Universidade Federal do Ceará
Centro de Ciências
Departamento de Computação
Mestrado em Ciências da Computação

Dissertação de Mestrado

Construção de Otimizadores com Reescrita de Consultas para Sistemas Paralelos de Bancos de Dados Orientados a Objetos

Por
Carlo Giovano da Silva Pires
giovano@lia.ufc.br

Fortaleza-CE. 2001

Orientador: Dr. Javam de Castro Machado.

RESUMO

Sistemas de bancos de dados orientados a objetos (SGBDOO) têm se mostrado, devido ao seu modelo de dados mais expressivo, mais apropriados ao armazenamento de dados de aplicações não-convencionais do que os sistemas de bancos de dados relacionais. Por outro lado, os SGBDOO's apresentam baixo desempenho no processamento de consultas declarativas.

A diminuição dos custos de hardware e o avanço nas tecnologias de redes tornam cada vez mais viável o uso de máquinas paralelas no processamento de consultas declarativas em bancos de dados. Assim, o uso de paralelismo no processamento de consultas em SGBDOO's surge como uma boa alternativa para melhorar o tempo de resposta dessas consultas.

No entanto, a utilização de paralelismo em SGBDOO's aumenta a complexidade do módulo de otimização de consultas, vital para o bom desempenho no processamento de consultas. Com o objetivo de construir otimizadores para SGBDOO's paralelos, este trabalho propõe a utilização de um modelo de otimizações em duas fases construído através de um sistema de regras no contexto de sistemas de construção de otimizadores. Esses sistemas aumentam o grau de reutilização de técnicas de otimização e a extensibilidade do próprio otimizador.

O trabalho fornece uma metodologia baseada em uma abstração do modelo de regras e da estratégia de busca *top-down* para facilitar a implementação de sistemas de otimização com o uso dos diversos sistemas de construção de otimizadores disponíveis.

O modelo de otimização implementado propõe a seleção de operadores algébricos com o uso de regras baseadas em custos locais das operações e custos de comunicação de dados referentes ao paralelismo intra-operador. O trabalho também propõe um algoritmo para reordenação de operadores com base nos mesmos custos. Por fim, o modelo de otimização utiliza regras de extração de paralelismo inter-operador de forma transparente às outras regras.

A proposta é validada através da implementação da técnica com o uso de um sistema de construção de otimizadores e posterior construção de um otimizador para um SGBDOO. Para viabilizar essa implementação, o trabalho também fornece a análise de uma álgebra orientada a objetos com relação ao paralelismo intra-operador. Uma análise preliminar dos resultados é realizada a fim de demonstrar a viabilidade da proposta.

ABSTRACT

Object Oriented Database Management Systems (OODBMS) have shown, due to their more expressive data model, to be more appropriate to the data storage of non-conventional applications than relationaldatabase systems. By other means, OODBMS present low performance in declarative query processing. Thus, the use of parallelism in OODMBS's query processing turns out to be a good alternative for improving query response time.

However, the use of parallelism in OODBMS increases the complexity of the query optimization module, which is very important for a good performance in query processing. Aiming to build optimizers for parallel OODMBS, this work proposes the use of a two-phase optimization model within rule-based systems in the context of optimizer frameworks. These frameworks improve the reuse degree of optimization techniques and also the extensibility of the optimizer.

The work provides a methodology based on an abstraction of the rule model and on the top-down search engine, in order to build optimization systems using several framework proposals.

The implemented optimization model proposes the algebric operator's selection, using rules based on local and data communication costs relative to intra-operator parallelism. The work also proposes an algorithm for operator reodering based on the same costs. Finally, the optimization model uses parallelism extraction rules for inter-operator parallelism in a transparent way to the other modules.

The proposal is validated through the implementation of the technique, using an optimizer framework and the subsequent construction of an OODBMS optimizer.

In order to implement the proposal, the work also provides an analysis of an object oriented algebra, regarding intra-operator parallelism. A preliminary analysis of the result is done in order to demonstrate the proposal's viability.

Agradecimentos

A Deus, pois sem Ele nada seria possível.

Aos meus pais, José Sales Pires e Lucia Alves da Silva, por todo carinho, atenção, educação e apoio que têm me dado desde os meus primeiros dias de vida.

A minha esposa, Gabriela Telles de Souza, por todo apoio e paciência durante todos os momentos e principalmente durante as longas horas de trabalho ao meu lado em Lisboa. Pelas suas inestimáveis revisões do trabalho e compartilhamento de seus conhecimentos sobre bancos de dados.

Ao meu filho, Luca de Souza Pires, que com seu sorriso e alegria me mostra a cada dia um grande motivo pelo qual continuar a caminhada.

Ao Dr. Javam de Castro Machado, por sua contribuição e valiosa revisão do trabalho na qualidade de orientador; por ter me apoiada no retorno ao mestrado e sempre acreditado na minha determinação e capacidade de desenvolver um bom trabalho.

Aos componentes da banca examinadora, Dr. Geovane Magalhães, Dr. Riverson Rios e Dr. Javam Machado, por terem aceito fazer parte do exame e pelas sugestões no intuito de melhorar a qualidade do trabalho.

Aos meus irmãos, Andréa Sales, Rejane Pires, Marcelo Pires e Sales Filho, que sempre fizeram companhia a mim e a minha família, mesmo separados por um oceano.

Aos meus sogros, José Tarcísio e Maria Bruhilda, pela disposição em ajudar a minha esposa e minha família na difícil tarefa de estar longe da pátria.

A bolsista Elaine Sampaio, por sua disposição em montar e manter um ambiente de prototipação e testes, fundamental para o desenvolvimento desse trabalho.

A todos os meus amigos, que mesmo espalhados ao redor do mundo sempre estiveram presentes para ouvir e apoiar.

SUMÁRIO

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Sistemas de bancos de dados relacionais têm se mostrado bastante eficientes para aplicações tradicionais, utilizando um modelo de processamento de consultas maduro e muito bem fundamentado. Contudo, o modelo relacional não possui expressividade suficiente para representar estruturas complexas, como dados multimídia, informações geográficas, CAD/CAM, armazenamento de objetos CORBA ou dados em formato XML. O modelo orientado a objetos vem sendo aplicado com sucesso em linguagens de programação e tem ganho bastante força com a padronização de linguagens para modelagem de objetos através da UML [BRJ97], bem como através de processos de desenvolvimento orientados a objetos. Esses fatos, além da necessidade da representação de tipos complexos nos bancos de dados, impulsionaram o surgimento de duas propostas que utilizam modelos mais representativos: sistemas gerenciadores de bancos de dados objeto-relacionais (SGBDOR) e sistemas gerenciadores de bancos de dados orientados a objetos (SGBDOO).

Os SGBDOR's são baseados em extensões sobre o modelo relacional. Estas extensões permitem uma diminuição das diferenças semânticas existente entre as linguagens de programação orientadas a objeto e os bancos relacionais. Em um sistema objeto-relacional, é possível a criação de tipos abstratos e a utilização de métodos fornecendo suporte ao encapsulamento. Essa proposta possui uma boa aceitação no mercado, já que utiliza um modelo híbrido que pode dar suporte a aplicações construídas sobre o modelo relacional e possui a linguagem de consulta SQL3 [ISO96a,b] bastante similar ao padrão SQL tradicional. Por outro lado, os SGBDOO's implementam o modelo de objetos de forma pura, dando amplo suporte ao encapsulamento e a relacionamentos semânticos entre objetos. A integração com as linguagens de programação é completa devido à utilização de um modelo comum. O *Object Database Management Group* (ODMG)

[CB97] define a ligação (*binding*) direta dos objetos da base de dados para objetos de aplicações Java, C++ e SmallTalk. Fornecedores de compiladores e de SGBDOO's como *Sun MicroSystems*, *Computer Associates, Object Design, Versant* e *POET* fornecem soluções de *binding* para Java de forma que o desenvolvedor não precisa escrever código para armazenar os objetos da aplicação. Sistemas objeto-relacionais não fornecem suporte para *binding* e assim o esforço para mapear esses objetos em estruturas da base de dados pode consumir até 30% do esforço total de codificação. Além disso, como o código para armazenamento e recuperação do estado do objeto faz parte do código da aplicação, isso pode resultar em problemas de portabilidade com relação ao SGBD utilizado.

O ODMG padronizou uma linguagem de consulta declarativa, denominada *Object Query Language* (OQL) [CB97], para sistemas de bancos de dados orientados a objeto. No entanto, o mesmo não ocorre com relação à álgebra para esses sistemas. Várias propostas foram apresentadas, mas ainda não se chegou a nenhuma padronização. Ademais o problema relacionado à eficiência na execução da consultas, tanto em SGBDOO's como em SGBDOR's, continua como fator chave para a aceitação e o sucesso desses sistemas.

Recentemente, o grande número de aplicações desenvolvidas para a *Internet* e a utilização de aplicações distribuídas surgiram como um novo impulso para SGBDOO's e servidores paralelos. As aplicações distribuídas vêm utilizando padrões baseados em objetos tais como CORBA [OHE96] ou DCOM [OHE96] e, assim, SGBD's que fornecem suporte a modelo de objetos são candidatos naturais a repositório de objetos. A especificação ODMG indica que os SGBDOO's possam ser utilizados pelo barramento CORBA como gerenciador de objetos, sendo assim responsáveis por tarefas como ativação e desativação do estado do objeto no barramento, bem como a chamada de métodos remotos.

Por outro lado, as aplicações para internet necessitam de:

- Suporte a um grande número de usuários e grande carga sobre servidores de aplicação e de bancos de dados.
- Suporte a armazenamento de grandes quantidades de conteúdos complexos tais como imagens, vídeos, sons, XML[W3C] e HTML.

- Funcionalidades concentradas sobre consultas e navegação de conteúdos.
- Maior integração entre os repositórios de dados e tecnologias baseadas em objetos (por exemplo, Java, EJB e COM), que cada vez mais vêm sendo utilizadas para o desenvolvimento desse tipo de aplicação.

A diminuição dos custos de *hardware*, principalmente em dispositivos de armazenamento de dados primário e secundário, juntamente com o avanço nas tecnologias de redes e arquiteturas distribuídas tornam cada vez mais viável o uso de servidores com arquiteturas paralelas. Esses servidores têm como características principais o alto desempenho e a disponibilidade. Apoiados nesse tipo de arquitetura, os sistemas paralelos de bancos de dados surgem como uma alternativa para melhorar o desempenho de consultas. Assim, a combinação de sistemas paralelos com o modelo orientado a objeto apresenta-se como uma boa alternativa para responder aos requisitos dos tipos de aplicações aqui discutidas.

1.1 Processamento de Consulta e SGBD's paralelos

Apesar das vantagens mencionadas com o uso dos SGBDOO's, ainda faz-se necessário melhorar o desempenho desses sistemas. Esse problema ocorre devido a fatores como a avaliação de expressões de caminhos, recuperação de objetos complexos no disco, atributos definidos sobre tipos abstratos e métodos. Além disso, o modelo orientado a objetos fornece suporte a diversos tipos de coleções como *bags, sets, lists* e *arrays* e, dessa forma, as operações de uma consulta podem envolver o processamento de diferentes tipos de coleções. Uma operação de junção pode, por exemplo, receber uma coleção *bag* e uma *list* como entrada e fornecer um *array* como saída. O cálculo do custo de execução dos métodos pode ser uma tarefa complexa, dado que os métodos podem encapsular outras consultas OQL, ou ainda ser escritos em linguagens de programação externas como C++ ou Java.

Outro problema é que os SGBDOO's disponíveis no mercado não exploram o paralelismo no processamento de consultas. A própria natureza complexa dos objetos sugere que o estado desses objetos poderia ficar fragmentado em diversos nós de processamento e recuperados de forma paralela. O uso de paralelismo no processamento de consultas pode fornecer melhoria considerável

no tempo de resposta de consultas. No entanto, um novo conjunto de dificuldades é adicionado ao processamento de consultas com a utilização do paralelismo, principalmente na fase de otimização do plano de execução da consulta. Em um SGBD paralelo, o espaço de busca de planos alternativos aumenta consideravelmente devido às diversas possibilidades de fragmentação de dados e de alocação de operadores aos diferentes nós de processamento disponíveis.

1.2 Sistemas de construção de otimizadores e paralelismo

O conceito de sistemas para construção de sistemas (*frameworks*) é baseado na reutilização de arquitetura e projeto em larga escala. Esses sistemas podem adotar uma abordagem de geração de código ou de composição através do paradigma da orientação a objetos. Com a abordagem de geração de código, o sistema recebe uma especificação de alto nível e retorna o código fonte do sistema a ser construído. Já com a abordagem orientada a objetos, o sistema é construído através da criação de novas classes obedecendo a uma hierarquia pré-determinada. Em um passo posterior, as novas classes e as classes pré-existentes são compiladas conjuntamente para formar um novo sistema.

Dada a complexidade de um sistema de otimização de consultas para bancos de dados, a abordagem de *frameworks* tem sido proposta com o objetivo de melhorar a extensibilidade e facilitar a construção de otimizadores. A extensibilidade deve ser obtida em relação a álgebra utilizada, regras de transformação, algoritmos para execução de operadores da álgebra e estratégias de busca do plano ótimo. Esses sistemas surgiram sob o contexto de bancos de dados relacionais com o objetivo de facilitar e abstrair a construção de otimizadores para as diferentes álgebras existentes.

Algumas dessas propostas, como [BMG93] e [Feg97a], já foram utilizadas para construir otimizadores para SGBDOO's. Contudo, esses trabalhos não abordam a utilização dessas ferramentas na construção de otimizadores para sistemas paralelos de bancos de dados. Em [Sam98] é apresentada uma primeira

abordagem na utilização de sistemas de construção de otimizadores[1] no contexto de SGBDOO's paralelos. O trabalho concentra-se na extração de paralelismo com a utilização de regras para fatorar a álgebra física utilizada em um SGBDOO seqüencial em uma álgebra paralela de granulidade fina. Essa álgebra paralela facilita a identificação de oportunidades de paralelismo inter-operador. No entanto, permanece em aberto a utilização de *frameworks* de otimizadores para a construção das fases de reordenação de operadores, reescrita de consulta e seleção de operadores em sistemas paralelos orientados a objetos. A proposta apresentada neste trabalho cobre todos esses aspectos.

1.3 Objetivos e Contextualização

O problema de otimização de consultas é *NP-Hard* [BB96] até mesmo para sistemas não-paralelos. No contexto de SGBD's paralelos, o espaço de busca e a complexidade são ainda maiores. Com o objetivo de diminuir a complexidade da otimização de consultas em sistemas paralelos, um modelo de otimização em duas fases foi proposto em [HS91]. Esse modelo diminui o espaço de busca do processo de otimização para sistemas paralelos utilizando na primeira fase uma abordagem de otimização para planos de sistemas seqüenciais. A segunda fase utiliza os planos gerados na primeira fase para gerar o plano paralelo final. Essa abordagem, apesar de reduzir muito a complexidade do problema, pode não conduzir a planos ótimos por restringir o espaço de busca inicial a planos para sistemas não-paralelos.

Em [Has96], o modelo de duas fases é estendido para considerar os custos do paralelismo na seleção e ordenação de operadores físicos logo na primeira fase, proporcionando um espaço de busca inicial para sistemas paralelos. Para isso, algoritmos de seleção e ordenação de operadores foram apresentados e o autor sugere que esses algoritmos podem ser utilizados com a modificação dos otimizadores convencionais, ou através da adição dos algoritmos em uma fase posterior.

[1] Os termos “sistemas de construção de otimizadores” e “frameworks” serão utilizados no trabalho como sinônimos

Este trabalho propõe uma extensão do modelo utilizado em [Has96], através da utilização de regras no contexto de *frameworks* de otimizadores para a seleção de operadores físicos. Essa abordagem facilita a extensibilidade do otimizador e a reutilização da técnica sobre diferentes contextos. Além disso, a proposta é desenvolvida sobre o processo de otimização de consultas OQL [CB97] para SGBDOO's. Esse fato fornece duas vantagens imediatas: Aborda o problema de otimização e paralelismo em SGBDOO's como modo de melhorar o desempenho de tais sistemas, e pode ser facilmente utilizado em sistema relacionais ou objeto-relacionais que utilizam as linguagens declarativas SQL ou SQL3. Quanto ao problema de reordenação de operadores, o trabalho estende o algoritmo de ordenação de operadores utilizado em um SGBDOO seqüencial e proposto em [Feg98]. A extensão realizada considera os custos inerentes ao paralelismo, obtendo um algoritmo com complexidade menor do que o proposto em [Has96].

O trabalho vai ainda mais além, quando propõe uma metodologia para especificação de otimizadores em sistemas de construção de otimizadores. Assim, o modelo de otimização pode ser também reutilizado no meta-nível, ou seja, a nível dos sistemas que são utilizados para a construção de sistemas de otimização.

As idéias apresentadas são validadas através da utilização da metodologia no sistema de construção de otimizadores OPTGEN [Feg97b] e através da construção de um otimizador para consultas com paralelismo a partir do otimizador original do SGBDOO Lambda-DB [FSRM00].

1.4 Organização da Dissertação

No capítulo 2, são apresentados os conceitos sobre processamento de consultas, suas fases e elementos, como modelos de custo e estratégia de busca. Além disso, são apresentadas as principais características e problemas investigados em bancos de dados orientados a objetos, principalmente no que se refere ao processamento de consultas.

O capítulo 3 fornece conceitos sobre arquiteturas e servidores paralelos de banco de dados. Aborda-se também o processamento de consultas no contexto

de sistemas paralelos, enfatizando-se os tipos de paralelismo de consultas e as questões relativas ao particionamento de dados que viabiliza o processamento paralelo. Outro tópico que merece destaque no capítulo é a otimização de consultas em sistemas paralelos. Finalmente, são discutidas questões relativas a arquiteturas paralelas para SGBDOO e apresenta-se a álgebra física paralela orientada a objetos utilizada no trabalho.

No capítulo 4, é apresentada a metodologia proposta no trabalho para construção de otimizadores para sistemas paralelos de banco de dados orientados a objetos. Essa metodologia utiliza como base o conceito de sistemas para construção de sistemas (*frameworks*) de otimização de consultas. Assim, no início do capítulo, apresenta-se e fornece-se uma análise das principais propostas de *frameworks* de otimizadores. A metodologia proposta é dividida em duas grandes fases: análise e especificação. O capítulo detalha a primeira fase, que tem o enfoque sobre a análise da álgebra paralela e transformações no espaço de busca.

A fase de especificação é abordada no capítulo 5. Essa fase utiliza os produtos gerados na fase de análise e caracteriza-se por seu caráter mais próximo da implementação da técnica apresentada. Assim, o capítulo aplica a metodologia a um *framework* específico, exemplificando a sua utilização. Finalmente, uma série de consultas OQL, juntamente com os respectivos planos de execução e quadros comparativo, é apresentada. Os planos de execução foram gerados pelo otimizador construído com a metodologia. Os quadros comparativos apresentam os tempos de otimização e execução do otimizador original de um SGBDOO, contra os tempos inerentes ao otimizador de consultas para sistemas paralelos construído com o uso da metodologia.

Capítulo 2

Processamento de Consulta e Sistemas de Banco de Dados Orientados a Objeto

2.1 Introdução

O processamento de consultas consiste em transformar uma consulta definida em uma linguagem de manipulação de dados (DML) em um plano de execução e em seguida executar esse plano sobre o conteúdo na base dados. Essa transformação deve fornecer corretude, de forma que o plano de execução final devolva os resultados requeridos na consulta de alto-nível, e eficiência, de forma que o tempo decorrido nesse processo não comprometa o tempo de resposta da consulta.

Este capítulo apresenta uma descrição das fases envolvidas no processamento de consultas, descrevendo-se os aspectos relevantes neste trabalho e concentrando-se na fase de otimização de consultas e seus elementos como modelo de custos, espaço e estratégias de busca. Apresenta-se, também, uma descrição sobre características específicas do modelo orientado a objetos e que devem ser observadas atentamente pelo processador de consultas. Descreve-se, também, a linguagem de manipulação de dados OQL [CB97], adotada como padrão para sistemas de banco de dados orientados a objetos.

2.2 Processamento de consultas

2.2.1 Introdução

Inicialmente, uma consulta fornecida pelo usuário em uma linguagem de alto nível é interpretada pelo sistema e transformada em uma expressão definida sobre uma álgebra, que é uma forma intermediária entre a consulta em alto-nível e o plano final de execução (veja a Figura 2.1). Durante essa fase, denominada de *Parsing e Tradução da Consulta*, a consulta passa por uma análise sintática e semântica. Essa análise determina, por exemplo, se a consulta utiliza as cláusulas da linguagem corretamente, e até, se os métodos e predicados podem ser aplicados a objetos definidos na base de dados. Até o final da fase, a consulta deve ser transformada de sua representação textual em uma expressão algébrica consistente. Essa expressão serve de entrada para o otimizador de consultas, que por sua vez, gera expressões em álgebra lógica e física, equivalentes quanto ao resultado, mas diferentes com relação ao custo e tempo de execução. Finalmente, a melhor expressão em álgebra física (plano de execução) é executada, retornando os dados requisitados pelo usuário.

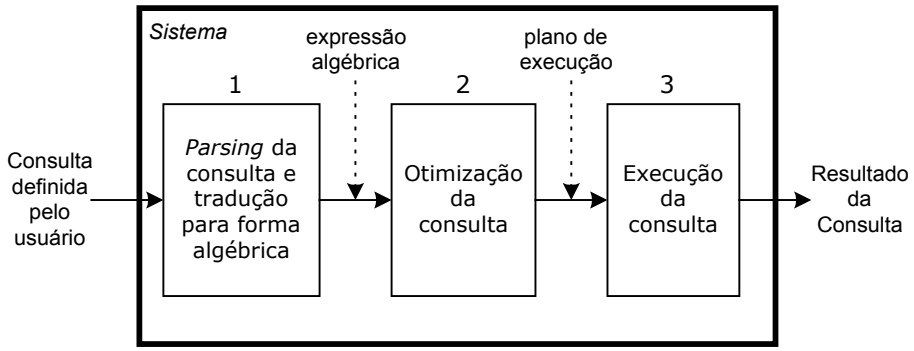


Figura 2.1 – Fases do processamento de consultas.

A seguir, dada a relevância da fase para o trabalho, são apresentados mais detalhes sobre a fase de otimização de consultas.

2.2.2 Otimização de Consultas

O objetivo do otimizador de consultas é escolher o melhor plano de execução, através de transformações sobre a expressão algébrica fornecida. Essas transformações são baseadas em propriedades da álgebra como, por exemplo, a transitividade de operadores na seleção de algoritmos para implementação dos operadores. Por exemplo, um operador lógico *join* pode ser implementado pelos métodos *nested loop* ou *sort merge* (ou outros métodos) dependendo da ordenação das tuplas que servem de entrada para o mesmo.

Pode-se dividir as metodologias de otimização em *Otimização Algébrica* e *Otimização Baseada em Custos*. No primeiro caso, a expressão a ser otimizada é transformada através de regras guiadas por heurísticas que supostamente geram expressões equivalentes mais eficientes. Uma exemplo bem conhecido de tais heurísticas é executar operações de seleção antes de operações de junção. A seguir, características da base de dados, como a existência de índices e estatísticas, são consideradas para a seleção de algoritmos para a implementação de cada operador.

A otimização baseada em custos é bastante similar ao processo descrito acima, no entanto, as transformações não são baseadas apenas em heurísticas. O otimizador utiliza funções de estimativa de custos para avaliar se uma transformação conduz a um bom plano ou não. Uma mesma consulta pode ter vários planos de execução equivalentes com relação ao resultado, mas com diferentes custos de execução. Entre todos esses planos o otimizador de consultas deve escolher o plano de menor custo. Em bancos de dados centralizados, o custo de um plano é formado por dois componentes: custo de E/S e custo de CPU. Contudo, devido à grande quantidade de planos equivalentes existentes para uma dada consulta, o otimizador deve utilizar uma estratégia de busca para a geração dos planos e para avaliar o melhor plano com relação a um dado modelo de custo. Em geral, essa abordagem conduz a melhores planos que a otimização algébrica pura.

