OTIMIZAÇÃO DE PROCESSAMENTO DE EXPRESSÕES DE CAMINHO

André Ormastroni Victor

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Marta Lima de Queirós Mattoso, D.Sc.

Prof. Ana Maria de Carvalho Moura, D.Ing.

Prof. Claudio Esperança , Ph.D.

RIO DE JANEIRO, RJ - BRASIL

DEZEMBRO DE 2001

VICTOR, ANDRÉ ORMASTRONI

Otimização de Processamento de Expressões de Caminho [Rio de Janeiro] 2001

IX, 99 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2001)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Expressões de Caminho
2. Processamento de Consultas
3. Sistemas Gerenciadores de Banco de Dados Orientados a Objetos

I. COPPE/UFRJ II. Título (série)

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

OTIMIZAÇÃO DE PROCESSAMENTO DE EXPRESSÕES DE CAMINHO

André Ormastroni Victor

Dezembro/2001

Orientadora: Marta Lima de Queirós Mattoso
Programa: Engenharia de Sistemas e Computação

A maioria das linguagens de consulta sobre dados estruturados e semi-estruturados apoia a formulação de consultas contendo expressões de caminho. Expressões de caminho permitem especificar uma navegação entre os objetos armazenados em diferentes coleções. Basicamente, a maioria das propostas de algoritmos para o processamento de expressões de caminho concentra-se na adaptação dos algoritmos baseados no operador de junção relacional. Muitos autores descartam o algoritmo clássico *Naïve Pointer Chasing* (NP) por considerá-lo uma estratégia ingênua e ineficiente no acesso a disco dos objetos do caminho.

Neste trabalho apresentamos um novo algoritmo propondo uma otimização sobre o algoritmo NP. Sua principal contribuição é melhorar o desempenho de E/S ao evitar a execução de caminhos já percorridos. Este novo algoritmo, chamado de *Smart Naïve* (SN), ainda mantém o baixo custo de CPU característico do NP. Para comprovar sua eficiência, foram realizados experimentos sobre diferentes configurações de base do *benchmark* OO7. Os resultados obtidos com o SN são comparados com o NP e com o melhor algoritmo baseado em junção proposto na literatura. Os experimentos mostraram o comportamento dos algoritmos e diversas situações não exploradas nos trabalhos da literatura. A análise desses resultados nos permitiu propor heurísticas a serem utilizadas por otimizadores de expressões de caminho. Os algoritmos e técnicas de otimizaçao abordados neste trabalho podem ser estendidos em outros contextos de aplicação, como por exemplo o processamento de consultas sobre documentos XML.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (D.Sc.)

PATH EXPRESSIONS PROCESSING OPTIMIZATION

André Ormastroni Victor

December/2001

Advisor: Marta Lima de Queirós Mattoso
Department: Computer Science and Systems Engineering

Most query languages for structured and semi-structured data support writing queries with path expressions. Path expressions provide a mean for specifying navigation between objects stored in data collections. Basically, most algorithm proposals for path expression processing focus on relational join based algorithms. Their authors discard the classical *Naïve Pointer Chasing* (NP) algorithm because they consider it a naïve and inefficient disk access strategy for traversing objects in the path.

In this work we present a new algorithm based on optimized version algorithm NP. Its main contribution is to increase I/O performance by avoinding to traverse paths already evaluated. This new algorithm, called *Smart Naïve* (SN), still presents the typical low CPU costs from NP. Different experiments done on OO7 *benchmark*'s base configurations were used for showing its efficiency. The SN results are compared to NP and to the best join based algorithm proposed on literature. The experiments show the algorithms behavior and many scenarios not explored in other works in the literature. The analysis of results allows us to propose heuristics that can be used for path expression optmizers. The algorithms and optimization techinques studied in this work can be extended into other aplication contexts, such as query processing over XML documents, for example.

Agradecimentos

Em primeiro lugar, agradeço a Deus por ter me propiciado saúde e condições para a concretização de mais uma etapa em minha vida.

À profª. Marta Mattoso por sua confiança em meu trabalho e atenciosa orientação durante todo o curso de mestrado. Em todos os momentos de minha pesquisa, principalmente nos que me via mais "perdido", seus conselhos e orientação formaram o alicerce para o sucesso deste trabalho. Realmente, não há palavras que expressem o meu reconhecimento e admiração por seu trabalho e dedicação à vida acadêmica.

Aos membros da banca examinadora, os professores Claudio Esperança e Ana Moura, pelas críticas pertinentes e valiosas sugestões que, certamente, enriqueceram a tese e serão importantes na elaboração de futuros trabalhos. Em particular, ao profº. Claudio Esperança, pelas valiosas discussões ocorridas durante uma de suas disciplinas, que germinaram algumas das idéias contidas nesta dissertação.

Ao nosso grupo de estudo em processamento de consultas chefiado pela profª. Marta Mattoso. As inúmeras reuniões e discussões realizadas pelo grupo foram muito importantes para o amadurecimento da proposta da tese. Em particular, a Fernanda Baião, por sua ajuda em diversos trabalhos e pela amizade que nos proporcionou momentos sempre agradáveis e divertidos desfrutados fora do ambiente acadêmico.

Aos amigos da linha de BD e ES, Marcelo, Robson, Melise, Alê, Carlete, Flavia, Gustavo, Marcio Duran, Catarina, Mangan, Alexandre Dantas, Gustavo e Marcio Barros, pela amizade, torcida e momentos de descontração. Agradeço, em particular, ao Robson por sua valiosa e generosa ajuda na depuração dos *bugs* do GOA. Sua ajuda foi fundamental para que meu esforço fosse mais concentrado na implementação dos algoritmos.

Aos amigos da turma *.93 da UFF, especialmente Bazílio, Dada, Lene, Roberta, Francine, Érika, Romário, Moska, Adriana e Rômulo, pela amizade sincera e incentivo nos momentos mais difíceis. Obrigado pela amizade de vocês, mesmo na ausência. Agradeço também aos amigos e professores Luiz Satoru, Alexandre Plastino e Lúcia Drummond por terem sidos meus primeiros incentivadores e mentores na vida acadêmica.

Aos amigos da Locksley (Sandro, Pinto, Luiz e Fabio) e da Eletrobrás (André, Claudia, Guilherme, Gustavo, Paulo Band, Rafael, Ricardo e Claudio Magalhães) pelo companheirismo e pela grande torcida para o sucesso do mestrado demonstrados desde o início do curso. Adiciono a essa lista os amigos Gante e Cristine.

Aos professores Jano e Blaschek pela oportunidade que me foi concedida no projeto SINGRA e pelos conhecimentos adquiridos dentro e fora da sala de aula. Sem dúvida, a experiência adquirida durante o tempo em que estive no projeto contribuiu significativamente para o meu amadurecimento profissional. Aos Comtes Gomes Lima, Frederico e Nelson, Cap. Louzada e Yura pelo apoio e companheirismo profissional e pessoal.

Aos novos colegas do LCP, Lauro, Raquel, André, Fábio, Carla, Lúcia, Leonardo, Edison, Thobias, Silvano, Rodrigo e Lobosco pela torcida e apoio nesta reta final, além do excelente ambiente de trabalho. Em particular, ao profº. Claudio Amorim pela oportunidade que me foi concedida em seu projeto, cuja importância para a continuação de minha pesquisa será fundamental.

A Patty, secretária da linha de Banco de Dados do PESC, por sua simpatia e disponibilidade em todas as ocasiões. A Claudia, Solange, Mercedes, Sueli e demais funcionários da secretaria do PESC pela eficiência e atenção com que atendem a todos.

Aos meus pais Elson e Idila, por seu esforço e dedicação demonstrados em todo o meu processo educacional, além do carinho complementar não menos importante. Essa conquista não é só minha, mas de vocês também. Obrigado pela torcida e pelo apoio!!!! Agradeço também ao meu irmão Fábio e aos meus avós Ítalo e Diva. Espero desfrutar mais da companhia de todos vocês, já que tenho estado um "pouco" ausente.

Ao apoio financeiro da CAPES, fundamental para manter minhas atividades de pesquisa.

Finalmente, agradeço a Kelly, por seu amor, carinho, compreensão, paciência, amizade e ombro amigo durante os momentos mais alegres e difíceis. Sou muito grato por sua força em me manter motivado, mesmo quando era você quem mais precisava de colo. Essa vitória também é sua e espero colher todos os frutos desta conquista ao seu lado. Fomos capazes de vencer o tempo e a distância. Agora, creio que somos capazes de vencer qualquer barreira. Agradeço também à minha segunda família, seu Eurípedes e Carla, que sempre me receberam com muito carinho em suas casas.

Índice do Texto

Lista de Figuras

Lista de Tabelas

Capítulo 1. Introdução

1.1 Motivações

Grande parte da pesquisa em processamento de consultas em sistemas de bancos de dados (SGBDs) tem sido baseada no desenvolvimento de novos algoritmos para processar o operador de junção, reconhecidamente o principal operador de avaliação para um processador de consulta (MISHRA e EICH, 1992, VALDURIEZ, 1987, IOANNIDIS e KANG, 1990, VALDURIEZ e GARDARIN, 1984, SHEKITA e CAREY, 1990). Isto se deve a sua complexidade e custo de processamento, assim como a sua freqüente utilização em consultas que expressam relacionamentos entre os dados armazenados.

Nos últimos anos, com os avanços adquiridos no campo da Engenharia de Software e das Linguagens de Programação, o domínio das aplicações de bancos de dados expandiu-se para novas classes de aplicações não convencionais, tais como Multimídia, Sistemas de Informações Geográficas (SIGs), Mineração de Dados e Projeto Assistido por Computador (CAD), não mais limitando-se às aplicações tradicionais tipicamente existentes nas organizações comerciais. Este fato impulsionou o surgimento e a utilização de Sistemas Gerenciadores de Bancos de Dados Orientados a Objetos (SGBDOOs) (CATTELL, 1994) e Relacionais Objeto (SGBDROs), ou simplesmente sistemas OO e sistemas RO. As principais vantagens oferecidas por esses SGBDs são a riqueza semântica proporcionada pelo modelo orientado a objetos (OO) e a possibilidade de utilização de um único modelo de dados durante todo o desenvolvimento da aplicação, desde as etapas de análise e projeto, até a programação, armazenamento (no caso de SGBDOOs) e recuperação dos dados armazenados.

Todas as dificuldades encontradas no processamento de consultas em sistemas relacionais continuam presentes no modelo OO. Entretanto algumas novas são inseridas neste contexto devido às propriedades intrínsecas deste modelo, como visto em DAYAL *et al.* (1993). Além disso, a diversidade de propostas de modelos de dados e de especificação de linguagens de consultas também vem contribuindo para que esta dificuldade seja maior, embora o padrão estabelecido pelo *Object Data Group Management*, ou ODMG (CATTELL *et al.*, 2000), venha minimizando este problema. Diferente do modelo relacional, o modelo OO permite que um objeto possa ter atributos

que são referências para outros objetos, enquanto relações são formadas apenas por atributos de tipos simples. Por causa disso, as linguagens de manipulação devem oferecer um mecanismo capaz de especificar consultas que contenham as chamadas *expressões de caminho*.

Expressões de caminho são um recurso simples e elegante para "navegar" pela estrutura dos objetos. As linguagens que apoiam a especificação de expressões de caminho fornecem dessa forma um mecanismo uniforme para a formulação de consultas que envolvem diferentes características do modelo OO, tais como relacionamentos de agregação, associação e herança.

Entretanto, o estudo do processamento eficiente de expressões de caminho não se restringe apenas ao contexto de SGBDOOs. Sistemas RO também beneficiam-se das técnicas de otimização para expressões de caminho, já que as tuplas das relações possuem referências (ponteiros) para tuplas de outras relações. Existem ainda outras propriedades do modelo OO que estão presentes em SGBDROs, tais como a definição de atributos multi-valorados. Alguns SGBDs comerciais permitem que um atributo possa ter como domínio uma outra relação definida no sistema.

Recentemente, uma nova área de aplicação para as técnicas de otimização e processamento de expressões de caminho vem ganhando popularidade. O processamento eficiente de consultas sobre documentos XML também pode se beneficiar dos algoritmos propostos para sistemas OO e RO, já que os objetos definidos dentro de um documento podem relacionar-se entre si. Se o armazenamento dos documentos XML for realizado por um SGBD (relacional ou não), as consultas realizadas serão processadas de acordo com as estratégias de otimização presentes no SGBD. Porém, se um SGBD não é utilizado, a presença de algoritmos que processem os relacionamentos entre os objetos de um documento passa a ser indispensável para a eficiência de sistemas gerenciadores de documentos XML, ou servidores XML (CERI *et al.*, 2000).

Em sistemas relacionais, uma consulta realizada sobre um conjunto de relações é processada utilizando-se o operador binário de junção. Basicamente, a escolha do melhor plano de execução pelo otimizador é restrita à ordem em que as junções são processadas. Normalmente, o otimizador de consultas dispõe de regras de transformação (heurísticas) para a escolha do melhor plano, já que esse é um problema NP completo. Em outras palavras, é computacionalmente intratável descobrir o melhor plano de

execução, segundo uma função de custo, já que são inúmeros os planos de execução que resolvem a mesma consulta, cada um com um custo de execução associado. Por isso é importante o estudo para construção de heurísticas que possam ser aproveitadas pelo otimizador de consultas.

Para os sistemas OO ou RO, as expressões de caminho podem ser resolvidas através de uma seqüência de junções, mas contam ainda com a possibilidade de algoritmos baseados no operador n-ário. Um algoritmo baseado no operador n-ário percorre em paralelo todas as referências entre os objetos relacionados, diferentemente dos algoritmos baseados em junção, que processam as coleções de dados duas a duas. GARDARIN *et al.* (1996) caracterizam o operador n-ário pela busca em profundidade, enquanto que a seqüência de operadores de junção como uma busca em largura sobre a expressão de caminho. Sendo assim, tornam-se ainda maiores as possibilidades de planos de execução que resolvem a expressão de caminho, tornando o problema ainda mais difícil.

O estudo do processamento eficiente de expressões de caminho na presença dos operadores binário e n-ário é importante não só para SGBDOOs e ROs, mas também em diferentes contextos. A presença do operador n-ário para um otimizador de consultas, embora aumente a complexidade do problema, cria uma alternativa viável para a construção de um processador de consultas eficiente e completo.

1.2 Objetivo

Esta dissertação tem por objetivo contribuir com o estudo da avaliação da eficiência do processamento de expressões de caminho. Outros trabalhos já abordaram o processamento de expressões de caminho, tanto no contexto seqüencial (GARDARIN *et al.*, 1996, SHEKITA e CAREY, 1990, BRAUMANDL *et al.*, 1998) como no contexto paralelo (TAVARES *et al.*, 2000, DEWITT *et al.*, 1993 e TANIAR, 1998). Porém muitos autores descartam algoritmos baseados no operador n-ário por considerá-los ineficientes. Quando não os descartam, apontam outras estratégias a serem utilizadas no processamento, normalmente baseadas em algoritmos de junção adaptados ao modelo OO para funcionarem com ponteiros, isto é, algoritmos baseados no operador binário.

O algoritmo clássico para o operador n-ário é o algoritmo *Naïve Pointer Chasing* (NP). Este algoritmo percorre a expressão de caminho através da navegação "ingênua" entre os objetos, ou seja, a navegação entre os objetos é realizada na ordem (aleatória)

em que os objetos aparecem nos relacionamentos. A principal vantagem deste algoritmo é o seu baixíssimo custo de CPU, já que ele não gera estruturas auxiliares para otimização de acesso aos dados. Porém, o seu maior problema é a alta exigência de memória para compensar o acesso aleatório ao disco durante a navegação dos caminhos.

No entanto, a quantidade de memória dedicada aos servidores de banco de dados vem deixando de ser um problema. A capacidade de processamento das máquinas vem crescendo exponencialmente e seus custos diminuindo. Por conseguinte, propomos nesta dissertação um algoritmo baseado no operador n-ário chamado "*Smart Naïve*" (SN). O algoritmo proposto é uma otimização do algoritmo clássico *Naïve Pointer Chasing* (NP), onde o principal objetivo é manter o baixo custo de CPU verificado no algoritmo NP e diminuir o acesso aleatório a disco, evitando o acesso a caminhos que se repetem durante o processamento da expressão de caminho.

O desempenho do SN é comparado experimentalmente ao algoritmo NP e ao algoritmo binário que obteve o melhor desempenho dentre todos os algoritmos binários existentes, o P(PM)*M, como mostrado em BRAUMANDL *et al.* (2000). Foram realizados experimentos onde o algoritmo SN alcançou um desempenho compatível, do ponto de vista do custo de E/S, quando comparado NP e mesmo ao P(PM)*M. Este fato já justifica a presença de algoritmos n-ários para otimizadores de consulta, contrariando muitos trabalhos que simplesmente descartam estes algoritmos (SHEKITA e CAREY, 1990, DEWITT *et al.*, 1993). Além disso, foram também analisados os custos de CPU desses algoritmos, baseados no modelo de custo proposto em BRAUMANDL *et al.* (2000). As análises mostram situações vantajosas para cada um dos algoritmos NP, SN e P(PM)*M.

Esta dissertação também apresenta uma análise abrangente dos aspectos relevantes para o processamento de expressões de caminho encontrados em trabalhos na literatura, tais como SHEKITA e CAREY (1990), DEWITT *et al.* (1993), BRAUMANDL *et al.* (2000), GARDARIN *et al.* (1996) e TANIAR (1998). Esses trabalhos apresentam limitações em suas análises, como por exemplo, a avaliação somente de algoritmos binários, expressões de caminho de comprimento dois, uma única direção de percurso do caminho, entre outras. Os aspectos relevantes que devem ser levados em consideração pelo otimizador de consultas no processamento de expressões de caminho são aqui abordados. Dessa forma, é possível obter heurísticas

que indiquem a melhor estratégia segundo esses aspectos, ou seja, inferir sob que circunstâncias a utilização de um algoritmo é mais vantajosa que a utilização dos outros.

Para a realização dos experimentos, foi utilizado o sistema de gerência de objetos GOA (MAURO *et al.,* 1997), um protótipo desenvolvido pelo grupo de pesquisa em banco de dados do Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ. As análises experimentais são realizadas sobre a base de dados especificada pelo *benchmark* OO7 (CAREY *et al.*, 1993). A escolha deste *benchmark* é justificada, pois ele vem sendo, nos últimos anos, a ferramenta padrão para a avaliação de desempenho de SGBDOOs, tanto comerciais como projetos acadêmicos (DEWITT *et al.*, 1996, GARDARIN *et al.*, 1996, ÖZSU *et al.*, 1998, SU *et al.*, 1999).

Para a geração das diferentes bases de objetos avaliadas experimentalmente foi desenvolvida uma aplicação chamada **OO7Gen**. Esta aplicação é responsável pela criação dos arquivos contendo os *scripts* que populam uma base de dados do OO7 no GOA. É possível ajustar, a partir de sua interface, parâmetros importantes da base, como por exemplo as cardinalidade das coleções, os graus de saída (*fan-out*) e compartilhamento (*share*) das coleções que se relacionam e a taxa de partição dos objetos nos relacionamentos. Depois que os arquivos são gerados pelo **OO7Gen**, eles são importados pelo servidor GOA sobre uma base totalmente vazia. Dessa forma, foi possível automatizar a geração de diferentes configurações de base, diminuindo o esforço na realização dos experimentos.

1.3 Organização do trabalho

O capítulo 2 apresenta uma caracterização de expressões de caminho, mostrando os diferentes aspectos de expressões de caminho que causam impacto no processamento de consultas. Além disso, são apresentadas as principais dimensões (direção e operador) que devem ser utilizadas para gerar uma estratégia de processamento de expressões de caminho. Cada estratégia apresentada é ilustrada com exemplos didáticos. É mostrada também a estratégia híbrida que, embora tenha sido abordada em GARDARIN *et al.* (1996), MATTOSO (1993) e MATTOSO e ZIMBRÃO (1994), não tem sido devidamente explorada. Na estratégia híbrida de processamento de expressões de caminho, as diferentes estratégias que podem ser geradas pela combinação das dimensões apresentadas são combinadas entre si para formarem novas estratégias.

Ainda neste capítulo, são revistos alguns trabalhos na literatura que fazem análises comparativas de algoritmos para expressões de caminho.

No capítulo 3 é apresentado o algoritmo proposto nesta dissertação, o algoritmo *Smart Naïve*. Além deste, são apresentados também os outros algoritmos que foram avaliados experimentalmente: o algoritmo n-ário *Naïve Pointer Chasing* e o algoritmos binários de junção $P(PM)^*M$ e junção por valor. São também apresentadas algumas heurísticas de processamento de expressões de caminho propostas por trabalhos encontrados na literatura e que são utilizadas neste trabalho.

A descrição do ambiente experimental é apresentada no capítulo 4. Neste capítulo é apresentada a arquitetura do GOA, assim como as modificações necessárias no seu código fonte para a incorporação dos algoritmos avaliados. Além disso, é apresentada a máquina utilizada para a realização dos experimentos, juntamente com a descrição do *benchmark* OO7.

Os cenários avaliados e os resultados obtidos na execução dos experimentos são mostrados no capítulo 5. Em cada cenário, foi discutida a importância do seu estudo no contexto do processamento de expressões de caminho. Neste capítulo também encontram-se as análises realizadas sobre os resultados obtidos e as heurísticas que puderam ser extraídas dos experimentos são apresentadas.

Finalmente, as conclusões obtidas a partir dos resultados experimentais e as contribuições deste trabalho são apresentadas no capítulo 6, bem como as propostas para trabalhos futuros.

Capítulo 2. Expressões de Caminho

Muitas são as dificuldades encontradas na otimização e no processamento de consultas em SGBDOOs. Embora muitas das dificuldades sejam as mesmas das encontradas em sistemas relacionais, algumas delas são intrínsecas do modelo orientado a objetos (OO) (MATTOSO *et al.*, 2001, KIM, 1995, DAYAL *et al.*, 1993).

A estrutura complexa dos objetos é um dos aspectos do modelo OO que mais dificultam a construção de um otimizador de consultas para SGBDOOs. Diferente do modelo relacional, o modelo OO permite que um objeto possa ter outros objetos como atributos, enquanto relações são formadas apenas por atributos de tipos simples. Por causa disso, as linguagens de consulta devem oferecer um mecanismo capaz de especificar consultas que contenham as chamadas *expressões de caminho*.

Expressões de caminho são um recurso simples e elegante para navegar pela estrutura dos objetos, pois representam um mecanismo uniforme para a formulação de consultas que envolvem diferentes características do modelo OO, incluindo relacionamentos de agregação e associação. Expressões de caminho também são conhecidas na literatura como cadeias de referências, travessias, predicados complexos ou junções implícitas.

O processamento otimizado de expressões de caminho é uma questão difícil e central em linguagens de consulta. Atualmente, diversas linguagens de consulta apoiam a especificação de expressões de caminho. Entre as linguagens que merecem destaque podemos citar a OQL (*Object Query Language*), estabelecida como padrão pelo ODMG (CATTELL *et al.*, 2000) para SGBDOOs e a SQL:1999 (EINSENBERG e MELTON, 1999), que é uma extensão da linguagem padrão SQL para SGBDROs.

Linguagens de consulta relacionais operam sobre tipos simples de dados contidos em relações homogêneas. O conjunto simples e bem definido dos operadores da álgebra relacional mantém o fechamento transitivo das operações. Entretanto, o mesmo pode não ser válido para uma álgebra de objetos (STRAUBE e ÖZSU, 1990). As coleções que formam os operandos podem conter objetos de tipos distintos, assim como os objetos da coleção resultante.

Mais recentemente, um novo contexto de aplicação para o processamento de expressões de caminho vem ganhando força. Trata-se do processamento de consultas

sobre documentos XML. São muitas as propostas de linguagens de consulta para a recuperação de dados armazenados no formato XML, tais como XQuery (BOAG *et al.*, 2001), estabelecida como padrão pelo consórcio W3C (http://www.w3c.org), Lorel (ABITEBOUL *et al.*, 1997), XML-QL, XQL, entre outras (DEUTSCH *et al.*, 1999, FEGARAS e ELMASRI, 2000). Em DEUTSCH *et al.* (1999) são apresentados os principais requisitos que uma linguagem de consulta deve oferecer e quais destes requisitos são suportados ou não pelas linguagens de consulta. Todas estas linguagens citadas fornecem meios para a especificação de expressões de caminho em suas sintaxes.

Muito se tem discutido sobre a utilização de SGBDOOs para o armazenamento de documentos XML (ABITEBOUL *et al.*, 1997), já que sistemas OO provêem a semântica natural para expressar consultas sobre uma coleção de objetos relacionados. Dessa forma, pode-se tirar proveito das propostas de algoritmos que se beneficiam das referências estabelecidas entre os objetos. Neste contexto, seria necessário traduzir a consulta especificada por uma linguagem de consulta XML para a linguagem de consulta nativa do sistema (OQL, por exemplo). Porém, como as técnicas de otimização e processamento de consultas para SGBDs relacionais já estão há bastante tempo consolidadas no mercado, alguns trabalhos (SHANMUGASUNDARAM *et al.*, 1999, MANOLESCU *et al.*, 2000) apontam como a tecnologia XML pode ser integrada a sistemas relacionais.

Portanto, fica claro que não se pode ignorar toda a pesquisa e tecnologia disponível em SGBDs diante desta nova área de aplicação (processamento de consultas sobre documentos XML). Mesmo com todo o desenvolvimento de técnicas para o armazenamento e processamento de consultas sobre dados semi-estruturados (ABITEBOUL, 1997), a comunidade de pesquisa em banco de dados ganhou novos desafios ao absorver esta recente tecnologia. Um deles é o processamento eficiente de expressões de caminho.

Entretanto, o processamento de expressões de caminho não representa um tópico de estudo tão novo assim. Os trabalhos para o processamento eficiente de uma consulta sobre um conjunto de coleções de dados relacionados vêm sendo realizados não só em função das propostas de linguagens de consulta e algoritmos para o processamento de consultas em SGBDOOs e SGBDROs, como serão mostrados em detalhes na seção 2.4, mas também em função das diversas propostas de algoritmos para o processamento de

junções relacionais sobre uma coleção de relações, onde o foco é determinar a melhor ordem de junção entre as relações (MISHRA e EICH, 1992, SHEKITA e TAN, 1993, LU *et al.*, 1991).

Existem na literatura algumas propostas de representação de expressões de caminho (GRAHAN, 1983, GARDARIN *et al.*, 1996, BERTINO, 1997). No entanto, as representações existentes não consideram os conceitos de **classe** e **coleção** como conceitos distintos, assim como não tornam explícito o atributo multi-valorado que está sendo utilizado para navegação, tornando a notação ambígua. Na próxima seção é apresentado o formalismo utilizado para representar expressões de caminho. Esta representação, que é utilizada no restante da dissertação, foi definida em MATTOSO *et al.* (2001) e suas vantagens em relação às representações existentes também são discutidas. Para esclarecer alguns conceitos desta nova representação, são fornecidos pequenos exemplos utilizando-se a linguagem de consultas OQL do padrão ODMG (CATTELL *et al.*, 2000).

2.1 Representação de Expressões de Caminho

Seja $\{C_1, C_2,..., C_\ell\}$ um conjunto de coleções, não necessariamente distintas entre si, cujas respectivas classes $\{T_1, T_2,..., T_\ell\}$ estão relacionadas através de atributos de referência A_i. Ou seja, em ordem crescente de i, A_i é um atributo dos objetos de C_i, cujos valores possíveis são um ou mais objetos da coleção C_{i+1}. Além disso, C'_i é a coleção de objetos da classe C_i apontados por objetos da coleção C_{i-1}, através do atributo A_{i-1}, onde $C'_i \subseteq C_i$ e $C'_1 = C_1$.

Uma expressão de caminho é definida por:

$\underline{P : X_1[(p_1)].A_1[-X_2][(p_2)].A_2[-X_3][(p_3)]\cdots.A_{\ell-1}[-X_\ell][(p_\ell)]}$, onde X_i é a variável, definida na consulta, que representa uma coleção de objetos do tipo T_i. Quando A_{i-1} é um atributo multi-valorado, X_i representa um cursor que percorre cada objeto de C_i que é referenciado por A_{i-1}. Neste caso, "$-X_i$" deve obrigatoriamente aparecer na expressão de caminho, caso contrário pode ser omitido. O comprimento de uma expressão de caminho P é definido pelo número de coleções, ℓ, em P. C_1 (ou C_r) é denominada coleção-raiz, e C_ℓ é denominada coleção-folha ou coleção-alvo. Ainda, p_i é um predicado aninhado opcional que seleciona objetos de C'_i. Se p_i é nulo, todos os objetos

de C'_i são qualificados para a expressão de caminho. Predicados aninhados são melhor descritos ainda nesta seção.

Esta representação de expressão de caminho fornece uma maneira uniforme de descrever qualquer caminho de navegação entre classes relacionadas no esquema. Entre os aspectos que merecem destaque nesta representação, podemos citar:

- definição precisa do atributo de referência (A_i) que é considerado entre cada par de classes do caminho de navegação, reduzindo ambigüidade;
- definição precisa de expressões de caminho multi-valoradas, já que para atributos do tipo coleção define-se um cursor ($-X_i$) para varrer a coleção referenciada;
- pode representar seleções sobre as classes no caminho de navegação através de predicados aninhados (p_i).

