ADRIANO LANGE

UMA AVALIAÇÃO DE ALGORITMOS NÃO EXAUSTIVOS PARA A OTIMIZAÇÃO DE JUNÇÕES

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

2010

ADRIANO LANGE

UMA AVALIAÇÃO DE ALGORITMOS NÃO EXAUSTIVOS PARA A OTIMIZAÇÃO DE JUNÇÕES

Dissertação aprovada como requisito parcial à 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. Fábio André Machado Porto
Laboratório Nacional de Computação Científica, LNCC, membro externo.

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

Curitiba, 26 de Agosto de 2010

AGRADECIMENTOS

Agradeço primeiramente a Deus, pela vida, saúde e oportunidade de realizar este estudo.

Ao meu orientador, principalmente pelo apoio, compreensão e paciência.

Agradeço também ao corpo docente do Departamento de Informática da UFPR, do qual tive a oportunidade de aprender muito.

Aos meus pais, por estarem sempre prontos a me ajudar.

Aos inúmeros amigos que tive a oportunidade de conhecer durante a minha estada em Curitiba, dos quais agradeço pela companhia e pelas valiosas contribuições.

A Universidade Estadual de Mato Grosso do Sul e a todos os colegas Divisão de Informática, pelo valioso apoio e amizade.

SUMÁRIO

LISTA DE FIGURAS vi

LISTA DE TABELAS vii

RESUMO viii

ABSTRACT ix

1 INTRODUÇÃO 1

1.1 A Ordenação de Junções 2

1.2 O SGBD PostgreSQL 3

1.3 Motivação e Objetivo 4

1.4 Organização da Dissertação 6

2 O PROCESSAMENTO DE CONSULTAS EM SISTEMAS DE BANCOS DE DADOS RELACIONAIS 8

2.1 Fundamentos 8

2.1.1 Álgebra Relacional 9

2.1.1.1 Árvore de Consulta 11

2.1.1.2 Operador de Junção 12

2.1.2 A Linguagem de Consultas SQL 14

2.2 Arquitetura de Processamento de Consultas 15

2.2.1 O Planejamento 17

2.2.1.1 Restrições do Espaço de Busca 18

3 ALGORITMOS DE OTIMIZAÇÃO DE JUNÇÕES 22

3.1 Algoritmos Exaustivos 23

3.2 Algoritmos Não Exaustivos 25

3.2.1 Algoritmos Determinísticos 26

3.2.2 Algoritmos Aleatórios 26
3.2.2.1 Melhoria Iterativa (II) 28
3.2.2.2 Têmpera Simulada (SA) 28
3.2.2.3 Aplicação dos Algoritmos Aleatórios na Otimização de Junções 29
3.2.2.4 Two Phase Optimization (2PO) 32
3.2.3 Algoritmos Genéticos 34
3.2.3.1 Árvores em Profundidade à Esquerda 35
3.2.3.2 Árvores Fechadas de Junções 36

4 OTIMIZAÇÃO DE JUNÇÕES NO POSTGRESQL 38

4.1 Arquitetura 38
4.1.1 A Estrutura *RelOptInfo* 40
4.1.2 Contrato e Operações Básicas de um Otimizador 41
4.2 O Algoritmo GEQO 43
4.2.1 Espaço de Busca 43
4.2.2 Seleção, Crossover e Evolução 45
4.3 A Implementação do Algoritmo 2PO 48
4.3.1 Espaço de Busca 48
4.3.2 Movimentos 50
4.3.3 Encapsulamento do 2PO na Forma de *Plugin* 51

5 METODOLOGIAS DE AVALIAÇÃO ENCONTRADAS NA LITERATURA 53

5.1 A Ausência de Padrões Definidos de Avaliação 53
5.2 A Comparação de Algoritmos Aleatórios e o Uso da Escala de Custo 54
5.3 A Comparação Entre 2PO, SA e II 56
5.4 A Comparação de Algoritmos Determinísticos e Não Desterminísticos . . . 57
5.5 Propostas Determinísticas de Avaliação 59
5.6 O Uso de *Benchmarks* Tradicionais 62

5.7 Principais Elementos Encontrados nas Metodologias Apresentadas 63

6 METODOLOGIA UTILIZADA 65

6.1 Definição dos Parâmetros de Construção 65

6.2 Construção do Esquema de Dados 68

6.3 Construção das Consultas 68

6.3.1 Seleção dos Conjuntos de Relações 69

6.3.2 Combinação das Relações na Consulta SQL 69

6.4 Transitividade de Predicados 71

7 RESULTADOS EXPERIMENTAIS 74

7.1 Performance dos Otimizadores 76

7.2 Qualidade dos Planos Gerados 80

7.3 Variação das Cardinalidades das Relações 86

7.4 Análise dos Resultados Apresentados 88

8 CONCLUSÃO 91

REFERÊNCIAS BIBLIOGRÁFICAS 98

LISTA DE FIGURAS

2.1 Exemplo de uma árvore de consulta. 11

2.2 Exemplo de uma árvore de consulta usando um operador de junção. 12

2.3 Exemplo de um grafo de junções. Os nodos representam as relações e as arestas os predicados sobre essas relações. 14

2.4 Tipos de grafos de junções. 14

2.5 Mapeamento de equivalência entre uma consulta SQL em seus respectivos planos físicos. 16

2.6 Exemplos de árvores de junções. 19

2.7 Tipos de árvores de junções. 21

3.1 Otimização local do II *versus* SA [43]. 30

3.2 Exemplo de construção de uma árvore em profundidade à esquerda equivalente a lista de relações (A, C, D, B). 31

3.3 Forma do espaço de busca de árvores fechadas de junções apresentada por Ioannidis e Kang [29]. 33

3.4 Exemplo (a) grafo e (b) árvore de junções utilizado por Bennett *et al.* [13] para representar cromossomos em um ambiente de árvores fechadas. 37

4.1 Arquitetura de processamento de consultas do PostgreSQL [1]. 39

4.2 Exemplo de uma estrutura *RelOptInfo* utilizada pelo PostgreSQL para agregar vários planos de execução de um mesmo conjunto de relações. . . . 40

4.3 Exemplo de uma árvore alternativa utilizada pelo GEQO para representar cromossomos que contenham falhas em sua sequência normal de combinações. 45

4.4 Forma de representação de estados utilizada pelo 2PO implementado neste estudo. 49

4.5 Regras de transformação utilizadas na implementação do algoritmo 2PO. . 51

4.6 Diagrama de componentes representando a interface de conexão do *plugin* LJQO com o PostgreSQL. 52

6.1 Aumento da conectividade nas consultas do tipo grade e estrela devido a transitividade de seus predicados de junção. 73

7.1 Média da quantidade de planos gerados pelos otimizadores 2PO, GEQO e GEQO+ em função do número de relações e do grafo de junções. 77

7.2 Tempo médio de otimização apresentado pelos otimizadores 2PO, GEQO e GEQO+ em função do número de relações e do grafo de junções. 78

7.3 Média de planos gerados por segundo obtida pelos otimizadores 2PO, GEQO e GEQO+ em função do número de relações e do grafo de junções. 79

7.4 Média e variação das escalas de custo obtidas pelos otimizadores 2PO, GEQO e GEQO+ em função da quantidade de relações e do grafo de junções. Os gráficos nesta figura agrupam os dados de acordo com os grafos de junções. Cada retângulo representa a média das escalas de custo, enquanto que o intervalo apresentado em seu topo ou interior representa o 5º e o 95º percentis. 82

7.5 Média e variação das escalas de custo obtidas pelos otimizadores 2PO, GEQO e GEQO+ em função da quantidade de relações e do grafo de junções. Os gráficos nesta figura agrupam os dados de acordo com a quantidade de relações. Cada retângulo representa a média das escalas de custo, enquanto que o intervalo apresentado em seu topo ou interior representa o 5º e o 95º percentis. 83

7.6 Média e variação das escalas de custo obtidas pelos otimizadores 2PO, GEQO e GEQO+ em função da quantidade de relações, grafo de junções e *LOGRATIO*. Cada retângulo representa a média das escalas de custo, enquanto que o intervalo apresentado em seu topo ou interior representa o 5º e o 95º percentis. 87

LISTA DE TABELAS

2.1 Relação *Departamento* (D) 11

2.2 Relação *Funcionário* (F) 11

2.3 Número de tipos de árvores de junções e quantidade de possíveis soluções para consultas entre 1 e 12 relações. [39, 47]. 20

4.1 Parâmetros de configuração utilizados pelo 2PO em cada uma de suas fases, conforme proposto por Ioannidis e Kang [29]. 49

5.1 Catálogos de relações utilizados por Ioannidis e Kang [29]. 56

5.2 Distribuição das cardinalidades das relações utilizadas por Steinbrunn *et al.* [43]. 58

5.3 Distribuição dos domínios dos atributos utilizados por Steinbrunn *et al.* [43]. 58

6.1 Comparativo entre os parâmetros utilizados por Vance e Maier e por Shapiro *et al.* em relação a metodologia adotada. 66

6.2 Comparação da quantidade de arestas do grafo de junções de uma consulta SQL, antes e depois da reescrita. 72

7.1 Média de planos gerados por segundo obtida pelos otimizadores 2PO, GEQO e GEQO+ em função do número de relações e do grafo de junções. 80

7.2 Média das escalas de custo obtidas pelos otimizadores 2PO, GEQO e GEQO+ em função da quantidade de relações e do grafo de junções. 84

RESUMO

Os sistemas gerenciadores de bancos de dados (SGBDs) relacionais proporcionam atualmente uma ambiente bastante produtivo para a manipulação de informações. A partir de uma linguagem de alto nível, tais sistemas permitem que seus usuários descrevam consultas de uma maneira simples e rápida, sem definir com isso detalhes relacionados ao seu processamento. Tais detalhes são de responsabilidade do próprio SGBD, o qual deve escolher, através de um sofisticado processo de otimização e planejamento, uma alternativa eficiente para a obtenção dessas informações. A otimização de junções é uma das mais importantes e complexas dentre todas as fases que compõem este processo. A definição da melhor ordem de junções somente pode ser realizada em condições relativamente simples, através do uso de algoritmos de busca exaustiva fortemente baseados na programação dinâmica. Para os demais casos, espera-se que apenas uma aproximação desta melhor ordem seja encontrada, utilizando para isso técnicas não exaustivas de busca. Este estudo concentra-se na avaliação de dois algoritmos não exaustivos de otimização de junções implementados para o SGBD de código aberto PostgreSQL: o *Genetic Query Optimization* (GEQO) e o *Two Phase Optimization* (2PO). Através de um esquema de testes multidimensional, este estudo apresenta diversos dados relevantes sobre o comportamento desses algoritmos. Estes resultados servem tanto para o processo de melhoria dos algoritmos avaliados como para a elaboração de novas abordagens de otimização de junções.

ABSTRACT

Relational Database Management Systems (RDBMS) currently provide a very productive environment for data manipulation. Using a high level language, these systems allow their users describe queries in a simple and fast manner without defining how these data will be retrieved. These details need to be supplied by RDBMS itself, through a sophisticated process of optimizing and planning. The join ordering optimization is one of the most important and complex phases that involve this process. The definition of the optimal join order can only be realized in simple conditions, by using exhaustive search techniques. For the other cases, it is expected that an approximation of the optimal join order should be found by a non-exhaustive search algorithm. This study concentrates on the evaluation of two non-exhaustive algorithms applied to an open source DBMS named PostgreSQL: Genetic Query Optimization (GEQO) and Two Phase Optimization (2PO). Through a multidimensional test schema, this study demonstrates several relevant information about the behavior of these algorithms. These results can be used both for improvement of such algorithms and for elaboration of new join ordering techniques.

CAPÍTULO 1

INTRODUÇÃO

Os sistemas gerenciadores de bancos de dados (SGBDs) são ferramentas de fundamental importância para o meio social em que vivemos. É atualmente difícil de se imaginar que qualquer pessoa ou empresa não utilize, direta ou indiretamente, informações fornecidas por algum SGBD, seja pelo simples uso de serviços bancários, pela submissão de uma pesquisa em um site de busca na Internet ou até mesmo pela geração de complexas informações gerenciais fornecidas por sistemas de apoio a decisão de grandes corporações.

O objetivo principal de um SGBD é abstrair a complexidade do armazenamento, controle de acesso e recuperação de informações, visando com isso aumentar a produtividade no desenvolvimento de aplicações ou mesmo facilitar o acesso aos dados por usuários finais inexperientes. Segundo Jarke e Koch [32], existem duas principais áreas de interesse na pesquisa e desenvolvimento de SGBDs. A primeira delas, compete à análise dos modelos conceituais de dados nos quais pode-se representar elementos do mundo real. Tais conceitos incluem, por exemplo, os modelos hierárquico, em rede e relacional. A segunda área de interesse consiste em implementar tais modelos de forma que sejam computacionalmente eficientes.

Dentre os modelos de bancos de dados existentes, o modelo relacional, definido por Codd em 1970 [17], possui uma sólida fundamentação teórica [32]. A característica básica deste modelo é proporcionar uma clara distinção entre os aspectos lógicos e físicos de um banco de dados, considerando para isso tanto sua representação como os métodos computacionais utilizados para a manipulação e recuperação de seus dados [19]. A partir deste modelo, diversas pesquisas foram desenvolvidas nos anos subsequentes à sua definição, de modo que protótipos de SGBDs relacionais começaram a surgir a partir da segunda metade da década de 70, como por exemplo o System R [11], desenvolvido nos laboratórios da IBM, e o INGRES [26], desenvolvido pela *University of California*, em Berkeley.

A partir da abstração proporcionada pelo modelo relacional, algumas linguagens de alto nível foram propostas durante o desenvolvimento desses SGBDs como forma de representar consultas submetidas a eles. A linguagem SQL (*Structured Query Language*) [12], por exemplo, foi proposta inicialmente para o System R, enquanto que o INGRES foi equipado com uma linguagem chamada QUEL [26]. Tanto a linguagem SQL como a linguagem QUEL eram não procedurais, as quais permitiam aos seus usuários descreverem os dados desejados em uma consulta sem a necessidade de especificar como eles deveriam ser obtidos.

A forma descritiva de suas linguagens de consulta proporciona aos SGBDs relacionais um ambiente de utilização muito mais produtivo que os antigos métodos de consulta adotados nos modelos hierárquico e em rede. Por outro lado, seu alto nível de abstração exigiu dos SGBDs relacionais um mecanismo de processamento de consultas muito mais sofisticado. Internamente, esses SGBDs implementam um conjunto de métodos de manipulação de dados que derivam da *álgebra relacional* [18]. Tais métodos podem ser combinados em um *plano de execução* para produzirem o resultado esperado pelo usuário. O principal problema, neste caso, é que uma consulta descrita por este tipo de linguagem de alto nível pode ser representada por diversos planos de execução diferentes. Embora cada um desses planos seja equivalente do ponto de vista de seu resultado final, a quantidade de recursos computacionais e de tempo exigido para o processamento de cada um deles pode variar consideravelmente. A partir deste fato, a tarefa mais crítica de um SGBD relacional ao receber uma consulta, consiste em determinar um plano de execução eficiente, através de etapas que envolvem sua otimização e planejamento.

1.1 A Ordenação de Junções

Desde o projeto dos primeiros SGBDs relacionais, o problema da otimização de consultas foi e continua sendo um dos mais complexos dentre todas as áreas que envolvem sua pesquisa e desenvolvimento. Neste contexto, diversos estudos foram realizados ao longo dos últimos 30 anos com o objetivo de tornar este processo cada vez mais eficiente.

O processo de otimização de consultas concentra-se geralmente em uma tarefa chamada

ordenação de junções ou *otimização de junções*. Como será descrito no capítulo seguinte, *junção* é uma operação da álgebra relacional que permite combinar duas *relações*, ou *tabelas*, de um banco de dados relacional, usando para isso uma condição lógica sobre seus atributos. A ordenação de junções é uma tarefa importante porque apresenta um impacto significativo sobre a eficiência de SGBDs relacionais. Segundo Ibaraki e Kameda [27], encontrar a melhor ordem de junções de uma consulta é um problema NP-Completo, visto que a quantidade de possíveis soluções para este problema é no mínimo exponencialmente proporcional à quantidade de relações existentes na mesma.

Existem diversos algoritmos propostos na literatura que buscam lidar com o problema da ordenação de junções. Estes algoritmos podem ser divididos em duas categorias principais: *exaustivos* e *não exaustivos*. Os algoritmos exaustivos sempre retornam a melhor ordem de junções possível para o problema. Esses algoritmos também são chamados de algoritmos de *programação dinâmica*, devido a utilização desta técnica, introduzida inicialmente para o System R em 1979 por Selinger *et al.* [41]. Os algoritmos de programação dinâmica estão presentes em praticamente todos os SGBDs relacionais existentes atualmente. Sua aplicação se restringe, contudo, a condições relativamente simples, quando o número de relações de uma consulta não ultrapassa 10 ou 15. Acima deste limite, a quantidade de tempo e de memória exigidos por estes algoritmos é grande o suficiente para tornar sua utilização impraticável. Os algoritmos não exaustivos, por sua vez, são aqueles aplicáveis em condições onde os algoritmos de busca exaustiva não são capazes de atuar. Tais algoritmos apresentam uma abordagem de aproximação sobre o que poderia ser uma possível melhor ordem de junções. Embora apresentem uma degradação em sua qualidade de otimização, a quantidade de tempo e consumo de memória desses algoritmos é significativamente menor que o exigido pela busca exaustiva.

1.2 O SGBD PostgreSQL

O PostgreSQL é um SGBD objeto-relacional de código aberto, o qual é resultado de um longo processo evolutivo que teve início na *University of California*, em Berkeley. O desenvolvimento deste SGBD deriva diretamente de um projeto chamado POSTGRES,

que quer dizer pós-INGRES, iniciado em 1986 pelo professor Michael Stonebraker [45] após várias retrospectivas feitas com relação ao seu predecessor, o INGRES [44].

Atualmente em sua versão 8.4, o PostgreSQL é amplamente conhecido como um dos mais avançados SGBDs de código aberto existentes, sendo este mantido deste 1996 por uma comunidade global de desenvolvedores [5]. Ao contrário do INGRES e de suas primeiras versões, ainda com o nome POSTGRES, a versão atual deste SGBD utiliza a SQL como linguagem de consulta em lugar da antiga linguagem QUEL. Além disso, o PostgreSQL apresenta uma série características que são bastante desejáveis em ambientes empresariais, como o controle de concorrência baseado por múltiplas versões (MVCC, *Multi-Version Concurrency Control*), a utilização de vários esquemas por banco de dados, incluindo sua distribuição em diversos dispositivos de armazenamento (*tablespaces*), replicação assíncrona, transações aninhadas (*savepoints*) e backup *online*.

1.3 Motivação e Objetivo

Devido aos diversos recursos oferecidos pelo PostgreSQL, bem como pela maturidade alcançada ao longo de vários anos de desenvolvimento, este SGBD tem sido alvo de uma atenção crescente no sentido de ser uma alternativa de baixo custo viável para diversas áreas onde somente SGBDs comerciais eram capazes de atuar [22]. Por consequência, esta atenção tem demandado novas melhorias, as quais seus desenvolvedores têm se esforçado em atendê-las de modo a tornar este SGBD cada vez mais completo e eficiente.

Uma das demandas encontrados no PostgreSQL está relacionada com sua capacidade em lidar com situações que requerem consultas complexas, principalmente aquelas que agregam um grande número de relações. Tais tipos de consultas são frequentemente encontradas em sistemas de apoio à decisão (DSS, *Decision Support System*), os quais são geralmente compostos por ferramentas de geração de complexos relatórios gerenciais (OLAP, *Online Analytical Processing*) e por ferramentas dedutivas de mineração automatizada de dados (*Data Mining*). Tais ferramentas são responsáveis por extrair informações armazenadas em grandes bases de dados, chamadas *Data Warehouses*, sendo estas muitas vezes capazes de gerar consultas com dezenas ou até centenas de relações. Além disso, duas

outras fontes potenciais para este tipo de consulta são os utilitários gráficos de geração automática de consultas e os recentes *frameworks* de persistência de dados. Nesses casos, tanto usuários como programadores podem vincular dados contidos em diversas relações, sem com isso terem ciência da complexidade exigida para seu processamento.

No PostgreSQL, o suporte a consultas com grande número de relações foi introduzido em 1997, por Martin Utesch, com o desenvolvimento de um algoritmo não exaustivo de otimização de junções chamado *Genetic Query Optimization* (GEQO) [52]. Este algoritmo foi desenvolvido dentro da *University of Mining and Technology*, em Freiberg, Alemanha, a qual estava enfrentando alguns problemas ao tentar utilizar o PostgreSQL como base de um sistema de apoio à decisão direcionado a um ambiente de transmissão e distribuição de energia elétrica. Como era de se esperar, algumas consultas produzidas neste sistema eram complexas demais para serem otimizadas pelo então algoritmo de programação dinâmica deste SGBD.

Como seu próprio nome sugere, o GEQO é um algoritmo genético, o qual teve boa parte de seu código fonte derivado de um projeto chamado GENITOR [54]. Em sua abordagem de otimização, este algoritmo é fortemente inspirado no problema do Caixeiro Viajante (TSP, *Travelling Salesman Problem*) [20], considerando cidades como relações e o caminho entre essas cidades como as operações de junção. Desde sua implementação até a mais recente versão estável do PostgreSQL, este algoritmo não sofreu grandes modificações, sendo este utilizado como única alternativa para a otimização de consultas com grande número de relações.

Embora seja um algoritmo que esteja em uso a vários anos, o GEQO tem gerado uma certa insatisfação por parte dos usuários e desenvolvedores do PostgreSQL. Tais fatos podem ser observados da seguinte forma:

1. A lista de discussão dos desenvolvedores do PostgreSQL [2] possui vários tópicos relacionados com o comportamento insatisfatório deste algoritmo. Por este motivo, a substituição do GEQO é atualmente um item da lista de tarefas pendentes deste projeto [3];

2. A partir da versão 8.3, o PostgreSQL apresenta algumas funcionalidades que visam

minimizar o uso do GEQO, como por exemplo os parâmetros *join_collapse_limit* e *from_collapse_limit*. Esses parâmetros permitem que uma consulta seja dividida em vários blocos com quantidades iguais de relações, o que possibilita ao algoritmo exaustivo de programação dinâmica otimizar esses blocos de forma iterativa;

3. Existem ainda problemas de regularidade nos planos apresentados pelo GEQO. Conforme relatado por Bini *et al.* [14], em alguns casos os planos gerados são impraticáveis, o que diminui significativamente a confiabilidade deste SGBD.

O objetivo deste estudo é apresentar informações que possam contribuir para a melhoria do processo de otimização de junções do PostgreSQL. A partir de um levantamento sobre as características que envolvem este processo, e de uma retrospectiva sobre diversos tipos de algoritmos propostos na literatura, este estudo buscou implementar outro algoritmo não exaustivo de otimização, chamado *Two Phase Optimization* (2PO) [29]. Este algoritmo foi adequado às estruturas internas do PostgreSQL, possibilitando assim uma comparação com o algoritmo GEQO. A metodologia utilizada nesta avaliação segue um esquema sistemático e multidimensional, a qual deriva de diversos outros estudos relacionados a este assunto [29, 42, 43, 46, 53].

1.4 Organização da Dissertação

Este estudo está organizado da seguinte forma. No Capítulo 2, são apresentados os fundamentos do modelo relacional e como suas características implicam no problema da otimização de junções. Em seguida, o Capítulo 3 apresenta vários algoritmos de otimização de junções encontrados na literatura, destacando principalmente a forma representativa adotada por cada um deles. O Capítulo 4 apresenta como o PostgreSQL realiza a otimização de junções. Neste capítulo é descrito o algoritmo GEQO e o algoritmo 2PO, implementado durante a realização deste estudo. Como forma de avaliar tais algoritmos, os três capítulos seguintes apresentam diversas questões que foram consideradas neste processo. O Capítulo 5 apresenta uma revisão literária sobre as possíveis formas de avaliação. O Capítulo 6 descreve a metodologia adotada neste estudo e o Capítulo 7 os resultados

experimentais obtidos nesta avaliação. Por fim, o Capítulo 8 conclui este estudo.

CAPÍTULO 2

O PROCESSAMENTO DE CONSULTAS EM SISTEMAS DE BANCOS DE DADOS RELACIONAIS

2.1 Fundamentos

O modelo relacional de banco de dados, definido por Codd em 1970 [17], pode ser considerado como o principal fundamento para a maioria dos SGBDs existentes atualmente. A característica básica deste modelo é proporcionar uma clara distinção entre os aspectos lógicos e físicos de um banco de dados, considerando para isso tanto sua representação como os métodos utilizados para a manipulação e recuperação de seus dados [19].

Na época em que este modelo foi proposto, os SGBDs existentes eram baseados nos modelos hierárquico e em rede. Para estes modelos, os usuário eram obrigados a escrever rotinas de acesso a seus dados. Essas rotinas eram altamente dependentes das característica nas quais os dados estavam armazenados, incluindo sua ordenação e a possível presença de ponteiros e índices. Desta forma, tais SGBDs apresentavam um baixo grau de isolamento a mudanças, de modo que programas escritos para eles eram frequentemente passíveis de alteração caso alguma característica física do banco de dados fosse alterada.

No modelo relacional, os dados podem ser representados em sua forma natural, sem qualquer imposição proveniente de suas estruturas físicas de armazenamento ou de qualquer algoritmo utilizado para acessá-los. Para isso, os dados são representados por meio de *relações*. Uma relação pode ser considerada como um conjunto de dados dispostos em uma tabela bidimensional, onde cada coluna representa um *atributo* e cada linha uma instância, ou *tupla*, desta relação. A quantidade de atributos que uma relação possui é denominada *grau da relação* e sua quantidade de tuplas é denominada *cardinalidade*. Para este modelo, não existe qualquer imposição quanto a forma ou ordem em que os atributos devem estar dispostos, nem qualquer imposição quanto a ordem de suas respectivas

tuplas.

2.1.1 Álgebra Relacional

Os SGBDs relacionais utilizam conjuntos de relações para representar seus esquemas de dados. As relações que compõem um banco de dados são denominadas *relações básicas*. Cada uma dessas relações deve possuir um nome único que a identifique dentro deste banco de dados. Do mesmo modo, cada um de seus atributos deve possuir um nome único que o identifique dentro de sua própria relação.

Para que os dados possam ser recuperados a partir das relações básicas, esses SGBDs implementam internamente um conjunto de operações derivadas da *álgebra relacional* [18]. O princípio básico desta álgebra é produzir relações que são obtidas a partir de relações já existentes. Para isso, cada uma de suas operações (ou operadores) deve assumir como entrada (ou operandos) uma ou mais relações, produzindo como resultado uma nova relação. Existem cinco operações principais que compõem a álgebra relacional:

Seleção ($\sigma_C(R)$): O operador de seleção (σ), aplicado a uma relação de entrada R, produz uma nova relação S com os mesmos atributos de R, onde as tuplas de S são um subconjunto de R que satisfazem uma condição lógica C sobre seus atributos. Ou seja:

$$S = \{t \in R | t \text{ satisfaz } C\} \tag{2.1}$$

A razão entre a cardinalidade de S, denotada por $|S|$, pela cardinalidade de R ($|R|$) é denominada *seletividade*.

