulta. Um *ponto* no espaço de busca é um plano

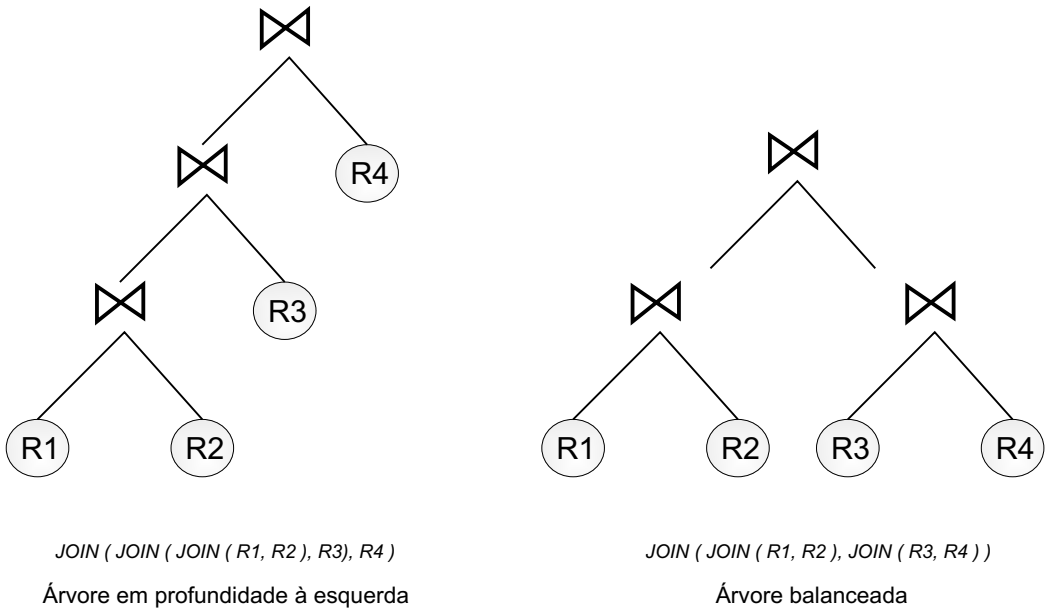


Figura 2.1: Possíveis topologias das árvores de execução de consultas.

particular para a solução de um problema. Cada ponto contém um custo associado e o objetivo da otimização de consultas é encontrar o ponto no espaço de busca com o menor custo possível. *Árvore de execução* é a estrutura que descreve o plano para a solução do problema. Essa árvore é binária, onde as relações básicas são os nodos folhas e os operadores de junção são os nodos intermediários. Cada relação básica aparece exatamente uma vez na árvore. As arestas denotam o fluxo de dados a partir das folhas até a raiz da árvore. Devido a comutatividade e associatividade da operação de junção, o número de pontos no espaço de busca cresce rapidamente com o aumento do número de relações envolvidas na consulta. As árvores de execução podem ter topologia em *profundidade à esquerda* (*left-deep tree*) ou *balanceada* (*bushy tree*), conforme ilustrado na figura 2.1.

Nas árvores em profundidade à esquerda, a relação *outer* pode ser tanto uma relação básica quanto o resultado de junções (*joinrel*) anteriores, enquanto a relação *inner* é sempre uma relação básica. Para uma consulta com n relações, existem $n!$ formas de ordenar as relações na árvore [12]. Essa classe de árvores é capaz de explorar a técnica de redução de custo por encadeamento (*pipeline*) de cada junção processada.

Nas árvores balanceadas, tanto os operandos da esquerda quanto os operandos da direita podem ser junções entre tabelas e não necessariamente relações básicas. Essa

classe de árvores é a mais abrangente, englobando as árvores de outras topologias, como as árvores em profundidade à esquerda e as árvores em profundidade à direita. Por esse motivo, a cardinalidade deste conjunto é muito maior do que a cardinalidade das árvores em profundidade à esquerda. Entretanto, apesar do alto grau de liberdade ao construir árvores balanceadas, a capacidade de explorar a técnica de encadeamento é restrita nas àrvores em profundidade à direita por não apresentarem a propriedade associativa. Nessas árvores o operando da esquerda é uma relação básica e o operando da direita é uma junção e as operações devem seguir a ordem exata de execução conforme encadeadas na árvore, para que o resultado desejado seja alcançado.

Em [24] foi proposta uma estratégia de enumeração de planos para grafos lineares e grafos estrela, para reduzir o número de planos apenas àqueles com custos que devem ser considerados. Um grafo linear consiste em diversos nós, sendo que cada nó deve estar conectado a um ou mais nós através de arestas. Um grafo bipartido é um grafo no qual os nós são divididos em dois subconjuntos e todos as arestas do grafo ligam um nó de um subconjunto a um nó do outro. Um grafo estrela é um grafo bipartido de n vértices que possui um conjunto independente com um único vértice e o outro com *n-1* vértices. Com isso, para um grafo linear com n relações existem no espaço de busca $(n^3 - n)/6$ árvores balanceadas ou $(n-1)^2$ árvores em profundidade à esquerda que devem ser consideradas. Para grafos estrela existem $(n-1).2^{n-2}$ árvores balanceadas a serem consideradas.

Existem diversos estudos explorando as árvores balanceadas mas a maioria dos sistemas de otimização tem preferência para sequências lineares de junções, obtidas nas árvores em profundidade à esquerda.

2.2 Algoritmos Determinísticos

Um algoritmo determinístico é aquele que contém uma sequência finita de passos voltados para a resolução de um certo problema. O tempo de execução de um algoritmo determinístico é fixo e sucessivas aplicações do algoritmo para uma mesma entrada de dados resultam em uma mesma saída com tempos de processamento sempre idênticos.

Os algoritmos desta classe constroem a solução passo a passo utilizando busca exaus-

tiva no espaço de busca ou aplicando heurísticas. A seguir serão apresentados diferentes algoritmos determinísticos.

2.2.1 Programação Dinâmica

Na programação dinâmica todas as possíveis árvores de execução para uma dada consulta são construídas e avaliadas, através de uma busca exaustiva nos pontos do espaço de busca. Para isso, a consulta é dividida em partes menores e a melhor solução para o problema como um todo é dada utilizando as partes menos custosas, que são conectadas por uma fórmula recorrente. Esta fórmula retroativa também aborta todas as soluções intermediárias que são candidatas mas que possivelmente não serão as melhores soluções. As informações sobre as escolhas que ainda podem ser a melhor solução são guardadas e podem ser utilizadas nos passos seguintes.

Os passos de construção da solução são:

Passo 1: Para cada relação básica da consulta, todas as possíveis formas de acessá-la são obtidas, tanto por índice quanto por busca sequencial, e planos de execução são gerados. Estimativas sobre os custos de cada plano são obtidas e o plano de menor custo é mantido.

Passo 2: Para cada par de relações são calculados os custos de todas as formas de realizar a junção entre elas. O acesso físico a cada relação básica é realizado através da operação de menor custo obtida no passo anterior. Os planos das junções que apresentam custo elevado são eliminados.

Passo N: Todos os possíveis planos para responder à consulta são gerados a partir dos planos obtidos nos passos anteriores. O plano mais barato é então escolhido como o plano final a ser executado.

Para consultas com poucas relações esse algoritmo garante o plano ótimo, enquanto para consultas com mais de 10 relações o número de soluções geradas é muito grande, tornando o tempo de execução do algoritmo inviável. Outra desvantagem deste algoritmo é o alto consumo de memória para armazenar as soluções parciais.

A essência da programação dinâmica considera que o modelo de custo satisfaz o

princípio da optimabilidade, ou seja, para obter um plano de execução ótimo a partir de uma dada consulta Q que apresenta k junções é suficiente considerar apenas os sub-planos ótimos que contemplam *(k-1)* junções e então extender estes planos adicionando uma junção. Todas as relações básicas a serem relacionadas são enumeradas e um algoritmo *bottom-up* é executado. No final do passo i, o algoritmo produz os planos ótimos para todas as subconsultas de tamanho i. Para obter um plano ótimo para a subconsulta que contém $i+1$ relações são consideradas todas as formas possíveis de contruir um plano através da extensão dos planos contruídos no passo i. Por exemplo, um plano ótimo para {`R1, R2, R3, R4`} é obtido após a comparação entre os custos dos planos ótimos gerados por `Join({R1,R2,R3},R4), Join({R1,R2,R4},R3), Join({R1,R3,R4},R2)` e `Join({R2,R3,R4},R1)`, onde é escolhido o plano de menor custo.

A primeira vez que a programação dinâmica foi proposta como uma estratégia de busca na otimização de consultas foi no System-R por Sellinger et al. [12]. Além do custo, a técnica de *interesting order* é utilizada no System-R para efetuar o corte dos planos de execução que não são candidatos a solução ótima. Uma particular ordenação de tuplas pode ser caracterizada pelo conjunto de atributos utilizados para ordenação da consulta. *Interesting order* são ordenações onde o conjunto de atributos de ordenação podem ser usados por mais de um operador no plano de execução, contribuindo na diminuição do custo de computação dos operadores físicos subsequentes. Considerando uma consulta com as relações {R1, R2, R3} e junções R1.a = R2.a = R3.a, e que existam duas possíveis operações para a junção entre as tabelas, uma através de *nested-loop* e outra utilizando *merge-sort*. Caso o custo da junção entre R1 e R2 através de *nested-loop* seja menor do que o custo da junção através de *merge-sort* e por este motivo a operação for realizada com *nested-loop*, não será aproveitada a ordenação do atributo a para a futura junção do conjunto {R1, R2} com a relação R3. Portanto, apesar de apresentar um custo menor entre duas relações, a junção entre as três relações terá custo total mais baixo se no primeiro nível de junção a operação mais custosa for utilizada, que é *merge-sort*. Um plano de execução só é considerado equivalente a outro se os dois representarem a mesma expressão e apresentarem os mesmos atributos de ordenação. Neste caso, os planos são

comparados e apenas o de menor custo é mantido como canditado a plano ótimo.

Em vários sistemas, a sequência das operações de junção é restrita para limitar o espaço de busca. No System-R, apenas sequências lineares de junções são consideradas e o produto cartesiano só é considerado após todas as junções com predicados terem sido realizados. A operação de junção é associativa e comutativa e portanto uma consulta pode ser representada por diversas expressões diferentes logicamente equivalentes, e cada uma delas pode ser implementada por um conjunto distinto de operadores. Encontrar os operadores que utilizam menos recursos de processamento, acesso a disco, acesso a memória e recursos de rede é de fundamental importância. A estimativa do custo precisa ser eficiente pois será muito utilizada no processo de otimização de consultas e é ela que direcionará o resultado da otimização.

No System-R, as estatísticas sobre os dados são coletadas e estimativas dos custos das operações são calculadas com os dados coletados. Em uma árvore de operadores (*operator tree*), o custo de cada operação é estimado e o custo total da árvore é uma combinação desses custos unitários. Algumas informações estatísticas importantes são (1) o número de tuplas em uma tabela, já que esse valor interfere no custo da busca de dados, no custo da junção e na quantidade de memória necessária para armazenar os dados, (2) o número de páginas físicas e (3) informações que possam determinar a seletividade dos predicados sobre os atributo.

2.2.2 Seletividade Mínima

Boas soluções na otimização de consultas são geralmente caracterizadas por operações com resultados intermediários de baixa cardinalidade. A heurística de seletividade mínima contrói uma árvore em profundidade à esquerda passo a passo enquanto tenta manter as relações intermediárias tão pequenas quanto possível. O fator de seletividade pode ser utilizado para alcançar esse objetivo.

Os métodos gulosos fazem parte dessa classe de algoritmos, buscando a construção da árvore geradora mínima. Nestes algoritmos, o ganho em cada etapa é máximo, sem verificar as consequências dessa escolha para os próximos passos ou para a solução final.

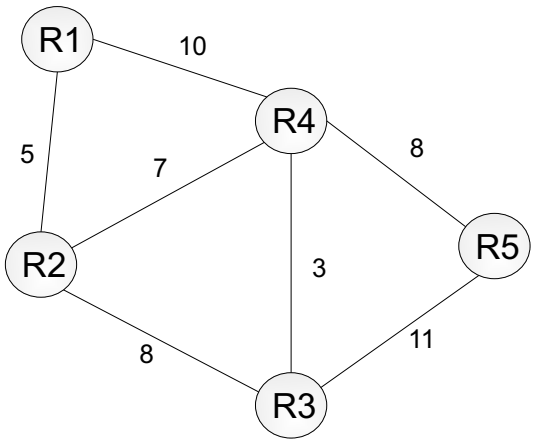


Figura 2.2: Grafo G representando uma consulta e simulação de execução do algoritmo guloso.

A cada etapa, vários candidatos são descartados e apenas o melhor candidato é escolhido. Devido a isso, um caminho com um ganho menor pode ser selecionado ao invés do melhor caminho.

Por exemplo, dado o grafo G da figura 2.2 e utilizando o algoritmo guloso para definir o caminho de menor custo entre o vértice inicial R1 e o vértice final R5, teremos o caminho gerado custando 26. Todavia, o caminho de menor custo é R1, R4, R5 que apresenta custo igual a 18.

Dois dos algoritmos gulosos clássicos são os algoritmos de Prim e Kruskal, que buscam uma árvore geradora mínima para um grafo conexo com pesos. Uma árvore T é denominada árvore geradora de um grafo conexo G se T é um sub-grafo de G que contém todos os vértices de G. Isto significa que os algoritmos gulosos encontram um subconjunto de arestas que formam uma árvore contendo todos os vértices e cujo peso total da árvore é minimizado.

2.2.2.1 Algoritmo de Prim

O algoritmo de Prim foi apresentado por Robert C. Prim em 1957. No algoritmo de Prim, deseja-se obter uma árvore geradora mínima dado um grafo G. Um vértice é escolhido aleatoriamente e definido como vértice inicial *v0*. Ele é incluído no conjunto T (Terminal). Os demais vértices de G são incluídos no conjunto P (Provisório). A cada iteração do algoritmo, um vértice de P é anexado à árvore, sendo removido de P e incluído em T. Esse vértice é escolhido baseado no custo para se chegar até ele a partir de qualquer um dos vértices que estão na árvore (em T). O algoritmo é encerrado quando o conjunto T contém todos os vértices de G e o conjunto P é vazio. Este algoritmo tem complexidade *O(E log V)*, onde E é o número de arestas no grafo e V é o número de vértices.

2.2.2.2 Algoritmo de Kruskal

O algoritmo de Kruskal foi apresentado por Joseph Kruskal em 1956 [21]. Dado um grafo G, deseja-se obter uma árvore geradora mínima. No início, cada vértice de G é considerado uma árvore de um único vértice. A cada iteração do algoritmo, duas árvores são conectadas formando uma árvore maior. A aresta escolhida é aquela que apresenta o menor custo e que não conecta dois vértices pertencentes à mesma árvore, pois resultaria em um ciclo. O algoritmo é encerrado após todas os vértices estarem presentes na árvore. Este algoritmo tem complexidade *O(E log V)*, onde E é o número de arestas no grafo e V é o número de vértices, assim como o algortimo de Prim.

2.2.3 Algoritmos Heurísticos

Algoritmos heurísticos são aqueles que encontram uma solução entre todas as possíveis mas não garantem que a solução encontrada é a solução ótima, já que são algoritmos de aproximação e não de precisão. Estes algoritmos geralmente encontram uma solução próxima da melhor, de uma forma fácil e rápida.

O algoritmo A* é um algoritmo amplamente utilizado na solução de problemas com buscas complexas na Inteligência Artificial. Esse algoritmo encontra a solução ótima se a

heurística utilizada for admissível.

Quando a grande quantidade de soluções e a pequena quantidade de tempo disponível impedem que sejam usados algoritmos de busca que explorem todas as soluções, é preciso encontrar uma estratégia que não percorra todos os vértices de soluções durante a busca, já que não há necessidade de que o caminho encontrado seja mínimo, sendo aceitável um caminho sub-ótimo. O algoritmo A* encontra o caminho de menor custo de um vértice a outro em um grafo examinando apenas os vizinhos mais promissores do vértice atual da busca.

O algoritmo mantém duas listas de vértices, Abertos e Fechados, para vértices a explorar e explorados, respectivamente. No início, Fechados está vazia e Abertos contém apenas o vértice de origem. A cada iteração, o melhor vértice é removido da lista de Abertos para ser explorado. Para cada vizinho do vértice explorado, são feitas as seguintes operações:

- Se está em Fechados, é ignorado;
- Se está em Abertos, suas informações são atualizadas quando o custo do caminho atual é menor do que o custo calculado anteriormente para ele;
- Se não está em nenhuma das duas listas, é um novo vértice e portanto é colocado na lista Abertos.

O vértice removido da lista Abertos é colocado na lista Fechados após todos os seus vizinhos serem processados. A cada iteração, o vértice da lista de Abertos com o menor custo estimado será escolhido pelo algoritmo. O critério para escolher o melhor vértice a ser examinado é essencialmente uma estimativa da distância entre cada vértice e o vértice de destino. O custo estimado do caminho que passa por um determinado vértice é dado pela seguinte equação:

```
f(v) = g(v) + h(v)
f(v): custo total do caminho do vértice de origem ao de destino passando
      pelo vértice v
g(v): custo calculado do caminho do vértice de origem ao vértice v
h(v): custo estimado do caminho do vértice v ao vértice de destino
```

Em particular, o componente *h(v)* da equação acima merece destaque. Ele representa uma estimativa heurística da distância entre um determinado vértice e o vértice destino. Essa estimativa pode obedecer a qualquer função que subestime o real valor da distância entre o vértice e o destino, chamada de função admissível. [29] prova que, se *h(v)* é uma sub-estimativa desse valor, então o algoritmo A* garante encontrar o caminho mínimo entre a origem e o destino. A complexidade teórica do algoritmo A* é $O(n^2)$ onde n é o número de vértices do grafo.

2.3 Algoritmos Aleatórios

Os algoritmos aleatórios também são conhecidos como algoritmos randômicos. As primeiras publicações nessa área datam dos anos 70 e tratam do problema de identificação de números primos. A partir de então, os pesquisadores têm utilizado em seus trabalhos cada vez mais técnicas que envolvem modelos probabilísticos.

Nesta classe de algoritmos, a busca pela solução ótima ocorre através de transformações dos planos de execução. As soluções são pontos no espaço de busca e a conexão entre os pontos é definida por um conjunto de transformações. As arestas que conectam duas soluções do espaço de busca representam a transformação ocorrida em uma das soluções e que origina a outra solução. As transformações ocorrem randomicamente, seguindo algumas regras pré-definidas, e o algoritmo termina quando não há mais transformações possíveis ou quando é atingido um tempo definido como limite. A melhor solução encontrada até o momento é então definida como solução final.

Segundo [26], os problemas pertencentes à Classe P, que são os problemas para os quais existem algoritmos que indicam a solução em tempo polinomial, podem ser tratados por métodos probabilísticos, evitando a complexidade teórica existente nos métodos determinísticos e beneficiando-se da flexibilidade provida pelos modelos probabilísticos. Os problemas da Classe NP também podem utilizar os métodos aleatórios em suas buscas por soluções aproximadas ou sub-ótimas. Neste caso, espera-se que o tempo seja polinomial e a solução encontrada esteja próxima o bastante da solução ideal.

Os algoritmos aleatórios, ao contrário dos algoritmos determinísticos, podem gerar

resultados diferentes para sucessivas execuções com a mesma entrada de dados, e em tempos diferentes de processamento. Isso porque as funções que compõem esse tipo de algoritmo são baseadas em sementes randômicas aplicadas à entrada de dados, e escolhas são realizadas aleatoriamente. Para análise de tempo de processamento é necessário haver uma grande compreensão da distribuição de probabilidade associada.

CAPÍTULO 3

POSTGRESQL

Para estudar um otimizador de consultas na prática, procurou-se entre os SGBDs disponíveis no mercado aqueles que têm seu código fonte disponível e aberto, o que possibilita uma avaliação detalhada dos algoritmos utilizados e também permite alterações e/ou inclusões de processos.

Entre os SGBDs encontrados com estas características, os mais utilizados são o PostgreSQL e o MySQL. O PostgreSQL foi escolhido por possuir mais recursos que o MySQL, possibilitando avaliação do comportamento da otimização para consultas complexas.

Outro motivo para esta escolha foi uma falha na reescrita de consultas do PostgreSQL detectada em [4]. O otimizador não simplificou a cláusula *where* que continha diversas operações *OR* duplicadas e por este motivo a consulta não foi executada, apresentando erro de tempo limite esgotado (*timeout*). Após a reescrita manual da consulta, o resultado da mesma foi retornado em um tempo tolerável. Este erro foi prontamente corrigido em dezembro de 2003 pela comunidade de desenvolvedores PostgreSQL [22] mas indicou que o otimizador ainda não é totalmente eficaz, demonstrando uma certa fragilidade na reescrita de consultas e gerando insegurança com a possível existência de outras falhas.

O PostgreSQL é um sistema gerenciador de banco de dados objeto-relacional, desenvolvido na University of Califórnia, Berkeley Computer Science Department, com versão estável desde 1996. Tem mais de 15 anos de desenvolvimento ativo e uma arquitetura com forte reputação devido à sua confiabilidade, integridade de dados e exatidão. É bastante portável podendo ser executado nos principais sistemas operacionais, incluindo Linux, UNIX e Windows. Possui as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade) e vários recursos comuns aos tradicionais SGBDs comerciais, como suporte a consultas complexas, gatilhos (*triggers*), chaves estrangeiras (*foreign keys*), visões (*views*), controle de concorrência e integridade relacional. É atualmente o banco de dados

de código aberto disponível no mercado que contém o conjunto mais completo de recursos.

3.1 Processamento de Consultas no PostgreSQL

O processamento de consultas no PostgreSQL, desde o fornecimento da consulta por uma aplicação ou por um usuário até que seu resultado seja obtido, é composto pelas etapas da figura 3.1 descritas abaixo, segundo [31, 25].

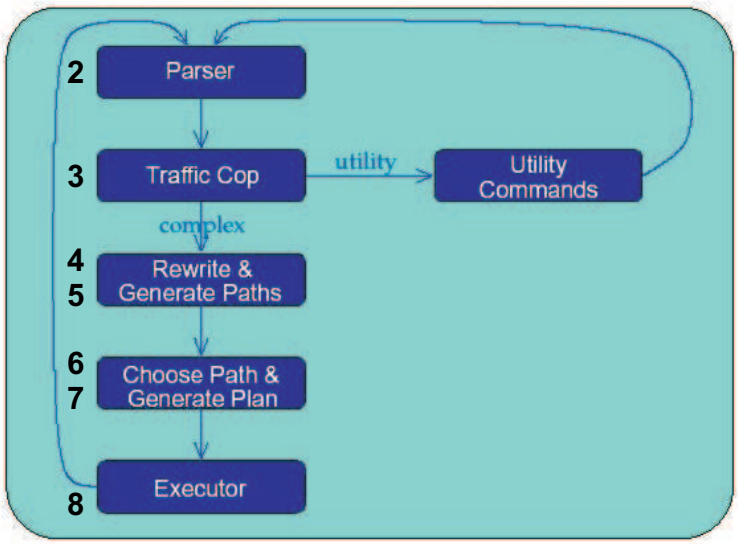


Figura 3.1: Módulos do processamento de consultas do PostgreSQL.

1. É estabelecida uma conexão da aplicação com o PostgreSQL. A aplicação transmite a consulta e aguarda o seu resultado.

2. O `Parser` verifica a sintaxe da consulta, avaliando se a estrutura SQL está correta. Recebe como entrada uma *string* SQL, divide-a em *tokens* e gera como saída a estrutura `parse tree`.