A seguir são mostrados alguns exemplos de consultas OQL expressas sobre o esquema do *benchmark* OO7 (CAREY *et al.*, 1993) e que foram extraídas de MATTOSO *et al.* (2001). A figura 2.1 mostra a definição em ODL - *Object Definition Language* (CATTELL *et al.*, 2000) das classes do *benchmark* envolvidas nos exemplos. A definição do esquema foi simplificada para melhor entendimento.

```
//-------------------------------------------------------------------------------------------------
// DesignObj é a raiz da hierarquia de classes para a maioria dos objetos OO7
//-------------------------------------------------------------------------------------------------
class DesignObj {
  attribute string type;
  attribute long buildDate;
};

class AtomicPart extends DesignObj
(extent AtomicParts)
{
  attribute int x;
  attribute int y;
  attribute string name;
  relationship set(Connection) to
       inverse Connection::from;
  relationship set(Connection) from
       inverse Connection::to;
  relationship CompositePart part_of
       inverse CompositePart::parts;
};
//-------------------------------------------------------------------------------------------------
// Instâncias de Connection são usadas para conectar 2 instâncias de  AtomicPart
//-------------------------------------------------------------------------------------------------
class Connection
(extent Connections)
{
   attribute string type;
   attribute int length;
  relationship AtomicPart from
       inverse AtomicPart::to;
  relationship AtomicPart to
       inverse AtomicPart::from;
};
//-------------------------------------------------------------------------------------------------
// Instâncias de CompositePart são compostas de instâncias de AtomicPart
//-------------------------------------------------------------------------------------------------
class CompositePart extends DesignObj
(extent CompositeParts)
{
    relationship Document document
        inverse Document::composite;
    relationship set(AtomicPart) parts
         inverse AtomicPart::part_of;
    relationship AtomicPart rootPart;
};
//-------------------------------------------------------------------------------------------------
// Instâncias de Document são usadas para descrever instâncias de CompositePart
//-------------------------------------------------------------------------------------------------
class Document
(extent Documents)
{
    attribute string;
    attribute long date;
    relationship CompositePart composite
         inverse CompositePart::document;
};
```

Figura 2.1: Definição do esquema do *benchmark* OO7 em ODL

Exemplos:

1. A consulta abaixo possui uma expressão de caminho de comprimento um.

```
select d
from d in Documents
where d.date > 10/01/1910
```

A expressão correspondente será:

```
P1: d(date > 10/01/1910)
```

2. Seja a seguinte consulta, escrita em OQL:

```
select c
from c in CompositeParts, a in c.parts
where a.x < 100000
```

A expressão de caminho contida nesta consulta possui comprimento dois, a coleção-raiz é `CompositeParts`, a coleção-folha é `AtomicParts` e a sua representação é a seguinte:

```
P2: c.parts-a(x < 100000)
```

3. A consulta seguinte contém duas expressões de caminho, devido aos diferentes relacionamentos da classe `CompositePart`:

```
select c
from c in CompositeParts, a in c.parts
where a.x < 900000 and c.document.date < 10/01/1940
```

As expressões correspondentes, ambas com comprimento dois, são:

```
P3: c.parts-a(x < 900000)
P4: c.document(date < 10/01/1940)
```

4. Observe agora a diferença entre a consulta anterior e a seguinte, que contém apenas uma expressão de caminho:

```
select a
from a in AtomicParts
where a.x < 900000 and a.part_of.document.date <
10/01/1940
```

Neste caso, temos apenas uma expressão com comprimento três, que é representada por:

```
P5: a(x < 900000).part_of.document(date < 10/01/1940)
```

A cada coleção C_i está associado um predicado p_i, também chamado de **predicado aninhado**. Cada predicado p_i é opcional e estando vazio qualificará todos os objetos de C_i. Os predicados aninhados podem ser formados por uma combinação lógica de expressões matemáticas sobre atributos das classes pertencentes à expressão de caminho, onde cada expressão é da forma `<atributo> <operador> <valor>` (por exemplo, "`a(x < 900000)`" ou "`c.document(date < 10/01/1940)`") e `<valor>` é qualquer expressão de mesmo tipo que `<atributo>` (podendo conter inclusive outros atributos).

Quando todos os atributos A_i $(1 \leq i < \ell)$ de uma expressão de caminho são atributos de referência monovalorados, temos uma expressão de caminho **monovalorada**. Caso exista pelo menos um atributo do tipo coleção na expressão de caminho, esta é dita **multivalorada**. Consideramos que os atributos multivalorados possuem domínio restrito a apenas uma classe.

Exemplos:

5. A consulta abaixo possui uma expressão de caminho monovalorada de comprimento dois.

```
select c
from c in CompositeParts
where c.rootPart.buildDate < 10/01/1940
```

Representada por:

```
P9: c.rootPart(date < 10/01/1940)
```

6. A expressão de caminho P2 do exemplo 2 é multivalorada.

```
P2: c.parts-a(x < 100000)
```

2.2 Operadores Algébricos

Diversos trabalhos (CATTELL, 1994, ÖZSU e BLAKELEY, 1995, STRAUBE e ÖZSU, 1990) se propõem a definir um modelo para consultas, envolvendo cálculo, álgebra e linguagem para SGBDOOs. A linguagem de consulta OQL, definida pelo padrão ODMG, expressa consultas que têm como base uma álgebra de objetos. Embora

o padrão ODMG não defina uma álgebra, as diversas propostas de álgebra (SHAW e ZDONIK, 1990, FEGARAS, 1998, CLUET e DELOBEL, 1992, GEPPERT *et al.*, 1990, STRAUBE e ÖZSU, 1990) possuem um subconjunto de operadores semanticamente similares. Nesta seção são mostrados alguns operadores relevantes do ponto de vista do processamento de expressões de caminho. Estes operadores estão definidos na maioria das álgebras, embora a definição precisa dos operandos e do resultado para cada operador varie de álgebra para álgebra, assim como a própria nomenclatura atribuída a cada operador.

Por motivos de organização, os operadores mostrados a seguir foram classificados em dois grupos: operadores clássicos, que têm correspondência com a álgebra relacional e que são provenientes desta; e operadores específicos do modelo OO, direcionados a atender a riqueza semântica do modelo.

2.2.1 Operadores Clássicos

- **Seleção (SELECTION(P, σ))** - Este operador recebe uma coleção de objetos **P** e um predicado de seleção σ como parâmetros. O predicado de seleção é uma expressão semelhante ao predicado aninhado definido na seção anterior, ou seja, trata-se da combinação lógica de expressões matemáticas do tipo `<atributo> <operador> <valor>`. O resultado da aplicação deste operador é a coleção **Q**, tal que para todo objeto p ∈ Q, p também pertence a coleção P (p∈ P) e satisfaz o predicado de seleção σ.

- **Projeção (PROJECTION(P, Π))** - Este operador recebe uma coleção de objetos **P** e um subconjunto Π dos atributos e/ou métodos da classe que qualifica os objetos em P. O resultado da aplicação deste operador é a coleção **Q**, tal que para todo objeto p ∈ Q, p também pertence a coleção P (p∈ P) e apresenta somente os atributos e/ou métodos definidos em Π.

- **Interseção (INTERSECTION(P_1, P_2))** - A operação de interseção entre coleções tem a semântica análoga à sua definição tradicional sobre conjuntos, isto é, retorna como resultado uma coleção **Q** de objetos, tal que para cada objeto p ∈ Q, p também pertence às coleções de objetos P_1 e P_2.

- **União (UNION(P_1, P_2))** - Também análoga à sua definição tradicional sobre conjuntos, a operação de união entre coleções retorna como resultado uma

coleção **Q** de objetos, tal que para todo objeto p ∈ Q, ou q pertence a coleção P_1, ou q pertence a coleção P_2 ou pertence a ambas as coleções.

- **Junção (JOIN (P_1, P_2, σ))** – Este operador é análogo ao operador junção definido no modelo relacional. Para as duas coleções de entrada P_1 e P_2, o operador junção produz a coleção **Q** de objetos do tipo $<p_1, p_2>$, onde $p_1 \in P_1$ e $p_2 \in P_2$, tal que p_1 e p_2 satisfazem ao predicado σ de junção. Ou seja, o resultado da aplicação deste operador é uma coleção de objetos formados pela concatenação dos objetos de P_1 e P_2.

2.2.2 Operadores específicos

- **Desaninhar (UNNEST(P, att))** - O operador desaninhar atua sobre uma coleção **P** de objetos e produz uma coleção **Q** de saída. Para cada objeto p ∈ P, o atributo **att** é avaliado. Se o atributo é multivalorado (do tipo coleção), cada elemento **x** da coleção formada pelo atributo é extraído e adicionado à coleção Q de saída, concatenado ao objeto p. Seja por exemplo, uma coleção ***P*** composta por dois objetos p_1 e p_2. A classe definida no esquema para estes objetos é ***C*** e possui um atributo ***c*** do tipo coleção. Se, para os objetos p_1 e p_2, os elementos da coleção formada pelo atributo ***c*** são $\{i, j\}$ e $\{m, n\}$ respectivamente, então o resultado da operação **UNNEST(*P*, *c*)** é a coleção ***Q*** contendo os elementos $<p_1, i>$, $<p_1, j>$, $<p_2, m>$ e $<p_2, n>$. Caso o atributo ***c*** seja monovalorado, seu valor é extraído de forma análoga.

- **Aninhar (NEST(P, att))** – Este operador realiza a operação inversa do operador UNNEST, isto é, agrupa os objetos da coleção **P** segundo o atributo **att**. A coleção **Q** de saída contém todos os objetos de P. Porém, todos os objetos da coleção P são avaliados quanto aos atributos definidos na classe do esquema que representa a coleção P. Se duas ou mais instâncias de P diferenciam-se entre si apenas pelos valores do atributo **att**, somente umas destas instâncias é adicionada à coleção de saída Q. Além disso, o atributo **att** da instância adicionada a Q transforma-se num atributo do tipo coleção (multivalorado) contendo todos os valores distintos do atributo **att** para as diferentes instâncias originais p da coleção P. No exemplo mostrado para o operador UNNEST, se a coleção de saída do UNNEST for aplicada ao operador NEST sobre o atributo ***c***,

a saída produzida pelo operador NEST será a mesma coleção P que representa a coleção de entrada do operador UNNEST.

- **Percurso (MAP(P_1, P_2, ..., P_n, a_1, a_2, ..., a_{n-1}))** – Este operador recebe **n** coleções de objetos como parâmetros de entrada. Além das coleções de entrada, este operador também recebe uma lista dos **n-1** atributos (a_1, a_2, ..., a_{n-1}) definidos nas classes que representam as coleções P_1, P_2, ..., P_{n-1}, respectivamente. Cada atributo a_i ($1 \leq i < n$, onde n é o número de coleções envolvidas) possui como domínio uma instância (ou conjunto de instâncias) presente(s) na coleção P_{i+1}. A coleção de saída resultante da aplicação deste operador sobre os parâmetros de entrada contém todas as instâncias de P_n que estão relacionadas a algum caminho formado pela navegação dos objetos de P_1 através dos atributos a_1, a_2, ..., a_{n-1}.

Segundo ÖZSU e BLAKELEY (1995), uma fase importante no processamento de consultas, em especial para o processamento de expressões de caminho, é a fase de otimização algébrica. A otimização algébrica é feita pela aplicação das regras de transformação algébrica sobre a expressão de consulta já traduzida do cálculo para a álgebra. Entretanto, estas regras não são gerais, já que elas são muito dependentes da álgebra específica, ou seja, as regras de transformação para uma álgebra não são necessariamente válidas em outra álgebra, até porque os operadores podem ser diferentes. Por causa desta diversidade de propostas algébricas, os trabalhos em otimização de consultas em SGBDOOs são bastante variados, já que não existem regras e heurísticas padrões como existem para o modelo relacional.

Os operadores algébricos definidos nesta seção são os mais representativos para uma álgebra de objetos. Toda expressão de caminho, como aquelas especificadas na notação apresentada na seção anterior, pode ser reescrita na forma algébrica, utilizando-se os operadores aqui apresentados. Entretanto, o maior interesse desta dissertação está na avaliação do desempenho dos operadores **junção** e **percurso**. Uma expressão de caminho formada por **n** coleções pode ser processada ou pela aplicação de um operador percurso, ou pela sucessiva aplicação de **n-1** junções. Pode-se ainda combinar os dois operadores para processarem partes isoladas da mesma expressão de caminho. Posteriormente, um dos operadores é aplicado para combinar os resultados intermediários.

Uma discussão mais detalhada destes operadores é apresentada na próxima seção. Basicamente, a escolha do operador a ser utilizado no processamento de expressões de caminho é um dos aspectos que deve ser tratado pelo otimizador de consultas. As diferentes implementações encontradas na literatura para cada operador são comentadas na seção 2.4. Devido à diversidade de propostas algébricas e, conseqüentemente, à diversidade de nomenclaturas para os operadores junção e percurso, estes operadores são tratados no restante da dissertação como operadores binário e n-ário, respectivamente.

2.3 Estratégias de processamento

Duas dimensões são importantes no processamento de expressões de caminho, a direção e o operador algébrico. A direção é referente à ordem em que as coleções são percorridas durante o processamento da consulta. As duas possíveis direções são descendente (*forward*) e ascendente (*reverse*). A direção descendente indica que a expressão de caminho deve ser percorrida na ordem em que foi especificada na consulta, ou seja, da classe raiz para a classe folha, sendo T_1 a classe raiz e T_n a classe folha, onde **n** é o comprimento da expressão de caminho. A direção ascendente indica que a expressão deve ser percorrida na ordem inversa (da classe folha para a classe raiz).

Vale a pena ressaltar que a formulação de uma expressão de caminho pode sugerir uma direção de execução que não necessariamente é a mais eficiente para o processador de consulta. Às vezes pode ser mais eficiente processar a expressão de caminho na direção inversa à qual foi definida.

A outra dimensão importante no processamento de expressões de caminho diz respeito aos operadores algébricos utilizados na avaliação dos relacionamentos. Existem basicamente dois operadores importantes no processamento dos relacionamentos entre as coleções da expressões de caminho: o operador n-ário e o binário. Segundo GARDARIN *et al.* (1996), o operador n-ário caracteriza-se por uma varredura em profundidade sobre os objetos das coleções envolvidas na expressão de caminho, ao passo que o operador binário varre em largura.

O operador n-ário processa a expressão de caminho ao percorrer todo o caminho formado pelos objetos das **n** coleções envolvidas na expressão de caminho. Dado um objeto da coleção de partida (coleção-raiz ou coleção-folha, dependendo da direção a

ser avaliada), todo o seu caminho é avaliado antes do próximo objeto dentro da mesma coleção ser avaliado. O algoritmo mais conhecido que implementa o operador n-ário é o "*Naïve Pointer Chasing*" (BRAUMANDL *et al.*, 1998).

Já o operador binário processa a expressão de caminho ao aplicar o operador de junção binária, semelhante ao operador de junção relacional, sobre um par de coleções. Isto significa que uma expressão de caminho de comprimento **n** é processada através de **n-1** junções sobre pares de coleções. O resultado intermediário serve como parâmetro de entrada juntamente com a próxima coleção da expressão para uma nova junção, e assim sucessivamente até a última coleção da expressão de caminho (coleção-raiz ou folha, dependendo da direção). São muitos os algoritmos que implementam este operador, desde os algoritmos relacionais adaptados para funcionarem com valores de IDOs (baseados em valor ou *value based join*), até algoritmos que tiram proveito dos ponteiros entre os atributos de referência para navegarem entre os objetos (baseados em ponteiros ou *pointer based join*).

A combinação destas dimensões (operador/direção) formam diferentes estratégias de processamento de expressões de caminho. Para um melhor entendimento destas estratégias, a figura 2.2 a seguir mostra a representação em árvore de cada uma das combinações na seguinte consulta OQL especificada sobre o esquema do *benchmark* OO7 (CAREY *et al.*, 1993).

```
select a
from a in AtomicParts
where a.part_of.document.date > 10/01/1990
```

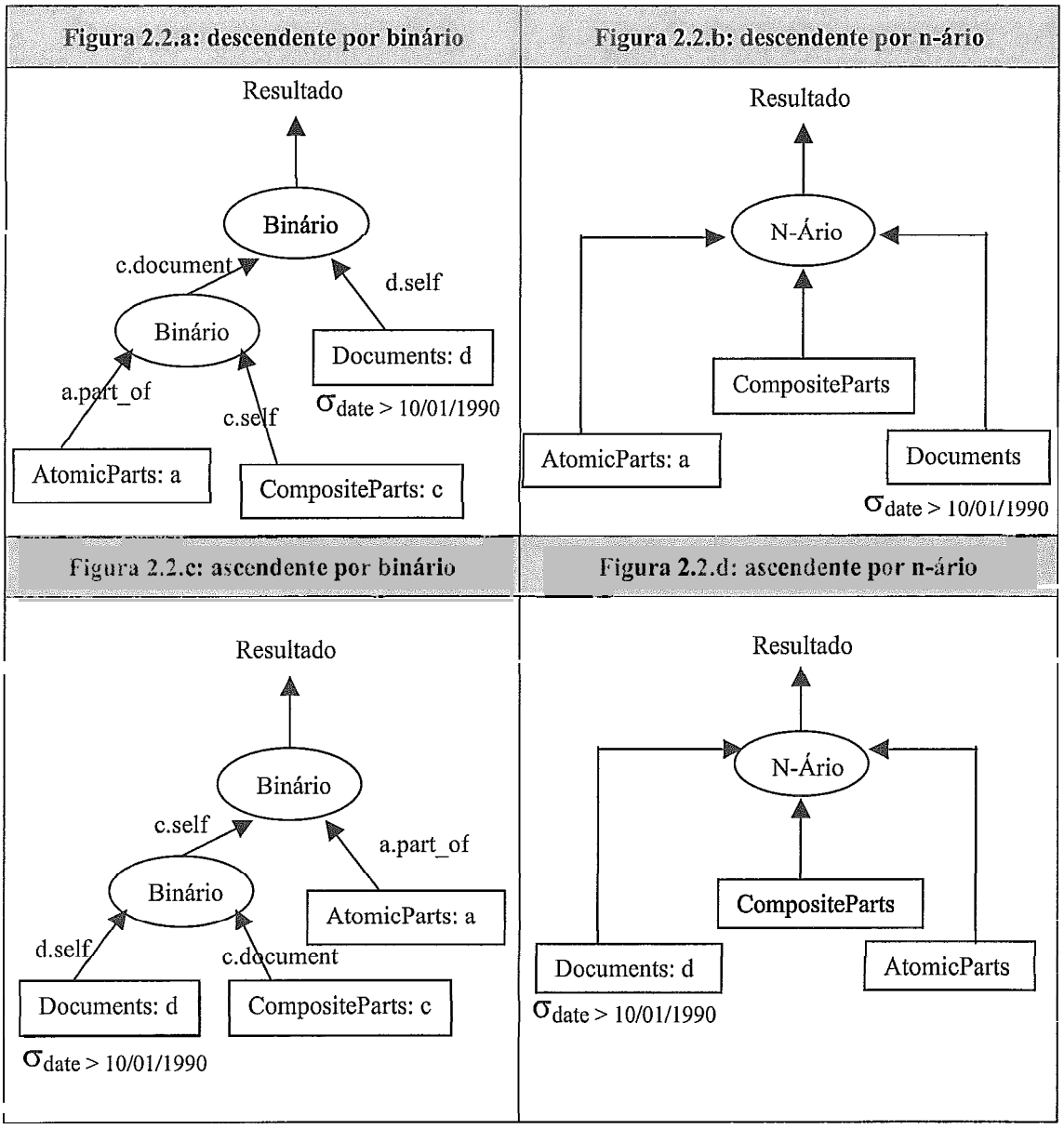

Figura 2.2: Representação em árvore das estratégias de processamento de expressões de caminho

Na notação das árvores de consulta da figura 2.2, as elipses e os retângulos representam os nós das árvores, enquanto as setas representam as arestas. As elipses representam operadores algébricos, enquanto os retângulos representam os operandos, coleções de objetos. Uma seta incidindo sobre um operador representa uma coleção que serve como operando. Uma seta saindo de um operador representa a coleção de objetos produzida pela aplicação do operador sobre os parâmetros de entrada (setas incidindo). No caso dos nós folhas, as setas indicam que coleção representada pelo nó folha (retângulo) é aplicada como parâmetro para o nó pai. A seta saindo do nó raiz representa

a coleção resultado. Uma consulta é processada dos nós folhas para os nós intermediários, até que o resultado do nó raiz seja gerado (resultado da consulta).

Todos os nós folhas (retângulos) são rotulados com o nome das coleções envolvidas na expressão de caminho. É importante observar que, somente em alguns nós folhas, o nome da coleção aparece seguido de dois pontos(":") e uma variável. Esta variável representa um cursor que percorre a coleção, ou seja, todos os objetos da coleção são percorridos e cada instância é materializada na variável. Note que somente as árvores do operador binário (figuras 2.2.a e 2.2.c) possuem o cursor definido para todos os nós folhas. Isto se justifica pela necessidade do operador binário em varrer as duas coleções operando (somente no caso dos exemplos mostrados, pois o algoritmo que implementa o operador binário nos exemplos é baseado em valor), enquanto o operador n-ário só vai percorrer os objetos de uma coleção que são apontados pela coleção anterior na expressão de caminho.

Outro ponto que deve ser esclarecido nesta notação é a presença do predicado de seleção ($\sigma_{date > 10/01/1990}$). Nas árvores do operador binário, somente os objetos que satisfazem o predicado de seleção são materializados na variável que representa o cursor, ou seja, o predicado é aplicado sobre todos os objetos da coleção, mas somente os que satisfazem o predicado são materializados. Entretanto, nas árvores do operador n-ário, o predicado de seleção é aplicado a cada objeto que é acessado pelo atributo de referência da expressão de caminho. Portanto, nem todos os objetos da coleção têm o predicado associado aplicado.

As árvores do operador binário (figuras 2.2.a e 2.2.c) possuem arestas rotuladas. Cada rótulo representa o atributo da classe usado no processamento da junção binária. O atributo `self` indica que o IDO do objeto é usado como atributo de comparação na junção binária. Portanto, as árvores do operador binário mostradas na figura 2.2 representam árvores de execução para os algoritmos de junção binária baseados em valor (*value based join*).

As figuras 2.2.b e 2.2.d representam as árvores do operador n-ário no sentido descendente e ascendente, respectivamente. A direção em que a expressão de caminho é percorrida fica evidenciada pela ordem em que as coleções aparecem no operador. Nesta notação em árvore, o operador n-ário manipula as coleções da esquerda para a direita. Na consulta da figura 2.2, a expressão de caminho foi especificada de

`AtomicParts` para `Documents`. Por causa disso, na figura 2.2.b, a coleção mais à esquerda é a coleção `AtomicParts`. Entretanto, na direção ascendente a coleção mais à esquerda é a coleção `Documents`.

No operador n-ário, a navegação pelos objetos que formam o caminho especificado na consulta é realizada através dos atributos de referência (relacionamentos) definidos no esquema. Para a direção descendente, os atributos de referência são os que aparecem especificados na consulta (`part_of` e `document`), enquanto na direção ascendente, os atributos de referência são os respectivos atributos inversos (`parts` e `composite`), quando existirem. Como o padrão ODMG não determina que atributos inversos sejam obrigatórios, esta estratégia (n-ário/ascendente) pode ter sua aplicabilidade reduzida. No caso do OO7, todos os relacionamentos especificados na consulta da figura 2.2 possuem atributos inversos.

2.3.1 Estratégia de Processamento Híbrida

Até então foram apresentadas diversas estratégias para o processamento de expressões de caminho. Os exemplos adotados (figura 2.2) escolhem uma combinação direção/operador e a aplicam em toda a extensão da expressão de caminho. Entretanto, uma expressão de caminho pode ser decomposta em duas ou mais subexpressões, definidas de modo recursivo. No processamento de uma consulta contendo uma expressão de caminho, diferentes estratégias podem ser aplicadas a cada subexpressão da mesma. Isto implica em um acréscimo considerável de complexidade no processo de otimização deste tipo de consulta.

Existe ainda uma estratégia pouco explorada na literatura (GARDARIN *et al.*, 1996, MATTOSO, 1993, MATTOSO e ZIMBRÃO, 1994), mas que não pode ser descartada para uma análise mais abrangente. Trata-se da estratégia híbrida. Nesta estratégia, parte da expressão de caminho segue uma destas combinações e a outra parte da expressão segue outra combinação, diferente da primeira. O ponto da expressão de caminho que separa estas duas partes é chamado de ponto de quebra e os resultados parciais de cada parte da expressão são reunidos para formar o resultado (via operador de junção, por exemplo).

Pode-se ainda dividir a expressão de caminho em mais de duas subexpressões, ou seja, dois ou mais pontos de quebra podem estar presentes na expressão de caminho. A presença da estratégia híbrida para um otimizador de consulta aumenta

consideravelmente a complexidade de avaliação de uma expressão de caminho, já que o espaço de soluções de planos de execução cresce exponencialmente.

Para facilitar o entendimento da estratégia híbrida, seja o seguinte exemplo sobre a consulta do *benchmark* OO7:

```
select c from c in Connections
where c.from.part_of.document.date > 10/01/1990
```

De acordo com a representação proposta na seção 2.1, esta consulta pode ser representada como mostra a expressão a seguir:

```
c.from.part_of.document(date < 10/01/1990)
```

Para esta expressão monovalorada de tamanho quatro, podemos definir o ponto de quebra como mostra a expressão:

```
c.from.|part_of.document(date < 10/01/1990)
```

ponto de quebra

Suponha, por exemplo, que a subexpressão esquerda seja processada pela estratégia ascendente/n-ário e a subexpressão direita pela estratégia ascendente/binário. A representação em árvore desta estratégia de execução é mostrada na figura 2.3.

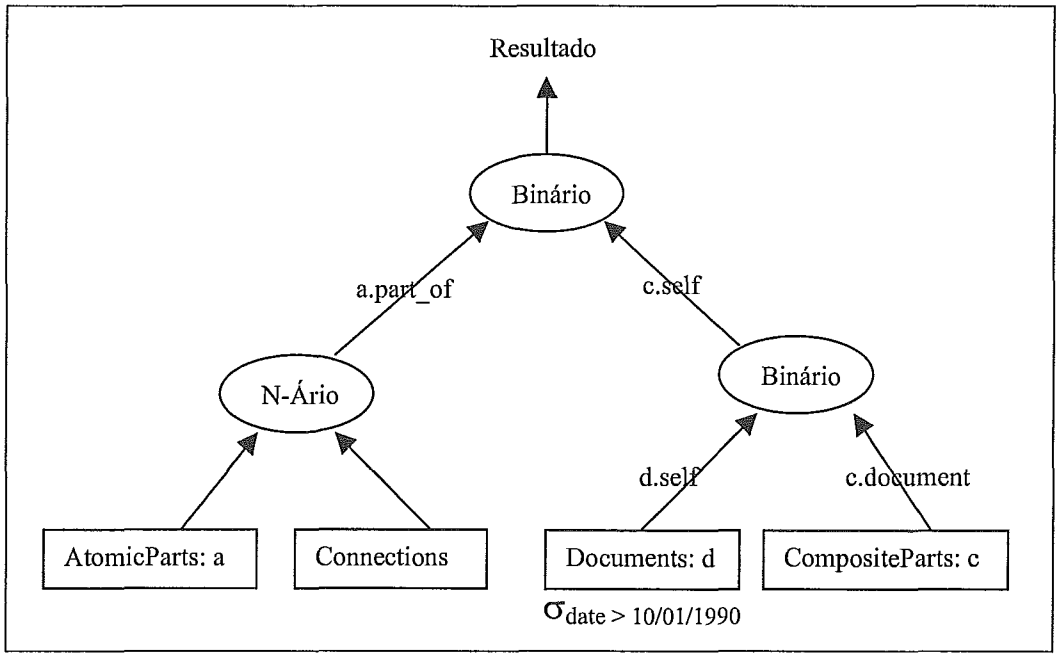

Figura 2.3 – Representação em árvore da estratégia híbrida

Repare que na árvore da figura 2.3 existem dois operadores binários e um operador n-ário. O operador n-ário é raiz da subárvore esquerda e o operador binário de menor altura na árvore é a raiz da subárvore direita. O operador n-ário produz uma coleção de objetos da classe `Connections` que estejam relacionados a algum objeto da classe `AtomicPart`. O operador binário de menor altura gera uma coleção de objetos da classe `CompositePart` que estejam relacionados a objetos da classe `Document` que satisfaçam ao predicado `date > 10/01/1990`. Os resultados das duas subárvores são combinados pelo operador de junção de maior altura na árvore, que finalmente processa as duas coleções intermediárias geradas e produz o resultado. Repare que o operador n-ário está sendo aplicado a somente duas coleções, diferente do exemplo da figura 2.2. Neste caso especial, o operador n-ário comporta-se de forma análoga ao operador de junção binária na versão baseada em ponteiros (*pointer based join*).

Neste exemplo, os resultados das duas subárvores foram combinados por um operador binário. Porém, esta idéia não é uma regra geral. Os resultados intermediários poderiam ter sido combinados por um operador n-ário.

2.4 Um Panorama dos Algoritmos Existentes para Expressões de Caminho

Um dos problemas em analisar e comparar os trabalhos existentes sobre o processamento de expressões de caminho reside na ausência de uma álgebra de objetos universalmente aceita. Assim, as propostas de algoritmos para os operadores que podem representar as expressões de caminho variam muito não só na terminologia, mas em técnicas e quanto ao modelo de representação de objetos. Na verdade, muitos deles não tratam especificamente de expressões de caminho, mas sim de novos algoritmos de junção (operador binário) adaptados para funcionar com referências (ponteiros). Além disso, poucos são os trabalhos que avaliam a variação das duas dimensões (operador/direção) simultaneamente e trazem resultados experimentais. A maioria dos trabalhos descarta a análise de um algoritmo que implemente o operador n-ário e quando o fazem, normalmente o aplicam em condições desfavoráveis.

Como muitos dos algoritmos propostos são adaptações dos algoritmos tradicionais de junção relacional para funcionarem com ponteiros, vale a pena fazer uma breve explanação sobre como é a adaptação destes algoritmos. Os algoritmos laços aninhados

(*nested loops*), ordenação-fusão (*sort-merge*) e *hybrid hash* são os algoritmos mais representativos do operador de junção relacional.