Projeção ($\pi_L(R)$): O operador de projeção (π) é usado para modificar os atributos retornados por uma relação. Dada uma relação R qualquer com $A_1, A_2, A_3, ..., A_n$ atributos, a expressão $\pi_{A_1,A_2,A_3}(R)$ produzirá como resultado uma nova relação S,

com as mesmas tuplas[1] de R mas com apenas os atributos A_1, A_2 e A_3.

$$S = \pi_{A_1,A_2,A_3}(R) \tag{2.2}$$

Produto Cartesiano ($R \times S$): O produto cartesiano ($\times$) é um operador binário que, dadas duas relações R e S, produz uma nova relação T de modo que suas tuplas são a combinação de cada tupla de R por cada tupla de S. O grau da nova relação T é igual a soma dos graus de R e S, enquanto que sua cardinalidade é igual ao o produto das respectivas cardinalidades de R e S.

$$T = R \times S \tag{2.3}$$

União ($R \cup S$) e Diferença ($R - S$): Estas são operações muito parecidas com o que se utiliza para conjuntos matemáticos, exceto pelo fato de poderem ou não permitirem a repetição de tuplas com os mesmos valores em seus atributos. As operações que permitem a repetição de tuplas utilizam uma notação de *sacolas*[2] ao invés de conjuntos [49].

O que torna a álgebra relacional uma ferramenta poderosa para o processamento de relações em um banco de dados relacional é a capacidade de combinação desses operadores, como é o caso do exemplo a seguir:

$$\pi_{\text{F.nome, D.nome}}(\sigma_{\text{F.id_dep = D.id}}(D \times \sigma_{\text{F.salario} \geq 100}(F))) \tag{2.4}$$

onde D e F representam, respectivamente, as relações *Departamento* e *Funcionário* das Tabelas 2.1 e 2.2.

[1] No modelo relacional puro, as relações são tratadas como conjuntos matemáticos, de modo que não são permitidas tuplas que tenham exatamente os mesmos valores em seus respectivos atributos. Contudo, em implementações de SGBDs relacionais, esta repetição é permitida por questões de performance [49]. Por isso, a operação de projeção descrita aqui segue esta permissão.

[2] O termo original em inglês é *Bag* [49].

Tabela 2.1: Relação *Departamento* (D)

id	nome
1	Contabilidade
3	RH
2	Vendas

Tabela 2.2: Relação *Funcionário* (F)

id	nome	salário	id_dep
1	Joaquim	$80,00	2
2	Francisco	$200,00	1
4	Camila	$150,00	2

2.1.1.1 Árvore de Consulta

As expressões feitas por meio da álgebra relacional também podem ser representadas na forma de um grafo, chamado *árvore de consulta*, ou *query tree*. Nesta notação, os nodos folha representam as relações básicas de um banco de dados. Os nodos internos, por sua vez, representam operações algébricas sobre essas relações básicas ou sobre *relações intermediárias*, que são o resultado de outras operações. Por fim, as arestas representam os fluxos dos dados, que partem das folhas até chegarem ao nodo raiz. A Figura 2.1 exibe um exemplo de árvore de consulta equivalente ao exemplo da Expressão 2.4.

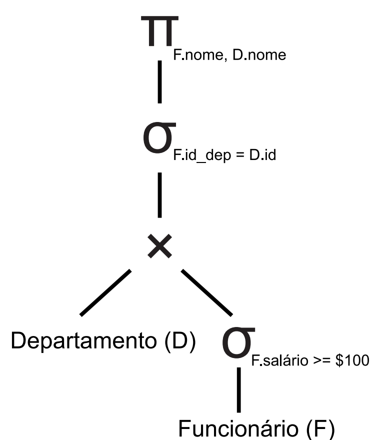


Figura 2.1: Exemplo de uma árvore de consulta.

2.1.1.2 Operador de Junção

Além dos operadores básicos já apresentados, a álgebra relacional é composta por diversos outros operadores especializados [49]. O mais importante deles para o processamento de consultas em um SGBD relacional é o operador de *junção* ou *join* (símbolo $\bowtie$). Basicamente, este é um operador binário que agrega para si um produto cartesiano seguido de uma seleção. Ou seja:

$$R \bowtie_C S \equiv \sigma_C(R \times S) \tag{2.5}$$

onde R e S são relações e C uma expressão lógica (ou *predicado de junção*) sobre atributos de ambas as relações. A notação apresentada acima é uma representação genérica de um operador de junção. Em [49], os autores definem algumas de suas variações. Neste estudo, o operador de junção utilizado refere-se ao *equi-join*, o qual assume que C seja uma condição de igualdade entre atributos de R e S.

Usando o operador de junção, a árvore de consulta da Figura 2.2 representa uma expressão algébrica equivalente ao apresentado na Figura 2.1. Embora ambas as expressões sejam equivalentes em resultado, o uso de operadores de junção por SGBDs relacionais é mais eficiente do ponto de vista computacional. O principal motivo disso é que um algoritmo que implementa um produto cartesiano entre duas relações também pode realizar simultaneamente uma seleção sem aumentar significativamente seu custo computacional [49]. Além disso, esta seleção agregada neste operador pode descartar o mais cedo possível as tuplas que não satisfazem sua condição de junção, o que evita um esforço desnecessário por parte do SGBD.

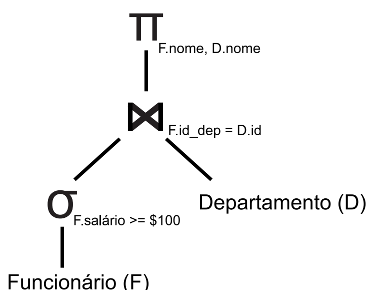


Figura 2.2: Exemplo de uma árvore de consulta usando um operador de junção.

Quanto a equivalência de expressões algébricas, os operadores de junção possuem duas propriedades principais, as quais são derivadas de propriedades existentes nos operadores de seleção e produto cartesiano [50]:

Comutativa:

$$R \bowtie_C S \equiv S \bowtie_C R \tag{2.6}$$

Associativa:

$$R \bowtie_{C_1} (S \bowtie_{C_2} T) \equiv (R \bowtie_{C_1} S) \bowtie_{C_2} T \tag{2.7}$$

A propriedade associativa apresentada acima somente é válida se o predicado de junção C_1 puder referenciar atributos de R e S. Caso contrário, a junção $R \bowtie_{C_1} S$ não pode ser considerada válida, cabendo assim apenas um produto cartesiano entre R e S ou a reescrita desta expressão usando a relação T ao invés de S:

$$R \bowtie_{C_1} (S \bowtie_{C_2} T) \equiv (R \times S) \bowtie_{C_1 \wedge C_2} T \tag{2.8}$$

ou

$$R \bowtie_{C_1} (S \bowtie_{C_2} T) \equiv (R \bowtie_{C_1} T) \bowtie_{C_2} S \tag{2.9}$$

Note que no caso da Expressão 2.8 o predicado de junção C_1 foi deslocado para junto de C_2 de forma conjuntiva, pois ambos os predicados somente poderão ser avaliados após o produto cartesiano entre R e S [50].

Uma forma bastante comum de expressar graficamente quais as relações de uma consulta possuem predicados de junção é através do uso de um grafo não orientado, denominado *grafo de junções* (ou *join graph* [43][3]). Neste grafo, os nodos representam as relações presentes em uma consulta e as arestas representam os predicados de junção entre suas respectivas relações. A Figura 2.3 é um exemplo deste tipo de grafo para uma consulta com apenas três relações e dois predicados de junção.

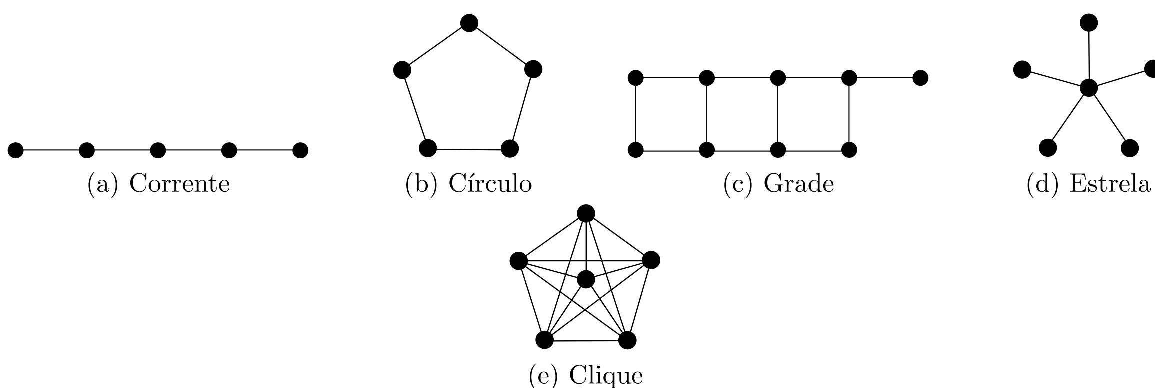
A representação de consultas na forma de grafos de junções permite que algumas análises provenientes da teoria de grafos sejam aplicadas, como por exemplo a verificação

[3] *object graph* [32], *query graph* [13, 30]

de possíveis ciclos ou árvores [32]. Quanto às possíveis formas de um grafo de junções, a Figura 2.4 apresenta cinco tipos geralmente encontrados na literatura, sendo estes os tipos corrente, círculo, grade, estrela e clique (ou grafo completo) [29, 37, 42, 43, 53].

R.x = S.y
S.w = T.k
R S T

Figura 2.3: Exemplo de um grafo de junções. Os nodos representam as relações e as arestas os predicados sobre essas relações.



(a) Corrente (b) Círculo (c) Grade (d) Estrela (e) Clique

Figura 2.4: Tipos de grafos de junções.

2.1.2 A Linguagem de Consultas SQL

Embora a álgebra relacional seja uma notação eficiente para a elaboração de consultas, sua aplicação se restringe, de um modo geral, aos mecanismos internos de um SGBD. Já para seus usuários, os SGBDs relacionais disponibilizam uma forma mais simples e produtiva de elaborá-las. A SQL (*Structured Query Language*) [12], é um exemplo de linguagem de consulta que está disponível em grande parte dos SGBDs existentes atualmente. Esta linguagem permite que uma consulta seja elaborada de forma descritiva, sem definir com isso a sequência de operações relacionais necessárias para a obtenção deste resultado [32].

Uma consulta SQL típica possui três partes básicas: (1) a lista de atributos que devem ser obtidos; (2) a lista de relações básicas que fazem parte da consulta; e (3) as condições lógicas sobre os atributos das relações listadas em (2). O Código 2.1 apresenta em notação SQL o mesmo exemplo da Expressão 2.4, sendo também equivalente à árvore de consulta da Figura 2.1.

Comparando com a álgebra relacional, a lista de atributos no Código 2.1 representa as operações de projeção que devem ser aplicadas. A cláusula "FROM", por sua vez, indica quais as relações são participantes da consulta. Caso haja mais de uma relação, deve ser considerado o uso de produtos cartesianos ou de junções, o que também depende das condições de seleção contidas na cláusula "WHERE" [50].

Código 2.1 Exemplo de consulta SQL.

```
SELECT F.nome, D.nome               -- 1. lista de atributos
FROM F, D                           -- 2. lista de relações do
                                    --    banco de dados
WHERE F.id_dep = D.id               -- 3. condições lógicas da
  AND F.salario >= 100              --    consulta
```

2.2 Arquitetura de Processamento de Consultas

Basicamente, os SGBDs relacionais precisam lidar com formas bastante distintas de representações de consultas. Como mencionado anteriormente, as consultas são recebidas por esses SGBDs na forma de uma linguagem declarativa, geralmente a SQL. Por outro lado, seu mecanismo interno de processamento é composto por diversos *métodos* que representam as operações da álgebra relacional. Existe uma grande variedade de métodos utilizados pelos SGBDs relacionais para o processamento de consultas, sendo que os principais são apresentados a seguir [51]:

Métodos de Junção: São métodos de processamento de junções ou produtos cartesianos. Esses métodos são geralmente baseados em loops aninhados (*nested-loop-join*), baseados em tabelas *hash* (*hash-join*) ou baseados em ordenação e junção (*merge-join* ou *sort-merge-join*). Cada um desses métodos podem apresentar diversas variações, o que depende da arquitetura de armazenamento e processamento do SGBD. Em [36], os autores Mishra e Eich apresentam uma visão detalhada de cada um desses métodos.

Métodos de Acesso: São métodos que definem a forma de acesso a cada relação básica

da consulta. Esses podem ser sequenciais (*seq-scan*) ou baseados em índices (*index-scan*). Tanto os métodos sequenciais como os métodos baseados em índices também dependem das características físicas de armazenamento do SGBD.

Uma vez que a linguagem SQL simplesmente descreve quais os dados precisam ser obtidos em uma consulta, um *plano de execução*, ou *plano físico*,[4] descreve a sequência de métodos necessária para a obtenção da mesma. A Figura 2.5 exemplifica, através de um modelo genérico de entidade-relacionamento, o contexto de equivalência de cada uma das representações de consultas utilizadas por um SGBD. De um modo geral, existem várias expressões algébricas que correspondem a uma mesma consulta SQL, do mesmo modo que existem diversos planos de execução para uma mesma expressão algébrica.

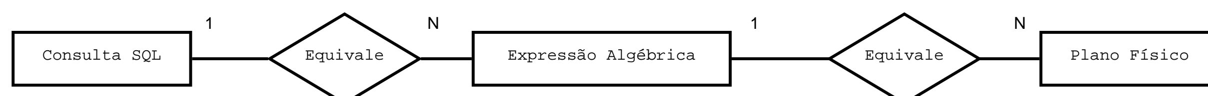


Figura 2.5: Mapeamento de equivalência entre uma consulta SQL em seus respectivos planos físicos.

Para que uma consulta SQL possa ser transformada em um plano de execução, os SGBDs relacionais implementam uma série de verificações e transformações. Embora não seja um padrão bem definido, o mecanismo de processamento de consultas desses SGBDs é geralmente composto pelas seguintes fases [28, 50]:

***Parse*:** É responsável por transformar a consulta SQL em uma representação interna, de modo que ela possa ser manipulada pelo SGBD. Além disso, uma análise sintática também é feita para garantir a corretude da consulta submetida. O resultado deste processo pode ser representado na forma de uma árvore, chamada *parse tree*, ou em qualquer outra representação que descreva melhor a forma declarativa desta consulta.

Reescrita: A fase de reescrita analisa questões semânticas da consulta com relação ao esquema de dados. Além disso, esta fase tenta reescrever a consulta submetida em

[4] *query evaluation plan* (QEP) [46], *physical plan* [50] ou *access plan* [28]

uma forma mais eficiente do ponto de vista declarativo. O resultado obtido consiste em uma representação simplificada da consulta.

Planejamento: Diferente da fase de reescrita, onde a consulta é analisada usando o ponto de vista declarativo, a fase de planejamento consiste em enumerar e avaliar os possíveis planos de execução que correspondem a esta consulta. Uma vez que existem vários planos equivalentes, o objetivo principal desta fase é encontrar um plano de execução que seja computacionalmente mais eficiente que os demais. O resultado desta fase é o plano de execução utilizado pelo SGBD para a execução da consulta.

Execução: Nesta fase, o plano selecionado na fase anterior é interpretado e executado, de modo que cada método descrito por ele seja então acionado em sua ordem determinada. Além disso, são realizadas requisições dos recursos de hardware necessários para o seu processamento, como por exemplo a reserva de memória principal e as requisições de leituras e escritas aos dispositivos de armazenamento secundário. O resultado desta fase é a consulta processada pelo SGBD.

2.2.1 O Planejamento

A fase de planejamento é a mais complexa dentre todas as fases do processamento de consultas. Basicamente, para transformar uma representação declarativa em um plano de execução, esta fase utiliza dois espaços de busca principais: o *espaço algébrico* e o *espaço de métodos e estruturas* [28]. O espaço algébrico de busca corresponde ao conjunto de possíveis expressões de álgebra relacional equivalentes a uma consulta, de modo que essas expressões são geralmente representadas por árvores de consulta. A partir de cada árvore de consulta, o espaço de métodos e estruturas corresponde aos possíveis métodos internos do SGBD que podem ser aplicados a cada operação algébrica.

Do ponto de vista computacional, cada plano de execução que corresponde a uma mesma consulta difere na quantidade de esforço exigido ao SGBD para seu processamento. A partir do conjunto de possíveis planos, o objetivo da fase de planejamento é determinar

qual deles é mais eficiente em sua execução, ou seja [13]:

$$c(s_0) = \min_{s \in S} c(s) \tag{2.10}$$

onde S representa o conjunto de possíveis planos de execução para uma mesma consulta e s_0 o plano desejado. A função $c(s)$, por sua vez, corresponde ao *custo* de cada plano avaliado, o qual serve de comparação entre os planos. Este custo é calculado com base nas estimativas de custo de cada método utilizado, as quais são obtidas a partir do *modelo de custo* e das *estatísticas* fornecidas pelo SGBD. O modelo de custo representa os diversos fatores que compõem esforço computacional de cada método implementado, como o tempo de CPU, a quantidade exigida de memória principal e o número de acessos aleatórios e sequenciais ao dispositivo de armazenamento secundário. Já as estatísticas do SGBD fornecem informações aproximadas sobre a cardinalidade e a seletividade das relações referenciadas na consulta.

2.2.1.1 Restrições do Espaço de Busca

A tarefa de encontrar o melhor plano possível, considerando todo o espaço algébrico de busca, é computacionalmente intratável mesmo para um pequeno conjunto de relações básicas [32]. Neste sentido, se fossem considerados todos os possíveis arranjos entre seleções, projeções, produtos cartesianos e junções, a fase de planejamento de consultas poderia facilmente levar mais tempo enumerando esses possíveis planos do que o tempo gasto por todas as outras fases juntas. Por outro lado, existem ainda diferenças significativas entre os custos de cada plano de execução. Em alguns casos, diferença de custo entre dois planos para uma mesma consulta pode ser de centenas de milhares de vezes, o que impede que um SGBD simplesmente escolha um plano de forma arbitrária, sem qualquer avaliação prévia.

Para restringir o espaço de busca em níveis que sejam tratáveis computacionalmente, os SGBDs utilizam diversas heurísticas. A primeira e mais importante delas, é não considerar as seleções e projeções como operações algébricas separadas, de modo que elas não possam

gerar, por si só, relações intermediárias [28]. As operações de seleção devem sempre estar agregadas ou a um produto cartesiano, na forma de junção, ou a uma relação básica, de modo que sua posição deva ser sempre a mais baixa possível na árvore de consulta [50]. As projeções também podem ser agregadas da mesma forma que as seleções, de modo a descartar atributos desnecessários sempre que possível [46]. Em outros casos, essas projeções são executadas apenas no final do plano de execução [28].

Uma forma especial de árvore de consulta que deriva da restrição sobre os operadores de seleção e projeção é a chamada de *árvore de junções*, ou *join tree*[5]. Esta árvore apresenta apenas dois tipos de nodos: as relações básicas e as operações de junção. Nesta notação, os nodos correspondentes às operações de junção também podem significar um produto cartesiano, sendo que a única diferença entre eles é a presença ou não de um predicado de junção[6]. A Figura 2.6 apresenta dois exemplos de árvores de junções, onde a árvore da direita possui um produto cartesiano entre as relações A e B.

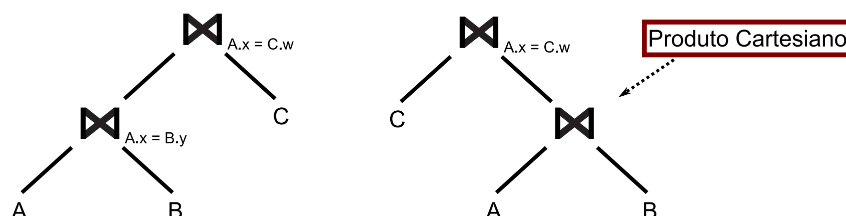


Figura 2.6: Exemplos de árvores de junções.

O uso de árvores de junções para representar as possíveis expressões algébricas a serem avaliadas na fase de planejamento permite restringir o espaço de busca a ser explorado. Contudo, esta restrição não altera a natureza combinatória do problema. Devido às propriedades comutativa e associativa dos operadores de junção e de produto cartesiano, uma consulta com N relações pode ser representada por $\frac{1}{N}\binom{2N-2}{N-1}$ tipos diferentes de árvores de junções, sendo que cada uma delas pode assumir $N!$ arranjos diferentes de relações. Desta forma, o número total de soluções para uma consulta com N relações é de $\binom{2N-2}{N-1}(N-1)!$ [35, 46]. A Tabela 2.3 apresenta número de tipos de árvores de junções

[5] Ou ainda *árvore binária de junções* (*binary join tree*) [46].

[6] Note que o símbolo "$\bowtie$", sem predicado de junção, é definido na literatura como *junção natural* [49]. A utilização desta notação para indicar produtos cartesianos serve apenas para simplificar a definição de uma árvore de junções.

e a quantidade de possíveis soluções para consultas entre 1 e 12 relações.

Tabela 2.3: Número de tipos de árvores de junções e quantidade de possíveis soluções para consultas entre 1 e 12 relações. [39, 47].

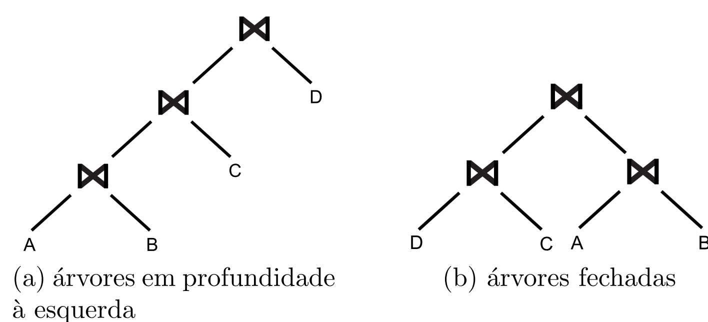
Relações N	Tipos de Árvores $\frac{1}{N}\binom{2N-2}{N-1}$	Número de Soluções Algébricas $\binom{2N-2}{N-1}(N-1)!$
1	1	1
2	1	2
3	2	12
4	5	120
5	14	1.680
6	42	30.240
7	132	665.280
8	429	17.297.280
9	1.430	518.918.400
10	4.862	17.643.225.600
11	16.796	670.442.572.800
12	58.786	28.158.588.057.600
...	...	...

A partir dessas possíveis árvores de junções, existem ainda duas outras técnicas de restrição que opcionalmente são aplicadas para reduzir ainda mais o espaço de busca a ser avaliado:

Restrição do uso de produtos cartesianos: Geralmente, a presença de produtos cartesianos em uma consulta é um fator indesejado, visto que este pode aumentar significativamente a cardinalidade das relações intermediárias. Por causa disso, em muitos SGBDs o seu uso é evitado sempre que possível. Esta restrição está fortemente relacionada com a conectividade dos grafos de junções que correspondem a cada consulta, conforme foi apresentado na Figura 2.4. Quanto menor a conectividade desses grafos, maior será a restrição imposta.

Restrição dos tipos de árvores de junções: O conjunto de *árvores fechadas de junções* (*bushy trees*) refere-se ao número de soluções apresentado na Tabela 2.3, o qual não impõe qualquer restrição quanto ao tipo de árvore a ser utilizado. Contudo, alguns dos métodos de junções implementados em SGBDs (como o *nested-loop*) apresentam uma distinção entre as relações recebidas como operandos, sendo que a

relação da esquerda é denominada *outer* e a relação da direita denominada *inner*. Esses métodos podem apresentar um desempenho maior quando a relação da direita (*inner*) for uma relação básica [13, 41, 46]. Por isso, algumas técnicas de planejamento consideram como espaço de busca apenas o tipo de árvore de junções onde todas as relações da direita são obrigatoriamente relações básicas. Este conjunto de soluções é denominado *árvores em profundidade à esquerda* (*left-deep trees*), o qual corresponde a apenas um tipo de árvore de junções e consequentemente a $N!$ possíveis soluções algébricas. A Figura 2.7 ilustra os dois conjuntos de soluções apresentados aqui.



(a) árvores em profundidade à esquerda

(b) árvores fechadas

Figura 2.7: Tipos de árvores de junções.

CAPÍTULO 3

ALGORITMOS DE OTIMIZAÇÃO DE JUNÇÕES

Um *algoritmo de otimização de junções*, ou simplesmente *otimizador*, é a parte do planejamento de consultas responsável por definir qual a melhor ordem algébrica de junções a ser utilizada para uma determinada consulta. Além dessa ordem, um otimizador deve ainda discriminar quais métodos internos do SGBD são necessários para cada uma de suas operações, definindo assim o seu plano de execução.

Conforme demonstrado por Ibaraki e Kameda [27], a tarefa ordenação de junções é de modo geral um problema NP-Completo. Mesmo considerando as restrições sobre as árvores de junções apresentadas no Capítulo 2, a quantidade de possíveis soluções é no mínimo exponencialmente proporcional à quantidade de relações presentes em uma consulta. Devido a este fato, os algoritmos propostos na literatura para a otimização de junções se enquadram em duas categorias principais:

Algoritmos Exaustivos: Este tipo de algoritmo aplica técnicas de busca exaustiva apoiadas por podas dinâmicas sobre partes de planos equivalentes. Para seu respectivo espaço de busca, um algoritmo exaustivo é capaz de retornar o melhor plano possível, ou *plano ótimo*. Contudo, devido a complexidade desse espaço de busca, sua aplicação se restringe a consultas com poucas quantidades de relações.

Algoritmos Não Exaustivos: Os algoritmos não exaustivos são geralmente aplicados para consultas em que a utilização da busca exaustiva se torna proibitiva. Esses algoritmos são geralmente apoiados por heurísticas ou por componentes aleatórios, os quais podem produzir apenas aproximações do plano ótimo.

As seções a seguir apresentam os principais algoritmos encontrados na literatura para cada uma dessas categorias.

3.1 Algoritmos Exaustivos

Um dos primeiros algoritmos para a otimização de junções encontrado na literatura foi proposto por Selinger *et al.* [41] para o System R [15], um protótipo de banco de dados relacional desenvolvido nos laboratórios da IBM durante a década de 70. O *algoritmo de programação dinâmica*, como geralmente é chamado por causa da aplicação desta técnica, é um algoritmo de busca exaustiva que explora propriedades físicas de relações intermediárias que possam influenciar no custo de operações subsequentes em um plano de execução.

Em sua proposta original, o algoritmo de programação dinâmica considera como propriedade física unicamente a ordem das relações intermediárias. Esta ordem é denominada pelos autores como "*interesting order*". Embora a ordem das relações não seja parte do espaço de busca algébrico, esta característica é muito importante para métodos de junção baseados em ordenação, como é o exemplo do *sort-merge-join*. Este método de junção primeiramente ordena as relações pelos atributos contidos no predicado de junção antes combiná-los. Caso uma ou ambas as relações já estejam ordenadas, tal operação pode ser perfeitamente desconsiderada, o que diminui significativamente o custo deste método de junção.