3. O `Traffic Cop` recebe como entrada a `parse tree` gerada na etapa anterior e se a consulta for um comando utilitário (*CREATE, ALTER, DROP*), executa o comando. Senão passa para a próxima etapa.

4. O `Rewrite` realiza a análise semântica na `parse tree`, verificando se as tabelas, atributos e funções utilizadas na consulta realmente existem no banco de dados e se

estão relacionados. Realiza também expansão de funções, simplificação de funções constantes, transformação de subconsultas e visões (*views*) em tabelas na cláusula *from* e predicados na cláusula *where*, dedução de igualdades implícitas, conforme [17, 33]. Gera como saída uma `query tree`, que é uma estrutura com todos os elementos usados por uma consulta complexa, na qual as partes do código SQL são armazenadas separadamente (cláusula *from* na estrutura `Rangetable`, cláusula *where* na estrutura `Qualification`, etc).

5. O `Generate Paths` gera todos os caminhos possíveis que levam ao resultado, através de uma busca quase exaustiva. O objetivo é gerar o plano ótimo. A estrutura de dados utilizada é a `path tree`, semelhante a `query tree`, mas contendo apenas informações essenciais para a geração dos planos. Esta etapa será detalhada na seção 3.2.

6. O `Choose Path` estima o custo de execução de cada `path`, utilizando como métrica a quantidade de páginas acessadas em disco. São utilizados nos cálculos dados estatísticos armazenados em tabelas do sistema (`pg_class e pg_statistics`), como quantidade de registros e de blocos ocupados em disco pelas tabela e seus índices, seletividade das tabelas, fração nula, quantidade estimada de valores distintos, lista com valores mais comuns e frequência dos valores mais comuns. O plano mais barato é escolhido.

7. O `Generate Plan` gera detalhes do plano escolhido na etapa anterior e armazena esses dados na estrutura `plan tree`, com todas as informações necessárias para a consulta ser executada.

8. O `Executor` realiza os passos indicados na `plan tree` para recuperar as tuplas desejadas e envia o resultado para a aplicação.

3.2 Otimizador de consultas no PostgreSQL

O otimizador do PostgreSQL é composto por dois algoritmos distintos para geração dos possíveis planos de execução. Um deles é o algoritmo de programação dinâmica, utilizado na otimização das consultas com até 11 tabelas. Este algoritmo utiliza a técnica de buscas exaustivas com um método de corte para determinar a solução ótima. O outro algoritmo é chamado GEQO e é um algoritmo genético, utilizado para otimização das consultas com 12 ou mais tabelas. Este algoritmo utiliza valores aleatórios como *fitness* para geração dos planos de execução. Os dois algoritmos serão apresentados em detalhes a seguir.

3.2.1 Programação Dinâmica

O algoritmo principal utiliza programação dinâmica, através de busca quase exaustiva sobre o espaço das possíveis estratégias, conforme descrito em [11]. Este algoritmo foi baseado no System-R e é bastante semelhante ao algoritmo descrito em [34]. Este método é utilizado para consultas com até n junções, onde n é parametrizável e seu valor padrão é 11.

Para cada relação individual (tabela), são gerados todos os planos possíveis para se fazer uma busca (*scan*) nessa tabela. Sempre é gerado o plano de execução para a busca seqüencial (*sequential scan*), e para cada índice encontrado na tabela, é gerado um plano adicional de busca através desse índice (*index scan*). Se a consulta tem apenas uma relação, esses planos são comparados e o de menor custo é escolhido.

Para consultas com mais de uma relação, o próximo passo é gerar planos para a junção de duas tabelas. São criados planos para todas as junções possíveis entre 2 tabelas, utilizando os três métodos de junção disponíveis no PostgreSQL descritos a seguir.

- *Nested-loop join*, onde para cada linha encontrada na relação da esquerda, uma busca completa é realizada na relação da direita relacionando os dados entre as duas tabelas.

- *Merge sort join*, onde as duas relações são ordenadas pelos atributos da junção e então buscas são iniciadas em paralelo nas duas relações.

- *Hash join*, na qual a relação da direita é carregada em uma tabela *hash*, usando o atributo da junção como chave do *hash*. Depois, a busca é realizada na relação da esquerda e os valores apropriados são usados como chave para localizar as linhas na tabela *hash*.

Caso a consulta contenha mais de duas relações, o próximo passo consiste em realizar a junção entre uma das tabelas que ainda não está no plano e o resultado de uma junção já incluída no plano. Este passo é repetido até que todas as relações que fazem parte da consulta sejam incluídas no plano de execução. Por exemplo, para as duas consultas abaixo, possíveis junções das tabelas 1, 2, 3 e 4 a cada passo do algoritmo são mostradas.

```
SELECT   *
FROM     tab1, tab2, tab3, tab4
WHERE    tab1.col = tab2.col AND
             tab2.col = tab3.col AND
             tab3.col = tab4.col

{1 2},{2 3},{3 4}
{1 2 3},{2 3 4}
{1 2 3 4}
(outras possibilidades foram ignoradas pela falta de cláusula de junção)
```

```
SELECT   *
FROM     tab1, tab2, tab3, tab4
WHERE    tab1.col = tab2.col AND
             tab1.col = tab3.col AND
             tab1.col = tab4.col

{1 2},{1 3},{1 4}
{1 2 3},{1 3 4},{1 2 4}
{1 2 3 4}
```

O resultado final é construído em uma árvore binária de operações que indica a ordem em que as operações físicas devem ser realizadas. Seqüências de execução diferentes são analisadas e a que possui o menor custo é escolhida. A figura 3.2 encontrada em [8] é um exemplo de árvore binária representando um plano de execução para a consulta SQL abaixo.

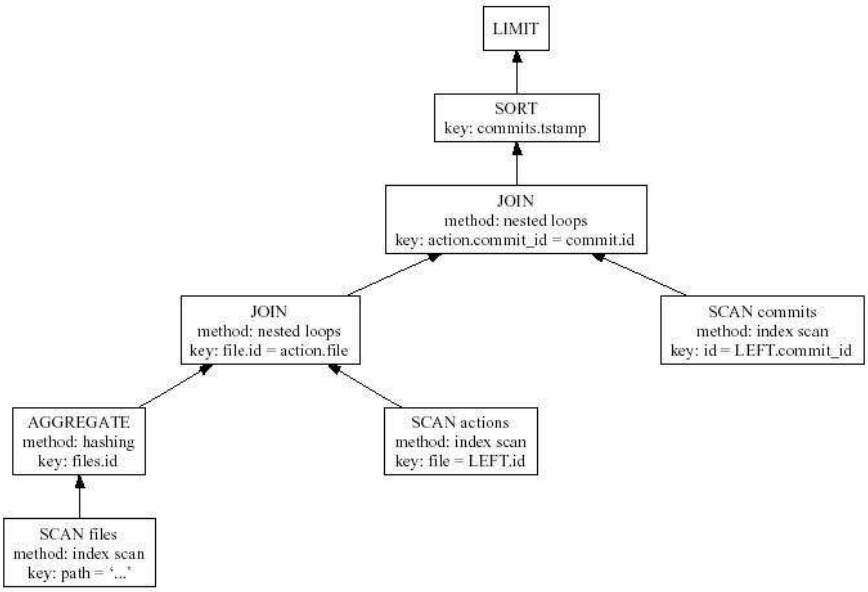


Figura 3.2: Possível plano de execução p a consulta SQL .

```
SELECT   c.tstamp
FROM     commits c, actions a
WHERE    a.file IN (SELECT id FROM files WHERE path = .....)
       AND a.commit_id = c.id
ORDER BY c.tstamp DESC
LIMIT 1;
```

3.2.2 Algoritmo Genético

Para consultas com um número grande de relações, o algoritmo que realiza a busca quase exaustiva sobre o espaço das possíveis estratégias pode levar um tempo enorme de processamento e ocupar um grande espaço em memória. Essa dificuldade em explorar o espaço dos possíveis planos da consulta gerou a necessidade de uma outra técnica de otimização, e a técnica escolhida para implementação do novo módulo no PostgreSQL utiliza algoritmo genético.

O módulo do algoritmo genético, chamado *Genetic Query Optimization* (GEQO), foi implementado por Martin Utesch em 1997 [37]. Este módulo permite que consultas com muitas relações sejam suportadas pelo otimizador através de um método de buscas não-exaustivas e é automaticamente acionado pelo otimizador do PostgreSQL no processamento de consultas com 12 ou mais relações.

O GEQO opera sobre buscas determinadas e aleatórias. O conjunto de possíveis planos para a execução de uma consulta é definido como população de indivíduos. Através das operações de recombinação, mutação e seleção natural, gerações de indivíduos que apresentam melhor valor de adaptação (*fitness*) que os ancestrais são formadas.

Por ser um método aleatório e utilizar como *fitness* uma variante que não atende com eficiência o problema do planejamento de junções, muitas vezes são gerados e mantidos planos impraticáveis e por isso seu desempenho não é considerado suficiente.

3.2.3 Considerações Finais sobre os Métodos

Como visto neste capítulo, os dois métodos de otimização presentes no PostgreSQL são o algoritmo de programação dinâmica e o algoritmo genético. Ambos apresentam desempenho razoável, mas não totalmente satisfatórios para todos os tipos de consultas, como demonstrado pelo Eduardo em [4].

No próximo capítulo será proposta uma abordagem distinta para a otimização de consultas com o intuito de avaliar a utilização de outros métodos no processamento de consultas utilizando o PostgreSQL.

CAPÍTULO 4

ESTUDO DE CASO

Com o intuito de avaliar o funcionamento de outros algoritmos no processo de otimização de consultas e utilizando o SGBD PostgreSQL na implementação, alguns métodos foram analisados. Ao utilizar novos algoritmos espera-se suprir as deficiências encontradas no otimizador de consultas desse SGBD.

Este capítulo descreve as pesquisas e os trabalhos realizados sobre o processo de otimização de consultas durante este estudo, desde as tentativas de utilizar a linguagem PDDL e o algoritmo A* (A-Estrela), apresentando as razões pelas quais esses dois metódos não foram utilizados, até a escolha do algoritmo de Kruskal para ser implementado no otimizador de consultas do PostgreSQL.

4.1 Planejamento automático e PDDL

O primeiro método avaliado é baseado na utilização de técnicas de planejamento automático [38].

Um planejamento automático parte da simulação do comportamento de um agente em um determinado ambiente e enumera as ações adotadas por esse agente para alcançar um conjunto de objetivos previamente definidos. Se o agente conseguir alcançar os seus objetivos, a enumeração de suas ações é chamada de *plano para alcançar esses objetivos*. Caso contrário, dizemos que não foi possível gerar um plano para o problema.

A linguagem de descrição de domínios em planejamento automático denominada *Planning Domain Definition Language* (PDDL) foi criada por Drew McDermott em 1998 [27], para a primeira versão da competição mundial de planejamento automático - *AI Planning Systems 98* [28]. O objetivo dessa linguagem é estabelecer uma notação padrão para a descrição de domínios usados como referência (*benchmark*) na análise do desempenho dos planejadores.

À primeira vista, o processo de otimização de consultas parece se encaixar nesse modelo, em que os *agentes* seriam os objetos (tabelas, atributos, índices) relacionados à consulta, as *ações* seriam as operações físicas realizadas sobre os objetos (junção, projeção, seleção, busca), e o *objetivo* seria o resultado final da consulta. O plano de execução seria determinado pela seqüência de ações executadas para chegar ao objetivo.

Entretanto, na tentativa de descrever o processo de otimização de consultas usando PDDL, um impasse foi encontrado na forma de descrever os efeitos das ações, já que para cada consulta o objeto resultante é diferente. Uma amostra de uma possível descrição da otimização de consultas em PDDL é mostrada na tabela 4.1.

Ação	Definição
`(:predicates)`	(tabela ?t) (atributo ?a) (indice ?i) (compoe ?a ?t) (indiceprimario ?a ?t) (indicesecundario ?a ?t) (filtro ?f) (estanofiltro ?a ?f) (resultado ?t ?f) (resultado ?t1 ?t2 ?f)
`(:action)`	ScanSequencial
`(:param)`	(?t ?a ?f)
`(:precondition)`	(and (tabela ?t) (atributo ?a) (compoe ?a ?t) (filtro ?f) (estanofiltro ?a ?f))
`(:effect)`	?????
`(:action)`	ScanIndicePrimario
`(:param)`	(?t ?a ?f)
`(:precondition)`	(and (tabela ?t) (atributo ?a) (compoe ?a ?t)
`(:precondition)`	(indiceprimario ?a ?t) (filtro ?f) (estanofiltro ?a ?f))
`(:effect)`	?????

Tabela 4.1: Uma descrição em PDDL para otimização de consultas.

Houve também dificuldade em descrever os objetivos conforme mostrado na tabela 4.2

Ação	Definição
`Exemplo`	SELECT emp.nome FROM empregado WHERE emp.matricula = 561
`Objects`	empregado emp.nome emp.matricula emp.matricula=561
`Init`	(tabela empregado) (atributo emp.nome) (atributo emp.matricula) (filtro emp.matricula=561) (compoe emp.nome empregado) (compoe emp.matricula empregado) (indiceprimario emp.matricula empregado) (estanofiltro emp.matricula emp.matricula=561)
`Goal`	?????
`Execução desejada:`	ScanIndicePrimario(empregado emp.matricula emp.matricula=561)

Tabela 4.2: Um exemplo de aplicação de uma consulta na descrição PDDL.

Este impasse aconteceu porque PDDL trabalha com as propriedades dos objetos, ou seja, o que muda durante a busca pela solução não são os objetos mas sim suas propriedades e por isso todos os objetos que fazem parte do problema devem ser declarados no início do mesmo, assim como as suas propriedades. Como na otimização de consultas não sabemos a priori todos os objetos intermediários que podem ser gerados durante a ação da máquina, não conseguimos definir o *efeito* nem o *objetivo*.

Devido à incerteza da possibilidade de transcrição do processo de geração de planos de execução de consultas em PDDL, e ainda por existirem outros métodos mais interessantes para utilizar na otimização de consultas do *PostgreSQL*, a opção foi por não utilizar o PDDL nesse trabalho e direcionar a pesquisa para classes de algoritmos diferentes das já utilizadas no PostgreSQL.

Apesar disto, como esta foi apenas a primeira tentativa em mapear a otimização de consultas em PDDL e existem diversas máquinas de buscas eficientes já desenvolvidas para PDDL, a dificuldade encontrada não deve desencorajar novas tentativas em futuras pesquisas pois se uma forma de modelar otimização de consultas em PDDL for encontrada, o plano ótimo pode ser obtido em um tempo bastante pequeno.

4.2 Algoritmo A* (A-estrela)

O próximo método avaliado está na classe dos algoritmos heurísticos e que baseiam-se no custo das operações para encontrar o plano de execução ótimo.

Os métodos heurísticos utilizam regras pré-definidas e funções heurísticas para determinar a ordem em que as operações físicas referentes a uma consulta devem ser executadas. Os métodos baseados em custo utilizam dados estatísticos coletados sobre a base de dados para estimar o custo total dos planos e então escolher o mais barato para ser executado.

No SGBD Oracle, um dos maiores e mais utilizados SGBDs comerciais, o otimizador baseado em regras (RBO) foi utilizado como método principal durante muito tempo, mas desde a versão 7 o otimizador baseado em custo (CBO) ganhou projeção e na última

versão do Oracle (10g) o RBO foi descontinuado. Isto indica que um modelo baseado no custo das operações é mais eficiente que um modelo com regras pré-definidas, pois algumas regras que são ótimas para um conjunto de dados de certo tamanho podem ser totalmente ineficientes para um conjunto de tamanho diferente [9].

A partir disso, buscou-se para o otimizador um método que contenha heurística com um conjunto de regras para fazer transformações em árvores de execução, e que se baseie nos dados estatísticos coletados do modelo de dados para calcular o custo dessas árvores transformadas.

Foi escolhido o algoritmo A* como algoritmo heurístico, um algoritmo amplamente utilizado na solução de problemas com buscas complexas na Inteligência Artificial. Uma das maiores aplicações do algoritmo A* é na busca do caminho mínimo entre dois pontos, e por isso é muito utilizado em algoritmos de jogos.

Para utilizar o algoritmo A* na otimização de consultas, decidiu-se representar as consultas SQL no formato de grafos nos quais os vértices representam as tabelas referenciadas na consulta e as arestas entre os vértices representam as cláusulas de junção (*where*) da consulta. Esse grafo é valorado e o valor das arestas é o cálculo do custo estimado para a junção que a mesma representa.

Com a consulta estruturada no grafo descrito anteriormente, esperava-se que ao aplicar do algoritmo A* fosse encontrado um caminho de custo mínimo envolvendo todos os vértices do grafo. Porém foi necessário informar como parâmetros do algoritmo os vértices origem e destino para então ser traçado um caminho de custo mínimo entre esses vértices. Possivelmente esse caminho não contém todos os vértices do grafo, portanto a aplicação desse método no processo de otimização de consultas é impróprio, uma vez que o plano ótimo de execução de uma consulta deve conter todas as tabelas da consulta.

Na aplicação do algoritmo A* no grafo da figura 4.1 que representa uma consulta SQL, ao indicar como vértice origem a relação R2 e como destino a relação R5, o resultado será o caminho R2-R4-R5, que exclui as relações R1 e R3. Não incluir tabelas no plano de execução final de uma consulta significa um resultado incompleto, pois dados necessário não estariam inclusos no resultado.

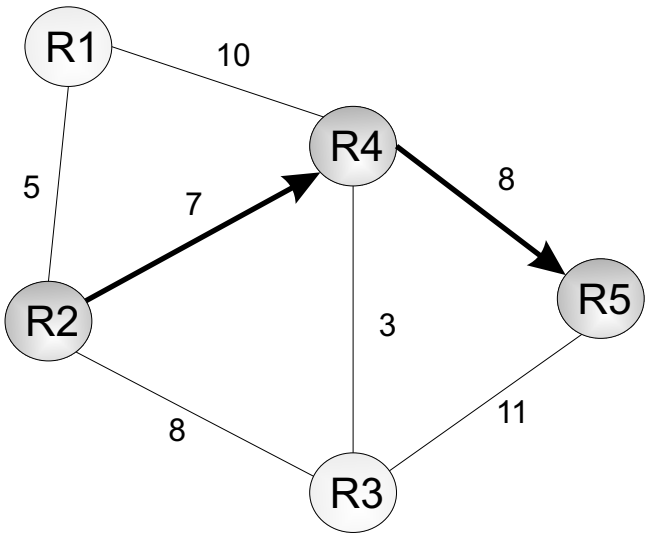


Figura 4.1: Caminho ótimo após aplicação do A* com origem em R2 e destino R5.

Seria possível adaptar a utilização do algoritmo no processo de otimização de consultas com a criação de um vértice destino fictício conforme figura 4.2, e então executando o A* diversas vezes, sendo que a cada iteração uma tabela diferente da consulta seria o vértice inicial, até que todas as tabelas fossem indicadas como vértice origem. Para isso seria necessário também criar um mecanismo para impedir que um caminho fosse gerado sem passar por todos os vértices do grafo. Uma possibilidade seria atribuir um custo extremamente alto às arestas que ligam os vértices ao vértice destino fictício até que todos os vértices tenham sido incluídos à solução final. Após geração de planos de execução partindo de cada um dos vértices do grafo, seria escolhido o plano de menor custo entre todas as iterações realizadas.

Outro fator interessante ao utilizar o algoritmo A* é que a definição do caminho sub-ótimo é dada em um tempo razoável, já que esse algoritmo inspeciona apenas os vizinhos mais promissores do vértice corrente da busca. Mas com as modificações citadas anteriormente, o tempo de processamento na busca por um plano de execução de consultas ótimo tornaria-se muito grande, já que a complexidade é multiplicada pelo número de relações existentes na consulta.

Por conter muitas adaptações que podem engessar o funcionamento e desempenho do algoritmo, essa implementação não pareceu interessante e foi descartada.

Partindo para a representação da consulta de uma forma diferente da descrita anteri-

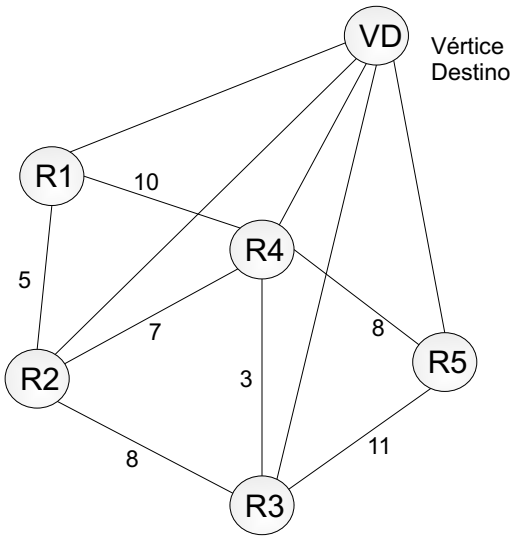


Figura 4.2: Adaptação da representação da consulta com vértice destino fictício, para execução do algoritmo A*.

ormente, outra implementação do algoritmo A* foi considerada. Desta vez as consultas seriam grafos nos quais os vértices representariam as árvores de execução da consulta e as arestas representariam as transformações realizadas entre as árvores, conforme figura 4.3.

Nessa implementação o algoritmo A* funcionaria agregado a um algoritmo aleatório que geraria as transformações nas árvores de execução. A seguir a descrição da implementação:

1. Gera qualquer uma das possíveis árvores de execução para esta consulta e define-a como árvore base;

2. Faz transformações na árvore base, a partir de funções heurísticas e regras pré-definidas, resultando em novas árvores alternativas e equivalentes a primeira;

3. Avalia o custo das árvores geradas utilizando os dados estatísticos armazenados no banco;

4. Mantém as árvores que têm um custo menor do que o custo da árvore base;

5. Usa como base para nova aplicação dos passos 2 a 5 as árvores que foram mantidas no passo anterior.

Essa implementação foi descartada pela sua complexidade de desenvolvimento e testes. Além disso, em um único passo do algoritmo podem ser realizadas todas as transformações

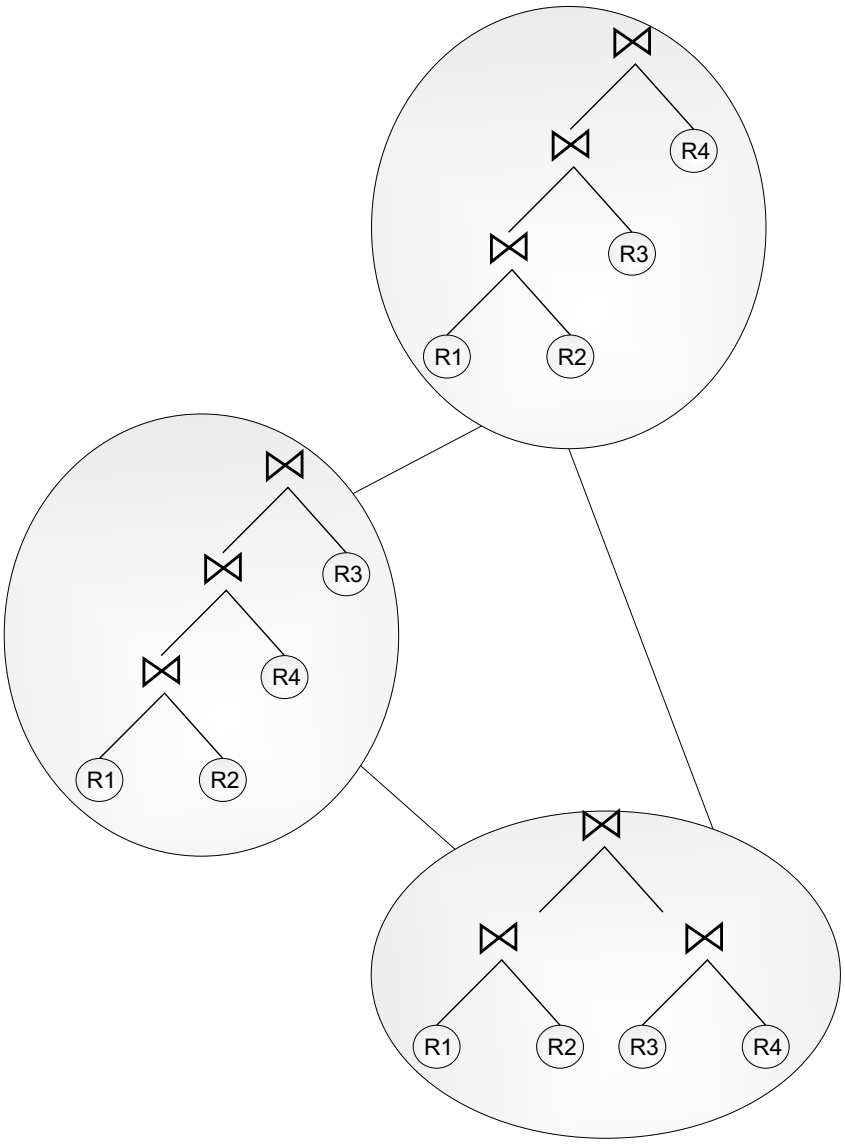


Figura 4.3: Representação da consulta na qual os vértices são árvores de execução e as arestas são transformações entre árvores.

viáveis em uma árvore de execução e geradas todas as árvores correspondentes à essas transformações, o que tornaria o espaço de busca igual ao do algoritmo de busca exaustiva.