Como visto anteriormente, o otimizador recebe como entrada uma expressão algébrica consistente. Essa expressão é representada como uma árvore de operadores (Figura 2.2 - a). Essa árvore possui como nós os operadores lógicos

da álgebra. Cada nó possui zero ou mais (geralmente um de máximo dois) operadores lógicos como entrada. A árvore de operadores determina a ordem em que os operadores são executados, dado que para executar o operador topo, suas entradas devem ser executadas antes. Quando o otimizador seleciona os operadores físicos (algoritmos) para implementação dos operadores lógicos e a árvore passa a ser formada exclusivamente por nós com operadores físicos (Figura 2.2 - b), tem-se então um *Plano de Execução de Consultas* (PEC). Esses planos especificam como a consulta deve ser executada, determinando a ordem de execução dos operadores e o algoritmo que deverá implementar cada um deles. Um plano pode também receber anotações, como por exemplo, o atributo de particionamento dos dados no caso de bancos de dados paralelos. É o caso do operador BLOCK_NESTED_LOOP no exemplo, que é processado com os dados particionados de acordo com o predicado da junção, no caso o identificador dos objetos existentes em *Departments*.

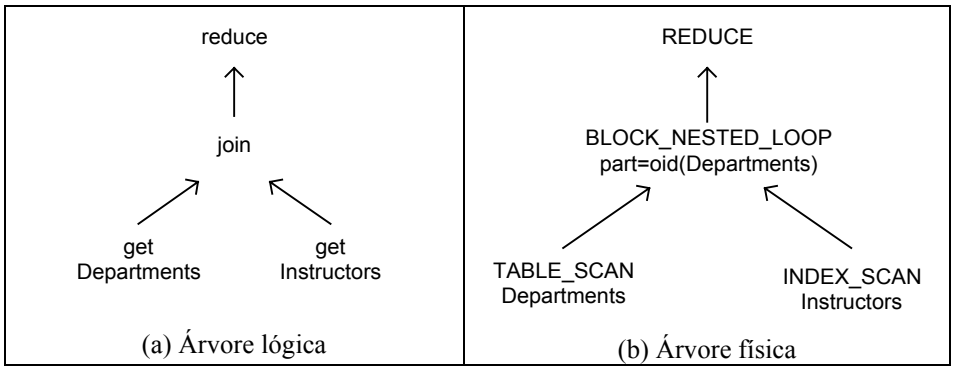


Figura 2.2 - Árvores de Operadores

2.2.3 Componentes de um otimizador de consultas

Um otimizador de consultas é formado por três componentes: Espaço de busca, estratégia de busca e modelo de custos.

Dá-se o nome de *Espaço de Busca* ao conjunto de árvores de operadores equivalentes, lógicas ou físicas, e que podem ser gerados a partir da árvore lógica de operadores inicial com o uso de regras de transformação. Devido ao grande número de expressões equivalentes que se pode gerar a partir da

expressão inicial, o espaço de busca tende a crescer exponencialmente. Assim, o custo para selecionar o melhor plano de execução dentro de um espaço de busca muito grande pode ser proibitivo de forma que o tempo de otimização acabe por exceder o próprio tempo de execução.

Para restringir o espaço de busca e tornar o tempo de otimização aceitável, faz-se necessária a utilização de uma estratégia de busca. A estratégia de busca mais utilizada é a *Programação Dinâmica*. Essa estratégia foi utilizada inicialmente no System R [SAC79] e continua a ser utilizada em grande parte dos SGBD's comerciais, principalmente no que se refere a funções para estimativas de custos. Essa estratégia de busca é realizada de forma *bottom-up*, ou seja, a otimização inicia-se pelas folhas da árvore, os melhores planos são encontrados a esse nível e passados para o nível seguinte. Os níveis mais altos utilizam os custos e tamanho dos resultados estimados nos níveis inferiores para estimar seu custo e ganho de resultados. Esse processo continua até se chegar à raiz da árvore. Para evitar o crescimento exponencial do espaço de busca, o *System R* utiliza heurísticas para descartar certos tipos de expressões. No entanto, essa estratégia pode não conduzir a planos ótimos devido às heurísticas não serem baseadas em custos e utilizarem apenas informações lógicas para descartar os planos.

O sistema *StarBurst* [Loh88] estendeu o System R através do uso de regras de transformação baseadas em produção gramatical para a geração do espaço de busca. No entanto foi o sistema Exodus [GD87] que propôs pela primeira vez o uso de uma estratégia *Top-down* para otimização de consultas. A otimização *Top-down* ajuda a evitar o problema da eliminação de expressões que poderiam participar do plano ótimo que ocorre no processo *Bottom-up*. Essa estratégia começa a otimização a partir da raiz e envia para os níveis inferiores informações que determinam propriedades que as sub-expressões devem fornecer para os níveis mais altos. Os níveis mais altos utilizam as informações sobre custo e tamanho do resultado, como também outras propriedades físicas, para estimar suas propriedades. Assim, os níveis mais altos são "virtualmente" otimizados antes dos níveis inferiores. O que ocorre na verdade é que esse processo se repete até as folhas da árvore. Quando o processo chega às folhas,

inicia-se um processo *Bottom-up*, que utiliza também as informações enviadas pelo nível superior para selecionar os planos. Dessa forma, mesmo que o plano não pareça atrativo com relação a heurísticas ou até mesmo quanto ao custo, ele será selecionado se for necessário para construção de um plano promissor no nível superior. Em [SMB01] foi demonstrado que, apesar de tender a gerar mais expressões, a estratégia *Top-down* pode atingir melhores resultados que a *Bottom-up*.

Sistemas recentes como Cascades [Gra95] e OPTGEN [Feg97b] utilizam a estratégia *Top-down* para a construção de otimizadores para bancos de dados comerciais e orientados a objetos.

O terceiro componente do otimizador é o modelo do custo. É através desse modelo que o otimizador pode escolher, dentro do espaço de busca, qual o plano mais eficiente. O custo deve refletir o nível de utilização por parte do plano de recursos como tempo de CPU, custo de E/S, utilização de memória e custos de comunicação de acordo com a arquitetura utilizada. Como dito anteriormente, o otimizador do System R foi o primeiro a utilizar funções para estimativa de custos. Essas funções são chamadas durante o processo de otimização e utilizam informações estatísticas da base de dados como o número de tuplas existentes nas tabelas e o nível de utilização de memória de operações como junções. O modelo de custos deve fornecer valores próximos da realidade, para que o otimizador escolha bem os planos, e deve computar esses valores de forma eficiente para não degradar o tempo de execução do próprio processo de otimização.

2.3 Bancos de dados orientados a objetos

2.3.1 Características do modelo orientado a objetos

SGBDOO's fornecem um modelo de dados mais expressivo que o modelo relacional, permitindo, assim, que eles possam ser utilizados em uma maior variedade de aplicações. No entanto, essa característica por si só não garante a adoção desses sistemas. Para que eles possam ser realmente utilizados em aplicações com grande massa de dados, esses sistemas devem ser competitivos

com relação ao desempenho. Nesse aspecto, o processamento de consultas desempenha um papel chave na arquitetura do sistema.

Um SGBDOO difere-se dos sistemas relacionais devido ao seu modelo de dados utilizar os conceitos do paradigma orientado a objetos, introduzido pela linguagem de programação *Smalltalk*. Entre os conceitos utilizados por linguagens de programação e adotados no modelo de dados do SGBDOO, podemos citar herança, encapsulamento de dados, e o princípio da identidade do objeto.

Além dos conceitos herdados das linguagens, os SGBDOO's adicionam ao modelo características de servidores de dados, como a persistência dos objetos, gerenciamento de transações, controle de concorrência e suporte a linguagens de consultas.

No entanto, as características do modelo de objetos aumentam a complexidade dos sistemas de base de dados, que necessitam de técnicas específicas para obter o bom desempenho necessário para o sucesso dos SGBDOO's. Entre essas características podemos citar:

- O modelo orientado a objetos apresenta grande heterogeneidade, já que os operadores devem trabalhar com diversos tipos de coleções (sets, bags, lists, etc), diferentemente do modelo relacional que sempre trabalha com dados planos em forma de *tuplas*. Em [FM00] apresenta-se o cálculo *monóide* que trata as coleções de forma uniformizada.

- O encapsulamento dos dados através dos métodos dificulta o acesso a informações sobre o armazenamento do objeto. Além disso, avaliar o custo de execução de um método é uma tarefa bastante complicada, uma vez que esses métodos podem ser escritos em linguagens externas como C++ ou Java. Mesmo que os métodos sejam escritos em linguagem nativa do SGBD, surge, também, o problema de como otimizar o acesso aos objetos através dos valores retornados por métodos, já que os índices são construídos sobre os valores dos atributos.

- Objetos podem ser compostos por outros objetos ou coleções de objetos, que por sua vez podem ser compostos por outros objetos. Através de expressões de caminhos que permitem a navegação sobre essa estrutura composicional, consultas complexas podem ser formuladas pelos usuários de forma a evitar as várias junções comuns ao modelo relacional. Contudo, essa flexibilidade introduz grande complexidade ao processamento da consulta, já que esses caminhos devem ser avaliados de forma a serem transformados em operações que demandem o menor custo de processamento possível. Tipos especiais de índices para expressões de caminho, como *Nested Index* e *Path Index* [Ber94], podem ser utilizados para otimizar a recuperação de informações. Outra técnica utilizada para a otimização de consultas com essa característica é a utilização de junções com ponteiros (*pointer joins*) [FM00] para a avaliação de expressões de caminho.

- A composição de objetos por outros mais simples causa também o problema da montagem de objetos em memória. Como os métodos operam sobre o estado dos objetos, uma simples chamada a um método pode gerar buscas no disco através das partes dos objetos, sem garantia de que esses fragmentos estão armazenados em espaços contíguos do disco. Isso exige que os SGBDOO's forneçam políticas eficientes para o agrupamento do estado do objeto no disco (*clustering*) e a leitura desses objetos, como apresentado em [TN92] e [GHLZ94]. Um modo eficiente de se montar é processá-los em conjunto (set-oriented) ao invés de tratar um objeto por vez (object-at-a-time). Graefe propõe em [KGM91] um operador, chamado *assembly*, que utiliza informações físicas e lógicas para a recuperação e montagem de objetos de modo orientado a conjuntos. A utilização de índices de estrutura hierárquica [XH94] ajuda a otimizar a montagem de objetos.

- A necessidade de desaninhar expressões em SGBDOO's é bem mais freqüente do que em SGBDR's, devido à maior liberdade para a construção de consultas aninhadas na linguagem OQL e ao caráter composicional dos objetos, que permite que um objeto seja composto por outros objetos. Em [Feg98]

apresenta-se um algoritmo para desaninhamento de consultas e um operador algébrico para representação do algoritmo.

2.3.2 O modelo ODMG

Na Figura 2.3 temos um esquema UML baseado no exemplo ODMG apresentado em [CB97] e utilizado na base de dados de *benchmarks* do Lambda-DB [FSRM00].

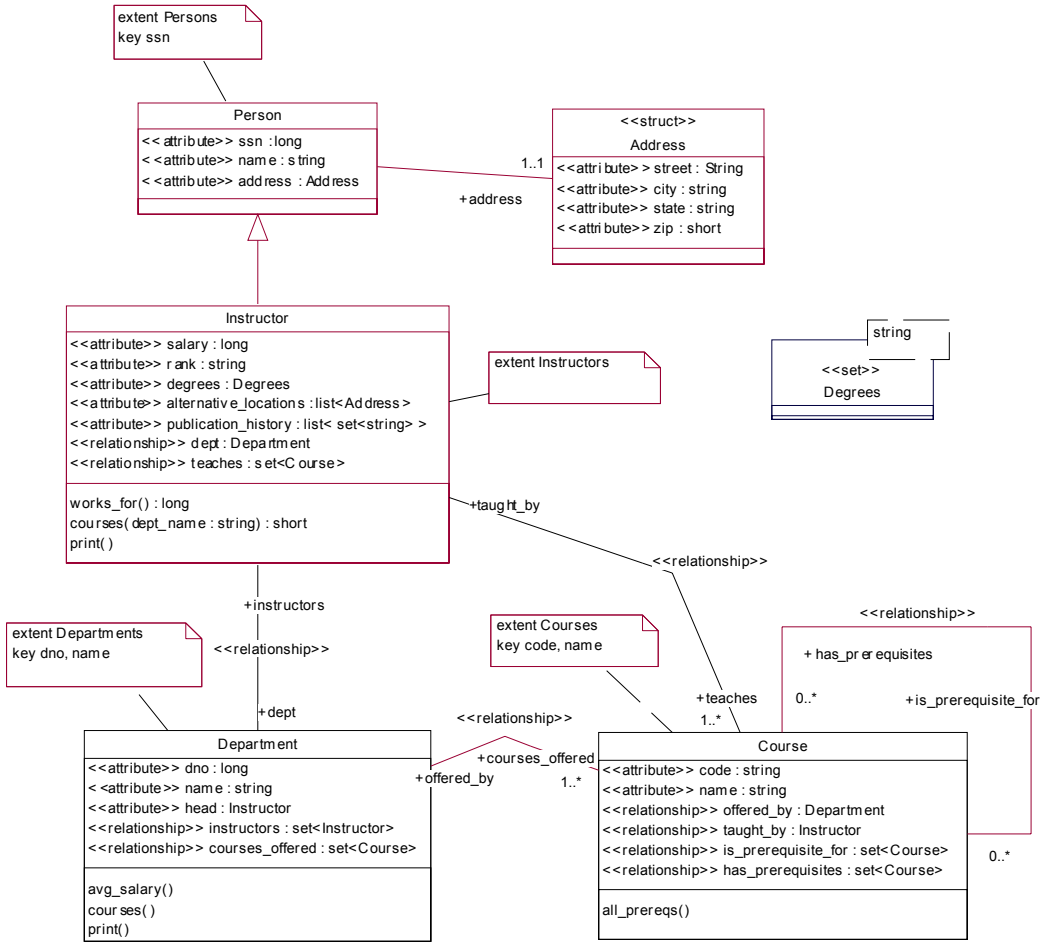


Figura 2.3 - Esquema Exemplo ODMG

O exemplo apresenta um esquema de uma aplicação orientada a objetos para armazenar professores, departamentos, cursos e seus relacionamentos. A classe *Instructor* é um subtipo de *Person* e assim herda seus atributos e o relacionamento com a estrutura de endereços (*Address*). Um objeto da classe *Instructor* está alocado a um departamento (*Department*) e pode ministrar vários cursos (*Course*).

As classes também apresentam operações como *Instructors::Works_for* (retorna o identificador do departamento no qual o instrutor está alocado) e *Department::courses* (retorna a lista dos cursos alocados a um departamento). Nesse esquema podem-se visualizar os principais elementos do modelo ODMG.

Atributos que representam relacionamentos (*relationships*) com cardinalidade superior a um são representados como coleções. Essas coleções podem ser:

Set: Uma coleção de objetos sem ordenação e que não permite duplicação de elementos.

Bag: Uma coleção de objetos sem ordenação e que permite duplicação de elementos.

List: Uma coleção de objetos com ordenação e que permite duplicação de elementos.

Array: Uma coleção com um número fixo de elementos que podem ser localizados pela sua posição.

Atributos (*attribute*) definem o estado de um objeto, já os relacionamentos (*relationship*) são definidos entre dois tipos e utilizam declarações de caminhos de travessia para possibilitar o uso das conexões lógicas existentes entre os objetos dos tipos envolvidos. Por exemplo, na Figura 2.3, temos um relacionamento entre as classes *Intructor* e *Course* que dará origem a uma coleção *teaches* aninhada em *Instructors*. Já *salary* e *rank* são exemplos de atributos de *Instructor*.

Uma extensão (*extent*) de uma classe é uma coleção de todas as instâncias da classe dentro de uma determinada base de dados. Por exemplo, na Figura 2.3, temos a extensão *Persons* (indicada por uma nota UML) para a classe *Person*. Por algumas vezes, as instâncias individuais da classe podem ser identificadas unicamente por valores existentes nos seus atributos. Esses atributos, conhecidos como chaves candidatas no modelo relacional, são denominados simplesmente de chave (*key*). No entanto, eles não funcionam como identificador de objeto, papel desempenhado pelo *Object_Id* e mantido internamente pelo SGBD. Por exemplo, na Figura 2.3, podemos observar que a classe *Department* apresenta *dno* e *name* como chave.

2.3.3 A linguagem de consultas OQL

A linguagem de consulta é a principal interface do usuário com o sistema de banco de dados e seu processador de consultas. A linguagem SQL vem sendo utilizada em sistemas relacionais com grande sucesso. Esse tipo de linguagem utiliza um formato declarativo, no qual o usuário informa os dados que deseja obter juntamente com condições de seleção da informação. Assim, o usuário não precisa determinar como a informação será obtida, como ocorre em linguagens procedimentais.

A linguagem de consultas padrão para SGBDOO é a *Object Query Language* (OQL). Essa linguagem segue os mesmos princípios do SQL, mas apresenta características específicas que permitem explorar a expressividade do modelo orientado a objetos. Na verdade, essa linguagem é um super-conjunto do SQL, de forma que qualquer consulta SQL que execute sobre tabelas relacionais pode ser executada com a mesma sintaxe e semântica sobre coleções de objetos. Uma característica poderosa em OQL é que todo resultado de consulta possui um tipo pertencente ao modelo de tipos ODMG, e assim esse resultado pode ser consultado por outra instrução OQL. Diferentemente de SQL, a linguagem OQL não define operadores explícitos para atualizações. Ao invés disso, utiliza os métodos definidos nos objetos para esse propósito.

Considere a consulta OQL abaixo, definida sobre o esquema da Figura 2.3, para que se possa exemplificar algumas das características da linguagem. Essa consulta recupera o nome dos instrutores que trabalham para o departamento 1, juntamente com o número de cursos oferecidos pelo departamento.

```
select struct(E: e.name,
              S: count(select c.code
                       from c in e.dept.courses_offered))
from e in Instructors
where e.works_for = 1
```

A linguagem permite o uso de construtores como *struct, set, list, bag* para estruturar o resultado da consulta. Nesse exemplo, usa-se o contrutor *struct*. O atributo *S* da estrutura é criado a partir do uso da função de agregação *count*

sobre o resultado de uma consulta aninhada. Essa consulta aninhada, por sua vez, utiliza a expressão de caminho `e.dept.courses_offered` para associar os cursos ao instrutor através do departamento onde ele trabalha. As variáveis *e* e *c* são denominadas variáveis de intervalo (*range variables*) pois são utilizadas para referenciar os objetos que pertencem ao tipo referenciado na expressão. Finalmente, no predicado da consulta utiliza-se o método *work_for* para retornar o código do departamento para o qual o instrutor trabalha.

2.3.4 Álgebra orientada a objetos

Para suportar características específicas do modelo orientado a objetos e formalizar a semântica da linguagem OQL, faz-se necessário o uso de um cálculo e uma álgebra específicos para o modelo orientado a objetos. Normalmente têm-se dois tipos de álgebras: Álgebras Lógicas e Álgebras Físicas. Uma álgebra lógica define a semântica da operação, enquanto a álgebra física define o método (algoritmo) de execução de uma operação. No entanto, ao contrário do que ocorre com a linguagem de consulta OQL, não existe uma padronização de uma álgebra para tal modelo. Algumas propostas podem ser encontradas em [SZ90], [Van93], [Ste95]. Essa falta de padronização dificulta o aproveitamento das técnicas de execução de consultas desenvolvidas sobre diferentes álgebras. Apesar da falta de padronização, vários pontos comuns são encontrados entre as propostas, entre eles podemos citar: suporte a herança, identificadores de objetos, tratamento de coleções e montagem de objetos.

Entre os operadores encontrados nas propostas existentes, este trabalho aponta os mais comuns e essenciais. Esses operadores podem ser divididos em dois grandes grupos:

- Operadores para recuperação de informação: Leitura em disco (*select/get*), Junção (*join*), União (*union*), Materialização (*materialize*).
- Operadores para estruturação da informação: Planificação de dados (*flatten*), Aninhamento – criação de coleções aninhadas em atributos (*nest*), Desaninhamento – extração de coleções aninhadas para forma normalizada (*Unnest*).

Para exemplificar os principais pontos necessários a uma álgebra orientada a objetos, essa seção apresenta a álgebra lógica monóide definida em [Feg97a], [FM00] e utilizada neste trabalho. A álgebra física será apresentada no capítulo 4, mas no contexto do processamento paralelo que é o foco principal do trabalho.

A álgebra monóide tem entre suas principais características a uniformização da representação dos resultados e dos parâmetros de cada operação de uma consulta. Essa álgebra é uma proposta recente e fornece suporte ao modelo de objetos ODMG [CB97]. Ela é utilizada no SGBDOO Lambda-DB, e representa um bom *framework* para otimização de consultas em SGBDOO's como demonstrado em [FM00].

De acordo com o modelo ODMG, os dados em uma consulta OQL podem ser representados como coleções de objetos. Essas coleções podem ser do tipo *bag, set* ou *list*. No entanto, não há nada definido na especificação ODMG sobre como deve ser, por exemplo, a execução de uma junção entre um *set* e uma *bag* e qual tipo deve ter o seu resultado. O cálculo monóide [FM00] fornece operações com o objetivo de uniformizar o tratamento sobre diferentes tipos de coleções. Um monóide (*monoid*) fornece exatamente uma abstração sobre o tipo de coleção. Um monóide pode ser utilizado para estruturar um resultado no formato de *bags*, *sets, lists, arrays* ou reduzir coleção a valores resultantes de operações de soma.

O cálculo é utilizado como uma primeira forma de representação da consulta por ter uma tradução mais direta a partir do OQL. Depois esse cálculo é mapeado na álgebra intermediária (álgebra lógica), que é mais próxima e fácil de mapear para os algoritmos que implementam as operações. Veja os operadores lógicos abaixo:

- O operador *get* cria uma coleção onde cada item corresponde a um objeto da extensão de acordo com um dado predicado. Um parâmetro chamado *monoid* especifica como os dados serão estruturados (*set, bag* ou *list*)

- O operador *reduce* estrutura a coleção de objetos de saída, tomando cada objeto que satisfaz ao predicado e transformando-a em um valor ou item de uma coleção.

- O operador *join* é similar ao operador relacional, onde as tuplas de entrada da direita são concatenadas com as tuplas de entrada da esquerda.

- O operador *unnest* desaninha a coleção interna de objetos da coleção externa e retira todas as tuplas que não satisfazem ao predicado.

- O operador *nest* agrupa expressões de acordo com variáveis definidas em *group by*.

O operador que merece maior destaque é o *unnest*. Esse operador, juntamente com um algoritmo de "desaninhamento" (*unnesting*) [Feg98] pode remover qualquer tipo de aninhamento com técnicas de reescrita de consulta. A reescrita da consulta aninhada não depende do seu contexto. Cada consulta aninhada é reescrita independentemente e os resultados são compostos para formar a consulta final. Essa operação tem grande importância no contexto de SGBDOO, dada a presença de expressões de caminho e uma grande liberdade em OQL para definição de consultas aninhadas em qualquer parte da instrução. Essa operação por si só não traz grandes vantagens quanto à otimização, no entanto ela transforma os dados para um formato mais "plano", permitindo que outros operadores e técnicas de otimização possam ser utilizados.

2.4 Conclusões do capítulo

Este capítulo abordou o processamento de consultas em banco de dados em geral. Apresentou-se também as diferenças do modelo orientado a objetos no que se refere ao processamento de consultas, juntamente com características do modelo ODMG e sua linguagem padrão de consultas. Discutiram-se também as operações necessárias a uma álgebra orientada a objetos e apresentou-se a álgebra orientada a objetos utilizada no desenvolvimento do trabalho. O próximo

capítulo aborda o processamento de consultas em sistemas paralelos e as questões referentes ao paralelismo em SGBDOO's.