De um modo geral, o algoritmo de programação dinâmica funciona de acordo com os seguintes passos:

P.1 Para cada relação na consulta, todas as formas de acesso, por exemplo, acesso sequencial e por índice se existir, são obtidas. Os planos parciais (neste caso as relações básicas) são particionados em *classes de equivalência* baseadas na ordenação que seus resultados possam produzir. Uma classe de equivalência adicional é formada para os planos parciais que não produzirem resultados ordenados. As estimativas de custo são obtidas do modelo de custo e o plano mais barato de cada classe de equivalência é mantido para os passos seguintes. Contudo, o plano mais barato da classe sem ordenação não é mantido se ele não for mais barato do que todos os outros planos ordenados.

P.2 Para cada par de relações da consulta, todas as formas possíveis de junção são avaliadas usando todos os planos de acesso às relações básicas mantidos no passo P.1. Novamente, os planos parciais produzidos neste passo também são classificados conforme P.1, mantendo apenas os planos mais baratos de cada classe de equivalência.

P.i Para cada conjunto de $i-1$ relações da consulta, o passo anterior mantém o plano mais barato para "juntá-los", incluindo suas respectivas ordenações. Desta forma, este passo avalia todas as possíveis formas de adicionar mais uma relação a esses planos parciais, sem introduzir com isso produtos cartesianos. Para cada conjunto de i relações, todos os planos parciais são classificados e "podados" como no passo P.1.

P.N Neste ponto, todos os planos de execução relevantes (contidos em um único conjunto de N relações) foram gerados e mantidos no passo anterior. O plano de menor custo é finalmente escolhido e retornado pelo otimizador para ser usado na execução da consulta.

Este algoritmo sempre encontra melhor plano de seu espaço de busca, sendo que este espaço pode ser facilmente adaptado. Por exemplo, no passo P.i da descrição anterior, o algoritmo de programação dinâmica somente pode gerar árvores em profundidade à esquerda (*left-deep trees*), visto que apenas uma relação de cada vez é adicionada aos planos parciais. Na maioria dos SGBDs que implementam este algoritmo, este passo é alterado para considerar todos as combinações não sobrepostas de planos intermediários entre 1 e $i-1$ que formem um plano com i relações. O objetivo desta alteração é permitir que este otimizador trabalhe sobre o conjunto de árvores fechadas de junções (*bushy trees*).

Outra consideração importante para este algoritmo é a forma de construção dos planos intermediários. O algoritmo original, possui uma abordagem iterativa que constrói os planos de baixo para cima (*bottom-up*), onde o passo P.i somente pode ser realizado se todos os planos intermediários entre 1 e $i-1$ já estiverem formados. Uma outra abordagem foi proposta por Graefe e McKenna [24] como parte do projeto Volcano. Neste algoritmo, os planos são construídos recursivamente, ou seja, de cima para baixo (*top-down*). Uma

das vantagens desta última abordagem é que o otimizador pode conhecer o custo de planos completos de execução muito antes que na abordagem *bottom-up*. Isto possibilita que o otimizador, durante seu processo de enumeração, descarte planos parciais tão logo seu custo ultrapasse o custo do melhor plano completo encontrado até o momento [42, 53].

3.2 Algoritmos Não Exaustivos

Para a maioria dos SGBDs, a aplicação do algoritmo de programação dinâmica é restrita para consultas com até 10 ou 15 relações. Até este limite, esta técnica possui um rendimento considerável. Contudo, se aplicado para consultas maiores, a quantidade de tempo e de memória principal exigida por esse algoritmo para a avaliação de planos parciais cresce exponencialmente em função da quantidade de relações.

Os algoritmos não exaustivos são geralmente aplicados para consultas com um grande número de relações. O objetivo deste tipo de algoritmo é encontrar boas aproximações do plano ótimo, usando com isso um esforço computacional muito menor. Existem vários algoritmos propostos na literatura para esta classe de problema, os quais podem ser classificados em três grupos principais:

Determinísticos: São algoritmos geralmente guiados por alguma função heurística, os quais se beneficiam de características importantes, relacionadas a algumas classes de consultas ou aos métodos de junção utilizados em seu processamento.

Aleatórios: Para os algoritmos aleatórios, os possíveis planos de execução são vistos como pontos em um grafo não orientado. Esses pontos são conectados entre si por arestas que representam as transformações entre esses planos. O objetivo deste tipo de algoritmo é realizar séries de *movimentos* entre os nodos desse grafo em busca de soluções que sejam cada vez melhores em relação ao seu custo.

Genéticos: Os algoritmos genéticos utilizam uma estratégia de busca muito similar ao processo biológico de evolução. Em sua representação, os possíveis planos de execução são chamados *indivíduos*. A partir de uma *população*, este tipo de al-

goritmo aplica diversas combinações e mutações sobre seus indivíduos por várias *gerações*, simulando com isso um processo de seleção natural.

3.2.1 Algoritmos Determinísticos

Os algoritmos determinísticos de busca não exaustiva utilizam técnicas de heurística que permitem tratar de forma eficiente casos específicos do problema de otimização de junções, tomando por base as características de certos tipos de consulta ou o mesmo comportamento dos métodos de junção utilizados em seu processamento. Em [27], os autores Ibaraki e Kameda analisaram este processo de otimização usando exclusivamente o *nested-loop* como método de junção. Mesmo neste caso, os autores demonstraram ser um problema NP-Completo a escolha do melhor plano. Contudo, em casos mais específicos, onde as consultas não apresentavam ciclos em seus grafos de junções, os autores conseguiram apresentar um algoritmo heurístico capaz de encontrar planos ótimos em tempo $O(N^2 \log N)$. Já para as consultas cíclicas, os autores propuseram outro algoritmo capaz de encontrar apenas uma aproximação do plano ótimo em tempo $O(N^3)$.

No estudo realizado por Krishnamurthy, Boral e Zaniolo [33], foram analisadas certas características pertencentes aos métodos de junção baseados em *nested-loop* e *hash*. Com base nos resultados de sua análise, os autores propuseram um algoritmo, chamado KBZ, capaz de encontrar o plano ótimo de consultas acíclicas em tempo $O(N^2)$.

Em [25], os autores Guttoski *et al.* analisaram a aplicação do algoritmo de Kruskal [34]como forma de orientar a construção de árvores de junções. Com base nesta técnica de construção, os autores propuseram um algoritmo que usava como heurística os custos de junção associados a cada par de relações de uma consulta. Em sua avaliação, os autores demonstraram um bom comportamento desta heurística para consultas utilizadas em ambientes de apoio à decisão.

3.2.2 Algoritmos Aleatórios

Os algoritmos aleatórios são geralmente técnicas de busca aplicadas a problemas combinatórios de otimização. Para esses algoritmos, o conjunto de possíveis soluções é re-

presentado na forma de um grafo, sendo que cada nodo corresponde a uma solução (ou *estado*) e cada aresta um *movimento* válido neste grafo. Esses movimentos são definidos por regras de transformação que permitem que um estado possa ser obtido a partir de outro estado. Se um estado S pode ser alcançado a partir de um estado R com apenas um movimento, então R e S são considerados *vizinhos*.

Este tipo de grafo apresenta ainda uma notação de relevo, a qual representa a altura de cada solução em relação ao objetivo de otimização. No caso da otimização de junções, o objetivo é o melhor plano possível e a altura de cada estado é dada pelo custo de seus respectivos planos de execução. Se dois estados vizinhos possuem alturas diferentes, o movimento que tem por origem um estado mais baixo que o estado de destino é chamado de *subida* (ou *uphill*). De modo contrário, o movimento que tem por origem um estado mais alto que seu destino é chamado *descida* (ou *downhill*).

Os algoritmos aleatórios podem percorrer esse espaço de busca usando a conectividade entre seus estados. Um *caminho* entre dois estados S e T é uma sequência de movimentos que permite, a partir do estado S, alcançar o estado T. Um estado é um *mínimo local* se, para todos os caminhos partindo dele, não existir nenhum movimento de descida sem antes ocorrer um movimento de subida. Um estado está em um *plateau* se ele não possuir estados vizinhos mais baixos, porém ainda existam caminhos que levem a estados mais baixos sem a necessidade de qualquer movimento de subida. Um estado é um *mínimo global* se ele for um mínimo local e ao mesmo tempo não existir outro mínimo local de altura menor.

Encontrar o mínimo global nesse espaço de busca é o objetivo principal do processo de otimização. Uma vez que esta tarefa é extremamente custosa, é aceitável que pelo menos algum mínimo local não muito elevado seja encontrado usando uma quantidade de esforço inferior. Quanto mais baixo for o mínimo local encontrado, melhor será a qualidade da solução.

Dentre os tipos algoritmos aleatórios encontrados na literatura, os algoritmos *Melhoria Iterativa* (*Iterative Improvement* – II) e *Têmpera Simulada* (*Simulated Annealing* – SA), incluindo ainda algumas de suas derivações, são geralmente os mais estudados para o

processo de otimização de junções. A estrutura básica desses algoritmos será apresentada a seguir.

3.2.2.1 Melhoria Iterativa (II)

O algoritmo Melhoria Iterativa (II, *Iterative Improvement*) consiste de várias *otimizações locais* que são iniciadas a partir de diversos pontos do espaço de busca selecionados aleatoriamente, os quais são chamados de *estados iniciais*. A partir de cada estado inicial, este algoritmo percorre o espaço de busca de forma aleatória, sempre aceitando movimentos de descida, até encontrar um mínimo local. Este processo de otimizações locais se repete até que uma certa *condição de parada* seja alcançada. O Código 3.1 contém uma representação genérica deste algoritmo.

Código 3.1 Pseudocódigo do algoritmo Melhoria Iterativa (II) [29].

```
Função II(Q) {
  minS = S∞;
  Enquanto não(condição_de_parada) faça {
    S = estado_aleatório(Q);
    Enquanto não(mínimo_local(S)) faça {
      S’ = estado_vizinho_aleatório(S);
      Se custo(S’) < custo(S) então S = S’;
    }
    Se custo(S) < custo(minS) então minS = S;
  }
  retorne(minS);
}
```

3.2.2.2 Têmpera Simulada (SA)

Um desvantagem do algoritmo II é o fato de não permitir movimentos de subida em seu processo de otimização local. Desta forma, este processo é extremamente suscetível de ser “capturado” por qualquer mínimo local próximo ao estado inicial. Se o mínimo local selecionado possuir uma altura muito acima do mínimo global, então este não pode ser considerado como uma boa solução. Uma forma de evitar este tipo de problema é

permitir que a otimização local aceite não somente movimentos de descida, mas também movimentos de subida de forma controlada.

O algoritmo Têmpera Simulada (SA) [31,40,46] é uma abordagem que tem por objetivo evitar o problema apresentado pela otimização local do algoritmo II. O nome Têmpera Simulada, do inglês *Simulated Annealing*, refere-se analogamente ao processo físico de têmpera, geralmente aplicado para o enrijecimento de metais e vidros. Neste processo, um material é submetido a uma temperatura elevada e então sofre um resfriamento gradativo, alcançando assim um estado cristalino de baixa energia. Inspirado neste processo, o algoritmo SA emprega alguns termos, como *temperatura* e *congelamento*, que são usados para orientar o processo de otimização.

O Código 3.2 apresenta uma versão genérica do algoritmo SA. Diferente do algoritmo II, que usa vários estados iniciais aleatórios, o SA inicia a partir de um único estado (S_0). A ideia principal deste algoritmo é fazer uma espécie de "caminhada aleatória" por regiões limitadas por uma certa altura a qual diminui gradativamente à medida que o processo de otimização avança. Este método dá uma noção de diminuição gradiente do espaço de busca [40]. Durante o processo de otimização, o SA sempre aceita estados vizinhos mais baixos ($\Delta C \leq 0$). Contudo, os estados vizinhos mais altos podem ser aceitos apenas com uma certa probabilidade ($e^{-\Delta C/T}$). Durante o processo de otimização, a temperatura (T) é reduzida gradativamente, o que diminui também a probabilidade de se aceitar movimentos de subida.

A Figura 3.1 apresenta um comparativo entre o processo de otimização local do algoritmo II e o algoritmo SA. Enquanto que a otimização local fica presa no primeiro mínimo local encontrado, o SA é capaz de superar pequenos obstáculos, atingindo assim regiões mais baixas do espaço de busca.

3.2.2.3 Aplicação dos Algoritmos Aleatórios na Otimização de Junções

Uma das primeiras propostas apresentadas para o uso de algoritmos aleatórios na otimização de junções foi feita por Swami e Gupta [46]. Em seu estudo, os autores

Código 3.2 Pseudocódigo do algoritmo Têmpera Simulada (SA) [29, 46].

```
Função SA(S0) {
  S = S0;
  T = T0;
  minS = S;
  Enquanto não(congelado) faça {
    Enquanto não(equilibrio) faça {
      S' = estado_vizinho_aleatório(S);
      ΔC = custo(S') - custo(S);
      Se ΔC ≤ 0 então S = S';
      Se ΔC > 0 então S = S' com probabilidade e^(-ΔC/T);
      Se custo(S) < custo(minS) então minS = S;
    }
    T = reduzir(T);
  }
  retorne(minS);
}
```

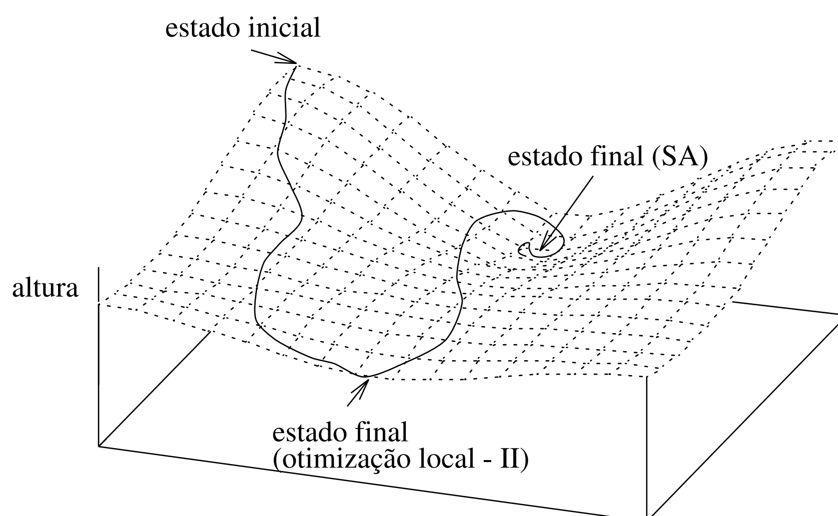


Figura 3.1: Otimização local do II *versus* SA [43].

avaliaram os algoritmos II e SA na obtenção de planos contendo somente o *hash-join* como método de junção. Além disso, os autores utilizaram como espaço de busca apenas o conjunto de árvores em profundidade à esquerda (*left-deep trees*) que não apresentassem produtos cartesianos.

Uma vez que esses algoritmos são apenas técnicas genéricas de busca, Swami e Gupta definiram ainda diversas questões relacionadas a este processo de otimização, como a forma de representar os estados no espaço de busca e como são realizadas suas transformações. Para a forma de representação das árvores de junções, os autores utilizaram listas de relações básicas, de modo que sua sequência pudesse representar o arranjo dessas relações na árvore de junções. A Figura 3.2 apresenta um exemplo dessa representação para um conjunto de quatro relações. No caso desta figura, a árvore de junções corresponde a seguinte sequência de relações:

$$(A, C, D, B) \Rightarrow ((A \bowtie C) \bowtie D) \bowtie B \tag{3.1}$$

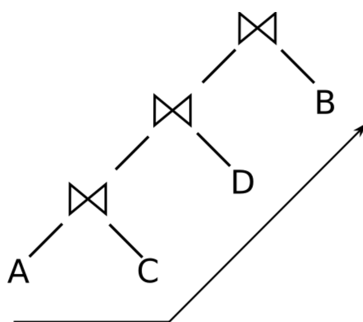


Figura 3.2: Exemplo de construção de uma árvore em profundidade à esquerda equivalente a lista de relações (A, C, D, B).

Para representar os movimentos entre estados vizinhos, os autores propuseram dois tipos de transformação:

Swap: Neste movimento, duas relações selecionadas aleatoriamente são trocadas de posição. Exemplo:

$$(A, B, C, D) \rightarrow (A, C, B, D) \tag{3.2}$$

3Cycle: Neste movimento, três relações selecionadas aleatoriamente são rotacionadas:

$$(A, B, C, D) \rightarrow (C, A, B, D) \tag{3.3}$$

Um problema encontrado na representação proposta por Swami e Gupta é a possibilidade de formar planos considerados *inválidos* por causa da presença de produtos cartesianos. Por causa disso, cada movimento realizado por esses algoritmos necessitava de uma operação de checagem para verificar se o novo plano obtido era realmente válido.

Com base nas definições feitas anteriormente, Swami e Gupta avaliaram os planos obtidos pelos algoritmos II e SA em diversas consultas acíclicas. A partir de seus resultados, os autores observaram que o algoritmo II, mesmo usando uma técnica de busca menos sofisticada, sempre gerava planos com custos melhores que os apresentados pelo algoritmo SA.

3.2.2.4 Two Phase Optimization (2PO)

Em [29], os autores Ioannidis e Kang avaliaram os algoritmos II e SA em um contexto mais amplo que o apresentado por Swami e Gupta. Neste estudo, os autores exploraram o comportamento desses algoritmos para a otimização de junções baseados nos métodos *nested-loop* e *merge-join*. Além disso, os autores utilizaram como espaço de busca o conjunto de árvores fechadas de junções (*bushy trees*), mantendo-se ainda a restrição anterior sobre produtos cartesianos. Para representar os estados neste contexto, os autores utilizaram árvores binárias, uma vez que uma simples lista de relações básicas não permitia descrever tipos diferentes de árvores de junções.

Outra mudança proposta por Ioannidis e Kang foi com relação ao conjunto de regras de transformações utilizado para definir os movimentos neste espaço de busca. Ao todo, foram apresentadas cinco regras que levavam em consideração tanto a troca de métodos de junções como suas propriedades comutativas e associativas:

- Troca do método de junção: $A \bowtie^{nested\text{-}loop} B \rightarrow A \bowtie^{merge\text{-}join} B$

- Comutatividade: $A \bowtie B \rightarrow B \bowtie A$
- Associatividade: $(A \bowtie B) \bowtie C \rightarrow A \bowtie (B \bowtie C)$
- Troca da junção da esquerda: $(A \bowtie B) \bowtie C \rightarrow (A \bowtie C) \bowtie B$
- Troca da junção da direita: $A \bowtie (B \bowtie C) \rightarrow B \bowtie (A \bowtie C)$

A partir desta definição, os autores avaliaram os planos de execução obtidos pelos algoritmos II e SA. Os autores constataram que o II era capaz de gerar planos com custos mais baixos que o SA em um tempo relativamente curto. Porém, somente após um longo período de tempo é que o SA passava a superar o II com relação ao custo de seus planos.

Ioannidis e Kang avaliaram ainda a forma do espaço de busca produzida por sua representação. Eles observaram que em algumas regiões, havia uma grande concentração de mínimos locais de custo baixo muito próximos uns dos outros, separados apenas por pequenos obstáculos. Segundo os autores, o formato produzido por essas regiões se assemelhava a um "poço" ou "copo", conforme ilustrado na Figura 3.3.

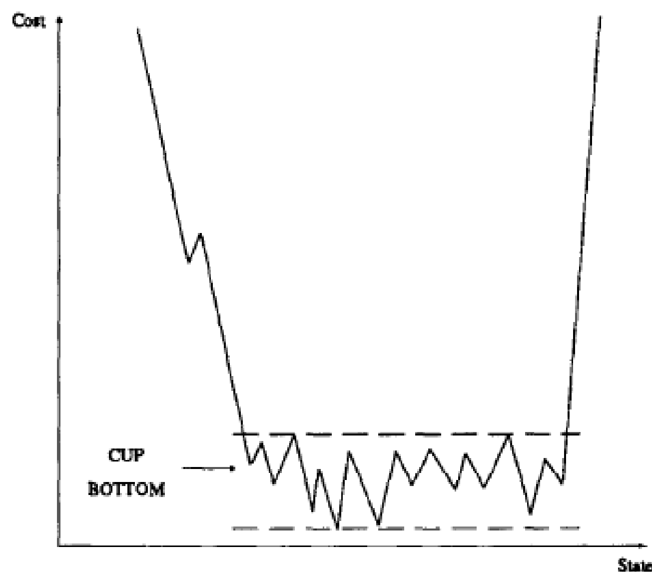


Figura 3.3: Forma do espaço de busca de árvores fechadas de junções apresentada por Ioannidis e Kang [29].

A partir dessas observações, Ioannidis e Kang propuseram um outro algoritmo que buscasse aproveitar, de forma mais eficiente, as características apresentadas pelos algoritmos II e SA nesse espaço de busca. Este algoritmo, chamado *Two Phase Optimization* (2PO),

é composto por duas fases principais, que correspondem aos dois algoritmos apresentados acima. Na primeira fase, o algoritmo II é acionado por um determinado número de iterações, permitindo assim encontrar algum mínimo local de custo relativamente baixo. Após o término desta fase, o mínimo local selecionado é então utilizado como estado inicial para o algoritmo SA. Nesta implementação, o algoritmo SA utiliza temperatura inicial (T_0) relativamente baixa, o que permite a ele explorar outros mínimos locais próximos sem efetuar com isso grandes subidas no espaço de busca.

3.2.3 Algoritmos Genéticos

Os algoritmos genéticos formam outra classe de algoritmos não exaustivos que pode ser aplicada ao problema de otimização de junções. Proposto por John Holland e acadêmicos da Universidade de Michigan, este tipo de algoritmo tem seu método de otimização inspirado no processo biológico de seleção natural [23]. De um modo geral, sua técnica difere das técnicas apresentadas pelos algoritmos II e SA. Em vez de considerar apenas uma solução por vez, aplicando a esta várias regras de transformação para produzir novas soluções, os algoritmos genéticos são capazes de trabalhar com diversas soluções simultaneamente. Deste modo, novas soluções são obtidas a partir de uma combinação das características de soluções já existentes.

Como o seu próprio nome sugere, os algoritmos genéticos utilizam uma terminologia bastante familiar ao estudo da genética de seres vivos. Cada possível solução do problema é denominada *indivíduo*, sendo este geralmente representado por um ou mais *cromossomos*. Os cromossomos, por sua vez, são o conjunto de características que compõem o indivíduo. Essas características são chamadas de *genes*, os quais podem assumir diferentes valores (*alelos*) e estarem em diferentes *posições* dentro de um cromossomo [43].

O princípio básico de um algoritmo genético é produzir aleatoriamente uma *população* inicial de indivíduos, da qual se permita cultivá-los por várias *gerações*. Em cada geração, os melhores indivíduos são *selecionados* de acordo com o seu grau de *adaptação* ao ambiente (*fitness*) para formarem a base de uma nova geração. Uma fração desses indivíduos é então *combinada* (*crossover*) para formarem novos *descendentes*, enquanto que outros

indivíduos podem sofrer *mutações*. Este processo de seleção, combinação e mutação continua até que algum dos seguintes critérios seja alcançado: (a) quando for possível obter um indivíduo que esteja perfeitamente adaptado ao ambiente; (b) quando for atingido um determinado número de gerações; ou (c) quando não for possível obter novas melhorias na população.

No caso da otimização de junções, os indivíduos correspondem aos planos de execução, enquanto que os custos desses planos correspondem ao seu grau de adaptação ao ambiente. Assim como no espaço de busca dos algoritmos II e SA, o ambiente onde os indivíduos pertencem corresponde ao conjunto de possíveis soluções do problema. A partir deste, os cromossomos devem ser codificados de forma adequada e as ações de combinação e mutação definidas. Em [13], os autores Bennett *et al.* apresentam duas implementações de algoritmos genéticos para a otimização de junções, uma para o conjunto de árvores em profundidade à esquerda e outra para o conjunto de árvores fechadas de junções.

3.2.3.1 Árvores em Profundidade à Esquerda

Para o conjunto de árvores em profundidade à esquerda, os cromossomos foram representados por listas de relações associadas com seus respectivos métodos de junção:

$$S = (^m A,^n C,^n B,^m D,^n F,^m E) \tag{3.4}$$

Esta forma é semelhante ao apresentado por Swami e Gupta [46], de modo que sua construção é idêntica ao apresentado na Figura 3.2. A única diferença neste caso é que Bennett *et al.* consideram dois métodos de junção, *nested-loop* e *merge-join*. Esses métodos estão indicados respectivamente pelas letras n e m ao lado esquerdo de cada relação. Nesta forma de representação, somente os métodos de junção presentes a partir da segunda relação (gene) desta lista são realmente utilizados.

Para representar as mutações sobre os indivíduos, Bennett *et al.* utilizam ainda a mesma técnica de *swap* apresentada por Swami e Gupta [46], de modo que dois genes selecionados aleatoriamente são trocados de posição. Assim como definido para os algo-

ritmos II e SA, esta operação também foi equipada com um mecanismo de checagem para evitar a presença de produtos cartesianos.

Para as operações de *crossover*, Bennett *et al.* definem dois métodos, *M2S* e *CHUNK*. O método M2S consiste em selecionar aleatoriamente dois genes de um cromossomo K e substituí-los pelos genes correspondentes de um outro cromossomo L, incluindo sua respectiva ordem. Exemplo:

$$\left.\begin{array}{rcl} K & = & ({}^{\mathbf{m}}\mathbf{A}, {}^{n}C, {}^{m}B, {}^{\mathbf{m}}\mathbf{D}, {}^{n}F, {}^{n}E) \\ L & = & ({}^{m}B, {}^{n}C, {}^{\mathbf{m}}\mathbf{D}, {}^{m}F, {}^{m}E, {}^{\mathbf{n}}\mathbf{A}) \end{array}\right\} \Rightarrow ({}^{\mathbf{m}}\mathbf{D}, {}^{n}C, {}^{m}B, {}^{\mathbf{n}}\mathbf{A}, {}^{n}F, {}^{n}E) \tag{3.5}$$

onde os genes que correspondes às relações A e D são selecionados do cromossomo K para serem substituídos pelos seus respectivos genes presentes no cromossomo L.