4.3 Algoritmo de Kruskal

Após as limitações dos métodos estudados anteriormente no processo de otimização de consultas, buscou-se algoritmos utilizados na Teoria do Grafos para serem aplicados no otimizador de consultas, já que as etapas de execução de uma consulta podem ser facilmente representadas através de grafos e árvores.

Os algoritmos gulosos são utilizado na Teoria dos Grafos para encontrar a *árvore geradora mínima* de um grafo, que é um subconjunto de arestas formando uma árvore que contém todos os vértices do grafo e cujo peso total das arestas é minimizado. Isto torna a classe de algoritmos gulosos interessante para utilização na otimização de consultas, pois um plano de execução de consultas ótimo pode ser descrito como uma árvore geradora mínima do grafo que representa a consulta.

Dois dos algoritmos que pertencem à classe dos algoritmos gulosos são o algoritmo de Prim e o algoritmo de Kruskal. Esses algoritmos foram descritos no capitulo 2. A implementação desses métodos é simples e eles apresentam complexidade computacional baixa, principalmente se comparados com os algoritmos de busca exaustiva, o que torna interessante a realização de testes com essa classe de algoritmos. Por outro lado, apesar da construção da solução ótima ocorrer em um curto espaço de tempo, como esses algoritmos consideram a melhor operação de cada etapa e não o melhor conjunto de operações, a solução final pode ficar comprometida exibindo um resultado indesejável.

Entre os dois algoritmos, o algoritmo de Kruskal parece apresentar a forma mais adequada de construir a solução final no processo de otimização de consultas pois executar primeiro as junções de menor custo em uma consulta tende a reduzir o custo das junções seguintes que envolvem as tabelas maiores e mais custosas. Além disso, o algoritmo de Kruskal foi citado em outros trabalhos que envolvem otimização de consultas. [5, 10, 3, 20]

tabela	♯ registros
Empresa	2
Departamento	10
Funcionário	250
Salário	370
Cargo	30
Categoria	5

Tabela 4.3: Tabelas e número de registros do modelo de dados Funcionários.

Como exemplo, consideramos o modelo de dados da tabela 4.3. Para selecionar os funcionários de determinada empresa que possuem salário acima de R$ 5000,00 e que pertencem a determinada categoria de trabalhadores, teríamos a seguinte consulta:

```
SELECT  f.nome, s.salario
FROM    funcionario f,
        empresa e,
        departamento d,
        salario s,
        cargo c,
        categoria t
WHERE   e.empresa = 'empresa ltda'
        AND e.empresa = d.dep_empresa
        AND d.departamento = f.fun_departamento
        AND f.funcionario = s.sal_funcionario
        AND s.sal_funcionario > '5000'
        AND c.cargo = f.fun_cargo
        AND t.categoria = c. car_categoria
        AND t.categoria = 'engenharia'
```

Na figura 4.4 é mostrado o grafo que representa a consulta descrita anteriormente, considerando os custos das junções entre as tabelas como o número de registros resultante de cada junção. As junções entre tabelas que não possuem cláusula restritiva têm como resultado o produto entre o número de tuplas das relações. As tabelas `Empresa`, `Salario` e `Categoria`, que contêm cláusulas restritivas, possuem um fator de seletividade diferente de 1 e o resultado das junções em que estão envolvidas é o produto entre o número de tuplas das relações multiplicado pelo fator de seletividade.

Aplicando o algoritmo de Kruskal nesse caso, as primeiras operações executadas são `Join(Empresa, Departamento)` e `Join(Cargo, Categoria)`. Como o conjunto de dados resultante dessas operações é menor do que o número de registros existente nas tabelas

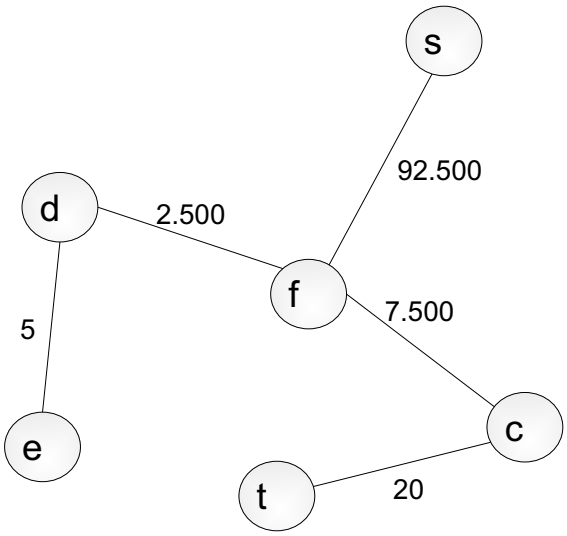


Figura 4.4: Grafo representando a consulta ao modelo Funcionários, com os custos das junções.

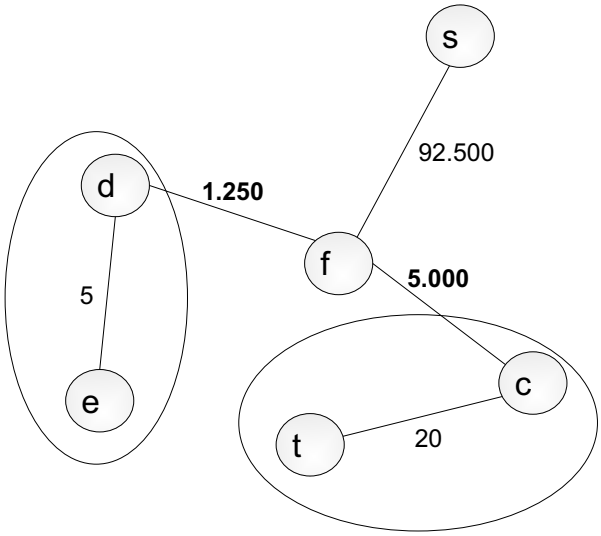


Figura 4.5: Aplicação do algoritmo de Kruskal no grafo da figura 4.4.

Departamento e *Cargo*, as junções *Join(Join(Empresa, Departamento), Funcionário)* e *Join(Join(Cargo, Categoria), Funcionário)* terão seus custos reduzidos, conforme ilustrado na figura 4.5, gerando como resultado final um plano de execução de custo minimizado.

Por esses motivos, para geração do plano ótimo de execução de consultas foi escolhido implementar o algoritmo de Kruskal no otimizador de consultas.

O algoritmo inicia calculando o custo da junção entre duas relações quaisquer pertencentes à consulta e segue até que todas as junções de apenas duas relações tenham seus

custos calculados. Em seguida, é escolhida a junção que possui o menor custo e a aresta *(i,j)* que a representa é incluída na árvore de execução da consulta. Caso exista mais de uma junção com o mesmo custo, uma delas é escolhida aleatoriamente.

No passo seguinte, a junção com o segundo menor custo é escolhida e avaliada se pode ser incluída ou não na árvore de execução da consulta. A avaliação garante que não será incluída na árvore de execução uma junção entre tabelas que forme um ciclo na representação da execução da consulta. Portanto, se a inclusão da aresta na árvore formar um ciclo, a junção é descartada e não é incluída na árvore.

O algoritmo segue buscando a próxima junção de menor custo que ainda não foi analisada para que a avaliação seja feita e ela possa ser incluída na árvore ou então descartada, até que todas as relações presentes na consulta estejam na árvore de execução da consulta.

No próximo capítulo será descrito como foi realizada a implementação do algoritmo de Kruskal no otimizador de consultas do PostgreSQL. Também serão apresentados os experimentos realizados e os resultados obtidos.

CAPÍTULO 5

IMPLEMENTAÇÃO DO ALGORITMO DE KRUSKAL NO POSTGRESQL

Após estudo do processo de otimização de consultas do PostgreSQL e de outros métodos que podem ser utilizados na geração do plano ótimo de execução de consultas, foi escolhido o algoritmo de Kruskal para implementação no otimizador do PostgreSQL.

Este capítulo descreve o processo de implementação, assim como os experimentos realizados e os resultados obtidos.

5.1 Codificação

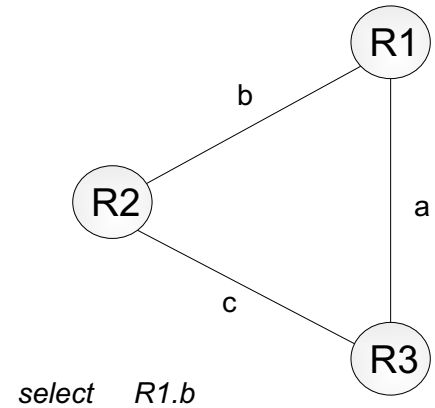
A linguagem escolhida para implementação do algoritmo foi o C++, por uma questão de compatibilidade com os módulos de processamento de consultas do PostgreSQL, que são desenvolvidos em C/C++. Outro fator que influenciou na escolha da linguagem foi a grande quantidade de implementações de algoritmos em C e C++ disponíveis em livros, revistas, projetos e na Internet, inclusive do algoritmo de Kruskal, o que facilita a construção da solução.

A definição da estrutura de dados para representar as consultas foi o primeiro passo na implementação do algoritmo de Kruskal no otimizador de consultas do PostgreSQL.

Uma consulta Q com predicados q é representada pelo grafo de consulta *G(V,E)* no qual

```
V = conjunto de todas as relações referenciadas em Q
E = (i,j), onde i e j são elementos diferentes de V e existe predicado q que faz
    referência tanto à relação i quanto à relação j
```

Um exemplo de consulta SQL e o seu respectivo grafo de consulta são mostrados na figura 5.1. O grafo de consultas é valorado, sendo que cada aresta *(i,j)* contém o




```
select   R1.b
from     R1, R2, R3
where    R1.a = R3.a
         and R1.b = R2.b
         and R2.c = R3.c
```

Figura 5.1: Exemplo de consulta SQL e sua representação em grafo.

custo da junção entre as relações i e j . O custo é calculado baseado em informações sobre as relações, como número de tuplas e seu tamanho em *bytes*, e informações sobre os atributos referenciados na consulta, como cardinalidade, seletividade e tamanho em *bytes*. Na implementação do algoritmo de Kruskal foi utilizado o módulo do PostgreSQL que coleta essas informações e realiza o cálculo da estimativa de custo para determinar o custo das operações de junção entre as tabelas.

No otimizador Kruskal foi possível utilizar as mesmas estruturas de dados utilizadas pelo PostgreSQL. O otimizador constrói uma estrutura chamada `RelOptInfo` para cada tabela referenciada na consulta, que são as relações básicas de uma consulta. A estrutura `RelOptInfo` também é utilizada para armazenar as junções consideradas durante o processo de planejamento de execução de consultas. A `RelOptInfo` contém informações importantes para os processos de planejamento e otimização, como os identificadores das relações básicas, número de linhas e quantidade de bytes estimados, lista dos possíveis índices a serem utilizados e custo estimado das operações.

Existe apenas uma `RelOptInfo` para cada conjunto de relações, independente da ordem na qual as junções entre as tabelas ocorrerão. Por exemplo, a junção de três tabelas *R1, R2, R3* pode ser construída primeiro com a junção de *R1* e *R2* e depois com a junção desse conjunto com a relação *R3*, ou então a junção de *R2* e *R3* pode ser realizada antes e na sequência realizar a junção com *R1*. As diferentes formas de se construir a junção

entre relações são representadas por *caminhos*, armazenados na estrutura *Path*, e para cada `RelOptInfo` uma lista de caminhos é construída, indicando diferentes formas de se implementar a busca em uma relação básica ou as junções para um conjunto de relações. Um caminho para uma junção é uma árvore na qual o nodo raiz representa o método da junção e as subárvores da direita e da esquerda representam as relações que fazem parte da junção. Após considerar todos os possíveis caminhos, aquele com o menor custo calculado pelo módulo de estimativa de custos é selecionado. O plano final de execução da consulta é construído a partir do caminho menos custoso da `RelOptInfo` que contém todas as relações da consulta.

Para uma relação básica, os possíveis caminhos incluem as operações físicas de busca sequencial e buscas através dos índices da tabela. Já as operações de junção ocorrem entre duas `RelOptInfo`, onde uma é denominada *outer* e a outra *inner*. As relações que são *outer* direcionam a pesquisa nas relações que são *inner*. São três os métodos físicos para realização das junções no PostgreSQL:

- No *nested loop*, a relação *inner* é percorrida uma vez para cada tupla da relação *outer* até que sejam encontradas as tuplas relacionadas.
- No *merge join*, tanto a relação *inner* quanto a *outer* são ordenadas e acessadas simultaneamente em ordem, portanto é necessário percorrer o conjunto apenas uma vez para realizar a junção das duas tabelas.
- No *hash join*, a relação *inner* é percorrida e todas as suas tuplas são incluídas em uma tabela *hash*. A relação *outer* é então percorrida, utilizando a chave da junção para comparação das suas tuplas com as entradas da tabela *hash*.

Com as estruturas de dados definidas, o próximo passo para incluir o algoritmo de Kruskal no código do otimizador do PostgreSQL foi identificar o trecho de código que contém a chamada às funções de otimização de consultas. A função `make_rel_from_joinlist` verifica se a consulta contém mais de uma relação e em caso positivo, realiza chamada à função padrão de otimização de consultas ou ao algoritmo genético, de acordo com

o número de relações que formam a consulta. Esta função está localizada no arquivo `allpath.c` e um trecho do seu código original é mostrado abaixo.

```
static RelOptInfo *
make_rel_from_joinlist(PlannerInfo *root, List *joinlist)
{

...

    if (enable_geqo && levels_needed >= geqo_threshold)
        return geqo(root, levels_needed, initial_rels);
    else
        return make_one_rel_by_joins(root, levels_needed, initial_rels);
}
```

O código do PostgreSQL é bastante modularizado, o que permitiu a adição do algoritmo de Kruskal ao seu otimizador apenas incluindo a chamada ao algoritmo de Kruskal na função `make_rel_from_joinlist` mostrada anteriormente. A alteração realizada na função é mostrada a seguir.

```
static RelOptInfo *
make_rel_from_joinlist(PlannerInfo *root, List *joinlist)
{

...

    if (enable_geqo && levels_needed >= geqo_threshold && !enable_kruskal)
        return geqo(root, levels_needed, initial_rels);
    else if (enable_kruskal){
        return kruskal_make_one_rel_by_joins(root, levels_needed, initial_rels);}
    else
        return make_one_rel_by_joins(root, levels_needed, initial_rels);
}
```

A função criada para o Kruskal foi denominada `kruskal_make_one_rel_by_joins`, e seu código está no arquivo `kruskal_main.c`, criado nesta pesquisa. Essa função contém três parâmetros de entrada, sendo o primeiro uma estrutura do tipo `PlannerInfo` que contém diversas informações a respeito da consulta, o segundo é o número de tabelas presentes na consulta e o último é uma lista contendo os `RelOptInfo` das relações básicas da consulta. A nova função foi escrita de maneira que ficasse o mais semelhante possível da função que realiza a otimização de consultas pelo algoritmo padrão do PostgreSQL, denominada `make_one_rel_by_joins`.

No início da função `kruskal_make_one_rel_by_joins`, o vetor `joinitems[1]` armazena a lista recebida como parâmetro contendo as `RelOptInfo` das relações básicas da consulta. Essas `RelOptInfo` contêm os possíveis caminhos (`Path`) para acessar cada relação, seja por busca sequencial, seja por índice.

Em seguida, para cada par de `RelOptInfo` são gerados todos os caminhos representando os possíveis métodos de junção. Caso o par de tabelas não contenha predicado de junção (*where*) entre elas, um caminho representando o produto cartesiano da junção é gerado. A estimativa do custo de cada caminho é calculada e as `RelOptInfo` criadas para todas as junções possíveis entre duas tabelas são armazenadas no vetor `joinitems[2]`.

Caso existam apenas duas tabelas na consulta, o caminho mais barato é escolhido e retornado para a função inicial `make_rel_from_joinlist()`. Caso existam mais de duas tabelas, o algoritmo de Kruskal entra de fato em ação, criando uma árvore de junções. Os valores estimados dos custos das junções entre pares de tabelas são incluídos em um vetor e o mesmo é ordenado. A cada iteração, a junção de menor custo é selecionada e então avaliada. Se a inclusão dessa junção na árvore final de execução não gerar um ciclo, então ela deve ser adicionada à solução. Caso contrário, a junção é descartada e uma nova iteração é iniciada até que todas as tabelas da consulta estejam presentes na árvore final de execução.

A construção da árvore final de execução pelo algoritmo de Kruskal pode gerar árvores com topologia em profundidade à esquerda (*left-deep tree*) ou balanceada (*bushy tree*). A característica do algoritmo de Kruskal de gerar árvores com topologia balanceada amplia a quantidade de soluções possíveis.

Nas árvores em profundidade à esquerda, a relação *outer* pode ser tanto uma relação base quanto o resultado de junções (*joinrel*) anteriores, e a relação *inner* é sempre uma relação base. Nas árvores balanceadas, tanto a subárvore da esquerda quanto a subárvore da direita podem ser junções entre tabelas.

A árvore final de execução é gerada na estrutura `result_rels`. No início a `result_rels` está vazia e a `RelOptInfo` que representa a primeira junção é incluída nessa lista. A partir da segunda junção aprovada para adição na árvore, é necessária uma avaliação para

determinar como ocorrerá a junção entre as n relações. As tabelas que formam a junção a ser adicionada são comparadas com as tabelas das junções que já fazem parte da lista *result_rels* e três situações podem ocorrer:

- Nenhuma das tabelas *(i,j)* que compõem a junção a ser adicionada fazem parte das junções que estão em `result_rels`. Nesse caso, a `RelOptInfo` da nova junção *(i,j)* é adicionada à `result_rels`.

- As duas tabelas *(i,j)* que compõem a junção a ser adicionada fazem parte das junções *(i,x)* e *(j,y)* que estão em `result_rels`. Nesse caso é realizada a junção entre as duas estruturas `RelOptInfo` de *(i,x)* e *(j,y)* que estão na lista `result_rels`, resultando em uma única `RelOptInfo` que representa *join(join(i,x),join(j,y))*. Essa nova `RelOptInfo` é uma árvore balanceada que é incluída na lista `result_rels`, enquanto as `RelOptInfo` que representam as junções *(i,x)* e *(j,y)* são removidas da lista.

- Uma das tabelas *(i,j)* que compõem a junção a ser adicionada faz parte de uma das junções *(i,x)* que estão em `result_rels`. Nesse caso é realizada a junção entre a `RelOptInfo` *(i,x)* que está em `result_rels` e a `RelOptInfo` que representa a relação *j*, formando uma `RelOptInfo` com árvore em profundidade à esquerda representando *join(join(i,x),j)*. Essa nova estrutura é incluída na lista `result_rels` enquanto a `RelOptInfo` que representa a junção *(i,x)* é removida da lista.

No final do algoritmo, a `RelOptInfo` com a árvore de execução final considerada ótima e formada por todas as tabelas existentes na consulta estará em `result_rels`. Essa estrutura `RelOptInfo` é atribuída ao vetor *joinitems[3]*.

Considerando que o resultado da execução de uma junção *(i,j)* poderá afetar o custo das próximas junções que envolvem as relações i ou j, os custos das junções vizinhas a i e j deveriam ser recalculados após essa junção ser adicionada à árvore de execução final. Isso não foi realizado pois o módulo que calcula a estimativa do custo de uma junção é imbutido nas funções que criam a estrutura `RelOptInfo` da junção, e alterar a máquina de estimativa de custos está fora do escopo deste trabalho.

5.2 Ambiente Experimental

A implementação e os experimentos foram realizados em um computador com processador Intel Pentium III a 800 MHz com *cache* de 256KB integrada, 256MB de memória SDRAM a 100MHz e sistema operacional Debian Linux 3.1 (Sarge) com *kernel* Linux 2.4.27 i686.

A versão do PostgreSQL utilizada foi a 8.2devel, que era a versão mais recente de desenvolvimento no momento em que esse trabalho foi desenvolvido, e foi obtida na área destinada aos desenvolvedores da página oficial do SGBD. É importante destacar que na passagem da versão 8.1 para a versão 8.2devel houve um grande trabalho na reescrita do código do otimizador de consultas, proporcionando uma maior clareza e modularização das funções. O trabalho de inclusão do algoritmo de Kruskal no otimizador de consultas foi realizado com o código na nova estrutura.

O código fonte do PostgreSQL é armazenado e gerenciado pela ferramenta de gerenciamento de códigos chamada CVS (*Concurrent Versions System*) [14]. O PostgreSQL é um software de código livre bem organizado e documentado. Todos os códigos relacionados à máquina de otimização de consultas encontram-se em um único diretório na estrutura de arquivos que compõem o código fonte, descrita a seguir.

```
- src
  - backend
    - optimizer
      - geqo
      - path
      - plan
      - prep
      - util
```

Para inclusão dos arquivos-fonte com o algoritmo de Kruskal, foi criado um subdiretório `kruskal` em `src/backed/optimizer`.

Além da organização dos arquivos fontes, existe uma página que contém a documentação da codificação do PostgreSQL [35]. Nessa página pode ser encontrado o conteúdo de cada um dos arquivos fontes e são indicadas as relações entre esses arquivos. A referida página contém atalhos também para o código de cada função utilizada e atalhos para

arquivos com a definição dos tipos de dados utilizados. Essa documentação foi de suma importância no desenvolvimento deste trabalho.

Como o código do PostgreSQL é bastante modularizado, houve necessidade de alterar apenas um arquivo fonte para que o algoritmo de Kruskal fosse incluído no otimizador de consultas. Esse arquivo é o `allpath.c`, que contém a chamada às funções de otimização de consulta na função `make_rel_from_joinlist()`.

