COORDENAÇÃO
DOS PROGRAMAS
DE PÓS-GRADUAÇÃO
DE ENGENHARIA
UNIVERSIDADE
FEDERAL DO
RIO DE JANEIRO

Processamento de Consultas Orientadas a Objetos

Marta Mattoso

Gabriela Ruberg

Fernanda Baião e

André Victor

ES-547/01 – Abril

Programa de Engenharia de Sistemas e Computação

Índice do Texto

1. Introdução ________ 3

1.1. Organização do texto ________ 4

2. Aspectos de Processamento de Consultas em Sistemas OO ________ 4

2.1. Tipos abstratos de dados (classes) ________ 5

2.2. Objetos Complexos ________ 6

2.3. Métodos e Encapsulamento ________ 7

2.4. Criação de Objetos ________ 8

2.5. Modelo de Armazenamento ________ 8

Coleções e Extensões ________ 8

Tipos de Identificadores de Objetos ________ 9

Agrupamento de Objetos ________ 9

Índices ________ 11

3. Álgebra OO ________ 11

3.1. Operadores Clássicos ________ 11

3.2. Operadores específicos do modelo OO ________ 12

4. Expressões de Caminho ________ 13

4.1. Representação de uma Expressão de Caminho ________ 13

4.2. Extensão de uma Expressão de Caminho ________ 16

5. Linguagem de Consulta OO ________ 18

5.1. Requisitos Básicos ________ 18

5.2. Estrutura Geral ________ 18

5.3. Considerações ________ 21

6. Metodologia de Processamento de Consultas OO ________ 22

6.1. Otimização Algébrica ________ 24

Espaço de busca e regras de transformação ________ 24

Algoritmo de busca ________ 25

Função de custo ________ 25

Extensibilidade do Otimizador de Consultas 25

6.2. Representação de Planos de Execução 26

Algoritmos de Consulta 26

6.3. Geração do Plano de Execução 30

7. Consultas em Ambientes de Paralelismo e Distribuição 31

7.1. Fragmentação de Coleções no Modelo OO 31

Corretude da fragmentação 33

Modelo Relacional 33

Modelo OO 34

7.2. Alocação de Classes no Modelo OO 36

7.3. Otimização Algébrica 37

Álgebra OO 37

Espaço de busca e regras de transformação 37

Algoritmo de busca 38

Função de custo 38

7.4. Geração do Plano de Execução 41

Referências 42

PROCESSAMENTO DE CONSULTAS ORIENTADAS A OBJETOS

Marta Mattoso Gabriela Ruberg André Victor Fernanda Baião

[marta,gruberg,aovictor,baiao]@cos.ufrj.br

Programa de Engenharia de Sistemas e Computação

COPPE – Universidade Federal do Rio de Janeiro

Caixa Postal 68511, Rio de Janeiro, RJ, Brasil, CEP 21945-970

1. INTRODUÇÃO

Uma das funcionalidades mais importantes de um Sistema Gerenciador de Bancos de Dados (SGBD) é a recuperação dos dados armazenados no disco. Neste contexto, o processador de consultas torna-se um dos principais módulos do SGBD, responsável por receber as solicitações dos usuários descritas em linguagem declarativa, analisá-las, transformá-las adequadamente, executá-las da forma mais otimizada possível, e retornar os dados que serão posteriormente apresentados ao usuário como resultado.

Grande parte da pesquisa realizada até hoje sobre processamento de consultas em SGBDs relacionais tem sido baseada no desenvolvimento de novos algoritmos para processar o operador de *junção*, reconhecidamente o mais custoso operador de avaliação para um processador de consulta (MISHRA e EICH, 1992, VALDURIEZ, 1997, IOANNIDIS e KANG, 1990, VALDURIEZ e GARDARIN, 1984, CAREY *et al.* 1990). Este alto custo deve-se principalmente à sua complexidade e freqüente utilização em consultas que expressam relacionamentos entre os dados armazenados.

Nos últimos anos, no entanto, 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 crescente utilização de Sistemas Gerenciadores de Bancos de Dados Orientados a Objetos (SGBDOOs) (CATTEL, 1994) e Relacionais Objeto (SGBDROs). 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.

O processamento de consultas em sistemas relacionais já é reconhecidamente um problema bastante complexo, e torna-se ainda mais no contexto OO devido principalmente às características intrínsecas deste modelo. Dentre estas características, podemos citar a existência de tipos abstratos de dados (classes),

métodos, encapsulamento, relacionamentos de referência e de herança, identificadores de objetos, entre outros. Um dos principais impactos para o processador de consultas OO é na forma de expressar e de processar relacionamentos entre objetos, que em SGBDOOs ou SGBDROs transformam-se em atributos complexos. Existem três tipos de atributos complexos (CATTEL, 1994): atributos de referência (que apontam diretamente para um único objeto), atributos de coleção (que apontam diretamente para dois ou mais objetos) e métodos (que referenciam indiretamente objetos de outras classes). Muito embora alguns dos algoritmos de junção já existentes no modelo relacional possam ser adaptados para funcionarem com esta nova abordagem dos relacionamentos OO, há na literatura propostas de novos algoritmos que são baseados em referências para processar esses relacionamentos, o que aumenta ainda mais o número de combinações possíveis de estratégias para o processamento, tornando ainda mais difícil e complexa a tarefa do processador de consultas de escolha da melhor estratégia (TAVARES *et al.,* 2000).

Neste contexto, a implementação de um processador de consultas eficiente, capaz de processar consultas declarativas em tempo aceitável, torna-se essencial para a consolidação dos sistemas OO. Outra dificuldade para a implementação de um processador de consultas OO é a carência de uma álgebra padrão, como a que existe para o modelo relacional. Iniciativas como a do *Object Data Group Management* (ODMG) (CATTELL *et al.*, 2000) de estabelecer padrões para o modelo OO, tanto a nível do modelo de dados quanto das linguagens de consulta, têm contribuído para minimizar este problema, embora ainda haja uma grande diversidade de modelos de dados e de especificações de linguagens de consultas propostas na literatura em uso nos sistemas existentes.

1.1. Organização do texto

O objetivo deste capítulo é fazer uma apresentação dos principais conceitos e considerações sobre o desempenho do processamento de consultas em SGBDs que utilizam o modelo orientado a objetos (SGBDOOs e SGBDROs). Na Seção 2 são detalhados alguns aspectos do modelo OO e sua influência sobre o processamento de consultas no SGBD. Na Seção 3 é apresentada uma álgebra de objetos, enquanto a Seção 4 define diversos conceitos e propõe uma representação formal para expressões de caminho no modelo OO, que são ilustradas na apresentação de uma linguagem de consultas na Seção 5. Uma metodologia de processamento de consultas é descrita na Seção 6, e finalizando o trabalho na Seção 7 são feitas considerações sobre desempenho e uso de paralelismo no processamento de consultas em banco de dados orientados a objetos.

2. ASPECTOS DE PROCESSAMENTO DE CONSULTAS EM SISTEMAS OO

Como dito na Seção anterior, muitas são as dificuldades encontradas no processamento de consultas em sistemas OO. Embora muitas das dificuldades sejam as mesmas das encontradas em sistemas relacionais, algumas delas são intrínsecas do modelo OO. Em especial, destacam-se a utilização de tipos abstratos de

dados (classes), objetos complexos e relacionamentos, métodos e encapsulamento, criação de novos objetos, e aspectos quanto ao armazenamento de coleções e extensões de classes, tipos de identificadores de objetos, agrupamento dos objetos no disco e utilização de índices como sendo alguns dos complicadores importantes no processamento de consultas para sistemas OO. Estes fatores são mais bem detalhados a seguir.

2.1. Tipos abstratos de dados (classes)

Linguagens de consultas relacionais operam sobre tipos previamente definidos de dados, denominados relações. Desta forma, o conjunto simples e bem definido dos operadores da álgebra relacional mantém o fechamento transitivo das operações[1]. Neste contexto, as relações operando dos operadores relacionais possuem tuplas de mesmo tipo, e o resultado das operações também produz tuplas de um só tipo previamente definido. Entretanto, o mesmo não pode ser dito para uma álgebra de objetos. As coleções operando no modelo OO podem conter objetos de tipos distintos, assim como os objetos da coleção resultante, em função da possibilidade de definição por parte do usuário de novos tipos abstratos de dados. Se a álgebra de objetos for fechada transitivamente, o conjunto resultante poderá servir de entrada para outras operações. Portanto, é necessário um sistema de tipos elaborado que permita a determinação de que métodos podem ser aplicados sobre todos os objetos de uma coleção. Porém, esta determinação é bastante complexa e pode inclusive não ser possível se o sistema possuir ligação dinâmica (*dynamic binding*) (DAYAL *et al.*, 1993), já que neste caso o tipo do objeto pode ser conhecido apenas em tempo de execução. Além disso, o sistema de tipos pode também afetar a aplicabilidade das regras de transformação algébrica na presença de checagem estática de tipos, já que no modelo de objetos há a possibilidade de definição de novos tipos abstratos de dados que não são previamente conhecidos pelo processador de consultas. Por exemplo, as regras de comutatividade ou associatividade para operações como união ou interseção podem não ser aplicáveis como são para o modelo relacional, como ilustrado no exemplo a seguir:

[1] Fechamento transitivo da álgebra significa que toda operação algébrica sobre um conjunto de operandos retorna resultados de mesmo tipo dos operandos (relações no modelo relacional) e os resultados podem ser alvo de outras operações algébricas que respeitam a mesma regra

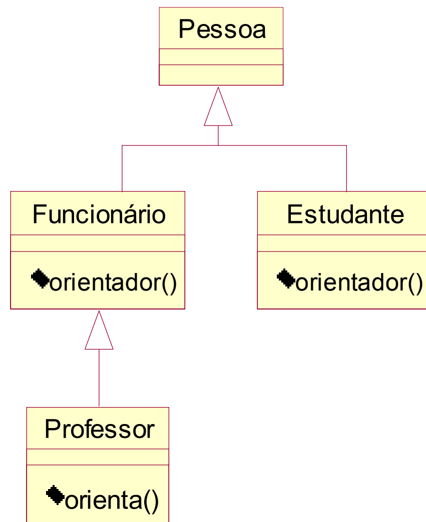


Figura 1: esquema de classes em notação UML

Suponha que as classes `Estudante` e `Funcionário` possuam um método `orientador()`. Para a classe `Estudante`, esse método retorna o relacionamento entre `Estudante` e `Professor`, ou seja, retorna um objeto da classe `Professor`. Para a classe `Funcionário`, esse método retorna um auto-relacionamento da classe `Funcionário`, o que significa dizer que a chamada deste método retorna um outro objeto da classe `Funcionário`. Portanto, tanto os objetos da classe `Estudante` quanto `Funcionário` possuem orientadores, porém estes são instâncias de objetos de tipos diferentes.

Sabemos que a regra da comutatividade entre o operador de união e os demais operadores algébricos é uma regra de transformação comum no modelo relacional. Porém, no modelo OO o mesmo pode não ser verdade. No exemplo acima, a expressão:

```
Union(Map(Estudantes e, e.orientador()), Map(Funcionários f, f.orientador()))
```

não pode ser substituída pela expressão a seguir,

```
Map(Union(Estudantes, Funcionários) p, p.orientador())
```

já que os objetos resultantes da união entre estudantes e funcionários são objetos do tipo `Pessoa`, onde o método `orientador()` não está definido.

2.2. Objetos Complexos

A estrutura complexa dos objetos no modelo OO também é outro fator de dificuldade no processamento de consultas. Diferente do modelo relacional, no qual as relações são formadas apenas por atributos de tipos

simples, o modelo de objetos permite que um objeto (denominado "objeto complexo") possa ter atributos de referência (para outros objetos). Ainda, os atributos de coleção existentes no modelo OO contribuem ainda mais para a dificuldade da tarefa. Por causa disso, as linguagens de manipulação (OQL para SGBDOOs e SQL:1999 para SGBDROs) 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 as linguagens que suportam a escrita de expressões de caminho fornecem um mecanismo uniforme para a formulação de consultas que envolvem diferentes características do modelo OO, incluindo relacionamentos binários, derivados e herança.

No contexto do processamento de consultas, vale a pena ressaltar que a existência de uma expressão de caminho pode sugerir uma ordem de execução que não necessariamente pode ser a mais eficiente. Uma expressão de caminho pode às vezes ser processada mais eficientemente usando-se junções (ao invés de percorrer diretamente as referências entre os objetos), ou mesmo sendo percorrida na ordem inversa à qual foi definida, percorrendo-se os atributos inversos (neste caso, porém, o padrão ODMG não determina que relacionamentos inversos sejam obrigatórios, o que pode caracterizar uma limitação para o processador de consultas). O processamento de consultas com expressões de caminho torna-se ainda mais complexo quando levamos em conta a presença de métodos e aspectos de desempenho. Isto porque os métodos podem ser implementados em outras linguagens de programação, o que torna extremamente difícil a tarefa do otimizador de consultas de identificar regras de transformação entre operadores e estimar custos das operações. Além disso, a maioria das técnicas de otimização de consultas já proposta na literatura até hoje não são diretamente aplicáveis pois não consideram as características novas do modelo OO em relação ao modelo relacional. Para contornar os problemas de desempenho no processamento de expressões de caminho, novos recursos como a definição de índices para expressões de caminhos são considerados por trabalhos na literatura (BERTINO 1990). A definição de índices para expressões de caminhos é um problema bastante complexo, endereçado em OGASAWARA e MATTOSO (1999).