No método *CHUNK*, os genes são selecionados a partir de pedaços contínuos de cromossomos. Para um cromossomo de tamanho l, os pedaços são selecionados aleatoriamente a partir de uma posição inicial que vai de 0 até $l/2$ e tem por comprimento entre $l/4$ e $l/2$ genes. Esta operação de *crossover* consiste em extrair um pedaço de um cromossomo K e introduzí-lo em um outro cromossomo L, conforme o exemplo a seguir:

$$\left.\begin{array}{rcl} K & = & ({}^{n}A, {}^{n}C, {}^{\mathbf{m}}\mathbf{D}, {}^{\mathbf{n}}\mathbf{F}, {}^{n}E, {}^{m}B) \\ L & = & ({}^{\mathbf{m}}\mathbf{F}, {}^{n}C, {}^{m}B, {}^{m}E, {}^{m}A, {}^{\mathbf{n}}\mathbf{D}) \end{array}\right\} \Rightarrow ({}^{n}C, {}^{m}B, {}^{\mathbf{m}}\mathbf{D}, {}^{\mathbf{n}}\mathbf{F}, {}^{m}E, {}^{m}A) \tag{3.6}$$

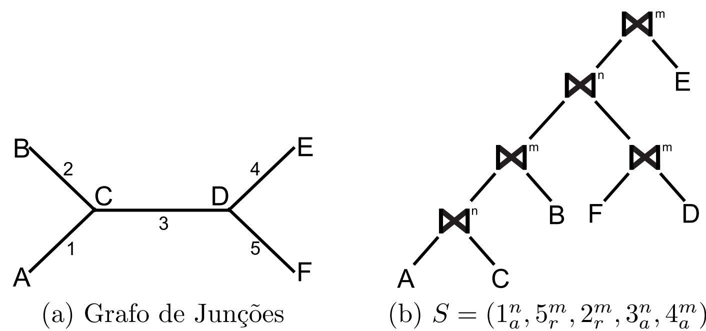
neste caso, os genes ${}^{m}D$ e ${}^{n}F$ do cromossomo K foram introduzidos no cromossomo L na mesma posição em que eles pertenciam em K. Além disso, os genes ${}^{m}F$ e ${}^{n}D$ do cromossomo L foram excluídos de modo que os demais genes de pudessem ser deslocados para esses espaços livres.

3.2.3.2 Árvores Fechadas de Junções

Para representar o conjunto de árvores fechadas de junções, os autores Bennett *et al.* utilizam como abordagem a ordenação das arestas de um grafo de junções. Neste caso, cada gene assume uma formação k_o^J, onde k é um número que identifique a aresta no grafo, J é o método de junção utilizado e o é o indicador de ordem das relações na

junção. O indicador de ordem refere-se à propriedade comutativa da operação de junção, o qual define se as relações deverão estar ordenadas alfabeticamente (a) ou inversamente ordenadas (r).

A partir do grafo de junções da Figura 3.4a, a Figura 3.4b apresenta um exemplo de árvore de junções que corresponde ao cromossomo $S = (1^n_a, 5^m_r, 2^m_r, 3^n_a, 4^m_a)$. A construção dessa árvore é feita de forma gradativa por diversas subárvores, as quais vão sendo adicionadas e posteriormente unidas à medida que os genes são avaliados da esquerda para a direita. O primeiro gene avaliado (1^n_a) corresponde a primeira subárvore deste processo: $A \bowtie^n C$. Em seguida, a junção $F \bowtie^m D$, que corresponde ao gene 5^m_r, é colocada em uma subárvore separada, visto que esta subárvore não possui intersecção com a primeira. No terceiro gene (2^m_r), a junção $C \bowtie^m B$ apresenta uma intersecção com a subárvore $A \bowtie^n C$. Deste modo, a relação C da junção $C \bowtie^m B$ é substituída pela subárvore $A \bowtie^n C$, ou seja: $(A \bowtie^n C) \bowtie^m B$. Este processo continua até todos os genes serem avaliados, de modo a existir apenas uma árvore completa de junções.



(a) Grafo de Junções (b) $S = (1^n_a, 5^m_r, 2^m_r, 3^n_a, 4^m_a)$

Figura 3.4: Exemplo (a) grafo e (b) árvore de junções utilizado por Bennett *et al.* [13] para representar cromossomos em um ambiente de árvores fechadas.

Para as operações de *crossover*, Bennett *et al.* utilizam os mesmos métodos *M2S* e *CHUNK* apresentados anteriormente. Já para as operações de mutação, além da forma anteriormente proposta, os autores definem ainda uma segunda operação de mutação. Nesta, um gene selecionado aleatoriamente sofre uma alteração em seu método de junção ou na orientação de suas relações.

CAPÍTULO 4

OTIMIZAÇÃO DE JUNÇÕES NO POSTGRESQL

Com base nos conceitos apresentados nos capítulos anteriores, este capítulo busca descrever a arquitetura e o processo de otimização de junções do PostgreSQL, dando maior ênfase aos algoritmos avaliados neste estudo: o *Genetic Query Optimization* (GEQO), o qual atualmente faz parte deste SGBD, e o *Two Phase Optimization* (2PO), implementado durante a realização deste estudo. Embora a versão estável mais recente do PostgreSQL seja a 8.4, a maior parte da investigação de seu código fonte e do posterior desenvolvimento do algoritmo 2PO foram realizados antes de seu lançamento. Por este motivo, as informações aqui apresentadas referem-se mais precisamente à versão 8.3.

4.1 Arquitetura

A arquitetura utilizada pelo PostgreSQL para o processamento de consultas está organizada de forma semelhante ao apresentado no Capítulo 2. Conforme ilustrado na Figura 4.1, as principais fases deste processo são o *parse*, a reescrita, o planejamento e a execução:

1. A fase de *parse* tem por entrada uma consulta SQL submetida pelo usuário em sua forma textual. Esta consulta passa então por uma verificação de sintaxe, sendo posteriormente convertida para uma estrutura interna chamada *parse tree*. Esta estrutura deve associar cada item do banco de dados referenciado na consulta.

2. A fase de reescrita verifica se os itens referenciados na fase anterior podem ser reescritos em uma *parse tree* mais eficiente ou mesmo se existem declarações implícitas que podem ser acrescentadas a ela. Tais verificações incluem: (a) conversão de visões (*views*) em suas respectivas declarações; (b) adição de predicados implícitos em *classes de equivalência*; e (c) substituição consultas aninhadas (sub-consultas)

por formas equivalentes não aninhadas.

3. A fase de planejamento deve receber a *parse tree*, reescrita na fase anterior, e escolher um plano eficiente para sua execução, tomando por base as estatísticas fornecidas pelo banco de dados e o custo de cada possível método a ser utilizado.

4. Por fim, a fase de execução deve processar o plano de execução fornecido pela fase de planejamento, buscando do banco de dados as tuplas que são correspondentes com a consulta solicitada pelo usuário.

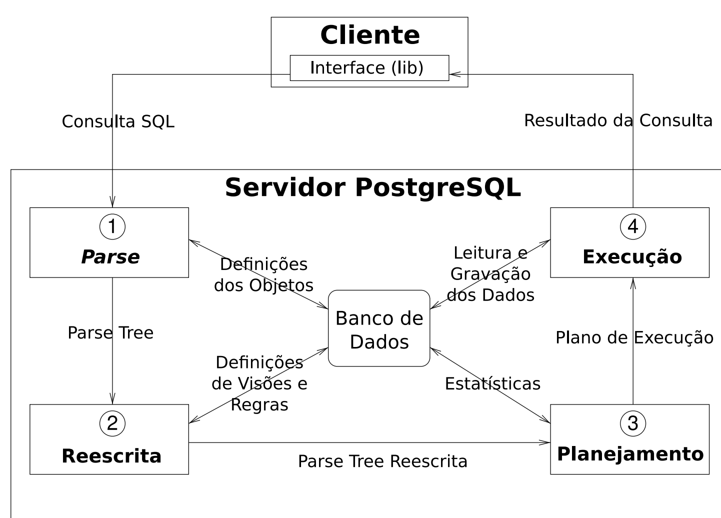


Figura 4.1: Arquitetura de processamento de consultas do PostgreSQL [1].

A fase de planejamento do PostgreSQL concentra-se principalmente na definição de uma ordem eficiente de junções. Este processo é realizado atualmente por dois algoritmos de otimização: o de programação dinâmica, também denominado *algoritmo padrão*, e o GEQO. A seleção desses algoritmos nesta fase de planejamento depende unicamente da quantidade de relações presentes em uma consulta. Por padrão, o algoritmo de programação dinâmica é executado sempre que uma consulta possuir até 11 relações. Acima deste limite, o algoritmo GEQO deve ser acionado.

O PostgreSQL possui uma estrutura modular bem definida em seu processo de otimização de junções. Conforme será apresentado a seguir, algumas estruturas de dados e operações básicas permitem abstrair os detalhes da construção dos planos de execução dos seus respectivos algoritmos de otimização.

4.1.1 A Estrutura *RelOptInfo*

O *RelOptInfo* é uma estrutura de dados que tem por objetivo auxiliar os algoritmos de otimização de junções em seu processo de elaboração de planos de execução. A função básica desta estrutura é agregar uma lista de planos (*pathlist*) que correspondem a um mesmo conjunto de relações (*relids*). A Figura 4.2 ilustra de forma simplificada um *RelOptInfo*, contendo como exemplo dois planos de execução para um conjunto de duas relações básicas $\{A, B\}$. Os retângulos contidos nesta figura representam estruturas instanciadas em posições distintas de memória, enquanto que as setas correspondem a referências para essas posições.

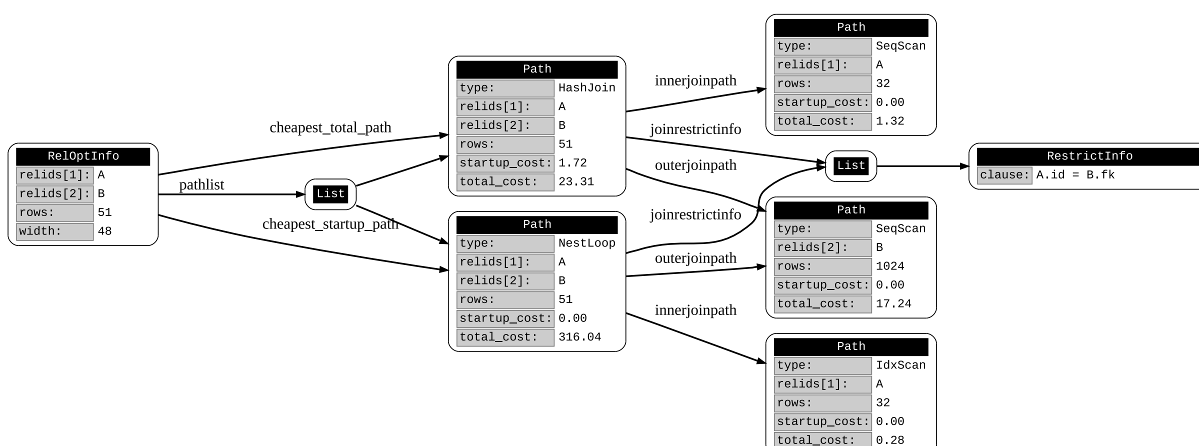


Figura 4.2: Exemplo de uma estrutura *RelOptInfo* utilizada pelo PostgreSQL para agregar vários planos de execução de um mesmo conjunto de relações.

Os planos contidos em um *RelOptInfo* possuem a forma de árvore, onde cada um de seus nodos é chamado *Path*. Esses nodos representam os métodos internos do SGBD, podendo assumir tanto a forma de junções (*NestLoop*, *MergeJoin* ou *HashJoin*) como a forma de métodos de acesso às relações básicas (*SeqScan* ou *IdxScan*)[1]. Assim como em uma árvore de junções, os nodos que correspondem a métodos de junção devem possuir ponteiros para outros dois nodos filhos, *innerjoinpath* e *outerjoinpath*. Por sua vez, os nodos que representam relações básicas estão sempre posicionados nas folhas desta árvore. Nesta estrutura, existem ainda nodos auxiliares, chamados *RestrictInfo*, os quais

[1]Existem ainda outros tipos de métodos utilizados pelo PostgreSQL, como materializações de resultados intermediários em memória secundária, agregações e ordenações. Esses métodos não serão avaliados neste estudo.

representam os predicados que devem ser considerados pelos nodos *Path* durante seu processamento.

Os nodos *Path* também armazenam informações sobre a quantidade de tuplas (*rows*) que se espera produzir em seu respectivo método bem como suas estimativas de custo (*startup_cost* e *total_cost*). O atributo *startup_cost*, ou custo inicial, corresponde ao custo estimado de um nodo, incluindo seus respectivos nodos filhos, até o momento imediatamente anterior ao retorno da primeira tupla. Este custo é geralmente consequência de operações prévias de ordenação ou de *hash*. Já o atributo *total_cost*, ou custo total, corresponde à estimativa de custo apresentada por um nodo para o retorno de todas as suas tuplas. Assim como para o atributo *startup_cost*, este também inclui os custos de seus nodos filhos, quando existirem.

Tanto o custo inicial como o custo total de um plano de execução podem ser obtidos pelos respectivos atributos *startup_cost* e *total_cost* do nodo *Path* que corresponde à raiz desta árvore. A partir de sua lista de planos (*pathlist*), um *RelOptInfo* possui ainda duas outras referências para esses planos, *cheapest_startup_cost* e *cheapest_total_cost*, as quais correspondem respectivamente aos planos de menor custo inicial e menor custo total contidos nesta estrutura. Note na Figura 4.2 que nem sempre um plano com menor custo inicial também implica em um custo total mais baixo.

4.1.2 Contrato e Operações Básicas de um Otimizador

Para um otimizador implementado no PostgreSQL, várias instâncias de *RelOptInfo*'s são utilizadas para representar os diversos estágios de seu processo de otimização. Esses *RelOptInfo*'s podem ser classificadas em três categorias:

Básicos: Representam a entrada do problema de otimização, sendo também as menores unidades manipuláveis neste processo. Cada *RelOptInfo* básico pode referenciar tanto uma relação básica como uma sub-consulta proveniente de etapas anteriores de otimização.

Intermediários: Representam os estágios intermediários do processo de otimização.

Cada *RelOptInfo* intermediário é resultado da combinação de outros *RelOptInfo*'s básicos e/ou intermediários, porém, não possuem planos completos de execução.

Completos: Representam as possíveis soluções do problema. Um *RelOptInfo* completo agrega todos os *RelOptInfo*'s básicos fornecidos ao otimizador.

O processo de geração de um *RelOptInfo* completo depende de sucessivas combinações entre *RelOptInfo*'s básicos e intermediários. Dependendo da estratégia de busca adotada, cada otimizador pode gerar vários *RelOptInfo*'s até que seja possível obter um plano de execução de custo satisfatório.

A combinação entre *RelOptInfo*'s não é realizada diretamente por um otimizador, mas internamente pelo PostgreSQL através de uma função chamada *make_join_rel*. Esta função assume dois *RelOptInfo*'s como entrada, dos quais é produzido um novo *RelOptInfo* contendo as possíveis formas de junção entre cada elemento de suas respectivas *pathlist*'s. Esta operação será representada da seguinte forma:

$$\textit{make_join_rel}(\{A\}, \{B\}) \Rightarrow \{A, B\} \tag{4.1}$$

onde $\{A\}$ e $\{B\}$ representam *RelOptInfo*'s para as respectivas relações básicas A e B, enquanto que $\{A, B\}$ corresponde ao *RelOptInfo* produzido pela função *make_join_rel*. Esta função considera ainda a propriedade comutativa das operações de junção. Ao realizar a combinação de $\{A\}$ e $\{B\}$, cada método de junção avaliado também leva em conta a melhor ordem de seus operandos *inner* e *outer*. Deste modo, o *RelOptInfo* $\{A, B\}$ também equivale a $\{B, A\}$, o qual pode conter tanto junções que correspondam a $A \bowtie B$ como junções que correspondam a $B \bowtie A$.

Para produzir *RelOptInfo*'s com mais de duas relações, um otimizador precisa usar a função *make_join_rel* mais de uma vez, conforme o exemplo a seguir:

$$\begin{aligned} \textit{make_join_rel}(\{A\}, \{B\}) &\Rightarrow \{A, B\} \\ \textit{make_join_rel}(\{A, B\}, \{C\}) &\Rightarrow \{\{A, B\}, C\} \end{aligned} \tag{4.2}$$

ou

$$\begin{aligned} make_join_rel(\{A\},\{C\}) &\Rightarrow \{A,C\} \\ make_join_rel(\{A,C\},\{B\}) &\Rightarrow \{\{A,C\},B\} \end{aligned} \tag{4.3}$$

ou ainda

$$\begin{aligned} make_join_rel(\{B\},\{C\}) &\Rightarrow \{B,C\} \\ make_join_rel(\{B,C\},\{A\}) &\Rightarrow \{\{B,C\},A\} \end{aligned} \tag{4.4}$$

Note que todos os três exemplos acima correspondem ao mesmo conjunto de relações. A partir desta representação, um otimizador deve se concentrar principalmente na propriedade associativa das operações de junção, enquanto que a comutatividade é considerada implicitamente pela função *make_join_rel*.

4.2 O Algoritmo GEQO

O GEQO (*Genetic Query Optimization*) é um algoritmo genético de otimização de junções que foi implementado para o PostgreSQL em 1997 por Martin Utesch. Assim como outras técnicas não exaustivas de busca descritas no Capítulo 3, este algoritmo é utilizado sempre que for impraticável a utilização do algoritmo de programação dinâmica deste SGBD. Esta seção busca trazer algumas características do algoritmo GEQO, sendo estas extraídas de sua documentação e código fonte [7, 52].

4.2.1 Espaço de Busca

A implementação do GEQO deriva diretamente de diversos algoritmos propostos por Whitley *et al.* para um projeto chamado GENITOR [54,55], o qual corresponde a uma variação de algoritmo genético. O conceito básico deste algoritmo de otimização de junções é de certa forma análogo ao problema do Caixeiro Viajante. Este é um problema NP-Completo clássico, no qual existem várias cidades que são interligadas por rodovias e deseja-se encontrar o menor caminho (*tour*) que passe por todas essas cidades [20]. A partir desta concepção, o GEQO utiliza como genes (cidades) os *RelOptInfo*'s básicos fornecidos pelo PostgreSQL como entrada do problema de otimização. Por sua vez, um

cromossomo (*tour*) refere-se a uma possível ordem de combinações entre esses *RelOptInfo*'s, usando para isso a função *make_join_rel*.

A codificação dos cromossomos utilizada pelo GEQO é semelhante às listas de relações básicas utilizadas para representar a construção de árvores em profundidade à esquerda, conforme apresentadas por Swami e Gupta [46] e por Bennett *et al.* [13]. Contudo, como o GEQO utiliza *RelOptInfo*'s ao invés dessas relações básicas, a notação mais adequada é a seguinte:

$$(\{C\}, \{B\}, \{D\}, \{A\}) \Rightarrow \{\{\{C, B\}, D\}, A\} \tag{4.5}$$

Como mencionado anteriormente, a combinação entre *RelOptInfo*'s através da função *make_join_rel* é mais abrangente do que uma simples junção entre relações. Uma vez que essas operações de combinação são implicitamente comutativas, o espaço de busca utilizado pelo GEQO não engloba somente árvores em profundidade à esquerda, mas todo o conjunto de árvores em profundidade.

Do mesmo modo que outros algoritmos de otimização apresentados no capítulo anterior, o GEQO tenta restringir seu espaço de busca considerando, sempre que possível, apenas os planos que não apresentem produtos cartesianos. Uma vez que este fato depende apenas da associatividade entre as junções, é possível que alguns cromossomos apresentem falhas de combinação entre *RelOptInfo*'s, ou seja, a ausência de predicados de junção. Para esses casos, o GEQO tenta utilizar uma ordem alternativa de combinação ao invés de simplesmente considerar esses cromossomos como inválidos. Sempre que um possível produto cartesiano é encontrado em um cromossomo, o GEQO tenta criar uma ordem paralela de combinações, sendo esta novamente incorporada à ordem principal somente quando existir um predicado de junção compatível com a combinação dessas duas ordens. Este processo é parecido com a criação de uma árvore fechada de junções composta por subárvores em profundidade, conforme ilustrado na Figura 4.3.

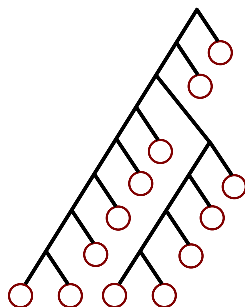


Figura 4.3: Exemplo de uma árvore alternativa utilizada pelo GEQO para representar cromossomos que contenham falhas em sua sequência normal de combinações.

4.2.2 Seleção, Crossover e Evolução

A abordagem proposta por Whitley *et al.* para o projeto GENITOR, apresenta algumas diferenças com relação aos algoritmos genéticos tradicionais [54]. Uma de suas principais característica é a não produção de uma nova população a cada geração avaliada. Em vez disso, a população inicial, gerada aleatoriamente, é realimentada com novos indivíduos à medida que eles são produzidos. Para este tipo de algoritmo, cada geração produz apenas um novo indivíduo, o qual é resultado da recombinação (*crossover*) de dois outros indivíduos selecionados desta população. Este novo indivíduo é então adicionado à população, a qual sofre uma reordenação de seus membros de acordo com os seus respectivos graus de adaptação (*fitness*). Para manter o mesmo tamanho da população inicial, a adição de um novo membro também implica na eliminação de outro, o qual deve ser sempre o pior membro desta população.

Para o GEQO, o grau de adaptação de um indivíduo depende do menor custo total (*cheapest_total_cost*) de seu respectivo *RelOptInfo*. Este valor é utilizado como critério de ordenação da população, a qual deve assumir uma forma não decrescente. A partir de uma população já ordenada, o mecanismo de seleção do GEQO utiliza uma função chamada *linear*, a qual deve retornar a posição de um indivíduo que fará parte do processo de recombinação [7, 54]. A equação abaixo descreve como esta posição é calculada:

$$\text{linear}(P, B) = P * \frac{B - \sqrt{B^2 - 4(B-1) * \text{random}()}}{2(B-1)} \tag{4.6}$$

onde B refere-se ao parâmetro de configuração do PostgreSQL chamado

geqo_selection_bias, o qual deve estar entre 1.5 e 2.0 (2.0 por padrão). O parâmetro P, por sua vez, representa o tamanho da população e a função *random()* corresponde a um número aleatório entre 0 e 1 inclusive.

O processo de recombinação de indivíduos adotado pelo GEQO é realizado por um algoritmo chamado "*edge recombination crossover*" (ERX) [7, 55]. De forma análoga ao problema do Caixeiro Viajante, os caminhos entre as cidades, que representam os dois cromossomos selecionados, são transformados em dois grafos circulares não orientados. Os grafos desses cromossomos são então combinados na forma de listas de adjacências, de modo que cada cidade possua uma lista de cidades alcançáveis por ela. O conjunto dessas listas de adjacências é chamado *edge_table*. No processo de criação da *edge_table*, as adjacências em comum a ambos os cromossomos são identificadas, sendo elas utilizadas preferencialmente pelo algoritmo.

A partir de uma *edge_table* formada pela combinação dos cromossomos selecionados, o algoritmo ERX seleciona um novo caminho, utilizando para isso os cinco passos descritos a seguir:

1. Escolha aleatoriamente uma das cidades, a qual deve ser chamada de *cidade atual*.

2. Remova da lista de adjacência de todas as outras cidades a referência para a *cidade atual*.

3. Se a *cidade atual* não possuir adjacências, vá para o passo 5, caso contrário continue no passo seguinte.

4. Da lista de adjacência da *cidade atual*, escolha aquela que represente um caminho em comum entre os dois cromossomos pais. Caso não exista, escolha uma cidade adjacente que contenha o menor número de adjacências, selecionando aleatoriamente os casos empatados. Neste ponto, a cidade selecionada passa a ser a nova *cidade atual* e o algoritmo deve retornar ao passo 2.

5. Se a *cidade atual* não possuir adjacências mas ainda restarem cidades a serem visitadas, escolha aleatoriamente uma dessas cidades para se tornar a nova *cidade atual*

e retorne ao passo 2. Caso contrário, o processo de recombinação está pronto, sendo que a ordem em que as cidades (genes) foram visitadas deve ser codificada na forma de um cromossomo.

Além das técnicas de seleção e recombinação apresentadas acima, o funcionamento do GEQO depende ainda do tamanho da população inicial e do número de gerações aplicadas em seu processo de otimização. Os valores que podem ser assumidos pelo GEQO para cada um desses casos depende de dois fatores: número de relações (N) e um parâmetro de esforço de otimização chamado *geqo_effort*, o qual pode variar entre 1 e 10. As equações descritas abaixo representam a forma com que o tamanho da população e o número de gerações é calculado. Por questões de simplificação o parâmetro *geqo_effort* foi substituído pela letra E.

$$f(N) = 2^{N+1} \tag{4.7}$$

$$\text{população}(N, E) = \begin{cases} f(N), & \text{se } 10E \leq f(N) \leq 50E \\ 10E, & \text{se } f(N) < 10E \\ 50E, & \text{se } f(N) > 50E \end{cases} \tag{4.8}$$

$$\text{gerações}(N, E) = \text{população}(N, E) \tag{4.9}$$

Por padrão, o GEQO adota o valor 5 para o parâmetro *geqo_effort*, o qual limita a função *população*$(N, 5)$ em um intervalo que vai de 50 até 250. Este intervalo implica ainda em um comportamento constante desta função para $N \leq 4$ ou $N \geq 7$.

Além das formas apresentadas acima para calcular o tamanho da população e o número de gerações, o GEQO permite que valores constantes possam ser atribuídos para cada um desses fatores. Esta configuração pode ser feita pelos respectivos parâmetros *geqo_pool_size* e *geqo_generations*.

4.3 A Implementação do Algoritmo 2PO

Como forma de comparar a estratégia de otimização apresentada pelo GEQO, este estudo buscou implementar outro algoritmo não exaustivo aplicável a este tipo de problema. A partir dos algoritmos apresentados no Capítulo 3, decidiu-se utilizar o 2PO, proposto por Ioannidis e Kang [29]. Este algoritmo, demonstrou ser de uma técnica mais elaborada que os demais, tanto em sua capacidade de administrar o espaço de busca de árvores fechadas de junções como em sua técnica de refinamento gradativo das possíveis soluções em função de seu tempo de otimização.

A implementação do 2PO foi realizada em linguagem C, usando para isso as estruturas e funções fornecidas pelo próprio PostgreSQL. Conforme proposto por Ioannidis e Kang, a implementação deste algoritmo nada mais é do que a implementação dos algoritmos Melhoria Iterativa (II) e Têmpera Simulada (SA), os quais foram implementados seguindo os pseudocódigos já apresentados nos Códigos 3.1 e 3.2. O Código 4.1 e a Tabela 4.1 apresentam o funcionamento geral do algoritmo 2PO, incluindo os parâmetros de configuração aplicados para cada uma de suas fases de otimização.

Código 4.1 Pseudocódigo do algoritmo 2PO, representando as chamadas para as fases II e SA.

```
Função 2PO(Q) {
  S0 = II(Q);
  minS = SA(S0)
  retorne(minS);
}
```

4.3.1 Espaço de Busca

Para representar os possíveis estados no espaço de busca de árvores fechadas de junções, os autores Ioannidis e Kang utilizaram árvores binárias. Nessas árvores, cada um de seus nodos de junção deveria discriminar ainda os respectivos métodos utilizados. Uma vez que a arquitetura adotada pelo PostgreSQL se baseia na estrutura *RelOptInfo*, tais métodos de junção são considerados implicitamente, o que torna sua identificação

Tabela 4.1: Parâmetros de configuração utilizados pelo 2PO em cada uma de suas fases, conforme proposto por Ioannidis e Kang [29].