A grande diferença entre os algoritmos - para expressões de caminho com somente duas coleções - laços aninhados, ordenação-fusão e *hybrid hash*, nas suas versões baseadas em ponteiros, reside no fato de que o primeiro é um algoritmo "ingênuo", ou seja, ele não faz nenhuma tentativa de evitar que páginas da coleção-folha sejam recarregadas, já que para cada objeto da "coleção-raiz" é buscada a página do objeto relacionado pelo atributo de referência. Sejam dois objetos o_i e o_k da coleção C_1, por exemplo, que apontam para um mesmo objeto da coleção C_2, ou mesmo para objetos diferentes, mas que residem na mesma página. Se a ordem na qual estes dois objetos são percorridos não é favorável, ou seja, depois que o caminho de o_i foi avaliado (página do objeto relacionado em C_2 foi carregada), muitos outros caminhos foram avaliados antes da avaliação do caminho de o_k, é bastante provável que ao "navegar" (junção baseada em ponteiro) pelo relacionamento de o_k, a página do objeto relacionado da coleção C_2 já tenha sido descartada da memória. Por causa disso, a página seria novamente carregada para a memória, aumentando assim o custo de E/S do algoritmo.

Já os algoritmos ordenação-fusão e *hybrid hash* tentam evitar a recarga de páginas da "coleção-folha" ao agruparem os objetos da "coleção-raiz" segundo as páginas que os mesmos acessam da "coleção-folha". A diferença é que o algoritmo ordenação-fusão realiza o agrupamento ao ordenar os objetos (em memória ou disco) da coleção-raiz pelo atributo de referência que representa o relacionamento entre as coleções raiz e folha. Para o exemplo comentado no parágrafo anterior, ao ordenar os objetos da coleção C_1 pelo atributo de referência, os objetos o_i e o_k são mantidos "próximos" um ao outro. Ao carregar a página do objeto da coleção C_2 apontado por o_i, evita-se a recarga desta página no percurso do relacionamento de o_k, já que o caminho deste objeto seria avaliado logo em seguida.

No algoritmo *hybrid hash*, o agrupamento é realizado ao aplicar uma função *hash* no atributo de referência entre as coleções. Quando a representação dos IDOs é física, ou seja, o IDO contém explicitamente a página cujo objeto está armazenado, a aplicação da função *hash* agrupa os objetos da coleção-raiz pela proximidade do armazenamento dos objetos da coleção-folha, ou seja, os objetos da coleção-raiz que apontam para objetos da coleção-folha fisicamente próximos são mantidos agrupados. Se a representação dos IDOs é lógica, será necessária ainda uma tradução para os endereços

físicos e em seguida um novo agrupamento (por ordenação ou *hash*) pela representação física dos relacionamentos.

Um dos trabalhos clássicos sobre o operador binário foi publicado por SHEKITA e CAREY (1990). Neste trabalho, os algoritmos laços aninhados, ordenação-fusão e *hybrid hash* foram implementados em duas versões: junções baseadas no valor do atributo de referência (*value based join*) e junções baseadas nos ponteiros dos atributos de referência (*pointer based join*). Todas as versões dos algoritmos tiveram seus desempenhos avaliados por simulação. Apenas expressões de caminho de comprimento dois e monovaloradas foram avaliadas. Além disso, todos os algoritmos foram testados somente na direção descendente.

Apesar do foco dos autores ter sido a implementação de algoritmos para o operador binário, a implementação do algoritmo laços aninhados baseados em ponteiros é conceitualmente idêntica à implementação do algoritmo "*Naïve Pointer Chasing*" que implementa o operador n-ário para o caso particular de uma expressão de caminho de comprimento dois. Portanto, embora o operador n-ário não estivesse "oficialmente" implementado, os resultados para o algoritmo de junção laços aninhados são válidos para representar o desempenho do operador n-ário.

Como resultado do seu trabalho, SHEKITA e CAREY (1990) apresentaram heurísticas na escolha do melhor algoritmo segundo o modelo de custo proposto. Basicamente, os algoritmos baseados em ponteiros tiveram desempenho superior aos baseados em valor nos cenários analisados. Entre os algoritmos baseados em ponteiros, o algoritmo *hybrid-hash* sobressai-se em relação aos outros. O algoritmo laços aninhados se mostrou ruim em quase todos os testes, exceto quando a seletividade aplicada à coleção-raiz era tão alta que a quantidade de objetos desta coleção que participavam da expressão de caminho era muito menor que a cardinalidade da coleção. Por este motivo, poucos acessos a disco foram realizados no percurso do caminho formado pelas duas coleções e, portanto, nivelou o desempenho dos algoritmos com relação ao custo de E/S. Com relação ao custo de CPU, o algoritmo laços aninhados levou vantagem devido a sua simplicidade.

O trabalho de SHEKITA e CAREY (1990) foi continuado por DEWITT *et al.* (1993) ao avaliarem também por simulação o comportamento dos mesmos três algoritmos de junção (laços aninhados, ordenação-fusão e *hybrid-hash*) em um ambiente paralelo. Entretanto, os algoritmos foram modificados para funcionarem com

atributos de referência do tipo coleção, ou seja, expressões de caminho multi-valoradas. Além disso, foram propostos três novos algoritmos para o operador binário. Dois deles são variações do *hybrid hash* e o terceiro é chamado de *probe-child*. Estes algoritmos têm a vantagem de não exigirem que os objetos apontados pelos relacionamentos da expressão de caminho estejam armazenados na extensão de alguma classe. Em outras palavras, os objetos presentes nos relacionamentos (atributos de referência A_i) não precisam estar armazenados em alguma coleção explicitamente construída no esquema. Foi apresentado um algoritmo que calcula uma extensão de classe para o atributo se a mesma não existir. Assim como em SHEKITA e CAREY (1990), na análise dos algoritmos, o operador n-ário foi descartado e somente expressões de caminho descendentes de comprimento dois foram avaliadas.

No trabalho de GARDARIN *et al.* (1996), o desempenho dos operadores n-ário e binário no processamento de expressões de caminho foi verificado experimentalmente. Os resultados experimentais foram validados junto a um modelo de custo também proposto no trabalho. Vários cenários foram testados, entre eles a variação da seletividade aplicada às coleções, o agrupamento dos objetos relacionados na expressão de caminho, a variação da memória disponível e do tamanho da expressão de caminho.

A partir dos resultados coletados, os autores propuseram heurísticas para a escolha do operador n-ário. Basicamente, o operador n-ário deve ser escolhido quando há memória suficiente para armazenar um caminho inteiro da expressão de caminho e se duas ou mais coleções envolvidas na expressão de caminho estão agrupadas fisicamente pelos atributos de referência. Os autores apresentaram ainda o desempenho da estratégia híbrida. Estes resultados são importantes, já que este foi o único trabalho encontrado na literatura que avalia esta estratégia. Por outro lado, os autores não avaliaram o desempenho dos algoritmos de junção baseados em valores. Além disso, a expressão de caminho só foi percorrida na direção ascendente pelos algoritmos baseados no operador binário.

Em BRAUMANDL *et al.* (1998) foi proposto um algoritmo baseado no operador binário para o processamento de expressões caminho. Esse algoritmo é denominado $P(PM)^*M$. Suas virtudes são a independência da implementação do IDO (lógico ou físico) e a habilidade em processar tanto expressões de caminho mono-valoradas como multi-valoradas. O desempenho do algoritmo proposto foi avaliado junto a outros algoritmos de junção e ao algoritmo n-ário *Naïve Pointer Chasing*. Os resultados

experimentais evidenciaram a eficiência do algoritmo P(PM)*M frente aos demais algoritmos implementados (operadores binário e n-ário). Entretanto, os resultados se limitam a duas coleções (expressões de caminho de comprimento dois), o que limita a análise do operador n-ário.

Os autores apresentaram também um estudo mais detalhado do algoritmo P(PM)*M em BRAUMANDL *et al.* (2000). Neste trabalho, um número maior de cenários foram testados e foi apresentado um modelo de custo cujos resultados puderam ser comparados e validados junto aos resultados experimentais. Variação de parâmetros importantes como o tamanho da memória, a cardinalidade das coleções, o número de objetos participando no relacionamento multi-valorado (grau de saída) e a seletividade nas coleções foram verificadas, assim como o impacto dos diferentes tipos de IDOs (físicos e lógicos) foram testados. Porém, os algoritmos foram testados somente na direção descendente. No próximo capítulo desta dissertação, maiores detalhes deste algoritmo (P(PM)*M) serão discutidos, já que esse é um dos algoritmos avaliados experimentalmente nesta dissertação.

Em TANIAR (1998) foram propostas heurísticas que auxiliassem o otimizador de consultas na escolha da direção a ser percorrida para o processamento de expressões de caminho (descendente ou ascendente). Entretanto, neste trabalho somente algoritmos baseados no operador n-ário são considerados ("*Naïve Pointer Chasing*" descendente e ascendente). O problema desta abordagem é que somente expressões de caminho de comprimento dois foram avaliadas, exatamente o caso onde os algoritmos de junção (operador binário) são considerados os melhores.

Basicamente, as heurísticas geradas pelo autor mostram que a direção descendente deve ser escolhida quando existe um predicado de seleção na coleção-raiz, enquanto a direção ascendente deve ser escolhida caso contrário. Apesar das cardinalidades e dos fatores de seletividade terem sido variados, seus valores não são levados em conta, pois mesmo que exista seletividade na coleção raiz, o autor não considera se a seletividade é alta ou baixa. Outro aspecto limitante do trabalho é a premissa de que as coleções cabem em memória principal. Porém, este aspecto é também interessante, pois a maioria dos algoritmos propostos na literatura são restritivos quanto à utilização deste recurso. Como a memória é um recurso computacional cada vez mais abundante nos sistemas atuais, o autor parte desta premissa na avaliação dos resultados.

O trabalho de TAVARES (1999) avaliou experimentalmente duas estratégias para o processamento seqüencial e paralelo de expressões de caminho. A primeira é baseada na estratégia descendente pelo operador n-ário, enquanto a segunda é baseada em junção (algoritmo *hybrid hash* baseado em valor) no sentido ascendente. Os resultados permitiram mostrar em TAVARES *et al.* (2000) o potencial de ambas as estratégias e tornou claro o potencial paralelo da estratégia baseada em junção. Ficou evidenciado também que a estratégia pelo operador n-ário é bastante eficiente se a coleção raiz da expressão de caminho possui cardinalidade bem menor que as cardinalidades das outras coleções envolvidas. Por outro lado, o operador n-ário só foi avaliado na direção descendente, enquanto o operador binário na ascendente. As consultas também correspondiam àquelas especificadas no *benchmark* OO7 (CAREY *et al.*, 1993).

Um modelo de custo para estimar o custo de cada uma das estratégias resultantes da combinação da direção e do operador foi proposto em RUBERG (2001). Neste modelo foram contemplados alguns parâmetros importantes no processamento de expressões de caminho, tais como a seletividade da expressão de caminho, direção do processamento (ascendente ou descendente), participação parcial dos objetos nos relacionamentos entre as classes e o grau de compartilhamento entre objetos de coleções relacionadas. Alguns resultados obtidos por simulação foram validados junto aos resultados experimentais coletados em TAVARES (1999). A partir dos resultados obtidos foi possível gerar heurísticas que indicassem a utilização de uma estratégia em detrimento de outra. Outro aspecto importante deste modelo é o efeito da distribuição na base de objetos, ou seja, as fórmulas que computam os custos dos algoritmos consideram o efeito de um projeto de distribuição da base (ÖZSU e VALDURIEZ, 1999).

As tabelas 2.1 e 2.2 comparam os trabalhos citados nesta seção a partir de cenários que consideramos importantes para o processamento eficiente de expressões de caminho. A tabela 2.1 classifica os trabalhos indicando que propriedades da expressão de caminho foram levadas em consideração pelos seus autores e a tabela 2.2 classifica os trabalhos segundo as dimensões (direção e operador) exploradas nas análises de seus autores.

Tabela 2.1: Avaliação das principais características dos algoritmos para expressões de caminho na literatura

	Shekita & Carey (1990)	*DeWitt et al. (1993)*	*Gardarin et al. (1996)*	*Taniar (1998)*	*Braumandl et al. (2000)*	*Tavares et al. (2000)*
IDOs	*Físico*	*Físico*	*Físico*	-	*Ambos*	*Físico*
Obrigatoriedade de extensões de classe	*Sim*	*Não*	*Sim*	*Sim*	*Sim*	*Sim*
Analisa operador Binário	*Sim*	*Sim*	*Sim*	*Não*	*Sim*	*Sim*
Analisa operador N-ário	*Não*	*Não*	*Sim*	*Sim*	*Sim*	*Sim*
Avalia expressões de caminho de comprimento > 2	*Não*	*Não*	*Sim*	*Não*	*Não*	*Sim*
Variação dos fatores de seletividade	*Sim*	*Não*	*Sim*	*Não*	*Sim*	*Sim*
Apresenta resultados experimentais	*Não*	*Não*	*Sim*	*Sim*	*Sim*	*Sim*
Considera atributos multivalorados	*Não*	*Sim*	*Sim*	*Sim*	*Sim*	*Sim*

Tabela 2.2: Classificação dos algoritmos para expressões de caminho

	Operador N-ário	**Operador Binário**	
		Baseada em Ponteiros	**Baseada em Valores**
Descendente	Gardarin *et al.* (1996), Taniar (1998), Tavares *et al.*(2000) Braumandl *et al.* (2000)	Shekita & Carey (1990), DeWitt *et al.* (1993), Gardarin *et al.* (1996) Braumandl *et al.* (2000)	Shekita & Carey (1990)
Ascendente	Taniar (1998)	Gardarin *et al.* (1996)	DeWitt et al. (1993) Tavares et al.(2000)

Vale a pena destacar alguns aspectos de implementação dos trabalhos mostrados na tabela 2.1. Em especial, o trabalho de DEWITT *et al.* (1993) apresenta um algoritmo que cria uma ou mais extensões de classe para os objetos presentes nos relacionamentos, se as mesmas não existirem. Por isso, os algoritmos para o processamento de junção propostos no mesmo trabalho não exigem que os objetos estejam fisicamente armazenados em coleções criadas no sistema. Esta é uma restrição presente em todos os demais trabalhos estudados. Outro aspecto importante é a avaliação do impacto da representação dos IDOs no desempenho dos algoritmos. O estudo realizado em BRAUMANDL *et al.* (2000) foi o único a considerar os diferentes tipos de implementação de IDOs (lógicos e físicos) e as diferentes técnicas de implementação do IDO lógico (tabela *hash*, árvore B e tabela de mapeamento direto). O algoritmo $P(PM)^*M$ proposto no mesmo trabalho foi projetado para ser independente da representação do IDO. Um dos cenários experimentais avaliou o impacto das diferentes

implementações do IDO. Pôde ser comprovado que a penalidade paga pelo uso de IDOs lógicos é pequena comparada ao uso de IDOs físicos.

Como pode ser observado pela tabela 2.1, a maioria dos autores analisa expressões de caminho de tamanho dois (com exceção de GARDARIN *et al.*, 1996 e TAVARES *et al.*, 2000), fato que acaba por inibir o potencial do operador n-ário. Por isso, o operador n-ário é quase sempre descartado pelos autores em suas heurísticas. Devido aos resultados experimentais publicados em TAVARES *et al.* (2000), verificou-se o potencial do operador n-ário sob certas condições. Além disso, simulações realizadas no modelo de custo proposto em RUBERG (2001) também apontaram cenários onde a eficiência do operador n-ário é clara.

Por estes motivos, consideramos importante que algoritmos baseados no operador n-ário estejam disponíveis para um otimizador de consultas. Embora grande parte dos trabalhos em processamento de consultas seja dedicada ao desenvolvimento de algoritmos de junção, como mostram as tabelas 2.1 e 2.2, não se pode descartar a utilização de algoritmos n-ários como uma alternativa válida para o otimizador. É claro que este operador não é eficiente em todos os cenários possíveis, principalmente o algoritmo "*Naïve Pointer Chasing*", que é o único algoritmo representativo deste operador.

É para suprir esta carência que propomos nesta dissertação um algoritmo baseado no operador n-ário. Este algoritmo, chamado de "*Smart Naïve*", é apresentado em detalhes no próximo capítulo. Além do algoritmo, as principais heurísticas extraídas dos trabalhos contidos nas tabelas 2.1 e 2.2 são apresentadas na seção 3.1. São também comentados os algoritmos de junção P(PM)*M e *hybrid hash* por valor que serão avaliados experimentalmente nesta dissertação.

Procurou-se realizar nesta seção uma análise bastante abrangente dos diversos algoritmos e estratégias disponíveis para o processamento de expressões de caminho. Note que nenhum dos trabalhos mostrados na tabela 2.2 se encaixa em todas as lacunas. Este trabalho pode, pela sua abrangência, ser inserido em todas elas, já que foram avaliados experimentalmente, não só as duas direções no processamento de expressões caminho, mas também o desempenho de algoritmos baseados no operador n-ário e binário. Para o operador binário, foram avaliados ainda os algoritmos baseados em valores e em ponteiros. Portanto, se pintarmos cada entrada da tabela 2.2 para

representar as dimensões que foram contempladas nesta dissertação, teremos a tabela 2.2 totalmente pintada, como mostra a tabela 2.3.

Tabela 2.3: Dimensões contempladas

	Operador N-ário	Operador Binário	
		Baseada em Ponteiros	Baseada em Valores
Descendente	Gardarin *et al.* (1996), Taniar (1998), Tavares *et al.*(2000) Braumandl *et al.* (2000)	Shekita & Carey (1990), DeWitt *et al.* (1993), Gardarin *et al.* (1996) Braumandl *et al.* (2000)	Shekita & Carey (1990)
Ascendente	Taniar (1998)	Gardarin *et al.* (1996)	DeWitt et al. (1993) Tavares et al.(2000)

Capítulo 3. Proposta do Algoritmo "*Smart Naïve*"

A motivação principal desta dissertação está na busca da eficiência ao processar expressões de caminho. Assim como um comando SQL no modelo relacional pode gerar um plano de execução ótimo contendo diversos operadores algébricos (SELECTION, PROJECTION, JOIN, entre outros), o mesmo pode ocorrer numa expressão de caminho. A grande diferença no modelo OO é que os relacionamentos são expressos, em OQL, por operadores de navegação envolvendo **n** coleções. Já no modelo relacional, os relacionamentos são expressos em SQL através do operador binário de junção.

Desta forma, um otimizador de consultas do modelo relacional se preocupa com duas escolhas básicas para resolver os relacionamentos. A primeira diz respeito à ordem em que os pares de relações serão escolhidos para cada junção. A segunda escolha concerne ao algoritmo de junção a ser utilizado em cada par de relações.

Analogamente, no modelo OO o otimizador também precisa fazer essas duas escolhas. Para a ordem, consideramos a direção ascendente, descendente ou a estratégia híbrida. Já para os algoritmos, além da junção baseada em ponteiros, o modelo OO acrescenta mais uma possibilidade de escolha que são os algoritmos associados ao operador n-ário de navegação. Numa expressão de caminho, o otimizador tanto pode optar por uma seqüência de operadores binários (algoritmos de junção), quanto em usar um operador n-ário (algoritmo tipo "*naïve*").

Além da escolha entre **n-1** junções versus uma navegação total por algum algoritmo n-ário, o otimizador precisa ter a sua disposição diversos algoritmos baseados em ponteiros para resolver os relacionamentos, além dos clássicos algoritmos baseados em valor de chaves estrangeiras. Nesse sentido, o surgimento do modelo OO trouxe várias propostas de novos algoritmos baseados em ponteiros para navegações em expressões de caminho, apresentados no capítulo 2. Entretanto, a maioria destas propostas se concentrou em algoritmos para o operador binário, também chamado de "*functional joins*" em BRAUMANDL *et al.* (2000). O algoritmo representativo do operador n-ário, o algoritmo "*naïve*" ou "*Naïve Pointer Chasing*" (NP), tornou-se a única proposta apresentada. Por se tratar de um algoritmo "ingênuo", a maioria dos trabalhos o descartava ou evidenciava sua ineficiência, conforme observado da seção 2.4 do capítulo 2.

O grande problema desses trabalhos que analisaram o algoritmo NP foi negligenciar sua principal característica, isto é, o fato de ser um algoritmo para o operador n-ário. Nos diversos trabalhos que mencionam o NP como o pior algoritmo (SHEKITA e CAREY, 1990, DEWITT *et al.*, 1993, BRAUMANDL *et al.*, 1998), a comparação entre os algoritmos é feita usando-se apenas duas "coleções" como operandos. Ao se fixar o **n** (comprimento da expressão de caminho) em dois, o algoritmo n-ário NP se equipara ao algoritmo de junção por ponteiro. Por mais que os autores digam que a junção é facilmente aplicada em seqüência, existe uma sobrecarga nos algoritmos binários que é a geração dos resultados intermediários a serem usados na junção subsequente. O único trabalho encontrado que compara um algoritmo n-ário com **n-1** seqüências de binários foi GARDARIN *et al.* (1996), cujos resultados revelaram situações onde o algoritmo NP tem melhor desempenho que as seqüências de junções.

Sendo assim, nosso grupo de pesquisa vem realizando diversos trabalhos experimentais e de simulação (TAVARES, 1999, TAVARES *et al.*, 2000, RUBERG, 2001) analisando o comportamento do operador n-ário. Foram evidenciadas as situações de superioridade do NP apontadas por GARDARIN *et al.* (1996), mas também foram identificadas outras situações entre os relacionamentos que favoreciam o NP.

Apresentamos na seção 3.1 uma análise das diversas conclusões e heurísticas propostas em trabalhos que avaliam o algoritmo NP (GARDARIN *et al.*, 1996, TAVARES *et al.*, 2000, RUBERG, 2001). A partir dessa avaliação, identificamos que o potencial do operador n-ário poderia ser ampliado caso o algoritmo NP sofresse algumas otimizações. Da mesma forma em que algoritmos otimizados para o operador binário vêm sendo propostos (BRAUMANDL *et al.*, 2000), apresentamos uma proposta de otimização para o algoritmo NP denominada "*smart naïve*" (SN). Embora o nome seja antagônico (ingênuo esperto), optamos por manter o "*naïve*" para reforçar a presença do operador n-ário.

Em BRAUMANDL *et al.* (2000) foi apresentado um estudo detalhado e minucioso da proposta do algoritmo P(PM)*M. Neste trabalho os autores mostram, através de resultados experimentais e simulados por uma função de custo proposta, que o P(PM)*M é o algoritmo de junção mais eficiente dentre os melhores algoritmos de junção existentes. Sendo assim, para analisarmos a eficiência da nossa proposta, comparamos a implementação do SN com o tradicional n-ário NP e o binário do estado da arte, o P(PM)*M.

Na seção 3.2, apresentamos o algoritmo binário P(PM)*M, na seção 3.3 o tradicional NP e na seção 3.4 a proposta desta dissertação, o algoritmo SN. Os três algoritmos foram implementados e avaliados em ambiente experimental descrito em detalhes no capítulo 4.

3.1 Heurísticas sobre Algoritmos N-Ários e Binários

Conforme visto na seção 2.4 do capítulo 2, verificou-se que a maioria dos trabalhos para o processamento de expressões de caminho baseia-se na implementação de algoritmos de junção relacional adaptados para funcionarem com ponteiros. Muitos autores simplesmente descartam a utilização do algoritmo NP por considerá-lo ineficiente. Entretanto, como analisado em GARDARIN *et al.* (1996), este algoritmo apresenta algumas vantagens importantes. Entre elas podemos citar o seu baixo custo de CPU, já que é um algoritmo extremamente simples, e a ausência completa de resultados intermediários, ao contrário dos algoritmos de junção quando o comprimento da expressão de caminho é maior que dois.

Porém, a crítica mais comum feita sobre o algoritmo NP é o seu alto custo de E/S provocado pela recarga de páginas, principalmente quando o grau de compartilhamento entre duas coleções (*share*) é alto (RUBERG, 2001). Por causa deste argumento, muitos trabalhos não avaliam a execução deste algoritmo (SHEKITA e CAREY, 1990, DEWITT *et al.*, 1993), ou quando o fazem, o analisam exatamente nas suas condições desfavoráveis, como por exemplo expressões de caminho curtas (tamanho dois) ou pouca quantidade de memória disponível (BRAUMANDL *et al.*, 2000). Estes dois parâmetros inibem o potencial do algoritmo NP.

GARDARIN *et al.* (1996) propuseram heurísticas que justificassem a escolha do algoritmo NP por um otimizador de consulta. Estas heurísticas são baseadas nos experimentos realizados e validados junto ao modelo de custo proposto no mesmo trabalho. Basicamente, o algoritmo NP deveria ser escolhido quando:

- há memória suficiente para armazenar um caminho inteiro da expressão de caminho, ou seja, pelo menos uma página de cada coleção envolvida na expressão de caminho.

- duas ou mais coleções envolvidas na expressão de caminho estão agrupadas por referência.

Em RUBERG (2001) também foi proposto um modelo de custo de execução de consultas. A partir deste modelo, foram realizadas simulações de execução de consultas do *benchmark* OO7 e para cada consulta foram avaliadas as diferentes combinações de dimensões de processamento de expressões de caminho. O principal diferencial deste modelo de custo para os demais modelos encontrados na literatura é a incorporação de fragmentos de bases de dados distribuídas e do agrupamento nos custos de execução das estratégias. Além disso, a participação parcial dos objetos nos relacionamentos entre as coleções é contemplada nos cálculos da seletividade da expressão de caminho, dos graus de saída (*fan-out*) e compartilhamento de objetos (*share*). Este aspecto é comum em muitas aplicações reais. Outra vantagem deste modelo é a estimativa de custo em ambas as direções (ascendente e descendente).

As heurísticas extraídas dos resultados da simulação de RUBERG (2001) são, além das duas já anunciadas em GARDARIN *et al.* (1996), mostradas a seguir:

- escolha do algoritmo NP quando há participação parcial de uma ou mais coleções nos relacionamentos do caminho.

- na presença de relacionamentos inversos, a estratégia ascendente (NP ou junção por ponteiros) deve ser escolhida quando há alta seletividade na coleção-folha da expressão de caminho. Caso contrário, isto é, se não há relacionamentos inversos, algoritmos tradicionais de junção por valor podem ser usados

- se o grau de compartilhamento é alto, algoritmos tradicionais de junção por valor são os mais indicados.

Em TAVARES (1999), foram avaliados experimentalmente o desempenho das estratégias descendente e ascendente, com o objetivo de avaliar o potencial de paralelismo no processamento de expressões de caminho. Para a estratégia descendente foi utilizado o NP, enquanto que para a estratégia ascendente foi utilizada a junção por valor (*hybrid hash*).

Os resultados obtidos em TAVARES (1999) foram analisados em TAVARES *et al.* (2000) e geradas algumas heurísticas. Consultas com a cardinalidade da coleção-folha menor que a da coleção-raiz e envolvendo expressões de caminho mono-valoradas apresentam o perfil ideal para a execução baseada em junção. Neste caso, a execução baseada em junção, além de apresentar um tempo de execução menor em todas as configurações, apresentou um ganho maior com a adição de mais processadores ao

sistema, ou seja, os tempos de execução da estratégia baseada em junção caíram de forma mais rápida com a adição dos processadores.

Já no caso de consultas onde a cardinalidade da coleção-folha é maior que a da coleção-raiz, a escolha da melhor estratégia de execução é influenciada pelo fator de seletividade dos predicados envolvidos. Para baixos fatores de seletividade, embora a execução baseada em junção apresente tempo de execução bem superior ao tempo da execução n-ária (NP) na versão seqüencial, com o paralelismo estes tempos se aproximam, especialmente na configuração máxima de 12 processadores, tornando a alternativa baseada em junção tão válida quanto a baseada no operador n-ário. Para altos fatores de seletividade, a execução baseada em referência (NP) apresenta sempre tempos de execução inferiores, o que a torna a alternativa mais atraente em todas as configurações.

Portanto, embora cada trabalho contribua com heurísticas que possam ser aplicadas dentro do contexto avaliado, fica claro que o algoritmo NP apresenta pontos positivos e negativos. Basicamente, a utilização do NP é descartada quando um objeto da coleção C_i é apontado por muitos objetos da coleção C_{i-1}, ou seja, o grau de compartilhamento entre duas coleções é alto. Isto se deve ao fato da provável possibilidade de uma mesma página ser carregada várias vezes para a memória principal, em decorrência do percurso dos diferentes caminhos que apontam para este mesmo objeto.

Além disso, ambientes com pouca quantidade de memória disponível também limitam a aplicabilidade do NP. Muitos dos algoritmos na literatura, principalmente os mais antigos, foram propostos quando a memória era um recurso computacional bem mais caro do que é hoje. A restrição de memória está cada vez menos presente nos ambientes computacionais existentes, já que o poder computacional das máquinas existentes vem crescendo e seus custos diminuindo. Atualmente, mesmo para uma máquina doméstica, a quantidade de memória RAM disponível dificilmente é inferior a 64Mbytes. Mesmo para as operações típicas de um SGBD que envolvam relações operando de gigabytes, os algoritmos não necessitam que as relações caibam integralmente em memória. Portanto, com um bom algoritmo de *cache* de sistema operacional e do próprio SGBD (AILAMAKI *et al.*, 2001, AILAMAKI *et al.*, 1999, BONCZ *et al.*, 1999), a memória é usada eficientemente sem necessidade de espera por

E/S, já que o *cache* se encarrega de manter as páginas necessárias antecipadamente armazenadas num *buffer* em memória.