2.3. Métodos e Encapsulamento

A determinação do custo de execução de uma consulta é complicada pela presença de métodos e encapsulamento. Otimizadores de consultas relacionais dependem do conhecimento das primitivas de armazenamento dos dados e estas estão disponíveis para o otimizador. No modelo OO, no entanto, o encapsulamento da representação física dos dados através de métodos dificulta a estimativa do custo de acesso aos atributos, já que métodos poderiam ser, teoricamente, escritos em qualquer linguagem de programação. Alguns sistemas contornam esta dificuldade ao permitir que o otimizador quebre o encapsulamento dos objetos, ou seja, o otimizador é encarado como uma aplicação especial que consegue ter acesso diretamente à representação interna do objeto, não sendo necessário o acesso via métodos de interface, violando dessa forma o princípio de encapsulamento oferecido pelo modelo de objetos (ÖZSU e BLAKELEY, 1995). Outra proposta encontrada na literatura é a de que os tipos "métodos" revelam seus custos como parte de sua própria interface. Em geral, informações sobre o tipo e representação física dos

objetos são necessárias para decidir a utilidade de uma regra de transformação algébrica normalmente utilizada pelos otimizadores baseados em regras. Por exemplo, uma heurística comum no modelo relacional é antecipar a execução das operações de seleção sobre as junções com o intuito de reduzir o tamanho das relações de entrada para a junção. A presença de métodos no predicado de seleção faz com que o custo de executar a operação de seleção seja baseado no custo de execução do método. Portanto, se o custo de execução do método for maior que o custo da junção, é mais adequado antecipar a junção, já que o método será executado sobre uma coleção menor.

2.4. Criação de Objetos

A identidade de objetos envolve decisões de modelagem e linguagem que podem afetar a otimização de consultas. Duas questões em especial devem ser discutidas pois podem ter diferentes impactos: a primeira é o quê constitui a igualdade entre dois objetos, que se reflete na linguagem de consulta onde operações de igualdade são usadas em predicados; a segunda é a decisão de se criar novos identificadores para os objetos retornados nas consultas ou usar os já existentes. A criação de novos objetos pode levar a novas definições de equivalência de objetos que afetam as regras de transformação algébricas disponíveis para um otimizador. Por exemplo, o resultado de uma consulta poderia ser uma nova coleção de objetos com identidades únicas e, portanto, mesmo duas respostas para a mesma consulta não seriam consideradas idênticas (DAYAL *et al.*, 1993).

2.5. Modelo de Armazenamento

O modo como os objetos são representados e armazenados no SGBDOO influencia decisivamente o desempenho do sistema durante a execução de uma consulta (GRUSER, 1996). Estas características representam o quão eficiente será a implementação das facilidades oferecidas pelo modelo OO no banco de dados.

Na maioria dos SGBDOOs, as definições sobre as classes e os esquemas são armazenadas separadamente dos objetos. Além disso, cada objeto pode ser armazenado independentemente da classe a qual pertença, geralmente em uma seqüência contígua de bytes. Além das características inerentes ao modelo de objetos implementado pelo SGBDOO (que incluem conceitos como tipos de dados, classes, atributos e relacionamentos), algumas características adicionais presentes no modelo de armazenamento dos objetos do SGBDOO podem exercer influência no processamento de consultas: a existência de coleções e extensões, tipos de IDOs, agrupamento dos objetos no disco e a existência de índices. A influência de cada um destes fatores é detalhada a seguir:

Coleções e Extensões

A extensão de um tipo é formada pelo conjunto de todos os objetos pertencentes a este tipo. Podemos também nos referir à extensão de uma classe, a qual contém todas as suas instâncias (BERTINO e FOSCOLI,

1997). Já uma coleção é sempre um subconjunto (possivelmente igual) da extensão de uma classe, pois reúne IDOs de objetos pertencentes a esta classe, a um atributo de referência/coleção (lista, vetor, conjunto, etc.) ou ao retorno de uma consulta (CATTEL, 1994). Na definição de uma classe pode ainda ser possível especificar a coleção que representará a extensão padrão dos objetos desta classe. Todavia, uma classe pode não possuir uma coleção associada e ainda assim ter uma extensão, ou seja, ter objetos na base. Este é o caso dos objetos armazenados através de um relacionamento de agregação (por exemplo, a classe `Capítulo` pode possuir objetos armazenados através do relacionamento com objetos da classe `Livro`). Além da coleção representando a extensão padrão, uma classe pode possuir coleções criadas e mantidas manualmente pelos usuários.

Tipos de Identificadores de Objetos

Os identificadores de objetos (IDOs) permitem ao SGBDOO identificar unicamente cada objeto armazenado. Este conceito é a base para a implementação de referências entre objetos e influencia diretamente o custo do processamento de consultas. Existem dois tipos básicos de IDOs: físicos e lógicos (BRAUMANDL et al., 1998). Na primeira abordagem, o banco de dados utiliza a composição de endereços físicos (segmento, página, posição no vetor, etc.) para permitir um acesso direto ao objeto, o que agiliza o armazenamento e a recuperação dos mesmos. Porém, IDOs físicos são restritivos quando é necessário que os objetos sejam movidos entre as páginas do SGBDOO.

Na abordagem de IDOs lógicos, o banco de dados gera um valor a partir de uma tabela de *hash*, ou de uma outra estrutura de índice, que corresponde à localização física do objeto. Com este tratamento o desempenho da manipulação dos objetos é degradado, devido à execução freqüente da função de mapeamento, porém é solucionado o problema da mobilidade interna dos objetos no SGBDOO (CATTELL, 1994).

Agrupamento de Objetos

O acesso aos dados em um SGBDOO é complexo devido à rica variedade disponível para construtores de tipos. Além das técnicas comuns para varredura da base, originárias dos sistemas relacionais, bancos de dados orientados a objetos também oferecem o acesso navegacional através dos relacionamentos entre os objetos (BENZAKEN, 1990). Considerando que, na prática, geralmente não é possível carregar o banco de dados inteiro na memória principal, realizar o percurso destas referências em disco pode ser uma operação de alto custo.

Uma estratégia de agrupamento visa minimizar as operações de entrada e saída (E/S) de dados armazenando os componentes de um objeto em uma mesma unidade de armazenamento. No melhor caso, quando um agrupamento cabe inteiramente em uma página do SGBDOO, a taxa de objetos solicitados não encontrados na memória (*object faults*) será mínima (BENZAKEN, 1990). O trabalho apresentado por

DEWITT *et al.* (1990) verifica que a existência de agrupamentos entre classes beneficia a arquitetura *servidor de páginas* para SGBDOOs.

Um agrupamento representa o armazenamento de objetos em áreas contíguas no banco de dados, segundo um determinado critério, de modo a minimizar o número de páginas acessadas e aumentar o desempenho das operações que acessam a base (SHRUFI, 1994). O tipo mais comum de agrupamento reúne objetos de uma mesma classe, também conhecido como agrupamento *por extensão*, padrão da maioria dos SGBDOOs.

Outro objetivo do agrupamento é a otimização do acesso a objetos com relacionamentos definidos entre si, também denominado agrupamento *por referência*. Neste sentido, objetos pertencentes a diferentes classes são armazenados juntos, ou seja, em áreas físicas próximas. Assim, é possível reduzir o número de acessos à páginas do SGBDOO na recuperação de um objeto com referências para outros objetos. Isto pode provocar um considerável ganho no desempenho do banco de dados.

Existem vários tipos possíveis de agrupamento por referência de objetos. O agrupamento simples armazena juntos os objetos pertencentes a duas classes relacionadas entre si. Caso os objetos de uma classe estejam armazenados próximo a objetos de outras duas ou mais classes, de acordo com relacionamentos de agregação ou associação da primeira para as demais, teremos um agrupamento conjuntivo. A Figura 2 ilustra os dois tipos de agrupamento por referência citados.

```
class Pessoa
 {
   List(Veiculo) carros;
   List(Imovel) imoveis;
   int idade;
   string nome;
   ...
 };

p1 = new Pessoa ( {c1,c2},{i1},    40, ”Jorge Silva” )
p2 = new Pessoa ( {c3},   {i2,i3}, 37, ”João Barros” )
```

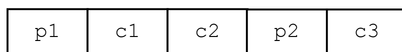


Figura 2.a - Agrupamento por referência simples da Classe `Pessoa` com `Veiculo`

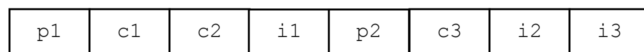


Figura 2.b - Agrupamento por referência conjuntivo da Classe `Pessoa` com `Veiculo` e `Imovel`

O problema de definir qual estratégia de agrupamento é mais adequada em um SGBDOO é semelhante ao problema da fragmentação de classes, pois ambos consistem em identificar grupos de objetos com padrões de acesso semelhantes. Entretanto, a definição de uma estratégia de agrupamento trata apenas de reunir fisicamente os objetos com forte grau de correlação. As metodologias para fragmentação de classes abordam também a separação eficiente dos grupos de objetos correlacionados (fragmentos), de modo que estes sejam distribuídos em diversos nós de processamento, visando explorar o possível paralelismo das operações

realizadas sobre a base. Este assunto é tratado em detalhes nos trabalhos de BAIÃO, MATTOSO e ZEVERUCHA (1998, 2000) e BAIÃO e MATTOSO (1998).

Índices

Índices são uma técnica vastamente explorada no ambiente relacional, e um mecanismo crucial em sistemas de bancos de dados, pois permitem a execução eficiente de consultas que retornam um pequeno subconjunto a partir de um grande volume de dados.

Existem diversas estruturas de índices propostas na literatura para bases de objetos. Índices sobre junções, advindos de técnicas relacionais, são utilizados para melhorar o desempenho da navegação entre referências de objetos. Multi-índices são baseados na alocação de um índice simples para cada classe percorrida em uma consulta. Já o índice aninhado tem o objetivo de reduzir o número de índices acessados durante a consulta, definindo-se um índice único para as diversas classes percorridas. Por fim, índices sobre caminhos apresentam grande importância para a avaliação de expressões de caminhos, pois provêem a ligação direta entre os objetos da classe inicial e as instâncias finais da navegação. Um exemplo de índices sobre caminhos são as relações de suporte ao acesso apresentadas em KEMPER e MOERKOTTE (1995).

3. ÁLGEBRA OO

Muita pesquisa vem sendo feita (CATTELL, 1994, ÖZSU e BLAKELEY, 1995, KIM, 1995) a fim de se 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, neste trabalho será adotada uma adaptação da álgebra proposta em BLAKELEY *et al.* (1993).

A álgebra de objetos que utilizamos define dois tipos de operadores. 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.

3.1. Operadores Clássicos

Seleção ($P_{\sigma[F]} < Q_1, Q_2, ..., Q_n >$). Tendo como argumento as n coleções de objetos $Q_1, Q_2, ..., Q_n$, a operação de seleção parametrizada tem como resultado uma coleção de objetos $p \in P$ tais que cada partição $<p,q_1,q_2,...,q_n>$ ($p \in P$, $q_1 \in Q_1$, $q_2 \in Q_2$, ..., $q_n \in Q_n$) satisfaça o predicado F.

Projeção ($P_{\Pi\beta}$) onde P é uma coleção e β um subconjunto dos atributos e/ou métodos definidos pelo tipo dos elementos de P. A coleção resultado contém os mesmos elementos de P, porém cada elemento só apresenta os atributos e métodos contidos em β.

Junção ($P_{\triangleright\triangleleft[F]}$< Q_1, Q_2, ..., Q_n >) onde $n \geq 1$. O operador de junção produz uma coleção de novos objetos criados a partir de cada permutação <q_1,q_2, ...,q_n > que satisfaça o predicado F.

Interseção ($P_1 \cap P_2$). A operação de interseção tem a semântica análoga à sua definição tradicional sobre conjuntos, isto é, retorna como resultado um conjunto de objetos que pertençam a P_1 e a P_2.

Union ($P_1 \cup P_2$). Também análoga à sua definição tradicional sobre conjuntos, a operação de união retorna como resultado um conjunto de objetos que pertençam a P_1 ou a P_2.

3.2. Operadores específicos do modelo OO

Unnest (Unnest P:p). Incorporado no padrão ODMG 3.0, o operador *unnest* é usado para manipular componentes que representam uma coleção de coleções de objetos. Neste caso, cada elemento p da coleção P representa também uma coleção de objetos. O operador *unnest,* quando aplicado a uma coleção P (que pode ser encarada como uma árvore de composição de objetos), vai de-referenciar recursivamente cada uma das coleções p pertencentes a P (onde cada p seria um dos nós filhos de P), e retorna como resultado objetos simples (isto é, não coleções) componentes de P (as folhas da árvore)

Materialize ou Mat (Mat p.a_1.a_2. … .a_n). Este operador também é conhecido como operador ponto (dot ou ".") presente nas linguagens de programação orientadas a objeto, e representa cada uma das ligações existentes em uma expressão de caminho (apresentada na Seção seguinte). O operador Mat indica de forma explícita a utilização de ponteiros (ou referências) entre objetos e, quando aplicado a um objeto p segundo uma determinada expressão de caminho p.a_1.a_2. … .a_n, de-referencia cada uma das referências a_1.a_2. … .a_n a fim de recuperar, em última instância, os objetos referenciados pelo atributo a_n segundo algum algoritmo de processamento. Ainda, todos os objetos acessados mesmo que por nós intermediários da expressão de caminho são mantidos prontos para uma possível utilização em operações subsequentes.

Segundo ÖZSU e BLAKELEY (1995), uma fase importante no processamento de consultas é a fase de otimização algébrica, apresentada na próxima Seção. 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.

Existem ainda outras propostas de álgebra (SHAW e ZDONIK, 1990, OPTGEN, 1998, CLUET e DELOBEL, 1994, GEPPERT *et al.*, 1990) encontradas na literatura, além desta apresentada aqui. Todas elas possuem operadores relativamente comuns, como os operadores de seleção, projeção, entre outros. Entre as que merecem destaque podemos citar a EQUAL (SHAW e ZDONIK, 1990), que inclui operações para manipular estruturas lógicas definidas no modelo ENCORE, e a álgebra que apoia o otimizador de consultas para a OQL gerado pela ferramenta geradora de otimizadores OPTGEN (OPTGEN, 1998).