Capítulo 3

Paralelismo em Bancos de Dados

3.1 Introdução

A informatização cada vez maior de diversos setores e o avanço na capacidade de armazenamento dos computadores incentivaram a construção de bases de dados muito grandes. Uma técnica utilizada para se obter um melhor tempo de resposta na recuperação de informações sobre grandes volumes de dados é o processamento paralelo de consultas. A diminuição dos custos de hardware e o avanço nas tecnologias de redes tornam cada vez mais viável o uso de máquinas paralelas no tratamento de consultas a grandes bases de dados. Um computador paralelo ou multiprocessado pode ser visto como um sistema distribuído onde os discos, processadores e memórias estão conectados através de uma rede de alta velocidade. Um sistema de banco de dados paralelo utiliza o paralelismo para fornecer servidores de bancos de dados de alto desempenho e disponibilidade.

O paralelismo melhora o desempenho dos sistemas de bancos de dados devido, principalmente, à diminuição do gargalo causado pelas operações de E/S [OV99]. Por exemplo, considere o armazenamento de um conjunto de dados de tamanho D em um único disco com *throughput* T. Nesse caso, o *throughput* de todo o sistema está limitado a T. Contudo, se a base de dados for particionada entre *n* discos, cada um com capacidade D/n e *throughput* igual a T' (pelo menos equivalente a T), teremos um *throughput* de n*T' para ser consumido idealmente por *n* processadores. Técnicas de balanceamento de carga e de exploração do paralelismo entre consultas diferentes, bem como em uma mesma consulta, são fundamentais para melhorar o desempenho desses sistemas.

Este capítulo apresenta os tipos de paralelismo que podem ser utilizados no processamento de consultas e os modelos de execução utilizados na avaliação de planos paralelos. Discutimos, também, aspectos de arquiteturas para a organização do *hardware* e *software* de um sistema com o objetivo de fornecer um servidor de dados paralelo de alto desempenho e disponibilidade.

Mais adiante, retornamos ao problema de otimização de consultas, mas desta vez sob a ótica do paralelismo. Finalmente, analisamos os aspectos de otimização e arquitetura para sistemas paralelos de bancos de dados orientados a objetos.

3.2 Arquiteturas

A arquitetura em alto-nível de um servidor paralelo é exibida na

Figura 3.1. Os componentes de *software* apresentados nesss figura podem ser divididos em três sub-sistemas [OV99]:

- **Gerenciador de Sessão (*Session Manager*)**: Responsável pela interação entre cliente e servidor e gerenciamento de conexões.

- **Gerenciador de Requisições (*Request Manager*)**: Responsável pelo recebimento de requisições dos clientes relativas à compilação e execução de consultas. Esse módulo conduz a transformação da consulta em um conjunto de planos de execução semanticamente equivalentes, e coordena a escolha do "melhor" plano e sua compilação.

- **Gerenciador de Dados (*Data Manager*)**: É considerado o sistema operacional do banco de dados paralelo, sendo responsável pelo gerenciamento de *buffer*/*cache*, balanceamento de carga e transações paralelas (tarefas não suportadas de forma eficiente pelos sistemas operacionais tradicionais).

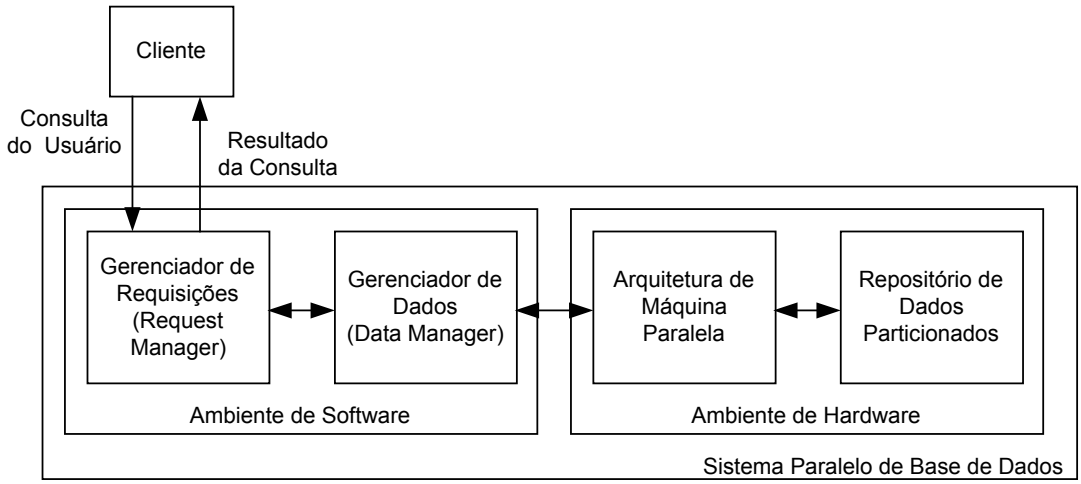


Figura 3.1 - Arquitetura em alto-nível de um SGBD paralelo

Os componentes de *hardware* da Figura 3.1 podem ser divididos em:

- ***Hardware* do Ambiente de Execução**: Para fornecer suporte ao paralelismo, *hardware* específico deve ser fornecido, incluindo uma máquina com arquitetura paralela e um esquema de armazenamento com suporte para particionamento de dados.

- **Arquitetura da Máquina Paralela**: No contexto de servidores de bancos de dados, as máquinas são classificadas de acordo com o compartilhamento de recursos como memória, processadores e discos. Com o avanço dos sistemas operacionais e da tecnologia de redes, as arquiteturas de sistemas distribuídos e banco de dados paralelos adquirem, cada vez mais, características em comum. Assim, arquiteturas sem compartilhamento podem ser implementadas como *clusters* de PC's conectados através de uma rede ATM e com um Kernel de sistema operacional paralelo com suporte para *Physical Virtual Machine* (PVM) e chamada de procedimentos remotos (RPC), entre outras características.

A seguir está a classificação mais utilizada para arquiteturas de máquinas paralelas. Essa classificação define como processadores, memórias e discos podem ser organizados em um servidor de banco de dados paralelo:

- **Memória compartilhada**: Todas as memórias e discos podem ser acessados por qualquer um dos processadores do sistema (Figura 3.2). A construção de bancos de dados nessa arquitetura difere pouco de sistemas monoprocessados, facilitando o reaproveitamento de diversas técnicas. Outra vantagem diz respeito ao balanceamento de carga simplificado devido à disponibilidade dos meta-dados e da tabela de bloqueios pelos diversos processadores. Contudo a escalabilidade do sistema é prejudicada devido ao gerenciamento de conflito dos acessos aos recursos (memória principal e discos), o que leva a um alto custo proporcionalmente crescente ao número de processadores adicionados ao sistema.

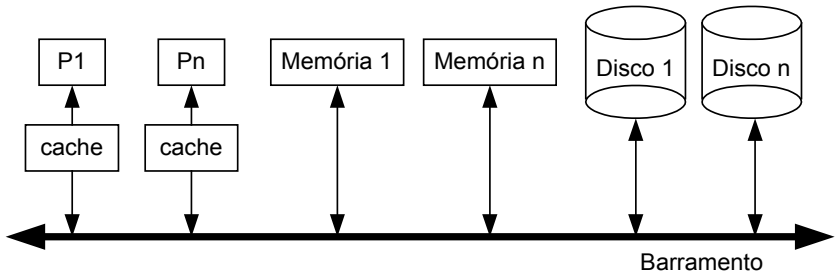


Figura 3.2 - Arquitetura de memória compartilhada

- **Compartilhamento de discos**: Nessa arquitetura (Figura 3.3), somente os discos são utilizados de forma compartilhada entre os processadores do sistema. O acesso à memória é exclusivo do respectivo processador. É ideal para aplicações onde predominam as consultas aos dados. As principais vantagens são: baixo custo, alta disponibilidade e extensibilidade e fácil migração a partir de sistemas monoprocessados.

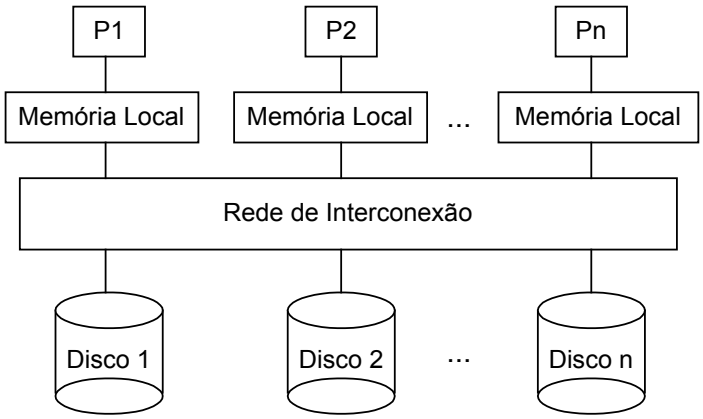


Figura 3.3 - Arquitetura de compartilhamento de discos

- **Sistemas sem compartilhamento**: Nessa arquitetura (Figura 3.4), cada processador faz acesso a um subconjunto dos discos de forma exclusiva e a memória principal é endereçada exclusivamente pelo seu processador. Essa arquitetura é bastante similar a um sistema de banco de dados distribuído, e assim a maioria das técnicas utilizadas para tais sistemas pode ser aproveitada para essa arquitetura. O tráfego de rede é bastante reduzido, já que grande parte dos dados pode ser filtrada dentro do próprio nó de processamento. Essa arquitetura tem se firmado como a que possui maior grau de escalabilidade para processamento de consultas em grandes bases de dados. A comunicação entre os nós é realizada através de trocas de mensagens. O custo é equivalente ao de sistemas com compartilhamento de discos e a complexidade chega a ser maior que a de sistemas com compartilhamento de memória.

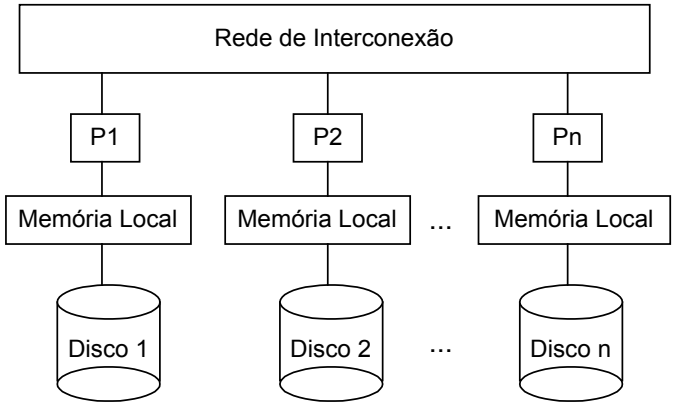


Figura 3.4 - Arquitetura sem compartilhamento

- **Arquiteturas hierárquicas**: São uma combinação entre as arquiteturas de memória compartilhada e sistemas sem compartilhamento (veja Figura 3.5). Essa combinação deve produzir uma máquina sem compartilhamento cujos nós possuem memória compartilhada. Essa arquitetura combina a grande extensibilidade fornecida por sistemas sem compartilhamento com a flexibilidade e desempenho de sistemas de memória compartilhada.

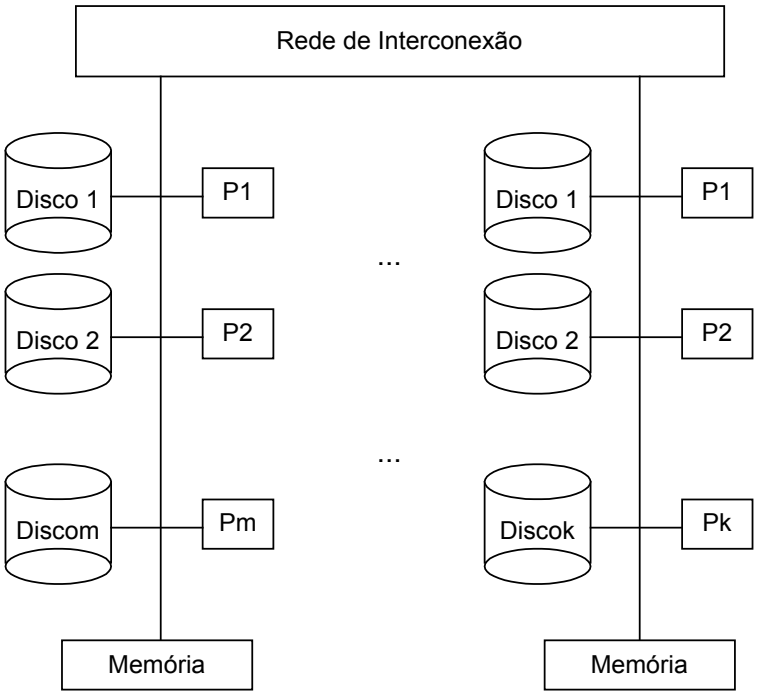


Figura 3.5 - Arquitetura hierárquica

A arquitetura sem compartilhamento tem se mostrado bastante viável devido ao avanço nos sistemas distribuídos e redes rápidas, como *Fast Ethernet* e *ATM*. Além disso, ela fornece uma boa escalabilidade com a simples adição de máquinas ao *Cluster*. No entanto, nessa arquitetura os custos de transmissão de dados se tornam mais críticos para operações de reparticionamento de dados.

3.3 Paralelismo em consultas

O paralelismo de consultas pode ser dividido em dois grandes grupos: Paralelismo inter-consultas e paralelismo intra-consulta [YM98]. O último é dividido em paralelismo inter-operadores e paralelismo intra-operador.

A Figura 3.6 exibe a relação custo-benefício entre os tipos de paralelismo, tempo de resposta e capacidade do sistema. O paralelismo inter-consultas possibilita um aumento no volume de dados processados através da execução paralela de consultas concorrentes, ou seja, aumenta-se o *throughput* do sistema. Por outro lado, o paralelismo intra-consulta possibilita uma diminuição no tempo de resposta de uma única consulta. Observa-se também que processamento paralelo de uma única operação da álgebra tende a melhorar o tempo de resposta de uma consulta, de forma comparativa à exploração do paralelismo entre operações diferentes.

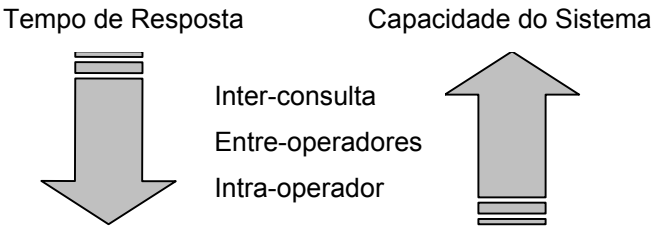


Figura 3.6 - Impacto dos tipos de paralelismo sobre tempo de resposta e capacidade do sistema

O paralelismo intra-operador, também conhecido como paralelismo particionado, é baseado na decomposição dos operadores em um conjunto de sub-operadores, ou *clones*, que executam a mesma tarefa sobre diferentes partições dos dados. Esse tipo de paralelismo divide o processamento entre os vários nós disponíveis no sistema de acordo com o particionamento dos dados. Após o término do processamento de cada sub-operador, os resultados individuais são integrados com o uso de um operador do tipo *merge*.

O paralelismo inter-operador pode ser dividido em duas classes: paralelismo *pipeline* e paralelismo independente. O paralelismo *pipeline* permite o processamento em conjunto de vários operadores em um esquema produtor-consumidor. À medida que os dados são produzidos por um operador, o outro operador recebe esse resultado intermediário e o processa imediatamente. Dessa forma, evita-se a materialização dos dados em estruturas temporárias, minimizando o uso de recursos de armazenamento do sistema. O paralelismo independente pode ser usado entre dois operadores que não necessitam dos dados produzidos por cada um. Os dois operadores podem ser executados paralelamente de forma independente.

Os dois principais modelos para execução paralela de consultas são o Modelo de Operador (*Operator Model*), utilizado em SGBD's como Volcano[GM93] e Tandem NonStop SQL [Hol88], e o Modelo de Suporte (*Bracket Model*), utilizados em SGBD's como Gamma [DGS88] e Bubba [Bor88].

O *Bracket Model* utiliza um processo "molde", denominado *template*, no qual podem ser encaixados diversos tipos de processos que implementem operadores físicos da álgebra utilizada na representação da consulta. O processo referente a cada operador é controlado pelo processo "molde", que determina as leituras e escritas de acordo com um esquema produtor-consumidor. O processo molde fornece dois pontos de acoplamento para controle da leitura dos dados produzidos pelos planos de entrada do operador, no entanto, se o operador for unário apenas um ponto é utilizado. Um outro ponto de acoplamento é fornecido para controlar a produção dos dados gerados pelo operador. Na Figura 3.7, visualiza-se como o operador binário de junção, *Join*, e operador unário de agrupamento, *Group By*, podem ser acoplados ao processo "molde".

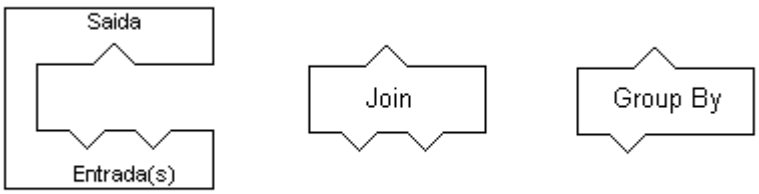


Figura 3.7 - Bracket Model

O Modelo de Operador assume que todo operador é implementado sob um paradigma de *iterador* e assim deve possuir uma interface bem definida, através da qual três funções são utilizadas para realizar a execução do processo que corresponde ao operador. Essas três funções são definidas da seguinte forma:

- Open
    1. Instancia um registro de estado, armazenando os argumentos de cada operador, incluindo funções de suporte.
    2. Chama função open para cada uma das entradas do operador
- Next
    1. É executado repetidamente até encontrar o fim da corrente de dados
- Close
    1. Finaliza o operador e estruturas criadas na inicialização (Open)
    2. Chama a função close para cada uma das entradas.

Para controlar a chamada de cada uma dessas funções um operador físico especial deve ser introduzido à álgebra original. Esse operador é denominado de *exchange* e quando situado em um processo consumidor ele se comporta como um iterador, recebendo os dados do processo produtor através de comunicação inter-processos (IPC). O operador *exchange* é implementado através de dois processos de transferência de dados. Um dos processos é executado antes do processo do operador consumidor e faz requisições ao processo *exchange* que fica sobre o processo do operador produtor. Esses dois processos realizam as trocas de mensagens e tarefas necessárias para a comunicação dos dados, isolando os outros operadores de detalhes relativos ao paralelismo. Na Figura 3.8 é apresentado um plano para uma consulta OQL que recupera o número do departamento dos empregados que ganham mais do que um determinado valor. Para recuperar essa informção é necessário relacionar os objetos da classe *Dept* com os objetos da classe *Emp*. Essa operação é implementada pelo operador *merge_join* que requer uma ordenação (*sort*) sobre os empregados. O operador *reduce* é usado para formatar o resultado da consulta. O operador *exchange* é inserido entre o *sort* e *merge_join*, e o *merge_join e reduce,* para realizar reparticionamento de dados necessário ao paralelismo intra-operador. Para que o

merge_join seja executado em paralelo, as duas entradas devem estar particionadas de acordo com *DeptNumber*. O *exchange* realiza o reparticionamento e se encarrega da transmissão dos dados. Já entre o *merge_join* e o *reduce*, o *exchange* permite a execução em *pipeling* (paralelismo inter-operador) entre os dois operadores, fornecendo os dados sob demanda, de forma que os dois operadores sejam executados em paralelo.

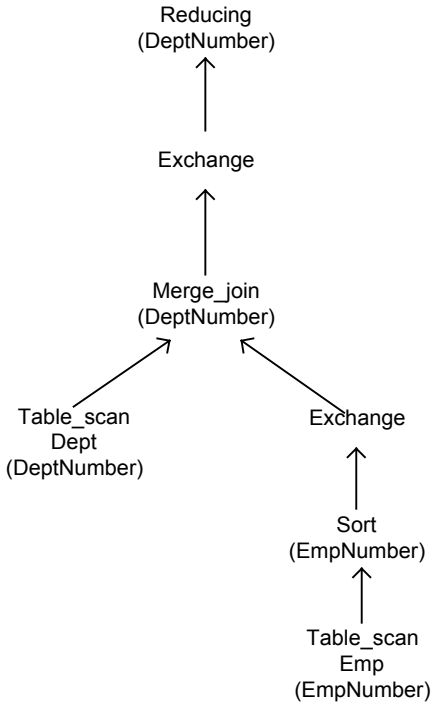


Figura 3.8 - Plano de execução com Operador Model

3.4 Particionamento

Como visto na seção 3.3.1, o paralelismo intra-operador é um dos principais tipos de paralelismo. Para sua utilização é necessário definir esquemas para distribuir os dados através dos nós de processamento do sistema, dividindo o conteúdo das estruturas de armazenamento sob determinado critério. A essa distribuição, dá-se o nome de particionamento dos dados. A maneira como os dados serão particionados através dos nós, influencia na execução dos operadores, já que estes devem ser executados o mais próximo possível dos dados.

O particionamento tende a melhorar o tempo de resposta, mas aumenta o trabalho total devido aos custos de comunicação. Já o agrupamento dos dados diminui o custo total, mas causa uma queda no tempo de respostas. As estratégias de particionamento devem considerar esses aspectos para atingir uma distribuição adequada dos dados.

O boa distribuição de dados tem impacto direto sobre o tempo de resposta de consultas em SGBD's paralelos. Se um nó de processamento receber mais dados que outros, o tempo de resposta total de uma operação será o tempo necessário para o término do *clone* da operação no nó mais carregado. O problema da má distribuição de dados é conhecido como *Data Skew* e algumas técnicas para a solução do problema e obtenção de um bom balanceamento de carga podem ser encontradas em [HL91], [RM93] e [MD97]. Abaixo são apresentadas três estratégias básicas para o particionamento de dados.

- **Particionamento circular**: É a estratégia mais simples. Os dados são distribuídos uniformemente de forma a facilitar o acesso seqüencial dos dados, sendo ideal para consultas que não acessam dados relacionados. O particionamento é feito posicionando a tupla i na partição (i mod n), considerando-se n o número de partições.

- **Particionamento *hash***: É a estratégia ideal para consultas associativas, onde se é necessário relacionar informações com operações do tipo junção. O particionamento é obtido através da aplicação de uma função *hash* sobre o valor de determinado atributo. A aplicação dessa função irá determinar o número da partição onde o dado será alocado. Esse particionamento funciona adequadamente quando os predicados das consultas utilizam a igualdade como forma de comparação. Essa estratégia tende a distribuir os dados de maneira aleatória quando existe uma grande variedade de valores no atributo selecionado. Isso pode prejudicar algumas técnicas que levam em consideração o agrupamento de dados relacionados (*clustering*). Em geral, as partições tendem a ter o mesmo tamanho.

- **Particionamento por faixa de valores**: Essa estratégia agrupa os dados de acordo com uma faixa de valores aplicados sobre um determinado atributo. Assim essa estratégia pode ser bem utilizada em consultas com acesso seqüencial, associativo e para dados agrupados. Aplicações que gerenciam dados com históricos podem se beneficiar com esse tipo de esquema criando partições sobre atributos do tipo data. Contudo o problema conhecido como *data skew* pode ocorrer devido a má distribuição dos dados ocasionada por uma maior ocorrência de valores dentro de faixas específicas, e assim, provocando a alocação uma grande quantidade de dados a uma partição específica.

- **Particionamento composto**: Essa estratégia combina características do particionamento *hash* e por faixa de valores. Nesse caso, os dados são distribuídos inicialmente de acordo com os limites estabelecidos pelas faixas de valores. Em seguida, os dados são distribuídos em sub-partições (dentro de cada partição por faixa de valores) de acordo com um algoritmo de *hashing*.

Abaixo temos exemplos, em DDL do *Oracle Parallel Server*, da criação de tabelas com alguns dos tipos de particionamento citados.

No exemplo *(a)* a tabela de vendas (*sales*) é criada com particionamento por faixa de valores. A tabela é particionada sobre a data da venda, atributo *s_saledate*, em oito partições distintas.

O exemplo *(b)* particiona a mesma tabela através do identificador do produto, atributo `s_productid,` utilizando o método *hash*. Isso é justificado devido ao fato do identificador do produto ser utilizado para realizar consultas associativas através de junções.