Em resumo, a principal vantagem da utilização do algoritmo NP é o seu baixo custo de CPU quando comparado ao custo de junção. Sua utilização deve ser encorajada quando há participação parcial dos objetos nos relacionamentos do caminho, já que por este motivo nem todos os objetos serão navegados e, consequentemente, menor será o tempo para realização de E/S. Outro aspecto que deve encorajar a utilização do NP é a presença de agrupamento por referência entre as coleções da expressão de caminho, diminuindo também o tempo de navegação entre os objetos, já que todos os objetos de um caminho são carregados na carga da página de um objeto da coleção-raiz. Finalmente, um baixo fator de seletividade na coleção-raiz ou coleção-folha já serve como indício para determinar a direção em que o algoritmo NP será aplicado, já que por causa da alta seletividade, grande parte dos objetos da coleção (raiz ou folha) será descartada e nem todos os caminhos serão percorridos.

3.2 Algoritmo de Junção P(PM)*M

O algoritmo utilizado nesta dissertação para implementar o operador binário foi o proposto em BRAUMANDL *et al.* (1998). Este algoritmo é similar aos algoritmos do tipo "materialização" (*flatten*), que são baseados no operador algébrico **desaninhar** (*unnest*) (ÖZSU e BLAKELEY, 1995 e FEGARAS, 1998). O operador desaninhar foi comentado no capítulo 2 (seção 2.2).

A idéia do operador desaninhar é "normalizar" os objetos pertencentes a um atributo do tipo coleção presente em um relacionamento. Por exemplo, se um objeto `CompositePart` possui no seu atributo `parts` dez referências para instâncias da classe `AtomicPart` e este objeto é aplicado ao operador desaninhar, o operador produz uma coleção de saída com dez objetos. Cada objeto é a concatenação do objeto `CompositePart` com o IDO do objeto `AtomicPart` relacionado. Portanto, o objeto `CompositePart` é replicado para cada objeto relacionado no seu atributo.

Os algoritmos tradicionais de junção baseados em valores de chaves estrangeiras aplicam o operador desaninhar em uma das coleções, se esta possui um atributo do tipo coleção, e realiza uma junção por valor entre a coleção desaninhada e a coleção relacionada. Suponha, por exemplo, duas coleções R e S, onde R possui um atributo de referência para objetos em S. Para o algoritmo de junção por *hash* baseado em valor, é

aplicada uma junção por valor segundo a função *h* entre a coleção R e a coleção S, utilizando como atributos para a junção o IDO do atributo de relacionamento e o IDO do objeto S. A figura 3.1 mostra os passos do algoritmo.

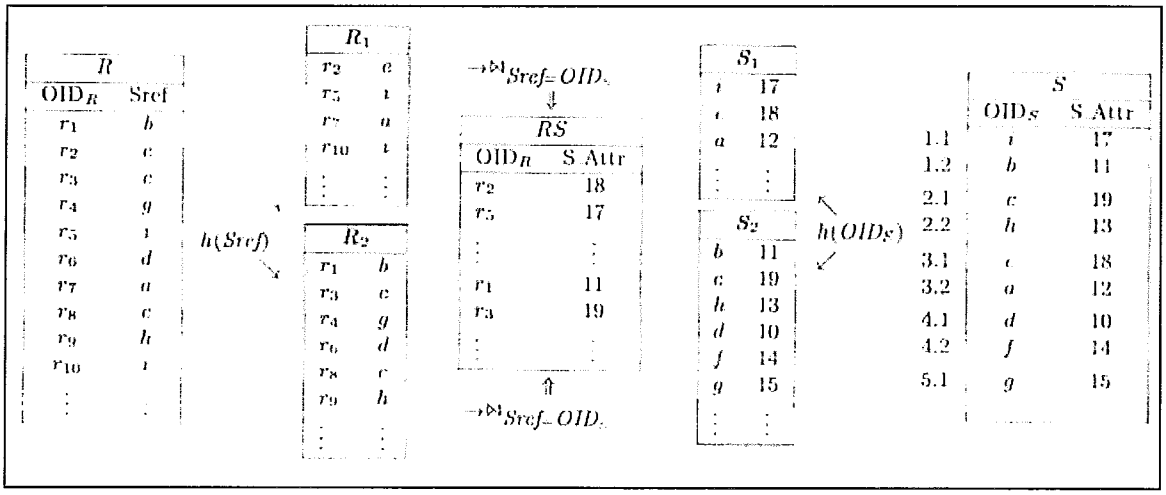

Figura 3.1 – Algoritmo de junção por valor *hybrid hash* (BRAUMANDL *et al.*, 2000)

Note que pela figura 3.1, ambas as coleções R e S são fragmentadas segundo a aplicação de uma função *hash*. As partições R_i e S_i são agrupadas em memória e produzem o resultado da junção. Se a coleção R possuísse um atributo do tipo coleção para os objetos em S, o primeiro passo do algoritmo mostrado na figura 3.1 seria desaninhar R. A figura 3.2 mostra um pseudo-código do algoritmo de junção por *hash*. baseada em valor.

```
// Particiona R em k partições
Para cada objeto r de R faça
    Adicione-o ao buffer da partição de R segundo a função h(r)
Fim-Para
// Particiona S em k partições
Para cada objeto s de S faça
    Adicione-o ao buffer da partição de S segundo a função h(s)
Fim-Para
Para i = 1 a k faça
   Leia Ri
   Para todo objeto r dentro de Ri faça
       Insira-o numa tabela hash h2(r)
   Fim-Para
   Leia Si
   Para todo objeto s dentro de Si faça
      Insira-o numa tabela hash h2(s)
   Fim-Para
   Gerar resultado < r , s > da tabela hash h2 para os objetos r e s que colidiram
Fim-Para
```

Figura 3.2 – Pseudo-código do algoritmo de junção por valor *hybrid hash*

O algoritmo P(PM)*M é um algoritmo semelhante ao da figura 3.2. Um diferencial deste algoritmo para os demais algoritmos de junção encontrados na literatura é que ele é independente do tipo de IDO (lógico ou físico). O funcionamento deste algoritmo pode ser melhor compreendido pela figura 3.3.

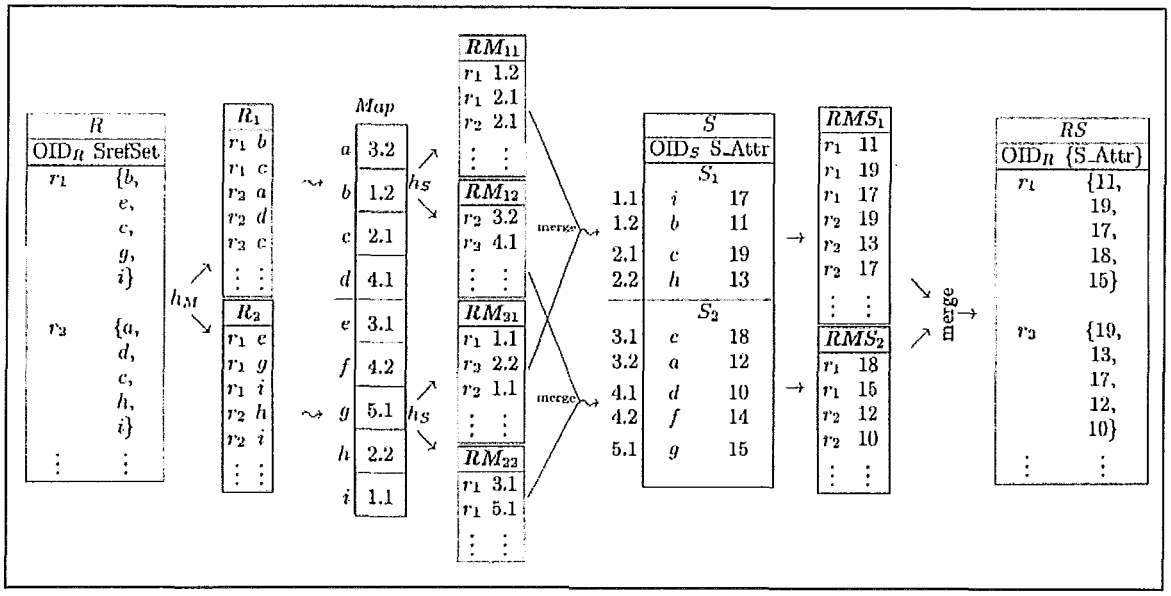

Figura 3.3 – Algoritmo binário P(PM)*M (BRAUMANDL *et al.*, 2000)

Assumindo as mesmas coleções R e S da figura 3.1, mas supondo que agora R possui um atributo do tipo coleção, R é primeiramente desaninhada. Depois disso, a coleção resultante é fragmentada em N fragmentos (N=2 fragmentos na figura 3.2) ao aplicar-se uma função *hash* h_M. Esta mesma função é aplicada a uma tabela de mapeamento direto (MAP) de endereços lógicos em endereços físicos. O algoritmo não considera o custo (CPU e E/S) de fragmentar esta tabela segundo esta função h_M, ou seja, a tabela já estaria previamente carregada e particionada em N fragmentos memória, ou pelo menos o fragmento desejado. Tendo a coleção de entrada R fragmentada (fragmentos R_1 e R_2) e a tabela MAP em memória, podemos converter, para cada fragmento de R, os IDOs lógicos do atributo do relacionamento de R em IDOs físicos.

Terminada esta etapa, a coleção R desaninhada está fragmentada em N fragmentos e os IDOs lógicos dos relacionamentos já foram convertidos para IDOs físicos. O número de partições é o mesmo do número de partições da tabela de mapeamento direto MAP, ou seja, N (partições RM_1, RM_2, ..., RM_N). O próximo passo é aplicar uma outra função *hash* h_S que vai agrupar os objetos de R segundo uma proximidade física dos IDOs de relacionamento. Para cada fragmento de RM, é aplicada a função h_S sobre o atributo de relacionamento que já foi desaninhado e convertido em endereço físico. São gerados K fragmentos para cada fragmento RM de R. Para o fragmento RM_1, por exemplo, são gerados K fragmentos (RM_{11}, RM_{12}, ..., RM_{1K}). No caso do exemplo da figura 3.3, K=2. Os objetos de R que apontam para páginas de objetos fisicamente próximas e que estejam em fragmentos diferentes são agrupados num único fragmento (o fragmento RM_{11} é agrupado ao fragmento RM_{21}, o fragmento RM_{12} com o RM_{22} e assim sucessivamente).

Finalmente, é realizada uma junção baseada em ponteiros entre cada fragmento agrupado e a partição correspondente de S, segundo a função h_S. Os fragmentos resultantes (RMS_1, RMS_2, ..., RMS_N) são reagrupados para formarem o resultado final da consulta. A figura 3.4 mostra o pseudo-código deste algoritmo.

```
Desaninha R
// Particiona R em N partições
Para cada objeto r de R faça
    Adicione-o ao buffer da partição de R segundo a função h_M(r)
Fim-Para
Para cada partição R_i de R faça
    // Geração das partições RM_i ao traduzir IDOs lógicos em físicos
    Converta o IDO lógico para IDO físico usando MAP
    // Particiona RM_i em K partições, isto é, geração das partições RM_ij
    Adicione-o ao buffer da partição de RM_i segundo a função h_S(s)
Fim-Para
Para j = 1 a K faça
    Leia RM_1j, RM_2j, ..., RM_Nj e agrupe todas as partições numa única partição
    // Geração da partição RMS_j
    Realiza a junção por ponteiro entre a partição gerada e S_j
Fim-Para
Gerar resultado ao agrupar todas as partições RMS para formar a resposta RS
```

Figura 3.4 – Pseudo-código do algoritmo Partition-Merge

Um dos benefícios trazidos por esse algoritmo é a sucessão de fases de fragmentação e agrupamento (*Partition-Merge*). Esta heurística beneficia E/S, pois ajuda a manter o agrupamento dos objetos presentes nos relacionamentos a serem percorridos numa localidade próxima. Outra vantagem deste algoritmo é a baixa exigência de memória, já que somente uma ou duas partições envolvidas em cada operação (*Partition* ou *Merge*) precisam estar carregadas em memória. Apesar do custo de E/S deste algoritmo ser bastante similar ao custo dos algoritmos de junção baseados em ponteiros que usam o operador materialização, a principal sobrecarga provocada pela execução deste algoritmo vem do custo de CPU, já que o número de acessos a tabelas *hash* de suporte é bem maior que o do algoritmo materializado baseado em ponteiros. Entretanto, note que, diferentemente do algoritmo de junção baseada em valor mostrado na figura 3.2, somente a coleção R precisa ser particionada.

Se a expressão de caminho a ser processada for de tamanho maior que dois, será necessária uma seqüência de aplicações deste algoritmo. O resultado do $P(PM)^1M$ serve como parâmetro de entrada para a junção, utilizando o mesmo algoritmo $P(PM)^1M$, com a próxima coleção envolvida na expressão de caminho de acordo com a direção a ser processada. Este processo se repete até que todas as coleções da expressão de caminho sejam processadas $(P(PM)^*M)$.

3.3 Algoritmo Naïve Pointer Chasing (NP)

O algoritmo "*Naïve Pointer Chasing*" ("caça ingênua a ponteiros") é baseado na navegação entre os objetos através dos atributos de relacionamento. Para que a navegação seja possível, muitos SGBDOOs e SGDBROs automaticamente convertem os identificadores de objetos (IDOs) armazenados como valores de atributo para ponteiros em memória quando o sistema carrega objetos da memória secundária para a memória principal. Este processo chama-se *pointer swizzling* e pretende otimizar a navegação de objetos associados em memória principal (CATTELL, 1994).

A idéia do algoritmo é extremamente simples e intuitiva. Para cada objeto dentro da coleção-raiz, todo o caminho da expressão é percorrido. No percurso dos atributos de relacionamento, se o IDO armazenado no atributo de relacionamento é de um objeto que não está carregado em memória principal, sua página é imediatamente carregada. No final do percurso, se o caminho for válido, isto é, os objetos participantes da navegação do caminho satisfazem aos predicados aninhados, as estruturas desejadas como resposta da consulta (projeção) são geradas e o próximo objeto da coleção-raiz tem o seu caminho avaliado de modo análogo. Se o caminho for inválido, isto é, o caminho iniciado por um IDO não alcança um objeto na coleção-folha ou se algum objeto do caminho não satisfaz o predicado aninhado, ele simplesmente é descartado. O processo se repete para todos os objetos de todas as páginas da coleção-raiz. A figura 3.5 ilustra o comportamento do algoritmo.

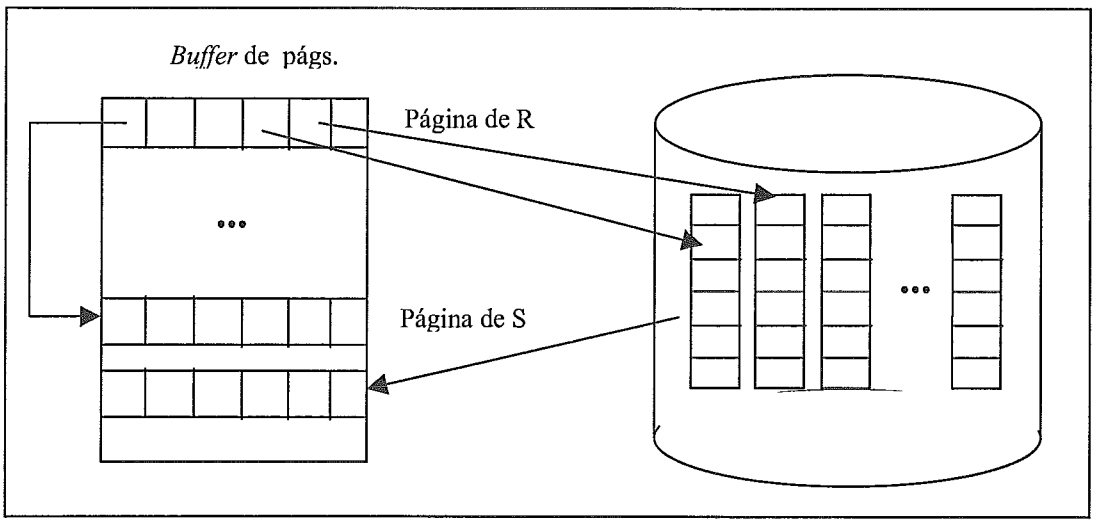

Figura 3.5 – Algoritmo *Naïve Pointer Chasing*

Vale a pena salientar que o processo acima é válido se a direção a ser processada é a direção descendente, ou seja, a direção natural em que a consulta foi especificada pelo

usuário. Caso a direção a ser percorrida pelo algoritmo seja a ascendente, o processo descrito no parágrafo anterior é o mesmo, mas a coleção de partida do percurso não é a coleção-raiz, mas sim a coleção-folha. Além disso, são usados os atributos dos relacionamentos inversos, se os mesmos existirem. Note que o NP é muito mais simples que o algoritmo P(PM)*M. Este último possui sucessivas fases de fragmentação e agrupamento, incrementando o custo de CPU consideravelmente em relação ao primeiro. A figura 3.6 apresenta o pseudo-código do algoritmo NP. O algoritmo da figura 3.6 deve ser executado para todos os objetos da coleção-raiz ou folha, dependendo da direção a ser percorrida. Repare também a natureza recursiva do algoritmo, já que ele pode ser usado para percorrer todos os sub-caminhos da expressão.

```
// Algoritmo Naive
// parâmetro de entrada: IDO  (identificador do objeto a ser percorrido)
Leia objeto Obj cujo identificador seja IDO
Se o predicado aninhado da coleção é válido para o objeto Obj Então
   Se Obj ∉ coleção-folha Então    // ou coleção-raiz se a direção for ascendente
      oid_rel := IDO do atributo de referência Ai da expressão de caminho
      Se o relacionamento é monovalorado Então
         Naive(oid_rel)
      Senão
         Para oid_rel = todos os IDOs dentro do atributo do tipo coleção Ai Faça
            Naive(oid_rel)
         Fim-Para
      Fim-Se
   Fim-Se
Fim-Se
```

Figura 3.6 – Pseudo-código do algoritmo *Naïve Pointer Chasing*

3.4 Algoritmo Smart Naïve (SN)

A seção 3.1 serviu para analisar as vantagens e desvantagens da utilização do algoritmo NP. Esta análise nos motivou à elaboração de um algoritmo que preservasse as vantagens da utilização do NP e que permitisse o bom desempenho nas condições em que ele não se comportasse tão bem. Este algoritmo foi denominado de ***Smart Naïve*** (SN).

A idéia básica do algoritmo NP se manteve: para cada objeto de uma coleção (raiz ou folha), é percorrido todo o caminho antes de avaliar o caminho do próximo objeto da coleção. Entretanto, ao custo de uma tabela de suporte em memória, todos os IDOs dos objetos percorridos durante o caminho são armazenados na tabela de suporte (se o

caminho for válido). Sempre que houver a necessidade de se percorrer um novo caminho através do IDO de um objeto, é verificado se este IDO já não foi percorrido por outro caminho, ou seja, se o IDO está presente na tabela de suporte. Se estiver, este mesmo caminho não deverá ser mais percorrido, pois algum objeto já o percorreu e o resultado deste percurso foi válido. Se o IDO não está presente na tabela de suporte, o caminho iniciado por este IDO será percorrido até que se chegue ao final do caminho ou até a um outro sub-caminho que já foi percorrido e está armazenado na tabela de suporte.

Dessa forma, a principal desvantagem do algoritmo NP (alto grau de compartilhamento) passou a configurar uma situação favorável à utilização do SN. O problema do NP é que uma mesma página poderia ser carregada muitas vezes, já que objetos de páginas diferentes poderiam apontar para o mesmo caminho numa ordem desfavorável (aleatória) ao uso do *cache* de páginas do SGBD. Se esta situação ocorrer no SN, quando um objeto que aponta para um caminho válido já percorrido for iniciar sua navegação, este caminho já estará armazenado na tabela de suporte. Sendo assim, todas as páginas pertencentes aos objetos da coleção C_i que são apontados por mais de um objeto da coleção C_{i-1} deixam de ser recarregadas e processadas, assim como as páginas dos objetos das coleções seguintes (C_{i+1}, C_{i+2}, ...), economizando assim tempo de E/S e de CPU. A figura 3.7 ilustra o comportamento do algoritmo.

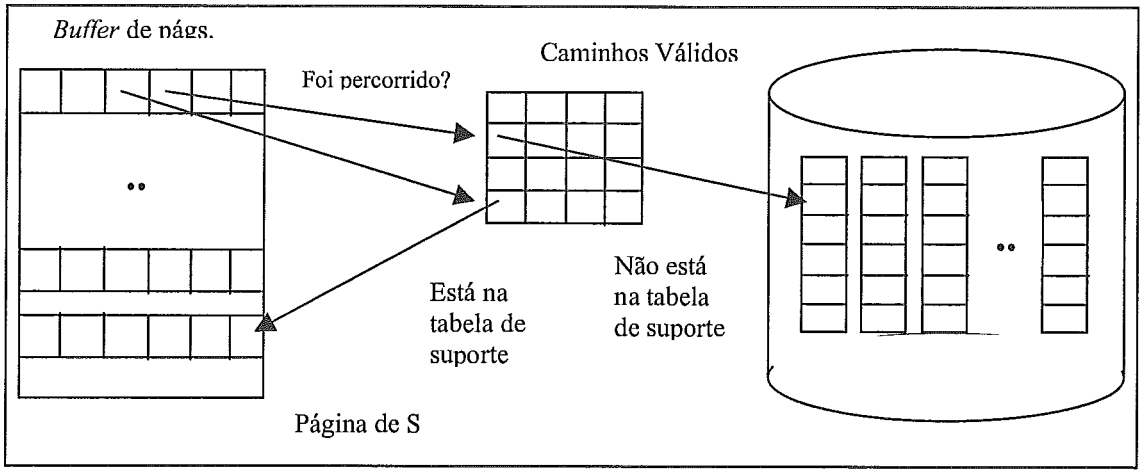

Figura 3.7 – Algoritmo *Smart Naïve*

A tabela de suporte pode ser encarada como uma "memória" de caminhos válidos percorridos e pode ser ajustada para ter o tamanho necessário para a quantidade de memória disponível no sistema. Assumimos que a tabela de suporte cabe em memória,

mas se essa premissa não for verdadeira, podemos particioná-la e trabalhar somente com um conjunto de partições em memória. A implementação da tabela de suporte também não sofre de complexidade elevada. A tabela de suporte nada mais é que uma tabela *hash* e pode ser utilizada também na implementação de algoritmos de junção binária. A figura 3.8 mostra o pseudo-código deste algoritmo.

```
// Algoritmo Smart_Naive
// parâmetro de entrada: IDO        (identificador do objeto a ser percorrido)
// Tabela hash CAMINHOs_VALIDOS inicialmente vazia
Leia objeto Obj cujo identificador seja IDO
Se o predicado aninhado da coleção é válido para o objeto Obj Então
   Se Obj ∈ coleção-folha Então          // ou coleção-raiz se a direção for ascendente
      Insere IDO em CAMINHOS_VALIDOS
   Senão
      oid_rel := IDO do atributo de referência Ai da expressão de caminho
      Se o relacionamento é monovalorado Então
         Se oid_rel está armazenado na tabela CAMINHOS_VALIDOS Então
            Insere IDO em CAMINHOS_VALIDOS
         Senão
            Smart_Naive(oid_rel)
         Fim-Se
      Senão
         Para oid_rel ∈ todos os IDOs dentro do atributo do tipo coleção Ai Faça
            Se oid_rel está armazenado na tabela CAMINHOS_VALIDOS Então
               Insere IDO em CAMINHOS_VALIDOS
            Senão
               Smart_Naive(oid_rel)
            Fim-Se
         Fim-Para
      Fim-Se
   Fim-Se
Fim-Se
```

Figura 3.8 – Pseudo-código do algoritmo *Smart Naïve*

Note que, pela figura 3.8, este algoritmo é bastante semelhante ao algoritmo NP. A diferença básica entre eles é que o SN evita o percurso de um caminho já percorrido ao checar nas tabelas de suporte se o mesmo caminho já não foi percorrido.

Existe ainda uma otimização que pode ser realizada sobre este algoritmo. Pode-se guardar em outra tabela de suporte os IDOs de objetos que pertencem a caminhos inválidos na expressão. Sendo assim, ao procurar um IDO na tabela de suporte que guarda os caminhos válidos, se esse IDO não existir na tabela, pode-se ainda procurá-lo na tabela de suporte que guarda os caminhos inválidos antes de iniciar a navegação. Se o IDO estiver armazenado nesta tabela, seu caminho também não precisa ser percorrido.

Caso o IDO também não esteja armazenado nesta tabela, finalmente o objeto deverá ser carregado e o caminho restante deverá ser percorrido.

É possível, embora não seja freqüente, que a definição de uma expressão de caminho contenha a mesma coleção (extensão da classe) em diferentes pontos. Nesse caso, é necessário que a tabela de suporte armazene, além do IDO do objeto que inicia o percurso de um caminho (válido ou não, dependendo do contexto da tabela), o índice i associado ao ponto da expressão de caminho em que a coleção C_i está sendo avaliada. Dessa forma, elimina-se a ambigüidade provocada pela participação do mesmo objeto em diferentes pontos dos caminhos formados, já que ao recuperar o IDO do objeto na tabela de suporte, verifica-se também se o índice armazenado é igual ao ponto da expressão de caminho que está sendo avaliado no momento.

Com relação ao operador binário, a principal vantagem do algoritmo SN sobre o algoritmo P(PM)*M está na ausência completa de resultados intermediários durante o processamento da expressão de caminho. A geração de partições intermediárias para a memória secundária ocorre nas duas fases de fragmentação do algoritmo P(PM)*M. Na primeira fragmentação, são gerados N fragmentos em disco contendo as partições de R com os IDOs lógicos dos atributos de referência. Na segunda fragmentação, para cada uma das N partições previamente salvas em disco, são gerados K fragmentos (partições RM_i) contendo os objetos de R agrupados pelo atributo de referência já convertido para o IDO físico. Além disso, são geradas N partições em disco como resultado da navegação dos objetos (junção por ponteiros) entre as partições RM_i e a partição de S correspondente (geração da partição RMS_i). Portanto, toda a criação e manutenção das partições geradas durante o processamento de apenas duas coleções (R e S) da expressão de caminho causam uma sobrecarga no custo de E/S e CPU que não pode ser desprezada.

Entretanto, o custo de E/S do P(PM)*M não fica tão comprometido, já que o número de acessos a disco durante a junção por ponteiros é baixo por causa do benefício trazido pelo agrupamento dos objetos. Por outro lado, a fusão de partições RM_i para formarem uma única partição a ser usada na junção por ponteiros eleva consideravelmente o custo de CPU. O mesmo pode ser dito para a fusão das partições RMS_i para compor o resultado.

Por isso, o custo de CPU do algoritmo SN mostra-se (intuitivamente, por enquanto) mais barato do que o custo do algoritmo P(PM)*M, já que não há resultados

intermediários a serem guardados no processamento da expressão de caminho. Todo o caminho de um objeto é percorrido sem que alguma estrutura em memória secundária seja armazenada e/ou recuperada. Embora haja um custo embutido na manipulação de tabelas *hash*, que também são utilizadas no algoritmo P(PM)*M, o algoritmo SN se mostra eficiente principalmente se pensarmos em expressões de caminho longas.

Além disso, se a memória for um recurso abundante no ambiente computacional do SGBD, as páginas trazidas durante o processamento de um caminho podem perdurar no *buffer* de páginas até o final do processamento da consulta, evitando que sejam descartadas e recarregadas posteriormente. Vale a pena ressaltar que a manipulação da tabela de suporte no algoritmo SN ajuda a diminuir o custo de E/S, caso o grau de compartilhamento entre os objetos seja alto, já que para cada IDO encontrado na tabela durante o percurso de um caminho, as páginas das coleções subseqüentes deixam de ser carregadas.

No próximo capítulo, apresentamos os resultados para os custos de E/S e CPU obtidos durante a execução dos experimentos. Realizamos também uma análise qualitativa dos resultados, apresentando as principais heurísticas extraídas dos experimentos e que podem ser incorporadas a um otimizador de consultas para auxiliar o processamento de expressões decaminho.

Capítulo 4. Ambiente Experimental

O principal objetivo deste trabalho é avaliar experimentalmente a eficiência de diferentes algoritmos para o processamento de expressões de caminho. Portanto, para que os algoritmos pudessem ser implementados, era necessário que houvesse um ambiente em que o código fonte do SGBD utilizado estivesse disponível, já que só assim era possível analisar o comportamento de cada algoritmo de forma correta, além de conhecer as estruturas internas utilizadas pelo SGBD para que cada algoritmo pudesse ser implementado. Além disso, a depuração e as otimizações que podem ser implementadas sobre qualquer algoritmo é facilitada pela presença do código fonte do SGBD. Seria impossível implementar um algoritmo e avaliar sua eficiência em qualquer SGBD comercial, já que seu código fonte não estaria disponível.

Sendo assim, foi utlizado como sistema de persistência de objetos o GOA (MAURO *et al.*, 1997). Suas principais características e serviços são descritos detalhadamente na seção 4.1. As principais extensões e modificações no código fonte do GOA para a implementação dos algoritmos são mostradas na seção 4.1.2. Finalmente, na seção 4.2 é apresentado o *benchmark* OO7, que foi utilizado na implementação dos experimentos.

Vale a pena ressaltar que muitos trabalhos encontrados na literatura sobre processamento de expressões de caminho ou junção entre coleções realizam, como visto na seção 2.4, análises por simulações. Embora a construção de um modelo de custo seja importante para a compreensão correta do problema, além da praticidade de avaliação de um vasto número de cenários sem o mesmo esforço gasto para a avaliação do mesmo cenário experimentalmente, é essencial que toda a análise de simulação tenha uma validação experimental. Muitos detalhes de implementação e do ambiente experimental acabam causando impacto nos resultados e esses detalhes muitas vezes não são contemplados na função de custo.