4. EXPRESSÕES DE CAMINHO

Conforme já mencionado, 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 binários, derivados e herança. Existem na literatura algumas propostas de representação de expressões de caminho (BERTINO, 1997, GARDARIN *et al.*, 1996). 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 mais difícil para o usuário. Outro ponto é que tais representações não consideram projeções de atributos das classes.

Desta forma, esta Seção apresenta em detalhes todos os conceitos relativos à expressões de caminho no modelo de objetos e propõe uma nova representação formal que será utilizada no restante do trabalho. As principais características desta nova representação (que incluem vantagens em relação às representações existentes na literatura mencionadas anteriormente) são:

- define precisamente qual o atributo de referência é considerado entre cada par de classes do caminho de navegação, reduzindo ambigüidade. Ainda, no caso de atributos de coleção define-se um cursor para varrer a coleção referenciada;
- pode representar seleções em cada predicado aninhado sobre as classes no caminho de navegação.
- pode representar projeções de atributos de uma classe

Para esclarecer alguns conceitos desta nova representação, são fornecidos pequenos exemplos utilizando-se a linguagem de consultas OQL do padrão ODMG, que é apresentada na Seção seguinte.

4.1. Representação de uma Expressão de Caminho

Seja $\{C_1, C_2, \cdots, C_\ell\}$ um conjunto de coleções, não necessariamente distintas entre si, cujas respectivas classes $\{T_1, T_2, \ldots, T_\ell\}$ 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 1 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:

$\underline{\mathrm{P} : X_1[(p_1)].A_1[-X_2][(p_2)].A_2[-X_3][(p_3)]\cdots .A_{\ell-1}[-X_\ell][(p_\ell)]}$, 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 ou projeta seus atributos. Se p_i é nulo, todos os objetos de C'_i são qualificados para a expressão de caminho. Predicados aninhados são mais bem descritos ainda nesta Seção.

A representação de expressão de caminho proposta neste trabalho fornece uma forma uniforme e não ambígua de descrever qualquer caminho de navegação entre classes relacionadas no esquema.

A seguir são mostrados alguns exemplos de consultas OQL expressas sobre o esquema do benchmark OO7 (CAREY *et al.*, 1993)

<u>*Exemplos:*</u>

1. 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 $\ell = 2$, a classe-raiz é `CompositeParts`, a classe-folha é `AtomicParts` e a sua representação é a seguinte:

```
P1: c.parts-a(x < 100000)
```

2. A consulta abaixo possui uma expressão de caminho de comprimento $\ell = 1$.

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

A expressão correspondente será:

```
P2: d(date > 10/01/1910)
```

3. A seguinte consulta 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 2, 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 3, 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 and c.document.date < 10/01/1940`" ou apenas "`date`"). Em particular, uma expressão da forma `<atributo>` representa a projeção do atributo no resultado da consulta (conforme o exemplo 6, predicado P7).

Um predicado aninhado pode conter atributos simples, complexos, ou uma combinação de atributos simples e complexos. A existência de um atributo complexo em um predicado aninhado indica apenas a recuperação dos IDOs dos objetos relacionados, sem que haja entretanto a navegação pelos ponteiros correspondentes. Neste caso, os ponteiros contidos nos atributos complexos (os quais representam IDOs de objetos relacionados) são recuperados e podem participar de comparações, por exemplo, mas os objetos apontados pelos mesmos não são necessariamente acessados. A navegação não irá ocorrer, por exemplo, quando o predicado aninhado contendo o atributo complexo for referente à coleção-alvo .

Exemplos:

5. A expressão de caminho contida na consulta abaixo não contém predicados aninhados especificados. Ou seja, todos os objetos pertencentes ao caminho serão recuperados.

```
select a
from c in CompositeParts, a in c.parts
```

Representada por:

```
P6: c.parts-a
```

6. Na consulta abaixo, é recuperada uma coleção formada pelos ponteiros (IDOs) contidos no atributo complexo `document`. Entretanto, os objetos apontados por este atributo não são acessados.

```
select c.document
from c in CompositeParts, a in c.parts
where a.x < 900000
```

A expressão contida nesta consulta é representada a seguir. Observe o predicado aninhado p_1, onde o atributo complexo `document` está presente indicando sua projeção para o resultado da consulta:

```
P7: c(document).parts-a(x < 900000)
```

7. A expressão de caminho P1 do exemplo 1 anterior possui o seguinte predicado aninhado:

   ```
   p2 = x < 100000
   ```

8. O resultado da expressão de caminho a seguir é formado pela projeção dos atributos `title` e `date`, onde esta projeção é representada no predicado aninhado p_2.

   ```
   select c.document.title, c.document.date
   from c in CompositeParts
   ```

   Representada por:

   ```
   P8: c.document(title ∧ date)
   ```

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 de coleção na expressão de caminho, esta é dita **multivalorada**. Consideramos que os atributos multivalorados possuem domínio restrito a apenas uma classe.

Exemplos:

9. A consulta abaixo possui uma expressão de caminho monovalorada de comprimento $\ell = 2$.

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

   Representada por:

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

10. A expressão de caminho P1 do exemplo 1 é multivalorada.

    ```
    P1: c.parts-a(x < 100000)
    ```

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 algoritmos podem ser aplicados a cada subexpressão da mesma. Isto implica em um acréscimo considerável de complexidade no processo de otimização deste tipo de consulta.

4.2. Extensão de uma Expressão de Caminho

Uma expressão de caminho representa um conjunto de tuplas da forma $\{O_1.O_2.\cdots.O_\ell\}$, onde cada tupla contém objetos pertencentes ao caminho que satisfazem os predicados aninhados da expressão (GARDARIN *et al.*, 1996). Embora o objetivo de uma expressão de caminho seja recuperar apenas os objetos que satisfazem todo o percurso especificado, a existência de relacionamentos opcionais na base de objetos permite que caminhos incompletos sejam varridos durante a materialização da expressão.

Instâncias parciais de uma expressão de caminho ocorrem quando existem relacionamentos opcionais entre as classes da base de objetos, ou seja, quando é possível a ocorrência de nulos nos atributos que

compõem a expressão de caminho. Podemos identificar os seguintes tipos de extensão para uma dada expressão de caminho (KEMPER e MOERKOTTE, 1995):

1. Extensão canônica, que contém apenas informações sobre caminhos completos, ou seja, caminhos sempre originados em X_1 e sempre terminados em X_ℓ;
2. Extensão completa à esquerda, que contém todos os caminhos originados em X_1 que não necessariamente terminam em X_ℓ, mas que podem terminar em um valor nulo;
3. Extensão completa à direita, definida analogamente à anterior, contém os caminhos terminados em X_ℓ, mas possivelmente originados em algum objeto de X_i que não é referenciado por qualquer objeto do X_{i-1} através do atributo A_{i-1};
4. Extensão completa, que contém todos os caminhos parciais, mesmo que não seja originado em X_1 ou termine em um valor nulo.

O padrão ODMG 3.0 (CATTEL *et al.*, 2000) determina que, quando da existência de relacionamentos opcionais, o acesso a uma propriedade (atributo) de um objeto nulo retornará o valor especial **`UNDEFINED`**. O relacionamento opcional possui impacto em alguns fatores importantes da base de objetos. Dentre eles, destacamos:

- Na definição da fragmentação horizontal derivada de uma classe deve existir um fragmento cuja cláusula de seleção seja definida pelo operador especial **ELSE**, onde serão armazenados todos os objetos que não estão relacionados à classe primária;
- Caso haja agrupamento físico por referência entre as coleções *A* e *B*, existirão três tipos de partições físicas para armazenar os objetos correspondentes: Cl_A, Cl_B e $Cl_{A \to B}$. A partição Cl_A reúne apenas os objetos de C_A que não possuem referências para objetos de C_B. Cl_B armazena apenas objetos de C_B que não são referenciados por objetos de C_A. Por fim, a partição $Cl_{A \to B}$ reúne objetos pertencentes à C_A e à C_B, armazenados juntos, conforme o relacionamento;
- O fator de seletividade de uma expressão de caminho deverá refletir a participação parcial das coleções correspondentes, visto que a cardinalidade de C'_i poderá ser inferior à cardinalidade de C_i.

5. LINGUAGEM DE CONSULTA OO

5.1. Requisitos Básicos

Uma linguagem de consulta é um dos principais componentes de um SGBD. No paradigma puro da orientação a objetos, a linguagem de consulta deve atender aos seguintes requisitos:

1. Ser integrada com as linguagens de programação
   - mesmo sistema de tipos
   - verificação de tipos incluindo a linguagem de programação e a interface do BD
   - consulta sobre objetos persistentes, temporários, distribuídos ou objetos com versões
2. Trabalhar com o modelo de dados orientado a objetos
   - IDO, encapsulamento, herança, relacionamentos complexos, polimorfismo, ...
3. Ser declarativa

No entanto, até recentemente, não havia consenso na literatura sobre um padrão para o modelo de objetos ou para uma linguagem de consulta orientada a objetos, fato que comprometia a consolidação dos SGBDOOs. Visando minimizar este problema, esforços foram realizados por membros do ODMG (*Object Data Management Group*) no sentido de definir e especificar um padrão formal a ser adotado pelos trabalhos da área (CATTELL *et al.*, 2000).

Dentre as especificações deste padrão, a linguagem OQL (*Object Query Language*) é uma linguagem no estilo da SQL que permite a manipulação de objetos segundo o modelo de objetos do ODMG e atende aos requisitos acima. Entre seus principais objetivos, estão a manipulação de objetos sem privilegiar as operações com conjuntos e uma maior liberdade em relação à tradicional cláusula *select-from-where*. A OQL consiste em uma linguagem de consulta sobre objetos de fácil compreensão e uso, não sendo computacionalmente completa.

5.2. Estrutura Geral

A estrutura geral de uma consulta em OQL é a seguinte:

```
select resultado
from operando
[where predicado]
```

onde:

– **`resultado`** representa uma das variáveis (ou combinação) presentes em operando e identifica a origem da lista de objetos/valores que será fornecida como resultado da consulta.

– **`operando`** da consulta consiste de expressões do tipo

```
v in coleção
```

sendo v uma variável que representa os objetos em coleção.

– **`predicado`** é o conjunto de cláusulas que representam as condições que devem ser satisfeitas pelas variáveis de lista de variáveis a fim de servirem como resultado da consulta.

Tomemos como exemplo a seguinte definição de classes (sintaxe C++):

```
Class Pessoa
 {
   Empresa empresa;
   List<Pessoa> dependentes;
   int idade;
   string nome;
   ...

 };
Set<Ref<Pessoa>> Pessoas;
```

```
class Empresa
{
   string nome;
   real fatur;
   Produto prod;
   ...
};
```

```
class Produto
{
  string nome;
  Set<Empresa> fabric;
  ...
};

Set<Ref<Produto>> Prods;
```

De acordo com as definições acima, a seguinte consulta é possível:

```
select p
from p in Pessoas
where p.idade > 50
```

Esta consulta tem como resultado a lista de todos as pessoas com mais de 50 anos.

A estrutura básica de uma consulta OQL (*select-from-where*) não restringe que operadores da OQL possam ser utilizados independentemente desta estrutura. Seja o exemplo a seguir, baseado no *benchmark* 007, onde são recuperados todas as conexões cujo comprimento seja inferior a 500000:

```
select c
from c in Connections
where c.length < 500000
```

A consulta seguinte recupera o conjunto de todas as peças atômicas da peça composta `BracoMecanico`:

```
BracoMecanico.parts
```

Embora a segunda consulta não utilize a expressão *select-from-where*, esta é válida em OQL. Observe, entretanto, que neste caso é essencial especificar o nome do ponto de entrada da consulta.

Resultado de uma consulta. De acordo com a definição do padrão ODMG para a linguagem OQL o resultado de uma consulta OQL é formado por objetos com ou sem identidade, onde alguns objetos são gerados pelo processador de consultas e outros são recuperados a partir da base de dados. Ainda, o resultado

de uma consulta é aberto, podendo ter qualquer combinação de tipos. Uma das grandes diferenças entre a SQL e a OQL é a possibilidade da OQL lidar com valores complexos. OQL pode criar qualquer valor complexo como seu resultado final, ou intermediário dentro de uma consulta aninhada.

Para construir valores complexos, a OQL utiliza construtores como *struct*, *set*, *bag*, *list* e *array* definidos a partir das variáveis definidas no operando da consulta. Um exemplo de consulta que mostra o nome de produtos e o nome de seus fabricantes cujo faturamento é inferior a 1000:

```
select   struct (prod: p.nome, fabr: e.nome)
from     p in Prods,
         e in p.fabric
where    e.fatur < 1000
```

O resultado da consulta pode ser também uma coleção de objetos. Um exemplo de consulta que retorna uma coleção de objetos da classe Produto que possui fabricantes cujo faturamento é inferior a 1000:

```
select   p
from     p in Prods,
         e in p.fabric
where    e.fatur < 1000
```

Operando de uma consulta. Conforme já observado nos exemplos anteriores, o operando da consulta consiste de expressões do tipo :

```
v in coleção
```

sendo `v` uma variável que representa os objetos em `coleção`. `Coleção` pode ser a extensão de uma classe (ou uma coleção com nome), o atributo de um objeto, um caminho de navegação ou ainda o resultado de uma outra consulta. Por exemplo, a consulta abaixo tem como resultado a lista de todos os dependentes de pessoas com mais de 50 anos.

```
select   d
from     p in (select e from e in Pessoas where e.idade > 50),
         d in p.dependentes
```

Predicado de uma consulta. *Predicado* é o conjunto de cláusulas que representam as condições que devem ser satisfeitas pelas variáveis das listas de *operando* a fim de servirem como resultado da consulta. Mais uma vez, nestas cláusulas podem ser utilizados caminhos e atributos de objetos como, por exemplo:

```
select   p
from     p in Pessoas
where    p.empresa.prod.nome = "Avião a Jato"
```

Este predicado seleciona empregados de empresas que produzem aviões a jato.