Para definir qual dos três algoritmos de otimização deve ser utilizado pelo SGBD, no início do arquivo `allpath.c` foi definida uma variável booleana chamada `enable_kruskal` que é ajustada para *true* se for desejado que o otimizador utilize o algoritmo de Kruskal e ajustada para *false* caso contrário.

O arquivo `kruskal_main.c` foi criado e armazenado em `src/backed/optimizer/kruskal`. Ele contém todo o código necessário para a utilização do algoritmo de Kruskal no processo de otimização de consultas do PostgreSQL.

Para testar o novo algoritmo de otimização de consultas o código do PostgreSQL foi compilado através do seu *makefile*.

5.3 Resultados dos Experimentos

A ferramenta do PostgreSQL de interação com o banco de dados utilizada chama-se `psql`. Esta ferramenta permite informar a consulta, enviá-la ao PostgreSQL e visualizar os dados resultantes. Foi habilitada a função `\timing` no `psql` para informar o tempo de execução das consultas, em milisegundos (*ms*).

5.3.1 Validação do Algoritmo de Kruskal

O objetivo do primeiro teste era validar o funcionamento do algoritmo de Kruskal no processo de otimização de consultas do PostgreSQL, sendo esperado como resultado a geração de um plano de execução de consultas no mesmo formato dos planos gerados através dos algoritmos padrões do otimizador do PostgreSQL.

Foi utilizado um modelo de dados real de um sistema de controle de acervo literário.

A base de dados é formada por 13 tabelas descritas na tabela 5.1.

tabela	♯ registros
Alocaçao	2
Atributo	100
Composição	47528
Conteudo	352354
Estado	5
Histórico	130218
Maquina_Estado	127
Material	70259
Objeto	15
Propriedade_Objeto	13
Suspensao_Usuario	7
Tabela	13
Valor_Tabela	1320

Tabela 5.1: Exemplo: modelo de dados Acervo.

A primeira consulta executada é mostrada abaixo. A consulta contém três tabelas e nenhuma cláusula de junção foi especificada. As tabelas *objeto* e *estado* têm uma quantidade pequena de registros enquanto a tabela *material* possui um número grande de registros e por isso é esperado que, com a utilização do algoritmo de Kruskal, ocorra pimeiro a junção entre as duas tabelas menores e depois a junção desse resultado com a tabela maior. O plano de execução gerado com a utilização do algoritmo de Kruskal demonstra que o algoritmo teve um funcionamento conforme esperado e é mostrado abaixo.

```
SELECT  mat_id
FROM    material, objeto, estado;
```

```
                                QUERY PLAN KRUSKAL
-------------------------------------------------------------------------------
 Nested Loop  (cost=41558.35..2686568549.17 rows=114719400000 width=4)
   ->  Seq Scan on material  (cost=0.00..1088.82 rows=67482 width=4)
   ->  Materialize  (cost=41558.35..64369.35 rows=1700000 width=0)
         ->  Nested Loop  (cost=23.75..34047.35 rows=1700000 width=0)
               ->  Seq Scan on objeto  (cost=0.00..23.60 rows=1360 width=0)
               ->  Materialize  (cost=23.75..36.25 rows=1250 width=0)
                     ->  Seq Scan on estado  (cost=0.00..22.50 rows=1250 width=0)
```

Acompanhando a execução do algoritmo de Kruskal no processo de otimização da consulta anterior, observou-se os seguintes passos.

- Primeiro são calculadas as estimativas de custos para cada uma das possíveis junções: *join(material, objeto), join(material, estado) e join(objeto, estado).*

- Em seguida ocorre a avaliação de inclusão das junções na árvore de execução, iniciando pela junção de menor custo *join(objeto, estado).* A primeira junção é sempre incluída.

- A próxima junção da lista *join(material, estado)* é avaliada e uma das tabelas que a compõem faz parte da junção incluída anteriormente. Neste caso, é realizada a junção entre a relação básica *material*, que ainda não estava incluída, com a junção *join(objeto, estado)*, já incluída anteriormente.

- A última junção *join(material, objeto)* é descartada porque formaria um ciclo na árvore se fosse incluída.

Abaixo são mostradas as informações geradas durante a execução do algoritmo de Kruskal.

```
Início do Kruskal

NumEdge = 0, edgeTail = 1, edgeHead = 3, edgeWeight = 2204572.500000
NumEdge = 1, edgeTail = 1, edgeHead = 2, edgeWeight = 2398574.000000
NumEdge = 2, edgeTail = 2, edgeHead = 3, edgeWeight = 46522.500000

tail: 2, head: 3, cost: 46522.500000 > included
Primeira aresta incluída.

tail: 1, head: 3, cost: 2204572.500000 > included
Rel2: 3 é membro existente na árvore final.
Faz o leftjoin com singlerel 1

tail: 1, head: 2, cost: 2398574.000000 > discarded

cheapest->total_cost = 2998234522.500000 num_relids = 3
Saiu do Kruskal.....
```

5.3.2 Comparação entre os Métodos de Otimização

Após validação satisfatória do funcionamento do algoritmo de Kruskal no otimizador do PostgreSQL, foram iniciados os testes de comparação entre as técnicas de otimização.

Cada consulta avaliada foi executada com o algoritmo de Kruskal e em seguida com os algoritmos do PostgreSQL de programação dinâmica ou algoritmo genético, dependendo do número de tabelas que a consulta continha.

Para medição do desempenho foi utilizada a metodologia TPC-H [2], desenvolvida pelo Transaction Processing Performance Council (TPC), que é uma corporação sem fins lucrativos fundada para definir o processamento de transações e *benchmarks* de bancos de dados. O TPC-H é um *benchmark* que mede o desempenho em sistemas de apoio à decisão, como *data warehouse*, *data mining*, BI e OLAP. O TPC-H define um banco de dados sintético e um conjunto de consultas para avaliar características do SGBD, examinando grandes volumes de dados e execução de consultas com elevado grau de complexidade. As consultas compreendidas no TPC-H são projetadas para exercitar as funcionalidades de um sistema buscando representar aplicações complexas de análises de negócios em um contexto realista, demonstrando a atividade de um fornecedor de vendas.

Para popular as tabelas do banco de dados foi utilizada uma ferramenta disponibilizada pelo TPC chamada `dbgen`. O TPC-H contém escalas variadas de tamanhos de bases de dados, a partir de 1GB até 1TB. Neste trabalho foi utilizada escala de 1GB de dados pois o objetivo é avaliar o funcionamento do algoritmo e não o seu desempenho em bases de dados de larga escala. As tabelas que compõem o TPC-H, com as quantidades de registros e páginas na escala de 1GB utilizados neste trabalho, estão descritos na tabela 5.2, .

tabela	♯ registros	♯ páginas
`Region`	5	1
`Nation`	25	1
`Supplier`	10.000	261
`Customer`	150.000	4.326
`Part`	200.000	5.132
`Partsupp`	800.000	19.500
`Orders`	1.500.000	33.704
`Lineitem`	6.000.380	168.246

Tabela 5.2: Modelo de dados TPC-H em escala de 1GB.

Como parte do *benchmark*, o TPC indica 22 consultas para recuperação de dados. Essas consultas possuem como características alto grau de complexidade, tipos de acessos

variados, natureza ad-hoc, examinam uma grande porcentagem dos dados disponíveis, abrangem as mais diversas operações de bancos de dados, além de possuirem parâmetros que se alteram entre as execuções. Para gerá-las foi utilizada a ferramenta `qgen` disponibilizada pelo TPC.

O teste executado foi o *power test*, no qual é medido o tempo de execução de cada consulta com um único usuário. As 22 consultas foram executadas sequencialmente 10 vezes, tanto utilizando o algoritmo de Kruskal quanto o algoritmo padrão do PostgreSQL. A tabela 5.3 apresenta o resultado obtido, com a média do tempo de execução das consultas em cada um dos algoritmos avaliados, e informa se os planos de execução gerados pelos algoritmos foram diferentes ou idênticos. As execuções que atingiram 200 minutos sem retornar o resultado foram consideradas com seu tempo limite de execução esgotado *timeout*.

Consultas TPC-H	Planos de execução diferentes?	Tempo médio de execução (ms)	
		Alg. Kruskal	Alg. Padrão
Q1	não	58.269	59.075
Q2	sim	24.500	15.603
Q3	não	30.695	31.414
Q4	não	97.470	94.735
Q5	sim	30.865	30.859
Q6	não	57.025	57.039
Q7	sim	30.675	30.582
Q8		erro	30.542
Q9	sim	30.945	30.429
Q10	sim	30.708	30.680
Q11	não	9.863	9.503
Q12	não	30.864	30.501
Q13	não	57.904	58.250
Q14	não	56.805	57.142
Q15	não	57.499	57.853
Q16	não	21.132	20.938
Q17	não	56.900	57.027
Q18	sim	31.152	30.511
Q19	não	57.857	58.377
Q20	sim	205	timeout
Q21	sim	30.674	30.438
Q22	não	timeout	timeout

Tabela 5.3: Comparação entre as médias dos tempos de execução utilizando o algoritmo de Kruskal e o algoritmo padrão do PostgreSQL.

No geral, observa-se que os tempos de execução das consultas através dos algoritmos avaliados são bastante próximos. Isso indica que o algoritmo de Kruskal gera resultados na otimização de consultas tão bons quanto os algoritmos implementados no PostgreSQL.

A consulta Q8 não pode ser executada através do Kruskal devido a um erro no qual o banco de dados foi encerrado inesperadamente e por isso a comparação não pode ser realizada. Este erro não foi analisado neste trabalho.

A maioria das consultas obtiveram o mesmo plano de execução tanto na geração através do algoritmo de Kruskal quanto na geração através do algoritmo padrão do PostgreSQL. Foram 13 consultas nessa situação onde:

- 8 consultas (Q1, Q3, Q6, Q13, Q14, Q15, Q17, Q19) obtiveram tempo de execução médio um pouco menor quando executados através do Kruskal. Isto pode ser explicado pelo algoritmo de Kruskal não gerar tantas soluções no espaço de busca quantas são geradas pelo algoritmo de programação dinâmica utilizado no PostgreSQL, necessitando portanto de um tempo menor para indicar a solução.

- 4 consultas (Q4, Q11, Q12, Q16) obtiveram tempo de execução médio menor quando utilizado o algoritmo padrão do PostgreSQL. A consulta Q4 foi a que apresentou a maior diferença, as demais consultas tiveram seus tempos de execução muito próximos utilizando os dois algoritmos.

- 1 consulta (Q22) permaneceu em execução sem retornar resultado por 200 minutos e seu tempo limite de execução foi considerado atingido, tanto com o algoritmo de Kruskal quanto com o algoritmo padrão do PostgreSQL.

A tabela 5.4 mostra os custos dos planos gerados através do algoritmo de Kruskal e do algoritmo padrão do PostgreSQL para as consultas que tiveram planos de execução diferentes para o Kruskal e o algoritmo de programação dinâmica.

Consultas TPC-H	Custo do plano de execução	
	Alg. Kruskal	Alg. Padrão
Q2	117617.38..117617.38	57847.19..57847.19
Q5	22.80..22.81	22.75..22.76
Q7	41.40..41.43	23.05..23.09
Q9	27.30..27.34	27.30..27.34
Q10	8838.01..8838.02	17.53..17.54
Q18	21.14..21.17	21.14..21.17
Q20	433.60..434.60	3372633.77..3372633.78
Q21	25.77..25.77	25.77..25.77

Tabela 5.4: Custos mínimos e máximos dos planos de execução gerados através do algoritmo de Kruskal e do algoritmo padrão do PostgreSQL.

Das 8 consultas que obtiveram planos de execução diferentes gerados através dos algoritmos padrão do PostgreSQL e do Kruskal:

- 6 consultas (Q5, Q7, Q9, Q10, Q18, Q21) obtiveram tempo médio de execução levemente melhor através do algoritmo padrão do PostgreSQL, onde a diferença entre os tempos dos algoritmos foi sempre menor do que 0,5 segundo. Alguns planos de execução gerados pelo algoritmo de Kruskal apresentaram custos um pouco maiores do que os custos dos planos gerados pelo algoritmo de programação dinâmica. Outros apresentaram custos iguais apesar da ordem das junções e operações serem diferentes. Destaque para a consulta Q10 que apresentou um custo extremamente maior pelo Kruskal mas manteve os tempos médios de execução muito próximos nos dois algoritmos avaliados.

- 1 consulta (Q2) apresentou tempo médio de execução muito maior quando foi utilizado o algoritmo de Kruskal, praticamente dobrando o tempo de execução obtido com o algoritmo padrão. Isso ocorreu porque o plano de execução gerado pelo Kruskal obteve custo maior do que o plano gerado pelo algoritmo padrão.

- 1 consulta (Q20) apresentou tempo médio de execução extremamente menor utilizando o algoritmo de Kruskal, pois com o algoritmo padrão a consulta não retornou resultados após 233 minutos de execução e portanto foi considerado atingido o tempo limite de execução. Isso ocorreu devido aos planos de execução gerados,

onde o Kruskal obteve um plano com custo extremamente baixo enquanto o oposto ocorreu com o plano gerado pelo algoritmo de busca exaustiva, que obteve um custo extremamente alto.

Uma diferença entre o algoritmo implementado e o algoritmo padrão de busca exaustiva do PostgreSQL é que o Kruskal gera apenas uma árvore de execução de consultas, que será a árvore final, enquanto o algoritmo padrão do PostgreSQL gera diversas árvores e então as compara para escolher a solução final. Por isso, quanto maior o número de relações que a consulta tiver, o processo de construção do plano ótimo através do algoritmo de Kruskal será mais rápido.

O algoritmo genético implementado no PostgreSQL não pôde ser testado com as consultas do TPC-H pois as mesmas não atingem o número mínimo necesário de tabelas para acionar o algoritmo GEQO. Portanto foram construídas e executadas consultas englobando mais de 11 tabelas, utilizando o modelo de dados apresentado na tabela 5.1. Uma das consultas construídas contém predicados de junção entre todas as tabelas e foi denominada Q23. A outra consulta denominada Q24 não contém predicados e as junções são realizadas por produto cartesiano entre as relações. As consultas Q23 e Q24 são mostradas respectivamente abaixo.

Q23:

```
select  *
from    material m, estado e, conteudo c, historico h, atributo a,
        tabela t, valor_tabela v, maquina_estado q, objeto o,
        estado e1, atributo a1, estado e2
where   m.mat_estado = e.est_id and c.con_mat_id = m.mat_id
        and h.his_mat_id = m.mat_id and c.con_atr_id = a.atr_id
        and t.tab_id = a.atr_tab_id and v.val_tab_id = t.tab_id
        and q.maq_est_ini = e.est_id and o.obj_id = m.mat_obj_id
        and e1.est_id = e.est_id and a1.atr_id = a.atr_id
        and e2.est_id = q.maq_est_fin
limit 100;
```

Q24:

```
select  *
from    material m, estado e, conteudo c, historico h, atributo a,
        tabela t, valor_tabela v, maquina_estado q, objeto o, estado e1,
        material m1, historico h1, atributo a1, conteudo c1
limit 10000;
```

Foi observado que cada execução de uma mesma consulta utilizando o algoritmo genético gera um plano de execução diferente. Isso ocorre devido à característica dessa classe de algoritmos, que utiliza valores aleatórios como *fitness*. A tabela 5.5 demonstra os resultados obtidos na execução das consultas Q23 e Q24 através do algoritmo de Kruskal e do algoritmo genético.

Consultas Acervo	Planos de execução diferentes?	Tempo de execução (ms)	
		Alg. Kruskal	Alg. Genético
`Q23`	sim	26.779	30.609
		26.043	8.352
		27.668	12.199
		26.590	124.516
		26.376	662
		26.362	20.392
		26.175	41.954
		26.177	9.288
		26.188	31.078
`média`		26.484	31.006
`Q24`	sim	1.131	1.430
		1.108	1.187
		1.108	1.188
		1.160	1.368
		1.103	1.191
		1.103	1.188
		1.103	1.194
`média`		1.117	1.250

Tabela 5.5: Comparação entre as médias dos tempos de execução utilizando o algoritmo de Kruskal e o algoritmo GEQO do PostgreSQL.

A consulta Q23 com o algoritmo GEQO apresenta uma grande variação no tempo de execução em suas diversas execuções, enquanto que com o algoritmo de Kruskal o tempo de execução mantém uma regularidade. A média obtida neste experimento foi menor com a utilização do algoritmo de Kruskal, mas o resultado pode ser diferente em outros testes pois o tempo de execução através do algoritmo genético depende dos valores aleatórios gerados como *fitness* para o algoritmo.

A consulta Q24 apresentou tempos menores em todas as execuções utilizando o algoritmo de Kruskal. Todos os planos de execução gerados pelo GEQO obtiveram o mesmo custo apesar das junções serem ordenadas de formas diferentes em cada plano. O custo

mínimo do plano gerado pelo Kruskal foi sempre menor do que o custo mínimo do plano gerado pelo algoritmo genético e isto proporcionou um ganho na execução por Kruskal, embora o custo máximo do plano gerado pelo Kruskal seja um pouco maior. Os custos dos planos de execução para as consultas Q23 e Q24 são mostrados na tabela 5.6.

Consultas Acervo	Custo do plano de execução	
	Alg. Kruskal	Alg. Genético
Q23	175.30..179109.06	106.54..5122.23 2292.94..37963.44 9.97..31.69 4005.88..9400.47
Q24	0.00..274.44	24.52..224.72

Tabela 5.6: Custos mínimos e máximos dos planos de execução gerados através do algoritmo de Kruskal e do algoritmo GEQO do PostgreSQL.

Na maioria dos casos, o algoritmo de kruskal retornou os resultados desejados em um tempo bastante próximo do obtido com os outros algoritmos, tanto oferecendo um tempo menor quanto um tempo maior, sendo portanto considerado um algoritmo viável para a técnica de otimização de consultas.

CAPÍTULO 6

CONCLUSÃO E TRABALHOS FUTUROS

O processo de tradução de uma consulta em alto-nível para seus possíveis planos de execução, com posterior avaliação desses planos e escolha do plano considerado ótimo, é fundamental na garantia da rapidez na entrega dos dados solicitados pelos usuários e aplicações.

Ao longo dos anos, os mais diversos métodos de otimização de consultas e combinações deles foram estudados e implementados, tanto em Sistemas Gerenciadores de Bancos de Dados (SGBD) disponíveis no mercado, quanto em protótipos de projetos desenvolvidos por universidades e centros de pesquisas, mas nenhum desses métodos realiza o processo de forma totalmente satisfatória para todas as classes de consultas e por isso não há a definição do melhor método. Portanto, novas pesquisas nessa área continuam sendo interessantes.

Com o fortalecimento do movimento do software livre e disponibilização do código-fonte de SGBDs, as pesquisas na área de otimização de consultas ganham a possibilidade de aplicação prática da teoria estudada, com a validação e comparação dos métodos em um ambiente real.

Nesta pesquisa foi proposta uma nova metodologia de otimização de consultas para o SGBD PostgreSQL, utilizando o algoritmo de Kruskal para gerar o plano ótimo de execução de consultas. A implementação do algoritmo de Kruskal no otimizador de consultas do PostgreSQL foi possível devido à grande organização e modularização do código-fonte desse SGBD.

Na maioria dos experimentos realizados o algoritmo de Kruskal retornou os resultados esperados em tempos bastante próximos dos obtidos com os algoritmos implementados no PostgreSQL, demonstrando que o Kruskal é um algoritmo viável para a técnica de otimização de consultas.

Uma vantagem na utilização do algoritmo de Kruskal é a sua fácil implementação devido à grande simplicidade desse algoritmo. Outra vantagem é o seu espaço de busca reduzido pois apenas uma árvore de execução de consultas é gerada como árvore final, enquanto o algoritmo de programação dinâmica do PostgreSQL gera diversas árvores e então as compara para escolher a solução final. Por isso, quanto maior o número de relações que compõem a consulta, o processo de construção do plano ótimo através do algoritmo de Kruskal será mais rápido.

Em relação às consultas que contêm mais de 11 tabelas e que são otimizadas através do algoritmo genético do PostgreSQL, o Kruskal gerou bons resultados. Ao utilizá-lo para esse conjunto de consultas é garantido que os resultados desejados serão obtidos em um tempo aceitável, situação que não é constante e certa com a utilização do algoritmo genético devido ao seu teor aleatório.

Trabalhos Futuros Utilizando o algoritmo de Kruskal, quando uma junção é selecionada e adicionada na árvore final de execução, os custos das junções vizinhas são afetados e deveriam ser recalculados para refletir os novos tamanhos e seletividades. Esta atualização dos custos no grafo de consultas é um possível trabalho a ser realizado. Definir com maior exatidão quais grupos de consultas se beneficiam da utilização do Kruskal também é um trabalho futuro.

Outro trabalho a ser realizado é a revisão do módulo do PostgreSQL que define o custo dos planos de execução, verificando a eficiência dos cálculos de estimativa de custo utilizados atualmente.

Realizar a tradução do problema de otimização de consultas para a visão do planejamento automático utilizando PDDL também é um trabalho interessante.