O sistema computacional utilizado para os experimentos é composto por uma máquina Intel (Pentium II – 500 Mhz) com 256 Mb de memória RAM rodando o sistema operacional Windows NT Server 4.0. Enquanto o servidor GOA executava nesta máquina, eram enviadas consultas OQL provenientes de uma máquina cliente. O custo de comunicação entre o cliente e o servidor para o envio do resultado da consulta

foi desprezado, sendo relevante apenas o custo de processamento da consulta no servidor.

4.1. GOA

O GOA (http://www.cos.ufrj.br/~goa) é um sistema desenvolvido pelo grupo de pesquisa em banco de dados do Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ. Nele são oferecidos serviços de persistência de objetos e gerência de coleções de objetos. Entre os serviços de gerência oferecidos estão incluídos o processamento de consultas sobre coleções, agrupamento de objetos no disco, utilização de técnicas para gerência de *cache*, técnicas de indexação de objetos e armazenamento e exportação de documentos XML. Ele foi desenvolvido tendo em mente as especificações dos padrões ODMG (CATTELL *et al.*, 2000). Sendo assim, o GOA possui um módulo cliente capaz de criar esquemas baseados na interface ODL e consultar as coleções de objetos armazenados através de consultas escritas na linguagem OQL. Entretanto, apenas um sub-conjunto destas linguagens foi implementado (MATTOSO *et al.*, 2000).

O modelo de comunicação das aplicações que acessam bases no GOA é baseado na arquitetura cliente/servidor. A interação entre o usuário e o GOA é realizada através de aplicações clientes que se comunicam com o servidor via *sockets*. A versão atual do GOA está disponível tanto para o sistema Windows como para os sistemas Linux. Atualmente estão implementadas APIs para as linguagens C++, Java e Object Pascal (Delphi). Todas as APIs implementadas até o momento possuem a mesma interface, ou seja, o conjunto de classes disponíveis para cada uma destas linguagens é o mesmo, facilitando a tarefa do programador.

4.1.1. Arquitetura

Desde a última publicação acadêmica de seus serviços (MATTOSO *et al.*, 2000), o GOA já evoluiu bastante da sua proposta inicial (MAURO *et al.*, 1997). Isto se justifica não somente por motivos corretivos, mas também em decorrência da implementação de novos trabalhos do grupo de pesquisa em banco de dados da COPPE/UFRJ. Além disso, seu código fonte vem sendo constantemente inspecionado e algumas classes foram reprojetadas para melhorar a modularidade e manutenabilidade. Entretanto, seu conjunto básico de funções e serviços vem sendo mantido.

Atualmente, o GOA possui 4 módulos principais, como pode ser visto pela figura 4.1. São eles: Gerente de Esquemas (*GoaSchemaManager*), Gerente de Páginas (*GoaPageManager*), Gerente de Objetos (*GoaObjectManager*) e Gerente de Consultas (*GoaQueryManager*).

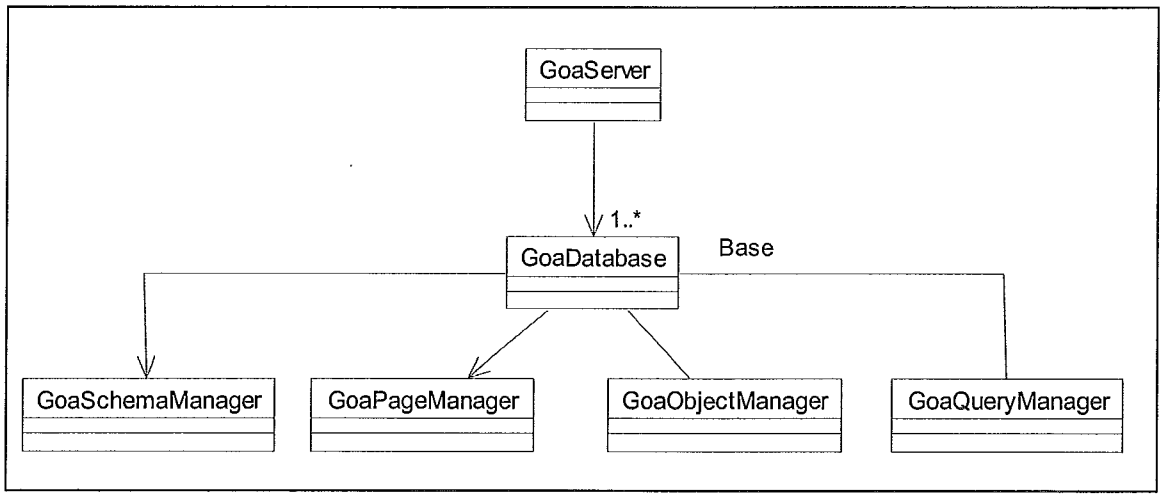

Figura 4.1 – Diagrama de classes (UML) da arquitetura do GOA

O Gerente de Esquemas é responsável pelo controle do esquema das classes armazenadas e definidas pelo usuário no servidor GOA. O Gerente de Páginas é o módulo responsável pela transferência de páginas do disco para a memória principal e pelo gerenciamento do *cache*. O Gerente de Objetos é o módulo responsável pela persistência e gerência de objetos e coleções. Finalmente, o Gerente de Consultas é responsável pela execução e otimização das consultas submetidas ao servidor pelo usuário.

O módulo de maior interesse nesta dissertação é o Gerente de Consultas. O Gerente de Consultas cria estruturas em memória que representam a consulta submetida pelo usuário. A partir destas estruturas, a consulta é analisada e processada pelo processador de consultas. A figura 4.2 mostra o diagrama de classes que implementam a otimização e execução de consultas no GOA.

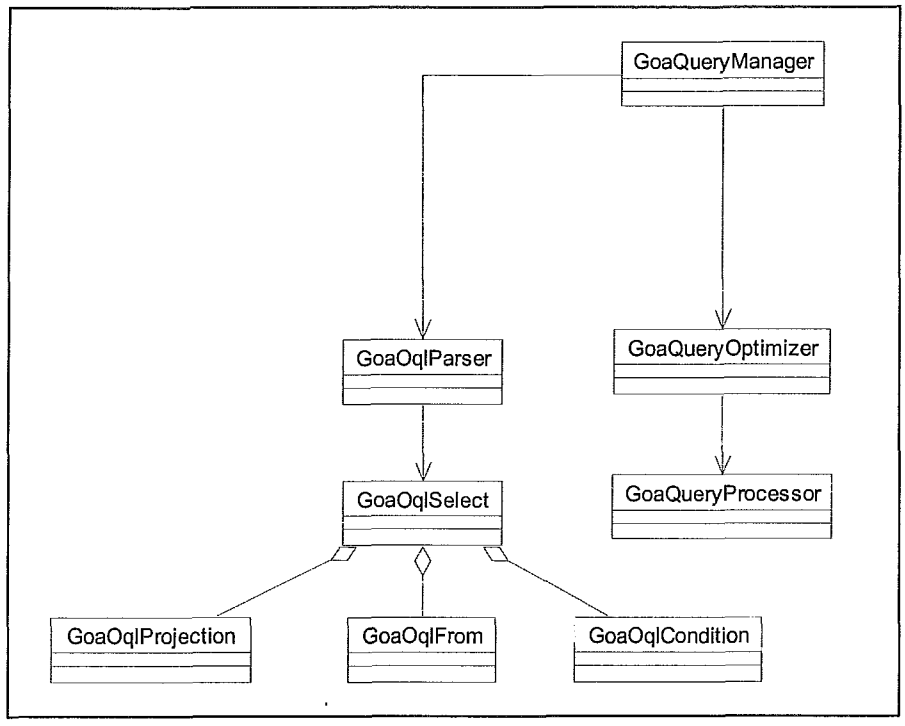

Figura 4.2 – Diagrama de classes (UML) do Gerente de Consultas

O usuário submete uma consulta OQL ao servidor através do módulo cliente. A consulta, ao ser recebida pelo servidor, é repassada ao Gerente de Consultas. O Gerente de consultas a repassa para Analisador OQL (*GoaOqlParser*), que a interpreta. A interpretação da expressão de consulta OQL é feita pelos analisadores gerados pelo Flex e Bison (LEVINE *et al.*, 1992). Se a consulta estiver sintaticamente correta, uma representação interna da consulta (instância da classe *GoaOqlSelect*) é gerada e servirá de entrada para o Otimizador de Consultas (*GoaQueryOptimizer*). O Otimizador de Consultas vai escolher dentre todas as possíveis instâncias da classe *GoaQueryProcessor* conhecidas aquela que for mais adequada para execução. Depois de escolhida a instância *GoaQueryProcessor* adequada, a consulta é executada pela instância escolhida.

Outro módulo importante para o processamento de consultas no GOA é o Gerente de Páginas. O Gerente de Páginas é o módulo responsável por recuperar objetos do disco e gerenciar as páginas recuperadas no *cache*. Toda solicitação de objetos ao Gerente de Objetos é repassada para o Gerente de Páginas. O Gerente de Páginas testa se a página do objeto solicitado está armazenada no *cache*. Se isto for verdade, o Gerente de Páginas devolve a página do objeto ao Gerente de Objetos, que extrai o

objeto da página. Entretanto, se a página do objeto solicitado não estiver no *cache*, o Gerente de Páginas recupera a página do disco e a armazena no *cache* (se tiver espaço no *cache*). Se não houver espaço no *cache*, o Gerente de Páginas libera a página menos acessada recentemente e armazena a página recém lida do disco na posição liberada.

4.1.2. Alterações e extensões realizadas no GOA

Para a implementação dos algoritmos a serem avaliados experimentalmente, foram implementadas classes derivadas da classe *GoaQueryProcessor*. A figura 4.3 mostra as três classes construídas por herança da classe *GoaQueryProcessor* e que implementam os algoritmos P(PM)*M, *Smart Naïve* e de junção baseada em valor. Esses três algoritmos foram avaliados experimentalmente e representam as instâncias de processadores de consulta disponíveis para o otimizador da classe *GoaQueryOptimizer*.

Observe que o algoritmo *Naïve Pointer Chasing* não é mostrado na figura 4.3. Isto se justifica pelo fato do algoritmo *Smart Naïve* se comportar de forma análoga ao algoritmo *Naïve Pointer Chasing* quando não há utilização de tabelas de suporte. Por isso, o algoritmo *Smart Naïve* foi construído de forma que a utilização das tabelas de suporte fosse parametrizada. Assim, a classe *GoaSmartNaiveProcessor* é utilizada tanto para a execução do algoritmo *Smart Naïve* quanto para o algoritmo *Naïve Pointer Chasing*.

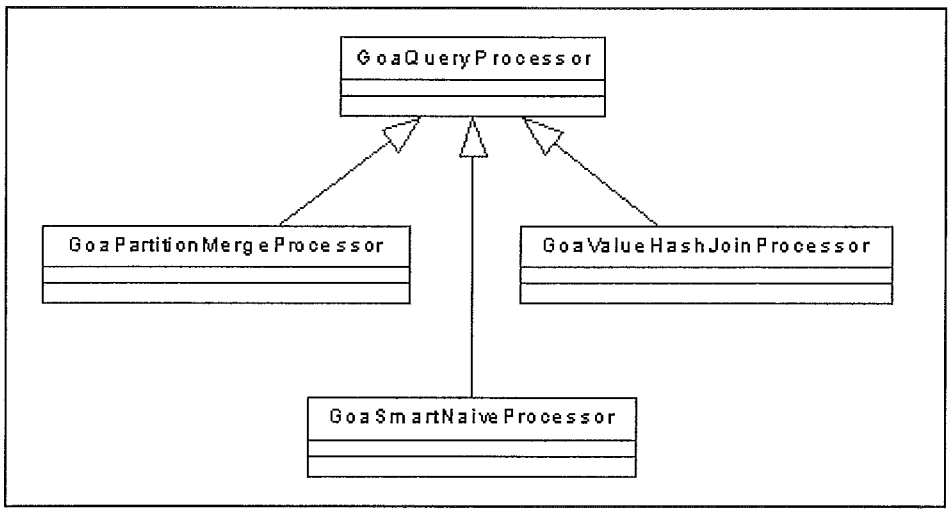

Figura 4.3 – Processadores de consulta implementados no GOA

Além disso, foi necessário criar uma classe para armazenar os parâmetros da expressão de caminho, tais como as coleções envolvidas, os predicados a serem aplicados dentro de cada coleção, os relacionamentos envolvidos entre as classes e a

direção a ser utilizada no processamento. Esta classe foi denominada de *PathExpression*.

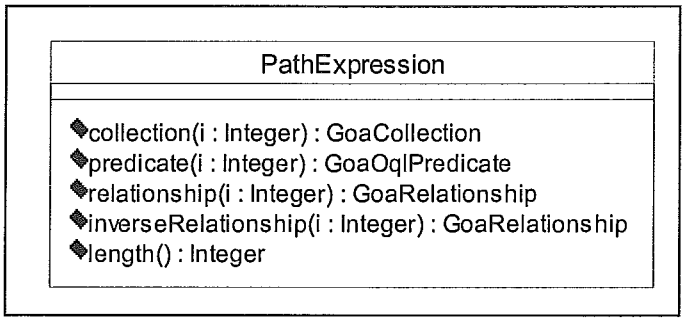

Figura 4.4 – Classe *PathExpression*

A implementação desta classe é mostrada resumidamente na figura 4.4. O projeto de interface da classe teve como objetivo se aproximar da representação de expressão de caminho mostrada no capítulo 2. A partir de uma instância desta classe, o Otimizador de Consulta pode estimar o custo de processamento de cada uma das estratégias em ambas as direções (ascendente ou descendente). Dessa forma, dependendo do melhor custo de execução estimado sobre o objeto, a instância da classe *GoaQueryProcessor* (*GoaSmartNaiveProcessor* ou *GoaPartitionMergeProcessor*) será criada e responsável pela execução da expressão de caminho.

Existe ainda uma proposta de integração de um otimizador de consultas OQL gerado pelo OPTGEN (FEGARAS, 1998) com o GOA (VICTOR e RUBERG, 1999). Nesta proposta foram estabelecidas as construções necessárias no GOA para que ele pudesse absorver o otmizador gerado pela OPTGEN. É interessante observar que novas regras de otimização de consulta podem ser programadas no OPTGEN, baseadas na função de custo proposta por RUBERG (2001), por exemplo. Dessa forma, pode-se construir um otimizador de consultas personalizado que leve em conta os algoritmos disponíveis do operador n-ário, como o algoritmo SN proposto, por exemplo.

4.2. Benchmark OO7

O *Benchmark* OO7 (CAREY *et al.*, 1993) vem sendo, nos últimos anos, a ferramenta padrão para a avaliação de desempenho de SGBDOOs (DEWITT *et al.*, 1996, GARDARIN *et al.*, 1996, ÖZSU *et al.*, 1998, SU *et al.*, 1999, dentre outros). Seu objetivo é testar o desempenho do processamento de consultas e travessias nas mais variadas formas de sistemas orientados a objeto. Permite ainda a análise de importantes aspectos do modelo OO, tais como herança e agregação, no desempenho das consultas,

assim como o desempenho de operações como acesso ao *cache* e ao disco bem como atualizações na base, inclusão e exclusão de objetos. A figura 4.5 representa o esquema de classes do *benchmark* na notação UML.

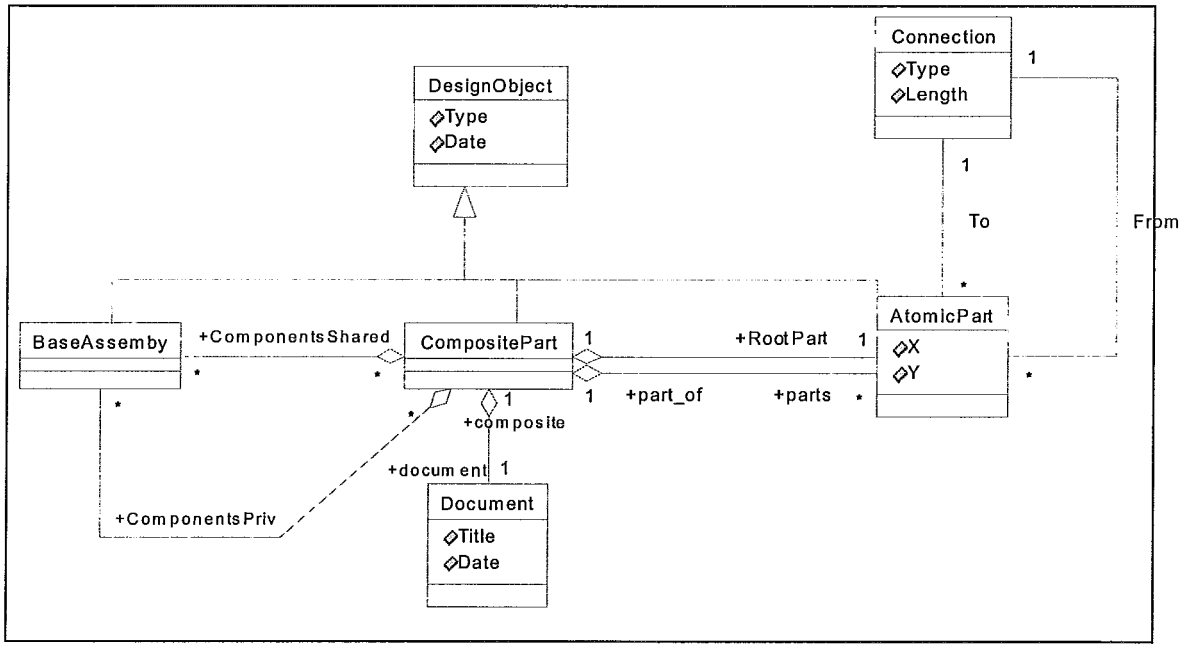

Figura 4.5 – Diagrama de Classes do OO7 em UML

Algumas simplificações foram feitas no esquema de classes do OO7 original (figura 4.5): as classes *Module*, *ComplexAssembly* e *Manual* foram retiradas do esquema, já que as consultas analisadas não envolvem percurso por essas classes. Entretanto, a sua utilização não interfere nos resultados coletados. A descrição em ODL das classes utilizadas do *benchmark* pode ser vista na figura 2.1 do capítulo 2. As cardinalidades especificadas na configuração média do OO7 são mostradas na tabelas 4.1.

A idéia geral deste *benchmark* é representar um modelo de aplicação CAD/CAM. Seu modelo baseia-se no projeto de uma peça (*DesignObject*), que pode ser uma peça atômica (*AtomicPart*) ou uma peça composta (*CompositePart*), que é uma peça cuja composição é formada por várias peças atômicas. As peças atômicas podem conectar-se umas às outras para formarem peças mais funcionais. A conexão entre duas peças atômicas é representada pela classe *Connection*. As peças compostas também podem ser combinadas, formando peças da classe *BaseAssembly*. Finalmente, toda peça composta possui uma documentação associada, representada pela classe *Document*.

Tabela 4.1. Parâmetros das coleções

Parâmetro	Valores
Tam. página (bytes)	2048
Número de páginas	
CompositePart	16
Document	18
AtomicPart	3421
Connection	9019
Número de objeto/pág.	Cardinalidade
CompositePart	31 (496 objetos)
Document	29 (522 objetos)
AtomicPart	29 (99209 objetos)
Connection	33 (297627 objetos)
Tam. Objeto (bytes)	
CompositePart	64
Document	69
AtomicPart	69
Connection	61
NumAtomic/Comp	200
NumConn/Atomic	3

Foi desenvolvida nessa dissertação uma aplicação chamada **OO7Gen (OO7 Generator)** que gera *scritps* para a carga de objetos do *benchmark* OO7 no GOA. Depois que o OO7Gen gera os arquivos de *scripts*, os mesmos são carregados no servidor. Todos os parâmetros mostrados na tabela 4.1 são definidos no OO7Gen e os arquivos com os *scritps* de carga dos objetos são gerados instantaneamente. Além destes parâmetros, o grau de participação dos objetos nos relacionamentos e o grau de compartilhamento também podem ser definidos. Portanto, o trabalho de geração de uma base OO7 no GOA foi automatizado e simplificado pelo OO7Gen, pois nos permitiu gerar diferentes configurações de bases rapidamente e construir cenários de avaliação experimental com menor esforço.

O ajuste dos relacionamentos entre objetos de classes diferentes, como por exemplo `AtomicPart` e `CompositePart`, é realizado de forma aleatória pelo OO7Gen. Dessa forma, muitos objetos de páginas diferentes podem ser escolhidos para compor a coleção do relacionamento. Sendo assim, a distribuição dos objetos no relacionamento permite uma análise mais justa do potencial de cada um dos algoritmos.

Capítulo 5. Resultados Obtidos no Processamento de Expressões de Caminho

Este capítulo apresenta os resultados obtidos na execução de consultas OQL com expressões de caminho sobre diferentes configurações de base do *benchmark* OO7. Cada configuração representa um cenário específico em que desejou-se mostrar o desempenho dos algoritmos NP, SN, P(PM)*M e junção por valor. Na legenda dos gráficos apresentados nas seções seguintes, as siglas dos algoritmos aparecem seguidas da direção de avaliação da expressão de caminho. As linhas pontilhadas representam os algoritmos na direção ascendente. Para o operador binário, os algoritmos são chamados de PM (algoritmo P(PM)*M) e JV (algoritmo *hybrid hash join* por valor). Duas métricas são relevantes na análise dos resultados: o custo de E/S e o custo de CPU.

A consulta avaliada experimentalmente é mostrada na figura 5.1. Justifica-se a escolha desta consulta pelo fato dela conter uma expressão de caminho cujo tamanho é maior que dois (tamanho três) e pelo fato das coleções envolvidas possuírem cardinalidades bem distintas (496 objetos da classe `CompositePart`, 99209 objetos de `AtomicPart` e 297267 objetos de `Connection`). Além disso, a consulta da figura 5.1 permite que dois tipos de expressão de caminho (monovalorada e multivalorada) sejam avaliadas. Na direção descendente, a expressão de caminho é multivalorada devido a presença dos atributos de referência do tipo coleção (`parts` e `to`). Como os atributos inversos foram definidos para estes relacionamentos, na direção ascendente a expressão de caminho é monovalorada (atributos `from` e `part_of`). A figura 5.2 mostra como a consulta da figura 5.1 pode ser representada em ambas as direções pela notação definida na seção 2.1. Outro aspecto importante desta consulta é que a aplicação do predicado de seleção pode ser estudada em duas situações distintas: no final (descendente) ou no início (ascendente) da expressão de caminho.

```
select c
from c in CompositeParts, a in c.parts, con in a.to
where con.Length > 0
```

Figura 5.1 - Consulta OQL avaliada experimentalmente

```
Direção descendente: c.parts-a.to-con(Length > 0)
Direção ascendente: con(Length > 0).from.part_of
```

Figura 5.2 - Representação da expressão de caminho contida na consulta da figura 5.1

O custo de E/S dos algoritmos foi obtido experimentalmente. Nesta dissertação, o custo de E/S é expresso pelo número de acessos a disco provocados por "falhas" no *cache* de páginas do servidor GOA (*cache miss*). Por "falha" no *cache* entende-se que, durante o processamento da consulta, não foi possível recuperar a página de algum objeto solicitado diretamente do *cache* do GOA. Quando isso acontece, o servidor GOA solicita a página do objeto através de uma operação de E/S e a carrega no *buffer* de páginas do *cache* do GOA. Portanto, foram contabilizados, para cada algoritmo, a freqüência com que o Gerente de Páginas do GOA realiza operação de E/S para a recuperação de uma página.

É importante frisar que não podemos garantir que uma solicitação de E/S ao sistema operacional realize, de fato, um acesso a disco, já que a página pode estar no *cache* do disco ou do próprio sistema operacional. Entretanto, devido às dificuldades em "forçar" o acesso a disco de fato, optamos por considerar o acesso ao *cache* do GOA significativo para representar as operações de E/S realizadas pelo sistema operacional.

O custo de CPU, por sua vez, foi avaliado de forma simulada pela função de custo proposta em BRAUMANDL *et al.* (2000). Isto se deve ao fato do ambiente de execução dos experimentos não ter propiciado recursos para a extração do tempo gasto somente pela CPU. A única medida de tempo experimental que foi possível obter na realização dos experimentos foi o tempo de resposta da execução da consulta. Esta medida, entretanto, não é adequada para representar o custo das operações de CPU, já que nela também está embutido todo o tempo gasto nas operações de E/S e outras interrupções. Por outro lado, a análise dos algoritmos pelo modelo de custo proposto por BRAUMANDL *et al.* (2000) é bastante representativa, pois foi validada experimentalmente e reflete de forma precisa o custo das operações de CPU de cada um dos algoritmos avaliados.

Os cenários utilizados para a análise dos resultados de cada algoritmo foram escolhidos de acordo com as heurísticas geradas em RUBERG (2001). Estes cenários

oferecem condições de execução abrangentes e que permitem uma análise efetiva do operador n-ário e que muitas vezes não são contemplados em outros trabalhos. Como já comentado na seção 3.1 do capítulo 3, SHEKITA e CAREY (1990), DEWITT *et al.* (1993) e BRAUMANDL *et al.* (2000) são alguns dos trabalhos que não avaliam algumas destas situações, como por exemplo, a variação da participação dos objetos nos relacionamentos. Os cenários avaliados neste capítulo são:

- Variação do fator de seletividade na coleção `Connections` (10%, 50% e 90%).
- Variação da participação dos objetos da classe `AtomicPart` nos relacionamentos. A participação é classificada em:
  - participação total – 100% dos objetos participam;
  - participação parcial – 1%, 10% e 50% dos objetos participam.
- Variação da memória através do tamanho do *cache* de páginas (2, 4, 8, 16 e 32 Mbytes disponíveis).
- Variação do grau de compartilhamento dos objetos nos relacionamentos (1, 3 e 10).

Vale a pena salientar que o tamanho da página que contém os objetos armazenados no GOA foi fixado em 2 Kbytes (2048 bytes). Com exceção do cenário que se propõe a analisar justamente a variação do *buffer* (*cache*) de páginas disponível para o servidor, todos os cenários restantes tiveram suas consultas executadas com o *cache* fixo em 4 Mbytes (2048 páginas de 2048 bytes cada). Outro aspecto que deve ser mencionado é que os resultados coletados na realização dos experimentos correspondem à execução "fria" das consultas. Em outras palavras, toda vez que uma consulta era submetida para a execução no servidor GOA, o *cache* de páginas era primeiramente esvaziado para que os algoritmos não se beneficiassem das páginas previamente armazenadas.

A aplicação **OO7Gen**, utilizada na geração da base de objetos, associou dez valores distintos (0, 10, 20, 30, 40, 50, 60, 70, 80, 90) para o domínio do atributo `Length` da classe `Connection`. A distribuição destes valores pelos atributos dos objetos da coleção `Connections` é feita de maneira uniforme e circular.

As seções seguintes mostram os resultados obtidos para os custos de E/S e CPU. Primeiramente, são mostrados os custos de E/S dos algoritmos em cada um dos cenários apresentados nesta seção. Para cada cenário, uma explicação mais detalhada sobre a sua utilidade no contexto de expressões de caminho também é fornecida. Finalmente, uma análise da função de custo proposta em BRAUMANDL *et al.* (2000) é realizada para a comparação do custo de CPU entre os algoritmos.

5.1 Custo de Entrada e Saída

5.1.1 Variação do Fator de Seletividade

O objetivo em avaliar este cenário é apurar o impacto da seletividade aplicada sobre uma das coleções da expressão de caminho. Na consulta da figura 5.1, a seletividade é aplicada sobre a coleção `Connections`. Além disso, como mostra a figura 5.2, foi possível avaliar o impacto da seletividade tanto no início (direção ascendente) quanto no final (direção descendente). O tamanho do *cache* de páginas foi mantido em 4Mbytes e a participação dos objetos nos relacionamentos é total, ou seja, todos os objetos das coleções participam dos relacionamentos presentes na expressão de caminho.