As cláusulas dos predicados podem contar com operadores especiais para trabalhar com coleções operando, como por exemplo `for all` e `exists`.

```
for all/exists x in coleção: predicado
```

O operador `for all` representa o quantificador universal. Por exemplo, no predicado abaixo é retornado o valor *verdadeiro* se todos os dependentes tiverem idade superior a 16 anos.

```
for all d in dependentes: d.idade > 16
```

O operador `exists` representa o quantificador existencial. Por exemplo, no predicado abaixo é retornado o valor *verdadeiro* se alguma pessoa trabalha na empresa ACME.

```
exists p in Pessoas: p.empresa.nome = "ACME"
```

As cláusulas dos predicados também podem conter chamadas de métodos, inclusive no meio de um caminho de navegação.

Operadores Especiais. O modelo de consultas OQL conta também com os seguintes operadores especiais:

– *Ordenação*

```
select  p
from    p in Prods
        e in p.fabric
where   e.fatur < 1000
order by
        p.nome
```

– *Agregação*

```
max ( select p.idade from p in Pessoas )

select  p
from    p in Pessoas
group by      jovem: p.idade <= 18,
              meia_idade: p.idade > 18 and p.idade < 65
              idoso: p.idade >= 65
```

– *Conjunto*

```
Intersect, union, except
```

– *Conversão*

```
element(singleton), flatten(collection_of_collection)
```

5.3. Considerações

Baseada no SQL-92, a linguagem OQL oferece suporte a objetos complexos, ao tratamento de identidade de objetos, expressões de caminho, polimorfismo, invocação de métodos, dentre outras funcionalidades necessárias à manipulação de dados em uma base de objetos, como ilustrado nos exemplos anteriores. Além disso, as primitivas disponíveis na OQL abrangem não só o tratamento de conjuntos, mas também de listas, estruturas, vetores, etc, com a mesma eficiência.

O ponto de entrada de uma consulta OQL consiste em um objeto nomeado ou em uma coleção de objetos nomeada. Isto significa que uma consulta OQL precisa de uma referência conhecida na base de objetos, a

partir da qual a consulta será executada. Selecionar objetos armazenados na base corresponde, basicamente, à recuperação de seus IDOs.

A linguagem de consulta OQL suporta tanto objetos, os quais possuem um IDO associado, como literais (objetos simples), cuja identidade é representada por seus valores. Uma expressão OQL pode retornar um literal, um objeto, ou ainda uma coleção de literais ou de objetos. Quanto à identidade de objetos em consultas OQL, é possível criar novos objetos a partir do resultado de uma consulta, ou selecionar objetos pré-existentes na base. Para criar um novo objeto, um construtor de tipo disponível no SGBDOO deve ser utilizado (conjunto, lista, estrutura, etc).

Uma das vantagens da linguagem OQL ser baseada no mesmo sistema de tipos do padrão ODMG é que suas construções poderão ser invocadas a partir de uma linguagem de programação orientada a objeto, como *Java* ou C++. De maneira recíproca, esta característica permite que operações definidas nestas linguagens possam ser chamadas a partir de uma consulta OQL.

Em uma consulta escrita em OQL, é possível percorrer ponteiros (IDOs) referentes aos relacionamentos entre os objetos complexos, "navegando" entre estes relacionamentos. Isto implica em uma maneira diferente de processar a consulta, com acesso direto aos objetos através dos relacionamentos. As expressões de consulta correspondentes a essa navegação entre ponteiros são chamadas de **expressões de caminho,** apresentadas anteriormente.

Além da navegação por expressões de caminho, coleções que não estão relacionadas explicitamente entre si podem ser declaradas para a realização de junções. Neste caso, o processamento é realizado de maneira análoga ao processamento do SQL padrão.

Não existe na OQL uma sintaxe equivalente aos comandos `insert`, `delete` e `update` do SQL. Isto ocorre porque a linguagem OQL considera que a propriedade de encapsulamento deve ser respeitada, onde todas as alterações sobre os objetos de uma classe devem ser realizadas através de sua interface (métodos). A chamada a métodos em uma consulta OQL ocorre de maneira transparente.

6. METODOLOGIA DE PROCESSAMENTO DE CONSULTAS OO

Consultas em um SGBD são expressas em linguagens declarativas que têm como um de seus objetivos minimizar o conhecimento requerido do usuário sobre aspectos como: implementação interna da estrutura do objeto, existência de índices sobre atributos, e estratégias de processamento e otimização existentes. O processamento de uma consulta, desde a sua submissão até o retorno do resultado, inclui diversos passos descritos a seguir.

Otimização do Cálculo. Primeiramente, a expressão da consulta submetida é normalizada para eliminar predicados redundantes ou duplicados.

Transformação Algébrica. A expressão normalizada é então convertida para uma representação equivalente na álgebra de objetos. Existem diversas formas de representação interna de uma consulta, sendo a árvore de operadores a mais utilizada na literatura. Neste caso, a consulta pode ser visualizada como uma árvore cujas folhas são extensões de classe ou coleções de objetos, e os nós internos são operadores, cujos operandos são seus nós filhos, que retornam outras coleções de objetos. Uma vez construída a árvore de representação de uma consulta, o seu processamento acontece a partir dos nós folha em direção à raiz da árvore.

Outra forma de representação interna de consultas, proposta por SU *et al*. (1998), utiliza grafos ao invés de árvores para representar a consulta e o banco de dados nos níveis intensional e extensional. No nível intensional, um banco de dados é definido por uma coleção de classes inter-relacionadas na forma de um grafo de esquema (s*chema graph*). Cada vértice do grafo de esquema corresponde a uma classe do esquema e cada aresta corresponde a um relacionamento (associação, agregação ou herança) entre os dois vértices (classes) do grafo (esquema). No nível extensional, o banco de dados pode ser visto como uma rede de objetos de diferentes classes ligados pelos seus relacionamentos. Isto pode ser representado por um grafo de objetos (*object graph*). Uma consulta OO é então especificada como um grafo de consulta (*query graph*), que é um sub-grafo conectado do grafo do esquema contendo as classes e os relacionamentos envolvidos na consulta. Também são especificadas uma cláusula de restrição de consulta (*Query Restriction Clause – QRC*) para cada classe existente no grafo de consultas. Cada QRC é definida como uma combinação booleana de predicados conectados por operadores lógicos (tais como AND, OR e NOT) e seleciona todos os objetos desta classe pelo seu predicado de consulta.

Checagem de tipos. Depois da normalização de predicados e da reescrita algébrica da consulta, a expressão algébrica é avaliada quanto a correção de tipos. A complexidade desta avaliação reside no fato de os resultados intermediários, que são conjuntos de objetos, poderem ser de tipos distintos.

Otimização Algébrica. O próximo passo é aplicar regras de transformação algébrica que substituem uma expressão algébrica corretamente tipada por outra expressão equivalente com desempenho superior.

Geração do plano. Finalmente, é encontrado um plano de execução físico da consulta a partir da expressão algébrica anteriormente otimizada, onde são levados em consideração aspectos físicos que não eram levados em fases anteriores, tais como cardinalidades das coleções e índices.

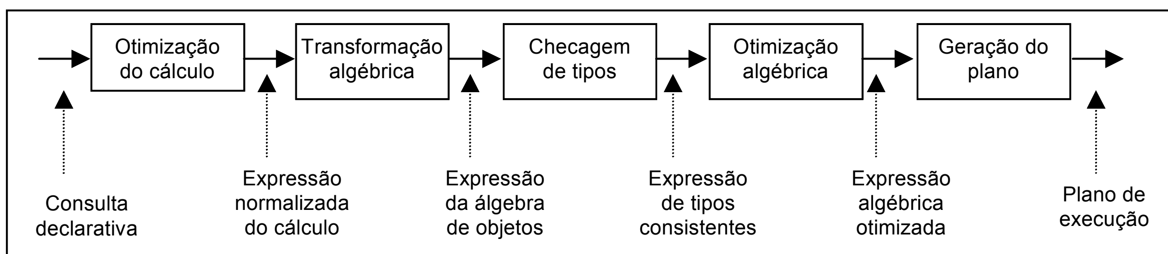


Figura 3 - Metodologia de processamento de consultas proposta por ÖZSU e BLAKELEY(1995)

As três primeiras etapas da metodologia (otimização do cálculo, transformação algébrica e checagem de tipos) podem ser consideradas etapas de preparação, ou pré-processamento, da consulta submetida ao processador de consultas. Já as últimas duas etapas (otimização algébrica e geração do plano) são responsáveis pelo processamento efetivo da consulta, e podem ser consideradas as etapas mais importantes do processo. A separação em duas etapas distintas – otimização algébrica (processo lógico de alto nível) e geração do plano de execução (processo físico de baixo nível) – é de vital importância para o processador de consultas, pois aumenta o seu grau de extensibilidade e modularidade, permitindo que cada uma das etapas possa ser modificada isoladamente independente da outra, exceto pelas regras que mapeiam um operador algébrico num plano físico. A próxima Seção descreve em maiores detalhes estas duas importantes etapas do processamento de consultas.

6.1. Otimização Algébrica

O problema de otimização de consultas pode ser modelado como um problema genérico de procura por uma solução ótima, dentro de um conjunto de soluções viáveis, de acordo com um algoritmo de busca. Este algoritmo de busca é responsável por avaliar as possíveis soluções, e escolher a(s) melhor(es) segundo uma função de custo. No caso específico do problema de otimização de consultas, a solução ótima retornada pelo algoritmo de busca seria a expressão algébrica de menor custo que represente a consulta em questão. O conjunto de soluções viáveis (ou espaço de busca) seria o conjunto de todas as possíveis expressões algébricas equivalentes à expressão considerada. Portanto, os componentes essenciais de um otimizador de consultas são, segundo ÖZSU e BLAKELEY (1995), o espaço de busca e regras de transformação, o algoritmo de busca, e a função de custo, detalhados nesta Seção.

Espaço de busca e regras de transformação

As regras de transformação são responsáveis por gerar expressões de consultas alternativas que irão compor o espaço de busca, utilizando para esta geração propriedades da álgebra tais como comutatividade, distributividade e associatividade. A inexistência de uma definição de álgebra de objetos padrão na literatura torna esta geração do espaço de busca um problema em aberto e tópico atual de pesquisa na área, já que não há regras de transformação algébrica universalmente aceitas, visto que as regras são altamente dependentes da álgebra.

Algoritmo de busca

O algoritmo de busca permite definir a estratégia de varredura do espaço de busca, definindo a ordem de geração de novas soluções a partir das regras de transformação. As soluções alternativas geradas pelas regras substituem as soluções de mais alto custo já inclusas no espaço de busca. Os algoritmos diferenciam-se basicamente em dois tipos: enumerativos e “aleatórios”. Os enumerativos são os que exploram todas as possíveis soluções do espaço de busca, enquanto os aleatórios exploram apenas uma amostra do espaço de busca aleatoriamente gerada.

Função de custo

Funções de custo são tipicamente utilizadas pelo processador de consultas para estimar o custo total de uma expressão de consulta. Este custo total engloba os custos de processamento (ou utilização da CPU) e os custos de transferência de dados entre a memória secundária e a memória principal (ou custos de entrada/saída - E/S), além do custo de comunicação no caso de sistemas distribuídos.

Diversos modelos de custo para o processamento de consultas sobre objetos foram apresentados na literatura. GARDARIN *et al.* (1995) propõem uma função de custo para otimização de consultas sobre bases de objetos onde são consideradas diferentes políticas de armazenamento da base. Ainda nesta linha, GARDARIN *et al.* (1996) estimam o custo de diferentes métodos para execução de consultas. OZKAN *et al.* (1996) apresentam um modelo de custo para o processamento de consultas com expressões de caminho, considerando índices e diferentes estratégias de avaliação de expressões. Para o cálculo de seletividade em modelos de custo, BERTINO e FOSCOLI (1997) abordam métodos para estimar o número de objetos envolvidos na avaliação de expressões de caminho multivaloradas, com possíveis referências nulas, em uma base de objetos com herança.

Extensibilidade do Otimizador de Consultas

Uma característica importante em um otimizador de consultas é a sua extensibilidade, ou seja, a possibilidade de o otimizador de consultas trabalhar com diferentes estratégias de busca e/ou diferentes funções de custo sem que grandes mudanças na estrutura do gerente de objetos sejam necessárias. Esta facilidade permite ao desenvolvedor de um SGBD definir um ambiente experimental para que sejam testadas diferentes estratégias de execução de consultas e verificadas sob que condições algumas regras são mais adequadas ou não. Dessa forma, é possível definir heurísticas a serem usadas em outros otimizadores.

Otimizadores baseados em regras fornecem a extensibilidade através da definição de novas regras de transformação. Entretanto, a extensibilidade em outras dimensões são mais limitadas para alguns otimizadores e ferramentas encontradas na literatura. Normalmente, alguns oferecem fácil extensão para operadores e algoritmos de execução, enquanto fornecem uma estratégia de busca fixa. Por outro lado, otimizadores que oferecem estratégia de busca extensível, não fornecem extensibilidade a nível de álgebra.