APÊNDICE A

PROGRAMA KRUSKAL NO POSTGRESQL

```
1   /*-------------------------------------------------------------------------
     *
     * kruskal_main.c
     *        solution to the query optimization problem
     *        by means of Kruskal Algorithm
     *
     * Portions Copyright (c) 1996-2006, PostgreSQL Global Development Group
     * Portions Copyright (c) 1994, Regents of the University of California
     *
     *-------------------------------------------------------------------------
     */

    /* contributed by:
     =*=========================================================*=
     *  Pryscila Barvick Guttoski          * Departamento de Informática     *
     *                                     * Universidade Federal do Paraná  *
     *  pryscila@ufpr.com.br               * Curitiba, Brasil                *
     =*=========================================================*=
    */


    #include "postgres.h"
    #include "nodes/print.h"
    #include <math.h>

    #include "optimizer/kruskal.h"
    #include "optimizer/pathnode.h"

    static List *make_rels_by_clause_joins(PlannerInfo *root,
                                                RelOptInfo *old_rel,
                                                ListCell *other_rels);
    static List *make_rels_by_clauseless_joins(PlannerInfo *root,
                                                RelOptInfo *old_rel,
                                                ListCell *other_rels);
    static bool has_join_restriction(PlannerInfo *root, RelOptInfo *rel);

37  /*************************************************************
     *  Estrutura utilizada pelo Kruskal para comparar os
     *  custos das arestas
    *************************************************************/
    typedef struct edge {
      RelOptInfo *rel;
      double weight;
      int tail, head;
      } edgeType;

    /*************************************************************
     *  Otimização de consultas utilizando o algoritmo de Kruskal
     *  para gerar o plano de execução de consultas ótimo
    *************************************************************/
    RelOptInfo *
    kruskal_make_one_rel_by_joins(PlannerInfo *root, int levels_needed, List *initial_rels)
    {
            List            **joinitems;
            int             lev;
            RelOptInfo      *rel;
            ListCell        *x;
            int             numEdges, numEdge;
            edgeType        *edgeTab;

            numEdges = 0;
```

```
        numEdge = 0;

        joinitems = (List **) palloc0((levels_needed + 1) * sizeof(List *));

        /*************************************************************
         *  joinitems[1] armazena a lista com os RelOptInfos
         *  referentes às relações bases recebidas como parâmetro
         *  da função
        *************************************************************/
        joinitems[1] = initial_rels;

        /*************************************************************
74      *  joinitems[2] armazena a lista com os RelOptInfos
         *  referentes às junções entre todos os pares possíveis
         *  de relações existentes na consulta
        *************************************************************/
        joinitems[2] = kruskal_make_rels_by_joins(root, joinitems);

        foreach(x, joinitems[2])
                numEdges++;

        edgeTab = (edgeType*) palloc0(numEdges * sizeof(edgeType));

        /*************************************************************
         *  Para cada junção gerada entre 2 tabelas, identifica
         *  o caminho de menor custo e armazena as informações
         *  que serão utilizadas pelo Kruskal na estrutura edgeTab
        *************************************************************/
        foreach(x, joinitems[2])
        {
                rel = (RelOptInfo *) lfirst(x);
                set_cheapest(rel);

                edgeTab[numEdge].rel = rel;
                edgeTab[numEdge].tail = bms_first_member(rel->relids);
                edgeTab[numEdge].head = bms_first_member(rel->relids);
                rel->relids = bms_add_member(rel->relids, edgeTab[numEdge].tail);
                rel->relids = bms_add_member(rel->relids, edgeTab[numEdge].head);
                edgeTab[numEdge].weight = rel->cheapest_startup_path->total_cost;

                numEdge++;
        }

        /*************************************************************
         *  joinitems[3] armazena o RelOptInfo final gerado pelo
         *  algoritmo de Kruskal, que contém o plano de execução
         *  ótimo.
        *************************************************************/
        joinitems[3] = kruskal(root, joinitems, edgeTab, numEdges, levels_needed);
111

        /*************************************************************
         *  Define o caminho de menor custo do RelOptInfo final
         *  gerado pelo Kruskal
        *************************************************************/
        foreach(x, joinitems[3])
        {
                rel = (RelOptInfo *) lfirst(x);
                set_cheapest(rel);
        }

        /*************************************************************
         *  Retorna a função inicial o RelOptInfo que representa
         *  o plano de execução ótimo encontrado pelo Kruskal
        *************************************************************/
        rel = (RelOptInfo *) linitial(joinitems[3]);
        return rel;
}


/*************************************************************
 *  Função que realiza a junção entre todos os pares
 *  possíveis de relações básicas existentes na consulta
```

```
*************************************************************/
List *
kruskal_make_rels_by_joins(PlannerInfo *root, List **initialrels)
{
        List       *result_rels = NIL;
        List       *new_rels;
        ListCell   *r;

        /*************************************************************
         *  Para cada relação básica da consulta, cria as estruturas
         *  que representam a junção dela com cada uma das outras
         *  relações básicas.
        *************************************************************/
148     foreach(r, initialrels[1])
        {
                RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);
                ListCell   *other_rels;

                other_rels = lnext(r);

                if (old_rel->joininfo != NIL)
                {
                        new_rels = make_rels_by_clause_joins(root, old_rel, other_rels);

                        if (new_rels == NIL && has_join_restriction(root, old_rel))
                                new_rels = make_rels_by_clauseless_joins(root,old_rel, other_rels);
                }
                else
                {
                        new_rels = make_rels_by_clauseless_joins(root, old_rel, other_rels);
                }

                result_rels = list_concat_unique_ptr(result_rels, new_rels);
        }

        if (result_rels == NIL && root->oj_info_list == NIL && root->in_info_list == NIL)
                elog(ERROR, "failed to build any 2-way joins");

        return result_rels;
}


/*************************************************************
 * Funções do Algoritmo de Kruskal
*************************************************************/

int find(int x, int *parent)
{
        int i, root;

185     for (i=x; parent[i]!=i; i=parent[i]);
        root = i;

        return root;
}

void makeEquivalent(int i, int j, int *weight, int *parent, int numTrees)
{
        if (weight[i]>weight[j])
        {
                parent[j]=i;
                weight[i]+=weight[j];
        }
        else
        {
                parent[i]=j;
                weight[j]+=weight[i];
        }
        numTrees--;
}

int weightAscending(const void* xin, const void* yin)
// Used in call to qsort()
```

```
{
        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;
}
```

222

```
/*************************************************************
 *  Gera a estrutura contendo o plano de execução ótimo
 *  para uma consulta utilizando o algoritmo de Kruskal
*************************************************************/
List *
kruskal(PlannerInfo *root, List **initialrels, edgeType *edgeTab, int numEdges, int numVertices)
{
        List    *result_rels = NIL;
        List    *new_rel;
        int     i, numTrees;
        int     *parent, *weight;
        int     root1, root2;


        /*************************************************************
         *  Estruturas que servirão como base para o funcionamento
         *  do Kruskal
        *************************************************************/
        parent = (int*) palloc0(numVertices * sizeof(int));
        weight = (int*) palloc0(numVertices * sizeof(int));

        for (i=0; i < numVertices+1; i++)
        {
                parent[i] = i;
                weight[i] = 1;
        }

        /*************************************************************
         *  Ordena as junções entre os pares de tabelas
         *  pelos seus custos
        *************************************************************/
        numTrees = numVertices;
        qsort(edgeTab, numEdges, sizeof(edgeType), weightAscending);

        ListCell   *l, *r;
        RelOptInfo *jrel, *rel;
259     RelOptInfo *rel1=NIL, *rel2=NIL;
        int member_join;

        for (i=0; i < numEdges; i++)
        {
                root1 = find(edgeTab[i].tail, parent);
                root2 = find(edgeTab[i].head, parent);

                /*************************************************************
                 *  Se a inclusão da aresta formar um ciclo na árvore de
                 *  execução que está sendo construída, a aresta é descartada
                *************************************************************/
                if (root1 == root2)
                {
                        printf("\ntail: %d, head: %d, cost: %f > discarded\n", edgeTab[i].tail, edgeTab[i].head, edgeTab
                        fflush(stdout);
                }
                /*************************************************************
                 *  ...senão a aresta deve fazer parte da árvore de execução
                 *  e a sua inclusão é realizada.
                *************************************************************/
                else
```

```
{
        makeEquivalent(root1, root2, weight, parent, numTrees);
        printf("\ntail: %d, head: %d, cost: %f > included\n", edgeTab[i].tail, edgeTab[i].head, edgeTab[

        ListCell   *x;

        /*********************************************************
         *  Adiciona a primeira junção avaliada na lista que formará
         *  a árvore de execução final
        *********************************************************/
        if (result_rels==NIL)
        {
                printf("Primeira aresta incluída.\n");fflush(stdout);
                result_rels = lappend(result_rels, edgeTab[i].rel);
        }
296     /*********************************************************
         *  Se já existem nodos na árvore de execução final, avalia
         *  como adiconar a junção na árvore
        *********************************************************/
        else
        {
                rel1 = rel2 = NIL;
                foreach(l, result_rels)
                {
                        RelOptInfo *other_rel = (RelOptInfo *) lfirst(l);

                        if (bms_is_member(edgeTab[i].tail, other_rel->relids))
                        {
                                rel1 = other_rel;
                                member_join = edgeTab[i].head;
                        }
                        if (bms_is_member(edgeTab[i].head, other_rel->relids))
                        {
                                rel2 = other_rel;
                                member_join = edgeTab[i].tail;
                        }
                }

                if ((rel1==NIL) && (rel2==NIL))
                {
                        result_rels = lappend(result_rels, edgeTab[i].rel);
                        printf("Não teve join\n");fflush(stdout);
                }
                else if ((rel1!=NIL) && (rel2!=NIL))
                {
                        printf("Faz o bushy join.\n"); fflush(stdout);
                        new_rel = make_rel_by_clauseless_join(root, rel1, rel2);

                        if (new_rel)
                        {
                                result_rels = list_concat_unique_ptr(result_rels, new_rel);
                                result_rels = list_delete_ptr(result_rels, rel1);
333                             result_rels = list_delete_ptr(result_rels, rel2);
                                rel1 = rel2 = NIL;
                        }

                }
                else
                {
                        foreach(r, initialrels[1])
                        {

                                RelOptInfo *single_rel = (RelOptInfo *) lfirst(r);
                                Relids *relids;

                                relids = single_rel->relids;

                                if (bms_is_member(member_join, relids))
                                {
                                        printf("Faz o leftjoin com singlerel %d\n", member_join);fflush(

                                        if (rel1!=NIL)
                                        {
```

```
                                                        new_rel = make_rel_by_clauseless_join(root, rel1, single
                                                        result_rels = list_concat_unique_ptr(result_rels, new_re
                                                        Assert(list_member_ptr(result_rels, rel1));
                                                        result_rels = list_delete_ptr(result_rels, rel1);
                                                        rel1 = NIL;
                                                }
                                                else
                                                {
                                                        new_rel = make_rel_by_clauseless_join(root, rel2, single
                                                        result_rels = list_concat_unique_ptr(result_rels, new_re
                                                        Assert(list_member_ptr(result_rels, rel2));
                                                        result_rels = list_delete_ptr(result_rels, rel2);
                                                        rel2 = NIL;
                                                }

                                        }
370
                                }

                        }

                        foreach(x, result_rels)
                        {
                                rel = (RelOptInfo *) lfirst(x);
                                set_cheapest(rel);
                        }

                }
        }
        return result_rels;
}


/*
 *  * make_rels_by_clause_joins
 *   *        Build joins between the given relation 'old_rel' and other relations
 *    *        that are mentioned within old_rel's joininfo list (i.e., relations
 *     *        that participate in join clauses that 'old_rel' also participates in).
 *      *        The join rel nodes are returned in a list.
 *       *
 *        * 'old_rel' is the relation entry for the relation to be joined
 *         * 'other_rels': the first cell in a linked list containing the other
 *          * rels to be considered for joining
 *           *
 *            * Currently, this is only used with initial rels in other_rels, but it
 *             * will work for joining to joinrels too.
 *              */
static List *
make_rels_by_clause_joins(PlannerInfo *root,
                                                RelOptInfo *old_rel,
                                                ListCell *other_rels)
{
407     List       *result = NIL;
        ListCell   *l;

        for_each_cell(l, other_rels)
        {
                RelOptInfo *other_rel = (RelOptInfo *) lfirst(l);

                if (!bms_overlap(old_rel->relids, other_rel->relids) &&
                        have_relevant_joinclause(old_rel, other_rel))
                {
                        RelOptInfo *jrel;

                        jrel = make_join_rel(root, old_rel, other_rel);
                        if (jrel)
                                result = lcons(jrel, result);
                }
        }
        return result;
}
```

```
/*
 *   make_rels_by_clauseless_joins
 *           Given a relation 'old_rel' and a list of other relations
 *           'other_rels', create a join relation between 'old_rel' and each
 *           member of 'other_rels' that isn't already included in 'old_rel'.
 *           The join rel nodes are returned in a list.
 *
 *           'old_rel' is the relation entry for the relation to be joined
 *           'other_rels': the first cell of a linked list containing the
 *            other rels to be considered for joining
 *
 *            Currently, this is only used with initial rels in other_rels, but it would
 *            work for joining to joinrels too.
 */
static List *
444 make_rels_by_clauseless_joins(PlannerInfo *root,
                                                        RelOptInfo *old_rel,
                                                        ListCell *other_rels)
{
        List       *result = NIL;
        ListCell   *i;

        for_each_cell(i, other_rels)
        {
                RelOptInfo *other_rel = (RelOptInfo *) lfirst(i);

                if (!bms_overlap(other_rel->relids, old_rel->relids))
                {
                        RelOptInfo *jrel;

                        jrel = make_join_rel(root, old_rel, other_rel);

                        /*
                         *                          * As long as given other_rels are distinct, don't need to test to
                         *                                                  * see if jrel is already part of output lis
                         *
                         */

                        if (jrel)
                                result = lcons(jrel, result);
               }
        }

        return result;
}

static List *
make_rel_by_clauseless_join(PlannerInfo *root,
                            RelOptInfo *old_rel,
                            RelOptInfo *other_rel)
{
        List       *result = NIL;
481     RelOptInfo *jrel;

        jrel = make_join_rel(root, old_rel, other_rel);

                        /*
                         *   * As long as given other_rels are distinct, don't need to test to
                         *   * see if jrel is already part of output list.
                         *
                         */

        if (jrel)
                result = lcons(jrel, result);

        return result;
}


/*
 *  has_join_restriction
```

```
 *              Detect whether the specified relation has join-order restrictions
 *              due to being inside an OJ RHS or an IN (sub-SELECT).
 */
static bool
has_join_restriction(PlannerInfo *root, RelOptInfo *rel)
{
        ListCell   *l;

        foreach(l, root->oj_info_list)
        {
                OuterJoinInfo *ojinfo = (OuterJoinInfo *) lfirst(l);

                if (bms_is_subset(rel->relids, ojinfo->min_righthand))
                        return true;
        }

        foreach(l, root->in_info_list)
        {
                InClauseInfo *ininfo = (InClauseInfo *) lfirst(l);

                if (bms_is_subset(rel->relids, ininfo->righthand))
                        return true;
        }
       return false;
}

/*
 *  * have_relevant_joinclause
 *   *              Detect whether there is a joinclause that can be used to join
 *    *              the two given relations.
 *     */
bool
kruskal_have_relevant_joinclause(RelOptInfo *rel1, RelOptInfo *rel2)
{
        bool            result = false;
        Relids          join_relids;
        List       *joininfo;
        ListCell   *l;

        join_relids = bms_union(rel1->relids, rel2->relids);

        if (list_length(rel1->joininfo) <= list_length(rel2->joininfo))
                joininfo = rel1->joininfo;
        else
                joininfo = rel2->joininfo;

        int i=0;
        foreach(l, joininfo)
        {
                RestrictInfo *rinfo = (RestrictInfo *) lfirst(l);

                if (bms_is_subset(rinfo->required_relids, join_relids))
                {
                        result = true;
                        break;
                }
        }

        bms_free(join_relids);

        return result;
}
```

APÊNDICE B

FUNÇÃO MAKE_REL_FROM_JOINLIST() DO POSTGRESQL INCLUINDO O KRUSKAL

```
1  /*-------------------------------------------------------------------------
    *
    * allpaths.c
    *        Routines to find possible search paths for processing a query
    *
    * Portions Copyright (c) 1996-2006, PostgreSQL Global Development Group
    * Portions Copyright (c) 1994, Regents of the University of California
    *
    *
    * IDENTIFICATION
    *        $PostgreSQL: pgsql/src/backend/optimizer/path/allpaths.c,v 1.144 2006/03/05 15:58:28 momjian Exp $
    *
    *-------------------------------------------------------------------------
    */

   #include "postgres.h"
   ...


   /* These parameters are set by GUC */
   bool            enable_geqo = false;    /* just in case GUC doesn't set it */
   bool            enable_kruskal = true;
   //bool          enable_kruskal = false;
   int                     geqo_threshold;


   /*
    * make_rel_from_joinlist
    *        Build access paths using a "joinlist" to guide the join path search.
    *
    * See comments for deconstruct_jointree() for definition of the joinlist
    * data structure.
    */
   static RelOptInfo *
   make_rel_from_joinlist(PlannerInfo *root, List *joinlist)
   {
37         int                     levels_needed;
           List       *initial_rels;
           ListCell   *jl;

           /*
            * Count the number of child joinlist nodes.  This is the depth of the
            * dynamic-programming algorithm we must employ to consider all ways of
            * joining the child nodes.
            */
           levels_needed = list_length(joinlist);

           if (levels_needed <= 0)
                   return NULL;                    /* nothing to do? */

           /*
            * Construct a list of rels corresponding to the child joinlist nodes.
            * This may contain both base rels and rels constructed according to
            * sub-joinlists.
            */
           initial_rels = NIL;
           foreach(jl, joinlist)
```

```
        {
                Node       *jlnode = (Node *) lfirst(jl);
                RelOptInfo *thisrel;

                if (IsA(jlnode, RangeTblRef))
                {
                        int                     varno = ((RangeTblRef *) jlnode)->rtindex;

                        thisrel = find_base_rel(root, varno);
                }
                else if (IsA(jlnode, List))
                {
                        /* Recurse to handle subproblem */
                        thisrel = make_rel_from_joinlist(root, (List *) jlnode);
                }
                else
74              {
                        elog(ERROR, "unrecognized joinlist node type: %d",
                                 (int) nodeTag(jlnode));
                        thisrel = NULL;         /* keep compiler quiet */
                }

                initial_rels = lappend(initial_rels, thisrel);
        }

        if (levels_needed == 1)
        {
                /*
                 * Single joinlist node, so we're done.
                 */
                return (RelOptInfo *) linitial(initial_rels);
        }
        else
        {
                /*
                 * Consider the different orders in which we could join the rels,
                 * using either GEQO, regular optimizer or Kruskal.
                 */
                if (enable_geqo && levels_needed >= geqo_threshold && !enable_kruskal)
                        return geqo(root, levels_needed, initial_rels);
                else
                if (enable_kruskal)
                        return kruskal_make_one_rel_by_joins(root, levels_needed, initial_rels);
                else
                        return make_one_rel_by_joins(root, levels_needed, initial_rels);
        }
}
```

APÊNDICE C

CONSULTAS DO TPC-H SUBMETIDAS AO POSTGRESQL NOS EXPERIMENTOS

```
1   -- using 594367600 as a seed to the RNG
    \timing

    select
            l.returnflag,
            l.linestatus,
            sum(l.quantity) as sumqty,
            sum(l.extendedprice) as sumbaseprice,
            sum(l.extendedprice * (1 - l.discount)) as sumdiscprice,
            sum(l.extendedprice * (1 - l.discount) * (1 + l.tax)) as sumcharge,
            avg(l.quantity) as avgqty,
            avg(l.extendedprice) as avgprice,
            avg(l.discount) as avgdisc,
            count(*) as countorder
    from
            lineitem l
    where
            l.shipdate <= date '1998-12-01' - interval '87 days'
    group by
            l.returnflag,
            l.linestatus
    order by
            l.returnflag,
            l.linestatus;
    --#SET ROWS_FETCH -1
    -- Q1

    select
            s.acctbal,
            s.name,
            n.name,
            p.partkey,
            p.mfgr,
            s.address,
            s.phone,
            s.comment
37  from
            part p,
            supplier s,
            partsupp ps,
            nation n,
            region r
    where
            p.partkey = ps.partkey
            and s.suppkey = ps.suppkey
            and p.size = 46
            and p.type like '%BRASS'
            and s.nationkey = n.nationkey
            and n.regionkey = r.regionkey
            and r.name = 'AMERICA'
            and ps.supplycost = (
                    select
                            min(ps2.supplycost)
                    from
                            partsupp ps2,
                            supplier s2,
                            nation n2,
                            region r2
```

```
                where
                        p.partkey = ps2.partkey
                        and s2.suppkey = ps2.suppkey
                        and s2.nationkey = n2.nationkey
                        and n2.regionkey = r2.regionkey
                        and r2.name = 'AMERICA'
        )
order by
        s.acctbal desc,
        n.name,
        s.name,
        p.partkey;
--#SET ROWS_FETCH 100
-- Q2
```

74

```
select
        l.orderkey,
        sum(l.extendedprice * (1 - l.discount)) as revenue,
        o.orderdate,
        o.shippriority
from
        customer c,
        orders o,
        lineitem l
where
        c.mktsegment = 'AUTOMOBILE'
        and c.custkey = o.custkey
        and l.orderkey = o.orderkey
        and o.orderdate < date '1995-03-09'
        and l.shipdate > date '1995-03-09'
group by
        l.orderkey,
        o.orderdate,
        o.shippriority
order by
        revenue desc,
        o.orderdate;
--#SET ROWS_FETCH 10
-- Q3


select
        o.orderpriority,
        count(*) as ordercount
from
        orders o
where
        o.orderdate >= date '1994-05-01'
        and o.orderdate < date '1994-05-01' + interval '3 months'
        and exists (
                select
                        *
                from
                        lineitem l
                where
                        l.orderkey = o.orderkey
                        and l.commitdate < l.receiptdate
        )
group by
        o.orderpriority
order by
        o.orderpriority;
--#SET ROWS_FETCH -1
-- Q4


select
        n.name,
        sum(l.extendedprice * (1 - l.discount)) as revenue
from
        customer c,
        orders o,
```