O exemplo *(c)* utiliza o particionamento composto, através do qual a tabela de vendas possui partições por faixa de valor sobre a data da venda, e dentro de cada uma delas, sub-partições *hash* sobre o identificador do produto.

```
CREATE TABLE sales
(s_productid NUMBER,
 s_saledate DATE,
 s_custid NUMBER,
 s_totalprice NUMBER)
 PARTITION BY RANGE(s_saledate)
 (
  PARTITION sal94q1 VALUES LESS THAN TO_DATE (01-APR-1994, DD-MON-YYYY),
  PARTITION sal94q2 VALUES LESS THAN TO_DATE (01-JUL-1994, DD-MON-YYYY),
  PARTITION sal94q3 VALUES LESS THAN TO_DATE (01-OCT-1994, DD-MON-YYYY),
  PARTITION sal94q4 VALUES LESS THAN TO_DATE (01-JAN-1995, DD-MON-YYYY),
  PARTITION sal95q1 VALUES LESS THAN TO_DATE (01-APR-1995, DD-MON-YYYY),
  PARTITION sal95q2 VALUES LESS THAN TO_DATE (01-JUL-1995, DD-MON-YYYY),
  PARTITION sal95q3 VALUES LESS THAN TO_DATE (01-OCT-1995, DD-MON-YYYY),
  PARTITION sal95q4 VALUES LESS THAN TO_DATE (01-JAN-1996, DD-MON-YYYY)
  );
```

Exemplo (a): DDL para criação de tabela com particionamento por faixa de valores

```
CREATE TABLE sales
(s_productid NUMBER,
 s_saledate DATE,
 s_custid NUMBER,
 s_totalprice NUMBER)
 PARTITION BY HASH(s_productid)
 PARTITIONS 4;
```

Exemplo (b): DDL para criação de tabela com particionamento hash

```
 CREATE TABLE sales
 (s_productid NUMBER,
  s_saledate DATE,
  s_custid NUMBER,
  s_totalprice)
  PARTITION BY RANGE (s_saledate)
  SUBPARTITION BY HASH (s_productid)
  SUBPARTITIONS 4
  (
   PARTITION sal94q1 VALUES LESS THAN TO_DATE(01-APR-1994, DD-MON-YYYY),
   PARTITION sal94q2 VALUES LESS THAN TO_DATE(01-JUL-1994, DD-MON-YYYY),
   PARTITION sal94q3 VALUES LESS THAN TO_DATE(01-OCT-1994, DD-MON-YYYY),
   PARTITION sal94q4 VALUES LESS THAN TO_DATE(01-JAN-1995, DD-MON-YYYY)
   );
```

Exemplo (c): DDL para criação de tabela com particionamento composto

3.6 Modelos de otimização

A otimização de consultas em bancos de dados paralelos utiliza os mesmos componentes básicos de um sistema seqüencial: um espaço de busca, um modelo de custo e uma estratégia de busca [OV99].

Quanto ao espaço de busca, as árvores de operadores podem receber anotações específicas para um sistema paralelo, como, por exemplo, se dois operadores podem ser executados através de *pipeline*.

O modelo de custo pode ser dependente ou independente da arquitetura utilizada. O modelo independente considera apenas o custo para os algoritmos que implementam os operadores. O modelo dependente, além do custo dos operadores, considera também os custos para reparticionamento de dados e consumo de memória. Funções de custo para um modelo dependente de arquitetura possuem os seguintes componentes: o trabalho total, o tempo de resposta e o consumo de memória. Os dois primeiros componentes são utilizados para medir o balanceamento entre tempo de resposta e *throughput*. O terceiro é utilizado apenas para descartar planos de execução que necessitem de mais memória do que a disponível.

A estratégia de busca encontra um espaço de busca muito maior do que os que ocorrem em sistemas sequenciais, pois além da exploração de diferentes algoritmos para a implementação dos operadores, deve-se também considerar quais os tipos de paralelismo a utilizar e as possíveis anotações para cada operador. Essas anotações podem ser, por exemplo, o atributo de particionamento para processamento dos operadores ou se dois operadores podem ser processados com *pipeline*.

A otimização de consultas para bancos de dados paralelos pode ser executada com a utilização de duas técnicas:

Otimização em duas fases: A primeira fase consiste da geração de um plano de execução para uma máquina seqüencial sem considerar a alocação dos recursos (processadores, unidades de disco e memória). A segunda fase aloca os recursos disponíveis aos operadores dos planos gerados na primeira fase [HS91].

Essa estratégia é mais simples e flexível, já que divide o problema de otimização paralela em duas fases, reaproveitando as técnicas de otimização desenvolvidas para a primeira fase. Além disso, essa abordagem reduz o espaço de busca do plano de execução de consultas (PEC), pois explora o paralelismo apenas nos planos ótimos de execução seqüencial. Outra vantagem dessa estratégia é que ela facilita o reaproveitamento de técnicas de otimização desenvolvidas para máquinas seqüenciais. No entanto, a restrição inicial ao espaço de busca pode evitar que se encontre o plano "ótimo". Uma variação dessa proposta é abordada em [Has96], onde a primeira fase realiza a reordenação de junções e seleção de métodos para os operadores considerando o paralelismo e os custos envolvidos, como, por exemplo, custos com reparticionamento dos dados. A segunda fase é dividida na extração do paralelismo e escalonamento da consulta (veja

Figura 3.9). Essa abordagem, não restringe tanto o espaço de busca e ainda continua facilitando o reuso e extensibilidade do otimizador.

Otimização em uma fase: Combina as duas fases descritas acima em uma só e tenta gerar ao mesmo tempo o plano de execução de consultas e o plano de alocação de recursos. Essa abordagem pode conduzir a planos mais eficientes que a otimização em duas fases, no entanto, a sua complexidade é muito maior, dificultando a extensibilidade do otimizador.

Neste trabalho, utiliza-se uma variação da abordagem de otimização em duas fases proposta em [Has96] para a otimização de sistemas relacionais. Essa abordagem facilita a extensibilidade do otimizador, e não compromete o resultado final por considerar o paralelismo já na primeira fase. A primeira fase, chamada de reescrita da consulta e também conhecida como JOQR, concentra-se na otimização da árvore com relação ao paralelismo particionado, ordenando as junções e selecionando operadores físicos de acordo com os custos de transmissão envolvidos no reparticionamento dinâmico dos dados. Este trabalho utiliza a fase JOQR em um contexto de bancos de dados orientados a objetos, e propõe um técnica baseada em regras de reescrita de termos para uma implementação mais extensível da fase. O capítulo 4 detalha essa fase e explica como ela é utilizada no contexto do trabalho.

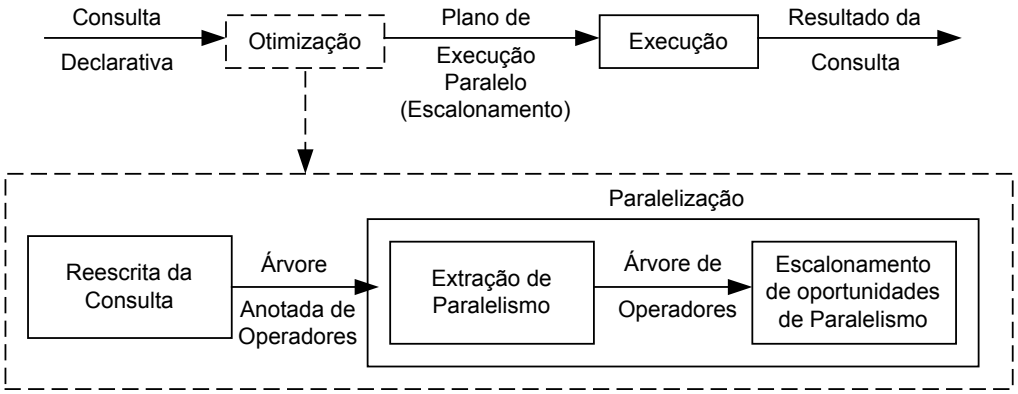


Figura 3.9 - Otimização paralela de consultas em duas fases

A segunda fase, chamada de paralelização, pode ser dividida em duas etapas, a extração do paralelismo e o escalonamento das oportunidades de paralelismo. A etapa de extração do paralelismo tem como objetivo obter as oportunidades de paralelismo, evidenciando no plano de execuções quais operadores podem ser executados com paralelismo do tipo *pipeline*. Para obter oportunidades de paralelismo, deve-se identificar as unidades atômicas de execução e as restrições de tempo entre essas unidades. As restrições de tempo são classificadas como restrição de precedência e restrição de paralelismo. Uma restrição de precedência indica que um operador só pode iniciar sua execução após o término do processamento do operador produtor, enquanto que uma restrição de paralelismo indica que um operador consumidor pode ser executado paralelamente com o produtor através de *pipeline*. Por exemplo, entre os operadores *Merge_Join* e *Table_Scan* [Feg97a], existe uma restrição de paralelismo, indicando que os dois operadores podem iniciar e terminar seus processamentos juntos. Oportunidades como essa podem ser utilizadas ou não, cabendo ao escalonador essa decisão.

Em um trabalho anterior, desenvolvido no contexto de uma dissertação de mestrado [Sam98] para a etapa de paralelização, foi desenvolvida uma técnica de extração de paralelismo em um otimizador para bancos de dados orientados a objetos através da fatoração apresentada em [Feg97a]. Esse trabalho será utilizado e estendido para possibilitar a criação de um otimizador extensível com o paralelismo intra-perador e inter-operador.

Um escalonador de consultas é o componente do SGBD que aloca os recursos disponíveis (memória, unidades de discos e processadores) aos nós de uma árvore de operadores [GI96]. Sua tarefa é receber um plano de execução anotado com oportunidades de paralelismo intra-operador e inter-operador e fornecer como saída um plano de alocação dos operadores aos nós de processamento disponível. Além do plano de execução da consulta, o escalonador utiliza informações disponíveis no meta-esquema, ou até em informações obtidas em tempo real sobre a carga de cada nó, para gerar o plano de alocação que determina os nós de processamento nos quais grupos de operadores serão executados.

3.7 Sistemas de bancos de dados paralelos orientados a objetos

3.7.1 Características

Nessa seção apresentamos os requisitos e a arquitetura para SGBDOO's paralelos, destacando os pontos que os diferenciam de sistemas relacionais.

Características específicas de bancos de dados orientados a objetos sequenciais, como objetos complexos, expressões de caminho e execução de métodos, também merecem atenção especial em sistemas de bancos de dados paralelos orientados a objetos. Além do processamento dos operadores da álgebra, deve-se considerar técnicas para a execução paralela de métodos [MC97]. Além disso, muitas técnicas do modelo relacional podem ser reaproveitadas e estendidas, de acordo com as características já discutidas, para o modelo orientado a objetos.

Entre essas diferenças, as que possuem maior impacto no projeto de um sistema paralelo são:

- Objetos em um banco de dados orientado a objetos podem ser referenciados por um identificador único, ou indiretamente como membros de coleções (*list*, *bags*, *sets*) que modelam os relacionamentos entre os diversos tipos de objetos de uma aplicação. Em banco de dados relacionais, apenas as tabelas (conjunto de linhas) podem ser

referenciadas individualmente, já que uma linha não possui identidade própria. O identificador do objeto em um sistema paralelo deve permitir a localização exata do objeto, e assim, devem fornecer mecanismos para se localizar o nó, disco onde o objeto se encontra. Por exemplo, se essa informação estiver armazenada diretamente no identificador, em caso de reparticionamento do objeto, esse identificador deverá ser atualizado. Nesse caso os identificadores retidos pelas aplicações clientes ficarão inválidos. Como manter a coerência?

- Objetos podem ser acessados de duas formas diferentes em uma base de dados. Através de uma linguagem de consulta declarativa, OQL por exemplo, ou através do mapeamento direto (materialização) dos objetos da base de dados em objetos da aplicação cliente, por exemplo, objetos Java, C++, ou objetos baseados em CORBA em geral. Em bases relacionais, a única maneira de acesso aos dados é através da linguagem de consulta SQL. Muitas técnicas, desenvolvidas para a recuperação com SQL no modelo relacional, podem ser utilizadas para o OQL. No entanto, a materialização de objetos da base de dados, diretamente em objetos nos programas clientes pode exigir a recuperação de informações distribuídas em diversos nós da máquina paralela. Essa operação não utiliza a otimização algébrica, e assim, novas técnicas devem ser desenvolvidas para determinar um bom plano de materialização de objetos.
- Objetos podem ter métodos definidos pelos usuários. Esses métodos podem ser usados dentro da linguagem de consulta ou nos objetos materializados nas aplicações clientes. Em bases relacionais existem apenas um conjunto de operações pré-definidas, como funções de datas, *strings* e matemáticas para utilização conjunta com o SQL. A utilização desses métodos dentro do OQL levanta algumas questões com relação à otimização no contexto de sistemas paralelos. Por exemplo, será possível definir particionamento sobre resultados de métodos como se faz para atributos? Como consultas OQL embutidas em métodos colaboram com a consulta global para geração do plano ótimo? Por exemplo, como gerar um plano para uma consulta que possui uma classe C1 que obedece a um

particionamento sobre um atributo A1 e que deve ser filtrada pelo valor retornado por um método M, que por sua vez utiliza consultas sobre outra classe C2 com um particionamento diferente? Outro problema ocorre quando o método é escrito em uma linguagem como Java ou C++. Como essas linguagens não fornecem suporte ao paralelismo, a execução do método pode se tornar um gargalo no processamento paralelo da consulta.

Da mesma forma que um sistema relacional, um SGBDOO paralelo deve ser capaz de suportar paralelismo intra-consulta. Para isso, ele deve receber as requisições em OQL enviadas pelos clientes, gerar um plano de execução paralelo de acordo com as características do modelo orientado a objetos e coordenar a execução dos operadores nos nós de processamento do sistema paralelo (Figura 3.10-a). Tal sistema deve igualmente dar suporte ao paralelismo inter-consulta, distribuindo as consultas OQL entre os nós de processamento e, assim, distribuindo a carga gerada por diversos clientes. Além disso, esse sistema deve permitir a materialização direta de um objeto ou coleção deles em estruturas equivalentes nas aplicações clientes. A materialização de um objeto da base de dados em objetos da aplicação (Figura 3.10-b) pode requerer a recuperação de diversos outros objetos relacionados sob a forma de coleções aninhadas (agregadas) no estado desse objeto. Em várias situações, a aplicação cliente pode estar interessada em apenas um único atributo desse objeto, e assim não existe razão para carregar e transferir para o cliente os dados de diversos outros atributos complexos, incluindo coleções de outros tipos de objetos. O sistema de banco de dados deve prover um mecanismo de carga sob demanda, aproveitando as vantagens do paralelismo, que gerencie a materialização de objetos.

SGBDOO's podem também ser utilizados no contexto de sistemas de objetos distribuídos (Figura 3.10-c) como servidores de objetos CORBA [OMG91]. Assim, as aplicações distribuídas podem utilizar o componente CORBA Object Database Adapter (ODA) para se conectar à base e solicitar a execução de métodos remotamente, no próprio servidor de dados. Para suportar grandes cargas geradas por aplicações distribuídas, o servidor de dados paralelo deve

garantir o processamento paralelo das requisições e conexões suportadas pelo barramento CORBA.

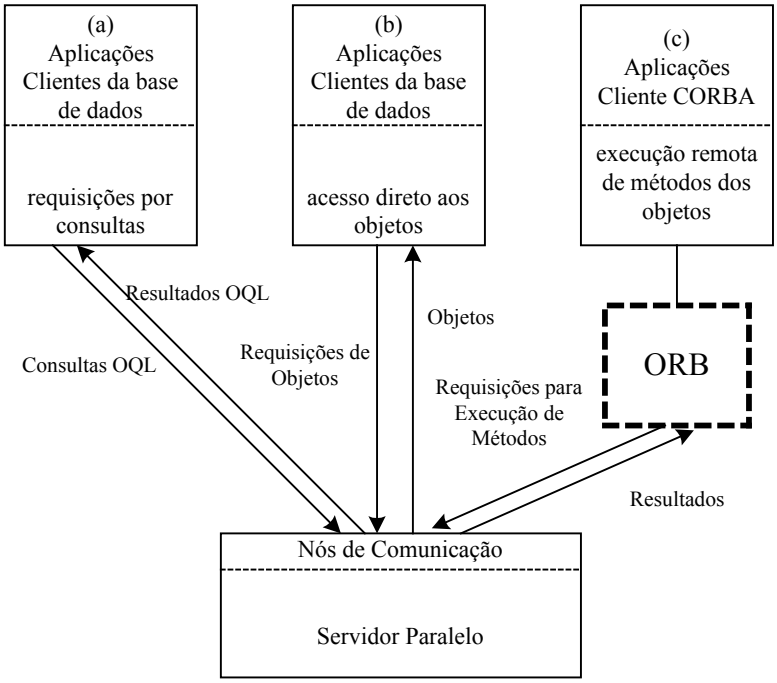


Figura 3.10 - Interações cliente-servidor em um SGBDOO paralelo

3.7.2 Álgebra física para SGBDOO's paralelos

Esta seção descreve a álgebra física orientada a objetos utilizada neste trabalho. Essa álgebra é uma extensão da álgebra física para sistemas não-paralelos apresentada em [Feg97a] e discutida na seção 2.4. Os operadores são equivalentes àqueles da álgebra original, com exceção do *Hash-Loop* [LDM93], *Materialize* (utilizados para *pointer joins*) e do *Exchange*. Esse último permite a introdução do paralelismo na álgebra seqüencial. O operador de paralelismo é inserido no meio do plano seqüencial, encapsulando o paralelismo e facilitando o desenvolvimento e a manutenção dos outros operadores. O uso do *Exchange* evidencia o uso da abordagem *operator model* [Gra90] para paralelização de planos de execução de consultas. Abaixo apresentamos uma listagem dos operadores dessa álgebra juntamente com uma breve descrição de suas funcionalidades. Exemplos com explicações detalhadas sobre o funcionamento

desses operadores podem ser encontrados nos planos gerados para as consultas do capítulo 5.

1. **Table_scan(extent, range_variable, predicate)**

   Cria uma corrente de objetos satisfazendo o predicado (*predicate*) informado, a partir da extensão (*extent*) dos dados. Cada objeto pode ser referenciado individualmente por *range_variable*.

2. **Index_scan(extent, range_variable, predicate, index, sort_order)**

   Da mesma forma que o *Table_scan*, esse operador cria uma corrente de objetos satisfazendo o predicado (*predicate*) informado, a partir da extensão (*extent*) dos dados, mas utilizando um índice (*index*) para disponibilizar os objetos ordenados de acordo com *sort_order*.

3. **Nested_loop(left_plan, right_plan, predicate, keep)**

   Cria uma corrente de objetos a partir da junção dos objetos dos planos de entrada da direita e esquerda, indicados respectivamente por *left_plan* e *right_plan*, e que satisfazem ao predicado indicado em *predicate*. O último parâmetro, *keep*, determina como a operação de junção deve se comportar. Se *keep=left*, então a operação se comporta como um *left outer join*. Se *keep=right*, então a operação se comporta como um *right outer join*. Se *keep=none*, então a operação se comporta como uma junção comum.

4. **Block_Nested_loop(left_plan, right_plan, predicate, keep)**

   Cria uma corrente de objetos a partir da junção dos objetos dos planos de entrada da direita e esquerda, indicados respectivamente por *left_plan* e *right_plan*, e que satisfazem ao predicado indicado em *predicate*. A corrente correspondente a *left_plan* é lida em blocos de 1000. O último parâmetro, *keep*, determina como a operação de junção deve se comportar. Se *keep=left*, então a operação se comporta como um *left outer join*. Se *keep=right*, então a operação se comporta como um *right outer join*. Se *keep=none*, então a operação se comporta como uma junção comum.

5. **Hash_loop(left_plan, right_plan, predicate, keep)**

   Cria uma corrente de objetos a partir da junção de objetos dos planos de entrada da direita e esquerda, indicados respectivamente por *left_plan* e *right_plan*, e que satisfazem ao predicado indicado em *predicate*. Os dados de *left_plan* são replicados de acordo com o particionamento de *right_plan*. O último parâmetro, *keep*, determina como a operação de junção deve se comportar. Se *keep=left*, então a operação se comporta como um *left outer join*. Se *keep=right*, então a operação se comporta como um *right outer join*. Se *keep=none* então a operação se comporta como uma junção comum.

6. **Nest(monoid, plan, variable, head, groupby, nestvars, predicate)**

   Agrupa os objetos do plano de entrada (*plan*) de acordo com o conjunto de atributos em *groupby*, aplicando *head* e aninhando os atributos em *nestvars* para cada grupo resultante. Mantém apenas os grupos que satisfazem ao predicado em *predicate*.

7. **Groupby(monoid, plan, range_variable, head, groupby, nestvars, predicate)**

   Similar ao *Nest*, mas requer que o plano de entrada (*plan*) seja ordenado de acordo com os atributos em *groupby*.

8. **Unnest(plan, variable, path, predicate, keep)**

   Concatena cada objeto no plano de entrada (*plan*) (coleção externa, ligada a *variable*) a todos os possíveis valores em *path* (coleção interna). Mantém apenas os objetos que satisfazem ao predicado (*predicate*). Se *keep=true*, a corrente de dados é preenchida com nulos para valores inexistentes em *path*.

9. **Sort(plan, sord_order)**

   Ordena o plano de entrada (*plan*) de acordo com *sort_order*

10. **Union(monoid, left_plan, right_plan)**

    Une as correntes de entradas, compatíveis com união, dos planos de entrada (*left_plan* e *right_plan*). O parâmetro *monoid* indica como o resultado deve ser estruturado.

11. **Map(plan, variable, function)**

    Estende a corrente de entrada do plano (*plan*) com a ligação de *variable* ao resultado da aplicação da função, especificada em *function,* aos objetos de entrada.

12. **Materialize(plan, path, predicate1, predicate2)**

    Concatena cada objeto do plano de entrada (*plan*) a uma corrente que contem um objeto de um *extent* ao qual os objetos de entrada estão relacionados através da expressão de caminho em *path*. O predicado indicado por *predicate1* filtra as correntes do plano de entrada e o predicadoindicado por *predicate2* filtra os planos gerados pelo processo de concatenação.

13. **Exchange(plan, variable, destination)**

    Recebe os objetos do plano de entrada em *plan* de diferentes nós de processamento da máquina paralela (cada objeto ligado a *variable*), e computa o nó de destino de cada objeto, utilizando o parâmetro de destino (indicado em *destination*), que é uma função de particionamento aplicada sobre o(s) atributo(s) que indicam a partição selecionada.

De acordo com [Has96], os operadores de uma álgebra paralela podem ser classificados em *sensíveis ao atributo* e *insensíveis ao atributo*. Um operador sensível ao atributo exige que o processamento paralelo com paralelismo particionado seja executado apenas sobre um atributo específico. Por exemplo, operadores físicos que implementam junções, em geral, requerem que os dados sejam particionados pelos atributos do predicado de junção. Operadores insensíveis ao atributos pode ser executados com paralelismo particionado sobre qualquer atributo.

Na Tabela 3.1 temos a classificação da álgebra abordada com relação à sensibilidade ao atributo.

Operador	**Sensível ao atributo**	**Insensível ao atributo**
Table_scan	**X**	
Index_scan	**X**	
Reduce		**X**

Nested_loop	**X**	
Block_Nested_Loop	**X**	
Hash_Loop		**X**
Sort		**X**
Unnest		**X**
Nest	**X**	
Groupby	**X**	
Union		**X**
Map	**X**	
Materialize		**X**
Exchange		**X**

Tabela 3.1 - Classificação dos operadores

A álgebra lógica apresentada em [Feg97a] pode ser implementada pelos operadores físicos da Tabela 3.1 de acordo com a Tabela 3.2. Note que os operadores *Sort* e *Exchange* não possuem correspondentes na álgebra lógica já que ambos são inseridos no plano apenas para garantir (*enforcers*) propriedades físicas como, por exemplo, ordenação e particionamento de dados.