Foram escolhidos três fatores de seletividade sobre a coleção `Connections`: 0.1, 0.5 e 0.9, ou seja, 10%, 50% e 90% dos objetos da classe `Connection` satisfazem ao predicado da consulta. Como a distribuição dos valores do domínio do atributo `Length` é uniforme sobre todos os objetos da coleção, aproximadamente 10% dos objetos possuem o valor 0 para este atributo, outros 10% possuem o valor 10, e assim sucessivamente. A figura 5.1 representa a consulta cujo fator de seletividade aplicado é aproximadamente 0.9. Para os fatores 0.5 e 0.1, o predicado da consulta da figura 5.1 foi modificado, como mostram as figuras 5.3 e 5.4. A tabela 5.1 apresenta o número de acessos a disco obtidos (*cache miss*) por cada algoritmo do operador binário e n-ário sobre as consultas da figura 5.2, 5.3 e 5.4. Na figura 5.5, o gráfico correspondente à tabela 5.1 que representa o comportamento dos algoritmos sobre cada uma das seletividades testadas é apresentado.

```
Direção descendente: c.parts-a.to-con(Length > 40)
Direção ascendente: con(Length > 40).from.part_of
```

Figura 5.3 - Expressão de caminho com 0.5 de fator de seletividade sobre `Connections`

```
Direção descendente: c.parts-a.to-con(Length < 10)
Direção ascendente: con(Length < 10).from.part_of
```

Figura 5.4 - Expressão de caminho com 0.1 de fator de seletividade sobre `Connections`

Tabela 5.1 – Número de acessos a disco de cada algoritmo (*cache miss*) na variação do fator de seletividade

Fator de Selet. / Algoritmo	0.9	0.5	0.1
NP-Descendente	446178	446178	446178
NP-Ascendente	129111	79135	28035
SN-Descendente	446178	446178	446178
SN-Ascendente	58031	53433	26743
JV-Descendente	113721	113721	113721
JV-Ascendente	113721	113721	113721
PM-Descendente	112155	112155	112155
PM-Ascendente	13625	13625	13537

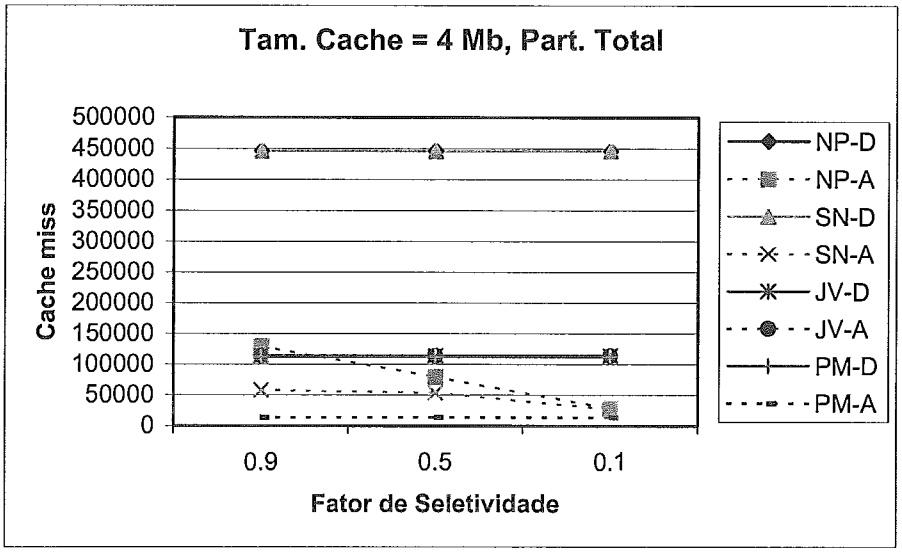

Figura 5.5 – Variação dos fatores de seletividade

A partir dos resultados apresentados na tabela 5.1 e no gráfico da figura 5.5, alguns fatos interessantes podem ser observados. O primeiro deles é que os algoritmos para o operador n-ário não apresentam bom desempenho na direção descendente. Eles se mostram igualmente ineficientes perante aos outros algoritmos na mesma direção (JV-D e PM-D). Como só há predicado de seleção na coleção-folha, todos os objetos das coleções intermediárias são percorridos até que o objeto da coleção-folha seja acessado.

Na direção descendente, mesmo que o objeto acessado da coleção-folha não satisfaça ao predicado de seleção, o percurso de todo o seu caminho foi necessário para que este fato fosse comprovado. Isto justifica a igualdade dos valores da tabela 5.1 e o pior desempenho, quando comparado à direção ascendente, para cada um dos algoritmos, independente do fator de seletividade aplicado. Este problema ocorre tanto para os algoritmos n-ários como para os binários, com exceção do JV. Porém, no algoritmo PM o acesso aos objetos presentes na expressão de caminho é realizado de forma agrupada (função *hash*), enquanto nos algoritmos n-ários ele é feito na ordem (aleatória) em que os objetos são percorridos na expressão de caminho. Nem a utilização da tabela de suporte pelo algoritmo SN foi capaz de melhorar o seu desempenho em relação ao NP, já que, nas consultas executadas, não há compartilhamento de objetos na direção descendente. Uma instância da classe `AtomicPart` só é referenciada por uma instância da classe `CompositePart`, assim como um objeto da classe `Connection` só é apontado por um da classe `AtomicPart`. Portanto, a utilização da tabela de suporte não trouxe benefícios na direção descendente, já que nenhum dos caminhos armazenados na tabela durante o processamento da expressão de caminho foi reutilizado para evitar o percurso dos caminhos restantes.

Por outro lado, na direção ascendente o fator de seletividade afeta positivamente o desempenho dos algoritmos baseados no operador n-ário. Embora o algoritmo PM-A tenha apresentado o melhor desempenho geral na execução das consultas com os três fatores de seletividade, os algoritmos n-ários (SN-A e NP-A) são os que apresentam a maior queda no número de acessos a disco à medida que o fator de seletividade diminui. Além disso, à medida que o fator de seletividade diminui, o desempenho dos algoritmos SN-A, NP-A e PM-A se aproxima, como mostra o gráfico da figura 5.5. Na direção ascendente, a utilização da tabela de suporte trouxe benefícios ao desempenho do

algoritmo SN-A, como mostra a tabela 5.1, já que o grau de compartilhamento entre os objetos aumenta. Na configuração do *benchmark* OO7 gerada para este cenário, cada instância de `AtomicPart` é apontada por três instâncias de `Connection`, assim como uma instância de `CompositePart` é apontada por duzentas de `AtomicPart`, como mostra a tabela 4.1 apresentada no capítulo 4. Ao percorrer o caminho iniciado por uma instância de `Connection` que aponta para uma de `AtomicPart` já percorrida, o caminho armazenado na tabela de suporte evita que as páginas dos objetos acessados no restante deste caminho sejam novamente carregadas.

Pela tabela 5.1, percebe-se que o algoritmo PM não é sensível ao fator de seletividade em ambas as direções. Isto pode ser explicado pela heurística utilizada neste algoritmo, que agrupa os objetos da coleção de partida (coleção-raiz ou folha, dependendo da direção) segundo os objetos fisicamente próximos da coleção relacionada. Por isso, quando a junção por ponteiros entre uma partição RM_i e a coleção S é realizada, poucos acessos a disco são necessários, independente do fator de seletividade, já que a leitura de apenas uma página do disco para o *buffer* resolve a junção de praticamente todos os objetos daquela partição RM_i.

Portanto, pode-se afirmar que o fator de seletividade é muito mais relevante para os algoritmos n-ários do que para os binários, já que os algoritmos SN e NP percorrem a expressão de caminho na ordem (aleatória) em que os objetos aparecem nos relacionamentos. Qualquer caminho que possa ser eliminado, em decorrência do fator de seletividade aplicado sobre a coleção-raiz (ou coleção-folha, dependendo da direção), contribui significativamente para o desempenho dos algoritmos n-ários, já que um expressivo conjunto de páginas deixa de ser carregado. Os algoritmos binários reduzem as operações de E/S pela própria natureza dos algoritmos (aplicação da função *hash* sobre o atributo de referência da expressão de caminho).

Finalmente, o desempenho do algoritmo JV se mostra indiferente não só ao fator de seletividade, mas também em relação a direção em que a expressão de caminho é percorrida. Note que, em todas as seletividades testadas, os algoritmos JV-D e JV-A apresentam o mesmo desempenho. Isto se justifica pelo fato do algoritmo JV não tirar proveito dos ponteiros presentes nos relacionamentos. Dessa forma, os IDOs dos atributos de referência são encarados como valores de chaves estrangeiras e, por conseguinte, na junção por valor entre duas coleções, ambas as coleções são percorridas e particionadas pela aplicação da função *hash*. Por causa disso, mesmo que existam

objetos da coleção relacionada que não participem da expressão de caminho, ou que sejam apontados por objetos da coleção-raiz (ou coleção-folha, dependendo da direção) que não satisfaçam ao predicado de seleção, eles serão percorridos durante a formação das partições *hash*. Note ainda que, na direção descendente, o desempenho do algoritmo JV-D é até satisfatório, pois seu custo é levemente superior ao do algoritmo PM, mas na direção ascendente, o algoritmo SN-A se mostra muito mais eficaz que o JV-A (na pior seletividade, o algoritmo SN-A chega a ser 50% mais eficiente que o JV-A), além de apresentar expressivos ganhos de desempenho conforme o fator de seletividade decresce.

5.1.2 Variação da Participação dos Objetos nos Relacionamentos

Neste cenário, o principal objetivo foi estudar o impacto provocado pelo grau de participação dos objetos nos relacionamentos da expressão de caminho. Este é um aspecto muito freqüente em aplicações reais, já que muitas vezes nem todos os elementos de uma numerosa coleção participam dos relacionamentos definidos no esquema.

Em SGBDs relacionais, por exemplo, o relacionamento multivalorado entre duas relações é implementado pela construção de uma relação intermediária. É comum que esta relação tenha uma cardinalidade menor, e às vezes bem menor, que a cardinalidade das relações participantes do relacionamento. Um exemplo típico é o cadastro de filmes, o cadastro de clientes e a relação de empréstimos numa locadora de vídeo. A freqüência com que este cenário é encontrado em aplicações reais justifica a sua análise, muito embora na maioria dos trabalhos revisados na seção 2.4, ela não seja contemplada pelos seus autores.

Duas situações distintas foram verificadas durante a execução dos experimentos: a participação total e a participação parcial dos objetos da classe `AtomicPart` nos relacionamentos da expressão de caminho definida na consulta da figura 5.1. Na participação parcial, diferentes graus de participação foram testados: aproximadamente 1%, 10% e 50% dos objetos da coleção `AtomicParts` participaram da expressão de caminho. O restante dos objetos não foi relacionado a nenhuma instância das classes `CompositePart` e `Connection` pelo **OO7Gen**, a aplicação responsável pela criação das diferentes configurações da base.

Na participação total, a configuração da base utilizada foi a mesma utilizada na avaliação do cenário apresentado na seção anterior, ou seja, todos os objetos da coleção `AtomicParts` relacionavam-se com três instâncias de `Connection` e com uma instância de `CompositePart`, como mostra a tabela 4.1 do capítulo 4. A tabela 5.2 apresenta os custos de E/S, em todos os graus de participação testados, para cada um dos algoritmos avaliados. O gráfico da figura 5.6 mostra o comportamento dos algoritmos.

Tabela 5.2 – Número de acessos a disco de cada algoritmo (*cache miss*) na variação do grau de participação dos objetos nos relacionamentos

Grau de Part. / Algoritmo	100%	50%	10%	1%
NP-Descendente	446178	159521	21780	2081
NP-Ascendente	129111	73004	25857	13549
SN-Descendente	446178	159521	21780	2081
SN-Ascendente	58031	36535	17524	11881
JV-Descendente	113721	113721	113721	113721
JV-Ascendente	113721	113721	113721	113721
PM-Descendente	112155	62354	16717	2081
PM-Ascendente	13625	13625	13318	11382

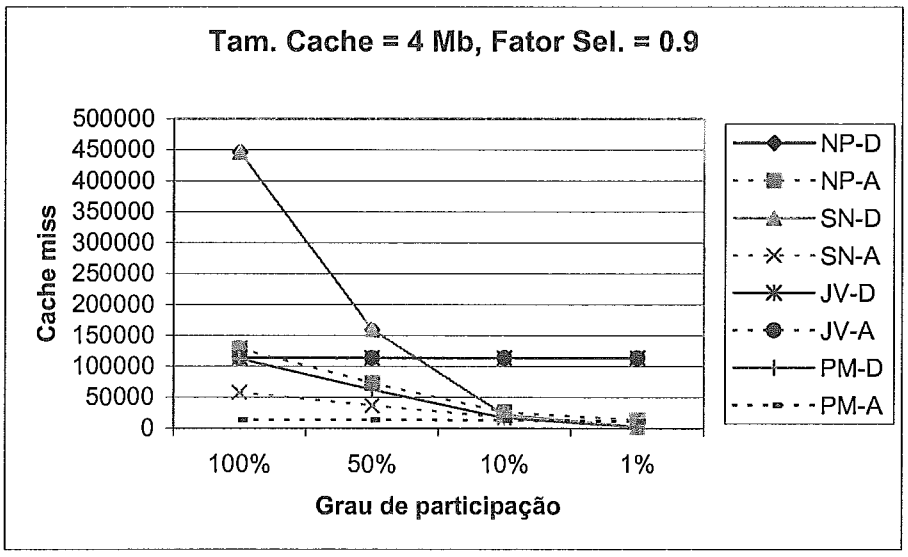

Figura 5.6 – Variação do grau de participação nos relacionamentos

Um fato importante a ser observado nos resultados da tabela 5.2 é que cada algoritmo na direção descendente apresenta uma queda no custo de E/S bem superior à queda apresentada pelo mesmo algoritmo na direção ascendente, exceto para o

algoritmo JV. Inclusive, em algumas colunas da tabela (10% e 1%), as versões descendentes superaram as versões ascendentes do mesmo algoritmo. No cenário avaliado na seção anterior, por exemplo, onde a participação dos objetos é total, as versões ascendentes superaram as descendentes em todas as configurações, já que o percurso na direção ascendente favorecia o uso do *cache* de páginas, devido ao compartilhamento de objetos presentes nos diferentes caminhos. Entretanto, este fato não se repetiu neste cenário. Com o grau de participação em 10%, o algoritmo NP-D superou o NP-A, e para 1% de participação, os algoritmos NP-D, SN-D e PM-D alcançaram um desempenho superior aos algoritmos NP-A, SN-A e PM-A, respectivamente.

A principal justificativa para este fato está associada a cardinalidade da coleção de partida no processamento da expressão de caminho. Quando o grau de participação dos objetos na consulta da figura 5.1 é baixo, grande parte dos objetos percorridos na coleção de partida são descartados, pois não estão relacionados a nenhuma instância da coleção seguinte. Por isso, é mais vantajoso percorrer a expressão de caminho a partir da coleção cuja cardinalidade seja a menor possível. Neste caso, o percurso na direção descendente oferece esta possibilidade, já que a cardinalidade de `CompositeParts` é bem menor que a cardinalidade de `Connections`.

Em todas as configurações testadas, o algoritmo PM se mostrou a melhor escolha do ponto de vista de E/S. Entretanto, para 1% de participação nos relacionamentos, os algoritmos n-ários SN-D e NP-D também chegaram ao melhor desempenho, empatados com o algoritmo PM-D. Além disso, pelo gráfico da figura 5.6, o desempenho destes três algoritmos se aproxima à medida que a taxa de participação diminui. Mais uma vez torna-se evidente que qualquer recurso (baixo fator de seletividade e baixa participação dos objetos no relacionamento, por exemplo) utilizado para reduzir o espaço de caminhos a serem percorridos na expressão de caminho causa mais impacto aos algoritmos n-ários do que aos binários, que já se beneficiam da ordem (agrupamento da função *hash* aplicada) em que os objetos são percorridos. Isto explica a brusca queda nos gráficos dos algoritmos n-ários, principalmente na direção descendente. Note que, para o algoritmo PM, a queda é mais tênue na direção ascendente, embora seja mais significativa na descendente.

O desempenho do algoritmo JV novamente se mostrou indiferente, não só em relação a direção em que a expressão de caminho é percorrida, mas também quanto ao

grau de participação dos objetos. Mesmo com uma participação parcial baixa (1%), os objetos de todas as coleções envolvidas na expressão de caminho são acessados durante a geração das partições *hash*. Dessa forma, a utilização deste algoritmo pode ser descartada quando houver participação parcial, devido ao melhor desempenho apresentado pelos demais algoritmos.

5.1.3 Variação da Memória Disponível

Durante muitos anos, o foco dos trabalhos em processamento de consultas se manteve em propostas de algoritmos que otimizassem o acesso a disco, pois este aspecto sempre foi considerado crítico nas aplicações em banco de dados. Aliado a isso, os algoritmos eram propostos visando a utilização moderada da memória, já que este sempre foi um recurso computacional caro e não abundante nos ambientes computacionais existentes.

Entretanto, a utilização de memória vem deixando de ser, mais recentemente, uma restrição para os SGBDs, principalmente porque os custos diminuíram e os sistemas estão cada vez mais robustos. Até mesmo uma máquina doméstica não possui menos do que 64 Mbytes de memória principal. A maioria dos SGBDs comerciais exigem como requisito mínimo de execução uma quantidade considerável de memória, muitas vezes superior a 256 Mbytes.

A quantidade de memória disponível para o *buffer* de páginas é um dos fatores que limita a aplicação dos algoritmos baseados no operador n-ário. Para que a navegação entre os objetos envolvidos numa expressão de caminho seja possível, é necessário que, a cada navegação, os objetos sejam carregados da memória secundária para a principal. Durante este processo, normalmente os sistemas convertem os IDOs armazenados como valores de atributo para ponteiros em memória. Este mecanismo chama-se *pointer swizziling* (CATTELL, 1994).

Se durante a navegação do caminho de um objeto, o IDO presente no atributo de referência for de um objeto cuja página já se encontra no *buffer* de páginas, o objeto relacionado é extraído da página e o próximo atributo de referência, se existir, é processado. No entanto, se a página do objeto relacionado não estiver no *buffer*, ela é automaticamente recuperada do disco. Nesse momento, pode não haver espaço disponível no *buffer* para recebê-la. Se isto ocorrer, alguma página deverá ser descartada para ceder o lugar à página recém chegada. Na pior das hipóteses, havendo

compartilhamento de objetos nos caminhos a serem processados na expressão de caminho, em algum momento a página recém liberada será novamente carregada no *buffer*.

Portanto, uma condição favorável à execução dos algoritmos do operador n-ário é o *cache* de páginas ser grande o suficiente para contabilizar um número mínimo de páginas descartadas durante o processamento da expressão de caminho. Isto compensaria o acesso aleatório realizado pelos algoritmos NP e SN. É importante também verificar se esta condição favorece a outros algoritmos, como o PM, por exemplo. Não adianta o algoritmo PM ser eficiente com pouca memória disponível se, num ambiente real de aplicação, esse recurso for abundante e o algoritmo não tira proveito desse excesso.

Sendo assim, esse cenário se propõe a avaliar a influência da quantidade de memória disponível (*buffer* de páginas) no desempenho dos algoritmos. A tabela 5.3 mostra o número de acessos a disco obtido por cada algoritmo, enquanto o gráfico da figura 5.7 mostra como cada algoritmo se comporta frente à variação de memória. A consulta executada neste cenário é a mesma da figura 5.1 e a configuração da base é a mesma utilizada na avaliação da participação total dos objetos nos relacionamentos da expressão de caminho.

Tabela 5.3 – Número de acessos a disco de cada algoritmo (*cache miss*) na variação da memória

Tamanho do *cache* / Algoritmo	2Mb	4Mb	8Mb	16Mb	32Mb
NP-Descendente	471106	446178	397440	306364	168343
NP-Ascendente	202766	129111	15033	13625	13625
SN-Descendente	471106	446178	397440	306364	168343
SN-Ascendente	83153	58031	17704	13626	13625
JV-Descendente	113721	113721	113721	113721	113721
JV-Ascendente	113721	113721	113721	113721	113721
PM-Descendente	112155	112155	112155	112155	112155
PM-Ascendente	13625	13625	13625	13625	13625

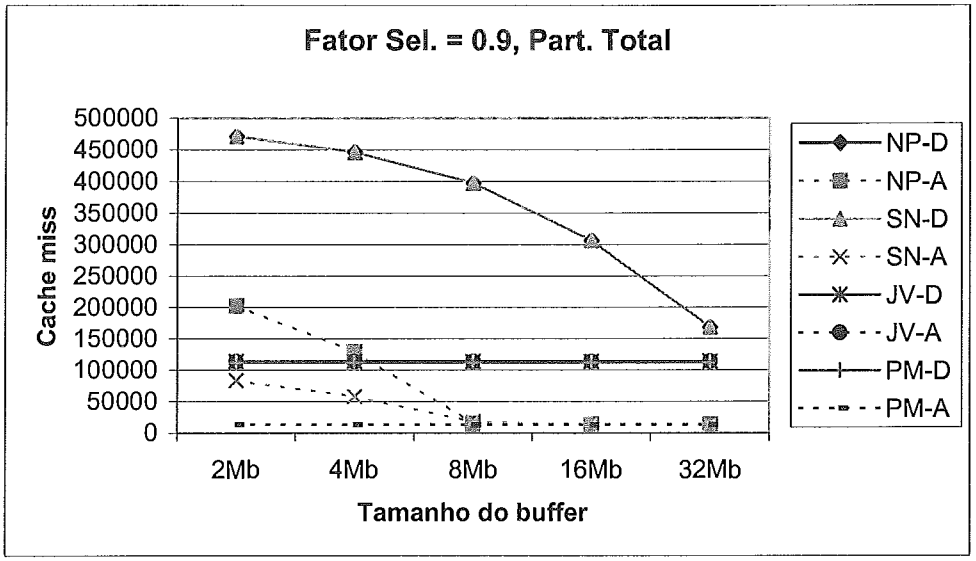

Figura 5.7 – Variação do tamanho do *buffer* de páginas

As versões ascendentes dos algoritmos voltaram a superar os seus similares descendentes, isto é, o mesmo algoritmo apresenta melhor desempenho na direção ascendente do que na descendente. Diferente do cenário anterior, onde a participação parcial dos objetos foi verificada, todos os objetos da coleção `Connections` que satisfazem ao predicado de seleção participam da expressão de caminho. Como o percurso na direção ascendente favorece o uso do *cache* de páginas, devido ao compartilhamento de objetos presentes nos diferentes caminhos, é menor o número de acessos a disco na execução dos algoritmos nesta direção.

Os resultados revelam novamente a eficiência do algoritmo PM-A. Em todas as configurações de memória, este algoritmo apresentou o melhor desempenho. No entanto, a partir de 16 Mbytes, os custos dos algoritmos NP-A e SN-A alcançaram o melhor custo obtido pelo algoritmo PM-A, como mostra a tabela 5.3. Analisando o gráfico da figura 5.7, os algoritmos SN-A, NP-A e PM-A praticamente se equivalem a partir de 8 Mbytes.

Além disso, percebe-se que os algoritmos n-ários são mais sensíveis à variação de memória do que os algoritmos binários, que se mostraram indiferentes a essa variação. A memória disponível para o *buffer* determina diretamente o desempenho dos algoritmos n-ários. Quanto maior a memória disponível, maior será o número de páginas armazenadas no *buffer* e, conseqüentemente, menor será o número de páginas descartadas durante a navegação entre os objetos. A figura 5.7 mostra que os únicos

gráficos que apresentam queda à medida que a memória cresce são as dos algoritmos n-ários.

Um fato interessante que pode ser observado a partir do gráfico e da tabela 5.3 é que o algoritmo NP, na direção ascendente, é mais sensível à variação de memória do que o algoritmo SN. Isto pode ser explicado pela utilização da tabela de suporte no algoritmo SN. O algoritmo SN evita o percurso sobre objetos compartilhados em diferentes caminhos, utilizando as tabelas de suporte. Apesar deste recurso ter propiciado uma economia de acessos a disco, quando o algoritmo não encontrava o IDO a ser percorrido em nenhuma das tabelas de suporte disponíveis (caminhos válidos e inválidos), verificou-se que, na maioria dos casos, a página do objeto solicitado também não estava no *cache*. Por conseguinte, uma operação de E/S era necessária para recuperar a página do objeto cujo IDO tinha que ser percorrido. Por outro lado, sempre que o algoritmo NP precisava de uma página que não estava no *cache*, ele iniciava uma operação de E/S automaticamente.

Portanto, comparando-se a utilização do *cache* entre os algoritmos NP e SN, o *cache* esteve mais carregado de páginas durante a execução do algoritmo NP. No algoritmo SN, muitos caminhos deixaram de ser percorridos em função da utilização da tabela de suporte. Uma vez que o algoritmo SN resolve o percurso de um caminho ao descobrir o IDO armazenado na tabela de suporte, ele deixa de trazer para a memória todos os objetos armazenados na mesma página que não foi recuperada. No percurso dos próximos caminhos formados por algum destes objetos, certamente um número maior de operações de E/S foi realizada pelo algoritmo SN, já que a tabela de suporte não tem armazenados os objetos ainda não percorridos. Isto justifica a vitória do algoritmo NP-A sobre o SN-A para 8 Mbytes de memória. Este foi o único cenário onde o SN apresentou um custo superior ao custo do NP.

Os algoritmos n-ários mostraram um fraco desempenho na direção descendente, principalmente nas menores configurações de memória. Este comportamento já era esperado, pois o grau de compartilhamento não pôde ser aproveitado. Porém, à medida que o tamanho da memória cresce, esses algoritmos também apresentaram melhoras no desempenho. De acordo com o gráfico, os algoritmos n-ários ascendentes não apresentam melhoras a partir de 8Mbytes. Já na direção descendente, eles foram capazes de apresentar ganhos em todos os tamanhos do *buffer*. Pelo comportamento apresentado

no gráfico, as linhas dos algoritmos n-ários am ambas as direções teriam se cruzado em ambientes com maiores configurações de memória.

Entretanto, apesar dos ganhos apresentados para os algoritmos do operador n-ário, é importante ressaltar um aspecto favorável ao algoritmo PM. Em todos os experimentos, admitiu-se que a tabela de suporte do algoritmo SN cabia na memória principal, assim como as partições RM_i e RMS_i do algoritmo PM (não simultaneamente). Se esta premissa não for válida, os algoritmos n-ários serão mais prejudicados em relação ao custo de E/S do que o algoritmo PM. Isto porque algumas entradas da tabela de suporte do algoritmo SN deverão ser armazenadas em disco. Ao recuperá-las, o custo de E/S do algoritmo SN é incrementado. No caso do algoritmo PM, a solução será diminuir o tamanho da partição, isto é, diminuir o número de objetos armazenados em cada partição (RM_i e RMS_i). Dessa forma, o custo de E/S do algoritmo PM se mantém praticamente inalterado, pois apesar do número de partições ter crescido, o tamanho de cada partição diminuiu, compensando esta sobrecarga. Obviamente, o número de acessos a disco será maior, embora o custo total de E/S permaneça aproximadamente inalterado.

Esse aspecto justifica os valores constantes do algoritmo PM na tabela 5.3, tanto na direção descendente, quanto na ascendente. Os resultados do algoritmo PM mostrados na tabela 5.3 correspondem a implementação do algoritmo para um número pequeno de partições. Mesmo na menor configuração de memória (2 Mbytes), o ambiente experimental proporcionou que as partições RM_i e RMS_i coubessem em memória principal. Por isso o algoritmo apresentou o mesmo desempenho em todas as configurações, independente do tamanho do *buffer*, já que desde a menor configuração de memória, o tamanho das partições foi sempre o mesmo. O mesmo pode ser dito para o algoritmo JV. Note que, embora os algoritmos n-ários apresentem ganhos à medida que a memória é incrementada, o melhor resultado desses algoritmos não foi melhor que o resultado do algoritmo PM-A.

5.1.4 Variação do Grau de Compartilhamento

O último cenário avaliado mostra a influência do grau de compartilhamento (*share*) entre as coleções envolvidas na expressão de caminho durante o processamento de consultas. Este aspecto já foi avaliado quando os algoritmos foram executados na direção ascendente nos cenários da seção 5.1.1, 5.1.2 e 5.1.3. Nas configurações de base

geradas até o momento, não existe compartilhamento na direção descendente. Nesta direção, o grau de compartilhamento entre os objetos das coleções é um, ou seja, não há objetos em nenhuma das três coleções envolvidas na consulta que participem de mais de um caminho percorrido pelos algoritmos.

Neste cenário, deseja-se avaliar o impacto da variação do grau de compartilhamento em ambas as direções. Porém, na definição original do esquema do OO7, como mostra a figura 2.1 do capítulo 2, o relacionamento entre as classes `AtomicPart` e `CompositePart` é muitos-para-um, ou seja, cada instância de `AtomicPart` é apontada por somente uma instância de `CompositePart`. O mesmo é válido para as instâncias das classes `Connection` e `AtomicPart`. Conseqüentemente, para que fosse possível realizar a análise do grau de compartilhamento em ambas as direções, foi necessário modificar o esquema original do *benchmark* OO7. O relacionamento entre as classes `AtomicPart` e `CompositePart` e entre `Connection` e `AtomicPart` foi transformado em muitos-para-muitos, ou seja, as classes `Connection` e `AtomicPart` tiveram seus atributos de referência transformados em atributos do tipo coleção. A figura 5.9 mostra a nova definição do esquema para estas duas classes.

```
class AtomicPart extends DesignObj
(extent AtomicParts)
{
    attribute int x;
    attribute int y;
    attribute string name;
    relationship set(Connection) to
        inverse Connection::from;
    relationship set(Connection) from
        inverse Connection::to;
    relationship set(CompositePart) part_of
        inverse CompositePart::parts;
};

class Connection
(extent Connections)
{
    attribute string type;
    attribute int length;
    relationship set(AtomicPart) from
        inverse AtomicPart::to;
    relationship set(AtomicPart) to
        inverse AtomicPart::from;
};
```

Figura 5.8 – Esquema OO7 (ODL) modificado

Foram testados três diferentes graus de compartilhamento entre as classes do novo esquema definido na figura 5.8: 1, 3 e 10. Isto quer dizer que para o grau um, cada instância de `AtomicPart` é apontada por somente uma instância de `CompositePart`, assim como cada instância de `Connections` só é apontada por uma de `AtomicPart`. Para o grau 3, cada instância de `AtomicPart` é apontada por três instâncias de `CompositePart`, e assim sucessivamente.

A tabela 5.4 apresenta os resultados obtidos na execução da consulta da figura 5.1. Foi gerada uma base diferente para cada grau de compartilhamento testado. Todas as bases foram contruídas sobre o novo esquema definido na figura 5.8. O gráfico da figura 5.9 mostra o comportamento dos algoritmos em cada configuração. Nas três bases geradas, a participação dos objetos nos relacionamentos da expressão de caminho é total.

É importante destacar que na geração das bases avaliadas neste cenário, o grau de saída (*fan-out*) foi relaxado para que o grau de compartilhamento fosse ajustado. A rotina implementada no **OO7Gen** que monta os relacionamentos entre os objetos gerados atribui aleatoriamente um número de objetos especificados pelo grau de compartilhamento. Por isso, um objeto pode ter sido mais selecionado do que outros durante o ajuste dos relacionamentos montados. Nos cenários avaliados anteriormente, o grau de saída é ajustado de acordo com a tabela 4.1 do capítulo 4.