Uma proposta da literatura para fornecer extensibilidade em todas as dimensões de um otimizador de consulta pode ser encontrada em KABRA e DEWITT (1999). Neste trabalho, é possível adicionar novos operadores e algoritmos de execução, assim como novas estratégias de busca. Além disso, é possível experimentar várias heurísticas que limitam o espaço de busca. A flexibilidade encontrada nesta ferramenta deve-se à exploração das características orientadas a objeto provenientes da linguagem C++. A ferramenta, que chama-se OPT++, define um conjunto de classes abstratas com métodos virtuais. Estas classes abstratas não possuem nenhuma informação sobre a álgebra de operadores, sendo portanto independentes da álgebra, seja ela relacional ou não. Toda a estratégia de busca é implementada via métodos destas classes abstratas. Outra proposta encontrada é a geração de um otimizador a partir do OPTGEN (OPTGEN, 1998), uma ferramenta para escrita de otimizadores. O OPTGEN é baseado em uma linguagem declarativa chamada OPTL. Esta linguagem possibilita a especificação de regras de reescrita que permitem o mapeamento entre diversas formas intermediárias de especificação de consulta (álgebra lógica e álgebra física).

6.2. Representação de Planos de Execução

Conforme já mencionamos, o operador de junção é utilizado em consultas que expressam relacionamentos entre os dados armazenados, e representa o mais custoso operador de avaliação para um processador de consulta. Desta forma, o desempenho de consultas no modelo OO com expressões de caminho é especialmente influenciado pela escolha da estratégia e dos algoritmos utilizados para processá-la. Esta Seção analisa os aspectos mais importantes sobre estratégias e algoritmos para o processamento de consultas que utilizam expressões de caminho.

Algoritmos de Consulta

O processamento eficiente de expressões de caminho é um problema que vem merecendo bastante atenção nos últimos anos. Expressões de caminho podem ser encontradas não só na OQL, mas também na SQL:1999 e XML. Isto nos mostra que sua aplicabilidade não se restringe somente a sistemas OO, sendo importante o seu estudo. Entretanto, 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. Além disso, muitos trabalhos na literatura não tratam explicitamente de expressões de caminho, mas exploram o uso de ponteiros no processamento de junções, o que não deixa de ser uma técnica válida para o processamento de expressões de caminho.

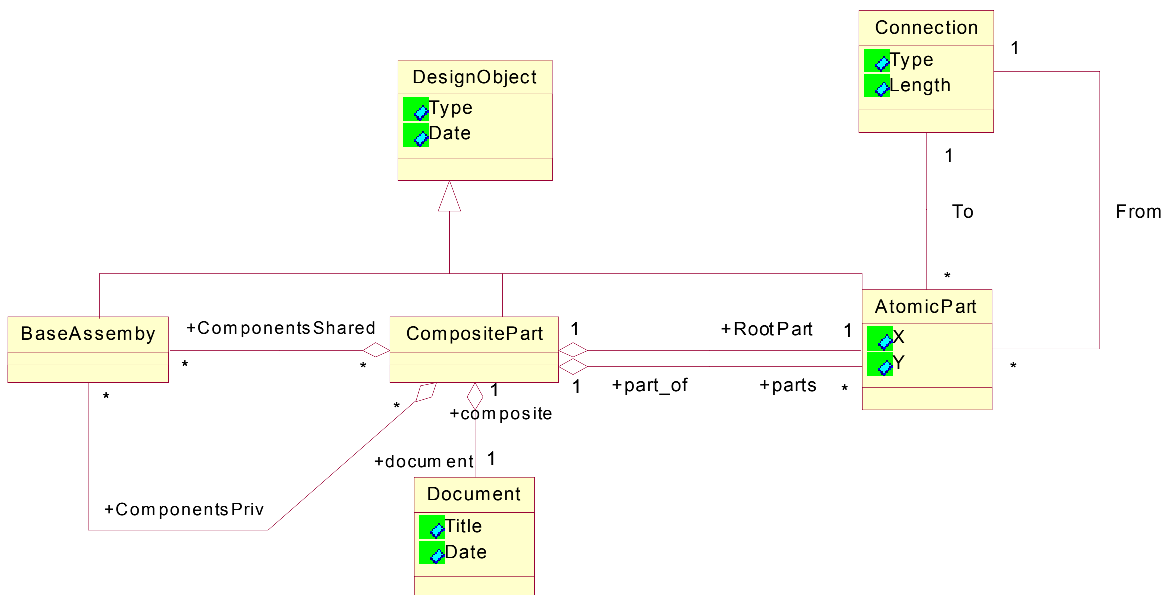
Duas dimensões são importantes no processamento de expressões de caminho. Uma delas é referente à direção em que a expressão de caminho é percorrida. As duas estratégias possíveis para esta dimensão são *descendente* e *ascendente*. A estratégia descendente indica que a expressão de caminho deve ser percorrida na

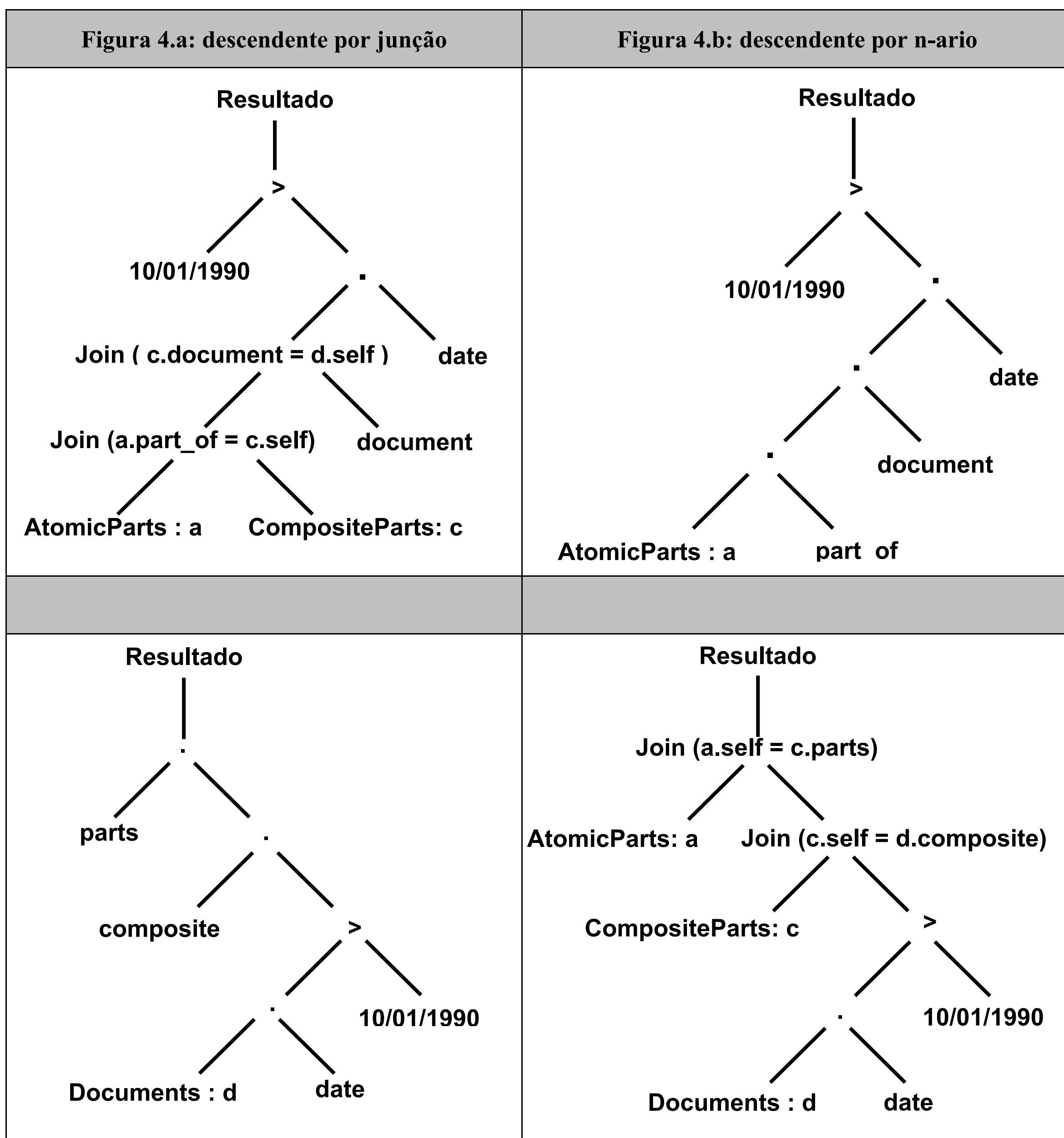
ordem em que foi escrita pelo usuário, ou seja, da classe raiz para a classe folha, enquanto a estratégia ascendente indica que a expressão deve ser percorrida na ordem inversa (da classe folha para a classe raiz).

Outra dimensão importante no processamento de expressões de caminho diz respeito ao operador utilizado algoritmos utilizados em cada uma destas estratégias. Existem basicamente duas classes de algoritmos para o seus operadores: algoritmos baseados em *operadores n-arios* e baseados em *junções (operadores binários)*. Os operadores da álgebra de objetos que são implementados por estes algoritmos correspondem ao *map* e à *junção*, respectivamente. O *map* materializa a referência ao objeto do relacionamento na expressão de caminho. O algoritmo mais conhecido para o operador n-ario é o "naive pointer chasing". Já a junção resolve o relacionamento através de uma operação análoga à junção do modelo relacional, só que realizada sobre referências (ou identificadores de objetos - OIDs) e não sobre valores (chaves). Basicamente, o *map* trabalha com referências a instâncias individuais enquanto que a junção opera sobre coleções de referências.