Operador Lógico	**Operador Físico Paralelo**
Get	Table_scan Index_scan
Reduce	Reduce
Join	Nested_loop Merge_join Block_Nested_Loop Hash_Loop Materialize
-	Sort
Unnest	Unnest
Nest	Nest Groupby
Union	Union
Map	Map
-	Exchange

Tabela 3.2 - Relação entre operadores lógicos e físicos

3.8 Conclusões do capítulo

Este capítulo apresentou os conceitos sobre sistemas paralelos e suas arquiteturas. Foram abordados, também, os tipos de paralelismo utilizados no processamento de consultas. A otimização de consultas foi novamente abordada, mas no contexto do paralelismo. Apresentou-se o modelo de otimização utilizado no trabalho. Por fim, as questões referentes ao paralelismo em SGBDOO's foram discutidas, fornecendo uma análise sobre os problemas que podem ser encontrados principalmente na otimização de consultas. O próximo capítulo apresenta os principais sistemas de construção de otimizadores e a metodologia proposta neste trabalho para a especificação de regras, utilizando esses sistemas, para a construção de otimizadores com paralelismo intra-operador.

Capítulo 4

Metodologia Baseada em Regras para Otimização de Consultas em SGBDOO Paralelo

Nesse capítulo apresenta-se a metodologia proposta para a construção de otimizadores para sistemas paralelos de banco de dados orientados a objetos. Essa metodologia tem como fundamento a utilização de sistemas de construção de otimizadores (*frameworks*). Assim, ela é dividida na fase de análise, independente do *framework*, e na fase de especificação, que segue indicações da fase anterior para a especificação do otimizador dentro de um *framework* específico.

Este capítulo detalha a fase de análise e os elementos de apoio a sua utilização e fornecidos pela metodologia. No capítulo seguinte, aborda-se a fase de especificação e detalhes de implementação. No entanto, antes de detalhar-se a fase de análise, o capítulo apresenta um resumo dos principais *frameworks* de construção de otimizadores e fornece uma análise sobre suas características.

4.1 Sistemas para construção de otimizadores

4.1.1 Introdução

Devido ao surgimento de diversas propostas de modelos de dados, na década de 80 começaram a surgir sistemas para especificação e geração de otimizadores de consultas para bancos de dados [GD87], [Loh88]. Esses sistemas tinham como objetivo facilitar e melhor estruturar o desenvolvimento de sistemas de banco de dados, em particular na implementação do módulo de processamento e

otimização de consulta. De um modo geral, um *framework* de otimizadores é um sistema computacional que recebe como entrada uma especificação de otimizador e fornece como saída um otimizador de consultas com as características especificadas. Alguns dos *frameworks*, como o OPTGEN [Feg97b], geram o código fonte do otimizador, outros, como o Cascades [Gra95], utilizam o conceito de *frameworks* orientados a objetos, através do qual o sistema é estendido ou modificado com a adição de novas classes. No entanto, a base de todos esses sistemas tem sido o modelo de regras de reescrita de termos. Esse modelo permite uma especificação clara e extensível para otimizadores.

Expressões em álgebras relacionais ou orientadas a objetos podem ser tratadas como termos, onde os nomes das relações ou classes são símbolos constantes e operadores são símbolos funcionais. Para melhorar as especificações de otimizadores, condições podem ser utilizadas juntamente com as regras de reescrita de termos para verificar se a regra será aplicada ou não a um termo. Essas condições são escritas na linguagem utilizada para implementar o otimizador e copiadas para o código gerado no momento da geração do otimizador. Além disso, elas podem fazer acesso a informações disponíveis em tempo de otimização, como informações estatísticas existentes no dicionário do banco de dados. O processo de reescrita de termos ocorre da seguinte forma:

1. Durante a otimização, as regras fornecem uma interface para comparação com os termos da árvore de consulta (*pattern matching)*.
2. Se a regra coincide com o termo, uma condição de aplicação especificada na regra é verificada.
3. Se a condição é bem sucedida, a regra é aplicada. Caso contrário, a regra é desprezada.
4. Se a regra é aplicada, um novo termo é gerado (reescrita) e adicionado ao espaço de busca do otimizador

Uma característica desejada para sistemas de desenvolvimento de otimizadores é a independência da especificação do otimizador em relação ao

modelo de dados, operadores, algoritmos de implementação de operadores e modelo de custos. Essa independência permite uma prototipação rápida e eficiente de novas técnicas de otimização.

Entre as principais propostas de sistemas de desenvolvimento de otimizadores podemos citar o Exodus [GD87], Volcano [GM93], Cascades e OPTGEN.

Nas próximas seções apresentamos as principais propostas, abordando as características mais importantes para o desenvolvimento do trabalho. Depois, da descrição das propostas, este trabalho apresenta uma análise com um quadro comparativo dessas propostas.

Este capítulo também apresenta, na seção 4.2, conceitos sobre a reescrita de consultas com paralelismo intra-operador. Finalmente, a seção 4.3 apresenta a técnica proposta para a construção de otimizadores. Essa técnica fornece uma abstração sobre o modelo de especificação de sistemas de otimização de forma que o trabalho possa ser aplicado a diferentes sistemas de construção de otimizadores.

4.1.2 Exodus

Esse sistema visa dar suporte a diversos modelos de dados conceituais, e assim não fornece um otimizador genérico para todos os sistemas. Ao invés disso, uma base de regras é utilizada, de forma que novos operadores, modelos de custos e métodos de acesso sejam informados ao sistema através da adição de novas regras. A abordagem dessa proposta divide um otimizador em duas partes: os componentes genéricos que desempenham o papel de mecanismo de busca e sistemas de suporte, e componentes dependentes do modelo de dados, tais como tipos especiais, operadores, e funções de custo. Fornecer um mecanismo simples e direto para a especificação desses componentes específicos é um dos principais objetos da proposta.

O gerador recebe como entrada um conjunto de operadores, um conjunto de métodos que realizam a implementação dos operadores e regras de transformação algébricas para a árvore de consulta. As regras lógicas são chamadas aqui de *regras de transformação*, e as regras físicas, que fazem a correspondência entre os operadores e os métodos que os implementam, são

chamadas de *regras de implementação*. Toda essa informação é armazenada em um chamado *arquivo de descrição do modelo*. Esse arquivo possui duas partes obrigatórias: A primeira, chamada de parte de declaração, é usada para declarar os operadores e métodos específicos, podendo também incluir código C adicional. A outra parte consiste das regras de transformação e implementação. Uma outra parte opcional permite informar código C que é adicionado ao código do otimizador gerado. A saída do sistema é um otimizador de consultas em C para um modelo de dados específico.

Para declarar os operadores e métodos as palavras-chave, *%operator* e *%method* são utilizadas, seguidas de um número que indica a cardinalidade e pelos operadores e métodos dessa cardinalidade. Por exemplo, abaixo, temos a declaração do operador *join* e seus métodos de acesso *hash_join*, *loops_join* e *cartesian-product*:

```
%operator 2 join
%method 2 hash_join   loops_join   cartesian_product
```

Além disso, a parte de declaração do arquivo aceita código C para definição de quatro tipos utilizados pelo gerador de otimizadores para especificação de argumentos e propriedades dos operadores e métodos associados aos nós da árvore de execução. São eles: OPER_ARGUMENT, METH_ARGUMENT, OPER_PROPERTY e METH_PROPERTY.

A segunda parte do arquivo é utilizada para a definição das regras lógicas e físicas. Nessa proposta, uma regra consiste de duas expressões e, opcionalmente, uma condição para execução da regra. As expressões consistem de operadores e listas de parâmetros, ou de métodos e seus argumentos no caso da expressão produzida por uma regra de implementação. Cada parâmetro pode ser uma outra expressão ou um número (indicando uma *stream* de entrada ou uma sub-consulta). As condições são procedimentos escritos em C e executados após o otimizador ter determinado que uma sub-consulta obedece ao padrão de uma regra, ou seja, a sub-consulta possui os mesmos operadores da regra e na mesma posição. Regras de transformação utilizam uma seta (->) para separar as expressões. A seta indica a direção da transformação e pode apontar nos sentidos

esquerda-direita, direita-esquerda, ou para os dois ao mesmo tempo. Um sinal de exclamação pode ser utilizado ao lado da seta para impedir que uma regra seja aplicada de forma redundante como em operações comutativas. Regras de implementação utilizam a palavra-chave *by*. No exemplo abaixo temos uma regra de transformação para a comutatividade do operador *join* e uma regra de implementação que utiliza o método *hash_join* para implementar esse operador.

```
join(1,2) ->! join(2,1)
join(1,2) by hash_join(1,2)
```

O outro exemplo apresentado abaixo, descreve a associatividade do operador *join*. A condição pode acessar os argumentos e propriedades dos operadores através das pseudovariáveis fornecidas pelo operador OPERADOR_1, OPERADOR_2, INPUT1, INPUT_2, etc. Cada uma dessas variáveis é uma estrutura que possui os campos oper_argument, oper_property, meth_argument e math_property. A condição é a expressão entre chaves. O procedimento *cover_predicate* (escrito em C) verifica se todos os atributos que ocorrem no predicado indicado pelo primeiro argumento da função são atributos das relações descritas no segundo e terceiro argumento.

```
join 7 (join 8 (1,2), 3)  <-> join 8 (1, join 7(2,3))
{
if ( NOT cover_predicate(OPERATOR_7 oper_argument,
                         INPUT_1   oper_property,
                         INPUT_2 oper_property))
       REJECT
}
```

4.1.3 Volcano

Esse sistema surgiu como uma extensão do Exodus e segue o mesmo princípio de geração de otimizadores a partir de uma especificação baseada em regras e extensível com relação ao modelo utilizado. Contudo, ele corrige alguns problemas encontrados no EXODUS e adiciona novas funcionalidades.

O sistema Volcano permite que propriedades físicas sejam utilizadas juntamente com expressões lógicas de forma a guiar as transformações das

expressões. Essa característica, inexistente no sistema EXODUS, contribuiu de forma significativa para a melhoria da estratégia de busca.

O Volcano inclui um novo tipo de operador físico chamado de *enforcer*. Esse tipo de operador não possui nenhum correspondente na álgebra lógica e é usado exclusivamente para garantir que algumas propriedades físicas, necessárias aos processamentos seguintes, sejam obtidas. Contudo, a principal melhoria do sistema Volcano diz respeito à extensibilidade da estratégia de busca. A estratégia de busca proposta para o sistema é denominada de *programação dinâmica direta*. Essa estratégia é ligada diretamente ao reconhecimento de padrões e código de aplicação de regras geradas a partir do modelo fornecido. Essa estratégia estende a técnica de programação dinâmica utilizada no sistema Starburst [Loh88], aplicada em consultas do tipo *seleção-projeção-junção*, para consultas algébricas em geral. Apesar do sistema já fornecer uma estratégia de busca específica, as tabelas *hash* são utilizadas para armazenar expressões lógicas e planos físicos. As operações executadas sobre essas tabelas são suficientemente genéricas para suportar diversos outros tipos de estratégias de busca.

Outra melhoria interessante é o uso de tipos abstratos de dados para a representação de custos. Dessa forma, o implementador pode modelar o custo dos planos de execução como uma variável, por exemplo, o tempo estimado; um registro, por exemplo o tempo de CPU e custo de E/S; ou qualquer outro tipo. Adicionalmente, operadores para funções aritméticas e comparações podem ser definidos sobre esses tipos abstratos. As propriedades lógicas e físicas dos planos podem ser modeladas também com o uso de tipos abstratos.

O trecho de código a seguir é um exemplo de modelagem com classes para um operator de leitura em disco. O exemplo apresenta um método para estimar o custo de um operador FILE_SCAN. Esse método recebe dois parâmetros do tipo LOG_PROP (Propriedade Lógica). O método tem como tipo de retorno a classe COST .

```
COST * FILE_SCAN::FindLocalCost (
                LOG_PROP    * LocalLogProp,
```

```
                    LOG_PROP ** InputLogProp)
{

   float Card = ((LOG_COLL_PROP *) LocalLogProp) -> Card;
   float Width = ((LOG_COLL_PROP *)
                 LocalLogProp)->Schema->GetTableWidth(0);
   COST * Result = new COST (
          Ceil(Card * Width) *
          ( Cm->cpu_read() + Cm->io() ));
   return (Result);
}
```

4.1.4 Cascades

O *Framework* de otimizadores Cascades [Gra95] é baseado em um processo *Top-down* de otimização e resolve muitos dos problemas encontrados no Exodus e no Volcano. Uma das grandes vantagens desse sistema é sua *interface* bem definida com o implementar através do paradigma da orientação a objetos. Todos os elementos do sistema, inclusive as regras, são implementados como objetos. A especificação do otimizador é feita de acordo com a abordagem de *framework* "caixa-branca", na qual as classes do sistema são especializadas através de herança e métodos virtuais são redefinidos e invocados pelos outros objetos do *framework* quando necessário (métodos *hook*). Para definir novas regras e operadores basta criar novas classes especializadas. A classe *RULE* fornece, por exemplo, um método *promise*() para avaliar se a regra deve ser aplicada ou não de acordo com as propriedades calculadas e esperadas. No Cascades, não existe separação entre módulos de regras lógicas e físicas. Como a regra é uma classe, cada sub-classe deve implementar os métodos *isLogical()* e *isPhysical()* para que o otimizador possa identificar a correta aplicação de cada regra. A sub-classe *ENFORCER* permite a definição de regras de garantia de propriedades físicas. As próprias propriedades lógicas e físicas são também representadas como objetos. O espaço de busca é representado pelo objeto MEMO. Esse objeto permite que o otimizador verifique se expressões já foram otimizadas através da técnica conhecida por *Memoization*.

A técnica de *Memoization* utiliza o objeto MEMO para armazenar as expressões lógicas otimizadas e os respectivos planos gerados para cada nível de otimização. Antes de uma expressão passar por uma transformação, o objeto

MEMO é verificado para garantir que a expressão ainda não foi otimizada. Se a expressão já existir na estrutura MEMO, os planos armazenados correspondentes são retornados e a expressão não passa por mais transformações.

O processo de otimização é realizado através de objetos do tipo *TASK*. Esses objetos coordenam a otimização de expressões e grupos de expressões através da aplicação de regras. Para cada expressão lógica são gerados sob demanda os possíveis planos lógicos e físicos equivalentes. O conjunto formado por esses planos é chamado de grupo e representado pela classe *GROUP*. O otimizador tem como objetivo otimizar cada grupo e o faz de forma *Top-down* a partir do grupo de expressões do nó raiz até o grupo dos nós folha da árvore de operadores.

Esse sistema é a base dos otimizadores de alguns dos sistemas de bancos de dados relacionais comerciais modernos, demostrando que a técnica de otimização *Top-down* conduzida por regras pode ser empregada com sucesso em aplicações reais.

4.1.5 OPTGEN

Este trabalho apresenta avanços com relação aos trabalhos descritos acima, principalmente no que diz respeito à forma de especificação do otimizador. A linguagem de especificação de otimizadores OPTL é introduzida como uma forma de capturar uma grande porção das informações necessárias à especificação de um otimizador de maneira declarativa. A ferramenta geradora de otimizadores OPTGEN [Feg97b] funciona como um pré-processador C++ que recebe como entrada um programa OPTL e gera um programa C++. Assim, um otimizador é especificado em um arquivo OPTL e o sistema OPTGEN gera um otimizador a partir dessa especificação.

OPTL é uma linguagem para especificação de otimizadores de consultas para bancos de dados. Essa linguagem possui como estrutura de dados básica uma árvore que representa ao mesmo tempo formas algébricas e planos de execução. Suporte a operações de manipulação de árvores e utilização de regras são dois pontos de destaque em OPTL.

Otimizadores são especificados em OPTGEN como sistemas de reescrita de termos modelados através de um conjunto de regras de reescrita. Cada regra representa a transformação de uma expressão algébrica em outra expressão algébrica ou plano físico. A especificação de um otimizador possibilita a declaração de duas listas de atributos: *inherited* e *synthesized*. Atributos *inherited* permitem a manutenção do contexto durante a reescrita de um termo, através da propagação de seus valores. Atributos *synthesized* são computados ao final de uma reescrita. Seus valores são passados e acumulados das folhas de um termo completamente reescrito até a raiz desse termo. Esses atributos substituem as propriedades de operadores e métodos apresentados no sistema EXODUS.

A estratégia de busca desse sistema é também *Top-down* e utiliza a técnica de *Memoization*. As expressões são otimizadas em níveis de forma equivalente à otimização de grupos no Cascades.

Um programa OPTL é um programa C++ estendido com construtores sintáticos que tornam a especificação de um otimizador uma tarefa mais simples.

```
C++code
          [ "%{"
                   C++code
                  "%inherited" name { "," name }
                   "%synthesized" "{" { name "=" C++type ";" } "}"
"=" C++code
                  { "%logical" { lrule } }
                     "%physical" { prule }
                      "%}"
           C++code ]
```

Figura 4.1 - Estrutura BNF de um programa OPTL

Um programa OPTL pode ter vários módulos de regras lógicas e um módulo de regras físicas (veja Figura 4.1). O otimizador gerado executa inicialmente as regras lógicas na ordem em que elas ocorrem e só depois aplica as regras físicas. Os dois tipos de regras fornecem estruturas de controles como laços e condições. Regras lógicas permitem a declaração de um *guard* que determina se a regra será disparada ou não.

Uma regra lógica tem sua forma expressa em BNF especificada abaixo, onde *#case* avalia a expressão em *expr* contra os padrões listados em *pattern*, *let* amarra as variáveis do tipo *Expr* em valores *Expr*, *#forall* define um *loop* que varre uma lista do tipo *list<expr>*. Cada regra lógica produz um termo, ou vários termos se houver um *#forall* na expressão. Uma lista de atributos *{attribute=pattern ...}* antes do cabeçalho de uma regra indica que cada atributo herdado deve combinar com o padrão indicado. Se o padrão especificado for complexo, essa lista funciona como um *guard*.

```
lrule ::=   [ inherited ] pattern [ guard ] "=>" body
          | [ inherited ] pattern [ guard ] "=" body
guard ::= ":" C++code
inherited ::= "{" { name "=" pattern ";" } "}"
body ::=    expr ";"
          | "let" name "=" C++code { "," name "=" C++code } "in" body
          | "#case" C++code { "|" pattern "=>" body } "#end" ";"
          | "#forall" name "in" C++code "do" body "#end" ";"
```

Figura 4.2 - Estrutura BNF de uma regra lógica OPTL.

Existem dois tipos de regras lógicas: reduções (indicadas pelo símbolo “=>”) e reescritas (indicadas pelo símbolo “=“). Se várias regras de redução são aplicáveis a um termo, então a primeira regra de redução é aplicada e as demais ignoradas; se não existe nenhuma regra de redução aplicável, então todas as regras de reescrita são aplicadas e os resultados acumulados. Esse processo é repetido dos termos para seus sub-termos até que nenhuma outra regra seja aplicável.

Uma regra física tem sua forma expressa em BNF de acordo com a Figura 4.3, onde a única diferença com relação às regras lógicas é a computação dos atributos *synthesized* ao final do corpo da regra. O termo gerado é um plano final e não é mais processado. Ou seja, ao contrário de uma regra lógica, um termo só é transformado por uma regra física uma única vez.

```
prule ::= [ inherited ] pattern [ guard ] "=" body
body ::= expr [ synthesized ] ";"
```

```
    | "let" name "=" C++code { "," name "=" C++code } "in" body
    | "#case" C++code { "|" pattern "=>" body } "#end" ";"
    | "#forall" name "in" C++code "do" body "#end" ";"
    synthesized ::= ":" "{" { name "=" C++code ";" } "}"
```

Figura 4.3 - Estrutura BNF de uma regra física OPTL.

4.1.6 Quadro comparativo entre geradores de otimizadores

A Tabela 4.1 mostra um comparativo sobre as características de três propostas de sistemas de construção de otimizadores. A contribuição do sistema EXODUS concentra-se na apresentação do paradigma de geração de otimizadores (modelo de especificação de otimizadores como entrada e código fonte como saída), separação entre álgebra lógica e física, separação de propriedades lógicas e físicas, o uso de regras de transformação e implementação e a ênfase em modularização. Contudo, esse sistema apresenta uma estratégia de busca fixa e ineficiente. Ele não é apresentado no quadro, dado que o sistema Volcano é uma extensão direta desse sistema com o objetivo de corrigir as deficiências existentes. O sistema Volcano, além de corrigir os problemas do EXODUS, ainda acrescentou novas características. Ele apresenta uma estratégia de busca mais eficiente, baseada em programação dinâmica. Essa proposta também apresenta facilidades para geração de otimizadores para sistemas paralelos através do operador *enforcer exchange*, apesar de não apresentar soluções específicas para a modelagem de arquiteturas paralelas e suas propriedades.

	VOLCANO	CASCADES	OPTGEN
Utiliza Base de Regras	Arquivo de descrição do modelo	Regras são artefatos de implementação em C++	Arquivo (Módulos) com especificação de regras
Modelo para Especificação de Regras	Conjunto de palavras-chaves	Utilização de classes para representação de regras	Linguagem própria (OPTL) baseada em SML com estruturas de controle e repetição
Acesso a propriedades do plano	Através de tipos abstratos de dados definidos pelo DBI	Através de objetos que representam as propriedades do plano	Através do conceito de *Atributos Gramaticais*
Suporte a Árvores e *Pattern-Matching*	Suporte a acesso a árvore e coincidência de padrões, mas ainda exige codificação.	Um objeto de regra permite a definição de um antecedente como um objeto com métodos de comparação	OPTL fornece estrutura para coincidência de padrões e navegação baseado em construtores gramaticais de alto-nível
Separação entre Regras Lógicas e Físicas	Esquema similar ao EXODUS, mas fornece uma melhor separação entre as expressões lógicas e físicas nas tabelas *hash*	Através de métodos nos objetos de regra	Através de módulos de regras lógicas e módulos de regras físicas expressas em OPTL.
Memoization	Tabela hash com transformações dos termos	Tabela MEMO com transformações dos termos	Tabela MEMO com transformações dos termos
Grau de Geração Automática de Código a partir da Especificação	Necessidade de codificação em C que é adicionado ao código gerado	Não existe geração de código, o otimizador é uma extensão do próprio Cascades através do conceito de *framework* "caixa-branca"	Maior grau de geração de código
Modelo de Otimização	Modelo baseado em custos	Modelo baseado em custos com busca exaustiva	Modelo baseado em custos
Estratégia de Busca	Não muito extensível. Inicialmente, utiliza programação dinâmica	Extensível através de hierarquias de classes e baseada em *top-down*.	Extensível através de reflexão e baseada em *top-down*.
Geração de otimizadores p/ sistemas paralelos	Utilização do operador *Exchange*	Utilização do operador *Exchange*	Ausente

Tabela 4.1 - Comparação entre sistemas de construção de otimizadores

O sistema Cascades apresenta sua principal contribuição através da extensão de seus antecessores Exodus e Volcano com a utilização do paradigma da orientação a objetos para a construção do otimizador.

O sistema OPTGEN apresenta sua principal contribuição na utilização de uma linguagem para especificação de otimizadores com grande suporte para manipulação de árvores e coincidência de padrões permitindo um maior grau de geração de código. No entanto, esse sistema não apresenta utilização com operadores paralelos como o EXGHANGE.

4.2 Reescrita de consulta com paralelismo intra-operador

O modelo de processamento proposto em [Has96] divide o problema de otimização paralela de consulta em duas fases: 1) Reordenação de Junções e Seleção de Métodos, 2) Extração do Paralelismo e Escalonamento do Plano. A primeira fase minimiza o trabalho total levando em consideração o paralelismo e seu custo de comunicação com reparticionamento de dados; a segunda fase visa dividir este trabalho através dos nós de processamento levando em consideração o paralelismo inter-operador.

A primeira fase utiliza um algoritmo baseado em coloração de árvores para selecionar métodos (operadores físicos) para cada operador lógico. A seleção desses métodos objetiva minimizar os custos relativos ao paralelismo com reparticionamento de dados. Em [Has96], é fornecida uma abstração sobre as propriedades físicas de um PEC através do conceito de cor. Com essa abstração, a tarefa de otimizar um plano e modificar suas propriedades físicas é representada como estratégias de coloração de árvores.