Tabela 5.4 – Número de acessos a disco de cada algoritmo (*cache miss*) na variação do grau de compartilhamento

Grau de Comp. / Algoritmo	1	3	10
NP-Descendente	444413	2714372	9110466
NP-Ascendente	461961	2795803	9470200
SN-Descendente	444413	435874	406975
SN-Ascendente	461961	452361	422369
JV-Descendente	113721	399733	1326745
JV-Ascendente	113721	399733	1326745
PM-Descendente	112156	113125	302043
PM-Ascendente	380609	380699	406205

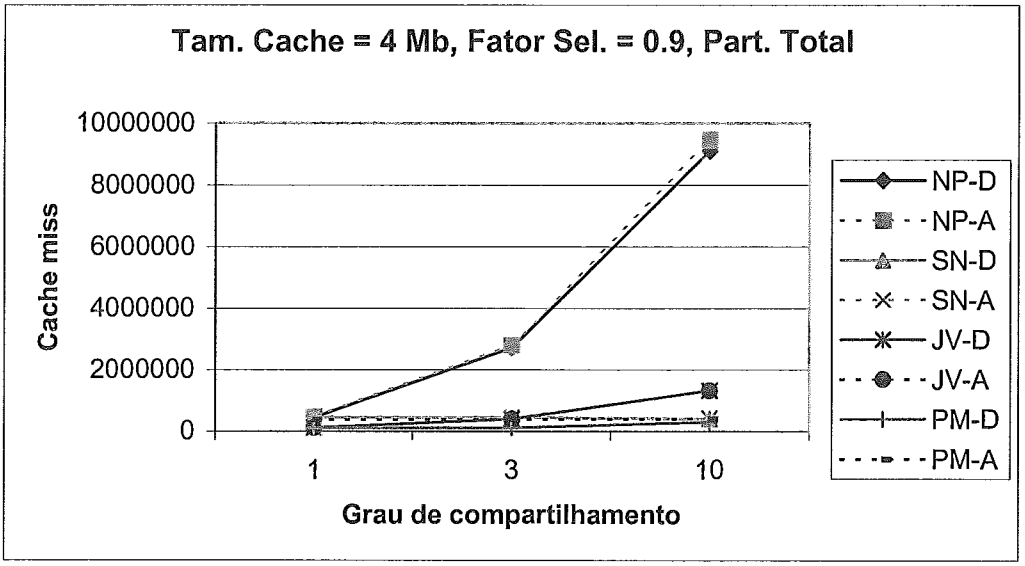

Figura 5.9 – Variação do grau de compartilhamento

Os resultados mostrados confirmam um fato já esperado. O grau de compartilhamento é um dos aspectos que prejudica o desempenho do algoritmo NP. Repare que, pela tabela 5.4, à medida que o grau de compartilhamento cresce, seu desempenho piora sensivelmente em ambas as direções. O gráfico da figura 5.9 mostra esse comportamento. Embora todos os algoritmos tenham apresentado uma leve queda de desempenho conforme o grau de compartilhamento foi sendo incrementado, com exceção do SN, a queda é mais acentuada no algoritmo NP, devido à recarga de páginas já prevista em RUBERG (2001) e provocada pela ordem aleatória no acesso aos objetos presentes nos relacionamentos da expressão de caminho.

O algoritmo PM novamente apresentou o melhor custo de E/S em todas as configurações testadas. A eficácia do agrupamento dos objetos de partida que estão relacionados a objetos fisicamente próximos da coleção relacionada é o principal motivo do bom desempenho deste algoritmo, não só neste cenário, mas em todos os demais. Porém, o benefício trazido pela otimização proposta no algoritmo NP que originou o algoritmo SN pode ser explicitamente percebido neste cenário. Enquanto todos os algoritmos apresentaram queda de desempenho em maiores graus de compartilhamento, o algoritmo SN, por sua vez, apresenta uma leve melhora. Pelo gráfico do algoritmo SN, esta melhora se traduz num comportamento constante frente a variação dos valores de compartilhamento.

O desempenho do algoritmo JV se mostrou satisfatório para o grau de compartilhamento um. No entanto, seu desempenho também é degradado quando o algoritmo executa em maiores taxas de compartilhamento. Quando o grau de compartilhamento é três, ele ainda leva uma pequena vantagem sobre o algoritmo SN, mas quando o grau de compartilhamento atinge o valor máximo, o custo do algoritmo JV chega a ser três vezes maior que o do SN. Portanto, o único algoritmo que beneficiou-se do acréscimo do grau de compartilhamento foi o SN, enquanto todos os demais algoritmos, em ambas as direções, foram piorando nos resultados apresentados. Em vista disso, espera-se que para maiores graus de compartilhamento entre as coleções, provavelmente o algoritmo SN será o algoritmo de menor custo de E/S, já que todos os demais algoritmos apresentam quedas sensíveis de desempenho à medida que este valor aumenta. A utilização das tabelas de suporte contribui significativamente para que o número de acessos a disco fosse reduzido.

É importante salientar que o acréscimo do grau de compartilhamento entre as coleções provoca uma considerável sobrecarga nas cardinalidade das coleções participantes da expressão de caminho. Como a expressão de caminho é multivalorada em ambas as direções, a aplicação do operador desaninhar (UNNEST) sobre uma coleção multiplica a sua cardinalidade pelo grau de compartilhamento testado. Por exemplo, a cardinalidade da coleção `Connections` é 297627. Na direção ascendente, por exemplo, cada instância desta coleção está relacionada a dez instâncias de `AtomicPart`, quando o grau de compartilhamento é 10. Portanto, o número de caminhos diferentes a serem percorridos é muito maior (2976270) do que no esquema original, onde a expressão de caminho nesta direção é monovalorada. Isto justifica o

melhor desempenho apresentado dos algoritmos na direção descendente em relação aos seus similares ascendentes.

5.2 Custo de CPU

Esta seção apresenta a análise do custo de CPU dos algoritmos baseados no operador n-ário e binário. A análise é baseada na função de custo proposta em BRAUMANDL *et al.* (2000). Pelos resultados apresentados na seção anterior, os algoritmos SN e PM foram os que se mostraram mais eficientes em relação ao custo de E/S. Na maioria dos testes realizados, o algoritmo PM obteve o melhor custo de E/S. Entretanto, em algumas situações, como mostram as tabelas 5.2 (1%) e 5.3 (16 e 32 Mbytes), o custo de E/S para PM, NP e SN é o mesmo. Nesses casos, o algoritmo NP, que tem reconhecidamente o menor custo de CPU, seria a escolha mais indicada. Além disso, foram verificadas outras situações onde o desempenho do SN se aproxima do desempenho do PM à medida que alguns fatores são variados (como o fator de seletividade e o grau de compartilhamento, por exemplo). Por isso, algumas questões sobre o custo dos algoritmos precisam ser respondidas. Quando o custo de E/S é o mesmo, qual dos algoritmos gasta mais CPU ? Ou ainda, se o custo de CPU do PM é maior que o custo do SN, o ganho provocado pelo custo de E/S compensa ?

Antes de comentar sobre os custos associados a cada algoritmo, é necessário apresentar os principais parâmetros e variáveis envolvidas nas fórmulas de custo. A tabela 5.5 mostra quais são estes parâmetros e fornece uma breve explicação sobre o que eles representam. A tabela 5.6 traz a notação e o significado das variáveis utilizadas no modelo de custo, enquanto a tabela 5.7 mostram os valores de algumas dessas variáveis para a consulta da figura 5.1.

Tabela 5.5 – Parâmetros utilizados no custo de CPU (BRAUMANDL *et al.*, 2000)

T_{hash}	Tempo para executar uma função *hash*
T_{cache}	Tempo para realizar a navegação entre dois objetos (tempo de acesso ao *cache*)
T_{copy}	Tempo para copiar um byte do/para o disco para a/da memória

Tabela 5.6 – Variáveis usadas no modelo de custo (BRAUMANDL *et al.*, 2000)

R	Coleção de partida no processamento da expressão de caminho (independente da direção)
P_i	Número de páginas da coleção C_i da expressão de caminho
$\|P_i\|$	Cardinalidade da coleção C_i da expressão de caminho
$\|RM_i\|$	Cardinalidade da partição RM_i
$\|RMS_i\|$	Cardinalidade da partição RMS_i
$Fan_{i,i+1}$	Grau de saída da coleção C_i para a coleção C_{i+1} (direção descendente)
$Fan_{i+1,i}$	Grau de saída da coleção C_{i+1} para a coleção C_i (direção ascendente – atributo inverso)
K	Número de fragmentos gerados pela aplicação da função *hash* h_S sobre uma coleção Ci desaninhada

Tabela 5.7 – Variáveis envolvidas na expressão de caminho da figura 5.1

	`c.parts-a.to-con(Length > 0)`
C_1	Coleção `CompositeParts`
C_2	Coleção `AtomicParts`
C_3	Coleção `Connections`
P_1	16
P_2	3421
P_3	9019
$\|P_1\|$	496
$\|P_2\|$	99209
$\|P_3\|$	297209
$Fan_{1,2}$	200
$Fan_{2,1}$	1
$Fan_{2,3}$	3
$Fan_{3,2}$	1

As variáveis mostradas na tabela 5.7 são utilizadas nas estimativas de custo dos algoritmos aqui avaliados. Com relação aos algoritmos PM e SN, suas fórmulas de custo são baseadas em algumas parcelas em comum. Estas parcelas são:

- Custo de cópia dos elementos de um conjunto ($Custo_{cópia}$): este fator refere-se ao tempo gasto pela CPU na cópia dos elementos de um conjunto da memória principal para o disco, ou vice-versa. Seu valor pode ser calculado a partir da expressão:

  $$Custo_{cópia} = |X| \,.\, x \,.\, T_{copy}, \qquad (1)$$

  onde x é o tamanho em bytes de um elemento do conjunto X.

- Custo de aplicação da função *hash* ($Custo_{hash}$): este fator refere-se ao tempo gasto pela CPU na aplicação de uma função *hash* sobre todos os elementos de uma coleção X. Seu valor pode ser calculado a partir da expressão:

  $$Custo_{hash} = | X | . T_{hash} \qquad (2)$$

- Custo de navegação entre duas coleções ($Custo_{cache}$): este fator refere-se ao tempo gasto pela CPU na navegação dos objetos da coleção Ci para os objetos da coleção C_{i+1} através do atributo de referência. Se o objeto relacionado não estiver no *cache* durante a navegação, o custo adicional de carga do objeto para a memória é computado pelas fórmulas de custo de E/S do mesmo modelo. Porém, nesta dissertação o custo de E/S foi avaliado somente na forma experimental. O valor desta parcela é dado pela expressão:

  $$Custo_{cache} = | P_i | . Fan_{i,i+1} . T_{cache} \quad \text{(direção descendente), ou}$$

  $$Custo_{cache} = | P_i | . Fan_{i+1,i} . T_{cache} \quad \text{(direção ascendente),} \qquad (3)$$

  onde i é o índice associado a i-ésima coleção (C_i) da expressão de caminho.

O custo de CPU do algoritmo NP é composto basicamente pela soma do custo de cópia da coleção de partida ($Custo_{cópia}$) mais o custo de navegação entre as coleções da expressão de caminho ($Custo_{cache}$). Para os algoritmos SN e PM, a expressão do custo de CPU é dada pela mesma fórmula do algoritmo NP acrescida do custo de manipulação das tabelas de suporte e das partições intermediárias RM_i e RMS_i, respectivamente. Portanto, observa-se que:

$$Custo_{NP} = Custo_{cópia} + Custo_{cache}$$

$$Custo_{SN} = Custo_{cópia} + Custo_{cache} + Custo_{hash(tab.\ suporte)}$$

$$Custo_{PM} = Custo_{cópia} + Custo_{cache} + Custo_{hash(partições\ RMi\ e\ RMSi)}$$

Assim, o custo de CPU do NP é sempre menor que os custos do PM e do SN. Resta saber se:

$$Custo_{SN} < Custo_{PM}$$

Essa comparação será função do benefício que as tabelas auxiliares puderem proporcionar. No caso do PM, as partições auxiliares não diminuem o número de objetos a serem navegados ($Custo_{cache}$), apenas otimizam o acesso aos dados em disco (número menor de operações de E/S). No caso do SN, essa tabela permite uma diminuição no número de objetos a serem percorridos ($Custo_{cache}$), o que proporciona de fato uma queda no tempo de CPU.

Quando não há compartilhamento entre as coleções, não há economia no $Custo_{cache}$ do algoritmo SN, já que as tabelas de suporte não evitarão o percurso de IDOs. Além disso, o $Custo_{hash(tab.\ suporte)}$ será maior que o $Custo_{hash(partições\ RMi\ e\ RMSi)}$, pois a tabela de caminhos válidos nunca vai evitar que a tabela de caminhos inválidos seja acessada. Entretanto, mesmo se o custo de CPU fosse menor, o que não é verdade, o algoritmo PM deveria ser escolhido, pois seu custo de E/S nas situações sem compartilhamento (resultados para a direção descendente da seção anterior) é bem melhor que o custo de SN.

Por outro lado, se houver compartilhamento o $Custo_{cache}$ do algoritmo SN pode ser bem menor que o $Custo_{cache}$ do algoritmo PM, dependendo do grau de compartilhamento, já que as tabelas de suporte efetivamente evitam o percurso de objetos já percorridos. Nesses casos, o $Custo_{hash}$ de ambos os algoritmos é aproximadamente o mesmo, já que a tabela de suporte de caminhos inválidos pode ser muito pouco acessada. Portanto, nessas condições, que são as mesmas verificadas nos cenários que mostram a equivalência dos algoritmos SN e PM no custo de E/S, o algoritmo a ser escolhido para o processamento da expressão de caminho é o SN, pois seu custo de E/S aproxima-se do PM, mas sua eficiência em relação ao custo de CPU fica claramente mostrada.

Para que fique mais claro o desempenho dos algoritmos SN e PM quanto ao custo de CPU, a tabela 5.8 apresenta o número total de solicitações atendidas pelo *cache* durante a execução dos algoritmos SN e PM. Esses valores correspondem aos acessos

que foram atendidos pelo *cache* sem a necessidade de E/S, ou seja, o número de *cache hits*. Estes valores não correspondem ao número de navegações realizadas, mas sim quantas vezes o *cache* foi acessado na recuperação de um objeto. Qualquer operação de leitura de um objeto cuja página esteja armazenada no *cache* é contabilizada na tabela 5.8. Porém, os resultados apresentados na tabela 5.8 dão uma idéia dos custos associados às parcelas $Custo_{cache}$ em cada um dos algoritmos.

Tabela 5.8 – Número de solicitações atendidas pelo *cache* (*cache hit*) na variação do fator de seletividade na direção ascendente

Fator de Selet. / Algoritmo	0.9	0.5	0.1
SN-Ascendente	340382	332730	299455
PM-Ascendente	507896	408905	310008

Note que o algoritmo SN apresentou uma taxa menor de "acertos no *cache*". Este fato foi responsável pelo maior custo de E/S do SN, como mostrado na tabela 5.1. Por outro lado, a parcela $Custo_{cache}$ para o algoritmo SN foi reduzida em relação a mesma parcela do algoritmo PM no cálculo do custo de CPU. A redução desta parcela é um dos fatores que justifica o menor custo de CPU apresentado pelo SN quando comparado ao PM. As tabelas 5.9 e 5.10 mostram os mesmos dados na variação do grau de participação dos objetos nos relacionamentos e na variação de memória do *cache*.

Tabela 5.9 – Número de solicitações atendidas pelo *cache* (*cache hit*) na variação do grau de participação na direção ascendente

Grau de Part. / Algoritmo	100%	50%	10%	1%
SN-Ascendente	340382	340131	339940	339908
PM-Ascendente	507896	451931	407926	289569

Tabela 5.10 – Número de solicitações atendidas pelo *cache* (*cache hit*) na variação do tamanho da memória

Tamanho do cache / Algoritmo	2Mb	4Mb	8Mb	16Mb	32Mb
SN-Ascendente	315260	340382	380709	384787	384788
PM-Ascendente	507896	507896	507896	507896	507896

Nesta seção procuramos mostrar intuitivamente como os algoritmos se comportam em relação a CPU. Apesar da informalidade em nossas análises, não podemos negar que o algoritmo SN é a melhor opção quando o grau de compartilhamento presente nas coleções é alto, pois seu desempenho de E/S se assemelha ao do algoritmo PM, mas seu custo de CPU é bem inferior. No anexo I desta dissertação, apresentamos uma demonstração mais detalhada da relação entre os custos de CPU dos algoritmos SN e PM. Essa análise é realizada sobre as parcelas de custo propostas em BRAUMANDL *et al.* (2000) e apresentadas nesta seção. Em particular, essa demonstração foi aplicada especificamente às condições de execução da consulta da figura 5.1 no cenário onde o desempenho de E/S do algoritmo SN melhora à medida que o grau de compartilhamento aumenta, ao contrário do PM.

5.3 Análise dos Resultados

Neste capítulo foram apresentados os resultados experimentais obtidos pela execução de expressões de caminho sobre diferentes configurações de base do *benchmark* OO7. Além disso, os custos de CPU dos algoritmos SN e PM foram comparados a partir do modelo de custo proposto em BRAUMANDL *et al.* (2000).

Basicamente, o algoritmo PM mostrou-se a melhor estratégia de execução quando a métrica de custo a ser considerada é o custo de E/S. Em todos os cenários avaliados, este algoritmo apresentou os melhores resultados, comprovando sua eficiência. Sua principal virtude é a seqüência de operações de fragmentação e agrupamento realizadas, cujo objetivo é determinar a melhor ordem de navegação de ponteiros para objetos armazenados em coleções relacionadas. Entretanto, quando o agrupamento físico dos objetos no disco já apresentar essa ordem de seqüência de execução dos relacionamentos, essas operações prévias (fragmentação e agrupamento) tornam-se uma sobrecarga que não se traduz em benefício. Esse agrupamento, freqüentemente realizado por administradores de banco de dados, é diretamente usufruído pelos algoritmos n-ários.

Embora nossos cenários não tenham utilizado o agrupamento de relacionamento, o algoritmo SN foi capaz de igualar o desempenho do algoritmo PM em alguns dos cenários avaliados. Em particular, quanto maior o percentual de objetos de uma coleção que não participa do relacionamento, maior também é o benefício causado pelo algoritmo SN. De uma forma geral, qualquer restrição aplicada sobre a expressão de

caminho, seja pela aplicação de fatores de seletividade sobre as coleções envolvidas, seja pela participação parcial dos objetos, contribui de modo significativo para o desempenho dos algoritmos baseados no operador n-ário. A redução no espaço de busca dos caminhos a serem percorridos compensa o prejuízo causado pela ordem aleatória no percurso dos objetos do caminho. Outro fator que indica um bom desempenho para o algoritmo SN é a quantidade de memória disponível para o *buffer* de páginas. Com o acréscimo de memória, o volume de páginas acessadas durante a execução das consultas contribui para que os próximos caminhos sejam beneficiados das páginas recentemente carregadas. Em algumas situações, como no percurso ascendente, o algoritmo NP também se beneficia do acréscimo de memória, tendo desempenho igual aos demais, porém superando os demais algoritmos no custo de CPU.

Na análise de custo de CPU dos algoritmo SN e PM, fica evidente a eficiência do operador n-ário. O foco do projeto do algoritmo PM foi otimizar o número de acessos a disco durante a junção entre duas coleções relacionadas. O preço pago por esta otimização foi a sobrecarga provocada pela manutenção dos resultados intermediários. Mesmo com algumas simplificações efetuadas sobre o algoritmo PM implementado (eliminação da primeira etapa de fragmentação e do agrupamento dos fragmentos relacionados, além do agrupamento final das partições RMS_i na montagem do resultado), o algoritmo SN ainda apresenta um melhor desempenho.

A partir dos resultados coletados nas seções 5.1 e 5.2, algumas heurísticas podem ser utilizadas na otimização do processamento de expressões de caminho. Primeiramente, a presença de um predicado de seleção sobre a coleção-raiz ou coleção-folha já é um bom indício de qual a direção em que a expressão de caminho deve ser percorrida. A escolha da direção também deve considerar o grau de compartilhamento de objetos, isto é, a direção que apresentar o maior compartilhamento dos objetos presentes nos diferentes caminhos percorridos deve ser a escolhida para o processamento da expressão de caminho.

Já na escolha do operador, a decisão depende da presença ou não do algoritmo binário PM. Se o processador de consultas contém apenas os algoritmos clássicos do modelo relacional (laços aninhados, ordenação-fusão e *hybrid hash*) e estes, por sua vez, encaram os atributos de referência como valores de chaves estrangeiras, o algoritmo SN é o mais indicado para a execução nos cenários avaliados na seção 5.1.

Se o algoritmo PM estiver disponível, sua escolha depende de alguns fatores. Se o ambiente de execução da consulta apresentar algumas das condições verificadas nos cenários, como por exemplo:

- Alta seletividade na expressão de caminho (baixos fatores de seletividade aplicados sobre uma ou mais coleções da expressão e caminho);
- Participação parcial dos objetos nos relacionamentos em baixas taxas;
- Tamanho da memória disponível para o *buffer* de páginas suficiente para que um número razoável de páginas das coleções permaneçam em memória principal;
- Alto grau de compartilhamento entre as coleções da expressão de caminho;

o algoritmo que deveria ser escolhido pelo otimizador de consultas é o SN. Nestas condições, o custo de E/S do algoritmo SN se iguala ao do PM, mas seu melhor desempenho sob o ponto de vista do custo de CPU justifica essa escolha. Em condições diferentes das apresentadas acima, o algoritmo PM mostra-se a melhor escolha para o processamento da expressão de caminho, já que seu custo de E/S é inferior ao custo de todos os demais algoritmos.

Vale a pena ressaltar que, em todas as configurações sugeridas para aplicação do algoritmo SN (baixo grau de participação dos objetos nos relacionamentos e grande quantidade de memória disponível), sua eficiência é baseada na premissa de que o sistema é capaz de armazenar as tabelas de suporte utilizadas. Se a memória não for suficiente para comportar essas estruturas de dados, o algoritmo PM é o mais indicado para a execução, pois é o que melhor se adapta às menores configurações de memória.

Capítulo 6. Conclusões

As fases de otimização e processamento de consultas sobre sistemas de banco de dados orientados a objeto (SGBDOOs) tornam-se mais complexas do que no modelo relacional devido às características intrínsecas deste modelo. Em especial, destaca-se a definição de objetos complexos, isto é, objetos que possuem um ou mais objetos como atributos. Nesses sistemas, as linguagens declarativas de consulta fornecem meios para a formulação de expressões de caminho em suas sintaxes.

Expressões de caminho permitem especificar uma navegação entre os objetos armazenados em diferentes coleções definidas no esquema. Existem dois operadores algébricos importantes no processamento de expressões de caminho: o operador binário e o n-ário. Os algoritmos representativos para o operador binário são baseados no operador de junção relacional. Esses algoritmos são adaptados para funcionarem com as referências (ponteiros) dos objetos armazenados. Já para o operador n-ário, o algoritmo mais utilizado para representar este operador é o algoritmo clássico *Naïve Pointer Chasing*.

Basicamente, a maioria das propostas de algoritmos para o processamento de expressões de caminho concentra-se nos algoritmos baseados no operador binário de junção. Muitos autores descartam o algoritmo *Naïve Pointer Chasing* por considerá-lo uma estratégia ingênua e ineficiente no acesso a disco dos objetos do caminho.

Nesta dissertação propomos uma otimização sobre o algoritmo *Naïve Pointer Chasing*. A otimização visa melhorar o desempenho de E/S ao evitar a execução de caminhos já percorridos. Este novo algoritmo, chamado de *Smart Naïve*, ainda mantém o baixo custo de CPU característico do *Naïve Pointer Chasing*. Para comprovar sua eficiência, foram realizados experimentos sobre diferentes configurações de base do *benchmark* OO7. Os resultados obtidos são comparados aos resultados do melhor algoritmo de junção proposto na literatura, o algoritmo P(PM)*M.

A partir da realização desta dissertação foi possível evidenciar alguns aspectos importantes no processamento de expressões de caminho. O primeiro deles é a comprovação da eficiência, sob o ponto de vista de custo de E/S, do algoritmo P(PM)*M sobre os demais algoritmos analisados (*Smart Naïve*, *Naïve Pointer Chasing* e junção *hash* por valor). Os resultados experimentais comprovaram que, no algoritmo P(PM)*M,

a ordenação das referências aos objetos através do endereço físico provoca um acesso agrupado que otimiza extremamente o acesso a disco. O acesso a uma coleção segundo seu atributo de referência ordenado é uma grande vantagem deste algoritmo sobre os demais, já que o acesso aos objetos agrupados beneficiam-se das páginas carregadas. Esta técnica evitou que uma página fosse recarregada após ter sido descartada da memória. A eficiência deste algoritmo foi comprovada em todos os cenários avaliados.

Ainda em relação ao custo de E/S, o algoritmo *Smart Naïve* obteve um desempenho equivalente ao algoritmo P(PM)*M no cenário correspondente à participação parcial dos objetos nos relacionamentos da expressão de caminho. Outro cenário em que os algoritmos *Smart Naïve* e P(PM)*M se igualaram foi a variação da memória de páginas disponível. A partir de dezesseis *megabytes* de memória disponível para o *cache* de páginas, estes algoritmos obtiveram o mesmo desempenho de E/S. Apesar do algoritmo P(PM)*M ter apresentado um desempenho superior nos diferentes fatores de seletividade, à medida que este valor é decrementado, o desempenho dos algoritmos *Smart Naïve* e P(PM)*M se aproximam.

Este fato é importante para futuras pesquisas em algoritmos de processamento de consultas, porque como já foi dito, a quantidade de memória disponível é um recurso computacional cada vez menos limitante nos sistemas atuais. Além disso, a pesquisa em técnicas de utilização e otimização do *cache* (AILAMAKI *et al.*, 1999, AILAMAKI *et al.*, 2001, BONCZ *et al.*, 1999) também contribui para que as futuras propostas de algoritmos beneficiem-se da memória disponível. Dessa forma, espera-se que a sobrecarga e a complexidade dos algoritmos diminua, pois muitos algoritmos propostos, como o P(PM)*M, por exemplo, são projetados para funcionarem com pouca memória e têm em seu foco principal a otimização do custo de E/S, pagando o preço na utilização de estruturas de dados que aumentam o custo de CPU.

Entretanto, embora as operações de E/S sejam muito mais caras que as de CPU, atualmente a gerência de *cache* dos sistemas operacionais é tão otimizada que o tempo predominante na maioria experimentos em banco de dados não está mais na E/S, mas sim na CPU (AILAMAKI *et al.*, 1999, AILAMAKI *et al.*, 2001, BONCZ *et al.*, 1999). Nesta dissertação tivemos que limitar o tamanho do *cache* para "simular" operações de E/S.

Com relação ao custo de CPU, a análise do modelo de custo proposto em BRAUMANDL *et al.* (2000) mostra que o algoritmo *Smart Naïve* mostrou-se mais

eficiente do que o algoritmo P(PM)*M conforme esperado. Algoritmos para o operador n-ário beneficiam-se da ausência de resultados intermediários no processamento de expressões de caminho. Esta foi a principal sobrecarga do algoritmo P(PM)*M em relação ao algoritmo SN. Além disso, devido a algumas questões de implementação, como a representação do IDO, por exemplo, foram realizadas algumas simplificações na proposta original do P(PM)*M que reduziram seu custo de CPU. Sem estas simplificações, este algoritmo teria uma sobrecarga no desempenho ainda maior.

O algoritmo junção *hash* por valor não obteve desempenho competitivo com seu similar. No caso do NP, embora seu desempenho com relação ao custo de CPU seja claramente superior aos demais algoritmos, seu custo de E/S é muito alto na maioria dos cenários, o que explica os péssimos resultados do *Naïve Pointer Chasing* em outros estudos e o fato de alguns artigos o descartarem (SHEKITA e CAREY, 1990, DEWITT *et al.*, 1993, BRAUMANDL *et al.*, 1998, BRAUMANDL *et al.*, 2000). Porém, as suas otimizações através do *Smart Naïve* mostraram sua competitividade junto ao P(PM)*M, sendo em alguns casos superior ao unir o custo de CPU e E/S.

Já o algoritmo de junção *hash* por valor não tira proveito dos ponteiros contidos nos atributos de referência. Isso obriga que todas as coleções que participam deste algoritmo sejam particionadas, aumentando não só o custo de E/S como também o custo de CPU. Este algoritmo não é sensível a participação dos objetos presentes nos relacionamentos, assim como a seletividade aplicada a alguma das coleções.

6.1 Contribuições desta tese

A principal contribuição deste trabalho é a proposta de um novo algoritmo de processamento de expressões de caminho. Este algoritmo é baseado no operador n-ário e teve sua eficiência analisada experimentalmente. Embora a construção de modelos de custo seja importante para a compreensão correta do problema e para avaliação sem muito esforço de diferentes cenários, a avaliação experimental é essencial para a validação das heurísticas e para a investigação de aspectos de implementação que causam impacto nos resultados.

Outra contribuição importante desta dissertação foi o estudo detalhado do processamento de expressões de caminho. Muitos trabalhos (SHEKITA e CAREY, 1990, DEWITT *et al.*, 1993, BRAUMANDL *et al.*, 1998) na literatura não analisam explicitamente expressões de caminho, assim como não tratam de todos os aspectos e

dimensões relevantes neste processamento, como o grau de participação dos objetos nos relacionamentos e o percurso em ambas as direções. Outra limitação dos trabalhos encontrados é que a grande maioria avalia somente expressões de caminho de tamanho dois. Esta característica inibe o potencial de algoritmos baseados no operador n-ário. Nesta dissertação procurou-se avaliar de modo abrangente as estratégias válidas e representativas para o processamento eficiente de expressões de caminho. Foram testadas experimentalmente as estratégias formadas pela combinação das dimensões de direção e operador, além da avaliação de diferentes tipos de expressões de caminho (monovaloradas e multivaloradas).