A combinação destas dimensões formam diferentes estratégias de processamento de expressões de caminho. Cada combinação – (descendente/n-ario), (ascendente/junção), (descendente/junção) e (ascendente/n-ario) – comporta-se de forma diferente dependendo de parâmetros da expressão como as cardinalidades das coleções operando e fatores de seletividade. Sob certas condições, alguma destas combinações terá desempenho superior às demais. Estas estratégias podem ser mais bem entendidas se mostrarmos a representação em árvore de cada uma delas 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
```





Repare que a estratégia **ascendente por operador n-ario** (Figura 4.c) necessita do atributo inverso do relacionamento. Entretanto, o padrão ODMG não determina que atributos inversos sejam obrigatórios. Portanto, esta estratégia pode ter sua aplicabilidade reduzida.

As tabelas 1 e 2 seguintes apresentam resumidamente os principais trabalhos na literatura e os classifica de acordo com as propriedades mais relevantes e segundo as dimensões de processamento.

Tabela 1: Avaliação das principais características dos algoritmos de expressões de caminho na literatura

	Braumandl et al. (1998)	***Shekita & Carey (1990)***	***DeWitt et al. (1993)***	***Keller & Maier (1991)***	***Tavares (1999)***
OIDs	-	Físico	Físico	-	-
Existência de extensões de classe opcional	Sim	Não	Não	Não	nao
Analisa operador N-ario	Sim	Não	Não	Sim	Sim
Analisa operador Junção	Sim	Sim	Sim	Não	Sim
Variação das diferenças de cardinalidades	Não	Sim	Não	Não	Sim
Variação do fatores de seletividade	Sim	Sim	Não	Não	Sim
Apresenta modelo Experimental	Sim	Não	Não	Sim	Sim
Considera atributos multivalorados	Sim	Não	Sim	Sim	Sim

Tabela 2: Classificação dos algoritmos de avaliação de expressões de caminho na literatura

	Operador N-ario (Baseada em ponteiros)	**Operador Junção**	
		Baseada em Ponteiros	**Baseada em Valores**
Descendente	Gardarin *et al.* (1996), Tavares et al.(2000), Keller & Maier (1991)	Shekita & Carey (1990), Braumandl *et al.* (1998), DeWitt et al. (1993)	Shekita & Carey (1990)
Ascendente	---		Tavares et al.(2000), Gardarin *et al.* (1996), DeWitt et al. (1993)

Um dos primeiros trabalhos a analisar a junção de coleções foi mostrado em VALDURIEZ (1987). Neste trabalho, uma estrutura de dados auxiliar chamada *join index* foi usada para acelerar o processamento de junções. Um *join index* para duas relações R e S essencialmente pré-calcula a junção entre as duas relações, armazenando pares de identificadores de tupla (TIDs). O desempenho desta técnica foi comparada ao processamento da junção pelo algoritmo *hybrid-hash join*.

Posteriormente, o trabalho clássico de SHEKITA e CAREY (1990) avaliou por simulação o desempenho de três algoritmos de junção baseados em ponteiros (*nested-loops*, *sort-merge* e *hybrid-hash*) em um modelo OO. Seus desempenhos foram comparados aos seus similares relacionais. Não foi avaliado o comportamento da estratégia baseada em operador n-ario. Os algoritmos foram usados para implementar relacionamentos muitos-para-um. Neste relacionamento, duas coleções R e S relacionavam-se unidirecionalmente ao incluir um atributo referência em S direcionado para R. Isto é diferente do relacionamento um-para-muitos que inclui um atributo coleção de S em R. Além disso, OIDs físicos foram utilizados. Este trabalho concluiu que para junções em que a coleção R tem baixa seletividade e as coleções R e S têm o mesmo tamanho aproximadamente, os algoritmos OO *sort-merge* e *hybrid-hash* sobressaem-se em relação aos outros. Entretanto, se R for muito maior que S, eles não executam tão bem quanto os algoritmos de junção tradicionais (*value-based*). Esse é o caso de usar a estratégia baseada em ponteiros. Se o fator de seletividade em R for aumentado, os algoritmos *sort-merge* e *hybrid-hash* mais uma vez se mostram melhores, pois o

tamanho de R se aproxima do de S. O algoritmo *nested-loops* se mostrou ruim em quase todos os testes, exceto quando a seletividade de R era tão alta que o tamanho de R era bem menor que o de S.

O trabalho de DEWITT *et al.* (1993) representa uma extensão ao trabalho de SHEKITA e CAREY (1990) para ambientes paralelos, e é avaliado em detalhes na Seção 7.3. No entanto algumas de suas características já são apresentadas nas Tabelas 1 e 2 para permitir uma comparação deste com os demais algoritmos aqui apresentados.

Em BRAUMANDL *et al.* (1998) foi proposto um algoritmo para processar expressões de caminho que reduz o acesso aleatório ao disco através de técnicas de particionamento e ordenação. Antes de sair referenciando os objetos apontados, o algoritmo ordena os identificadores baseado em seu endereço físico, desta forma aproveitando o agrupamento físico já existente. O algoritmo proposto é independente da implementação do OID (lógico ou físico) e processa expressões de caminho multi-valoradas, ou seja, relacionamentos um-para-muitos. O desempenho do algoritmo proposto foi avaliado tanto junto à estratégia de junção, quanto à estratégia baseada em operador n-ario. Foi apresentado um modelo teórico em que os resultados experimentais puderam ser comparados e validados junto ao mesmo. Variação de parâmetros importantes como o tamanho da memória, a cardinalidade e a seletividade nas coleções foram verificadas.

TAVARES (1999) avaliou as estratégias ascendente por junções e descendente com operador n-ario, apresentando resultados experimentais e heurísticas para a avaliação de expressões de caminho com paralelismo.

O trabalho de GARDARIN *et al.*(1996) mostrou as vantagens dos algoritmos baseados em operador n-ario e apresentou heurísticas para o seu uso através de um modelo de custo simplificado. A direção de avaliação (ascendente x descendente) também foi levada em conta. Foram apresentados resultados de simulação e experimentais para 3 estratégias de avaliação: descendente com operador n-ario (*forward naïve pointer chasing*), descendente por junção (*forward pointer-based join)* e ascendente por junção (*reverse value-based join*).

6.3. Geração do Plano de Execução

Quando uma consulta é submetida a um SBDOO, um dos principais requisitos é que sua execução seja realizada de forma eficiente pelo processador de consultas (ATKINSON et al., 1989). O processo de otimização de consultas consiste em gerar o melhor plano de execução para uma dada consulta, o qual deve minimizar uma determinada função de custo (ÖZSU e VALDURIEZ, 1999).

Como já dito anteriormente, os SGBDs relacionais beneficiam-se da correspondência direta entre os operadores da álgebra e as primitivas de acesso e armazenamento dos dados. Portanto, a escolha do melhor plano de execução resume-se a escolher os algoritmos mais eficientes que implementam os operadores relacionais e suas combinações (árvore de operadores). Esta escolha é fortemente baseada nas estatísticas armazenadas no sistema. Porém, esta vantagem não pode ser obtida em SGBDOOs devido à diferença

semântica entre a interface dos objetos (normalmente via métodos) e a estrutura física dos objetos, que está escondida do usuário através do encapsulamento.

Assim, o encapsulamento e o armazenamento de métodos com objetos caracterizam os fatores de dificuldade no processamento de consulta. Por causa disso, o acesso aos objetos são deixados por conta do gerente de objetos, que recebe solicitações do processador de consulta e devolve os objetos que satisfazem os predicados de consulta. Por outro lado, algumas implementações permitem que o otimizador acesse diretamente os objetos armazenados e o próprio otimizador tenha conhecimento da estrutura de armazenamento dos objetos, violando o encapsulamento. Esta é uma decisão de projeto do módulo de geração do plano (também denominado módulo de execução de consultas) que afeta a construção de todo o sistema. De fato, esta segunda estratégia de implementação foi a abordagem adotada pelo ODMG 3.0 como padrão.

Qualquer módulo de execução de consultas requer pelo menos três tipos de algoritmos sobre coleções de objetos: *scan, indexed scan* e *collection matching*. O primeiro é uma classe de algoritmos simples que acessam seqüencialmente todos os objetos presentes em uma coleção. *Indexed scan* é uma classe de algoritmos que usam alguma estrutura de índice para acessarem eficientemente os objetos de uma coleção. Deve ser possível definir índices sobre atributos simples ou até mesmo sobre expressões de caminho, como mostrado em OGASAWARA e MATTOSO (1999). A última classe de algoritmos funciona sobre múltiplas coleções de entrada para produzir uma coleção de saída. Muitos algoritmos de junções relacionais são modificados para funcionarem sobre coleções de objetos e se enquadram nesta categoria. Além disso, existem outros algoritmos propostos na literatura que lidam com expressões de caminho e muitos trabalhos tratam de avaliar como os algoritmos propostos se comportam em diversas situações e benchmarks, para dessa forma tentar extrair regras e heurísticas que possam ser usadas no processo de otimização de consultas. Os trabalhos de TAVARES (1999), GARDARIN *et al.* (1996), SHEKITA e CAREY (1990) e BRAUMANDL *et al.* (1998) merecem destaque neste sentido.

7. CONSULTAS EM AMBIENTES DE PARALELISMO E DISTRIBUIÇÃO

7.1. Fragmentação de Coleções no Modelo OO

A fragmentação de coleções no modelo OO é responsável pelo agrupamento, em fragmentos de coleção de uma classe, das informações que são acessadas por uma consulta ou transação executada sobre a base de dados. Estes fragmentos de coleção serão posteriormente alocados pelos nós do sistema.

As principais vantagens alcançadas com a fragmentação são:

☺ a **unidade de distribuição**, pois a maior parte das consultas e transações manipula apenas um subconjunto das informações pertencentes aos objetos de uma determinada classe, fazendo com que fragmentos de coleção mostrem-se mais adequados para servirem como unidades de distribuição, já que eliminam a quantidade de dados irrelevantes acessados por estas operações;

☺ a possibilidade **de transações concorrentes e de execução paralela** de uma operação que acesse diversos fragmentos; e

☺ o **aumento de desempenho** de cada nó do sistema quando este trabalha com um volume menor de informação, permitindo que haja otimização do uso da memória local, como mostrado por LIMA e MATTOSO (1996).

No entanto, podemos citar algumas desvantagens da fragmentação, como:

☹ **perda de desempenho** em operações que necessitam acessar diversos fragmentos em diversos nós da rede; e

☹ **aumento da complexidade** no controle semântico, para que sejam mantidas a integridade referencial e a consistência das informações fragmentadas.

Existem dois tipos básicos de fragmentação de uma classe: **horizontal** e **vertical**. (ÖZSU e VALDURIEZ, 1999). Para definir cada um destes dois tipos, consideremos uma classe genérica C como uma relação C = (**Id, A, M, I**), onde **Id** é o identificador da classe, **A** é o seu conjunto de atributos, **M** o seu conjunto de métodos e **I** o conjunto das instâncias da classe (objetos) que são definidos utilizando-se **A** e **M**. Na fragmentação horizontal, distribuem-se as instâncias de uma classe em diversos fragmentos, os quais terão a mesma estrutura mas o conteúdo diferente. A representação de um fragmento horizontal C_h da classe C definida anteriormente seria C_h = (**Id, A, M, I'**), onde ($I' \subseteq I$). Já na fragmentação vertical, divide-se a estrutura lógica (atributos e métodos) das instâncias da classe, distribuindo-a entre os fragmentos, de modo que os fragmentos conterão os mesmos objetos, mas com estruturas diferentes. Um fragmento vertical C_v de C seria da forma C_v = (**Id, A', M', I**), com ($A' \subseteq A$) e ($M' \subseteq M$). Há também a fragmentação híbrida, que combina as duas anteriores. Neste caso, um fragmento híbrido C_{hv} de C seria da forma C_{hv} = (**Id, A', M', I'**), com ($A' \subseteq A$), ($M' \subseteq M$) e ($I' \subseteq I$).

A fragmentação horizontal, assim como no modelo relacional, pode ser subdividida entre fragmentação horizontal primária e derivada. Entretanto, devido a fatores como direção do relacionamento e multiplicidade de atributos nas duas direções do relacionamento, a definição do papel da entidade dono e membro não é tão evidente quanto no modelo relacional normalizado. (BAIÃO et al. 2000) apresentam definições para esses conceitos e analisam o comportamento do projeto de fragmentação

Um aspecto importante para a fragmentação de dados é o **nível de transparência** que é fornecido pelo SGBDOO. Esta transparência diz respeito ao tratamento do acesso aos dados remotos, que pode ser feito de forma transparente ou pode ficar a cargo do usuário. No último caso o usuário deve ter o conhecimento de onde residem os dados, e é obrigado a direcionar os seus acessos. Segundo a documentação do SGBDOO O_2 (O_2 TECHNOLOGY, 1993), por exemplo, o nível de transparência fornecido por este sistema no acesso aos fragmentos horizontais de uma classe é insatisfatório, pois o usuário é obrigado a direcionar manualmente os seus acessos, ao contrário do acesso aos fragmentos verticais onde existe um mecanismo de troca de contexto

automática pelo qual o próprio sistema identifica a localização dos dados, e direciona os acessos automaticamente. BAIÃO e CALEGARIO (1996) apresentam um experimento realizado no SGBDOO O_2 que ilustra este aspecto.

Corretude da fragmentação

Durante a fragmentação de uma base de dados, existem duas características que devem ser observadas, que são a garantia da consistência da base de dados como um todo e a garantia da semântica das informações nela armazenadas. Para garantir tais características, ÖZSU e VALDURIEZ (1999) definem, no modelo relacional, três regras de correção que devem ser respeitadas por qualquer algoritmo de fragmentação. Esta Seção apresenta a definição destas regras no modelo relacional, e uma proposta de adaptação para o modelo orientado a objetos que foi apresentada em (BAIÃO, 1997), da definição destas três regras.

Modelo Relacional

As três regras de corretude da fragmentação definidas para o modelo relacional em (ÖZSU e VALDURIEZ, 1999) são descritas abaixo:

- **Abrangência:** Assegura que os dados em uma relação global sejam mapeados em fragmentos de relações sem nenhuma perda:

  Seja R uma instância de uma relação decomposta em fragmentos R_1, R_2, ... R_n, cada item de dado que pode ser achado em R também poderá ser achado em um ou mais R_i.

Na fragmentação horizontal, um `item de dado` se refere a uma tupla, enquanto que na fragmentação vertical este refere-se a um atributo.

- **Disjunção:** Assegura que os fragmentos horizontais sejam disjuntos:

  Seja R uma relação decomposta em fragmentos horizontais R_1, R_2, ... R_n, se um item de dado se encontra em R_i então este item de dado não está presente em nenhum outro R_j, $j \neq i$.

No caso da fragmentação vertical, todos os atributos chaves primários da relação global estarão repetidos em todos os fragmentos verticais, e a regra de disjunção se aplica apenas aos atributos chaves não primários da relação.

- **Reconstrução:** Assegura a preservação das restrições definidas sobre os dados em forma de dependências:

  Seja R uma relação decomposta em fragmentos R_1, R_2, ... R_n, deve ser possível definir-se um operador relacional ∇ tal que $R = \nabla R_i$.

O operador ∇ vai ser diferente para cada tipo de fragmentação, mas é importante que possa ser identificado.

Modelo OO

Para o modelo orientado a objetos, não existe na literatura nenhuma definição de regras de corretude para o processo de fragmentação de classes, apesar de também ser muito importante que a consistência e a semântica dos dados sejam mantidas. Esta Seção apresenta uma proposta de adaptação para o modelo orientado a objetos da definição das três regras descritas anteriormente.

- **Abrangência:** Assegura que o mapeamento das classes em fragmentos de classes seja realizado sem nenhuma perda, ou seja, cada `item de dado` pertencente a base de dados centralizada também pertencerá a um ou mais fragmentos da base de dados distribuída.

    Seja C uma classe decomposta em fragmentos C_1, C_2, ... C_n, cada item de dado que pode ser achado em C também poderá ser achado em um ou mais C_i.

O conceito de `item de dado`, como no modelo relacional, varia de acordo com a estratégia de fragmentação utilizada. No caso da fragmentação horizontal, um item de dado seria um objeto ou instância da classe, enquanto que na vertical este poderia ser um atributo ou um método da classe.

- **Disjunção:** Assegura que todos os fragmentos de uma classe sejam disjuntos, ou seja, cada item de dado presente na base de dados centralizada deverá estar presente em apenas um dos fragmentos da base de dados, não considerando técnicas de replicação de dados.

    Seja C uma classe decomposta em fragmentos C_1, C_2, ... C_n, se um item de dado se encontra em C_i então este item de dado não está presente em nenhum outro C_j, $j \neq i$.

No caso da fragmentação vertical, dependendo da implementação do SGBDDOO, apenas o identificador interno dos objetos (que é transparente ao usuário) pode estar replicado em cada fragmento da classe, para permitir a reconstrução da classe a partir dos seus fragmentos. Quando esta replicação não acontece, o próprio SGBDDOO é responsável por permitir, de algum modo, esta reconstrução. Quanto aos métodos das classes, cada sistema apresenta um nível de suporte diferente para tratar a replicação de seus códigos-fonte.

- **Reconstrução**: Assegura a preservação das restrições definidas sobre os dados. Analogamente ao modelo relacional, deve ser possível apresentar um conjunto de operações definidas em uma álgebra adequada ao modelo orientado a objetos que podem, quando aplicadas aos fragmentos da base de dados distribuída, reconstruir a base de dados centralizada.

    Seja C uma classe decomposta em fragmentos C_1, C_2, ... C_n, é possível definir-se um operador ∇ tal que $C = \nabla C_i$.

O operador ∇ é definido sobre o tipo `Coleção` (*Collection<T>*) de acordo com o padrão ODMG (CATTELL *et al.*, 2000). Considera-se que a extensão de uma classe seja do tipo `Conjunto` (*Set<T>*), que é sub-tipo de `Coleção`.

Para esclarecer um pouco mais a regra de reconstrução em função do operador ∇, considere o exemplo abaixo no qual procurou-se mostrar a operação realizada por um algoritmo de fragmentação horizontal e a obtenção de um operador que desfizesse esta operação e reconstruísse a classe original. No exemplo da fragmentação vertical de uma classe, a definição de um operador de reconstrução é mais complicada, e dependente da implementação do SGBDDOO, já que a princípio é necessário fazer uma junção sobre o identificador do objeto, que não é visível ao usuário. A linguagem utilizada nos exemplos segue a sintaxe da **linguagem de definição de objetos** (**ODL**) e da **linguagem de consultas a objetos** (**OQL**), definidas no padrão ODMG:

Seja a definição da classe `Pessoa` e a representação de sua extensão dadas na figura 5:

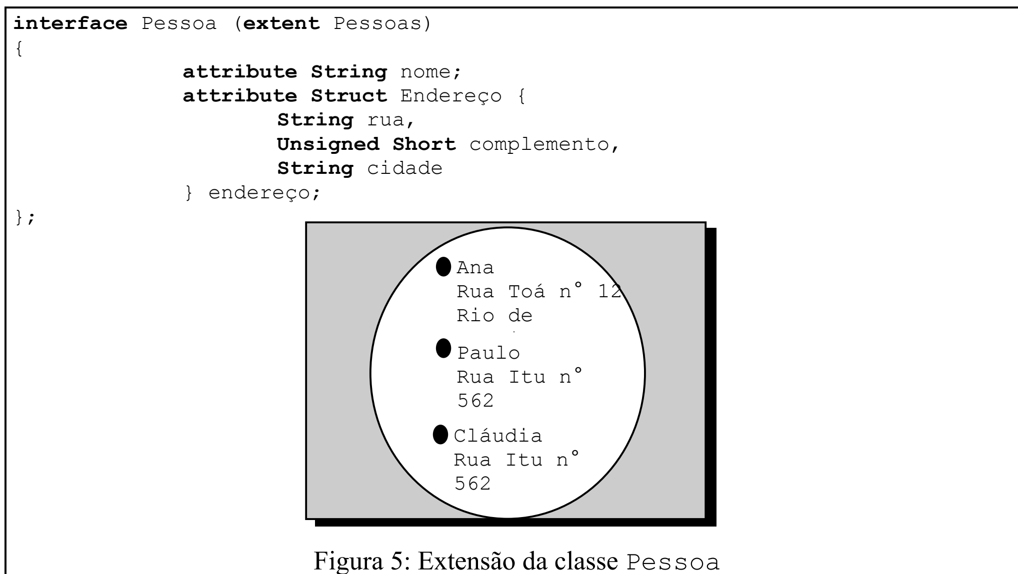


Figura 5: Extensão da classe `Pessoa`

Suponha que se fragmente horizontalmente a classe `Pessoa` em relação ao atributo `nome`, de acordo com os predicados (`nome < 'N') e (nome ≥ 'N')`. Neste caso, criaríamos dois fragmentos:

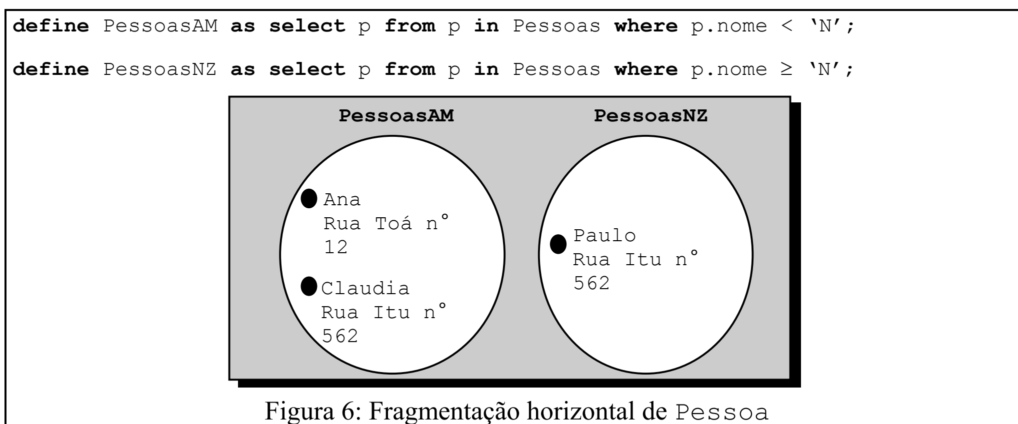


Figura 6: Fragmentação horizontal de `Pessoa`

A reconstrução da classe `Pessoa`, com todos os seus objetos, pode ser feita a qualquer momento a partir do operador união (*union*):