O custo mínimo de uma árvore com anotações de consulta é obtido através da estratégia a seguir (denominada de *EstrategiaCalculoOptc*). A Equação 1 exibe essa estratégia que pode ser apresentada da seguinte forma: O custo mínimo de um nó (*Optc*) para um valor de propriedade física *a* (uma cor, ou uma restrição de saída a) é obtido recursivamente através do menor custo de todas as estratégias aplicáveis ao operador representado pelo nó mais o custo mínimo dos nós de entrada mais o custo para *recolorir* (reparticionar, reordenar e reindexar) os dados produzidos pelos nós de entrada de acordo com o valor de propriedade física *a*. O conjunto *S* é conhecido como *Strategies(i, a)*, ou seja, são as estratégias que podem ser aplicadas ao nó *i* (operador lógico) para produzir a propriedade física *a* (restrição de saída).

$$SubplanCost(i,a) = \sum_{j=1}^{k} min_{p \in P}\left[Optc(\alpha j, p) + repartition\left(R_j^S, p, inpCol(s, A, j) \right) \right]$$

$$Optc(i,a) = min_{S \in S}\left[StrategyCost\left(s, R_1^S, ..., R_n^S \right) + SubplanCost(i,a) \right]$$

Equação 1 - Cálculo do Optc

Na equação, a função *StrategyCost* indica a estimativa do custo do operador físico de acordo com as informações e estatísticas existentes sobre as relações envolvidas. *P* indica as configurações alternativas da propriedade física. *InpCol* indica o valor de propriedade física que a estratégia *s* precisa receber do sub-plano *j* para produzir o valor *a* para a mesma propriedade física. Esse valor esperado indica a restrição de entrada para o sub-plano. O valor em *a* indica a restrição de saída.

Com base nessa estratégia para cálculo do custo mínimo *Optc*, o seguinte algoritmo é proposto para obter o plano e seu custo mínimo (*Opt*):

Algoritmo *ExtendedColorSplit*

1. **Para cada** nó **i** em ordem pós-fixada **faça** o passo 2
2. Utilize *EstrategiaCalculoOptc* para calcular ***Optc(i, a)*** e ***OptcStrategy(i, a)*** para cada cor **a** $\in$**C**.
3. Seja ***r*** a raiz da árvore e **a** uma cor tal que ***Optc(r, a)*** $\leq$ ***Optc(r, c)*** para todas as cores **c** $\in$**C**.
4. A cor ótima para ***r*** é **a** e a estratégia (método) ótima é ***OptcStrategy(r, a)***.
5. Para cada nó diferente da raiz **faça**, em ordem pré-fixada, o passo 6
6. Calcule cores ótimas e suas estratégias aplicando (de forma *top-down*) *EstrategiaCalculoOptc* de forma inversa.

Esse algoritmo tem seu pior caso com um tempo de execução de $\mathbf{nS|C|^2}$ onde **S** é o número de estratégias (operadores físicos) **|C|,** o número de configurações

de propriedades físicas (cores) disponíveis e **n**, o número de nós da árvore. Já que **n** e **S** são tipicamente pequenos, o tempo de execução é dependente de |**C**|, que pode ser mantido pequeno observando que:

- Nenhuma estratégia produz tuplas de saída com índices. Assim nós internos não devem considerar a propriedade física *índice*.
- Apenas as cores úteis aos nós subseqüentes devem ser consideradas.

O algoritmo tem como objetivo selecionar operadores físicos que reduzam os custos de comunicação. Isso ocorre quando um operador atende às restrições de entrada-saída existentes, evitando o custo adicional de reparticionamento dos dados (recolorir operadores). A base para essa estratégia está nos experimentos sobre execução paralela realizados em [Has96] que identificaram que:

- Custos de inicialização dos processos de execução de operadores podem ser desprezados em relação ao custo de comunicação entre os processos
- Custos de comunicação consistem do custo de uma CPU enviar e receber mensagens
- Custos de comunicação podem exceder os custos de operadores como *scan*, *join* ou *grouping* tornando um plano paralelo menos eficiente que um plano sequencial.

[Has96] sugere a utilização do seu algoritmo das seguintes formas:

1. Como um "post-pass" em um otimizador convencional, de forma a não alterar as fases anteriores.
2. Em substituição às fases equivalentes de um otimizador convencional

Propõe-se aqui uma terceira abordagem, na qual o algoritmo *EstrategiaCalculoOptc* seja utilizado no contexto de um *framework* baseado em regras para a construção de otimizadores. Dessa forma, a implementação é realizada no meta-nível, proporcionando a reutilização da especificação em

diversos otimizadores e possibilitando uma definição clara e extensível dos cálculos dos custos e seleção de operadores.

A abordagem deste trabalho utiliza propriedades físicas calculadas e esperadas para guiar a estratégia através do espaço de busca. O caráter recursivo de *EstrategiaCalculoOptc* é muito bem assimilado pela aplicação de regras às entradas de operadores e repasse dos valores das propriedades entre eles.

Abaixo apresentamos alguns conceitos utilizados no algoritmo e como esses conceitos podem ser modelados em um sistema de regras de reescrita de termos.

Optc(i,A) – É o custo mínimo de uma sub-árvore com raiz no nó **i** tal que esse nó tenha como saída a cor **A**.

Modelo de Regras – É o valor mínimo do atributo *cost* da regra aplicável ao operador lógico correspondente ao nó **i** e que produz propriedades físicas compatíveis com o conjunto de atributos físicos **A**.

Opt(i) – É o custo mínimo de uma sub-árvore com raiz no nó **i** independentemente da cor.

Modelo de Regras – É o valor mínimo do atributo *cost* entre todos os sub-planos (referentes ao operador correspondente ao nó **i**) produzidos pelas regras aplicáveis ao operador lógico **i** independentemente das outras propriedades físicas produzidas.

inpCol(s, A, j) – É o padrão de cor necessário a uma estratégia **s** (para o operando de entrada **j**) para que o padrão da cor de saída obedeça a **A**.

Modelo de Regras – São os valores esperados para os atributos físicos produzidos no sub-termo de entrada **j** de um termo (operador lógico) para que o operador físico (estratégia **s**) gerado pela regra tenha atributos físicos de acordo com o padrão **A**.

Strategies(i, A) – É o conjunto de estratégias aplicáveis ao operador representado pelo nó **i** e no qual as restrições de entrada-saída geram **A** como cor de saída.

Modelo de Regras – Conjunto de regras físicas aplicáveis ao operador **i** tal que *enforcer*s e passagem de contextos de atributos gerem as propriedades físicas em **A.**

OptcStrategy(i, A) – É alguma estratégia na qual o valor mínimo é atingido.

Modelo de Regras – Plano referente ao nó **i** com menor valor para o atributo *cost* gerado pelo conjunto de regras físicas aplicáveis ao termo correspondente.

recolor(R^a, c_{old}, c_{new}) – Custo para recolorir a relação **R** da cor c_{old} para a cor c_{new}.

Modelo de Regras – Função de suporte que estima (utilizando informações do catálogo R^a) o custo para transformação das propriedades físicas (reparticionar os dados, por exemplo) do plano referente a uma relação **R** de c_{old} para c_{new}.

4.3 Metodologia baseada em *frameworks* com regras para construção de otimizadores

4.3.1 Análise da álgebra paralela orientada a objetos

Nessa seção apresentamos a metodologia (Figura 4.4) proposta para construção de otimizadores para sistemas paralelos. Como visto na seção anterior, a complexidade de um otimizador de consultas exige a utilização de um *framework* que facilite o seu desenvolvimento e extensibilidade. Essa metodologia fornece um modelo para construção de otimizadores para sistemas paralelos com a utilização de *frameworks* baseados em regras, abstraindo os detalhes específicos de cada um deles. O modelo utilizado concentra-se na utilização de paralelismo intra-operador e nos custos inerentes ao reparticionamento de dados.

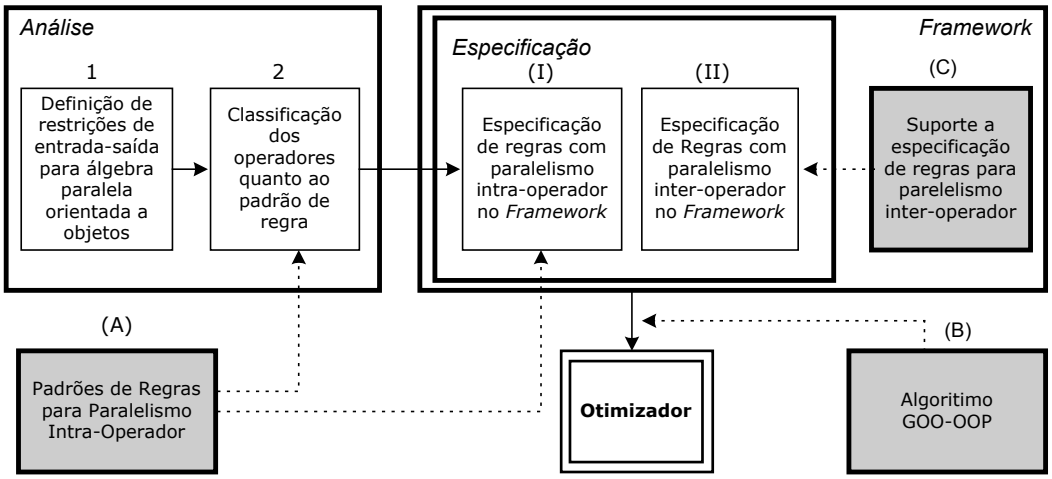


Figura 4.4 - Metodologia para construção de otimizadores

A metodologia para construção do otimizador divide o processo na fase de análise e especificação do otimizador. Os itens 1, 2, I e II são realizados no meta-nível, ou seja, ao nível dos sistemas utilizados para construir sistemas de otimização. Já o item B é um algoritmo que deve ser introduzido ao otimizador construído para fornecer suporte a reordenação de operadores. OPTGEN e Volcano permitem a declaração de algoritmos no meta-nível e no momento da geração do otimizador esses algoritmos são incorporados ao sistema gerado. O item A é a abstração de regras (meta-meta-nível) fornecida pelo trabalho para a especificação de regras no meta-nível.

A fase de análise é composta pela análise da álgebra física paralela para especificação das restrições de entrada-saída dos operadores com paralelismo intra-operador e na classificação dos operadores quanto ao padrão de regras (itens 1 e 2 da Figura 4.4). Essa fase também compreende a análise da álgebra para identificação de unidades atômicas da álgebra e identificação das restrições de tempo de acordo com a técnica de extração de paralelismo proposta em [Sam98]

A fase de especificação é dividida na especificação para paralelismo intra-operador e inter-operador (itens I e II da Figura 4.4). O item (I) consiste na definição de regras para mapeamento dos operadores da álgebra lógica em operadores da álgebra física paralela com o uso das restrições e da classificação

dos operadores quanto ao padrão de regras. O item (II) consiste na definição de regras de extração de paralelismo inter-operador. Os itens I e II podem ser implementados de diferentes formas nos *frameworks* existentes. No Cascades, por exemplo, cada item pode ser implementado através de uma hierarquia de sub-classes de RULE. Já no OPTGEN, o conceito de módulo de regras pode ser utilizado para a declaração e processamento separado de cada conjunto de regras.

O modelo proposto fornece três elementos de suporte para a metodologia (itens *a, b* e *c* da Figura 4.4). Os padrões de regras (item *a*) definem como as regras devem ser construídas para cada classificação de operadores. Esses padrões são baseados em estratégias de busca *Top-down* e são responsáveis pela abstração do algoritmo de seleção de operadores com particionamento de dados proposto em [Has96].

O algoritmo GOO-OOP (item *b*) é incorporado ao *framework* de forma que a reordenação de operadores seja transparente com relação à especificação do otimizador e possa ser reutilizado em diversas especificações. Essas reordenações são baseadas nos custos da estratégia e de reparticionamento de dados. Esse algoritmo estende o GOO-OO proposto em [Feg98], a partir da estratégia apresentada em [Has96], mas agora no contexto de uma especificação baseada em regras.

O item (*c*) fornece transparência entre os itens (I) e (II). A fase de extração do paralelismo é baseada na implementação das estratégias selecionadas para o plano físico. Através desse suporte, essa fase pode ser executada de forma transparente em relação à seleção de estratégias e reordenação de operadores.

4.3.2 Análise da álgebra paralela orientada a objetos

Para a construção de regras físicas que considerem o particionamento dos dados e os custos de particionamento, uma análise sobre o código que implementa os operadores da álgebra orientada a objetos selecionada deve ser realizada. Este trabalho utiliza a álgebra monóide (ver seção 3.4.2). A partir da análise determina-se quais as restrições de entrada e restrições de saída de cada operador com relação às propriedades físicas selecionadas: índice, ordem e

principalmente o particionamento. Como resultado, obtêm-se a Tabela 4.2 que é utilizada para a definição das regras de reescrita. A estratégia *Hash-Loop* foi introduzida à álgebra física original para adicionar possibilidades de replicação de dados no processamento de consulta. A notação <p(r): X> na Tabela 4.2 indica que o operador *Hash-Loop* replica os dados da corrente de objetos que possuem a coleção aninhada de objetos, para os nós que possuem os itens da coleção.

Operador Físico	***Saída***	***Entrada 1***	***Entrada 2***	***Condições Adicionais***
Nested-Loop	<p: X>	<p: X>	<p: X>	Predicado do Join em X
Merge-Join	<p: X>	<p: X>	<p: X>	Predicado do Join em X
Block-Nested-Loop	<p: X>	<p: X>	<p: X>	Predicado do Join em X
Hash-Loop	<p: X>	<p(r): X>	<p: X(*)>	replicação sobre objetos que possuem coleção
Nest	<p: X>	<p: X>		X é um atributo de aninhamento
Group-By	<p: X>	<p: X>		X é um atributo de agrupamento
Table-Scan	<p: X>	<p: X>		X é o atributo de particionamento do *extent*
Index-Scan	<p: X>	<p: X>		X é o atributo de particionamento do *extent*
Map	<p: X>	<p: X(*)>		
Materialize	<p: X>	<p: X(*)>		
Reduce	<p: X>	<p: X(*)>		
Sort	<p: X>	<p: X(*)>		
Unnest	<p: X>	<p: X(*)>		
Union	<p: X>	<p: X(*)>	<p: X(*)>	

Tabela 4.2 - Restrições de entrada e saída para operadores físicos

De acordo com a Tabela 4.3, dividimos os operadores físicos da álgebra selecionada em três tipos básicos com relação ao tipo de regra para particionamento de dados:

a) *Operadores de base*: Fazem acesso diretamente às extensões (*extents*) das classes do esquema e são processados de acordo com o particionamento nelas definido

b) *Operadores condicionados*: Operadores que são executados de acordo com um particionamento específico e assim obrigam o reparticionamento dos planos de entrada

c) *Operadores Livres*: Operadores que podem ser executados sobre qualquer particionamento e assim podem ter planos alternativos gerados para cada particionamento possível

Os operadores da álgebra utilizada no trabalho são classificados de acordo com a divisão acima (tipo de regra) na Tabela 4.3.

Operador	**Base**	**Condicionado**	**Livre**
Table_scan	**X**		
Index_scan	**X**		
Reduce			**X**
Nested_loop		**X**	
Merge_join		**X**	
Hash_Loop			**X**
Block_Nested_Loop		**X**	
Sort			**X**
Unnest			**X**
Nest		**X**	
Groupby		**X**	
Union			**X**
Map			**X**
Materialize		**X**	

Tabela 4.3 - Classificação dos operadores quanto ao tipo de regra

4.3.3 Padrões de regras com paralelismo intra-operador

Para facilitar a definição da metodologia proposta, iremos definir abaixo uma abstração do modelo de regras de reescrita de termos para a especificação de otimizadores. Esse modelo visa a agregação das funcionalidades e elementos identificados nos sistemas analisados (Exodus, StarBurst, Volcano, Cascades e OPTGEN entre outros) de forma que as técnicas aqui definidas possam ser aplicadas mais facilmente a uma grande variedade de sistemas.

Uma regra de reescrita de termos para especificação de otimizadores deve fornecer os seguintes elementos:

- **Cabeçalho da Regra**: Esse cabeçalho consiste do termo que deve ser comparado com os operadores da consulta para que a regra seja aplicada aos operadores que coincidirem com o termo do cabeçalho. Esse termo pode utilizar padrões (*patterns*) para facilitar a comparação com os operadores.
- **Corpo da Regra**: Determina o termo que será gerado com aplicação da regra.
- **Condição de Aplicação**: Expressão que determina se a regra será aplicada ou não. Após o cabeçalho da regra coincidir com algum operador, a condição é verificada e a regra só é aplicada se a verificação for bem sucedida.
- **Restrição de Entrada:** Identifica valores esperados para propriedades das entradas de um operador. Essa restrição pode ser utilizada para verificar a aplicabilidade da regra.
- **Restrição de Saída:** Identifica valores esperados para propriedades da saída do termo produzido pela regra.
- **Propriedades Lógicas:** Conjunto de atributos que descrevem características lógicas dos termos, como nome de objetos e seus atributos. Essas propriedades podem ser passadas entre um termo e outro.
- **Propriedades Físicas:** Conjunto de atributos que descrevem características físicas dos termos, como a ordem de uma corrente de dados

(*stream*) produzida ou um atributo de particionamento dos dados em uma base com paralelismo.

Uma regra pode ser lógica, realizando transformações sobre operadores lógicos ou físicos, selecionando os métodos (operadores físicos) que implementam os operadores lógicos e garantindo propriedades. As regras físicas que são utilizadas para garantir propriedades físicas exigidas por um plano são conhecidas como *enforcers*.

Por exemplo, a regra OPTL abaixo modela a geração do método MERGE_JOIN para o operador lógico *join*. Na primeira linha da regra, temos uma *restrição de saída* que requer uma determinada ordem (identificada pela propriedade física *order*) para a *stream* produzida pelo termo. Na segunda e terceira linhas uma *restrição de entrada* requer uma determinada ordem (propagada pela função *required_order*) para os operandos de entrada do operador lógico *join* (identificados pelas variáveis com padrões *'x* e *'y*). O *cabeçalho* da regra é composto pelas linhas 2, 3 e 4, onde o operador *join* e seus sub-termos serão comparados aos termos da consulta para aplicação da regra e ligação (*binding*) das variáveis existentes. Nas linhas 5 e 6 (a expressão entre : e =) temos uma *condição* que verifica se o predicado determina um *equijoin* e se a ordem esperada é atendida pelo operando (sub-termo) da esquerda. Se a condição for atendida, a regra é aplicada; caso contrário, ela é rejeitada. No *corpo* da regra (código após =, nas linhas de 7 a 14), o termo produzido pela regra é gerado (MERGE_JOIN), e atributos do plano, como tamanho, custo e ordem real, são calculados (linhas 11, 12 e 13).

```
1.   { order=`expected_order; }
2.   join(`x<-
     {order=`(required_order(pred,all_tables(x),all_tables(y))); },
3.        `y<-
     {order=`(required_order(pred,all_tables(y),all_tables(x))); },
4.        `pred)
5.      : equijoinp(pred,all_tables(x),all_tables(y))
6.        && subsumes(expected_order,^x.order)
7.      = MERGE_JOIN(`x,`y,`pred,
```

```
8.          `(required_order(pred,all_tables(x),all_tables(y))),
9.          `(required_order(pred,all_tables(y),all_tables(x))))
10.         : {
11.             size = int((^x.size+^y.size)*selectivity(pred));
12.             cost = ^x.cost+^y.cost+(^x.size+^y.size);
13.             order = ^x.order;
14.           };
```

Assim, utilizaremos a seguinte notação para representar uma regra: *R=(E, S, CA, TP, CB, CO)* , onde *E* representa as restrições de entrada, *S* as restrições de saída, *TP* o tipo da regra, *CB* o cabeçalho, *CA* a condição de aplicação e *CO* o corpo da regra. As restrições de entrada (*E*) identificam as propriedades que devem ser fornecidas pelos sub-termos (operandos) do termo que está sendo otimizado. A restrição de saída (*S*) identifica as propriedades que devem ser disponibilizadas no termo gerado após a transformação (*CO*). A condição de aplicação (*CA*) determina se a regra será disparada ou não. O cabeçalho (*CB*) determina uma expressão que será comparada pela estratégia de busca do otimizador a um termo do plano (operador e seus parâmetros) para verificar se a regra se aplica ou não a esse termo. O tipo (*TP*) indica se a regra é lógica ou física (livre, condicionada, base, ou garantia, de acordo com a classificação dos operadores na Seção 4.3.2). O corpo da regra (*CO*) identifica a nova expressão, operador, parâmetros e cálculo de atributos, gerados pela transformação disparada pela regra.

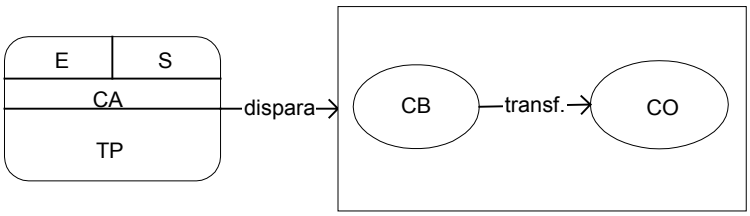


Figura 4.5 - Abstração de Regra

O conjunto de propriedades físicas utilizadas no modelo proposto em [Has96] é composto pelo atributo de particionamento, o atributo de ordenação e o índice. Vamos chamar esse conjunto de PFR (propriedade físicas para restrição), juntamente com uma valoração para cada propriedade, pode ser facilmente

generalizado para mais atributos, mas iremos utilizar esses atributos como nosso conjunto mínimo de propriedades físicas para as restrições de entrada e restrições de saída. Uma restrição (de entrada ou saída) pode ser vista como um conjunto PFR que deve ser satisfeito para a efetiva aplicação da regra. Quando as propriedades físicas esperadas nos sub-termos de entrada de uma regra não são atendidas, essas propriedades de entrada têm que ser garantidas para que a regra seja aplicada. Ou seja, o termo de entrada que não atende à restrição de entrada (propriedades físicas esperadas) tem que ser reescrito como um termo que atenda essas propriedades. Para garantir essas propriedades, temos um custo adicional, por exemplo, para reordenar um *stream* de entrada de acordo com a ordem esperada. Iremos denominar esse custo de *custoGarantia(PFR, RE)*.

A seguir, os padrões de regras para a especificação de otimizadores com paralelismo são apresentados. Esses padrões são definidos de acordo com a abstração de regra apresentada na Figura 4.5. Os padrões de regras são definidos para suportar a estratégia de busca e são baseados na classificação dos operadores dada na Tabela 4.3.

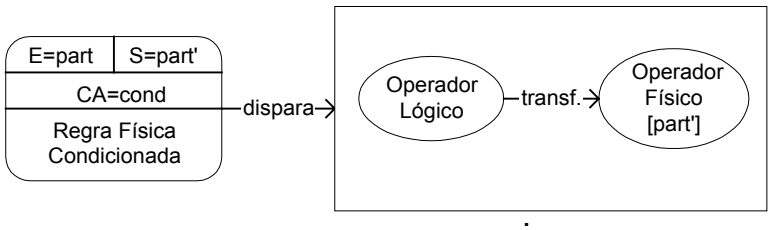


Figura 4.6 - Padrão para regra física condicionada

O padrão para regra física condicionada (Figura 4.6) é aplicado aos operadores condicionados e determina que a regra deve passar para o sub-plano a partição exigida pelo operador. A passagem dessa informação é feita através do contexto de otimização com o uso de propriedades físicas esperadas. A partição exigida pelo operador é determinada com o uso das restrições de entrada-saída da álgebra utilizada (Tabela 4.2). Na figura essa propriedade física de entrada é representada em *E* através de *part*. Essa partição (*part*) é enviada ao sub-plano como propriedade necessária para que o operador produza o resultado (saída) particionado de acordo com S=*'part*. Essa passagem de contexto permite um processo de otimização *Top-Down*.