Parâmetro	Fase	Valor
estado_inicial (S_0)	II	aleatório
condição_de_parada	II	10 otimizações locais
estado_inicial (S_0)	SA	melhor mínimo local obtido em II
temperatura inicial (T_0)	SA	$0,1 * \text{custo}(S_0)$
congelamento	SA	$T < 1$ e minS não alterado por 4 estágios
equilibrio	SA	$16 * N$ iterações
redução de temperatura	SA	$T = 0,95 * T$

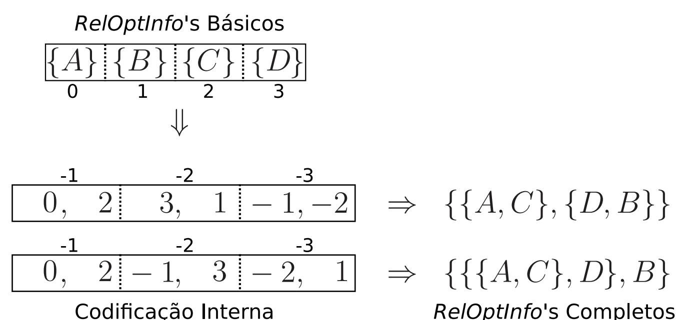


Figura 4.4: Forma de representação de estados utilizada pelo 2PO implementado neste estudo.

desnecessária. A Figura 4.4 representa como os possíveis estados foram representados internamente pelo 2PO implementado neste estudo. De forma análoga ao proposto por Ioannidis e Kang, as relações são representadas pelos respectivos *RelOptInfo*'s básicos. As operações de junção, por sua vez, são representadas pelas combinações desses *RelOptInfo*'s, através da função *make_join_rel*.

A codificação interna utilizada para representar a ordem de combinações entre os *RelOptInfo*'s se baseou em um vetor de tamanho $2(N-1)$, sendo N o número de *RelOptInfo*'s básicos fornecidos. Este vetor é dividido em $N-1$ pares, os quais são enumerados de forma decrescente, de -1 até $-N+1$. Cada par representa uma combinação entre dois *RelOptInfo*'s, os quais são representados por números positivos ou negativos. Se um número for maior ou igual a zero, este representa um *RelOptInfo* básico. Caso o número seja negativo, este refere-se a um *RelOptInfo* intermediário, indicando outro par deste vetor.

A codificação apresentada acima não apresenta restrição quanto a ordem de de-

pendência entre os *RelOptInfo*'s intermediários. Na Figura 4.4, por exemplo, a representação interna $(0, 2|3, 1| - 1, -2)$ poderia ser escrita como $(-2, -3|0, 2|3, 1)$. Desta forma, a construção de um *RelOptInfo* completo pode ser realizada tanto iterativamente como recursivamente, o que depende da posição em que se encontram as referências para os *RelOptInfo*'s intermediários.

4.3.2 Movimentos

Conforme apresentado no Capítulo 3, o mecanismo de busca utilizado pelo algoritmo 2PO depende de um conjunto de regras de transformação, a quais permitem a este algoritmo realizar uma série de movimentos no espaço de busca até que se encontre um mínimo local aceitável. Em sua proposta original, este algoritmo utiliza as cinco regras de transformação:

1. Troca do método de junção: $A \bowtie^{\textit{nested-loop}} B \rightarrow A \bowtie^{\textit{merge-join}} B$
2. Comutatividade: $A \bowtie B \rightarrow B \bowtie A$
3. Associatividade: $(A \bowtie B) \bowtie C \rightarrow A \bowtie (B \bowtie C)$
4. Troca da junção da esquerda: $(A \bowtie B) \bowtie C \rightarrow (A \bowtie C) \bowtie B$
5. Troca da junção da direita: $A \bowtie (B \bowtie C) \rightarrow B \bowtie (A \bowtie C)$

Uma vez que a implementação deste algoritmo para o PostgreSQL depende das estruturas fornecidas por este SGBD, as regras apresentadas acima precisaram ser readequadas de modo a comportar as características deste novo ambiente. As regras 1 e 2, por exemplo, puderam ser desconsideradas, visto que a estrutura fornecida pelo *RelOptInfo* já contempla essas funcionalidades. Com relação às regras 3, 4 e 5, observa-se uma certa equivalência entre elas por causa da comutatividade da função *make_join_rel*. Por exemplo,

$$\{\{A,B\},C\} \rightarrow_{(3)} \{A,\{B,C\}\} \tag{4.10}$$

$$\{\{A,B\},C\} \equiv \{\{B,A\},C\} \rightarrow_{(4)} \{\{B,C\},A\} \equiv \{A,\{B,C\}\} \tag{4.11}$$

$$\{\{A,B\},C\} \equiv \{C,\{A,B\}\} \rightarrow_{(5)} \{A,\{C,B\}\} \equiv \{A,\{B,C\}\} \tag{4.12}$$

Devido a este fato, essas três últimas regras foram readequadas na forma de duas novas regras: "troca do *RelOptInfo* da direita" e "troca do *RelOptInfo* da esquerda", conforme demonstrado na Figura 4.5. O principal objetivo dessas novas regras é definir qual dos *RelOptInfo*'s mais internos (neste caso $\{A\}$ ou $\{B\}$) deve ser trocado com o *RelOptInfo* mais externos ($\{C\}$). Para que não sejam inseridos produtos cartesianos, a utilização de cada uma dessas regras também implica na verificação dos predicados de junção aplicáveis ao novo arranjo produzido. Caso não exista um predicado de junção aplicável, a regra de transformação utilizada é então substituída pela outra regra.

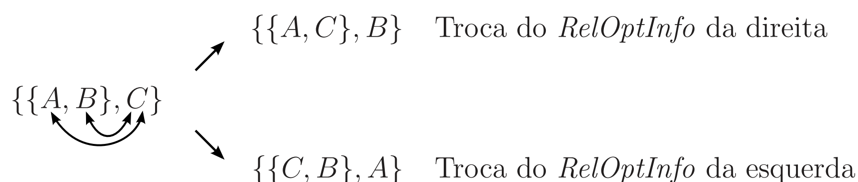


Figura 4.5: Regras de transformação utilizadas na implementação do algoritmo 2PO.

4.3.3 Encapsulamento do 2PO na Forma de *Plugin*

A partir da versão 8.3, o PostgreSQL permite que suas funcionalidades possam ser modificadas ou estendidas sem a necessidade de alteração de seu código fonte. Para isso, este SGBD utiliza a técnica de *plugins*, a qual possibilita que trechos de código sejam compilados separadamente e encapsulados na forma de bibliotecas. Essas bibliotecas, por sua vez, podem ser carregadas tanto na inicialização do SGBD como por demanda, em cada conexão realizada.

Com relação à dinâmica de desenvolvimento do PostgreSQL, o suporte a *plugins* traz uma série de vantagens. Assim como em outros projetos de código aberto, como por

exemplo o Mozilla-Firefox [6], esta extensibilidade permite que a equipe principal de desenvolvimento se concentre nas funcionalidades essenciais. Por sua vez, os casos particulares ou novas funcionalidades que ainda estão em fase de maturação podem ser tratadas separadamente, sem a necessidade de seu envolvimento.

Devido a esta possibilidade, o otimizador 2PO foi encapsulado na forma de um *plugin* chamado LJQO [4]. Esta sigla refere-se ao termo *Large Join Query Optimization*, introduzido por Swami e Gupta em [46]. A Figura 4.6 representa através de um diagrama de componentes, a interface de conexão do LJQO com o PostgreSQL. Ao ser carregado, este *plugin* assume o controle do mecanismo de otimização de junções do SGBD através de um ponteiro para função chamado *join_search_hook*. Para controlar o seu uso, este *plugin* também disponibiliza dois novos parâmetros de configuração:

ljqo_threshold: Define a quantidade mínima de relações em uma consulta para a ativação do *plugin*. Este parâmetro é análogo ao parâmetro *geqo_threshold*, o qual define a ativação do algoritmo GEQO.

ljqo_algorithm: Define qual algoritmo será utilizado pelo *plugin*. Atualmente, apenas o 2PO está disponível, o qual é referenciado pelo nome "twopo". A definição deste parâmetro é uma decisão de projeto, visando com isso permitir outras propostas de otimização.

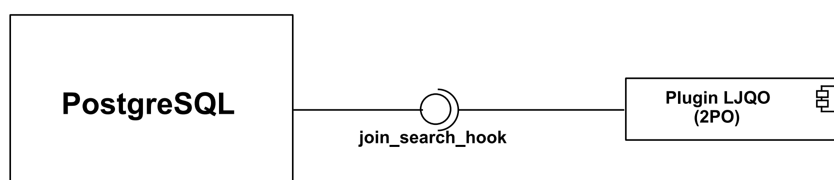


Figura 4.6: Diagrama de componentes representando a interface de conexão do *plugin* LJQO com o PostgreSQL.

O código fonte do LJQO, incluindo a implementação do algoritmo 2PO, encontra-se disponível em [4]. Sua versão atual permite a compilação para ambientes GNU/Linux, sendo compatível com as versões 8.3 e 8.4 do PostgreSQL.

CAPÍTULO 5

METODOLOGIAS DE AVALIAÇÃO ENCONTRADAS NA LITERATURA

Antes de uma avaliação efetiva dos algoritmos de otimização, é preciso primeiramente definir quais são os principais critérios a serem considerados. Este capítulo tenta elucidar melhor como tal processo de avaliação pode ser realizado na forma de uma revisão literária sobre o assunto, elencando ainda alguns valores que foram considerados como relevantes para o processo de avaliação adotado neste estudo.

5.1 A Ausência de Padrões Definidos de Avaliação

A comparação de algoritmos de otimização, de um modo geral, é uma tarefa difícil de ser realizada, visto que não existem padrões definidos de *benchmark* para tal. Além disso, a quantidade relativamente grande de possíveis variáveis que podem influenciar o comportamento dos algoritmos e as estratégias de busca diferentes empregadas por cada um deles dificultam ainda mais o processo de avaliação. Por exemplo, apenas a tarefa de escolha das cardinalidades das relações que envolvem uma consulta pode depender de fatores como o seu intervalo e distribuição desejados, sem considerar ainda o arranjo dessas cardinalidades entre as respectivas relações. Além disso, a quantidade de relações envolvidas, a sua seletividade, os métodos de junção e de acesso às relações e a presença ou não de heurísticas antes e durante o processo de otimização aumentam ainda mais a gama de dimensões passíveis de utilização.

Na maioria dos estudos comparativos encontrados na literatura, apenas um subconjunto desses fatores é considerado, podendo incluir ou não componentes aleatórios ou ainda características trazidas de outras técnicas de *benchmark* aplicadas a SGBDs relacionais. No caso de *benchmarks* tradicionais para SGBDs, é comum considerar questões como a carga do sistema ou o tempo total necessário para a execução de seus roteiros de

testes. Em contra-partida, a observação do comportamento de otimizadores pode ser focada mais especificamente a este processo propriamente dito, sendo possível ainda definir escalas qualitativas de valores para cada uma de suas características. Contudo, o tipo de algoritmo de otimização, se determinístico ou não, pode implicar na forma de avaliação empregada.

5.2 A Comparação de Algoritmos Aleatórios e o Uso da Escala de Custo

No que tange a comparação de algoritmos aleatórios de otimização, uma vez que se baseiam em técnicas de aproximação e não de busca exaustiva, os primeiros fatores comparativos a serem considerados são a qualidade dos planos gerados e o tempo ou esforço necessário para obtê-los. Em [46], os autores Swami e Gupta comparam o comportamento de alguns desses algoritmos, como o Melhoria Iterativa (II) e Têmpera Simulada (SA), aplicando-os na otimização de consultas com grande número de relações. Em sua abordagem, os autores criaram artificialmente tanto as relações como o conjunto de consultas utilizado neste processo de avaliação. Para cada consulta, diversas características foram consideradas. Tanto o grafo de junções, que determina quais os pares relações devem possuir os predicados de junção, assim como quais os atributos participantes desses predicados foram selecionados aleatoriamente. Contudo, a única restrição neste caso foi que o grafo de junções tivesse a forma de uma árvore, ou seja, um grafo conexo de $N-1$ arestas, sendo N o número de relações.

As relações também foram criadas aleatoriamente, usando para isso diversos componentes escolhidos a partir de critérios de distribuição pré-definidos, como a cardinalidade dessas relações, a presença ou não de predicados de seletividade, o percentual de valores distintos nas colunas de junção e a presença ou não de índices, conforme descrito a seguir:

Cardinalidade: A cardinalidade de cada relação foi selecionada aleatoriamente a partir de intervalos definidos de acordo com as seguintes proporções: 20% das relações apresentaram cardinalidades entre 10 e 99, 64% das relações com cardinalidades

entre 100 e 999 e 16% entre 1.000 e 9.999.

Seletividade: A quantidade de predicados de seletividade em cada relação variou de 0 a 2. A seletividade de cada predicado foi escolhida aleatoriamente a partir dos seguintes valores: $0,001$; $0,01$; $0,1$; $0,2$; $0,34$; $0,34$; $0,34$; $0,34$; $0,34$; $0,5$; $0,5$; $0,5$; $0,67$; $0,8$; $1,0$. Tais valores representam a fração da cardinalidade de uma relação.

Valores distintos: Os valores distintos em cada atributo usado nos predicados de junção foram escolhidos de acordo com as seguintes distribuições: 75% dos atributos com nenhum ou até 20% de valores distintos entre si; 5% dos atributos entre 20% e 100%; e 20% dos atributos com 100% dos valores distintos.

Índices: Aproximadamente 25% dos atributos usados nos predicados de junção foram selecionados aleatoriamente para possuírem índices.

Ao todo, foram criadas 500 consultas, sendo 50 delas para cada uma das seguintes quantidades de relações[1]: 11, 21, 31, 41, 51, 61, 71, 81, 91, 101. Cada consulta foi executada quatro vezes em cada um dos algoritmos avaliados. Durante cada execução, os autores coletaram o custo dos melhores planos obtidos em diversos momentos deste processo.

Em sua análise estatística, Swami e Gupta compararam a qualidade geral dos planos obtidos por cada algoritmo, usando para isso a média de suas *escalas de custo*. Uma vez que o custo de um plano pode ser representado por um valor escalar resultante da ponderação dos diversos fatores que compõem o seu esforço computacional, a escala de custo foi dada pela razão entre este custo pelo custo do melhor plano encontrado para a mesma consulta, independente do otimizador. Sendo assim, após uma consulta ter sido otimizada quatro vezes para cada algoritmo, os autores coletaram o menor custo obtido para a formulação de todas as escalas de custo desta consulta.

Como o próprio nome sugere, a escala de custo é uma representação de uma unidade, que varia de acordo com cada consulta em questão, e disposta em uma única escala

[1]Em seu texto original [46], os autores usam o número de junções como medida para o tamanho das consultas, sendo este igual ao número de relações menos um.

de valores. Ou seja, se um plano possuir uma escala de custo igual a 2, seu custo de execução é duas vezes maior que o custo do melhor plano. Isto garante a possibilidade de comparação de planos obtidos tanto para uma mesma consulta como entre consultas diferentes. Quanto maior for a escala de custo de um plano, pior é a sua qualidade.

5.3 A Comparação Entre 2PO, SA e II

Na comparação entre os algoritmos 2PO, Têmpera Simulada (SA) e Melhoria Iterativa (II), Ioannidis e Kang [29] avaliaram o comportamento de tais algoritmos em consultas com grandes quantidades de relações. Assim como Swami e Gupta [46], os autores também usaram consultas artificialmente criadas, baseando-se nos grafos de junções em forma de árvore. Contudo, uma maior ênfase foi dada para os grafos na forma de estrela. Em sua metodologia de testes, os autores usaram três catálogos de relações que variaram em cardinalidade e no percentual de valores distintos no atributo de junção. A Tabela 5.1 apresenta os critérios de construção das relações em cada catálogo.

Tabela 5.1: Catálogos de relações utilizados por Ioannidis e Kang [29].

Catálogo	Cardinalidade das relações	Valores distintos no atributo de junção (%)
relcat1	1.000	[90, 100]
relcat2	[1.000, 100.000]	[90, 100]
relcat3	[1.000, 100.000]	[10, 100]

Na Tabela 5.1, os catálogos foram selecionados para representar um aumento progressivo da variação de cardinalidades e da seletividade sobre as junções. Para cada atributo de junção criado, os autores assumiram que seus valores tivessem uma distribuição uniforme de acordo com seus respectivos percentuais de valores distintos. Deste modo, era possível controlar a seletividade de cada junção em uma consulta a partir desses fatores.

Quanto ao modelo de custo, Ioannidis e Kang selecionaram os algoritmos de junção baseados em *nested-loop* e *merge-join*. Os índices utilizados para a construção das relações foram baseados em árvores B+ e *hash*, de modo que cada atributo de junção tivesse 50% de probabilidade de possuir um índice e 50% de probabilidade de ser uma árvores B+ ou *hash*.

Com base nos critérios apresentados acima, as consultas foram criadas aleatoriamente com as seguintes quantidades de relações: 6, 11, 21, 41, 61, 81, 101. Para cada quantidade de relações entre 6 e 41, foram criadas 20 consultas em cada um dos três catálogos apresentados na Tabela 5.1. Já para as demais quantidades de relações, apenas 5 consultas foram criadas para cada catálogo. Cada consulta criada foi submetida 5 vezes em cada algoritmo avaliado, com exceção de casos em que seu tempo de execução ultrapassou duas horas.

Com base nos planos gerados, os autores avaliaram o comportamento dos otimizadores em relação a três dimensões principais: número de relações, catálogo e forma do grafo de junções. Para o grafo de junções foram utilizadas a forma de árvore simples e a forma de estrela. Em sua análise qualitativa, os autores apresentaram a média das escalas de custo obtidas por cada algoritmo em cada uma das três dimensões utilizadas. Além disso, seguindo a análise apresentada por Swami e Gupta, os autores Ioannidis e Kang avaliaram o comportamento dos algoritmos em função de seu tempo de otimização.

5.4 A Comparação de Algoritmos Determinísticos e Não Determinísticos

Outro estudo dedicado a comparar qualitativamente algoritmos de otimização foi realizado por Steinbrunn, Moerkotte e Kemper [43]. Neste estudo, um fato que se destaca é a diversidade de algoritmos avaliados, sendo ao todo três algoritmos determinísticos e nove não determinísticos. Para os algoritmos não determinísticos, os autores avaliaram três implementações do Melhoria Iterativa (II), três implementações do Têmpera Simulada (SA), o algoritmo 2PO, e duas implementações de Algoritmos Genéticos (GA). Dentre esses algoritmos, duas implementações de II, duas de SA e uma de GA eram versões específicas para o espaço de busca de árvores em profundidade à esquerda, enquanto que as demais implementações usaram o espaço de busca de árvores fechadas de junções. Em todas as versões dos algoritmos não determinísticos utilizados, os autores se basearam fortemente nas técnicas apresentadas por Swami e Gupta [46] e por Ioannidis, Bennett *et*

al. [13, 29].

Em seu ambiente de testes, Steinbrunn *et al.* utilizaram as seguintes especificações. As relações foram construídas de acordo com as proporções de cardinalidade e de domínio apresentadas nas Tabelas 5.2 e 5.3. Neste caso, ao contrário do apresentado por Ioannidis e Kang [29], apenas um catálogo de relações foi construído. Com base neste, as consultas foram geradas aleatoriamente seguindo dois critérios principais, o número de relações e o grafo de junções. O número de relações utilizado também foi menor que o apresentado pelos dois estudos anteriores, variando de 5 até 30 relações. Contudo, os autores incrementaram uma relação de cada vez, formando assim um total de 26 valores diferentes para a quantidade de relações.

Tabela 5.2: Distribuição das cardinalidades das relações utilizadas por Steinbrunn *et al.* [43].

Cardinalidade	Porcentagem
10-100	15%
100-1000	30%
1000-10000	35%
10000-100000	20%

Tabela 5.3: Distribuição dos domínios dos atributos utilizados por Steinbrunn *et al.* [43].

Tamanho do Domínio	Porcentagem
2-10	5%
10-100	50%
100-500	30%
500-1000	15%

Outra característica importante do estudo de Steinbrunn *et al.* foi o uso maior dos grafos de junções, passando assim a analisar melhor o efeito de sua conectividade sobre o comportamento dos algoritmos avaliados. Sendo assim, foram usadas quatro formas diferentes de grafos: corrente, círculo, grade e estrela. Assim como nos estudos apresentados anteriormente, os grafos corrente e estrela representaram basicamente as mesmas variantes de consultas na forma de árvores. Todavia, consultas cíclicas como círculo e grade ainda não haviam sido avaliadas efetivamente.

Quanto aos métodos de junção, foram utilizados o *nested-loop*, *hash* e *sort-merge*. O custo calculado para esses métodos foi estimado com base na quantidade de páginas lidas e armazenadas em memória secundária. Além disso, 20% dos atributos criados foram selecionados aleatoriamente para possuírem índices em árvore B+ ou *hash*.

As consultas utilizadas pelos autores foram construídas aleatoriamente de acordo com as especificações anteriores. Para cada combinação de número de relações e grafo de junções, os autores construíram uma consulta apenas. Cada consulta foi então submetida 30 vezes em cada otimizador, obtendo-se com isso os custos dos planos gerados bem como seus respectivos tempos de otimização. Como análise dos dados obtidos, os autores avaliaram a qualidade dos planos gerados por cada otimizador, utilizando para isso a média de suas respectivas escalas de custo. Esses dados foram então apresentado em função do grafo de junções e da quantidade de relações de cada consulta.

5.5 Propostas Determinísticas de Avaliação

Outra metodologia importante encontrada na literatura foi proposta por Vance e Maier em [53]. Neste estudo, os autores avaliaram o desempenho e a complexidade de seu algoritmo exaustivo de otimização, chamado *blitzsplit*, frente a outras técnicas também exaustivas, como os algoritmos usados pelos projetos Starburst [38] e Volcano [24]. Na elaboração de seu método de avaliação, Vance e Maier analisaram possíveis características pertinentes a consultas e ao modelo de custo que pudessem influenciar na quantidade de esforço exigido por um otimizador. Desta forma, os autores propuseram um método sistemático e multidimensional para a elaboração de seu caso de testes. Neste caso, a quantidade de relações foi fixada em 15, mas outros fatores puderam ser avaliados independentemente, produzindo um conjunto de quatro dimensões simultâneas: média geométrica das cardinalidades, variação das cardinalidades em torno da média geométrica, tipo do grafo de junções e o modelo de custo.

Diferentemente das metodologias apresentadas por Swami e Gupta [46], Ioannidis e Kang [29] e Steinbrunn *et al.* [43], os autores Vance e Maier conduziram a elaboração de seu caso de testes de forma determinística, sem que possíveis fatores aleatórios pudessem

influenciar no comportamento do esforço de otimização desses algoritmos exaustivos. Segue abaixo os fatores apresentados pelos autores para a construção de sua metodologia de avaliação:

- **Número de relações da consulta:** Como mencionado anteriormente, o número de relações foi fixado em 15. Deste modo, as relações podem ser referenciadas por R_1 a R_{15}, sendo R_1 a relação com menor cardinalidade e a relação R_{15} a de maior cardinalidade [2].

- **Média geométrica das cardinalidades das relações:** A média geométrica das cardinalidades das relações (μ) é dada por $(\prod_{i=1}^{15} |R_i|)^{1/15}$. Nos gráficos apresentados pelos autores, os valores de μ estiveram num intervalo de 1 até 10^8.

- **Variação das cardinalidades das relações:** A variação das cardinalidades (v) foi dada em um intervalo de 0 a 1, sendo 0 o indicativo de que todas as relações possuem a mesma cardinalidade. Para todos os valores de v, a cardinalidade das relações seguiu de modo que a cardinalidade de R_1 fosse igual a μ^{1-v} e a progressão geométrica das cardinalidades de R_1 até R_{15} fosse constante.

- **Grafo de junções e seletividade:** Foram selecionados quatro tipos de grafos: corrente, círculo+3, estrela e clique. As consultas do tipo corrente apresentaram os seguintes predicados de junção: $R_1 - R_9 - R_2 - R_{10} - R_3 - R_{11} - R_4 - R_{12} - R_5 - R_{13} - R_6 - R_{14} - R_7 - R_{15} - R_8$. Para a composição das consultas do tipo círculo+3, foram considerados os mesmos predicados apresentados nas consultas tipo corrente, incluindo ainda os seguintes: $R_1 - R_8$, $R_9 - R_{15}$, $R_2 - R_7$ e $R_{10} - R_{14}$. Nas consultas do tipo estrela, todas as relações se conectaram à relação R_{15}. Por fim, as consultas do tipo clique formaram um grafo completo entre todas as 15 relações. Para a seletividade de cada predicado de junção, foram observados critérios baseados na média geométrica, número total de arestas do grafo de junções e grau de cada relação no grafo.

[2] No testo original de Vance e Maier [53], as relações são nomeadas de R_0 até R_{14}.

- **Modelo de custo:** Os autores selecionaram três funções de custo: um modelo simples baseado apenas na cardinalidade das relações intermediárias, um baseado no método de junção *sort-merge* e outro baseado no método de junção *nested-loop*.

Em outro estudo dedicado a avaliar técnicas exaustivas de otimização de consultas, Shapiro, Vance, Maier *et al.* [42] apresentaram um ambiente de testes semelhante ao anteriormente proposto por Vance e Maier [53]. Neste estudo, os autores avaliaram seu algoritmo exaustivo, chamado *Columbia*, em relação ao algoritmo utilizado pelo projeto Starburst [38]. Assim como no estudo anterior, este se dedicou a medir o esforço de otimização dessas técnicas exaustivas frente a diversos fatores presentes em um espaço de busca.

A estrutura de parâmetros adotada por Shapiro *et al.* seguiu basicamente a mesma forma apresentada por Vance e Maier, no entanto com algumas modificações. Foram avaliadas consultas com o número de relações variando entre 4 e 13, sendo R_1 a relação de menor cardinalidade e R_n a de maior cardinalidade [3]. A média geométrica das cardinalidades foi fixada em 2^{12}. Já a variação das cardinalidades, denominada pelos autores como *LOGRATIO*, foi dada pela diferença logarítmica entre a relação de maior cardinalidade da consulta pela de menor cardinalidade, ou seja, $log_2(|R_n|/|R_1|)$. Os autores usaram três valores de *LOGRATIO* diferentes: 0, 12 e 24. Assim como proposto por Vance e Maier, a razão $|R_{i+1}|/|R_i|$ se manteve igual para todas as relações de cada consulta, formando assim uma progressão geométrica de suas cardinalidades. Para controlar a seletividade das junções, os autores usaram a própria variação entre as cardinalidades de suas relações, dada por $|R_i \bowtie R_{i+1}| = |R_{i+1}|$. Desta maneira, o aumento da seletividade poderia ser alcançado pela simples diminuição do *LOGRATIO*.