```
define Pessoas as PessoasAM.union(PessoasNZ);
```

No caso da realização de uma fragmentação vertical da classe `Pessoa`, utilizaremos como exemplo o SGBDOO O_2 (O_2 TECHNOLOGY, 1993) para considerar a sua implementação. Digamos que se deseja fragmentar verticalmente o atributo `endereço` da classe `Pessoa`, então seria criada uma classe `Endereço` em um fragmento:

```
interface Endereço
{
            attribute String rua;
            attribute Unsigned Short complemento;
            attribute String cidade;
};
```

A definição da classe `Pessoa`, cujos objetos seriam armazenados em outro fragmento, e a representação da sua extensão ficariam da forma:

```
interface Pessoa (extent Pessoas)
{
            attribute String nome;
            relationship Endereço endereço;
};
```

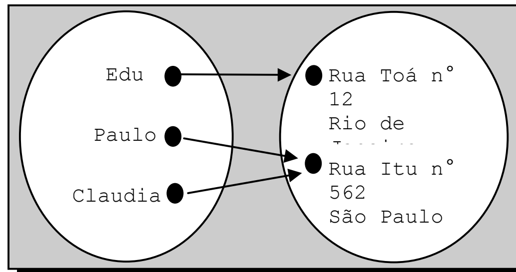


Figura 7: Fragmentação vertical de `Pessoa`

O acesso aos atributos e métodos da classe `Pessoa` seria independente da sua localização e pode ser feito de modo transparente, exatamente como se a classe não estivesse fragmentada, devido à criação de um relacionamento entre as classes `Pessoa` e `Endereço`. A reconstrução da classe `Pessoa` original não seria necessária, já que toda a estrutura do objeto original continua acessível através do operador "." (ponto ou *map*, como descrito anteriorermente). Por exemplo, poderíamos ter a consulta abaixo:

```
define Pessoas as select p from p in Pessoas where

p.endereço.cidade = 'Rio de Janeiro' and p.nome > 'J';
```

7.2. Alocação de Classes no Modelo OO

A alocação de dados é responsável pelo armazenamento físico dos fragmentos de classe que foram gerados pela etapa de fragmentação entre os nós do sistema e pela replicação de dados. Vale lembrar que o problema da replicação de dados em sistemas distribuídos relacionais comerciais ainda se apresenta resolvido de forma insatisfatória, utilizando *snapshots* ou transações de atualização de múltiplas cópias dos dados para

o gerenciamento da replicação de dados, como dito por MOLINA e HSU (1995). Já nos sistemas OO este problema se encontra ainda mais incipiente.

7.3. Otimização Algébrica

Álgebra OO

Os ambientes distribuídos e paralelos são um recurso utilizado por diversos projetistas de sistemas para o aumento de desempenho de suas aplicações. Neste sentido, é montada toda uma infraestrutura que dê suporte à execução das aplicações no ambiente distribuído e/ou paralelo. Em SGBDs que fornecem suporte à distribuição e paralelismo, o ideal é que sejam definidas camadas adicionais responsáveis por prover acesso a objetos remotos e mecanismos de sincronização de transações paralelas, no sentido de tornar este controle transparente ao processador de consultas, ao projetista da aplicação e ao usuário final.

A álgebra OO utilizada pelo processador de consultas em ambientes distribuídos e paralelos é muito semelhante à álgebra utilizada em ambientes sequenciais. Por outro lado, o processamento de consultas em SBDOOs distribuídos ou paralelos é uma tarefa muito complexa (SU et al., 1998), pois a álgebra de otimização deve conter, além dos operadores seqüenciais tradicionais, operadores responsáveis pela correspondência entre a representação lógica da consulta e o esquema distribuído. São exemplos os seguintes operadores:

- **`Split`** – fragmenta uma coleção pertencente à expressão de caminho. Este operador é próprio de ambientes paralelos;
- **`Merge`** – reagrupa os fragmentos de uma coleção;
- **`Exchange`** – representa a troca de uma coleção ou fragmento entre nós.

Espaço de busca e regras de transformação

Em ambientes distribuídos ou parelelos, o otimizador de consultas precisa reconhecer características especiais de otimização próprias da distribuição de processsamento, como a extração de paralelismo entre os operadores e a redução da comunicação entre os nós. Uma abordagem de otimização freqüentemente utilizada é apresentada na Figura 8.

Basicamente, a etapa inicial para otimização de uma árvore de consulta em um ambiente distribuído ou paralelo consiste em (1) identificar as unidades atômicas de execução e (2) identificar as restrições temporais entre estas unidades. Assim, estas unidades podem ser arranjadas de maneira a gerar uma árvore de execução com melhor desempenho. MENDES e SAMPAIO (1998) apresentam um algoritmo para realizar estes passos, cujo principal objetivo é extrair o paralelismo durante a otimização da consulta.

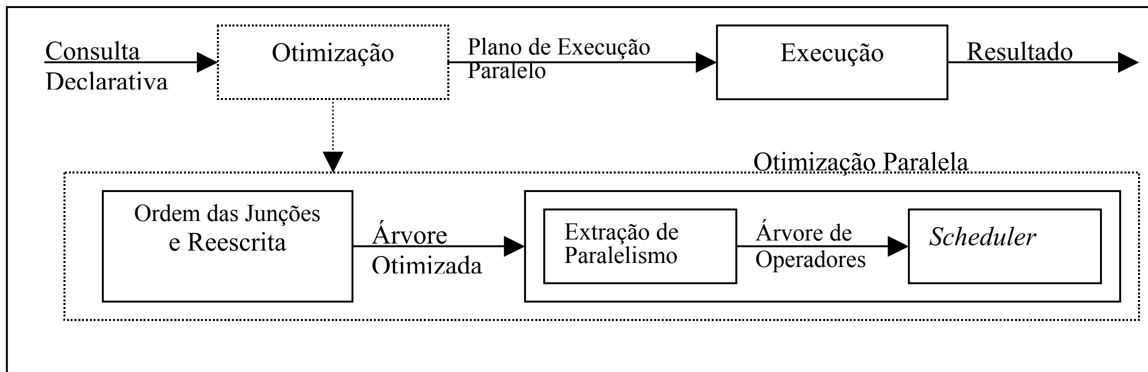


Figura 8. Processo de otimização de consultas paralelas (HASAN *et al.*, 1996).

Em um SBDOO distribuído ou paralelo, uma árvore balanceada para a execução de uma consulta apresenta em geral um melhor desempenho. Uma árvore balanceada é também conhecida por "*bushy tree*". A representação inicial de uma consulta resulta em uma árvore profunda à esquerda ou profunda à direita. Durante a etapa de otimização, transformações baseadas em heurísticas são aplicadas à árvore incial de maneira a diminuir a sua altura. Estas transformações podem ser realizadas em uma abordagem *top-down*, *bottom-up* ou híbrida (DU *et al.*, 1995).

Como a execução de consultas neste caso envolve um número maior de variáveis a ser consideradas, o tamanho do espaço de busca pode tornar o processo de otimização inviável. Muitas técnicas utilizadas no ambiente centralizado para reduzir e percorrer o espaço de soluções são válidas neste contexto, como por exemplo a programação dinâmica (KOSSMANN, 2000).

Algoritmo de busca

O trabalho de SHEKITA e CAREY (1990) apresentado na Seção 6.2 para ambientes centralizados foi continuado por DEWITT *et al.* (1993) ao avaliarem por simulação o comportamento destes algoritmos em um ambiente paralelo. Entretanto os algoritmos foram modificados para funcionarem com relacionamentos um-para-muitos, ou seja, atributos do tipo coleção. Além disso, foram propostos dois algoritmos que são variações do *hybrid-hash* e um algoritmo novo chamado *probe-child*. Estes algoritmos têm a vantagem de não exigirem que o atributo coleção possua objetos de uma classe com extensão. Foi apresentado um algoritmo que calcula uma extensão de classe para o atributo se a mesma não existir. Esta restrição está presente nos algoritmos implementados em SHEKITA e CAREY (1990). Na análise dos algoritmos, assim como em SHEKITA e CAREY (1990), foi descartada a análise da estratégia baseada no algoritmo *naïve pointer chasing*, embora algumas condições de análise da simulação sejam favoráveis a esta estratégia. Outra limitação encontrada em DEWITT *et al.* (1993) foi a ausência de variação das cardinalidades para a análise dos resultados ser mais completa.

Função de custo

Na maioria das pesquisas realizadas sobre técnicas de otimização de consultas orientadas a objetos, os otimizadores de consulta propostos presumem a utilização de um modelo de custo para selecionar os melhores

planos de execução para as consultas (GARDARIN *et al.*, 1995). Na etapa de geração do plano de execução são consideradas propriedades físicas relevantes, como o modelo de armazenamento de objetos e os métodos de acesso disponíveis. Uma função de custo atua como um mecanismo que permite descartar os planos de execução que sejam "reconhecidamente" ineficientes ou, ainda, que permite avaliar o melhor (ou os melhores) plano(s) de execução após o término da otimização. Além disso, algumas aplicações da álgebra lógica e da álgebra física dependem de informações físicas sobre a base de objetos. Este é o caso da escolha sobre a ordem em que são executadas certas junções.

Um modelo de custos utiliza estatísticas e parâmetros físicos armazenados pelo banco de dados para estimar o tempo de execução de uma dada consulta. A partir das estatísticas, parâmetros importantes são adicionalmente calculados, formando um conjunto de dados essenciais ao processo de otimização. Exemplos destes dados são:

- Cardinalidade de uma coleção;
- Número de referências entre objetos relacionados;
- Seletividade de um predicado sobre uma coleção;
- Número de fragmentos de uma coleção, em bases distribuídas;
- Número de páginas de memória ocupadas por uma coleção;
- Tamanho médio (em bytes) do objeto de uma dada coleção;
- Tipo de armazenamento físico (se existe agrupamento, qual tipo, etc.);
- Existência de índice de acesso;
- Tamanho da página de memória;
- Memória principal disponível.

Em um banco de dados centralizado, o custo do processamento de uma consulta pode ser simplificado como o somatório do custo das operações de E/S de dados e do custo de utilização de CPU. Por outro lado, em um ambiente distribuído deve ser considerado também o custo da comunicação entre os nós (OZSU e VALDURIEZ, 1999), de modo que:

$$Tempo_{TOTAL} = T_{CPU} \times \#insts + T_{E/S} \times \#E/Ss + T_{MSG} \times \#msgs + T_{TR} \times \#bytes$$

onde:

- $T_{E/S}$: custo para carregar uma página em memória (uma unidade de E/S);
- T_{CPU}: custo médio de execução de uma instrução na máquina;
- T_{MSG}: tempo fixo necessário para inicializar e receber uma mensagem;
- T_{TR}: tempo gasto para transmitir um pacote de dados (assumido como constante);
- *#insts*: número de instruções executadas pelo processador;
- *#E/Ss*: número de operações de entrada/saída de dados (E/S);
- *#msgs*: número de mensagens trocadas entre os nós;
- *#bytes*: número de *bytes* transmitidos durante a execução da consulta.

Em um ambiente distribuído ou paralelo, técnicas especiais podem ser consideradas no projeto do banco de dados, dentre elas a fragmentação (vertical, horizontal – primária ou derivada – ou híbrida) das coleções. Estas técnicas requerem que o SBDOO mantenha um número maior de estatísticas sobre as coleções. Além disso, parâmetros tradicionais passam a ser calculados de maneira diferente, como é o caso do número de páginas ocupadas por uma coleção. Em coleções fragmentadas, por exemplo, este parâmetro será dado pelo somatório das páginas ocupadas apenas pelos fragmentos que participarem da consulta, e não da coleção como um todo.

Na fragmentação horizontal primária, o número de objetos irrelevantes pode ser reduzido durante a execução de uma consulta. Por exemplo, a execução de uma junção entre as coleções de A, fragmentada horizontalmente tal que $A = A_1 \cup A_2$, e B pode ser realizada das seguintes maneiras:

$$(A_1 \cup A_2) \infty B \quad \text{ou} \quad (A_1 \infty B) \cup (A_2 \infty B).$$

Em certos casos, a função de fragmentação de A está definida de tal forma que é possível, pelo predicado de seleção da consulta submetida, identificar que algum termo $(A_i \infty B)$ resulta em um conjunto vazio, possibilitando que o processador de consultas desconsidere tal fragmento. Esta eliminação de objetos irrelevantes proporciona um considerável ganho nas operações de entrada e saída de dados (E/S) e de CPU durante o processamento da consulta. Quanto ao custo de E/S, apenas as páginas referentes aos fragmentos envolvidos na consulta serão carregadas. Caso existam réplicas de B alocadas em cada nó que armazena um fragmento de A, o custo de comunicação também poderá ser minimizado. Este ganho tende a ser máximo no caso da fragmentação horizontal derivada, considerando que os fragmentos correspondentes das coleções A e B estarão alocados juntos.

Por outro lado, a fragmentação vertical permite que o tamanho dos objetos envolvidos nas consultas seja reduzido, diminuindo assim o volume de dados manipulado durante o processamento e demandando uma quantidade menor de memória para a execução da consulta. Porém, o otimizador precisará reconhecer a equivalência entre as semi-junções expressas nas consultas e os fragmentos verticais correspondentes.

A funcionalidade básica de um modelo de custo é estimar a cardinalidade e, conseqüentemente, o volume de páginas acessadas pelas coleções envolvidas e pelos resultados intermediários gerados na expressão de caminho. Neste ponto, a estratégia de avaliação da expressão – ascendente ou descendente – e os algoritmos utilizados irão determinar o cálculo dos custos envolvidos. Os parâmetros *fan-out* e *share* refletem o grau de saída e o grau de compartilhamento, respectivamente, entre duas coleções (suponham Ci e Cj) relacionadas na consulta, e desta forma derivam de propriedades dos relacionamentos instanciados na base. Devem ser calculados considerando a participação parcial das classes no relacionamento (principalmente em relação à classe Cj). Eles são utilizados para o cálculo de resultados intermediários no processamento de expressões de caminho e são aplicados à coleção Ci, no caso do *fan-out*, e à coleção Cj, para o *share*.

Se duas classes, Ci e Cj, possuem um relacionamento entre elas, o cálculo dos parâmetros *fan-out* e *share* dependerá apenas da cardinalidade do relacionamento na base (i.e., do número de relacionamentos entre objetos armazenados). Neste caso, o valor destes parâmetros não mudará nas diferentes expressões de caminho que contenham Ci e Cj, estando estas coleções unidas pelo relacionamento em questão.

De maneira análoga ao apresentado em *CHO et al.* (1996), temos que o cálculo destes parâmetros será dado por:

$$fan_out(C_i, C_j) = \frac{n(C_i, C_j)}{n(C_i)} \quad \text{e} \quad share(C_i, C_j) = \frac{n(C_i, C_j)}{n(C_j)}$$

Onde n(Ci) e n(Cj) representam o número de elementos de Ci e de Cj, respectivamente. O parâmetro n(Ci,Cj) é o número total de referências (não necessariamente distintas) a partir de objetos da coleção Ci para objetos de Cj. É importante ressaltar que o parâmetro *fan-out* reflete o grau de saída de toda a coleção Ci para a coleção Cj. Por isso, devem ser considerados todos os objetos de Ci. De maneira análoga, devemos considerar todos os objetos da coleção Cj quando calculamos *share*.

Existem alguns trabalhos na literatura que abordam modelos de custo para o processamento distribuído ou paralelo de consultas em SBDOOs. BELLATRECHE *et al.* (1998) e FUNG *et al.* (1997) apresentam modelos de custos que avaliam os resultados obtidos por um projeto de fragmentação horizontal primária e vertical de classes, respectivamente, mas consideram uma política simplificada de armazenamento de objetos e tratam apenas uma estratégia para avaliação de expressões de caminho, o que geralmente não ocorre na prática. CHO *et al.* (1996) tratam das técnicas para estimar a seletividade em consultas orientadas a objetos, considerando a participação parcial das classes, sem abordar o cálculo de custos envolvidos. Embora não seja mencionado neste trabalho, a participação parcial é uma característica própria de bases fragmentadas horizontalmente. Por isso, consideramos esta proposta uma contribuição significativa para a modelagem de custos no processamento de consultas sobre bases de objetos distribuídas.

7.4. Geração do Plano de Execução

O plano de execução de uma consulta corresponde a uma determinada seqüência de algoritmos sobre as coleções envolvidas, escolhidos dentre os disponíveis no SBDOO (GRUSER, 1996). A geração de um plano de execução em um ambiente centralizado envolve vários fatores, conforme vimos anteriormente, baseados em questões como:

- métodos de acesso disponíveis para a consulta;
- diferentes algoritmos de execução;
- ordem das junções a serem realizadas;
- materialização dos resultados intermediários.

Além de considerar estes fatores, o otimizador de consultas em um ambiente distribuído ou paralelo deve decidir também sobre características próprias da distribuição de processamento, como a seleção dos nós onde

cada parte da consulta será executada e o uso de técnicas especiais para a execução dos algoritmos escolhidos (KOSSMANN, 2000). Dentre estas técnicas especiais, citamos a execução de junções sobre coleções fragmentadas e o acesso a coleções replicadas.

Um plano de execução paralelo inclui algoritmos próprios, em geral adaptações dos algoritmos encontrados no contexto seqüencial como os apresentados na Seção anterior. Exemplos destes algoritmos podem ser encontrados em TAVARES (1999), DEWITT *et al.* (1993) e LIEUWEN *et al.* (1993).

REFERÊNCIAS

BAIÃO, F., CALEGARIO, V., 1996, "SIGO2-Sul: Experimento prático com a utilização de hipertextos do O2 e conceitos de SIG", em MATTOSO, M., edição, "Banco de Dados Orientados a Objetos: Aplicações em SIGs", Relatório Técnico ES-396/96, COPPE/UFRJ, junho

BAIÃO, F., 1997, "Uma estratégia para o Projeto de Distribuição de Bases de Dados Orientadas a Objetos", Tese de Mestrado, COPPE/UFRJ, Brasil

BAIÃO, F. MATTOSO, M., 1998, "A Mixed Fragmentation Algorithm for Distributed Object Oriented Databases" , Anais do "International Conference on Computing and Information" (ICCI'98), Winnipeg, Canada, junho, pp.141-148

BAIÃO, F. MATTOSO, M. ZAVERUCHA, G., 1998, "Towards an Inductive Distributed Design of Object Oriented Databases", Anais do "Third IFCIS Conference on Cooperative Information Systems" (CoopIS'98), Nova York, EUA, agosto, pp.188-197

BAIÃO, F. MATTOSO, M. ZAVERUCHA, G., 2000, "Horizontal Fragmentation in Object DBMS: New Issues and Performance Evaluation", Proceedings of the "19th IEEE International Performance, Computing, and Communications Conference" (IPCCC 2000), IEEE CS Press, Phoenix, Arizona, February, pp. 108-114.

BENZAKEN, V., 1990, "An Evaluation Model for Clustering Strategies in the O_2 Object-Oriented Database System", In: Proceedings of the 3rd International Conference on Database Theory (ICDT'90), Paris, France.

BERTINO, E., 1990, "Query Optimization Using Nested Indices", In: Proceedings of the 2nd International Conference on Extending Database Technology (EDBT'90), Venice, Italy, *Lecture Notes in Computer Science 416*, Springer-Verlag.

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

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*, New York

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

CATTEL,R., 1994, *Object Data Management*, Addison-Wesley Publishing Company, revised edition

CATTEL. R. *et al.*, 2000, *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann Publishers Inc., San Francisco, USA

CLUET, S., DELOBEL, C., 1994, "Towards a Unification of Rewrite-based Optimization Techniques for Object-oriented queries". In: *Query Processing for Advanced Database Systems*, FREITAG, D., MAIER, D., VOSSEN, G., ed., Morgan Kaufmann Publishers, California, pp. 246-272

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

DEWITT,D. *et al*., 1990, "The GAMMA Database Machine Project", *IEEE Transactions on Knowledge and Data Engineering*, Vol 2, Março, pp 44-62

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, USA, pp. 172-181

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*, India, pp.390-400

GEPPERT, A., DITTRICH, K., GOEBEL, V., SCHERRER, S., 1990, "An Algebra for the NO^2 Data Model", Relatório Técnico, Institut für Informatik, Universität Zürich, Alemanha

GRUSER, J., 1996, "Modèles de coût pour l'optimisation de requêtes objet", Tese de Doutorado, Université Pierre & Marie Curie – Paris IV, Paris, França

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, pp. 312-321

KABRA, N., DEWITT, D., 1999, "OPT++: An Object-Oriented Implementation for Extensible Database Query Optimization", *VLDB Journal,* vol. 8(1), pp. 55-78

KEMPER, A., MOERKOTTE, G., 1995, "Physical Object Management", In: *Modern Database Systems: The Object Model, Interoperability, and Beyond*, KIM, W. (ed.), Addison-Wesley Publishing Company

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

LIMA, F. MATTOSO, M., 1996, "Performance Evaluation of Distribution in OODBMS: a Case Study with O2". In: *Proceedings of the IX Intl. Conf on Parallel & Distributed Computing Systems (PDCS'96)*, France, pp. 720-726

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

MOLINA, H., HSU, M., 1995, "Distributed Databases". In: Kim, W. (ed), *Modern Database Systems*, ACM Press, pp.484-485

O2 TECHNOLOGY, 1993, "The O2 User Manual", versão 4.3.5

OGASAWARA, E., MATTOSO, M., 1999, "Uma Avaliação Experimental sobre Técnicas de Indexação em Bancos de dados Orientados a Objetos". In: Anais do XIV Simpósio Brasileiro de Banco de Dados (SBBD'99), SBC, Florianópolis, Brasil, pp.285-298

OPTGEN, 1998, "The OPTGEN Optimizer Generator", In: http://ranger.uta.edu/~fegaras/optimizer, Department of Computer Science and Engineering, University of Texas at Arlington (consultado em Janeiro de 2000)

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., VALDURIEZ, P., 1999, *Principles of Distributed Database Systems*, 2nd edition (1st edition 1991) New Jersey, Prentice-Hall

SHAW, G., ZDONIK, S., 1990, "A Query Algebra for Object-Oriented Databases", In: Proceedings of the 6th International Conference on Data Engineering (ICDE'90), 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, pp.300-311

SHRUFI, A., 1994, "Performance of Clustering Policies in Object Bases", In: Proceedings of the 3rd Conference of Information and Knowledge Management (CIKM'94), Galtherburg, USA, pp. 80-87

STRAUBE, D., ÖZSU, M., 1991, "Execution Plan Generation for an Object-Oriented Data Model", In: Proceedings of the 2nd International Conference on Deductive Object-Oriented Databases, Springer-Verlag, pp. 43-67

SU, S., HUANG, Y., AKABOSHI, N., 1998, "Graph-Based Parallel Query Processing and Optimization Strategies for Object-Oriented Databases", *Distributed and Parallel Databases*, 6, pp. 247-285

TAVARES, F., 1999, "Avaliação do Processamento Paralelo de Consulta no Modelo Orientado a Objetos", Tese de Mestrado, 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 (SBBD'2000), SBC, João Pessoa, Brasil, pp.720-726

VALDURIEZ, P., 1987, "Join Indices", *ACM Transactions on Database Systems*, vol. 12(2), pp. 218-246

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