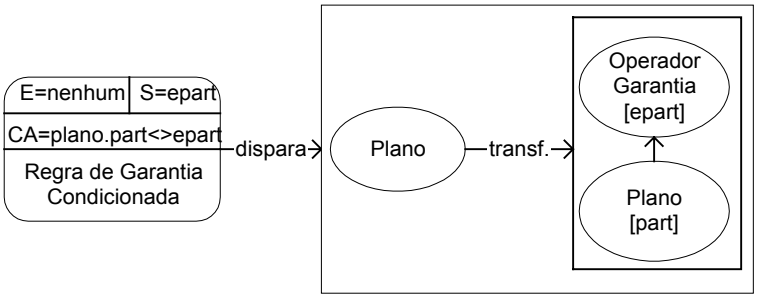


Figura 4.7 - Padrão para operador de garantia condicionado

O padrão para regra de garantia condicionada (Figura 4.7) é aplicado aos operadores *enforcers*. Uma regra construída com esse padrão só é aplicada quando recebe uma propriedade física requisitada não-vazia. Para que essa regra gere uma transformação no espaço de busca, a condição de aplicação (*CA*) tem de ser satisfeita, evitando que um plano já particionado de acordo com a propriedade requisitada seja reparticionado pela mesma propriedade. A transformação insere um operador de garantia a frente do sub-plano representado pelo operador lógico. Esse operador de garantia reparticiona os dados, garantindo que o plano terá o particionamento solicitado pelo operador físico condicionado responsável por consumir os dados produzidos. A regra envia para o sub-plano um contexto que sinaliza que o operador de garantia já foi aplicado (*E=nenhum*).

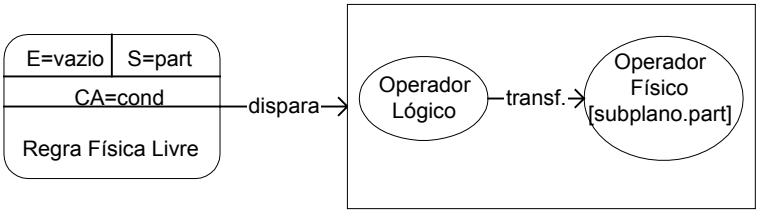


Figura 4.8 - Padrão de regra para operador físico livre

O padrão para regra física livre (Figura 4.8) é aplicado aos operadores físicos livres e determina que as regras construídas com esse padrão devem notificar o sub-plano com a propriedade física vazia. Essa notificação permite que o sub-plano escolha a melhor configuração de particionamento no seu nível,

comportando-se de acordo com um processo *Bottom-Up*. O plano gerado no espaço de busca é anotado com o particionamento oriundo do sub-plano (*subplan.part*).

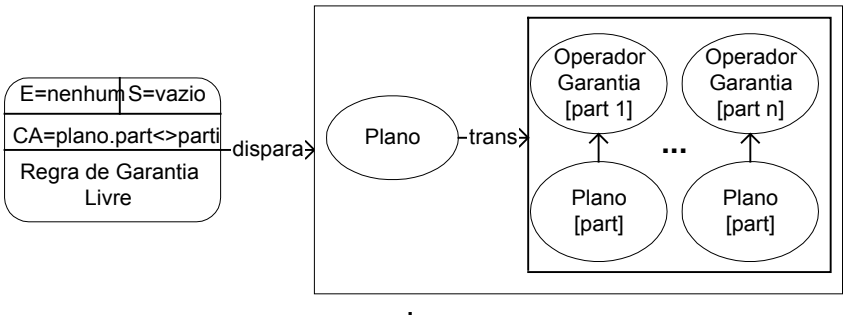


Figura 4.9 - Padrão para regra de garantia livre

O padrão para regra de garantia livre (Figura 4.9) é aplicado aos operadores físicos livres. A regra construída com esse padrão é aplicada quando recebe um contexto de particionamento vazio. Assim, a regra fica livre para escolher os particionamentos a gerar. No pior dos casos, ela pode gerar um plano para cada possível particionamento da coleção de dados. No entanto, técnicas de *prunning* podem ser utilizadas para selecionar os particionamentos mais promissores [SMB01]. Essas técnicas podem considerar, por exemplo, quais particionamentos fornecem um melhor balanceamento de carga, evitando *data skew*. O conjunto de planos gerados, um para cada partição selecionada, será processado como entrada no próximo nível de otimização. A regra envia para o sub-plano um contexto que sinaliza que o operador de garantia já foi aplicado (*E=nenhum*).

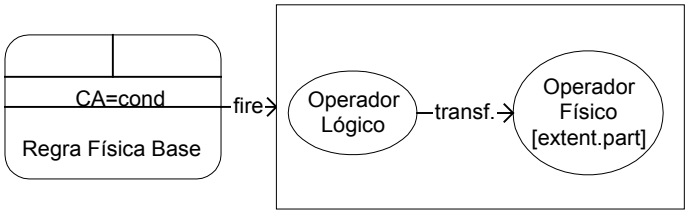


Figura 4.10 - Padrão para regra física de base

O padrão para regra física de base (Figura 4.10) é a base de toda a recursão do processo de otimização. Uma regra construída a partir desse padrão utiliza o meta-esquema para determinar qual a partição física definida na extensão da classe. Assim, o espaço de busca é transformado através da adição de um plano físico anotado com o mesmo particionamento da extensão (*extent.part*). Essa partição vai ser levada de forma *bottom-up* através dos estágios mais altos de otimização até que uma regra de garantia decida trocar o particionamento adicionando um operador de garantia a frente do plano.

Baseado na Equação 1, proposta em [Has96], o custo de um operador no modelo de regras com paralelismo intra-operador é obtido através do custo calculado na aplicação da regra que garante os valores esperados das propriedades físicas dos operadores de entrada mais o custo estimado na aplicação da própria regra. As propriedades físicas referentes ao paralelismo particionado devem ser calculadas, para cada tipo de padrão de regras, de acordo com a especificação abaixo:

Cálculo de propriedades para regra física de operador de base

```
partição=obter_particao(<argumento referente a extensão da classe>)
custo=funcao_estimativa_custo(<argumento referente a extensão da
cálculo das  demais propriedades físicas
```

Cálculo de propriedades para regra de garantia de operador condicionado

```
partição=partição referente à condição adicional
custo=subplano.custo +
     custoGarantia(subplano.partição, partição referente à condição
                                                        adicional )
cálculo das  demais propriedades físicas
```

Cálculo de propriedades para regra de garantia de operador livre

```
< Para cada partição_saída em Conjunto_Partições_Possíveis Faça>
  partição=partição referente à condição adicional
  custo= subplano.custo +
         custoGarantia(subplano.partição, partição_saída)
     cálculo das  demais propriedades físicas
< End Para >
```

4.3.4 Estratégia de Busca

Este trabalho utiliza uma estratégia de busca conduzida por regras e baseada em custos fornecidas pelos *frameworks* de construção de otimizadores. A estratégia de busca desses sistemas é baseada no modelo *top-down*. No entanto, para que essa estratégia possa suportar os elementos necessários para a otimização de planos para arquiteturas paralelas, este trabalho propõe que a técnica de otimização proposta em [Has96] seja estendida para o modelo de regras e que os elementos referentes aos custos do paralelismo sejam incorporados à estratégia do *framework*. Ou seja, utiliza-se o modelo de custos proposto em [Has96] para guiar as transformações realizadas pela estratégia de busca. A estratégia de busca está definida no meta-nível e, assim, os otimizadores construídos herdam a estratégia definida. O espaço de busca nos dois principais *frameworks* (Cascades e OPTGEN) é conhecido como MEMO. O processo é realizado em níveis de forma *Top-down*, iniciando-se da raiz da árvore de operadores até chegar às folhas. A cada nível, todas as regras definidas são aplicadas à expressão correspondente à sub-árvore do nível corrente de otimização. As regras produzem expressões que são adicionadas ao espaço de busca. No entanto, antes das regras de um nível serem aplicadas, a estratégia de busca verifica se a expressão já foi otimizada e se já está no espaço de busca. Se estiver, a expressão do nível seguinte é selecionada. Se não, a expressão é otimizada e as novas expressões geradas são adicionadas ao espaço de busca.

Como visto anteriormente, as regras dividem-se em regras lógicas e físicas. As regras físicas nos otimizadores para sistemas não-paralelos geram, a partir dos operadores lógico, expressões com diferentes estratégias de implementação desses operadores. Na estratégia de busca proposta neste trabalho, utiliza-se as regras físicas para gerar planos com diferentes estratégias e com diferentes alternativas de particionamento de dados. Esses planos e configurações de particionamento são utilizados pelas regras do próximo nível para geração de novos planos e configurações.

Dessa forma, adaptamos a proposta de otimização procedimental abordada em [Has96] para um modelo declarativo e baseado em regras, que fornece uma abordagem mais clara e extensível para especificação de otimizadores.

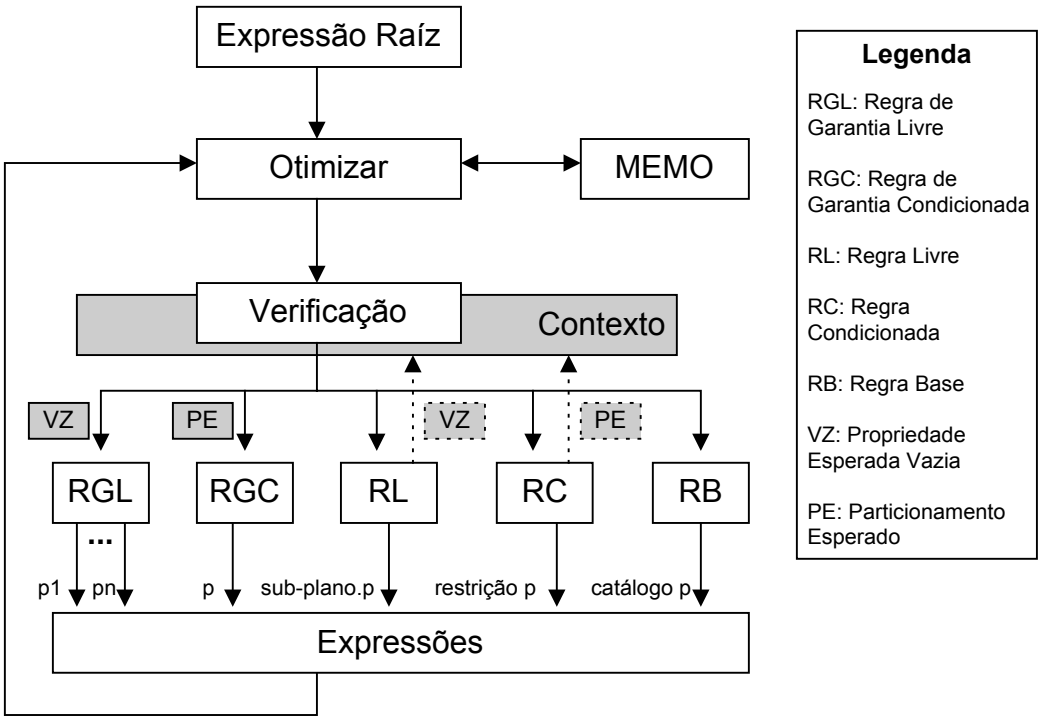


Figura 4.11 - Estratégia de busca proposta

A estratégia de busca estendida para utilização em um *framework* baseado em regras funciona da seguinte forma, sumarizada na Figura 4.11:

1. Para cada termo a partir da raiz da expressão, as regras são verificadas contra o termo de acordo com os seus cabeçalhos e com a prioridade determinada.
2. Se o operador é classificado como livre, então passa-se *vazio* como valor requerido para a propriedade física que representa a partição. Se é classificado como condicionado, passa-se o valor requerido pelo operador físico.
3. Para cada regra aplicável, o mesmo processo é utilizado para os termos operandos (sub-árvore).
4. Esse processo *top-down* é aplicado até se chegar aos termos para os operadores de base, onde as regras aplicáveis geram planos para o particionamento existente na extensão (*extent*). Esses planos são fornecidos como parâmetros para os termos nos níveis superiores da

expressão, e o custo é acumulado recursivamente de forma *bottom-up*. Esses planos são repassados para os níveis superiores até se chegar à raiz.

5. A cada iteração do processo, as regras de garantia (*enforcers*) são verificadas, antes das demais regras físicas, para garantir as restrições enviadas pelas regras físicas da iteração de nível superior. Nessas regras, encapsulam-se os detalhes do paralelismo intra-operador. Assim, os custos de reparticionamento e geração de planos alternativos para as várias partições disponíveis são tarefas executadas por essas regras.
6. O otimizador, por fim, seleciona o plano com menor custo (atributo *cost*), sendo que seu valor é calculado de acordo com a recorrência apresentada na Equação 1 na Seção 4.2. O plano com menor custo possui, para cada termo, informações sobre as estratégias e partições sobre as quais os operadores serão processados. O plano com menor custo é o plano com menor *Optc* entre todos os particionamentos e assim, o plano ótimo.
7. Após a seleção dos operadores físicos com o objetivo de minimizar a comunicação dos dados relativa ao reparticionamento, o otimizador realiza a extração de paralelismo inter-operador. Para isso, regras de extração de paralelismo recebem no cabeçalho operadores físicos e os transformam em dois ou mais operadores atômicos. Essa transformação não altera as propriedades físicas do plano como custo e tamanho. Seu objetivo é fatorar o plano para fornecer uma interface entre o otimizador e o escalonador de consultas.

Para exemplificar o funcionamento de uma estratégia de busca em um contexto de paralelismo com particionamento de dados, considere a consulta abaixo:

```
select struct(E: e.name, S: sum(select p.budget
  from p in e.dept.projs))
from e in Employees
where e.isManager("senior")
```

Essa consulta recupera, para cada nome de gerente do tipo *Senior*, o valor total do orçamento de todos os projetos pertencentes ao departamento controlado

por esse gerente. Agora considere o plano lógico de execução gerado para essa consulta (Figura 4.12).

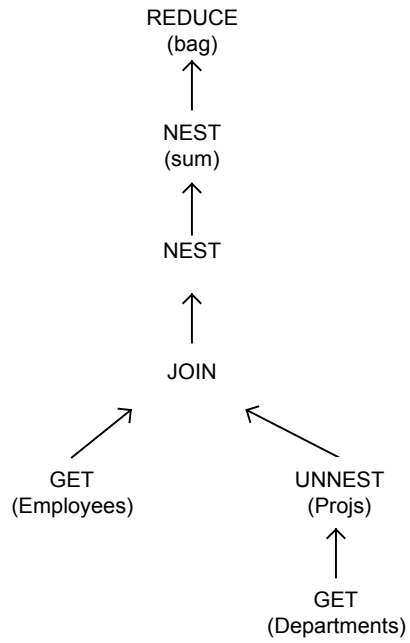


Figura 4.12 - Exemplo 1 de Plano Lógico

No plano da Figura 4.12 pode-se verificar que o operador *get* lê os dados armazenados nos *extents Employees* e *Departments*. Antes de realizar a junção entre essas duas informações, o operador *unnest* expande a informação contida nos objetos *Departments* para construir uma coleção que associa cada departamento a seus projetos (*Projects*). Logo em seguida, uma junção (*pointer-join*) é realizada entre os empregados do tipo *Senior* e a coleção expandida pelo operador *unnest* utilizando o *OID* (Identificador de Objeto) de *Department* como predicado. Sobre o resultado da junção (empregados associados a projetos e departamentos), uma operação de *nest* é executada para agrupar os valores de orçamento de projetos por empregado, preparando o resultado para o operador *reduce* com o monóide de agregação soma (*sum*). Finalmente, uma outra operação *reduce* é utilizada para estruturar o resultado da consulta.

Agora considere a seguinte configuração para a distribuição dos dados:

- O *extent Employees* está particionado sobre *OID(Positions)* (o identificador do objeto que determina o tipo de empregado, *Senior-Manager, Junior-Manager, Senior-Analist*, entre outros) através de três nós. A cardinalidade desse *extent* é de 1.000 objetos.

- O *extent Projects* está particionado sobre o atributo *month* (o mês de início de cada projeto) através de doze nós. A cardinalidade desse *extent* é de 1.000 objetos.
- A cardinalidade do *extent Departments* é de 8 objetos.
- A chamada do método *isManager("Senior")* retorna *True* quando um empregado é gerente *Senior* e possui seletividade de 10%. Assim, o nó *GET(Employees,(e.isManager("Senior")=true))* retorna 100 objetos.
- Cada atributo tem o tamanho de 10 *bytes*.

Os dois cenários abaixo podem ser utilizados para a geração do plano físico com paralelismo intra-operador para o plano lógico da Figura 4.12:

- Selecionar a estratégia MERGE_JOIN para a execução da junção. De acordo com a Tabela 4.2, essa estratégia requer o particionamento dos planos de entrada (operandos) de acordo com o predicado de junção. Assim, os dois planos de entrada deverão ser reparticionados. Os custos de comunicação associados à transmissão dos cinco atributos do operado GET sobre *Employees* é de 5 x 10 *bytes* x 100 objetos = 5.000 *bytes*. Considerando os quatro atributos da classe *Projects* e três atributos da classe *Departments*, temos um custo de (4 + 3) x 10 *bytes* x 10.000 objetos = 700.000 *bytes* para transmitir os dados da coleção gerada pelo operador *unnest*. O custo total de reparticionamento gerado por essa estratégia é de 705.000 *bytes*.
- Selecionar a estratégia HASH_LOOP para a execução da junção. De acordo com a Tabela 4.2, essa estratégia requer a replicação do plano de entrada referente aos objetos que possuem objetos filhos agregados através da expressão de caminho. Essa replicação deve transmitir os dados para os nós onde estão os objetos filhos (*Departments* + *Projects*). Assim o custo de comunicação relacionado à replicação é de 5 x 10 *bytes* x 12 nós x 100 objetos = 60.000 *bytes*. Devido ao processo de replicação, essa estratégia não exige nenhuma partição específica para a coleção (*Departments* + *Projects*). Assim, esses objetos podem permanecer no nó original, com custos de comunicação iguais a zero, ou serem

reparticionados para outros nós, com custos de comunicação iguais a (4 + 3) x 10 *bytes* x 10.000 objetos = 700.000 *bytes*. Assim, dependendo da estratégia de comunicação utilizada, o custo de transferência de dados pode variar entre 60.000 *bytes* e 710.000 *bytes* (mesmo considerando o menor grau de paralelismo – 2 nós).

Uma estratégia *bottom-up* selecionaria, para implementar o *pointer-join*, o operador HASH_LOOP com replicação sobre o atributo *month*, ou, como segunda opção, o operador MERGE_JOIN. No entanto, ainda é necessário realizar a análise de custos para o operador *nest* no plano de execução. Esse operador é utilizado para agrupar os resultado do *pointer-join* por objetos da classe *Employees* que são gerentes do tipo *Senior*. De acordo com a Tabela 4.3, esse operador requer que o plano de entrada seja particionado pelo atributo de aninhamento, nesse caso *OID*(*Employees*). O operador MERGE_JOIN produz um resultado particionado de acordo com o predicado da junção (o OID de instâncias de *Departments*). Assim, um processo de reparticionamento faz-se necessário, gerando um custo adicional para transmitir doze atributos (considerando a união dos atributos em (*Departments* + *Projects*) e *Employees*). Esse custo é de 12 x 10 *bytes* x 10.000 objetos = 1.200.000 *bytes*. O mesmo ocorre no cenário no qual o operador HASH_LOOP com reparticionamento sobre o atributo *month* é selecionado. Assim, o custo total de reparticionamento até o operador NEST é de 1.260.000 *bytes*. Agora observe que o operador HASH_LOOP pode produzir um sub-plano particionado sobre OID(*Employees*) por um custo de 710.000 *bytes* com grau de paralelismo 2. Observe também que seria necessário um grau de paralelismo equivalente a 112 nós (5 x 10 *bytes* x 112 nós x 100 objetos = 560.000 *bytes*) para que a segunda melhor opção fosse selecionada. Utilizando esse sub-plano, o custo total de comunicação (o plano produzido já estaria particionada de acordo com as restrições do operador NEST – o OID de *Employees*) é 760.000 *bytes*. Em um otimizador *top-down*, a restrição de entrada do operador NEST já é conhecida ao nível de otimização do *pointer-join*. Ou seja, o processo de otimização no nível superior envia uma restrição pelo contexto indicado que ele precisa um plano com particionamento

pelo OID(*Employees*). Assim o operador HASH_LOOP com replicação e reparticionamento sobre OID(*Employees*) é selecionado por fornecer um plano com a característica requerida pelo nível superior de otimização. Esse exemplo ajuda-nos a perceber como uma estratégia de busca *top-down* pode ser utilizada para produzir melhores planos de execução com paralelismo e particionamento de dados.

4.3.5 Algoritmo para reordenação de operadores com paralelismo intra-operador

A ordem de execução das operações de junção tem um grande impacto no desempenho de sistemas de base de dados relacionais e assim o problema de reordenação dessas operações tem sido alvo de diversas pesquisas. Em geral, durante o processo de reordenação de junções, o otimizador deve achar uma boa ordem para executar as operações, selecionar os operadores físicos para execução das junções e selecionar os caminhos de acesso a dados (índices). Em [Has96] mostrou-se que a separação entre a seleção de operadores e reordenação de junções pode ser executada em diferentes fases do otimizador com ganho de flexibilidade e transparência desses processos. Em SGBDOO's esse processo ganhou ainda mais importância devido às expressões de caminho. Essas expressões quando mapeadas em *pointer-joins* aumentam dramaticamente o número de junções no plano de execução.

Vários otimizadores utilizam programação dinâmica baseada nos algoritmos do *System R*. Essa abordagem é melhor que estratégias enumerativas, mas mesmo assim conduz a uma ordem exponencial de $O(2^n)$. Em [Has96] uma abordagem semelhante foi utilizada para ordenação de junções em sistemas relacionais paralelos obtendo uma complexidade de $O(3^n)$. Em [Feg98] um algoritmo polinomial para ordenação de operadores de junção, *nest* e *unnest* é apresentado. Esse algoritmo é denominado GOO-OO (*Greedy Operator Ordering – Object Oriented*) e é baseado em uma heurística que seleciona primeiro as junções de menor custo para execução. Em base de dados com paralelismo particionado, esse algoritmo pode não levar a planos ótimos dado que os custos de reparticionamento de dados podem exceder muito o custo de

processamento local do operador. Este trabalho propõe uma variação dessa heurística, de forma que os custos de reparticionamento possam ser computados para as possíveis partições e usados, juntamente com o custo de processamento local, para calcular o custo total da operação. A essa variação do algoritmo chamamos de GOO-OOP (GOO-OO with Parallelism) (Figura 4.13).

O GOO-OOP utiliza um grafo de consulta para representar a ordem de execução dos operadores. Esse grafo é construído de acordo com os seguintes passos:

1. Para cada extensão (*extent*) associada com operadores de junção, *nest* e *unnest* r_i, crie um nó *i*
2. Para cada predicado de junção P_{ij} entre *i* e *j*, construa uma arco entre os nós *i* e *j* .
3. Cada nó recebe uma anotação sobre o tamanho e particionamento.
4. Cada arco entre *i* e *j* é anotado com S_{ij} (a seletividade do junção entre os operandos *i* e *j*)
5. Para cada nó *i* que depende de um nó *j* (com relação a uma expressão de caminho) construa uma seta partindo de *j* em direção a *i*.

A cada passo, o algoritmo une dois nós *i* e *j* que tenham o valor mínimo para $size(r_i) \times size(r_j) \times S_{ij} + RepartitioningCost_{ij}$, de forma que o novo nó gerado por essa união represente uma junção a ser executada. A função *repartitioning_cost* retorna um custo proibitivo quando a partição não é aplicável à extensão (*extent*). Uma das diferenças com relação ao algoritmo original é a forma de calcular o custo considerado para a seleção dos nós a serem unidos. O algoritmo varre as possíveis partições e calcula o menor custo de reparticionamento considerando os dois nós participantes da operação. Assim o custo total da operação é a soma do custo de processamento local do operador mais o custo para reparticionar os dados da entrada do operador.