Quanto a forma dos grafos de junções, Shapiro *et al.* selecionaram apenas os tipos corrente e estrela. Contudo, diferente da proposta original, o arranjo entre as relações para as consultas do tipo corrente foi feita de forma decrescente quanto a cardinalidade de suas relações, ou seja, $R_n - R_{n-1} - R_{n-2} - \cdots - R_1$. Já para as consultas do tipo estrela, a relação central selecionada para a consulta foi a relação de maior cardinalidade

[3] No texto original, Shapiro *et al.* [42] nomeiam a relação R_1 como sendo a de maior cardinalidade.

(R_n).

De um modo geral, percebe-se que as características utilizadas por Vance e Maier e por Shapiro *et al.* para descrever seus respectivos ambientes de testes também estão presentes nas metodologias citadas anteriormente. Contudo, a grande contribuição deixada por esses autores foi o uso mais efetivo de valores escalares para a atribuição dessas características.

5.6 O Uso de *Benchmarks* Tradicionais

Geralmente, um mecanismo eficiente de otimização junções pode influenciar no bom comportamento de um SGBD. Por este motivo, uma forma conveniente de avaliar este comportamento como um todo pode ser através do uso de técnicas de *benchmark* bem conhecidas. O TPC (*Transaction Processing Performance Council*) [8] é uma organização independente e sem fins lucrativos que tem por objetivo estabelecer padrões de *benchmarks* direcionados a bancos de dados. Esta organização mantém diversos *benchmarks*, os quais são, em sua essência, especificações e métricas que visam avaliar a performance de um SGBD em diferentes cenários considerados relevantes pela indústria.

Em alguns dos estudos dedicados em avaliar algoritmos de otimização de junções, é possível encontrar como parte de suas metodologias o uso de *benchmarks* mantidos pelo TPC. Em sua maioria, essas metodologias utilizam apenas os esquemas de dados e as consultas fornecidas por eles, tomando por métricas de comparação entre os algoritmos, as suas respectivas escalas de custo, o tempo de otimização ou ainda o tempo total de execução de cada consulta. Uma vez que esses *benchmarks* estão ligados a cenários bem definidos, percebe-se ainda que este fato também é levado em consideração na escolha do *benchmark* utilizado.

Em um estudo apresentado por Chen *et al.* [16], os autores utilizam consultas derivadas do TPC-H [48] e TPC-DS [9], dois *benchmarks* direcionados para sistemas de apoio à decisão, como forma de avaliar sua proposta de algoritmo de otimização parcial de junções. Este algoritmo, chamado *Partial Join Order* (PJO), foi implementado sobre o *ParAccel Analytic Database* (PADB), uma versão paralela e orientada a colunas de SGBD derivada do PostgreSQL. Os autores compararam o PJO frente aos dois otimizadores existentes na

versão original do PostgreSQL, o GEQO e o de programação dinâmica, utilizando para isso consultas que variaram entre 2 e 42 relações.

Em [14], os autores Bini *et al.* avaliaram o comportamento dos algoritmos GEQO e Kruskal (KQO) [25] utilizando PostgreSQL como ambiente de testes. Uma vez que os ambientes OLTP (*On-Line Transaction Processing*) são propensos a serem mais rigorosos com relação a regularidade do tempo de resposta de suas operações, os autores avaliaram a estabilidade dos planos obtidos por ambos os algoritmos em várias consultas derivadas do TPC-E [10]. Essas consultas tinham entre 11 e 19 relações. Os autores demonstram uma variação elevada dos planos obtidos pelo GEQO, ressaltando ainda que esta poderia trazer perigosos níveis de imprevisibilidade, tanto no consumo de recursos do próprio SGBD, que processa as consultas, como para o tempo de resposta de toda uma cadeia de serviços que dependem dos resultados fornecidos por ele.

5.7 Principais Elementos Encontrados nas Metodologias Apresentadas

As metodologias apresentadas neste capítulo refletem, em grande parte, as técnicas encontradas na literatura para a avaliação de algoritmos aplicados na otimização de junções. Como foi mencionado anteriormente, existem inúmeros fatores independentes que podem ser levados em consideração para este tipo de avaliação. Deste modo, torna-se difícil, mesmo que fosse possível, assumir estes fatores por completo sem que o avaliador se depare com uma explosão de dados e dimensões a serem analisados. Portanto, o uso de apenas alguns fatores de considerável relevância, garante a tratabilidade dos mesmos bem como a exposição de seus respectivos resultados.

Na maioria dos estudos apresentados, são observados alguns fatores importantes e de uso comum. O número de relações, por exemplo, é um dos fatores determinantes para a explosão combinatória do espaço de busca de uma consulta, portanto fortemente empregado em relação ao tempo de otimização. Além disso, o uso de valores elevados para a quantidade de relações (acima de 10 ou 15) é ainda mais evidente na avaliação de algoritmos não

exaustivos. A cardinalidade das relações, por outro lado, está geralmente relacionada ao esforço computacional exigido para o processamento de cada possível plano de execução, principalmente com relação à demanda de uso das memórias primária e secundária. Esta demanda computacional depende ainda de fatores como os métodos disponíveis para cada operação algébrica, as características do ambiente computacional e as estruturas de dados utilizadas. Principalmente na avaliação de algoritmos não exaustivos, é comum o uso da escala de custo como forma de comparação de planos obtidos, tanto para uma mesma consulta como para consultas diversas.

A seletividade também é outro fator importante a ser considerado, pois tem efeito direto sobre a cardinalidade das relações intermediárias. Tanto as operações de seleção diretamente aplicadas sobre as relações básicas como os predicados de junção são responsáveis por isso. Contudo, para consultas com números elevados de relações, os predicados de junção, geralmente representados por grafos de junções, são objetos de maior atenção para as estratégias de busca apresentadas. Além disso, esta seletividade provida pelos predicados de junções pode ser influenciada pela diferença de cardinalidade das relações e pelo domínio dos atributos de junção utilizados, de maneira que esses fatores também são observados em diversas metodologias.

CAPÍTULO 6

METODOLOGIA UTILIZADA

A metodologia utilizada neste estudo para avaliar os algoritmos 2PO e GEQO, se baseia em grande parte nos experimentos apresentados no Capítulo 5. A partir desses experimentos, foram observados diversos critérios que permitissem avaliar o comportamento de tais algoritmos em diferentes condições. Este capítulo descreve como este ambiente de testes foi elaborado, levando em consideração a construção do esquema de dados e as consultas utilizadas. O ambiente computacional, bem como a análise estatística dos resultados obtidos pelos algoritmos avaliados serão apresentados no capítulo seguinte.

6.1 Definição dos Parâmetros de Construção

Neste estudo, a forma de construção do esquema de dados e do conjunto de consultas seguiu principalmente os moldes sistemático e multidimensional apresentados por Vance e Maier [53] e por Shapiro *et al.* [42]. Uma das vantagens encontradas na metodologia utilizada por esses autores foi a independência de seus diversos parâmetros de construção, de modo que fosse possível variá-los ou fixá-los dependendo do objetivo de análise desejado. Além disso, o uso de valores escalares para a maioria desses parâmetros, exceto no caso dos grafos de junções, tornou mais simples a definição dos mesmos.

Nos estudos apresentados por esses autores, os parâmetros de construção foram definidos visando avaliar algoritmos exaustivos. Isto reflete, por exemplo, na baixa quantidade relações em cada consulta. Por outro lado, tanto o algoritmo 2PO como o GEQO são direcionados a trabalhar em condições onde a aplicação de tais algoritmos exaustivos se torna proibitiva. Desta forma, alguns dos parâmetros originalmente utilizados por esses autores foram estendidos ou readequados de forma que fosse possível observar o comportamento dos algoritmos aqui estudados. Esses parâmetros foram de modo geral trazidos das metodologias propostas por Swami e Gupta [46], Ioannidis e Kang [29] e por Steinbrunn

et al. [43].

A Tabela 6.1 apresenta um comparativo das metodologias apresentadas por Vance e Maier e por Shapiro *et al.* em relação a metodologia adotada neste estudo. É possível observar que o número de relações em cada consulta foi estendido para valores bem acima do que usualmente é aplicado na avaliação de algoritmos exaustivos. Assim como utilizado por Swami e Gupta [46] e por Ioannidis e Kang [29], procurou-se avaliar consultas que chegassem na ordem de 100 relações. Todavia, a resolução escolhida para este parâmetro foi menor, uma vez que a adição dos parâmetros relacionados à variação das cardinalidades das relações e grafos de junções compensou esta ausência. Assim como no Capítulo 5, as relações serão representadas pelos termos R_1 a R_N, em ordem não decrescente, sendo R_1 a relação com menor cardinalidade e R_N a com maior cardinalidade.

Tabela 6.1: Comparativo entre os parâmetros utilizados por Vance e Maier e por Shapiro *et al.* em relação a metodologia adotada.

Parâmetro	Vance e Maier [53]	Shapiro *et al.* [42]	Metodologia utilizada
Número de relações (N)	15	4 a 13	10, 20, 50 e 100
Média geométrica (μ)	10^2 a 10^8	2^{12}	2^{12}
Variação das cardinalidades	$\|R_1\| = \mu^{1-v}$, v variando de 0 a 1	$LOGRATIO = 0$, 12 e 24	$LOGRATIO = 6$, 12 e 14
Razão entre as cardinalidades	$\|R_i\|/\|R_{i-1}\|$	$\|R_i\|/\|R_{i-1}\|$	$\mu = \sqrt{\|R_i\| \cdot \|R_{N-(i-1)}\|}$
Seletividade	Baseada na média geométrica, no grau dos vértices e na cardinalidade.	Baseado na cardinalidade das relações.	Baseado na cardinalidade das relações.
Grafos de junções	corrente, círculo+3, estrela e grafo completo	corrente e estrela	corrente, círculo, grade e estrela
Arranjo das relações no grafo	Determinístico	Determinístico	Aleatório

O valor escolhido para a média geométrica das cardinalidades seguiu o mesmo proposto por Shapiro *et al.* [42], ou seja, 2^{12}. Este valor pareceu não muito baixo, para depreciar a dificuldade do modelo de custo, nem muito alto, de modo que tornasse excessivamente

custosa a construção das relações no ambiente de testes. Contudo, outros valores para este parâmetro não foram avaliados, cabendo portanto futuras investigações caso necessário.

Também seguindo os moldes apresentados por Shapiro *et al.*, o termo usado para a variação das cardinalidades foi o *LOGRATIO*. Ou seja,

$$\begin{aligned} LOGRATIO &= \log_2(|R_N|/|R_1|) \\ &= \log_2(|R_N|) - \log_2(|R_1|) \end{aligned} \quad (6.1)$$

De um modo geral, o termo *LOGRATIO* lembra uma diferença de proporção entre as relações de maior e menor cardinalidades, sendo um valor escalar simples de ser utilizado e de fácil visualização.

Com relação à distribuição das cardinalidades em função de sua variação, os estudos propostos por Vance e Maier e por Shapiro *et al.* apresentam-na na forma de progressão geométrica, de modo que a razão $|R_i|/|R_{i-1}|$ fosse constante em toda uma consulta. Uma vez que o número de relações utilizado neste estudo é bem maior que o apresentado por esses autores, foi permitida que algumas relações tivessem mesma cardinalidade, contudo, mantendo-se a média geométrica da consulta. Além disso, esta mesma média geométrica serviu para os pares de relações equidistantes de suas respectivas extremidades. Ou seja,

$$\mu = \sqrt{|R_i| \cdot |R_{N-(i-1)}|} \quad (6.2)$$

para todo i entre 1 e $N/2$.

Com relação ao formato dos grafos de junções, seguiu-se o mesmo esquema apresentado por Steinbrunn *et al.* [43]. As formas escolhidas para os grafos de junções foram *corrente*, *círculo*, *grade* e *estrela* e o arranjo das relações nesses grafos se deu de forma aleatória. Tal arranjo aleatório, apresentado por Steinbrunn *et al.*, é diferente das formas determinísticas utilizadas por Vance e Maier e por Shapiro *et al.* Contudo, este mesmo arranjo aleatório é observado nos estudos apresentados por Swami e Gupta [46] e por Ioannidis e Kang [29].

Assim como proposto por Shapiro *et al.* a seletividade das consultas foi controlada

pelos seus respectivos predicados de junção e pela variação entre as cardinalidades das relações. Desta forma, quanto maior for o *LOGRATIO* utilizado, menor a cardinalidade final da consulta.

6.2 Construção do Esquema de Dados

O banco de dados utilizado neste estudo foi criado sinteticamente de forma que pudesse comportar os critérios e dimensões citados acima. Para isso, foram criadas diversas relações com cardinalidades variando exponencialmente de 2^5 até 2^{19}.

Todas as relações do banco de dados foram construídas seguindo um mesmo esquema básico de construção. Em cada uma dessas relações, foram criados três atributos numéricos: uma chave primária (*pk*) e dois outros atributos usados como chaves estrangeiras (*fk1* e *fk2*). Por padrão do próprio PostgreSQL, o atributo chave de cada relação criada recebeu um índice baseados em árvore B+.

As tuplas foram inseridas em cada relação em função de sua cardinalidade. Para o atributo *pk*, os valores foram atribuídos sequencialmente de 1 até cardinalidade da relação. Nos demais atributos, *fk1* e *fk2*, seus valores foram selecionados de forma aleatória, também seguindo intervalo de 1 até a respectiva cardinalidade, usando para isso uma distribuição uniforme. Durante todo este processo de construção, foram utilizadas sementes aleatórias diferentes para cada relação.

6.3 Construção das Consultas

As consultas utilizadas nos experimentos aqui apresentados foram produzidas usando o esquema de dados sugerido na Seção 6.2 e de acordo com a combinação dos critérios apresentados na Tabela 6.1. Para isso, foram usados dois passos independentes: a *seleção* dos conjuntos de relações e a *combinação* dessas relações na forma de consultas SQL.

6.3.1 Seleção dos Conjuntos de Relações

Neste primeiro passo, foram selecionados deterministicamente 12 conjuntos de relações de acordo com a combinação dos parâmetros *Número de relações* e *LOGRATIO*. Ou seja, para cada quantidade de relações ($N = \{10, 20, 50, 100\}$), foram gerados três conjuntos de relações com variações de cardinalidades diferentes. Em cada conjunto gerado, todas as relações selecionadas eram diferentes entre si, embora a cardinalidade de algumas delas pudesse ser a mesma. Por exemplo, para um conjunto com 10 relações e um *LOGRATIO* igual a 6, as respectivas cardinalidades das relações foram $(2^9, 2^9, 2^{10}, 2^{10}, 2^{11}, 2^{13}, 2^{14}, 2^{14}, 2^{15}, 2^{15})$. Já para um *LOGRATIO* igual a 14, as cardinalidades selecionadas foram $(2^5, 2^6, 2^7, 2^9, 2^{10}, 2^{14}, 2^{15}, 2^{17}, 2^{18}, 2^{19})$. Para todos os conjuntos de relações obtidos, a média geométrica das cardinalidades foi fixada em 2^{12}. Este mesmo valor médio também se manteve para cada par de relações $(R_i, R_{N-(i-1)})$, para todo i entre 1 e $N/2$.

6.3.2 Combinação das Relações na Consulta SQL

Uma vez selecionados os conjuntos de relações no passo anterior, o próximo passo foi combinar essas relações na forma de consultas SQL. Para cada um dos 12 conjuntos selecionados, foram geradas aleatoriamente 40 consultas, sendo 10 delas para cada tipo de grafo de junção escolhido (corrente, círculo, grade e estrela). Com isso, o número total de consultas geradas foi 480.

As consultas foram geradas da seguinte forma. A partir de um conjunto de relações (R_1 a R_N) selecionado no primeiro passo, cada consulta derivada obedeceu um arranjo aleatório dessas relações (T_1 a T_N). A partir deste arranjo, as relações foram "acomodadas" nas respectivas estruturas SQL correspondentes a cada grafo de junção.

O Código 6.1 apresenta o exemplo genérico de uma consulta do tipo corrente, de acordo com o arranjo aleatório de relações. Neste tipo de consulta, cada relação T_i ligou-se à relação T_{i+1} pelo seguinte predicado: $T_i.pk = T_{i+1}.fk1$.

As consultas do tipo círculo seguiram praticamente o mesmo formato das consultas do tipo corrente, exceto pela adição de mais um predicado ($T_N.pk = T_1.fk1$) no final da

declaração da consulta. O Código 6.2 exemplifica este tipo de consulta.

Código 6.1 Exemplo de consulta na forma de corrente (*chain*).

```
SELECT *
  from T1
  join T2 on T1.pk = T2.fk1
  join T3 on T2.pk = T3.fk1
  ...
  join TN on TN-1.pk = TN.fk1
```

Código 6.2 Exemplo de consulta na forma de círculo (*cycle*).

```
SELECT *
  from T1
  join T2 on T1.pk = T2.fk1
  join T3 on T2.pk = T3.fk1
  ...
  join TN on TN-1.pk = TN.fk1
  where TN.pk = T1.fk1
```

As consultas do tipo grade obedeceram uma combinação um pouco diferente das consultas do tipo corrente e círculo. Seu formato se assemelha a duas consultas do tipo corrente ligadas entre si por meio de um mapeamento um-para-um entre as relações de cada corrente. O Código 6.3 demonstra o esquema genérico de uma consulta na forma de grade.

Código 6.3 Exemplo de consulta na forma de grade (*grid*).

```
SELECT *
  from T1
  join T2 on T1.pk = T2.fk1
  join T3 on T1.pk = T3.fk1
  join T4 on T2.pk = T4.fk1 and T3.pk = T4.fk2
  join T5 on T3.pk = T5.fk1
  join T6 on T4.pk = T6.fk1 and T5.pk = T6.fk2
  ...
```

Por fim, o Código 6.4 apresenta a estrutura básica das consultas do tipo estrela. Este tipo de consulta possui uma relação central, a qual faz parte de todos os predicados de

junção. No esquema SQL adotado aqui, todos os atributos *fk1* das relações T_2 a T_N se ligaram com o atributo *pk* de T_1, sendo T_1 a T_N o arranjo aleatório das relações de uma consulta.

Código 6.4 Exemplo de consulta na forma de estrela (*star*).

```
SELECT *
  from T1
  join T2 on T1.pk = T2.fk1
  join T3 on T1.pk = T3.fk1
  ...
  join TN on T1.pk = TN.fk1
```

6.4 Transitividade de Predicados

As consultas SQL definidas acima possuem uma quantidade definida de predicados de junção, os quais são explicitamente informados ao SGBD no momento de sua submissão. Contudo, existe ainda a possibilidade de que outros predicados possam ser adicionados implicitamente pelo próprio SGBD, através do mecanismo de reescrita, sem que o usuário tenha ciência disso.

Uma propriedade importante que permite ao mecanismo de reescrita de um SGBD adicionar novos predicados em uma consulta é a *transitividade*. Ou seja, se existe um predicado, explicitamente contido em uma instrução SQL, que indica que $T_1.A$ é igual a $T_2.B$, e existe ainda uma conjunção (E) para outro predicado indicando que $T_1.A$ também é igual a $T_3.B$, logo, o mecanismo de reescrita pode presumir que $T_2.B$ é igual a $T_3.B$, adicionando assim este novo predicado:

$$((T_1.A = T_2.B) \wedge (T_1.A = T_3.B)) \Rightarrow (T_2.B = T_3.B) \tag{6.3}$$

Assim como a maioria dos SGBDs comerciais, o PostgreSQL também considera a transitividade dos predicados de junção durante o processo de reescrita de uma consulta. Como outras técnicas de reescrita, esta serve para enriquecer e orientar as opções fornecidas aos mecanismos subsequentes em seu processo de execução. Para um algoritmo de

otimização de junções, quando procura-se evitar por heurística os produtos cartesianos, a adição implícita de novos predicados de junção pode resultar em uma melhor qualidade do plano escolhido. Em contrapartida, o preço que se paga neste caso é o aumento dos possíveis planos a serem avaliados.

Ambos os algoritmos aqui estudados são orientados a evitar produtos cartesianos, de modo que a quantidade de arestas (ou predicados) no correspondente grafo de junções de uma consulta possui um impacto significativo no espaço de busca a ser avaliado. Neste sentido, as construções SQL, apresentadas na Seção 6.3 para os tipos de grafos utilizados nesta metodologia, refletem, dos tipos corrente até estrela, uma ordem crescente de suas respectivas quantidades de arestas. Esta ordem não pode ser observada pelas estruturas SQL em si, mas pela transitividade implícita de seus predicados de junção. A Tabela 6.2 apresenta a diferença entre a quantidade de arestas explicitamente declarada em cada tipo de consulta SQL e a quantidade final de arestas após o processo de reescrita. Note que as consultas do tipo corrente e círculo não possuem aumento de arestas, uma vez que não existem predicados compartilhando os mesmos atributos, conforme pode ser observado os Códigos 6.1 e 6.2.

Tabela 6.2: Comparação da quantidade de arestas do grafo de junções de uma consulta SQL, antes e depois da reescrita.

Tipo de grafo de junções	Quantidade de arestas explícitas (SQL)	Quantidade de arestas após a reescrita
corrente	$N-1$	$N-1$
círculo	N	N
grade	$(3/2)N-2$	$2N-3$
estrela	$N-1$	$N(N-1)/2$

Ao contrário das consultas do tipo corrente e círculo, as consultas do tipo grade e estrela possuem predicados que permitem a aplicação de propriedades transitivas. A Figura 6.1 ilustra a adição de arestas feita pelo mecanismo de reescrita do SGBD para esses tipos de consultas.

Nas consultas do tipo estrela, conforme foi apresentado no Código 6.4, todas as relações estão ligadas pelos seus respectivos predicados de junção a um único atributo central $T_1.pk$.

Logo, tem-se o seguinte:

$$((T_1.pk = T_i.fk_1) \wedge (T_1.pk = T_j.fk_1)) \Rightarrow (T_i.fk_1 = T_j.fk_1) \tag{6.4}$$

sendo T_i e T_j quaisquer relações da consulta diferentes de T_1. Deste modo, as consultas do tipo estrela são tratadas pelo otimizador como grafos completos (ou *cliques*).

Para as consultas do tipo grade, esta transitividade é mais difícil de ser descrita textualmente. De modo geral, a cada duas novas relações adicionadas a este tipo de consulta, são acrescentadas três novas arestas explicitamente e uma de forma implícita. No exemplo do Código 6.3, são adicionados implicitamente os predicados $T_2.fk_1 = T_3.fk_1$ e $T_4.fk_2 = T_5.fk_1$.

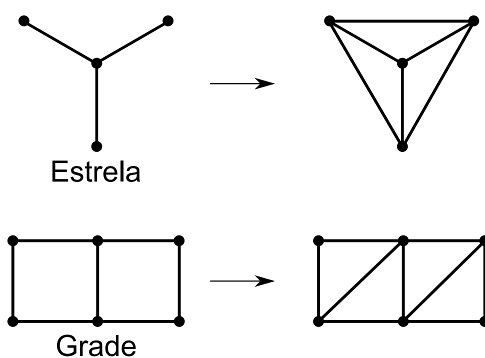


Figura 6.1: Aumento da conectividade nas consultas do tipo grade e estrela devido a transitividade de seus predicados de junção.

CAPÍTULO 7

RESULTADOS EXPERIMENTAIS

Este capítulo apresenta os resultados dos experimentos realizados na comparação dos otimizadores 2PO e GEQO aplicados ao PostgreSQL. O principal objetivo destes experimentos foi observar a performance e a qualidade dos planos obtidos por cada um desses otimizadores com base no ambiente de testes apresentado no Capítulo 6. Como já mencionado, as consultas utilizadas para este fim são resultantes da combinação de três dimensões distintas:

- Número de relações (N): 10, 20, 50 e 100;
- Variação das cardinalidades ($LOGRATIO$): 6, 12 e 14;
- Grafos de junções: corrente, círculo, grade e estrela.

Tais dimensões representam um conjunto de 48 possíveis pontos de observação, sendo cada um deles composto por 10 consultas aleatoriamente selecionadas. Nestes moldes, o número total de consultas utilizadas nesta avaliação foi de 480.

Os experimentos foram conduzidos em um computador com processador Intel Xeon com quatro núcleos de 64bits, tendo 2 GHz de clock em cada núcleo e 12MB de memória cache L2. A memória RAM utilizada foi de 2GB, operando em 667 MHz. Como memória secundária, foram utilizados dois discos rígidos SATA de 250GB cada, operando em Raid 0. O sistema operacional utilizado foi o GNU/Linux, com kernel versão 2.6.24 compilado para arquitetura X86_64.

Como já mencionado no Capítulo 4, a versão utilizada do PostgreSQL foi a 8.3, compilada a partir de seu código fonte. O *plugin* LJQO, contendo o algoritmo 2PO, foi configurado para assumir apenas o controle do mecanismo de otimização de junções, sem alterar com isso o comportamento de outras partes do SGBD. Tanto o PostgreSQL como o LJQO foram compilados usando o *GNU Compiler Collection* (gcc) versão 4.2.

A partir das consultas apresentadas, cada otimizador foi submetido a uma série de 10 otimizações, das quais foram extraídos o tempo de otimização, a quantidade de planos gerados e o custo do melhor plano encontrado. Para que fosse possível extrair tais dados, tanto o 2PO, implementado na forma de *plugin*, como o GEQO, dentro do próprio PostgreSQL, foram equipados com dispositivos de contagem de tempo e de planos gerados, juntamente com dispositivos saída para recipientes apropriados de coleta de dados. Durante toda a realização dos experimentos, as estatísticas do banco de dados se mantiveram inalteradas, de modo que foi possível comparar a qualidade dos planos obtidos em uma mesma consulta usando para isso suas respectivas escalas de custo.

Uma vez que a metodologia aqui apresentada deriva de uma comparação direta entre os otimizadores, o tempo total de execução dos planos obtidos não foi computado. Como será demonstrado na Seção 7.2, a qualidade dos planos obtidos pelo GEQO foi, em alguns casos, consideravelmente ruim. Isto impossibilitou, por questões de recursos computacionais e de tempo, a execução de tais planos. Além disso, o tempo de execução de cada plano pode ser considerado como uma consequência de sua qualidade, o que depende ainda da precisão e acurácia do modelo de custo e das estatísticas fornecidas pelo SGBD em questão.

Este capítulo se divide em quatro partes principais. Nas Seções 7.1 e 7.2, são avaliadas a performance e a qualidade dos planos obtidos pelos otimizadores. Como forma de simplificação, tais seções apresentam os dados em função de apenas duas das três dimensões disponíveis: número de relações e grafo de junções. Tais dimensões foram selecionadas pois apresentaram maior correlação entre as respectivas escalas de custo obtidas, de modo que pudessem demonstrar semelhança entre os membros de cada grupo de consultas bem como apresentar significado na comparação entre os tais. Em seguida, a Seção 7.3 trata da observação mais detalhada dos dados coletados, levando em consideração a variação das cardinalidades das relações. Por último, a Seção 7.4 apresenta os principais pontos observados na comparação entre os otimizadores.