111

```
        lineitem l,
        supplier s,
        nation n,
        region r
where
        c.custkey = o.custkey
        and l.orderkey = o.orderkey
        and l.suppkey = s.suppkey
        and c.nationkey = s.nationkey
        and s.nationkey = n.nationkey
        and n.regionkey = r.regionkey
        and r.name = 'AFRICA'
        and o.orderdate >= date '1996-01-01'
        and o.orderdate < date '1996-01-01' + interval '1 year'
group by
        n.name
order by
        revenue desc;
--#SET ROWS_FETCH -1
-- Q5


select
        sum(l.extendedprice * l.discount) as revenue
from
        lineitem l
where
        l.shipdate >= date '1996-01-01'
        and l.shipdate < date '1996-01-01' + interval '1 year'
        and l.discount between 0.05 - 0.01 and 0.05 + 0.01
        and l.quantity < 24;
--#SET ROWS_FETCH -1
-- Q6


select
        suppnation,
        custnation,
        lyear,
        sum(volume) as revenue
from
        (
                select
                        n1.name as suppnation,
                        n2.name as custnation,
                        extract(year from l.shipdate) as lyear,
                        l.extendedprice * (1 - l.discount) as volume
                from
                        supplier s,
                        lineitem l,
                        orders o,
                        customer c,
                        nation n1,
                        nation n2
                where
                        s.suppkey = l.suppkey
                        and o.orderkey = l.orderkey
                        and c.custkey = o.custkey
                        and s.nationkey = n1.nationkey
                        and c.nationkey = n2.nationkey
                        and (
                                (n1.name = 'JAPAN' and n2.name = 'IRAN')
                                or (n1.name = 'IRAN' and n2.name = 'JAPAN')
                        )
                        and l.shipdate between date '1995-01-01' and date '1996-12-31'
        ) as shipping
group by
        suppnation,
        custnation,
        lyear
order by
        suppnation,
        custnation,
```

```
        lyear;
--#SET ROWS_FETCH -1
-- Q7


select
        year,
        sum(case
                when nation = 'IRAN' then volume
                else 0
        end) / sum(volume) as mktshare
from
        (
                select
                        extract(year from o.orderdate) as year,
                        l.extendedprice * (1 - l.discount) as volume,
                        n2.name as nation
222             from
                        part p,
                        supplier s,
                        lineitem l,
                        orders o,
                        customer c,
                        nation n1,
                        nation n2,
                        region r
                where
                        p.partkey = l.partkey
                        and s.suppkey = l.suppkey
                        and l.orderkey = o.orderkey
                        and o.custkey = c.custkey
                        and c.nationkey = n1.nationkey
                        and n1.regionkey = r.regionkey
                        and r.name = 'MIDDLE EAST'
                        and s.nationkey = n2.nationkey
                        and o.orderdate between date '1995-01-01' and date '1996-12-31'
                        and p.type = 'SMALL POLISHED COPPER'
        ) as allnations
group by
        year
order by
        year;
--#SET ROWS_FETCH -1
-- Q8
*/

select
        nation,
        year,
        sum(amount) as sumprofit
from
        (
                select
                        n.name as nation,
259                     extract(year from o.orderdate) as year,
                        l.extendedprice * (1 - l.discount) - ps.supplycost * l.quantity as amount
                from
                        part p,
                        supplier s,
                        lineitem l,
                        partsupp ps,
                        orders o,
                        nation n
                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 '%floral%'
        ) as profit
group by
```

```
        nation,
        year
order by
        nation,
        year desc;
--#SET ROWS_FETCH -1
-- Q9


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 c,
        orders o,
        lineitem l,
        nation n
where
        c.custkey = o.custkey
        and l.orderkey = o.orderkey
        and o.orderdate >= date '1993-11-01'
        and o.orderdate < date '1993-11-01' + interval '3 months'
        and l.returnflag = 'R'
        and c.nationkey = n.nationkey
group by
        c.custkey,
        c.name,
        c.acctbal,
        c.phone,
        n.name,
        c.address,
        c.comment
order by
        revenue desc;
--#SET ROWS_FETCH 20
-- Q10


select
        ps.partkey,
        sum(ps.supplycost * ps.availqty) as value
from
        partsupp ps,
        supplier s,
        nation n
where
        ps.suppkey = s.suppkey
        and s.nationkey = n.nationkey
        and n.name = 'IRAQ'
group by
        ps.partkey having
                sum(ps.supplycost * ps.availqty) > (
                        select
                                sum(ps2.supplycost * ps2.availqty) * 0.0001000000
                        from
                                partsupp ps2,
                                supplier s2,
                                nation n2
                        where
                                ps2.suppkey = s2.suppkey
                                and s2.nationkey = n2.nationkey
                                and n2.name = 'IRAQ'
                )
order by
        value desc;
--#SET ROWS_FETCH -1
-- Q11
```

```
select
        l.shipmode,
        sum(case
                when o.orderpriority = '1-URGENT'
                        or o.orderpriority = '2-HIGH'
                        then 1
                else 0
        end) as highlinecount,
        sum(case
                when o.orderpriority <> '1-URGENT'
                        and o.orderpriority <> '2-HIGH'
                        then 1
                else 0
        end) as lowlinecount
from
        orders o,
        lineitem l
where
        o.orderkey = l.orderkey
        and l.shipmode in ('TRUCK', 'RAIL')
        and l.commitdate < l.receiptdate
        and l.shipdate < l.commitdate
        and l.receiptdate >= date '1996-01-01'
        and l.receiptdate < date '1996-01-01' + interval '1 year'
group by
        l.shipmode
order by
        l.shipmode;
--#SET ROWS_FETCH -1
-- Q12


select
        count,
        count(*) as custdist
from
        (
                select
                        c.custkey,
                        count(o.orderkey)
                from
                        customer c left outer join orders o on
                                c.custkey = o.custkey
                                and o.comment not like '%pending%accounts%'
                group by
                        c.custkey
        ) as orders (custkey, count)
group by
        count
order by
        custdist desc,
        count desc;
--#SET ROWS_FETCH -1
-- Q13


select
        100.00 * sum(case
                when p.type like 'PROMO%'
                        then l.extendedprice * (1 - l.discount)
                else 0
        end) / sum(l.extendedprice * (1 - l.discount)) as promorevenue
from
        lineitem l,
        part p
where
        l.partkey = p.partkey
        and l.shipdate >= date '1996-08-01'
        and l.shipdate < date '1996-08-01' + interval '1 month';
--#SET ROWS_FETCH -1
-- Q14
```

```
create view revenue0 (supplierno, totalrevenue) as
        select
                l.suppkey,
                sum(l.extendedprice * (1 - l.discount))
        from
                lineitem l
        where
                l.shipdate >= date '1996-02-01'
                and l.shipdate < date '1996-02-01' + interval '3 months'
        group by
                l.suppkey;


select
        s.suppkey,
        s.name,
        s.address,
        s.phone,
        totalrevenue
from
        supplier s,
        revenue0
where
        s.suppkey = supplierno
        and totalrevenue = (
                select
                        max(totalrevenue)
                from
                        revenue0
        )
order by
        s.suppkey;

drop view revenue0;
--#SET ROWS_FETCH -1
-- Q15


select
        p.brand,
        p.type,
        p.size,
        count(distinct ps.suppkey) as suppliercnt
from
        partsupp ps,
        part p
where
        p.partkey = ps.partkey
        and p.brand <> 'Brand#43'
        and p.type not like 'STANDARD BRUSHED%'
        and p.size in (14, 24, 40, 2, 43, 46, 6, 19)
        and ps.suppkey not in (
                select
                        s.suppkey
                from
                        supplier s
                where
                        s.comment like '%Customer%Complaints%'
        )
group by
        p.brand,
        p.type,
        p.size
order by
        suppliercnt desc,
        p.brand,
        p.type,
        p.size;
--#SET ROWS_FETCH -1
-- Q16
```

```
select
        sum(l.extendedprice) / 7.0 as avgyearly
from
        lineitem l,
        part p
where
        p.partkey = l.partkey
        and p.brand = 'Brand#31'
        and p.container = 'LG JAR'
        and l.quantity < (
                select
                        0.2 * avg(l2.quantity)
                from
                        lineitem l2
                where
                        l2.partkey = p.partkey
        );
--#SET ROWS_FETCH -1
-- Q17


select
        c.name,
        c.custkey,
        o.orderkey,
        o.orderdate,
        o.totalprice,
        sum(l.quantity)
from
        customer c,
        orders o,
        lineitem l
where
        o.orderkey in (
                select
                        l2.orderkey
                from
                        lineitem l2
                group by
                        l2.orderkey having
                                sum(l2.quantity) > 312
        )
        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;
--#SET ROWS_FETCH 100
-- Q18


select
        sum(l.extendedprice* (1 - l.discount)) as revenue
from
        lineitem l,
        part p
where
        (
                p.partkey = l.partkey
                and p.brand = 'Brand#24'
                and p.container in ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
                and l.quantity >= 10 and l.quantity <= 10 + 10
                and p.size between 1 and 5
                and l.shipmode in ('AIR', 'AIR REG')
                and l.shipinstruct = 'DELIVER IN PERSON'
        )
        or
```

```
        (
                p.partkey = l.partkey
                and p.brand = 'Brand#44'
                and p.container in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
                and l.quantity >= 14 and l.quantity <= 14 + 10
                and p.size between 1 and 10
                and l.shipmode in ('AIR', 'AIR REG')
                and l.shipinstruct = 'DELIVER IN PERSON'
        )
        or
        (
                p.partkey = l.partkey
                and p.brand = 'Brand#25'
                and p.container in ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
                and l.quantity >= 22 and l.quantity <= 22 + 10
                and p.size between 1 and 15
                and l.shipmode in ('AIR', 'AIR REG')
                and l.shipinstruct = 'DELIVER IN PERSON'
        );
--#SET ROWS_FETCH -1
-- Q19

592

select
        s.name,
        s.address
from
        supplier s,
        nation n
where
        s.suppkey in (
                select
                        ps.suppkey
                from
                        partsupp ps
                where
                        ps.partkey in (
                                select
                                        p.partkey
                                from
                                        part p
                                where
                                        p.name like 'papaya%'
                        )
                        and ps.availqty > (
                                select
                                        0.5 * sum(l.quantity)
                                from
                                        lineitem l
                                where
                                        l.partkey = ps.partkey
                                        and l.suppkey = ps.suppkey
                                        and l.shipdate >= date '1996-01-01'
                                        and l.shipdate < date '1996-01-01' + interval '1 year'
                        )
        )
        and s.nationkey = n.nationkey
        and n.name = 'MOZAMBIQUE'
order by
629     s.name;
--#SET ROWS_FETCH -1
-- Q20


select
        s.name,
        count(*) as numwait
from
        supplier s,
        lineitem l1,
        orders o,
        nation n
where
```

```
        s.suppkey = l1.suppkey
        and o.orderkey = l1.orderkey
        and o.orderstatus = 'F'
        and l1.receiptdate > l1.commitdate
        and exists (
                select
                        *
                from
                        lineitem l2
                where
                        l2.orderkey = l1.orderkey
                        and l2.suppkey <> l1.suppkey
        )
        and not exists (
                select
                        *
                from
                        lineitem l3
                where
                        l3.orderkey = l1.orderkey
                        and l3.suppkey <> l1.suppkey
                        and l3.receiptdate > l3.commitdate
        )
666     and s.nationkey = n.nationkey
        and n.name = 'ROMANIA'
group by
        s.name
order by
        numwait desc,
        s.name;
--#SET ROWS_FETCH 100
-- Q21

select
        cntrycode,
        count(*) as numcust,
        sum(acctbal) as totacctbal
from
        (
                select
                        substring(c.phone from 1 for 2) as cntrycode,
                        c.acctbal
                from
                        customer c
                where
                        substring(c.phone from 1 for 2) in
                                ('25', '11', '14', '28', '27', '16', '24')
                        and c.acctbal > (
                                select
                                        avg(c1.acctbal)
                                from
                                        customer c1
                                where
                                        c1.acctbal > 0.00
                                        and substring(c1.phone from 1 for 2) in
                                                ('25', '11', '14', '28', '27', '16', '24')
                        )
                        and not exists (
                                select
                                        *
703                             from
                                        orders o
                                where
                                        o.custkey = c.custkey
                        )
        ) as custsale
group by
        cntrycode
order by
        cntrycode;
--#SET ROWS.FETCH -1
-- Q22
```

APÊNDICE D

PLANOS DE EXECUÇÃO GERADOS ATRAVÉS DO ALGORITMO DE KRUSKAL PARA AS CONSULTAS DO TPC-H

```
Timing is on.
                                        QUERY PLAN Q1
-------------------------------------------------------------------------------------
 GroupAggregate  (cost=168246.01..168246.09 rows=1 width=138)
   ->  Sort  (cost=168246.01..168246.02 rows=1 width=138)
         Sort Key: returnflag, linestatus
         ->  Seq Scan on lineitem l  (cost=0.00..168246.00 rows=1 width=138)
               Filter: (shipdate <= '1998-09-05 00:00:00'::timestamp without time zone)
(5 rows)

Time: 16.157 ms
                                                        QUERY PLAN Q2
-------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=117617.38..117617.38 rows=1 width=220)
   Sort Key: s.acctbal, n.name, s.name, p.partkey
   ->  Nested Loop  (cost=8134.39..117617.37 rows=1 width=220)
         Join Filter: ("inner".suppkey = "outer".suppkey)
         ->  Hash Join  (cost=8131.89..117158.87 rows=1 width=45)
               Hash Cond: ("outer".partkey = "inner".partkey)
               Join Filter: ("outer".supplycost = (subplan))
               ->  Seq Scan on partsupp ps  (cost=0.00..27492.86 rows=799286 width=27)
               ->  Hash  (cost=8131.58..8131.58 rows=122 width=38)
                     ->  Seq Scan on part p  (cost=0.00..8131.58 rows=122 width=38)
                           Filter: ((size = 46) AND (("type")::text ~~ '%BRASS'::text))
               SubPlan
                 ->  Aggregate  (cost=158.87..158.88 rows=1 width=11)
                       ->  Hash Join  (cost=2.50..158.85 rows=4 width=11)
                             Hash Cond: ("outer".nationkey = "inner".nationkey)
                             ->  Nested Loop  (cost=0.00..156.21 rows=20 width=16)
                                   ->  Index Scan using partsupp_pkey on partsupp ps2  (cost=0.00..76.23 rows=20 width=1
                                         Index Cond: ($0 = partkey)
                                   ->  Index Scan using supplier_pkey on supplier s2  (cost=0.00..3.99 rows=1 width=12)
                                         Index Cond: (s2.suppkey = "outer".suppkey)
                             ->  Hash  (cost=2.49..2.49 rows=5 width=5)
                                   ->  Hash Join  (cost=1.06..2.49 rows=5 width=5)
                                         Hash Cond: ("outer".regionkey = "inner".regionkey)
                                         ->  Seq Scan on nation n2  (cost=0.00..1.25 rows=25 width=10)
                                         ->  Hash  (cost=1.06..1.06 rows=1 width=5)
                                               ->  Seq Scan on region r2  (cost=0.00..1.06 rows=1 width=5)
                                                     Filter: (name = 'AMERICA'::bpchar)
         ->  Hash Join  (cost=2.50..433.50 rows=2000 width=189)
               Hash Cond: ("outer".nationkey = "inner".nationkey)
               ->  Seq Scan on supplier s  (cost=0.00..361.00 rows=10000 width=165)
               ->  Hash  (cost=2.49..2.49 rows=5 width=34)
                     ->  Hash Join  (cost=1.06..2.49 rows=5 width=34)
                           Hash Cond: ("outer".regionkey = "inner".regionkey)
                           ->  Seq Scan on nation n  (cost=0.00..1.25 rows=25 width=39)
                           ->  Hash  (cost=1.06..1.06 rows=1 width=5)
                                 ->  Seq Scan on region r  (cost=0.00..1.06 rows=1 width=5)
                                       Filter: (name = 'AMERICA'::bpchar)
(37 rows)

Time: 16.406 ms
                                              QUERY PLAN Q3
```

```
-------------------------------------------------------------------------------------------------------
 Sort  (cost=16.12..16.12 rows=1 width=104)
   Sort Key: sum((l.extendedprice * (1::numeric - l.discount))), o.orderdate
   ->  HashAggregate  (cost=16.09..16.11 rows=1 width=104)
         ->  Nested Loop  (cost=0.00..16.08 rows=1 width=104)
               ->  Nested Loop  (cost=0.00..10.38 rows=1 width=113)
                     ->  Index Scan using lineitem_pkey on lineitem l  (cost=0.00..5.01 rows=1 width=96)
                           Filter: (shipdate > '1995-03-09'::date)
                     ->  Index Scan using orders_pkey on orders o  (cost=0.00..5.36 rows=1 width=27)
                           Index Cond: ("outer".orderkey = o.orderkey)
                           Filter: (orderdate < '1995-03-09'::date)
               ->  Index Scan using customer_pkey on customer c  (cost=0.00..5.69 rows=1 width=9)
                     Index Cond: (c.custkey = "outer".custkey)
                     Filter: (mktsegment = 'AUTOMOBILE'::bpchar)
(13 rows)

Time: 8.305 ms
                                                                    QUERY PLAN Q4
-------------------------------------------------------------------------------------------------------------------------------------
74  Sort  (cost=7580726.35..7580726.36 rows=1 width=19)
      Sort Key: orderpriority
      ->  HashAggregate  (cost=7580726.33..7580726.34 rows=1 width=19)
            ->  Seq Scan on orders o  (cost=0.00..7580563.49 rows=32567 width=19)
                  Filter: ((orderdate >= '1994-05-01'::date) AND (orderdate < '1994-08-01 00:00:00'::timestamp without time
                  SubPlan
                    ->  Index Scan using lineitem_pkey on lineitem l  (cost=0.00..5.02 rows=1 width=333)
                          Index Cond: (orderkey = $0)
                          Filter: (commitdate < receiptdate)
(9 rows)

Time: 3.205 ms
                                                                      QUERY PLAN Q5
-------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=22.80..22.81 rows=1 width=93)
   Sort Key: sum((l.extendedprice * (1::numeric - l.discount)))
   ->  HashAggregate  (cost=22.77..22.79 rows=1 width=93)
         ->  Nested Loop  (cost=0.00..22.77 rows=1 width=93)
               Join Filter: ("inner".nationkey = "outer".nationkey)
               ->  Nested Loop  (cost=0.00..17.07 rows=1 width=112)
                     Join Filter: ("outer".nationkey = "inner".nationkey)
                     ->  Nested Loop  (cost=0.00..14.38 rows=1 width=78)
                           ->  Nested Loop  (cost=0.00..9.01 rows=1 width=101)
                                 ->  Index Scan using lineitem_pkey on lineitem l  (cost=0.00..5.01 rows=1 width=128)
                                 ->  Index Scan using supplier_pkey on supplier s  (cost=0.00..3.99 rows=1 width=12)
                                       Index Cond: ("outer".suppkey = s.suppkey)
                           ->  Index Scan using orders_pkey on orders o  (cost=0.00..5.36 rows=1 width=19)
                                 Index Cond: ("outer".orderkey = o.orderkey)
                                 Filter: ((orderdate >= '1996-01-01'::date) AND (orderdate < '1997-01-01 00:00:00'::time
                     ->  Nested Loop  (cost=0.00..2.62 rows=5 width=34)
                           Join Filter: ("inner".regionkey = "outer".regionkey)
                           ->  Seq Scan on region r  (cost=0.00..1.06 rows=1 width=5)
                                 Filter: (name = 'AFRICA'::bpchar)
                           ->  Seq Scan on nation n  (cost=0.00..1.25 rows=25 width=39)
               ->  Index Scan using customer_pkey on customer c  (cost=0.00..5.68 rows=1 width=14)
                     Index Cond: (c.custkey = "outer".custkey)
(22 rows)
111
Time: 5.989 ms
                                                                                  QUERY PLAN Q6
-------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=168246.01..168246.02 rows=1 width=64)
   ->  Seq Scan on lineitem l  (cost=0.00..168246.00 rows=1 width=64)
         Filter: ((shipdate >= '1996-01-01'::date) AND (shipdate < '1997-01-01 00:00:00'::timestamp without time zone) A
(3 rows)

Time: 2.470 ms
                                                                                QUERY PLAN Q7
-------------------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=41.40..41.43 rows=1 width=126)
   ->  Sort  (cost=41.40..41.40 rows=1 width=126)
         Sort Key: n1.name, n2.name, date_part('year'::text, (l.shipdate)::timestamp without time zone)
         ->  Nested Loop  (cost=1.27..41.39 rows=1 width=126)
               Join Filter: ("inner".nationkey = "outer".nationkey)
```

```
              ->  Nested Loop  (cost=1.27..35.68 rows=1 width=140)
                    Join Filter: ("outer".nationkey = "inner".nationkey)
                    ->  Nested Loop  (cost=0.00..14.38 rows=1 width=82)
                          ->  Nested Loop  (cost=0.00..9.02 rows=1 width=105)
                                ->  Index Scan using lineitem_pkey on lineitem l  (cost=0.00..5.02 rows=1 width=132)
                                      Filter: ((shipdate >= '1995-01-01'::date) AND (shipdate <= '1996-12-31'::date))
                                ->  Index Scan using supplier_pkey on supplier s  (cost=0.00..3.99 rows=1 width=12)
                                      Index Cond: (s.suppkey = "outer".suppkey)
                          ->  Index Scan using orders_pkey on orders o  (cost=0.00..5.35 rows=1 width=19)
                                Index Cond: (o.orderkey = "outer".orderkey)
                    ->  Nested Loop  (cost=1.27..21.27 rows=2 width=68)
                          Join Filter: ((("outer".name = 'JAPAN'::bpchar) AND ("inner".name = 'IRAN'::bpchar)) OR (("ou
                          ->  Seq Scan on nation n1  (cost=0.00..1.25 rows=25 width=34)
                          ->  Materialize  (cost=1.27..1.52 rows=25 width=34)
                                ->  Seq Scan on nation n2  (cost=0.00..1.25 rows=25 width=34)
              ->  Index Scan using customer_pkey on customer c  (cost=0.00..5.68 rows=1 width=14)
                    Index Cond: (c.custkey = "outer".custkey)
(22 rows)

Time: 7.892 ms
                                                          QUERY PLAN Q9
-----------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=27.30..27.34 rows=1 width=140)
   ->  Sort  (cost=27.30..27.31 rows=1 width=140)
         Sort Key: n.name, date_part('year'::text, (o.orderdate)::timestamp without time zone)
         ->  Nested Loop  (cost=0.00..27.29 rows=1 width=140)
               Join Filter: ("outer".nationkey = "inner".nationkey)
               ->  Nested Loop  (cost=0.00..25.72 rows=1 width=116)
                     ->  Nested Loop  (cost=0.00..19.92 rows=1 width=185)
                           ->  Nested Loop  (cost=0.00..14.38 rows=1 width=176)
                                 ->  Nested Loop  (cost=0.00..9.01 rows=1 width=204)
                                       ->  Index Scan using lineitem_pkey on lineitem l  (cost=0.00..5.01 rows=1 width=1
                                       ->  Index Scan using supplier_pkey on supplier s  (cost=0.00..3.99 rows=1 width=1
                                             Index Cond: (s.suppkey = "outer".suppkey)
                                 ->  Index Scan using orders_pkey on orders o  (cost=0.00..5.35 rows=1 width=14)
                                       Index Cond: (o.orderkey = "outer".orderkey)
                           ->  Index Scan using part_pkey on part p  (cost=0.00..5.53 rows=1 width=9)
                                 Index Cond: (p.partkey = "outer".partkey)
                                 Filter: ((name)::text ~~ '%floral%'::text)
                     ->  Index Scan using partsupp_pkey on partsupp ps  (cost=0.00..5.79 rows=1 width=27)
                           Index Cond: ((ps.partkey = "outer".partkey) AND (ps.suppkey = "outer".suppkey))
               ->  Seq Scan on nation n  (cost=0.00..1.25 rows=25 width=34)
(20 rows)

Time: 6.586 ms
                                                                        QUERY PLAN Q10
-----------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=8838.01..8838.02 rows=1 width=259)
   Sort Key: sum((l.extendedprice * (1::numeric - l.discount)))
   ->  GroupAggregate  (cost=8837.96..8838.00 rows=1 width=259)
         ->  Sort  (cost=8837.96..8837.96 rows=1 width=259)
               Sort Key: c.custkey, c.name, c.acctbal, c.phone, n.name, c.address, c."comment"
               ->  Hash Join  (cost=11.70..8837.95 rows=1 width=259)
                     Hash Cond: ("outer".custkey = "inner".custkey)
                     ->  Hash Join  (cost=1.31..8077.51 rows=150008 width=195)
                           Hash Cond: ("outer".nationkey = "inner".nationkey)
                           ->  Seq Scan on customer c  (cost=0.00..5826.08 rows=150008 width=171)
                           ->  Hash  (cost=1.25..1.25 rows=25 width=34)
                                 ->  Seq Scan on nation n  (cost=0.00..1.25 rows=25 width=34)
                     ->  Hash  (cost=10.38..10.38 rows=1 width=73)
                           ->  Nested Loop  (cost=0.00..10.38 rows=1 width=73)
                                 ->  Index Scan using lineitem_pkey on lineitem l  (cost=0.00..5.01 rows=1 width=96)
                                       Filter: (returnflag = 'R'::bpchar)
                                 ->  Index Scan using orders_pkey on orders o  (cost=0.00..5.36 rows=1 width=19)
                                       Index Cond: ("outer".orderkey = o.orderkey)
                                       Filter: ((orderdate >= '1993-11-01'::date) AND (orderdate < '1994-02-01 00:00:00'
(19 rows)

Time: 5.592 ms
                                             QUERY PLAN Q11
----------------------------------------------------------------------------------------------------
 Sort  (cost=71281.31..71360.49 rows=31671 width=24)
   Sort Key: sum((ps.supplycost * (ps.availqty)::numeric))
```

```
   InitPlan
     ->  Aggregate  (cost=32302.49..32302.51 rows=1 width=15)
           ->  Hash Join  (cost=417.31..32223.31 rows=31671 width=15)
                 Hash Cond: ("outer".suppkey = "inner".suppkey)
                 ->  Seq Scan on partsupp ps2  (cost=0.00..27492.86 rows=799286 width=22)
                 ->  Hash  (cost=416.31..416.31 rows=400 width=7)
                       ->  Hash Join  (cost=1.31..416.31 rows=400 width=7)
                             Hash Cond: ("outer".nationkey = "inner".nationkey)
                             ->  Seq Scan on supplier s2  (cost=0.00..361.00 rows=10000 width=12)
                             ->  Hash  (cost=1.31..1.31 rows=1 width=5)
                                   ->  Seq Scan on nation n2  (cost=0.00..1.31 rows=1 width=5)
                                         Filter: (name = 'IRAQ'::bpchar)
   ->  GroupAggregate  (cost=35007.22..36194.89 rows=31671 width=24)
         Filter: (sum((supplycost * (availqty)::numeric)) > $0)
         ->  Sort  (cost=35007.22..35086.40 rows=31671 width=24)
               Sort Key: ps.partkey
               ->  Hash Join  (cost=417.31..32223.31 rows=31671 width=24)
                     Hash Cond: ("outer".suppkey = "inner".suppkey)
                     ->  Seq Scan on partsupp ps  (cost=0.00..27492.86 rows=799286 width=31)
                     ->  Hash  (cost=416.31..416.31 rows=400 width=7)
                           ->  Hash Join  (cost=1.31..416.31 rows=400 width=7)
                                 Hash Cond: ("outer".nationkey = "inner".nationkey)
                                 ->  Seq Scan on supplier s  (cost=0.00..361.00 rows=10000 width=12)
                                 ->  Hash  (cost=1.31..1.31 rows=1 width=5)
                                       ->  Seq Scan on nation n  (cost=0.00..1.31 rows=1 width=5)
                                             Filter: (name = 'IRAQ'::bpchar)
(28 rows)

Time: 5.069 ms
                                                                                                                        QUE
------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=10.40..10.44 rows=1 width=33)
   ->  Sort  (cost=10.40..10.40 rows=1 width=33)
         Sort Key: l.shipmode
         ->  Nested Loop  (cost=0.00..10.39 rows=1 width=33)
               ->  Index Scan using lineitem_pkey on lineitem l  (cost=0.00..5.02 rows=1 width=46)
                     Filter: ((shipmode = ANY ('{TRUCK,RAIL}'::bpchar[])) AND (commitdate < receiptdate) AND (shipdate <
               ->  Index Scan using orders_pkey on orders o  (cost=0.00..5.35 rows=1 width=29)
                     Index Cond: (o.orderkey = "outer".orderkey)
(8 rows)

Time: 27.355 ms
                                           QUERY PLAN Q13
------------------------------------------------------------------------------------------------
 Sort  (cost=346711.42..346711.92 rows=200 width=8)
   Sort Key: count(*), orders.count
   ->  HashAggregate  (cost=346701.28..346703.78 rows=200 width=8)
         ->  GroupAggregate  (cost=311846.89..344451.16 rows=150008 width=19)
               ->  Merge Left Join  (cost=311846.89..335080.68 rows=1499075 width=19)
                     Merge Cond: ("outer".custkey = "inner".custkey)
                     ->  Sort  (cost=20779.53..21154.55 rows=150008 width=9)
                           Sort Key: c.custkey
                           ->  Seq Scan on customer c  (cost=0.00..5826.08 rows=150008 width=9)
                     ->  Sort  (cost=291067.36..294815.05 rows=1499075 width=19)
                           Sort Key: o.custkey
                           ->  Seq Scan on orders o  (cost=0.00..52454.94 rows=1499075 width=19)
                                 Filter: (("comment")::text !~~ '%pending%accounts%'::text)
(13 rows)

Time: 4.132 ms
                                                          QUERY PLAN Q14
------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=168251.55..168251.58 rows=1 width=88)
   ->  Nested Loop  (cost=0.00..168251.54 rows=1 width=88)
         ->  Seq Scan on lineitem l  (cost=0.00..168246.00 rows=1 width=96)
               Filter: ((shipdate >= '1996-08-01'::date) AND (shipdate < '1996-09-01 00:00:00'::timestamp without time z
         ->  Index Scan using part_pkey on part p  (cost=0.00..5.53 rows=1 width=33)
               Index Cond: ("outer".partkey = p.partkey)
(6 rows)

Time: 2.826 ms
CREATE VIEW
Time: 264.835 ms
```

```
                                                                QUERY PLAN Q15
-------------------------------------------------------------------------------------------------------------------------------------------
 Merge Join  (cost=336492.10..337191.77 rows=1 width=115)
   Merge Cond: ("outer".suppkey = "inner".supplierno)
   InitPlan
     ->  Aggregate  (cost=168246.04..168246.05 rows=1 width=32)
           ->  HashAggregate  (cost=168246.01..168246.02 rows=1 width=96)
                 ->  Seq Scan on lineitem l  (cost=0.00..168246.00 rows=1 width=96)
                       Filter: ((shipdate >= '1996-02-01'::date) AND (shipdate < '1996-05-01 00:00:00'::timestamp withou
   ->  Index Scan using supplier_pkey on supplier s  (cost=0.00..674.65 rows=10000 width=83)
   ->  Sort  (cost=168246.06..168246.06 rows=1 width=64)
         Sort Key: revenue0.supplierno
         ->  Subquery Scan revenue0  (cost=168246.01..168246.05 rows=1 width=64)
               ->  HashAggregate  (cost=168246.01..168246.04 rows=1 width=96)
                     Filter: (sum((extendedprice * (1::numeric - discount))) = $0)
                     ->  Seq Scan on lineitem l  (cost=0.00..168246.00 rows=1 width=96)
                           Filter: ((shipdate >= '1996-02-01'::date) AND (shipdate < '1996-05-01 00:00:00'::timestamp wi
(15 rows)

Time: 4.212 ms
DROP VIEW
Time: 54.559 ms
                                                                QUERY PLAN Q16
-------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=56359.68..56366.07 rows=2553 width=49)
   Sort Key: count(DISTINCT ps.suppkey), p.brand, p."type", p.size
   ->  GroupAggregate  (cost=55545.57..56215.21 rows=2553 width=49)
         ->  Sort  (cost=55545.57..55673.12 rows=51018 width=49)
               Sort Key: p.brand, p."type", p.size
               ->  Hash Join  (cost=10831.12..50371.91 rows=51018 width=49)
                     Hash Cond: ("outer".partkey = "inner".partkey)
                     ->  Seq Scan on partsupp ps  (cost=386.00..27878.86 rows=399643 width=16)
                           Filter: (NOT (hashed subplan))
                           SubPlan
                             ->  Seq Scan on supplier s  (cost=0.00..386.00 rows=1 width=7)
                                   Filter: (("comment")::text ~~ '%Customer%Complaints%'::text)
                     ->  Hash  (cost=10131.30..10131.30 rows=25528 width=51)
                           ->  Seq Scan on part p  (cost=0.00..10131.30 rows=25528 width=51)
                                 Filter: ((brand <> 'Brand#43'::bpchar) AND (("type")::text !~~ 'STANDARD BRUSHED%'::tex
(15 rows)

Time: 3.375 ms
                                      QUERY PLAN Q17
-----------------------------------------------------------------------------------
 Aggregate  (cost=336497.57..336497.58 rows=1 width=32)
   ->  Nested Loop  (cost=0.00..336497.57 rows=1 width=32)
         Join Filter: ("outer".quantity < (subplan))
         ->  Seq Scan on lineitem l  (cost=0.00..168246.00 rows=1 width=96)
         ->  Index Scan using part_pkey on part p  (cost=0.00..5.53 rows=1 width=9)
               Index Cond: (p.partkey = "outer".partkey)
               Filter: ((brand = 'Brand#31'::bpchar) AND (container = 'LG JAR'::bpchar))
         SubPlan
           ->  Aggregate  (cost=168246.01..168246.02 rows=1 width=32)
                 ->  Seq Scan on lineitem l2  (cost=0.00..168246.00 rows=1 width=32)
                       Filter: (partkey = $0)
(11 rows)

Time: 2.727 ms
                                                  QUERY PLAN Q18
-------------------------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=21.14..21.17 rows=1 width=90)
   ->  Sort  (cost=21.14..21.14 rows=1 width=90)
         Sort Key: o.totalprice, o.orderdate, c.name, c.custkey, o.orderkey
         ->  Nested Loop  (cost=0.00..21.13 rows=1 width=90)
               ->  Nested Loop  (cost=0.00..15.43 rows=1 width=68)
                     ->  Nested Loop  (cost=0.00..10.06 rows=1 width=96)
                           Join Filter: ("outer".orderkey = "inner".orderkey)
                           ->  Index Scan using lineitem_pkey on lineitem l  (cost=0.00..5.01 rows=1 width=64)
                           ->  GroupAggregate  (cost=0.00..5.03 rows=1 width=64)
                                 Filter: (sum(quantity) > 312::numeric)
                                 ->  Index Scan using lineitem_pkey on lineitem l2  (cost=0.00..5.01 rows=1 width=64)
                     ->  Index Scan using orders_pkey on orders o  (cost=0.00..5.35 rows=1 width=36)
                           Index Cond: (o.orderkey = "outer".orderkey)
```

```
               ->  Index Scan using customer_pkey on customer c  (cost=0.00..5.68 rows=1 width=31)
                     Index Cond: (c.custkey = "outer".custkey)
(15 rows)

Time: 4.164 ms

-------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=168251.59..168251.61 rows=1 width=64)
   ->  Nested Loop  (cost=0.00..168251.59 rows=1 width=64)
         Join Filter: ((("inner".brand = 'Brand#24'::bpchar) AND ("inner".container = ANY ('{"SM CASE","SM BOX","SM PACK
         ->  Seq Scan on lineitem l  (cost=0.00..168246.00 rows=1 width=128)
               Filter: ((shipmode = ANY ('{AIR,"AIR REG"}'::bpchar[])) AND (shipinstruct = 'DELIVER IN PERSON'::bpchar))
         ->  Index Scan using part_pkey on part p  (cost=0.00..5.53 rows=1 width=41)
               Index Cond: (p.partkey = "outer".partkey)
               Filter: (size >= 1)
(8 rows)

Time: 6.213 ms
                                QUERY PLAN Q20
-------------------------------------------------------------------------------
 Sort  (cost=433.60..434.60 rows=400 width=64)
   Sort Key: s.name
   ->  Hash Join  (cost=1.31..416.31 rows=400 width=64)
         Hash Cond: ("outer".nationkey = "inner".nationkey)
         ->  Seq Scan on supplier s  (cost=0.00..361.00 rows=10000 width=69)
         ->  Hash  (cost=1.31..1.31 rows=1 width=5)
               ->  Seq Scan on nation n  (cost=0.00..1.31 rows=1 width=5)
                     Filter: (name = 'MOZAMBIQUE'::bpchar)
(8 rows)

Time: 3.744 ms
                                                    QUERY PLAN 21
-------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=25.77..25.77 rows=1 width=29)
   Sort Key: count(*), s.name
   ->  HashAggregate  (cost=25.75..25.76 rows=1 width=29)
         ->  Nested Loop  (cost=0.00..25.74 rows=1 width=29)
               Join Filter: ("outer".nationkey = "inner".nationkey)
               ->  Nested Loop  (cost=0.00..24.42 rows=1 width=34)
                     ->  Nested Loop  (cost=0.00..19.05 rows=1 width=66)
                           ->  Index Scan using lineitem_pkey on lineitem l1  (cost=0.00..15.05 rows=1 width=64)
                                 Filter: ((receiptdate > commitdate) AND (subplan) AND (NOT (subplan)))
                                 SubPlan
                                   ->  Index Scan using lineitem_pkey on lineitem l3  (cost=0.00..5.02 rows=1 width=333)
                                         Index Cond: (orderkey = $0)
                                         Filter: ((suppkey <> $1) AND (receiptdate > commitdate))
                                   ->  Index Scan using lineitem_pkey on lineitem l2  (cost=0.00..5.02 rows=1 width=333)
                                         Index Cond: (orderkey = $0)
                                         Filter: (suppkey <> $1)
                           ->  Index Scan using supplier_pkey on supplier s  (cost=0.00..3.99 rows=1 width=41)
                                 Index Cond: (s.suppkey = "outer".suppkey)
                     ->  Index Scan using orders_pkey on orders o  (cost=0.00..5.36 rows=1 width=10)
                           Index Cond: (o.orderkey = "outer".orderkey)
                           Filter: (orderstatus = 'F'::bpchar)
               ->  Seq Scan on nation n  (cost=0.00..1.31 rows=1 width=5)
                     Filter: (name = 'ROMANIA'::bpchar)
(23 rows)

Time: 5.146 ms

                                                          QUERY PLAN Q22
-------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=605298144.42..605298170.28 rows=862 width=30)
   InitPlan
     ->  Aggregate  (cost=8275.55..8275.56 rows=1 width=11)
           ->  Seq Scan on customer c1  (cost=0.00..8263.71 rows=4736 width=11)
                 Filter: ((acctbal > 0.00) AND ("substring"((phone)::text, 1, 2) = ANY ('{25,11,14,28,27,16,24}'::text[]
   ->  Sort  (cost=605289868.86..605289871.01 rows=862 width=30)
         Sort Key: "substring"((c.phone)::text, 1, 2)
         ->  Seq Scan on customer c  (cost=0.00..605289826.83 rows=862 width=30)
               Filter: (("substring"((phone)::text, 1, 2) = ANY ('{25,11,14,28,27,16,24}'::text[])) AND (acctbal > $0) A
               SubPlan
                 ->  Seq Scan on orders o  (cost=0.00..52454.94 rows=13 width=135)
```

```
                        Filter: (custkey = $1)
(12 rows)

Time: 3.052 ms
```