Finalmente, os resultados experimentais permitiram extrair novas heurísticas que podem ser incorporadas a um otimizador de consultas para um SGBDOO ou SGBDRO.

6.2 *Propostas para Trabalhos Futuros*

Os resultados obtidos neste trabalho levantam questões importantes a serem observadas em trabalhos futuros. Primeiramente, pretende-se avaliar experimentalmente o custo de CPU visando comprovar a validade da análise simulada realizada nesta dissertação. Além disso, é importante que novas configurações de base sejam testadas, não só em relação à cardinalidade das coleções participantes da expressão de caminho, mas também em relação a outros cenários que explorem o potencial do algoritmo *Smart Naïve*, como por exemplo, o agrupamento físico dos objetos em disco e o processamento de expressões de caminho cujo comprimento seja maior que três.

Outro aspecto que pretende-se abordar futuramente é a implementação de uma versão paralela dos algoritmos *Smart Naïve* e P(PM)*M. Verificou-se que a redução do conjunto de caminhos a serem percorridos afeta significativamente o desempenho dos algoritmos n-ários, como o *Smart Naïve*. A distribuição da base de objetos entre um conjunto de processadores, a partir das partições propostas no P(PM)*M, pode gerar um ganho de execução superior do algoritmo *Smart Naïve* quando comparado aos algoritmos binários.

Finalmente, a avaliação de desempenho da estratégia híbrida de processamento de expressões de caminho complementa a análise dos resultados obtidos nesta dissertação. A combinação dos algoritmos binários e n-ários avaliados neste trabalho formam diferentes estratégias híbridas. Seria interessante obter heurísticas que possam descobrir o melhor ponto de quebra da expressão de caminho, ou pelo menos indicar um ponto

favorável, e inferir sob que circunstâncias a utilização de um algoritmo híbrido é mais vantajosa que a utilização dos outros isoladamente, e vice-versa.

Referências Bibliográficas

ABITEBOUL, S., CLUET, S., CHRISTOPHIDES, V., MILO, T., MOERKOTTE, G., SIMEON, J., 1997, "Querying Documents in Object Databases", *International Journal on Digital Libraries*, v. 1, n. 1, pp. 5-19.

ABITEBOUL, S., 1997, "Querying Semi-Structured Data", In: *Proceedings of the 6th ICDT Conference*, Delphi, Grécia, pp. 1-18.

ABITEBOUL, S., QUASS, D., MCHUGH, J., WIDOM, J., WIENER, J., 1997, "The Lorel Query Language for Semistructured Data", *International Journal on Digital Libraries*, v. 1, n. 1, pp. 68-88.

AILAMAKI, A., DEWITT, D., HILL, M., WOOD, D., 1999, "DBMSs on a Modern Processor: Where Does Time Go ?", In: *Proceedings of the 25th VLDB Conference*, Edinburgo, Escócia, pp. 266-277.

AILAMAKI, A., DEWITT, D., HILL, M., SKOUNAKIS, M., 2001, "Weaving Relations for Cache Performance", In: *Proceedings of the 27th VLDB Conference*, Roma, Itália, pp. 169-180.

BERTINO, E., FOSCOLI, P., 1997, "On Modeling Cost Functions for Object-Oriented Databases", *IEEE Transactions on Knowledge and Data Engineering*, v. 9, n. 3, pp. 500-508.

BOAG, S., CHAMBERLIN, D., FERNANDEZ, M., FLORESCU, D., ROBIE, J., SIMÉON, J., STEFANESCU, M., 2001, *XQuery 1.0: An XML Query Language*, W3C Working Draft 20 December 2001. Disponível em: http://www.w3.org/TR/xquery/

BONCZ, P.A., MANEGOLD, S., KERSTEN, M.L., 1999, "Database Architecture Optimized for the New Bottleneck: Memory Access", In: *Proceedings of the 25th VLDB Conference*, Edinburgo, Escócia, pp. 54-65.

BRAUMANDL, R., CLAUSSEN, J., KEMPER, A., 1998, "Evaluating Functional Joins along Nested Reference Sets in Object-Relational and Object-Oriented Databases", In: *Proceedings of the 24th VLDB Conference*, Nova York, EUA, pp. 110-121.

BRAUMANDL, R., CLAUSSEN, J., KEMPER, A., KOSSMAN, D., 2000, "Functional Joins Processing", *The VLDB Journal*, v. 8, n. 3, pp. 156-177.

CAREY, M., DEWITT, D., NAUGHTON, J., 1993, "The 007 Benchmark", In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, v. 22, n. 2, Washington, EUA, pp.12-21

CATTELL, R., 1994, *Object Data Management*, Revised Edition, Addison-Wesley Publishing Company.

CATTELL, R., BARRY, D., BERLER, M., EASTMAN, J., JORDAN, D., RUSSEL, C., SCHADOW, O., STANIENDA, T., VELEZ, F., 2000, *The Object Data Standard: ODMG 3.0*, First ed. San Francisco, Morgan Kaufmann Publishers Inc.

CERI, S., FRATERNALI, P., PARABOSCHI, S., 2000, "XML: Current Developments and Future Challenges for Database Community", In: *Proceedings of the 7th EDBT Conference*, Konstanz, Alemanha, pp. 3-17.

CLUET, S., DELOBEL, C., 1992, "A General Framework for the Optimization of Object-Oriented Queries", In: *Proceedings of the 1992 ACM SIGMOD*, v. 21, n. 2, San Diego, EUA, pp. 383-391.

DAYAL, U., MITCHELL, G., ZDONIK, S., 1993, *Object-Oriented Query Optimization: What's the Problem ?*, Technical Report CS 91-4, Brown University, EUA.

DEUTSCH, A., FERNADEZ, M., FLORESCU, D., LEVY, A., MAIER, D., SUCIU, D., 1999, "Query XML Data", *Bulleting of the IEEE Computer Society Techical Committee on Data Engineering*, v. 22, n. 3, pp. 10-18.

DEWITT, D., LIEUWEN, D., MEHTA, M., 1993, "Pointer-based Join Techniques for Object-Oriented Databases", In: *Proceedings of the International IEEE Conference on Parallel and Distributed Information Systems (PDIS'93)*, California, EUA, pp. 172-181.

DEWITT, D. J., NAUGHTON, J. F., SHAFER, J. C, VENKATARAMAN, S., 1995, "Parallelizing OODMS traversals: a performance evaluation", *The VLDB Journal*, v. 5, n. 1, pp. 3-18.

EISENBERG, A., MELTON, J., 1999, "SQL:1999, formely known as SQL3", *ACM SIGMOD Record*, v. 28, n. 1, pp. 131-138.

FEGARAS, L., 1998, "*The OPTGEN Optimizer Generator*", Department of Computer Science and Engineering, The University of Texas at Arlington. Disponível em: http://ranger.uta.edu/~fegaras/optimizer

FEGARAS, L., ELMASRI, R., 2001, "Query Engines for Web Accessible XML Data", In: *Proceedings of the 27nd VLDB Conference*, Roma, Itália, pp.251-260.

GARDARIN, G., GRUSER, J., TANG, Z., 1996, "Cost-Based Selection of Path Expression Processing Algorithms in Object-Oriented Databases". In: *Proceedings of the 22nd VLDB Conference*, Bombay, India, pp. 390-401.

GRAHAN, M., 1983, "Path Expressions in Database", In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 366-378.

GEPPERT, A., DITTRICH, K., GOEBEL, V., SCHERRER, S., 1990, "*An Algebra for the NO² Data Model*", Relatório Técnico, Institut für Informatik, Universität Zürich, Alemanha.

IOANNIDIS, Y., KANG, Y., 1990, "Randomized Algorithms for Optimizing large Join Queries", In: *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, New Jersey, EUA, pp. 312-321.

KIM, W., 1995, *Modern Database Systems: The Object Model, Interoperability, and Beyond,* Addison-Wesley Publishing Company.

LEVINE, J., MASON, T., BROWN, D., 1992, *lex & yacc*, 2 ed., O'Reilly.

LU, H., SHAN, M.C., TAN, K.L., 1991, "Optimization of Multi-Way Join Queries for Parallel Execution", In: *Proceedings of the 17th VLDB Conference*, Barcelona, Espanha, pp. 549-560.

MANOLESCU, I., FLORESCU, D., KOSSMAN, D., XHUMARI, F., OLTEANU, D., 2000, "Agora: Living with XML and Relational", In: *Proceedings of the 26th VLDB Conference*, Cairo, Egito, pp. 623-626.

MATTOSO, M., 1993, "*Aspectos de Paralelismo na Gerência de Dados e Objetos no Geotaba*", Tese D. Sc., COPPE/UFRJ, Brasil.

MATTOSO, M., ZIMBRÃO, G., 1994, "Explorando o Processamento de Consultas no Servidor de Objetos GOA", In: *Anais do IX Simpósio Brasileiro de Banco de Dados*, São Carlos, Brasil, pp.98-103.

MATTOSO, M., WERNER, C., BRAGA, R., PINHEIRO, R., MURTA, L., ALMEIDA, V., COSTA, M., BEZERRA, E., SOARES, J., RUBERG, N., 2000, "Persistência de Componentes num Ambiente de Reuso", In: XIV Simpósio Brasileiro de Engenharia de Software, Caderno de Ferramentas, João Pessoa, Brasil.

MATTOSO, M., RUBERG, G., BAIÃO, F., VICTOR, A., 2001, "*Processamento de Consultas Orientado a Objetos*", Relário Técnico ES-547/01, COPPE/UFRJ.

MAURO, R. C., ZIMBRÃO, G., BRÜGGER, T. S., TAVARES, F. O., DURAN, M., LIMA, A. A. B., PIRES, P. F., BEZERRA, E., SOARES, J. A., BAIÃO, F. A., MATTOSO, M. L. Q., XEXEO, G., "GOA++: Tecnologia, Implementação e Extensões aos Serviços de Gerência de Objetos". In: *Anais do XII Simpósio Brasileiro de Banco de Dados*, pp. 272-277, Fortaleza, Brasil, Outubro 1997.

MISHRA, P., EICH, M., 1992, "Join Processing in Relational Databases", *ACM Computing Surveys*, v. 24, n. 1, pp. 63-113.

ÖZSU, M., BLAKELEY, J., 1995, "Query Processing in Object-Oriented Database Systems", In: *Modern Database Systems: The Object Model, Interoperability, and Beyond*, KIM, W. (ed.), Addison-Wesley Publishing Company, pp. 146-174.

ÖZSU, M., VORUNGATI, K., UNRAU, R., 1998, "An Asynchronous Avoidance-Based Cache Consistency Algorithm for Client Caching DBMSs", In: *Proceedings of the 24nd VLDB Conference*, Nova York, EUA, pp.440-451.

ÖZSU, M., VALDURIEZ, P., 1999, *Principles of Distributed Database Systems*, 2nd edition, New Jersey, Prentice-Hall.

RUBERG, G., 2001, "*Um Modelo de Custo para o Processamento de Consultas em Bases de Objetos Distribuídos*", Tese M. Sc., COPPE/UFRJ, Brasil.

SHANMUGASUNDARAM, J., TUFTE, K., HE, G., ZHANG, C., DEWITT, D., NAUGHTON, J., 1999, "Relational Databases for XML Documents: Limitations and Opportunities", In: *Proceedings of the 25th VLDB Conference*, Edinburgo, Escócia, pp. 302-314.

SHAW, G., ZDONIK, S., 1990, "A Query Algebra for Object-Oriented Databases", In: *Proceedings of the 6th International Conference on Data Engineering (ICDE'90)*, California, EUA, pp. 154-162.

SHEKITA, E., CAREY, M., 1990, "A Performance Evaluation of Pointer-Based Joins", In: *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, New Jersey, EUA, pp. 300-311.

SHEKITA, E., TAN, K.L., YOUNG, H., 1993, "Multi-Join Optimization for Symmetric Multiprocessors", In: *Proceedings of the 19th VLDB Conference*, Dublin, Irlanda, pp. 479-492.

STRAUBE, D., ÖZSU, M., 1990, "Queries and Query Processing in Object-Oriented Database Systems", *ACM Transactions on Database Systems*, v. 8, n. 4, pp. 387-430.

SU, S.Y.W., RANKA, S., He, X., 1999, *Performance Analysis of Parallel Query Processing Algorithms for Object-Oriented Databases*, Technical Report Database Systems Research and Development Center, Univ. of Florida, USA.

TANIAR, D., 1998, "Forward vs. Reverse Traversal in Path Expression Query Processing", In: *Proceedings of the 28th Technology of Object Oriented Languages and Systems (TOOLS)*, Melbourne, Australia, pp. 127-140.

TAVARES, F., 1999, "*Avaliação do Processamento Paralelo de Consulta no Modelo Orientado a Objetos*", Tese de M. Sc., COPPE/UFRJ, Brasil.

TAVARES, F., VICTOR, A., MATTOSO, M., 2000, "Parallel Processing Evaluation of Path Expressions". In: *Anais do XV Simpósio Brasileiro de Banco de Dados*, João Pessoa, Brasil, pp.720-726.

VALDURIEZ, P., 1987, "Join Indices", *ACM Transactions on Database Systems*, v. 2, n. 2, pp. 218-246.

VALDURIEZ, P., GARDARIN, G., 1984, "Join and Semijoin Algorithms for a Multiprocessor Database Machine", *ACM Transactions on Database Systems*, v.9, n.1, pp. 133-161.

VICTOR, A. O., RUBERG, G., 2000, *Um Otimizador de Consultas para o GOA++ através do OPTGEN*, Relatório da Disciplina **Laboratório de Banco de Dados**, COPPE/UFRJ, Rio de Janeiro, Brasil.

Anexo I – Análise de Custo dos algoritmos *Smart Naïve* e P(PM)*M

Esta seção anexa tem por objetivo mostrar a eficiência do algoritmo SN sobre o algoritmo PM nas operações de CPU realizadas durante a execução da figura 5.1. Em particular, nossa análise restringe-se à avaliação da direção ascendente de processamento de expressões de caminho sobre uma base cuja participação dos objetos nos relacionamentos é total. O percurso na direção ascendente se justifica pelo fato do algoritno SN ter apresentado os melhores desempenhos nesta direção (exceto no cenário da seção 5.1.2), devido ao maior compartilhamento dos objetos nos caminhos. Na direção descendente, o custo de E/S apresentado pelo algoritmo SN foi muito superior ao do PM, não justificando a sua escolha.

Poderíamos facilmente ter apresentado uma demonstração análoga para defender a eficiência do algoritmo SN nos outros cenários avaliados experimentalmente, como a participação parcial, por exemplo. Contudo, a principal meta da discussão em torno do custo de CPU dos algoritmos SN e PM foi mostrar que, embora a maioria dos autores descartem a sua utilização, algoritmos baseados no operador n-ário podem ser eficientes não só quanto ao custo de CPU, mas também em relação ao custo de E/S, como mostrado na seção 5.1.

Custo do algoritmo PM

O algoritmo PM implementado nesta dissertação sofreu algumas simplificações em relação a sua proposta original (BRAUMANDL *et al.*, 2000). Pela proposta original do algoritmo, existem duas etapas de fragmentação. A primeira etapa particiona a coleção de partida R em N fragmentos, onde o objetivo é traduzir os endereços lógicos em endereços físicos. Na segunda etapa, cada fragmento RM_i gerado na etapa anterior é particionado em K fragmentos, agrupando os objetos de R que apontam para objetos fisicamente próximos de S. Como a representação do IDO no GOA é física, a primeira etapa de fragmentação torna-se desnecessária. Por isso, apenas a segunda etapa de fragmentação, que aplica diretamente a função *hash* h_S sobre a coleção R, foi incorporada à versão do algoritmo PM implementado.

Outra simplificação realizada na versão do algoritmo PM implementada nesta dissertação ocorre na etapa final de agrupamento das partições RMS_i. Como a expressão

de caminho da consulta (figura 5.1) avaliada tem comprimento três, a aplicação do algoritmo sobre as duas primeiras coleções geram partições RMS_i que servem de entrada para a próxima execução do algoritmo. Na proposta original do algoritmo, após a geração das partições RMS_i, estas eram agrupadas para formar a coleção resultado. Porém, esta etapa pode ser descartada, já que a coleção resultado gerada na primeira execução seria novamente desaninhada e particionada, degradando o desempenho do algoritmo. A melhor estratégia é utilizar diretamente as partições RMS_i geradas no passo anterior na execução seguinte, ao invés de agrupá-las e fragmentá-las novamente na próxima execução do algoritmo. Dessa forma, o custo associado na montagem do resultado final da consulta foi descartado em nossas análises.

Considere a execução do algoritmo PM sobre duas coleções R e S. De acordo com a expressão apresentada na seção 5.2, o custo de CPU do algoritmo PM implementado nesta dissertação é ($Custo_{PM}(R,S)$):

$$Custo_{cópia(R)} + 2KCusto_{cópia(RM)} + KCusto_{cópia(RMS)} + Custo_{hash(R)} + Custo_{cache(RM)} \quad (4)$$

As parcelas $Custo_{cópia(RM)}$ e $Custo_{cópia(RMS)}$ representam o custo de cópia das partições da memória para o disco e vice-versa. A parcela $Custo_{cópia(RM)}$ é multiplicada por duas vezes o número de fragmentos (K) da coleção R no cálculo do $Custo_{PM}(R,S)$. Existem dois momentos distintos para a cópia dos fragmentos RM_i. Primeiramente são gravados em disco os fragmentos RM_i gerados pela aplicação da função *hash* h_S. Posteriormente o fragmento RM_i é carregado para a memória para realizar a junção por ponteiros. Portanto, para cada fragmento RM_i existem duas operações de cópia associadas. Uma da memória para o disco e outra do disco para a memória. Como existem K fragmentos, o valor para $Custo_{cópia(RM)}$ aparece multiplicado pelo fator 2K. No caso das partições RMS_i, somente uma operação de cópia (da memória para o disco) é realizada para cada partição. Como o número de partições RMS_i é igual ao número de partições RM_i, a parcela $Custo_{cópia(RMS)}$ é multiplicada pelo fator K.

Na expressão de caminho da figura 5.1, o custo do algoritmo PM na direção ascendente, por exemplo, é:

$$Custo_{PM}(\texttt{Connections}, \texttt{AtomicParts}) + Custo_{PM}(I,\texttt{CompositeParts}) \quad (5)$$

A parcela $Custo_{PM}$(I,CompositeParts) representa o custo de CPU do algoritmo PM sobre a coleção intermediária I e a coleção `CompositeParts`. A coleção intermediária I representa a coleção de objetos formada pelo conjunto dos objetos armazenados nas partições do resultado da execução anterior do algoritmo PM, isto é, a execução de PM sobre as coleções `Connections` e `AtomicParts`.

A expressão $Custo_{PM}$(`Connections`, `AtomicParts`) é calculada pela aplicação da fórmula definida em (4), isto é,

$$Custo_{cópia(Connections)} + 2KCusto_{cópia(RM)} + KCusto_{cópia(RMS)} + Custo_{hash(Connections)} + Custo_{cache(Connections)}, \text{ onde}$$

$$Custo_{cópia(Connections)} = 297627 \,.\, 61 \,.\, T_{copy}{}^{*}$$

$$Custo_{cópia(RM)} = |\,RM\,| \,.\, 16 \,.\, T_{copy} = (297627 / K) \,.\, 16 \,.\, T_{copy}{}^{\blacklozenge}$$

$$Custo_{cópia(RMS)} = |\,RMS\,| \,.\, 16 \,.\, T_{copy} = (297627 / K) \,.\, 16 \,.\, T_{copy}{}^{\blacktriangledown}$$

$$Custo_{hash(Connections)} = 297627 \,.\, Fan_{3,2} \,.\, T_{hash} = 297627\, T_{hash}$$

$$Custo_{cache(Connections)} = 297627 \,.\, Fan_{3,2} \,.\, T_{cache} = 297627\, T_{cache}$$

Portanto, o $Custo_{PM}$(`Connections`, `AtomicParts`) é dado por:

$$297627\,(61\, T_{copy} + 32\, T_{copy} + 16\, T_{copy} + T_{hash} + T_{cache}) =$$

$$297627\,(109\, T_{copy} + T_{hash} + T_{cache}) \qquad (6)$$

Já a parcela $Custo_{PM}$(I,`CompositeParts`) não é calculada pela fórmula definida em (4). Como as partições RMS_i geradas pelo algoritmo PM sobre as coleções C_3 e C_2 servem de entrada para a próxima junção, não é mais necessário computar o custo de cópia da coleção de partida. Então, o custo de $Custo_{PM}$(I,`CompositeParts`) é dado por:

$$2KCusto_{cópia(RM)} + KCusto_{cópia(RMS)} + Custo_{hash(I)} + Custo_{cache(I)}, \text{ onde}$$

---

* A tabela 4.1 mostra que uma instância de `Connections` ocupa 61 bytes no GOA

♦ Cada partição RM possui dois IDOs. Cada IDO no GOA ocupa 8 bytes

▼ O número de partições RMS é igual ao de partições RM

$$Custo_{cópia(RM)} = |RM| \,.\, 16 \,.\, T_{copy} = ((297627 / K) / K) \,.\, 16 \,.\, T_{copy}^{\spadesuit}$$

$$Custo_{cópia(RMS)} = |RMS| \,.\, 16 \,.\, T_{copy} = ((297627 / K) / K) \,.\, 16 \,.\, T_{copy}$$

$$Custo_{hash(I)} = K \,.\, ((297627 / K) / K) \,.\, Fan_{2,1} \,.\, T_{hash} = (297627 / K)\, T_{hash}^{\clubsuit}$$

$$Custo_{cache(I)} = K \,.\, ((297627 / K) / K) \,.\, Fan_{2,1} \,.\, T_{cache} = (297627 / K)\, T_{cache}$$

Assim, o $Custo_{PM}$(I, `CompositeParts`) é dado por:

$$297627 / K\ (32\ T_{copy} + 16\ T_{copy} + T_{hash} + T_{cache}) =$$

$$297627 / K\ (48\ T_{copy} + T_{hash} + T_{cache}) \qquad (7)$$

Finalmente, o custo de CPU do algoritmo PM para a expressão de caminho definida na consulta da figura 5.1 é, de acordo com a fórmula (5), dado pela soma de (6) e (7), ou seja,

$$297627\ (109\ T_{copy} + T_{hash} + T_{cache}) + 297627 / K\ (48\ T_{copy} + T_{hash} + T_{cache}) \qquad (8)$$

Custo do Algoritmo SN

A estimativa de custo do algoritmo SN também é realizada a partir das parcelas definidas na seção 5.2. Por tratar-se de um algoritmo para o operador n-ário, o SN apresenta algumas vantagens em relação ao algoritmo PM. A mais importante delas é a ausência de resultados intermediários no processamento da expressão de caminho. A criação e manutenção das partições RM_i e RMS_i acrescentam uma sobrecarga considerável ao custo de CPU do algoritmo PM. Esta sobrecarga é maior ainda se pensarmos em expressões de caminho mais longas.

Basicamente, o custo de CPU do algoritmo SN está diretamente relacionado ao custo de acesso às tabelas de suporte (caminhos válidos e inválidos). Além disso, é preciso considerar também o custo de CPU associado à transferência dos objetos da

---

♠ A partição RM na segunda execução do algoritmo contém o mesmo número de objetos (297627 / K) que a partição RMS da primeira junção. O número de objetos por partição, neste caso, é (297627 / K) / K

♣ A expressão ((297627 / K) / K) . $Fan_{2,1}$. T_{hash} representa o custo de aplicação da função *hash* sobre todos os objetos de uma partição da coleção I. Para todas as partições, multiplica-se a expressão por K

coleção de partida para a memória e à navegação dos objetos na expressão de caminho. O custo do algoritmo SN pode ser expresso por:

$$Custo_{SN} = Custo_{cópia(R)} + Custo_{hash(Válidos)} + Custo_{hash.(Inválidos)} + Custo_{cache}, \text{ onde} \quad (9)$$

$Custo_{cópia(R)}$ é o custo de cópia associado a todos os objetos da coleção de partida R

$Custo_{hash(Válidos)}$ é o custo associado à manipulação da tabela *hash* que armazena os IDOs percorridos em caminhos válidos da expressão de caminho

$Custo_{hash.(Inválidos)}$ é o custo associado à manipulação da tabela *hash* que armazena os IDOs percorridos em caminhos inválidos da expressão de caminho

$Custo_{cache}$ é o custo associado à navegação dos objetos da expressão de caminho

A análise de custo realizada para o algoritmo PM baseou-se no processamento da consulta da figura 5.1 no sentido ascendente. A fórmula definida em (8) expressa o custo de CPU do PM nesta direção. Para que a comparação seja possível, também faremos a análise de custo do SN nesta direção. Neste caso, a parcela $Custo_{cópia(R)}$ é idêntica à parcela já calculada anteriormente, ou seja,

$$Custo_{cópia(R)} = Custo_{cópia(Connections)} = 297627 \,.\, 61 \,.\, T_{copy}$$

Outra parcela facilmente calculada é $Custo_{hash(Válidos)}$. Na direção ascendente, a expressão de caminho da figura 5.1 é monovalorada. Portanto, todos os 297627 objetos da classe `Connections` formam um caminho a ser percorrido. Cada instância é a raiz de um caminho. Logo, o custo de $Custo_{hash(Válidos)}$ é dado por:

$$Custo_{hash(Válidos)} = 297627 \,.\, Fan_{3,2} \,.\, Fan_{2,1} \,.\, T_{hash} = 297627\ T_{hash}$$

Note que a cardinalidade de `Connections` é multiplicada pelo produto de $Fan_{3,2}$ e $Fan_{2,1}$. Isto porque o caminho de um objeto é navegado em toda a extensão da expressão de caminho, diferente dos algoritmos n-ários que navegam por pares de coleção.

Entretanto, não se pode determinar a priori os valores de $Custo_{hash(Inválidos)}$ e $Custo_{cache}$, pois eles são dependentes do resultado da busca ao IDO a ser percorrido na

tabela de suporte de caminhos válidos. Se o IDO procurado estiver armazenado na tabela, além do *cache*, a tabela de caminhos inválidos também não é acessada. Por outro lado, se o IDO não estiver armazenado na tabela de caminhos válidos, mas estiver na de caminhos inválidos, o acesso ao *cache* também é evitado. Dessa forma, para que nossa análise possa ser realizada, seja α o número de acessos à tabela de suporte de caminhos inválidos e β o número de acessos ao *cache* durante a navegação entre os objetos da expressão de caminho.

Apesar desses valores terem sido definidos como literais, note que eles são inferiores à $|C_3| \cdot \mathbf{Fan}_{3,2} \cdot \mathbf{Fan}_{2,1}$. Isto pode ser explicado pelo grau de compartilhamento entre os objetos presentes na direção ascendente, já que a tabela de suporte de caminhos válidos evita o percurso de outros caminhos contendo objetos compartilhados. Neste caso, a tabela de caminhos inválidos e o *cache* não são acessados. Pelo cálculo da expressão, α e β devem ser menores que 297627.

O custo do algoritmo SN, então, pode ser expresso como mostra fórmula definida em (9), por:

$$297627 \cdot 61 \cdot T_{copy} + 297627\ T_{hash} + \alpha T_{hash} + \beta T_{cache} \qquad (10)$$

Durante a apresentação do algoritmo SN no capítulo 3, foi comentado que o algoritmo SN mostra-se, intuitivamente, mais barato do ponto de vista do custo de CPU que o algoritmo PM, devido a sua simplicidade e a ausência de resultados intermediários. Este fato pode ser comprovado se as fórmulas (8) e (10) apresentarem a seguinte relação de desigualdade:

$$297627\ (109\ T_{copy} + T_{hash} + T_{cache}) + 297627 / K\ (48\ T_{copy} + T_{hash} + T_{cache}) \text{ é maior que}$$

$$297627\ (61 T_{copy} + T_{hash} + \alpha T_{hash} + \beta T_{cache})$$

Para a demonstração da validade dessa inequação, suponha por contradição que

$$297627\ (109\ T_{copy} + T_{hash} + T_{cache}) + 297627 / K\ (48\ T_{copy} + T_{hash} + T_{cache}) <$$

$$297627\ (61 T_{copy} + T_{hash} + \alpha T_{hash} + \beta T_{cache}) \qquad (11)$$

Após simplificação da inequação (11), a seguinte expressão é obtida:

$$48\,(K+1)\,T_{copy} + T_{cache}\,((K+1) - (K\,\beta\,/\,297627)) < T_{hash}\,((K\alpha\,/\,297627) - 1) \quad (12)$$

Considere, sem perda de generalidade, uma possível implementação do algoritmo PM que utiliza um conjunto mínimo de partições (K=1), isto é, todos os objetos da coleção de partida são armazenados em apenas uma partição. Esta versão simplificada transforma a fórmula (12) em:

$$96\,T_{copy} + T_{cache}\,(2 - (\beta\,/\,297627)) < T_{hash}\,((\alpha\,/\,297627) - 1) \quad (13)$$

Como $\beta < 297627$, podemos afirmar que

$$2 - (\beta / 297627) > 1$$

Em vista disso, o lado esquerdo da inequação (13) é positivo. Porém, a expressão do lado direito da inequação (13)

$$(\alpha\,/\,297627) - 1$$

é negativa, pois, como já foi dito, $\alpha < 297627$. Portanto, a inequação (13) não é válida, contrariando a suposição de que o custo do PM é menor que o do SN. Fica demostrada, então, a eficiência do custo de CPU do algoritmo SN sobre o algoritmo PM nas condições avaliadas nesta seção, ou seja, .

Dessa forma, a utilização do operador n-ário no processamento de expressões de caminho apresenta vantagens e desvantagens. Cabe ao otimizador escolher a melhor estratégia quando as condições forem favoráveis a sua utilização, como em alguns cenários avaliados nesta dissertação.