7.1 Performance dos Otimizadores

Os dados apresentados nesta seção são resultantes do tempo de otimização e da quantidade de planos gerados por cada um dos otimizadores avaliados. Como mencionado anteriormente, cada ponto de observação apresentado aqui é resultado da combinação da quantidade de relações e do grafo de junções de cada consulta. Desta forma, os valores apresentados para a performance dos algoritmos representam a média de 300 otimizações, sendo 10 delas para cada uma das 30 consultas utilizadas, independente de seus valores de *LOGRATIO*.

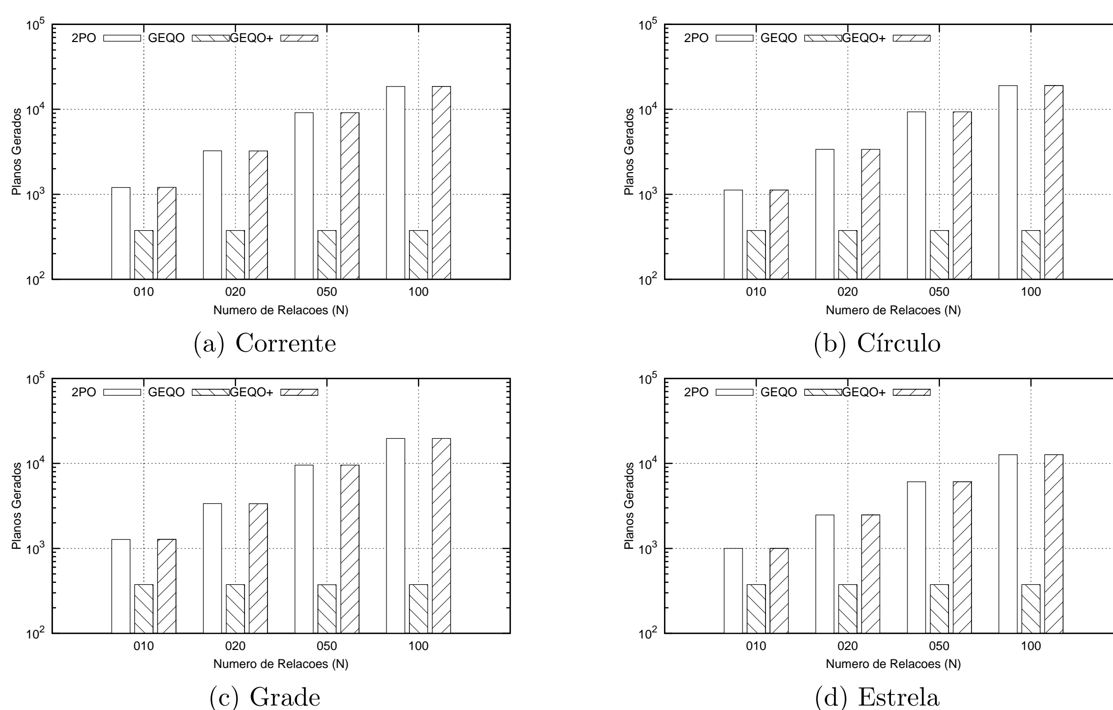
A Figura 7.1 apresenta um conjunto de quatro gráficos contendo a quantidade média de planos gerados por cada otimizador. Nestes gráficos, os dados obtidos estão agrupados por cada grafo de junção utilizado, permitindo assim observar a evolução de cada otimizador em função do aumento da quantidade de relações.

Tanto o 2PO como o GEQO apresentados neste estudo estão configurados usando os parâmetros originais para suas respectivas condições de parada. Nestas condições, observa-se na Figura 7.1, uma diferença significativa entre a quantidade média de planos gerados por cada um deles. Enquanto o 2PO tende a aumentar seu esforço de otimização à medida que a quantidade de relações aumenta, o GEQO foi projetado para apresentar uma quantidade aproximadamente constante de planos gerados para consultas acima de 7 relações. Observe que todos os gráficos desta figura estão em escala logarítmica de base 10, o que possibilita visualizar que para 100 relações, o GEQO gerou apenas pouco mais que 1% da quantidade de planos apresentada pelo 2PO.

Dada esta diferença na quantidade de planos gerados entre os dois otimizadores, um possível questionamento que poderia ocorrer é se o GEQO está operando em desvantagem de esforço frente ao 2PO, de modo que qualquer plano inferior gerado pelo GEQO pudesse ser justificado por este fato. Como forma de eliminar esta possível desvantagem, foi introduzido um segundo GEQO, representado pela sigla *GEQO+*, o qual foi configurado para gerar uma quantidade superior de planos.

Além dos dados obtidos pelo 2PO e GEQO, a Figura 7.1 também apresenta a quantidade média de planos gerados pelo GEQO+. Como é possível observar, este algoritmo

foi configurado para gerar aproximadamente a mesma quantidade de planos apresentada pelo 2PO. Esta configuração foi realizada individualmente em cada consulta, tomando por base os valores médios da quantidade de planos gerados pelo 2PO nas consultas com o mesmo número de relações e mesmo grafo de junções. Tanto o tamanho da população inicial quanto a quantidade de gerações do GEQO+ foram aumentadas em proporções iguais em cada configuração, usando para isso os parâmetros *geqo_pool_size* e *geqo_generations*. Assim como para os algoritmos 2PO e GEQO, o GEQO+ foi executado 10 vezes para cada consulta avaliada.

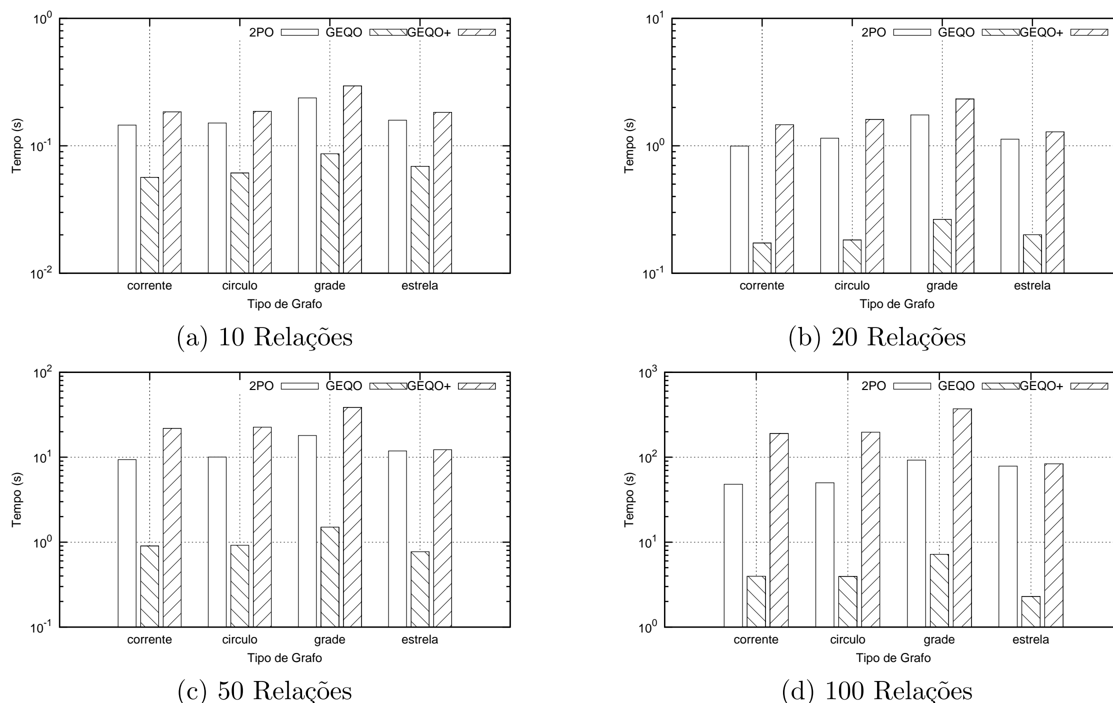


(a) Corrente (b) Círculo

(c) Grade (d) Estrela

Figura 7.1: Média da quantidade de planos gerados pelos otimizadores 2PO, GEQO e GEQO+ em função do número de relações e do grafo de junções.

Com relação ao tempo médio de otimização, todos os algoritmos demonstraram um comportamento similar à quantidade de planos gerados. A Figura 7.2 apresenta estes dados agrupados pela quantidade de relações de cada consulta. Nas consultas com 10 relações (gráfico 7.2a), a diferença entre os otimizadores não foi significativa. Já para as consultas maiores, como 50 e 100 relações (gráficos 7.2c e 7.2d), a diferença entre o 2PO e o GEQO foi de aproximadamente 10 vezes. A escala logarítmica dos gráficos demonstra claramente esta proporção.



(a) 10 Relações

(b) 20 Relações

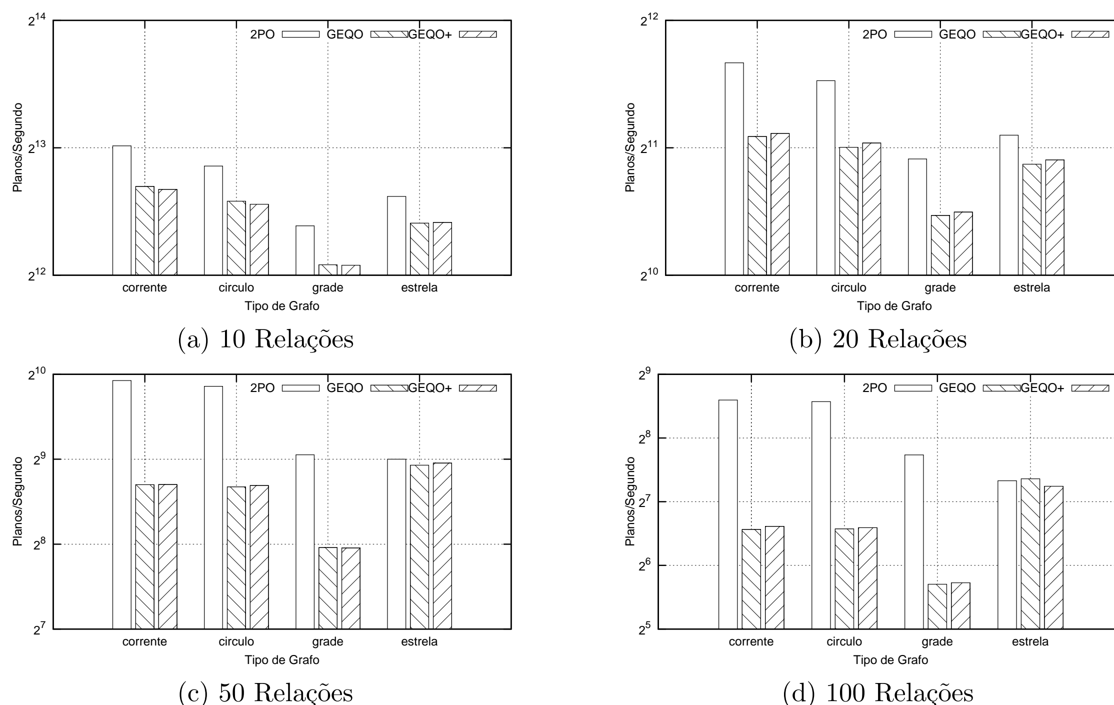
(c) 50 Relações

(d) 100 Relações

Figura 7.2: Tempo médio de otimização apresentado pelos otimizadores 2PO, GEQO e GEQO+ em função do número de relações e do grafo de junções.

Outro fato importante a ser notado na Figura 7.2, é o tempo de otimização apresentado pelo algoritmo GEQO+. Embora a quantidade de planos gerados por este algoritmo fosse aproximadamente a mesma em relação ao 2PO, seu tempo total de otimização foi relativamente maior para as consultas do tipo corrente, círculo e grade, chegando a 4 vezes o tempo gasto pelo 2PO nas consultas com 100 relações. Por outro lado, para as consultas do tipo estrela, seu tempo de otimização foi muito próximo do apresentado pelo 2PO.

Para verificar melhor este fato, a Figura 7.3 e a Tabela 7.1 apresentam a quantidade média de planos gerados por segundo em cada um dos três algoritmos. Os gráficos da Figura 7.3, apresentam os dados agrupados pela quantidade de relações de cada consulta, de modo a facilitar a comparação entre diferentes grafos de junções. Para verificar melhor a proporção dos valores apresentados nesta figura, a quantidade de planos por segundo está em escala logarítmica de base 2. Na Tabela 7.1, os mesmos dados estão dispostos em duas dimensões, sendo que as colunas representam o tipo do grafo de junções e as linhas a quantidade de relações de cada consulta.



(a) 10 Relações (b) 20 Relações (c) 50 Relações (d) 100 Relações

Figura 7.3: Média de planos gerados por segundo obtida pelos otimizadores 2PO, GEQO e GEQO+ em função do número de relações e do grafo de junções.

Em todos os gráficos da Figura 7.3, é possível observar que ambas as configurações do algoritmo GEQO apresentaram aproximadamente a mesma performance, independente da quantidade total de planos gerados por cada uma delas. Com relação à quantidade de relações de cada consulta, tanto o 2PO como o GEQO apresentaram uma diminuição gradativa da quantidade de planos gerados por segundo à medida que se aumentou a quantidade de relações.

Comparando a performance do GEQO em relação ao 2PO, observa-se na Figura 7.3 uma disparidade de comportamento de ambos os otimizadores em dois casos distintos. Primeiramente, para as consultas do tipo estrela, o GEQO apresentou uma quantidade de planos por segundo ligeiramente inferior ao 2PO para as consultas com 10 relações. À medida que se aumentou a quantidade de relações, a diferença entre o GEQO e o 2PO tendeu a diminuir. Para as consultas com 50 e 100 relações, praticamente não houve diferença de performance entre os otimizadores. Já para as consultas dos tipos corrente, círculo e grade com 10 relações, o GEQO também apresentou uma diferença pequena em relação ao 2PO. Contudo, à medida que se aumentou a quantidade de relações, esta

diferença tendeu ser mais acentuada ao invés de diminuir, como no caso anterior. Para as consultas com 100 relações, a quantidade de planos gerados por segundo apresentada pelo 2PO foi aproximadamente 4 vezes maior que o apresentado pelo GEQO.

Tabela 7.1: Média de planos gerados por segundo obtida pelos otimizadores 2PO, GEQO e GEQO+ em função do número de relações e do grafo de junções.

N	Alg.	*Corrente*	*Círculo*	*Grade*	*Estrela*
10	2PO	8323,3	7455,7	5405,8	6352,0
	GEQO	6645,3	6132,4	4352,9	5630,1
	GEQO+	6533,8	6026,9	4339,1	5649,2
20	2PO	3271,8	2961,5	1940,3	2229,1
	GEQO	2178,4	2055,4	1419,0	1990,1
	GEQO+	2214,7	2102,3	1444,9	2037,5
50	2PO	975,8	932,8	540,1	526,1
	GEQO	416,7	409,1	249,5	512,1
	GEQO+	417,0	413,6	248,6	522,0
100	2PO	387,2	381,3	216,9	164,3
	GEQO	95,0	95,6	52,3	168,2
	GEQO+	98,1	96,8	53,1	154,7

Em ambos os casos citados acima, a diferença de comportamento entre o algoritmo 2PO e GEQO sugere, de modo geral, uma maior robustez do otimizador 2PO no processo de construção de planos, independente da conectividade do grafo de junções de uma consulta. Para o algoritmo GEQO, contudo, a diminuição cada vez mais acentuada de sua performance para consultas de baixa conectividade, pode ser o indicativo de uma possível deficiência deste algoritmo na forma representativa de seus planos.

7.2 Qualidade dos Planos Gerados

Esta seção apresenta um comparativo da qualidade dos planos obtidos pelos otimizadores 2PO, GEQO e GEQO+, seguindo para isso as mesmas dimensões utilizadas na seção anterior. Como forma de representar esta qualidade, os planos obtidos por cada otimizador foram convertidos em suas respectivas escalas de custo. Como mencionado no Capítulo 5, a escala de custo de um plano é a razão entre seu custo e o custo do melhor plano encontrado para a mesma consulta, independente do otimizador utilizado.

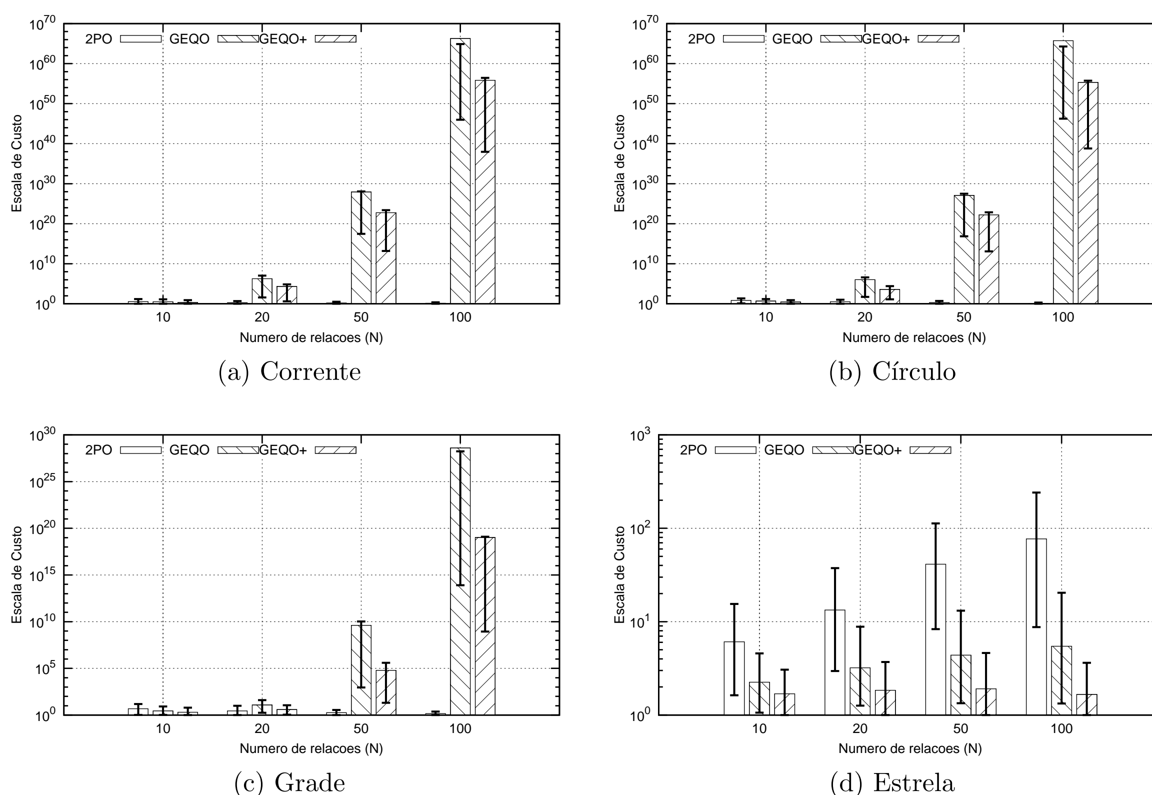
Uma vez que os otimizadores aqui avaliados não se comportam de forma determinística para a geração de planos, a comparação entre eles foi realizada a partir da média e do intervalo de variação das respectivas escalas de custo obtidas em cada uma das dimensões avaliadas. Como forma de representar o intervalo de variação das escalas de custo, foram calculados o 5º e o 95º percentis de cada conjunto de planos [21]. Este intervalo de variação corresponde a 90% dos planos obtidos, o que possibilita observar com mais detalhes sua distribuição.

As Figuras 7.4 e 7.5 apresentam a média e a variação das escalas de custo obtidas pelos otimizadores em função da quantidade de relações e do grafo de junções de cada consulta. Em ambas as figuras, os dados apresentados são os mesmos, contudo, sob diferentes ângulos de visão. Nos gráficos da Figura 7.4, os dados estão agrupados pelos respectivos grafos de junções, enquanto que na Figura 7.5, o agrupamento desses dados é pela quantidade de relações. Em todos os gráficos, os valores médios das escalas de custo estão representados pelos retângulos correspondentes a cada otimizador. No topo ou no interior de cada retângulo, é apresentado o intervalo de valores correspondente ao 5º e o 95º percentis das escalas de custo. O valor mais baixo de cada intervalo corresponde ao 5º percentil e o mais alto o 95º. Em todos os gráficos apresentados nessas duas figuras, as escalas de custo estão dispostas em escala logarítmica de base 10.

Como forma de facilitar ainda mais esta análise, os valores médios das escalas de custo também estão dispostos na Tabela 7.2. Cada linha desta tabela representa o agrupamento da Figura 7.5 e cada coluna, o agrupamento da Figura 7.4. No título de cada coluna da Tabela 7.2, além do nome dos tipos de grafos de junções avaliados, constam também as respectivas quantidades de arestas em função do número de relações de cada consulta. Para as consultas do tipo grade e estrela, esta quantidade de arestas é baseada na transitividade dos predicados de junção, conforme descrito no Capítulo 6.

Em uma primeira análise dos gráficos apresentados nas Figuras 7.4 e 7.5, observa-se que ambas as configurações do GEQO tiveram um comportamento similar em todas as consultas avaliadas. Embora o GEQO+ fosse configurado para gerar uma quantidade significativamente maior de planos, sua qualidade final foi timidamente melhor que o

resultado apresentado pelo GEQO em sua configuração original. Este comportamento pode ser observado tanto nos valores médios das escalas de custo como em sua variação. Isto demonstra que a quantidade de esforço empregada pelo GEQO não apresenta ganhos significativos na qualidade de seus planos. Porém, conforme mencionado anteriormente, a adição do GEQO+ nesta avaliação serviu para verificar qualquer deficiência de esforço deste otimizador em sua configuração original.



(a) Corrente

(b) Círculo

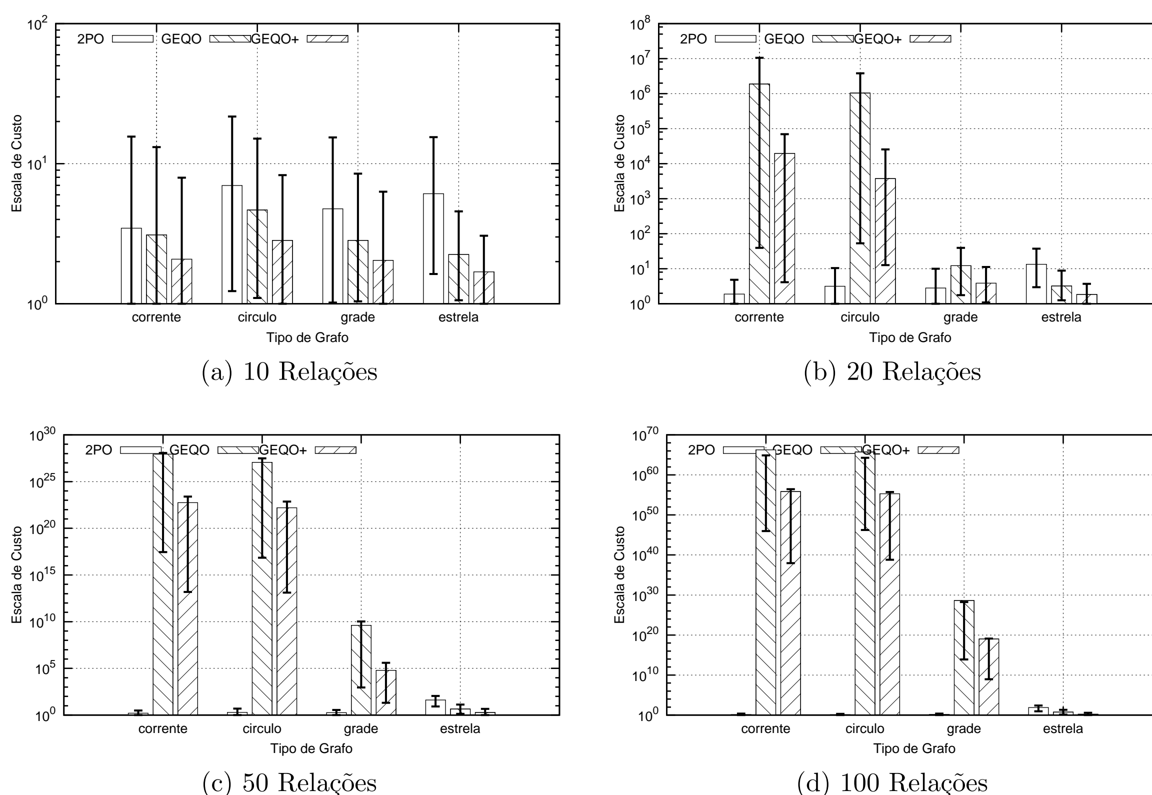
(c) Grade

(d) Estrela

Figura 7.4: Média e variação das escalas de custo obtidas pelos otimizadores 2PO, GEQO e GEQO+ em função da quantidade de relações e do grafo de junções. Os gráficos nesta figura agrupam os dados de acordo com os grafos de junções. Cada retângulo representa a média das escalas de custo, enquanto que o intervalo apresentado em seu topo ou interior representa o 5º e o 95º percentis.

Comparando a qualidade dos planos obtidos entre os otimizadores 2PO e GEQO, observa-se que ambos se alternaram na geração das melhores médias de escalas de custo. A Tabela 7.2 demonstra claramente dois grupos de consultas. Na parte grifada desta tabela, compreendendo nove dos dezesseis conjuntos de consultas avaliados, estão as consultas do tipo corrente, círculo e grade com relações maiores ou iguais a 20. Para estas consultas, o 2PO demonstrou planos com qualidade superior aos apresentados pelo GEQO. Também

neste caso, observa-se que o GEQO apresentou uma degradação acentuada na qualidade de seus planos à medida que se aumentou a quantidade de relações. Nos dois casos mais extremos, como observado nas consultas do tipo corrente e círculo com 100 relações, a diferença de qualidade dos planos obtidos entre o 2PO e as duas configurações do GEQO foi significativamente grande, de modo que torna-se difícil representar, nos gráficos 7.4a e 7.4b, a presença da qualidade dos planos obtidos pelo 2PO.



(a) 10 Relações (b) 20 Relações

(c) 50 Relações (d) 100 Relações

Figura 7.5: Média e variação das escalas de custo obtidas pelos otimizadores 2PO, GEQO e GEQO+ em função da quantidade de relações e do grafo de junções. Os gráficos nesta figura agrupam os dados de acordo com a quantidade de relações. Cada retângulo representa a média das escalas de custo, enquanto que o intervalo apresentado em seu topo ou interior representa o 5º e o 95º percentis.

Durante o início desta avaliação, os planos obtidos foram também submetidos para a execução no SGBD. Contudo, alguns dos planos apresentados pelo GEQO provocaram a queda do sistema, possivelmente por esgotamento de memória principal ou por sobrecarga de acesso à memória secundária. Para evitar o risco de interrupção do ambiente de testes toda vez que alguma consulta deste tipo fosse posta em execução, o tempo total de execução de cada consulta não foi computado.