APÊNDICE E

PLANOS DE EXECUÇÃO GERADOS ATRAVÉS DO ALGORITMO PADRÃO DO POSTGRESQL PARA AS CONSULTAS DO TPC-H

```
Timing is on.
                                    QUERY PLAN Q1
-------------------------------------------------------------------------------------
 GroupAggregate  (cost=168246.01..168246.09 rows=1 width=138)
   ->  Sort  (cost=168246.01..168246.02 rows=1 width=138)
         Sort Key: returnflag, linestatus
         ->  Seq Scan on lineitem l  (cost=0.00..168246.00 rows=1 width=138)
               Filter: (shipdate <= '1998-09-05 00:00:00'::timestamp without time zone)
(5 rows)

Time: 92.173 ms
                                                    QUERY PLAN Q2
-----------------------------------------------------------------------------------------------------------------------
 Sort  (cost=57847.19..57847.19 rows=1 width=220)
   Sort Key: s.acctbal, n.name, s.name, p.partkey
   ->  Hash Join  (cost=8570.39..57847.18 rows=1 width=220)
         Hash Cond: ("outer".partkey = "inner".partkey)
         Join Filter: ("outer".supplycost = (subplan))
         ->  Hash Join  (cost=438.50..33511.32 rows=158353 width=202)
               Hash Cond: ("outer".suppkey = "inner".suppkey)
               ->  Seq Scan on partsupp ps  (cost=0.00..27492.86 rows=799286 width=27)
               ->  Hash  (cost=433.50..433.50 rows=2000 width=189)
                     ->  Hash Join  (cost=2.50..433.50 rows=2000 width=189)
                           Hash Cond: ("outer".nationkey = "inner".nationkey)
                           ->  Seq Scan on supplier s  (cost=0.00..361.00 rows=10000 width=165)
                           ->  Hash  (cost=2.49..2.49 rows=5 width=34)
                                 ->  Hash Join  (cost=1.06..2.49 rows=5 width=34)
                                       Hash Cond: ("outer".regionkey = "inner".regionkey)
                                       ->  Seq Scan on nation n  (cost=0.00..1.25 rows=25 width=39)
                                       ->  Hash  (cost=1.06..1.06 rows=1 width=5)
                                             ->  Seq Scan on region r  (cost=0.00..1.06 rows=1 width=5)
                                                   Filter: (name = 'AMERICA'::bpchar)
         ->  Hash  (cost=8131.58..8131.58 rows=122 width=38)
               ->  Seq Scan on part p  (cost=0.00..8131.58 rows=122 width=38)
                     Filter: ((size = 46) AND (("type")::text ~~ '%BRASS'::text))
         SubPlan
37         ->  Aggregate  (cost=158.87..158.88 rows=1 width=11)
               ->  Hash Join  (cost=2.50..158.85 rows=4 width=11)
                     Hash Cond: ("outer".nationkey = "inner".nationkey)
                     ->  Nested Loop  (cost=0.00..156.21 rows=20 width=16)
                           ->  Index Scan using partsupp_pkey on partsupp ps2  (cost=0.00..76.23 rows=20 width=18)
                                 Index Cond: ($0 = partkey)
                           ->  Index Scan using supplier_pkey on supplier s2  (cost=0.00..3.99 rows=1 width=12)
                                 Index Cond: (s2.suppkey = "outer".suppkey)
                     ->  Hash  (cost=2.49..2.49 rows=5 width=5)
                           ->  Hash Join  (cost=1.06..2.49 rows=5 width=5)
                                 Hash Cond: ("outer".regionkey = "inner".regionkey)
                                 ->  Seq Scan on nation n2  (cost=0.00..1.25 rows=25 width=10)
                                 ->  Hash  (cost=1.06..1.06 rows=1 width=5)
                                       ->  Seq Scan on region r2  (cost=0.00..1.06 rows=1 width=5)
                                             Filter: (name = 'AMERICA'::bpchar)
(38 rows)

Time: 82.028 ms
```

QUERY PLAN Q3

```
 Sort  (cost=16.12..16.12 rows=1 width=104)
   Sort Key: sum((l.extendedprice * (1::numeric - l.discount))), o.orderdate
   ->  HashAggregate  (cost=16.09..16.11 rows=1 width=104)
         ->  Nested Loop  (cost=0.00..16.08 rows=1 width=104)
               ->  Nested Loop  (cost=0.00..10.38 rows=1 width=113)
                     ->  Index Scan using lineitem_pkey on lineitem l  (cost=0.00..5.01 rows=1 width=96)
                           Filter: (shipdate > '1995-03-09'::date)
                     ->  Index Scan using orders_pkey on orders o  (cost=0.00..5.36 rows=1 width=27)
                           Index Cond: ("outer".orderkey = o.orderkey)
                           Filter: (orderdate < '1995-03-09'::date)
               ->  Index Scan using customer_pkey on customer c  (cost=0.00..5.69 rows=1 width=9)
                     Index Cond: (c.custkey = "outer".custkey)
                     Filter: (mktsegment = 'AUTOMOBILE'::bpchar)
(13 rows)

Time: 8.137 ms
```

QUERY PLAN Q4

```
 Sort  (cost=7580726.35..7580726.36 rows=1 width=19)
   Sort Key: orderpriority
   ->  HashAggregate  (cost=7580726.33..7580726.34 rows=1 width=19)
         ->  Seq Scan on orders o  (cost=0.00..7580563.49 rows=32567 width=19)
               Filter: ((orderdate >= '1994-05-01'::date) AND (orderdate < '1994-08-01 00:00:00'::timestamp without time
               SubPlan
                 ->  Index Scan using lineitem_pkey on lineitem l  (cost=0.00..5.02 rows=1 width=333)
                       Index Cond: (orderkey = $0)
                       Filter: (commitdate < receiptdate)
(9 rows)

Time: 3.183 ms
```

QUERY PLAN Q5

```
 Sort  (cost=22.75..22.76 rows=1 width=93)
   Sort Key: sum((l.extendedprice * (1::numeric - l.discount)))
   ->  HashAggregate  (cost=22.72..22.74 rows=1 width=93)
         ->  Nested Loop  (cost=0.00..22.72 rows=1 width=93)
               Join Filter: ("outer".nationkey = "inner".nationkey)
               ->  Nested Loop  (cost=0.00..18.72 rows=1 width=135)
                     Join Filter: ("outer".regionkey = "inner".regionkey)
                     ->  Nested Loop  (cost=0.00..17.64 rows=1 width=140)
                           Join Filter: ("inner".nationkey = "outer".nationkey)
                           ->  Nested Loop  (cost=0.00..16.08 rows=1 width=101)
                                 ->  Nested Loop  (cost=0.00..10.38 rows=1 width=105)
                                       ->  Index Scan using lineitem_pkey on lineitem l  (cost=0.00..5.01 rows=1 width=1
                                       ->  Index Scan using orders_pkey on orders o  (cost=0.00..5.36 rows=1 width=19)
                                             Index Cond: ("outer".orderkey = o.orderkey)
                                             Filter: ((orderdate >= '1996-01-01'::date) AND (orderdate < '1997-01-01 00:
                                 ->  Index Scan using customer_pkey on customer c  (cost=0.00..5.68 rows=1 width=14)
                                       Index Cond: (c.custkey = "outer".custkey)
                           ->  Seq Scan on nation n  (cost=0.00..1.25 rows=25 width=39)
                     ->  Seq Scan on region r  (cost=0.00..1.06 rows=1 width=5)
                           Filter: (name = 'AFRICA'::bpchar)
               ->  Index Scan using supplier_pkey on supplier s  (cost=0.00..3.99 rows=1 width=12)
                     Index Cond: ("outer".suppkey = s.suppkey)
(22 rows)

Time: 12.014 ms
```

QUERY PLAN Q6

```
 Aggregate  (cost=168246.01..168246.02 rows=1 width=64)
   ->  Seq Scan on lineitem l  (cost=0.00..168246.00 rows=1 width=64)
         Filter: ((shipdate >= '1996-01-01'::date) AND (shipdate < '1997-01-01 00:00:00'::timestamp without time zone) A
(3 rows)

Time: 2.502 ms
```