Nos gráficos 7.5b até 7.5d, observa-se também que a degradação dos planos obtidos pelo GEQO está fortemente relacionada com a quantidade de arestas de cada grafo de junções. Quanto menor a quantidade de arestas de uma consulta, pior foi a qualidade dos planos obtidos pelo GEQO. Entre as consultas do tipo corrente ($N-1$ arestas) e círculo (N arestas), o que se observa é uma pequena melhoria na qualidade dos planos. Para as consultas do tipo grade ($2N-3$ arestas), a melhoria na qualidade dos planos é mais evidente. Por fim, as consultas do tipo estrela, com a quantidade máxima de arestas, apresentaram as melhores escalas de custo, sendo inclusive melhores que as escalas de custo apresentadas pelo 2PO.

Tabela 7.2: Média das escalas de custo obtidas pelos otimizadores 2PO, GEQO e GEQO+ em função da quantidade de relações e do grafo de junções.

N	Alg.	*Corrente* $N-1$	*Círculo* N	*Grade* $2N-3$	*Estrela* $N(N-1)/2$
10	2PO	3,5	7,0	4,8	6,1
	GEQO	3,1	4,7	2,8	2,3
	GEQO+	2,1	2,8	2,0	1,7
20	2PO	1,9	3,2	2,8	$1{,}3 \cdot 10^{1}$
	GEQO	$1{,}9 \cdot 10^{6}$	$1{,}0 \cdot 10^{6}$	$1{,}2 \cdot 10^{1}$	3,2
	GEQO+	$2{,}0 \cdot 10^{4}$	$3{,}8 \cdot 10^{3}$	3,9	1,8
50	2PO	1,6	2,0	1,8	$4{,}1 \cdot 10^{1}$
	GEQO	$8{,}8 \cdot 10^{27}$	$1{,}1 \cdot 10^{27}$	$4{,}1 \cdot 10^{9}$	4,4
	GEQO+	$5{,}6 \cdot 10^{22}$	$1{,}6 \cdot 10^{22}$	$6{,}4 \cdot 10^{4}$	1,9
100	2PO	1,4	1,3	1,4	$7{,}7 \cdot 10^{1}$
	GEQO	$1{,}9 \cdot 10^{66}$	$5{,}4 \cdot 10^{65}$	$4{,}1 \cdot 10^{28}$	5,5
	GEQO+	$6{,}8 \cdot 10^{55}$	$2{,}0 \cdot 10^{55}$	$1{,}0 \cdot 10^{19}$	1,7

Embora a qualidade dos planos apresentados pelo 2PO seja superior aos obtidos pelo GEQO para a maioria das consultas avaliadas, existem contudo dois casos em que este fato não ocorreu. O primeiro caso refere-se às consultas com apenas 10 relações, independente do grafo de junções. Já o segundo, como mencionado anteriormente, foi para todas as consultas do tipo estrela, independente da quantidade de relações. Ambos os casos correspondem à parte não grifada da Tabela 7.2.

Mesmo que o 2PO tenha gerado planos com qualidade inferior ao obtidos pelo GEQO nos dois casos apresentados acima, a diferença entre suas escalas de custo não foi tão

significativa como nos casos apresentados pelo GEQO na parte grifada da Tabela 7.2. Para as consultas do tipo corrente, círculo e grade com 10 relações (gráfico 7.5a), o que se observa é uma maior variação das escalas de custo obtidas pelo 2PO em relação ao GEQO. Esta variação demonstra ser mais acentuada na parte superior de cada intervalo, representada pelo 95º percentil. Contudo, na parte inferior desses intervalos, representada pelo 5º percentil, observa-se que o 2PO apresentou valores muito próximos dos obtidos pelo GEQO.

Com relação ao grafo de junções, as consultas do tipo corrente, círculo e grade com 10 relações não apresentaram uma distinção muito clara no que se refere ao comportamento de ambos os otimizadores. O pior caso observado parece ser para as consultas do tipo círculo, onde todos os otimizadores apresentaram um valor médio mais elevado para suas respectivas escalas de custo, como pode ser observado na Tabela 7.2. Já para as demais consultas, ambas as configurações do GEQO apresentaram uma pequena diminuição de suas escalas de custo para as consultas do tipo grade em relação as consultas do tipo corrente. Por outro lado, observa-se que o 2PO apresentou um comportamento inverso para esses dois tipos de consultas.

Para todas as consultas do tipo estrela (gráfico 7.4d), percebe-se que a qualidade dos planos obtidos pelo 2PO tendeu a diminuir, se comparada a ambas as configurações do GEQO, à medida que se aumentou a quantidade de relações de cada consulta. Este comportamento foi observado tanto pelos valores médios das escalas de custo como pelos respectivos intervalos de variação.

Uma vez que as consultas do tipo estrela são consideradas como grafos completos para ambos os otimizadores, a quantidade de possíveis planos sem a existência de produtos cartesianos é significativamente maior se comparada aos demais tipos de consultas avaliados aqui. Este fato parece ter maior influência sobre o 2PO, uma vez que este pode representar todas as possíveis árvores binárias de junção deste espaço de busca. Em contra-partida, a forma representativa de planos utilizada pelo GEQO, baseia-se exclusivamente em listas de relações. Nesta representação, o GEQO somente gera árvores em profundidade para consultas do tipo estrela, o que é apenas um subconjunto dos possíveis planos representa-

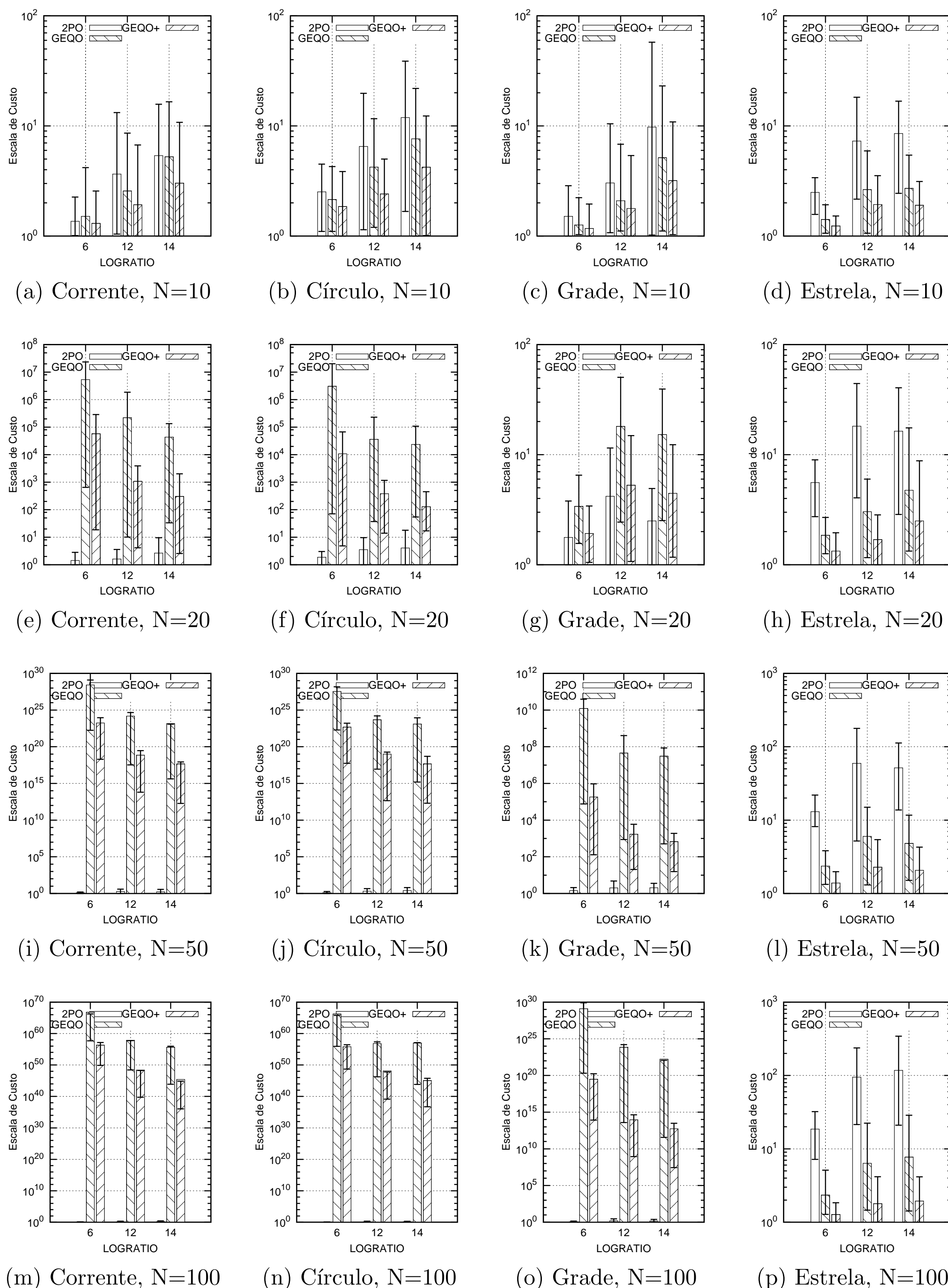
dos pelo 2PO. Apesar disso, observa-se que mesmo que a configuração original do GEQO tenha gerado uma quantidade significativamente menor de planos, a qualidade final dos mesmos é relativamente superior aos planos obtidos pelo 2PO. Isto sugere que o espaço de busca utilizado pelo GEQO neste tipo de consulta, é muito mais eficiente para se obter bons planos do que o espaço de busca utilizado pelo 2PO.

7.3 Variação das Cardinalidades das Relações

Esta seção apresenta a qualidade dos planos obtidos pelos otimizadores 2PO, GEQO e GEQO+ em um nível mais detalhado que o apresentado na seção anterior. Para isso, além de considerar o número de relações e o grafo de junções, as consultas foram divididas de acordo com suas respectivas variações de cardinalidades, indicadas por seus valores de $LOGRATIO$. Os dados apresentados aqui correspondem a 48 pontos de observação, sendo cada um deles o equivalente a 10 consultas aleatoriamente selecionadas.

Na Figura 7.6, são apresentados os valores médios e as variações das escalas de custo obtidas por cada otimizador. Esta figura é composta por dezesseis gráficos que estão organizados em quatro linhas, representado as quantidades de relações (N), e quatro colunas, representando os grafos de junções. Dentro de cada gráfico, assim como na seção anterior, constam os valores médios e a variação das escalas de custo obtidas por cada otimizador. Neste caso, o eixo "x" de cada gráfico representa os três valores de $LOGRATIO$ avaliados.

A primeira observação que merece ser citada aqui sobre gráficos da Figura 7.6, está relacionada ao comportamento relativo dos otimizadores. Em quase todos os gráficos desta figura, o que se observa é uma mesma ordem relativa da qualidade dos planos obtidos por cada otimizador, independente do valor de $LOGRATIO$ avaliado. Ou seja, nos gráficos onde o 2PO apresentou planos com qualidade inferior a ambas as configurações do GEQO, este comportamento se observa para todos os valores de $LOGRATIO$. Este mesmo comportamento também ocorreu para os gráficos onde o 2PO apresentou os melhores planos. A única exceção encontrada foi para o gráfico 7.6a, que corresponde às consultas do tipo corrente com 10 relações. Neste caso, o 2PO apresentou um comportamento



(a) Corrente, N=10 (b) Círculo, N=10 (c) Grade, N=10 (d) Estrela, N=10

(e) Corrente, N=20 (f) Círculo, N=20 (g) Grade, N=20 (h) Estrela, N=20

(i) Corrente, N=50 (j) Círculo, N=50 (k) Grade, N=50 (l) Estrela, N=50

(m) Corrente, N=100 (n) Círculo, N=100 (o) Grade, N=100 (p) Estrela, N=100

Figura 7.6: Média e variação das escalas de custo obtidas pelos otimizadores 2PO, GEQO e GEQO+ em função da quantidade de relações, grafo de junções e *LOGRATIO*. Cada retângulo representa a média das escalas de custo, enquanto que o intervalo apresentado em seu topo ou interior representa o 5º e o 95º percentis.

relativo ligeiramente diferente, em relação ao GEQO, para as consultas com *LOGRATIO* igual a 6.

Com relação ao comportamento individual de cada otimizador, percebe-se que as diferentes variações de cardinalidades influenciaram principalmente na variação das respectivas escalas de custo obtidas. Para o otimizador 2PO, as consultas com *LOGRATIO* igual a 6 renderam variações nitidamente menores do que o apresentado pelos demais valores de *LOGRATIO*. Por outro lado, entre os *LOGRATIOS*s 12 e 14, não se tem uma definição muito clara de comportamento. Em alguns casos, observa-se inclusive que as consultas com *LOGRATIO* igual a 14 produziram variações menores que as consultas de *LOGRATIO* igual a 12.

Considerando o comportamento da variação dos planos obtidos pelo GEQO em função da variação das cardinalidades, percebe-se dois casos distintos na Figura 7.6. Em 50% dos casos, correspondendo às consultas com 10 relações, mais as consultas do tipo estrela e mais as consultas do tipo grade com 20 relações, o comportamento das respectivas variações das escalas de custo obtidas pelo GEQO em função dos valores de *LOGRATIO* foi semelhante ao apresentado pelo 2PO. Para os demais casos, observa-se que as consultas com *LOGRATIO* igual a 6 foram as que renderam tanto as maiores médias como as maiores variações de suas respectivas escalas de custo. Para os valores de *LOGRATIO* iguais a 12 e 14, assim como apresentado anteriormente para o 2PO, seu comportamento não parece muito bem definido.

Por fim, comparando ambas as configurações do GEQO, os gráficos da Figura 7.6 não demonstram um ganho significativo da qualidade dos planos obtidos pelo GEQO+ em função de seu aumento na quantidade de planos gerados.

7.4 Análise dos Resultados Apresentados

Com base nos dados apresentados nas seções anteriores, é possível destacar alguns pontos considerados relevantes para este estudo. Em primeiro lugar, a quantidade de planos gerados por cada otimizador não pareceu ser um fator determinante para a qualidade final de seus planos. Como foi observado nas Seções 7.2 e 7.3, as escalas de custo apre-

sentadas pelo GEQO para as consultas do tipo estrela, foram relativamente melhores do que as escalas de custo obtidas pelo 2PO. Este fato ocorreu mesmo com uma quantidade significativamente menor de planos gerados pelo GEQO. Além disso, a distância entre as escalas de custo obtidas por esses otimizadores tendeu a aumentar com o aumento da quantidade de relações de cada consulta.

Os dados apresentados nas seções anteriores demonstraram ainda um comportamento deficiente do GEQO para consultas com baixa conectividade em seus respectivos grafos de junções. Para as consultas do tipo corrente, círculo e grade, este fato pôde ser observado tanto em sua capacidade de geração de planos por segundo como pela rápida degradação da qualidade dos mesmos. Mesmo com uma quantidade significativamente maior de planos gerados, o GEQO+ apresentou melhorias pouco relevantes para qualidade final de seus planos. Somente para as consultas com apenas 10 relações é que esta deficiência não foi demonstrada, provando ser este um limite aceitável para a utilização deste otimizador.

A deficiência apresentada pelo GEQO para as consultas com baixa conectividade parece estar relacionada com a forma representativa de seus planos. Este otimizador utiliza listas de relações para esta representação, o qual é um artifício geralmente útil para otimizadores que operam exclusivamente no espaço de busca de árvores em profundidade à esquerda [30, 43, 46]. Isto parece funcionar perfeitamente bem para as consultas do tipo estrela, uma vez que não existem representações inválidas de planos causadas por produtos cartesianos. Por outro lado, para as consultas com menos conectividade, a quantidade de planos inválidos é considerável.

Como mencionado no Capítulo 4, o GEQO tenta contornar o problema da geração de planos inválidos através de um artifício que consiste na quebra de uma sequência de relações em várias partes. Embora este artifício possa diminuir a quantidade de planos inválidos, o que se observou neste capítulo é que a qualidade final dos planos obtidos por este otimizador continua insatisfatória para a maioria dos casos em que isto foi necessário. Possivelmente, apenas para as consultas com 10 relações é que sua aplicação tenha sido realmente útil.

Por outro lado, o otimizador 2PO demonstrou ser superior ao GEQO para a maioria

das consultas avaliadas. Sua forma representativa de planos, baseada puramente em árvores binárias de junções, demonstrou ser mais robusta que a forma utilizada pelo GEQO. Isto foi observado tanto na eficiência do 2PO na geração de planos por segundo para as consultas do tipo corrente, círculo e grade, como na qualidade final dos planos obtidos. Mesmo assim, os resultados apresentados pelo GEQO para as consultas do tipo estrela, são um claro indicador de que algumas melhorias podem ser realizadas na estratégia de busca do 2PO de modo a ser mais competitivo para tais tipos de consultas.

CAPÍTULO 8

CONCLUSÃO

A otimização de junções é a parte do processamento de consultas que representa um impacto significativo sobre a eficiência dos SGBDs relacionais. Este estudo buscou apresentar a natureza deste problema, dando maior ênfase à consultas com um grande número de relações.

No PostgreSQL, foi possível observar, a partir da implementação do algoritmo 2PO e de sua posterior comparação com o algoritmo GEQO, diversos fatores importantes que merecem ser mencionados:

1. **A construção de algoritmos de otimização de junções depende da arquitetura de representação interna do SGBD.** Durante a implementação do 2PO, a arquitetura do PostgreSQL, principalmente com relação a estrutura *RelOptInfo* e a função *make_join_rel*, implicou em algumas adequações com relação a forma representativa de árvores de junções e movimentos adotados originalmente pelos autores Ioannidis e Kang [29].

2. **A comparação de algoritmos não exaustivos é uma tarefa complexa de ser realizada, principalmente pela grande quantidade de fatores envolvidos.** Existem muitas dimensões que podem compor um esquema de testes e várias características que descrevem o comportamento de algoritmos de otimização, implicando assim na geração de um grande volume de dados. Além disso, a natureza não determinística dos algoritmos avaliados requeriu que seus comportamentos fossem observados de forma estatística.

3. **O espaço de busca adotado por cada algoritmo de otimização parece produzir um impacto muito maior sobre sua qualidade final dos planos gerados do que a quantidade de esforço empregada em sua otimização.** Tanto

o GEQO como o GEQO+ apresentaram uma disparidade relativamente pequena com relação a qualidade dos planos obtidos, mesmo com uma diferença significativa entre a quantidade de planos gerados por cada um deles. Nem com relação ao 2PO, as duas configurações do GEQO apresentaram um comportamento diferente.

4. **O 2PO apresentou maior robustez para a maioria dos casos avaliados.** Principalmente em consultas com um grande número de relações e com baixa conectividade de seus respectivos grafos de junções, o 2PO apresentou uma qualidade de planos significativamente melhor em relação ao GEQO. Nesses casos, o GEQO demonstrou sérios problemas em explorar os possíveis planos. Contudo, para consultas do tipo estrela, onde a conectividade de seus grafos de junções foi total, o GEQO demonstrou ser muito eficiente.

De um modo geral, os resultados apresentados neste estudo não sugerem que o algoritmo 2PO, implementado para o PostgreSQL, esteja apto para substituir o atual algoritmo GEQO. Existem ainda outras questões que precisam ser avaliadas detalhadamente, como por exemplo o uso de agregações e ordenações e o uso de consultas recursivas. Mesmo assim, espera-se que estes resultados possam servir como base tanto para a melhoria de ambos os algoritmos como para elaboração de novas abordagens de otimização. Tanto o GEQO como o 2PO demonstraram pontos positivos e negativos nos testes realizados, os quais não devem ser desconsiderados.

Uma vez permitida pela própria arquitetura do PostgreSQL, o algoritmo 2PO está disponível na forma de um *plugin*, chamado LJQO [4]. Assim como a maioria dos projetos de software livre, este *plugin* pode ser obtido pela Internet por qualquer pessoa interessada em avaliá-lo ou melhorá-lo. Além disso, sua estrutura está preparada para receber novos otimizadores, os quais poderão, futuramente, serem incorporados a este projeto sem grandes esforços.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] A Tour of PostgreSQL Internals. Disponível em: http://www.postgresql.org/files/developer/tour.pdf, acessado em 27/06/2010.

[2] Lista de discussão dos desenvolvedores do PostgreSQL. Disponível em: http://archives.postgresql.org/pgsql-hackers, acessado em 27/06/2010.

[3] Lista de tarefas pendentes do PostgreSQL. Disponível em: http://wiki.postgresql.org/wiki/Todo, acessado em 27/06/2010.

[4] Plugin LJQO para o PostgreSQL. Disponível em: http://git.c3sl.ufpr.br/gitweb?p=lbd/ljqo.git;a=summary.

[5] PostgreSQL, object-relational database management system. Disponível em: http://www.postgresql.org, acessado em 27/06/2010.

[6] Projeto Mozilla Firefox. Disponível em: http://www.mozilla.com/firefox/, acessado em 27/06/2010.

[7] Repositório PostgreSQL (branch REL8_3_STABLE) – Genetic Query Optimizer. Disponível em: http://git.postgresql.org/gitweb?p=postgresql.git;a=tree;f=src/backend/optimizer/geqo;hb=REL8_3_STABLE, acessado em 27/06/2010.

[8] Transaction Processing Performance Council. Disponível em: http://www.tpc.org/, acessado em 24/04/2010.

[9] Transaction Processing Performance Council. TPC Benchmark DS (Decision Support). Disponível em: http://www.tpc.org/tpcds, acessado em 24/04/2010.

[10] Transaction Processing Performance Council. TPC Benchmark E (OLTP). Disponível em: http://www.tpc.org/tpce, acessado em 24/04/2010.

[11] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. Gray, P. P. Griffiths, W. Frank King III, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu,

I. L. Traiger, B. W. Wade, e V. Watson. System r: Relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, 1976.

[12] M. M. Astrahan e D. D. Chamberlin. Implementation of a structured english query language. *Commun. ACM*, 18(10):580–588, 1975.

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

[14] Tarcizio A. Bini, Adriano Lange, Marcos Sfair Sunye, e Fabiano Silva. Stableness in large join query optimization. *ISCIS*, páginas 639–644, 2009.

[15] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, e R. A. Yost. A history and evaluation of system r. *Commun. ACM*, 24(10):632–646, 1981.

[16] Y. Chen, R. L. Cole, W. J. McKenna, S. Perfilov, A. Sinha, e E. Szedenits Jr. Partial join order optimization in the paraccel analytic database. *SIGMOD Conference*, páginas 905–908, 2009.

[17] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[18] E. F. Codd. Relational completeness of data base sublanguages. *In: R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California*, 1972.

[19] E. F. Codd. Relational database: A practical foundation for productivity. *Commun. ACM*, 25(2):109–117, 1982.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, e C. Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2001.

[21] J. S. da Fonseca e G. de Andrade Martins. *Curso de Estatística*, capítulo Estatística Descritiva, páginas 101–165. Atlas, R. Conselheiro Nébias, 1384, 1996.

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

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

[24] G. Graefe e W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. *ICDE*, páginas 209–218, 1993.

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

[26] G. D. Held, M. Stonebraker, e E. Wong. Ingres–a relational data base management system. *AFIPS 1975 NCC*, volume 44, páginas 409–416, Montvale, N.J., 1975. AFIPS Press.

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

[28] Y. E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.

[29] Y. E. Ioannidis e Y. C. Kang. Randomized algorithms for optimizing large join queries. *SIGMOD Rec.*, 19(2):312–321, 1990.

[30] Y. E. Ioannidis e Y. C. Kang. Left-deep vs. bushy trees: an analysis of strategy spaces and its implications for query optimization. *SIGMOD Rec.*, 20(2):168–177, 1995.

[31] Y. E. Ioannidis e E. Wong. Query optimization by simulated annealing. *SIGMOD Conference*, páginas 9–22, 1987.

[32] M. Jarke e J. Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2):111–152, 1984.

[33] R. Krishnamurthy, H. Boral, e C. Zaniolo. Optimization of non-recursive queries. *In proc. of the Conference on Very Large Data Bases (VLDB)*, 2:128–137, 1986.

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

[35] R. S. G. Lanzelotte, P. Valduriez, e M. Zaït. On the effectiveness of optimization search strategies for parallel execution spaces. *VLDB*, páginas 493–504, 1993.

[36] P. Mishra e M. H. Eich. Join processing in relational databases. *ACM Comput. Surv.*, 24(1):63–113, 1992.

[37] T. Neumann. Query simplification: graceful degradation for join-order optimization. *SIGMOD Conference*, páginas 403–414, 2009.

[38] K. Ono e G. M. Lohman. Measuring the complexity of join enumeration in query optimization. *VLDB*, páginas 314–325, 1990.

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

[40] S. Russell e P. Norvig. *Artificial Intelligence: A Modern Approach*, capítulo Constraints Satisfaction Problems, páginas 115. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

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

[42] L. D. Shapiro, D. Maier, P. Benninghoff, K. Billings, Y. Fan, K. Hatwal, Q. Wang, Y. Zhang, H. Wu, e B. Vance. Exploiting upper and lower bounds in top-down query optimization. *IDEAS*, páginas 20–33, 2001.

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

[44] M. Stonebraker. Retrospection on a database system. *ACM Trans. Database Syst.*, 5(2):225–240, 1980.

[45] M. Stonebraker e L. A. Rowe. The design of postgres. *SIGMOD Conference*, páginas 340–355, 1986.

[46] A. Swami e A. Gupta. Optimization of large join queries. *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, páginas 8–17, New York, NY, USA, 1988. ACM.

[47] Tarcizio A. Bini. Aplicação do Algoritmo de Kruskal na Otimização de Consultas com Múltiplas Junções Relacionais. Dissertação de mestrado, Departamento de Informática, UFPR, Abril de 2009.

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

[49] J. D. Ullman, H. Garcia-Molina, e J. Widom. *Database Systems: The Complete Book*, capítulo Relational Algebra, páginas 189–237. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[50] J. D. Ullman, H. Garcia-Molina, e J. Widom. *Database Systems: The Complete Book*, capítulo The Query Compiler, páginas 787–874. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[51] J. D. Ullman, H. Garcia-Molina, e J. Widom. *Database Systems: The Complete Book*, capítulo Query Executor, páginas 713–785. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[52] M. Utesch. Postgresql documentation – genetic query optimizer. Relatório técnico, PostgreSQL Global Development Group. Disponível em: `http://www.postgresql.org/docs/current/static/geqo.html`, acessado em 27/06/2010.

[53] B. Vance e D. Maier. Rapid bushy join-order optimization with cartesian products. *SIGMOD Conference*, páginas 35–46, 1996.

[54] L. D. Whitley. The genitor algorithm and selection pressure: why rank-based allocation of reproductive trials is best. *Proceedings of the third international conference on Genetic algorithms*, páginas 116–121, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[55] L. D. Whitley, T. Starkweather, e D'Ann Fuquay. Scheduling problems and traveling salesmen: The genetic edge recombination operator. *ICGA*, páginas 133–140, 1989.