QUERY PLAN Q7

```
 GroupAggregate  (cost=23.05..23.09 rows=1 width=126)
   ->  Sort  (cost=23.05..23.06 rows=1 width=126)
         Sort Key: n1.name, n2.name, date_part('year'::text, (l.shipdate)::timestamp without time zone)
         ->  Hash Join  (cost=21.64..23.04 rows=1 width=126)
```

```
                Hash Cond: ("outer".nationkey = "inner".nationkey)
                Join Filter: ((("inner".name = 'JAPAN'::bpchar) AND ("outer".name = 'IRAN'::bpchar)) OR (("inner".name =
                ->  Seq Scan on nation n2  (cost=0.00..1.25 rows=25 width=34)
                ->  Hash  (cost=21.64..21.64 rows=1 width=102)
                      ->  Nested Loop  (cost=0.00..21.64 rows=1 width=102)
                            ->  Nested Loop  (cost=0.00..15.94 rows=1 width=106)
                                  ->  Nested Loop  (cost=0.00..10.58 rows=1 width=129)
                                        Join Filter: ("outer".nationkey = "inner".nationkey)
                                        ->  Nested Loop  (cost=0.00..9.02 rows=1 width=105)
                                              ->  Index Scan using lineitem_pkey on lineitem l  (cost=0.00..5.02 rows=1 w
                                                    Filter: ((shipdate >= '1995-01-01'::date) AND (shipdate <= '1996-12-3
                                              ->  Index Scan using supplier_pkey on supplier s  (cost=0.00..3.99 rows=1 w
                                                    Index Cond: (s.suppkey = "outer".suppkey)
                                        ->  Seq Scan on nation n1  (cost=0.00..1.25 rows=25 width=34)
                                  ->  Index Scan using orders_pkey on orders o  (cost=0.00..5.35 rows=1 width=19)
                                        Index Cond: (o.orderkey = "outer".orderkey)
                            ->  Index Scan using customer_pkey on customer c  (cost=0.00..5.68 rows=1 width=14)
                                  Index Cond: (c.custkey = "outer".custkey)
(22 rows)

148 Time: 11.760 ms
                                                          QUERY PLAN Q9
-------------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=27.30..27.34 rows=1 width=140)
   ->  Sort  (cost=27.30..27.31 rows=1 width=140)
         Sort Key: n.name, date_part('year'::text, (o.orderdate)::timestamp without time zone)
         ->  Nested Loop  (cost=0.00..27.29 rows=1 width=140)
               Join Filter: ("outer".nationkey = "inner".nationkey)
               ->  Nested Loop  (cost=0.00..25.72 rows=1 width=116)
                     ->  Nested Loop  (cost=0.00..21.72 rows=1 width=150)
                           ->  Nested Loop  (cost=0.00..16.36 rows=1 width=178)
                                 ->  Nested Loop  (cost=0.00..10.55 rows=1 width=201)
                                       ->  Index Scan using lineitem_pkey on lineitem l  (cost=0.00..5.01 rows=1 width=1
                                       ->  Index Scan using part_pkey on part p  (cost=0.00..5.53 rows=1 width=9)
                                             Index Cond: (p.partkey = "outer".partkey)
                                             Filter: ((name)::text ~~ '%floral%'::text)
                                 ->  Index Scan using partsupp_pkey on partsupp ps  (cost=0.00..5.79 rows=1 width=27)
                                       Index Cond: ((ps.partkey = "outer".partkey) AND (ps.suppkey = "outer".suppkey))
                           ->  Index Scan using orders_pkey on orders o  (cost=0.00..5.35 rows=1 width=14)
                                 Index Cond: (o.orderkey = "outer".orderkey)
                     ->  Index Scan using supplier_pkey on supplier s  (cost=0.00..3.99 rows=1 width=12)
                           Index Cond: (s.suppkey = "outer".suppkey)
               ->  Seq Scan on nation n  (cost=0.00..1.25 rows=25 width=34)
(20 rows)

Time: 11.918 ms
                                                                    QUERY PLAN Q10
-------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=17.53..17.54 rows=1 width=259)
   Sort Key: sum((l.extendedprice * (1::numeric - l.discount)))
   ->  GroupAggregate  (cost=17.48..17.52 rows=1 width=259)
         ->  Sort  (cost=17.48..17.48 rows=1 width=259)
               Sort Key: c.custkey, c.name, c.acctbal, c.phone, n.name, c.address, c."comment"
               ->  Hash Join  (cost=16.08..17.47 rows=1 width=259)
                     Hash Cond: ("outer".nationkey = "inner".nationkey)
                     ->  Seq Scan on nation n  (cost=0.00..1.25 rows=25 width=34)
                     ->  Hash  (cost=16.08..16.08 rows=1 width=235)
185                        ->  Nested Loop  (cost=0.00..16.08 rows=1 width=235)
                                 ->  Nested Loop  (cost=0.00..10.38 rows=1 width=73)
                                       ->  Index Scan using lineitem_pkey on lineitem l  (cost=0.00..5.01 rows=1 width=9
                                             Filter: (returnflag = 'R'::bpchar)
                                       ->  Index Scan using orders_pkey on orders o  (cost=0.00..5.36 rows=1 width=19)
                                             Index Cond: ("outer".orderkey = o.orderkey)
                                             Filter: ((orderdate >= '1993-11-01'::date) AND (orderdate < '1994-02-01 00:
                                 ->  Index Scan using customer_pkey on customer c  (cost=0.00..5.68 rows=1 width=171)
                                       Index Cond: (c.custkey = "outer".custkey)
(18 rows)

Time: 5.961 ms
                                              QUERY PLAN Q11
-----------------------------------------------------------------------------------------------
 Sort  (cost=71281.31..71360.49 rows=31671 width=24)
   Sort Key: sum((ps.supplycost * (ps.availqty)::numeric))
```

```
   InitPlan
     ->  Aggregate  (cost=32302.49..32302.51 rows=1 width=15)
           ->  Hash Join  (cost=417.31..32223.31 rows=31671 width=15)
                 Hash Cond: ("outer".suppkey = "inner".suppkey)
                 ->  Seq Scan on partsupp ps2  (cost=0.00..27492.86 rows=799286 width=22)
                 ->  Hash  (cost=416.31..416.31 rows=400 width=7)
                       ->  Hash Join  (cost=1.31..416.31 rows=400 width=7)
                             Hash Cond: ("outer".nationkey = "inner".nationkey)
                             ->  Seq Scan on supplier s2  (cost=0.00..361.00 rows=10000 width=12)
                             ->  Hash  (cost=1.31..1.31 rows=1 width=5)
                                   ->  Seq Scan on nation n2  (cost=0.00..1.31 rows=1 width=5)
                                         Filter: (name = 'IRAQ'::bpchar)
   ->  GroupAggregate  (cost=35007.22..36194.89 rows=31671 width=24)
         Filter: (sum((supplycost * (availqty)::numeric)) > $0)
         ->  Sort  (cost=35007.22..35086.40 rows=31671 width=24)
               Sort Key: ps.partkey
               ->  Hash Join  (cost=417.31..32223.31 rows=31671 width=24)
                     Hash Cond: ("outer".suppkey = "inner".suppkey)
                     ->  Seq Scan on partsupp ps  (cost=0.00..27492.86 rows=799286 width=31)
                     ->  Hash  (cost=416.31..416.31 rows=400 width=7)
                           ->  Hash Join  (cost=1.31..416.31 rows=400 width=7)
                                 Hash Cond: ("outer".nationkey = "inner".nationkey)
                                 ->  Seq Scan on supplier s  (cost=0.00..361.00 rows=10000 width=12)
                                 ->  Hash  (cost=1.31..1.31 rows=1 width=5)
                                       ->  Seq Scan on nation n  (cost=0.00..1.31 rows=1 width=5)
                                             Filter: (name = 'IRAQ'::bpchar)
(28 rows)

Time: 4.664 ms
                                                                                                                        QUE
----------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=10.40..10.44 rows=1 width=33)
   ->  Sort  (cost=10.40..10.40 rows=1 width=33)
         Sort Key: l.shipmode
         ->  Nested Loop  (cost=0.00..10.39 rows=1 width=33)
               ->  Index Scan using lineitem_pkey on lineitem l  (cost=0.00..5.02 rows=1 width=46)
                     Filter: ((shipmode = ANY ('{TRUCK,RAIL}'::bpchar[])) AND (commitdate < receiptdate) AND (shipdate <
               ->  Index Scan using orders_pkey on orders o  (cost=0.00..5.35 rows=1 width=29)
                     Index Cond: (o.orderkey = "outer".orderkey)
(8 rows)

Time: 3.400 ms
                                           QUERY PLAN Q13
------------------------------------------------------------------------------------------------
 Sort  (cost=346711.42..346711.92 rows=200 width=8)
   Sort Key: count(*), orders.count
   ->  HashAggregate  (cost=346701.28..346703.78 rows=200 width=8)
         ->  GroupAggregate  (cost=311846.89..344451.16 rows=150008 width=19)
               ->  Merge Left Join  (cost=311846.89..335080.68 rows=1499075 width=19)
                     Merge Cond: ("outer".custkey = "inner".custkey)
                     ->  Sort  (cost=20779.53..21154.55 rows=150008 width=9)
                           Sort Key: c.custkey
                           ->  Seq Scan on customer c  (cost=0.00..5826.08 rows=150008 width=9)
                     ->  Sort  (cost=291067.36..294815.05 rows=1499075 width=19)
                           Sort Key: o.custkey
                           ->  Seq Scan on orders o  (cost=0.00..52454.94 rows=1499075 width=19)
                                 Filter: (("comment")::text !~~ '%pending%accounts%'::text)
(13 rows)

Time: 20.105 ms
                                                          QUERY PLAN Q14
----------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=168251.55..168251.58 rows=1 width=88)
   ->  Nested Loop  (cost=0.00..168251.54 rows=1 width=88)
         ->  Seq Scan on lineitem l  (cost=0.00..168246.00 rows=1 width=96)
               Filter: ((shipdate >= '1996-08-01'::date) AND (shipdate < '1996-09-01 00:00:00'::timestamp without time z
         ->  Index Scan using part_pkey on part p  (cost=0.00..5.53 rows=1 width=33)
               Index Cond: ("outer".partkey = p.partkey)
(6 rows)

Time: 2.709 ms
CREATE VIEW
Time: 190.426 ms
```

```
                                                                QUERY PLAN Q15
-------------------------------------------------------------------------------------------------------------------------------------------
 Merge Join  (cost=336492.10..337191.77 rows=1 width=115)
   Merge Cond: ("outer".suppkey = "inner".supplierno)
   InitPlan
     ->  Aggregate  (cost=168246.04..168246.05 rows=1 width=32)
           ->  HashAggregate  (cost=168246.01..168246.02 rows=1 width=96)
                 ->  Seq Scan on lineitem l  (cost=0.00..168246.00 rows=1 width=96)
                       Filter: ((shipdate >= '1996-02-01'::date) AND (shipdate < '1996-05-01 00:00:00'::timestamp withou
   ->  Index Scan using supplier_pkey on supplier s  (cost=0.00..674.65 rows=10000 width=83)
   ->  Sort  (cost=168246.06..168246.06 rows=1 width=64)
         Sort Key: revenue0.supplierno
         ->  Subquery Scan revenue0  (cost=168246.01..168246.05 rows=1 width=64)
               ->  HashAggregate  (cost=168246.01..168246.04 rows=1 width=96)
                     Filter: (sum((extendedprice * (1::numeric - discount))) = $0)
                     ->  Seq Scan on lineitem l  (cost=0.00..168246.00 rows=1 width=96)
                           Filter: ((shipdate >= '1996-02-01'::date) AND (shipdate < '1996-05-01 00:00:00'::timestamp wi
(15 rows)

Time: 4.050 ms
DROP VIEW
Time: 36.367 ms
                                                                QUERY PLAN Q16
-------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=56359.68..56366.07 rows=2553 width=49)
   Sort Key: count(DISTINCT ps.suppkey), p.brand, p."type", p.size
   ->  GroupAggregate  (cost=55545.57..56215.21 rows=2553 width=49)
         ->  Sort  (cost=55545.57..55673.12 rows=51018 width=49)
               Sort Key: p.brand, p."type", p.size
               ->  Hash Join  (cost=10831.12..50371.91 rows=51018 width=49)
                     Hash Cond: ("outer".partkey = "inner".partkey)
                     ->  Seq Scan on partsupp ps  (cost=386.00..27878.86 rows=399643 width=16)
                           Filter: (NOT (hashed subplan))
                           SubPlan
                             ->  Seq Scan on supplier s  (cost=0.00..386.00 rows=1 width=7)
                                   Filter: (("comment")::text ~~ '%Customer%Complaints%'::text)
                     ->  Hash  (cost=10131.30..10131.30 rows=25528 width=51)
                           ->  Seq Scan on part p  (cost=0.00..10131.30 rows=25528 width=51)
                                 Filter: ((brand <> 'Brand#43'::bpchar) AND (("type")::text !~~ 'STANDARD BRUSHED%'::tex
(15 rows)

Time: 3.253 ms
                                      QUERY PLAN Q17
-----------------------------------------------------------------------------------
 Aggregate  (cost=336497.57..336497.58 rows=1 width=32)
   ->  Nested Loop  (cost=0.00..336497.57 rows=1 width=32)
         Join Filter: ("outer".quantity < (subplan))
         ->  Seq Scan on lineitem l  (cost=0.00..168246.00 rows=1 width=96)
         ->  Index Scan using part_pkey on part p  (cost=0.00..5.53 rows=1 width=9)
               Index Cond: (p.partkey = "outer".partkey)
               Filter: ((brand = 'Brand#31'::bpchar) AND (container = 'LG JAR'::bpchar))
         SubPlan
           ->  Aggregate  (cost=168246.01..168246.02 rows=1 width=32)
                 ->  Seq Scan on lineitem l2  (cost=0.00..168246.00 rows=1 width=32)
                       Filter: (partkey = $0)
(11 rows)

Time: 2.575 ms
                                                  QUERY PLAN Q18
-------------------------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=21.14..21.17 rows=1 width=90)
   ->  Sort  (cost=21.14..21.14 rows=1 width=90)
         Sort Key: o.totalprice, o.orderdate, c.name, c.custkey, o.orderkey
         ->  Nested Loop  (cost=0.00..21.13 rows=1 width=90)
               Join Filter: ("outer".orderkey = "inner".orderkey)
               ->  Nested Loop  (cost=0.00..16.10 rows=1 width=90)
                     ->  Nested Loop  (cost=0.00..10.41 rows=1 width=68)
                           ->  GroupAggregate  (cost=0.00..5.03 rows=1 width=64)
                                 Filter: (sum(quantity) > 312::numeric)
                                 ->  Index Scan using lineitem_pkey on lineitem l2  (cost=0.00..5.01 rows=1 width=64)
                           ->  Index Scan using orders_pkey on orders o  (cost=0.00..5.35 rows=1 width=36)
                                 Index Cond: (o.orderkey = "outer".orderkey)
                     ->  Index Scan using customer_pkey on customer c  (cost=0.00..5.68 rows=1 width=31)
```

```
                        Index Cond: (c.custkey = "outer".custkey)
                ->  Index Scan using lineitem_pkey on lineitem l  (cost=0.00..5.01 rows=1 width=64)
(15 rows)

Time: 4.418 ms

-------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=168251.59..168251.61 rows=1 width=64)
   ->  Nested Loop  (cost=0.00..168251.59 rows=1 width=64)
         Join Filter: ((("inner".brand = 'Brand#24'::bpchar) AND ("inner".container = ANY ('{"SM CASE","SM BOX","SM PACK
         ->  Seq Scan on lineitem l  (cost=0.00..168246.00 rows=1 width=128)
               Filter: ((shipmode = ANY ('{AIR,"AIR REG"}'::bpchar[])) AND (shipinstruct = 'DELIVER IN PERSON'::bpchar))
         ->  Index Scan using part_pkey on part p  (cost=0.00..5.53 rows=1 width=41)
               Index Cond: (p.partkey = "outer".partkey)
               Filter: (size >= 1)
(8 rows)

Time: 6.070 ms
                                                                                                    QUERY PLAN Q20
-------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=3372633.77..3372633.78 rows=1 width=57)
   Sort Key: s.name
   ->  Nested Loop  (cost=3372628.43..3372633.76 rows=1 width=57)
370          Join Filter: ("outer".nationkey = "inner".nationkey)
         ->  Nested Loop  (cost=3372628.43..3372632.44 rows=1 width=62)
               ->  HashAggregate  (cost=3372628.43..3372628.44 rows=1 width=7)
                     ->  Nested Loop  (cost=7631.65..3372628.43 rows=1 width=7)
                           ->  HashAggregate  (cost=7631.65..7631.66 rows=1 width=9)
                                 ->  Seq Scan on part p  (cost=0.00..7631.65 rows=1 width=9)
                                       Filter: ((name)::text ~~ 'papaya%'::text)
                           ->  Index Scan using partsupp_pkey on partsupp ps  (cost=0.00..3364996.68 rows=7 width=16)
                                 Index Cond: (ps.partkey = "outer".partkey)
                                 Filter: ((availqty)::numeric > (subplan))
                                 SubPlan
                                   ->  Aggregate  (cost=168246.01..168246.02 rows=1 width=32)
                                         ->  Seq Scan on lineitem l  (cost=0.00..168246.00 rows=1 width=32)
                                               Filter: ((partkey = $0) AND (suppkey = $1) AND (shipdate >= '1996-01-01':
               ->  Index Scan using supplier_pkey on supplier s  (cost=0.00..3.99 rows=1 width=69)
                     Index Cond: (s.suppkey = "outer".suppkey)
         ->  Seq Scan on nation n  (cost=0.00..1.31 rows=1 width=5)
               Filter: (name = 'MOZAMBIQUE'::bpchar)
(21 rows)

Time: 85.119 ms

                                                    QUERY PLAN Q21
-------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=25.77..25.77 rows=1 width=29)
   Sort Key: count(*), s.name
   ->  HashAggregate  (cost=25.75..25.76 rows=1 width=29)
         ->  Nested Loop  (cost=0.00..25.74 rows=1 width=29)
               ->  Nested Loop  (cost=0.00..20.37 rows=1 width=61)
                     Join Filter: ("outer".nationkey = "inner".nationkey)
                     ->  Nested Loop  (cost=0.00..19.05 rows=1 width=66)
                           ->  Index Scan using lineitem_pkey on lineitem l1  (cost=0.00..15.05 rows=1 width=64)
                                 Filter: ((receiptdate > commitdate) AND (subplan) AND (NOT (subplan)))
                                 SubPlan
                                   ->  Index Scan using lineitem_pkey on lineitem l3  (cost=0.00..5.02 rows=1 width=333)
                                         Index Cond: (orderkey = $0)
                                         Filter: ((suppkey <> $1) AND (receiptdate > commitdate))
407                                ->  Index Scan using lineitem_pkey on lineitem l2  (cost=0.00..5.02 rows=1 width=333)
                                         Index Cond: (orderkey = $0)
                                         Filter: (suppkey <> $1)
                           ->  Index Scan using supplier_pkey on supplier s  (cost=0.00..3.99 rows=1 width=41)
                                 Index Cond: (s.suppkey = "outer".suppkey)
                     ->  Seq Scan on nation n  (cost=0.00..1.31 rows=1 width=5)
                           Filter: (name = 'ROMANIA'::bpchar)
               ->  Index Scan using orders_pkey on orders o  (cost=0.00..5.36 rows=1 width=10)
                     Index Cond: (o.orderkey = "outer".orderkey)
                     Filter: (orderstatus = 'F'::bpchar)
(23 rows)

Time: 5.090 ms
```

QUERY PLAN Q22

```
 GroupAggregate  (cost=605298144.42..605298170.28 rows=862 width=30)
   InitPlan
     ->  Aggregate  (cost=8275.55..8275.56 rows=1 width=11)
           ->  Seq Scan on customer c1  (cost=0.00..8263.71 rows=4736 width=11)
                 Filter: ((acctbal > 0.00) AND ("substring"((phone)::text, 1, 2) = ANY ('{25,11,14,28,27,16,24}'::text[]
   ->  Sort  (cost=605289868.86..605289871.01 rows=862 width=30)
         Sort Key: "substring"((c.phone)::text, 1, 2)
         ->  Seq Scan on customer c  (cost=0.00..605289826.83 rows=862 width=30)
               Filter: (("substring"((phone)::text, 1, 2) = ANY ('{25,11,14,28,27,16,24}'::text[])) AND (acctbal > $0) A
               SubPlan
                 ->  Seq Scan on orders o  (cost=0.00..52454.94 rows=13 width=135)
                       Filter: (custkey = $1)
(12 rows)

Time: 16.256 ms
```

REFERÊNCIAS BIBLIOGRÁFICAS

[1] *PostgreSQL, object-relational database management system*, acessado em 03/05/2005.

[2] *Benchmark H (Decision Support) Standard Specification*, acessado em 10/03/2006. http://www.tpc.org/tpch.

[3] Steinbrunn M; Moerkotte G; Kemper A. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, August de 1997.

[4] Eduardo Cunha de Almeida. Estudo de viabilidade de uma plataforma de baixo custo para data warehouse. Dissertação de Mestrado, Universidade Federal do Paraná, 2004. http://dspace.c3sl.ufpr.br/dspace/handle/1884/662.

[5] Krishnamurthy R; Boral H; Zaniolo C. Optimization of non-recusrsive queries. *In proc. of the Conference on Very Large Data Bases (VLDB)*, páginas 128–137, 1986.

[6] M M; et al Chamberlin, D D; Astrahan. Sequel 2: A unified approach to data definition, manipulation and control. *IBM Journal of Research and Development*, November de 1976.

[7] E F Codd. A relational model for large shared data banks. *Communications of the ACM*, June de 1970.

[8] N Conway. Inside the postgresql query optimizer. Relatório técnico, Fujitsu Australia Software Tecnology. http://neilc.treehou.se/optimizer.pdf.

[9] Oracle Corporation. *Database Performance Tuning Guide and Reference*, acessado em 12/08/2005. http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96533/toc.htm.

[10] Leonidas Fegaras. A new heuristic for optimizing large queries. *Database and Expert Systems Applications, 9th International Conference, DEXA '98, Vienna, Austria, August 24-28, 1998, Proceedings*, páginas 726–735, 1998.

[11] Z Fong. The design and implementation of the postgres query optimizer. Relatório técnico, Univ. of California, Berkeley, CA, 1986.

[12] Selinger P G; Astrahan M M; Chamberlin D D; Lorie R A; Price T G. *Database Systems - Access Path Selection in a Relational Database System.* Morgan Kaufman, 1979.

[13] G Graefe. The cascades framework for query optimization. *Data Engineering Bulletin*, September de 1995.

[14] Dick Grune. *LINUX MAN - CVS(1).* Debian, 1986.

[15] Haas L; Freytag J C; Lohman G M; Pirahesh H. Extensible query processing in starburst. *In Proc. of ACM SIGMOD*, 1989.

[16] Nguyen G T; Ferrat L; Galy H. A high-level user interface for a local network database system. *In Proc. of IEEE Infocom*, 1982.

[17] Stonebraker M; Hanson E; Hong C H. The design of the postgres rules system. *Proc. IEEE Conference on Data Engineering*, 1987.

[18] Graefe G; Dewitt D J. The exodus optimizer generator. *In Proc. of ACM SIGMOD*, 1987.

[19] Graefe G; McKenna W J. The volcano optimizer generator: Extensibility and efficient search. *In Proc. of the IEEE Conference on Data Engineering*, 1993.

[20] Ilyas I F; Aref W G; Elmagarmid A K. Supporting top-k join queries in relational databases. *The VLDB Journal*, September de 2004.

[21] J B Kruskal. On the shortestspanning subtree of a graph and the travelling salesman problem. *American Math Soc.*, 1956.

[22] Tom Lane. *Documentação de alterações de códigos do otimizador PostgreSQL.* PostgreSQL, 2003. http://developer.postgresql.org/cvsweb.cgi/pgsql/src/backend/optimizer/README.

[23] Carey M J; DeWitt D J; Frank D; Graefe G; Richardson J E; Shekita E J; Muralikrishma M. The architecture of the exodus extensible dbms: A preliminary report. *In Proc. of the Internacional Workshop on Object-Oriented Database Systems*, 1986.

[24] Ono K; Lohman G M. Measuring the complexity of join enumeration in query optimization. *In Proc. of Conf Very Large Data Base - VLDB*, 1990.

[25] Stonebraker M; Rowe L A; Hirohama M. The implementation of postgres. *IEEE Transactions on Knowledge and Data Engineering*, 1990.

[26] Carlos Alberto de Jesus Martinhon. Algoritmos randômicos em grafos e otimização. Relatório técnico, Universidade Federal Fluminense, 2005.

[27] D. McDermott. *PDDL - the planning domain definition language.* Yale University/USA, 1998.

[28] D. McDermott. *The 1998 AI Planning Systems Competition*. Yale University/USA, 2000.

[29] Nils J Nilsson. *Artificial Intelligence A New Synthesis*. Morgan Kaufmann Publishers, 1998.

[30] Fatma Ozcan, Sena Nural, Pinar Koksal, Mehmet Altinel, e Asuman Dogac. A region based query optimizer through cascades query optimizer framework. *IEEE Data Eng. Bull.*, 18(3):30–40, 1995.

[31] M Rowe, L L; Stonebraker. The design of postgres. *Proc. ACM-SIGMOD Conference on Management of Data*, 1986.

[32] Silberschatz A; Korth H F; Sudarshan S. *Sistema de banco de dados*. Makron Books, 1997.

[33] Stonebraker M; Hearst M; Potamianos S. A commentary on the postgres rules system. *SIGMOD Record*, 1989.

[34] P; et. al Selinger. Access path selection in a relational data base system. *1979 ACM-SIGMOD Conference on Management of Data*, 1979.

[35] PostgreSQL Developer Team e Doxygen. *PostgreSQL Documentation - CVS-HEAD*, acessado em 17/01/2006. http://www.mcknight.de/pgsql-doxygen/cvshead/html/main.html.

[36] Ibaraki; Toshihide; Kameda; Tiko. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems*, September de 1984.

[37] Martin Utesch. *Genetic query optimization in database systems*. http://www.postgresql.org/docs/8.0/interactive/geqo.html.

[38] Felipe Vieira. Planejamento automático aplicado a problemas dependentes de recursos. Dissertação de Mestrado, Universidade Federal do Ceará, 2003. http://www.mdcc.ufc.br/disser/FelipeVieira.pdf.

PRYSCILA BARVICK GUTTOSKI

OTIMIZAÇÃO DE CONSULTAS NO POSTGRESQL UTILIZANDO O ALGORITMO DE KRUSKAL

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

2006