Algoritmo (GOO-OOP)

Entrada:
N: Número de relações
Predicate[i,j]: O predicado P_{ij} entre r_i e r_j (default é true)
S[i,j]: A seletividade S_{ij} de predicate[i,j](default é 1)

```
Tree[i] : A árvore de operadores para r_i (inicialmente r_i)
Size[i] : tamanho de r_i
Depends_on[i] : igual a j, se r_i depende de r_j, senão é 0
Dependency[i] : o tipo de dependência (nest ou unnest) se
depends_on[i]<>0
Repartition_cost[i,j] : O custo de reparticionamento para
execução a junção, nest, ou unnest entre os nós i e j

Saída: Uma ávore de operadores válida para bancos de dados
orientados a objetos e com baixo custo de processamento
(considerando custos de reparticionamento)

Algoritmo:
    Para n = N...2 faça
        Ache i, j ∈ [1...n]:
            dependency = (i <> j   E (depends_on[i]=0 OU
                                depends_on[i] = j)
                           E depends[j]=0)
            Se partition(i) <> partition(j) Então
            Início
               repartition_cost[i,j]=min(repartition_cost(pa
               rtition(i),p)+
               repartition_cost(partition(j),p)
              (para cada partição p em all_partitions))
            Senão
                Repartition_cost[i,j] = 0
            Fim (Se)
            Onde   ((size[i]   x   size[j]   x   S[i,j]   +
                   repartition_cost[i,j] é mínimo)
                 E dependency=true)
        Fim (Ache)
        Para k= 1...n,  k<>i Faça
            S[i,k] = S[k,i] = S[j, k]xS[j,k]
             Predicate[j,k]=predicate[k,i]=
            [[predicate[i,k] E predicate[j,k]]]
        Fim (Para)
        Para k= 1...n,  k<>j Faça
            S[j,k] = S[k,j] x S[n, k]
            Predicate[j,k]=predicate[k,j]=predicate[n,k]
        Fim (Pare)
        tree[j]=tree[n]
        depends_on[j]=depends_on[n]
        dependency[j]=dependency[n]
        size[j]=size[n]
        retorne tree[1]
Fim
```

Figura 4.13 - Algoritmo GOO-OOP

O exemplo a seguir mostra o funcionamento do GOO-OOP. O grafo de consulta na Figura 4.15 é construído para os predicados e partições na Figura 4.14. Supondo que *Project.Classification=SECRET* é um predicado altamente seletivo, a junção entre *EmpProj* e *Project* produz um resultado intermediário pequeno. Para o algoritmo GOO-OO, *EmpProj* e *Project* seriam selecionados primeiro. Contudo, isso implica em uma operação de reparticionamento em *EmpProj*, que é uma tabela com uma grande quantidade de dados. O cálculo para o custo de duas combinações de nós e atributos de particionamento mostra

que se o algoritmo considerar apenas o custo de processamento do operador, a junção entre *EmpProj* e *Project* será selecionada primeiro. Contudo, a adição do custo de reparticionamento faz com que GOO-OOP selecione a junção entre *EmpProj* e *Emp* para ser executada primeiro. As outras duas combinações são descartadas devido ao custo de particionamento proibitivo causado por partição não-aplicáveis.

Emp.EmpNumber=EmpProj.EmpNumber AND
Emp.ProjNumber=Proj.ProjNumber AND
Proj.Classification=SECRET AND Emp.City=Fortaleza
AllPartitions={EmpNumber, ProjNumber}

Figura 4.14 - Exemplo de predicados e partições

A complexidade do algoritmo original é $O(n^3)$ (onde n é o número de variáveis *range*). GOO-OOP multiplica essa complexidade por um fator de *P* (o número de partições possíveis).

- Partition: ProjNumber
  - repartition_cost(Proj, ProjNumber) = 0
  - repartition_cost(EmpProj, ProjNumber) = 8
  - Local cost = 3x6x0.5 = 9.0
  - Total cost = 9 + (8+0)=17.0
- Partition: EmpNumber
  - repartition_cost(Emp, EmpNumber) = 0
  - repartition_cost(EmpProj , EmpNumber) = 0
  - Local cost = 3x6x0.8 = 14.4
  - Total cost = 14.4 + (0+0)=14.4

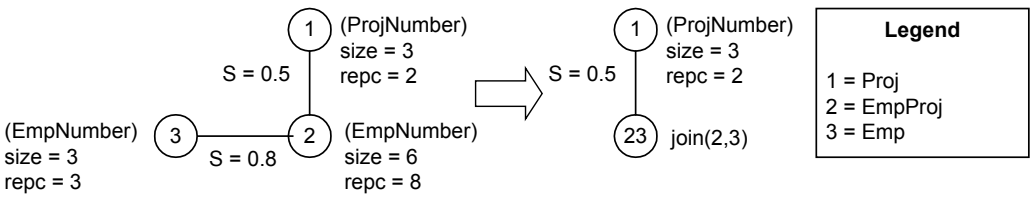


Figura 4.15 - Reordenação de operadores com reparticionamento

4.3.6 Paralelismo inter-operador

A técnica de extração de paralelismo inter-operador é apresentada em [Has96]. Essa técnica é composta das seguintes fases:

1. Identificação das unidades atômicas de execução
2. Identificação das restrições de tempo entre tais unidades

As unidades atômicas são obtidas através da análise e fatoração do código dos operadores da álgebra física. As unidades atômicas podem ser definidas como novos operadores físicos de uma álgebra paralela de nível mais baixo e que serve de interface entre o otimizador e o escalonador de consultas.

As restrições de tempo são obtidas através da análise do comportamento e do relacionamento produtor-consumidor existente entre as unidades atômicas. Os dois tipos de restrições de tempo são:

- **Restrição de paralelismo**: Identifica a necessidade de dois operadores iniciarem e terminarem a execução ao mesmo tempo. Essa restrição permite que um operador consuma os dados gerados pelo operador produtor a medida que eles são enviados através de *pipelining*. Esse processo evita a materialização intermediária dos dados.
- **Restrição de precedência**: Caracteriza-se pela necessidade de um operador iniciar sua execução apenas após o termino da execução do operador produtor.

Em [Sam98] a álgebra física monóide é fatorada de acordo com as seguintes unidades atômicas:

- REDUCE: EVALUATE_HEAD e REDUCING
- NESTED_LOOP: READ e MATCH
- SORT: FORM_RUNS e MERGE_RUNS

- NEST/GROUP_BY: GROUP, NESTING, EVALUATE_HEAD e REDUCING
- UNION: BUILD e PROBE

Os operadores TABLE_SCAN, INDEX_SCAN, MERGE_JOIN, UNNEST e MAP não são fatorados devido a sua natureza atômica. O presente trabalho propõe a fatoração do operador BLOCK_NESTED_LOOP recentemente adicionado ao Lambda-DB através dos operadores: READ_CHUNK – lê blocos da corrente de entrada de dados com menor cardinalidade para memória; e MATCH – concatena os objetos carregados em memória aos objeto da corrente de maior cardinalidade de acordo com o predicado definido.

Em [Sam98] a fase de extração do paralelismo é implementada como regras físicas que transformam os operadores lógicos diretamente em operadores da álgebra física atômica. No entanto, essa abordagem leva à mistura das transformações referentes ao paralelismo intra-operador e paralelismo inter-operador, comprometendo a extensibilidade do otimizador.

Dado que as transformações referentes ao paralelismo inter-operador são inerentes apenas aos operadores físicos, este trabalho propõe a utilização de regras exclusivas para a extração do paralelismo. É importante notar que essas transformações não provocam alteração nas propriedades físicas do planos e nem adicionam estratégias de execução alternativas ao espaço de busca. Após os operadores físicos serem selecionados na fase que engloba o paralelismo particionado, as regras de extração de paralelismo fatoram os operadores físicos do plano gerado, proporcionando uma interface atômica para o escalonador de consultas.

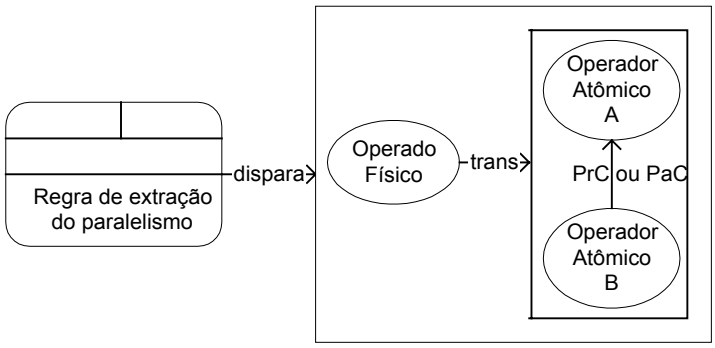


Figura 4.16 - Padrão de regra de extração de paralelismo

O padrão de regra para extração de paralelismo (Figura 4.16) fatora o operador físico em dois ou mais operadores atômicos sem alterar as propriedades física do plano. O relacionamento entre os operadores atômicos gerados pode ser indicado por uma restrição de precedência (*PrC*) ou restrição de paralelismo (*PaC*).

4.4 Conclusão

Este capítulo apresentou os sistemas de construção de otimizadores e forneceu um quadro comparativo entre eles. Depois, apresentou a metodologia proposta no trabalho, composta pela fase de análise e da fase de especificação. Este capítulo detalhou a fase de análise e deixou para o próximo capítulo o detalhamento da especificação e questões mais próximas da implementação. A metodologia proposta permite a construção modular e extensível de um otimizador para SGBDOO paralelo com paralelismo intra-operador e inter-operador, como será comprovado no capítulo a seguir com a implementação da técnica no framework OPTGEN.

Capítulo 5

Especificação e Construção de um Otimizador para um SGBDOO Paralelo

Esse capítulo aborda a fase de especificação de um otimizador conforme o *framework* descrito no capítulo 4. As etapas descritas neste capítulo utilizam os produtos gerados na fase de análise.

Inicialmente são apresentadas as modificações necessárias à especificação do otimizador e ao SGBDOO não-paralelo Lambda-DB para implementação da metodologia. Em seguida, as regras para exploração do paralelismo intra-operador e do paralelismo inter-operador são descritas. Mais adiante, as consultas utilizadas nos experimentos são apresentadas juntamente com os seus respectivos PEC's, exemplificando a nova estratégia de busca para planos paralelos. Finalmente, as medições realizadas e comparações com o otimizador seqüencial original são analisadas.

5.1 Introdução

O otimizador do SGBDOO Lambda-DB [FSRM00] foi originalmente desenvolvido para sistemas seqüenciais. Este trabalho parte do otimizador original e o estende a fim de incorporar as técnicas apresentadas no capítulo 4.

Em [Sam98], o problema de extração de paralelismo foi abordado para uma álgebra orientada a objetos e implementado através de regras de reescrita de termos em um gerador de otimizadores. Essas regras substituíram as regras físicas existentes e projetadas para um otimizador para bancos de dados não-paralelos. Dois problemas surgem a partir dessa abordagem. Primeiro, as regras

de seleção dos planos físicos não levam em consideração a execução em um sistema paralelo com o conseqüente particionamento dos dados, portanto não existe garantia de que o plano final será ótimo ou até mesmo próximo disso. Depois, as regras físicas originais são simplesmente substituídas por regras que geram os planos físicos em uma álgebra física paralela, misturando as fases propostas em [Has96], e assim produzindo uma especificação que dificulta a extensibilidade do otimizador. Atualmente, o gerador de otimizadores OPTGEN permite a declaração de vários módulos de regras lógicas e um módulo de regras físicas. O otimizador original do Lambda-DB é composto por um módulo de regras lógicas que realiza a transformação do cálculo monóide para a álgebra lógica com desaninhamento de consultas, e por um módulo de regras físicas que realiza a seleção de estratégias para a implementação dos operadores lógicos.

Para facilitar a geração de otimizadores para sistemas paralelos, este trabalho propõe a modificação do OPTGEN para a inclusão de um novo módulo de regras físicas para extração do paralelismo, e mudança na estratégia de busca para geração de planos físicos que levem em consideração o paralelismo particionado. Dessa forma a escolha dos métodos de execução dos operadores e a extração do paralelismo ocorrem de modo transparente, melhorando a especificação e a extensibilidade do otimizador gerado.

A Tabela 5.1 mostra as duas fases do modelo de otimização apresentado em [Has96] e como ele foi implementado no trabalho referenciado em [Sam98] e no presente trabalho.

O otimizador original do Lambda-DB é composto por 6 fases:

1. *Parsing* da consulta OQL
2. Tradução da consulta OQL para a forma *monoid comprehensions*
3. Validação de tipos, reescrita algébrica, incluindo normalização de *comprehensions* e materialização de expressões de caminho em *pointer joins*.
4. Desaninhamento de consultas e tradução de expressões em cálculo *monoid* para álgebra *monoid*.
5. Reordenação de junções utilizando uma heurística baseada em custos

6. Geração de plano físico utilizando um sistema de reescrita de termos baseado em regras e custos.

Modelo para Otimização Paralela de Hasan [Has96]	**Utilização do OPTGEN para extração do paralelismo em Sampaio [Sam98]**	**Modelagem através de regras e modificação do *Framework* para suportar otimização paralela**
Reordenação de Junções e geração de planos físicos com otimização paralela	Utiliza regras para otimização em sistemas seqüenciais	Modelagem de algoritmos de seleção de operadores físicos com paralelismo particionado através de regras e modificação da estratégia de busca
Extração do paralelismo	Mistura geração de planos físicos com extração do paralelismo	Adição de um novo módulo de regras físicas para extração de paralelismo e alteração da estratégia de busca original

Tabela 5.1 - Comparação entre propostas de implementação do modelo de otimização em duas fases.

Com a aplicação da técnica de construção de otimizadores apresentada, a etapa de ordenação de operadores com paralelismo [Has96] é incorporada ao otimizador. As fases 5 e 6 foram modificadas com o uso da técnica proposta (Figura 5.1). A fase 5 passa assim a utilizar regras para seleção de operadores com base nos custos locais e custos referentes ao paralelismo. A fase 6 utiliza o algoritmo GOO-OOP para a reordenação de operadores. A fase 7 foi adicionada para incorporar, de forma transparente, a técnica de extração de paralelismo proposta em [Sam98].

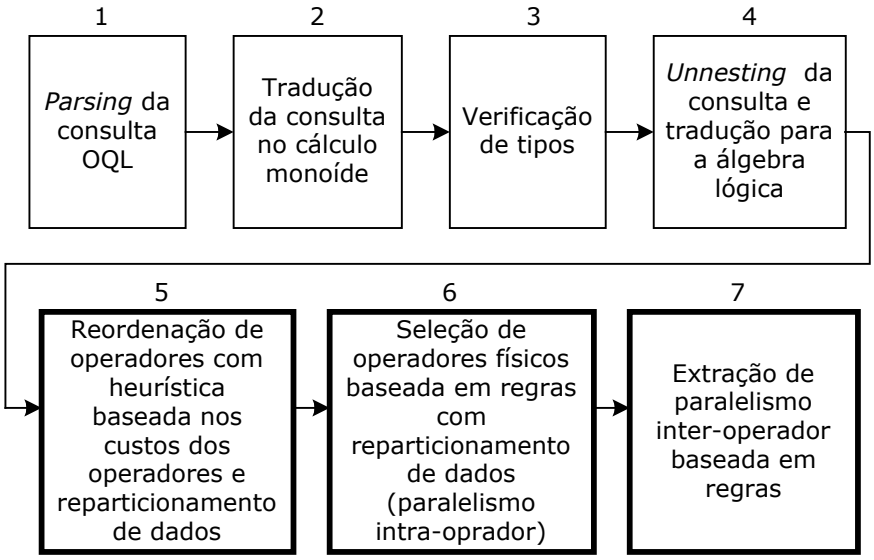


Figura 5.1 - Fases de otimização com paralelismo

5.2 Especificação de regras com paralelismo intra-operador

Nessa seção apresentamos as regras físicas OPTL para a seleção de estratégias de acordo com uma heurística baseada no custo dos operadores e custos de reparticionamento de dados. Essas regras foram construídas, com a aplicação da técnica proposta, a partir da especificação do otimizador original do Lambda-DB. Foi necessária a criação de algumas funções de suporte relativas ao paralelismo:

- *differs*: Determina se uma partição é diferente de outra.
- *make_nest_partition*: Cria uma função de particionamento a partir das variáveis de aninhamento.
- *get_parallelism_degree*: Retorna o número de nós de processamento nos quais os dados referentes a uma determinada classe e conjunto de atributos estão particionados.
- *requires_partition*: Retorna a partição de dados referente a uma determinada restrição de entrada.
- *get_extent_partition*: Retorna a partição definida para uma extensão de classe
- *all_partitions*: Retorna uma lista com as possíveis partições de uma coleção de dados

As regras descritas a seguir não representam a totalidade da especificação do otimizador, mas são suficientes para o entendimento da técnica. Acima de cada regra, o padrão de regras definido no capítulo 4 e utilizado na implementação é indicado. Os detalhes relativos ao paralelismo estão destacados em itálico no texto das regras.

Regra 1

Padrão: Regra Condicionada

```
1.  { order=order(); groups=groups();  }
2.  nest( `M,
3.        `e<-{ groups=`(function(#<groups>,pattern variables(gv)));
4.                  datapartition=`(make_partition_nest(gv));},
5.        `v, `hd, `gv, `bpred, `apred )
6.     : !differs(^e.datapartition, make_partition_nest(gv) )
7.     = NEST(`M,`e,`v,`hd,`gv,`bpred,`apred)
8.     : {     size = ^e.size;
9.             cost =  ^e.cost +
10.               (^e.size*^e.size)/
11.               get_parallelism_degree(make_partition_nest(gv));
12.            order = ^e.order;
13.            groups = ^e.groups;
14.            datapartition = make_partition_nest(gv); };
```

Regra 2

Padrão: Regra Condicionada

```
1.  { order=order(); groups=groups(`g,...gl);}
2.  nest( `M,
3.        `e<-{ groups=`(function(#<groups>,pattern_variables(gv)));
4.              datapartition=`(make_partition_nest(gv));},
5.        `v, `hd, `gv, `bpred, `apred )
6.     : ^e.groups->eq(#<groups(`g,...gl)>) &&
7.       !differs(^e.datapartition, make_partition_nest(gv) )
8.     = NEST(`M,`e,`v,`hd,`gv,`bpred,`apred)
9.          : {    size = ^e.size;
10.                cost = ^e.cost +
11.                  (^e.size*^e.size)/
12.                  get_parallelism_degree(make_partition_nest(gv));
13.                order = ^e.order;
14.                groups = ^e.groups;
15.                datapartition = make_partition_nest(gv); };
```

As regras 1 e 2 selecionam o operador físico *NEST* para implementar o operador lógico *nest*. A duas regras são diferenciadas apenas pelo agrupamento

livre na linha 1 da regra 2 (*groups()*)ou agrupamento especificado pelo plano consumidor na linha 1 da regra 2 *groups(`g,...gl)*). Assim, a primeira regra só verificada para contextos que não requerem um agrupamento específico. A segunda regra é utilizada quando o nível de otimização superior requer um agrupamento específico solicitado na consulta. A regra 2 só é disparada se a condição de aplicação especificada nas linhas 6 e 7 retornar *true*, ou seja, se o agrupamento do plano de entrada é igual ao agrupamento esperado para o plano.

De acordo com a Tabela 4.2, o operador físico *NEST* possui um condição adicional que o força a gerar apenas um plano para a partição de acordo com os atributos para aninhamento (grupo). Por exemplo, se um plano tem que retornar uma *struct* departamento e a lista de seus empregados (<*dept*, *employees*>) então o plano de entrada deve estar particionado por departamento, de forma que os objetos de empregados relacionados sejam aninhados na coleção *employees*. As regras utilizam a função de suporte *make_partition_nest(gv)* para criar uma propriedade física de particionamento sobre o atributo de aninhamento e enviar como restrição para o plano de entrada denotado pela variável *e*. Esse plano de entrada será por sua vez otimizado para gerar a melhor estratégia que possa produzir a propriedade requisitada. A propriedade física do próprio plano gerado pela regra é determinada pela função *make_partition_nest(gv)* de acordo com as restrições de entrada e saída.

Regra 3

Padrão: Regra Condicionada

```
1.  { order=`ord;  groups=groups }
2.  join( `M,
3.        `x<-{ order=`ord;
4.                 datapartition=`(required_partition(
5.                                   pred,range_vars(x),range_vars(y))); },
6.        `y<-{ order=order();
7.                  datapartition=`(required_partition(
8.                                   pred,range_vars(y),range_vars(x))); },
9.        `pred, `keep )
10.     : !differs(^x.datapartition,
11.                required_partition(pred,range_vars(x),range_vars(y))) &&
12.       !differs(^y.datapartition,
13.                required_partition(pred,range_vars(y),range_vars(x)))
14.    = NESTED_LOOP(`M,`x,`y,`pred,`keep)
```

```
15.      : { size = int(^x.size*^y.size*selectivity(pred));
16.            cost = ^x.cost+^y.cost+
17.              (^x.size*^y.size)/get_parallelism_degree(
18.                                 required_partition(
19.                                   pred,range_vars(x),range_vars(y)));
20.            order  = ^x.order;
21.            groups = ^x.groups;
22.            datapartition = required_partition(
23.                          pred,range_vars(x),range_vars(y)); };
```

Regra 4

Padrão: Regra Condicionada

```
1.  { order=order(); groups=groups(); }
2.  join( `M,
3.        `x<-{ order=order(); groups=groups();
4.              datapartition=`(required_partition(
5.                                  pred,range_vars(x),range_vars(y))); },
6.        `y<-{ order=order(); groups=groups();
7.              datapartition=`(required_partition(
8.                                  pred,range_vars(y),range_vars(x))); },
9.        `pred, `keep )
10.       : !differs(^x.datapartition,
11.                  required_partition(pred,range_vars(x),range_vars(y)))&&
12.         !differs(^y.datapartition,
13.                  required_partition(pred,range_vars(y),range_vars(x)))
14.    = BLOCK_NESTED_LOOP(`M,`x,`y,`pred,`keep)
15.      : {     size = int(^x.size*^y.size*selectivity(pred));
16.            cost = ^x.cost+^y.cost +
17.              ((^x.size*^y.size)/1000)/
18.                         get_parallelism_degree(required_partition(
19.                                pred,range_vars(x),range_vars(y)));
20.            order = #<order()>;
21.            groups = #<groups()>;
22.            datapartition = required_partition(
23.                           pred,range_vars(x),range_vars(y)); };
```

Regra 5

Padrão: Regra Condicionada

```
1.  { order=`ord; groups=groups(); }
2.  join( `M,
3.        `x<-{ order=`ord;
4.              datapartition=`(required_partition(
5.                                pred,range_vars(x),range_vars(y)));},
6.        `y<-{ order=order();
7.              datapartition=`(required_partition(
8.                                pred,range_vars(y),range_vars(x))); },
9.        `pred, `keep )
10.      : !differs(^x.datapartition,
11.                 required_partition(pred,range_vars(x),range_vars(y))) &&
12.        !differs(^y.datapartition,
13.                 required_partition(pred,range_vars(y),range_vars(x)))
14.    = #case y
15.      | INDEX_SCAN(`m,`Y,`v,`py,`index,`low,`high)
16.           : !applicable_index(index,Y,v,pred)->eq(#<none>)
17.           =>
18.         INDEXED_LOOP(`M,`x,`y,`pred,`keep,`index,
19.                      `(applicable_index(index,Y,v,pred)))
20.              : { size = int(^x.size*^y.size*selectivity(pred));
21.                  cost = ^x.cost+(4.0*^x.size)/
22.                                 get_parallelism_degree(required_partition(
23.                                        pred,range_vars(x),range_vars(y)));
24.                  order = ^x.order;
25.                  groups = ^x.groups;
26.                  datapartition = required_partition(
27.                                   pred,range_vars(x),range_vars(y)); };
28.      #end;
```

As regras 3, 4 e 5 selecionam os operadores físicos NESTED_LOOP, BLOCK_NESTED_LOOP ou INDEXED_LOOP para a implementação de junções. Esses operadores podem funcionar como junções por valor ou junções de ponteiros, de acordo com a utilização de *OID*'s pelo predicado. A seleção é feita de acordo com condições (*guards*) que avaliam propriedades como a ordenação e o tipo de predicado da junção, ou a existência de índices (caminhos de acesso para os dados). Todos esses operadores físicos possuem condições adicionais na Tabela 4.2 que requerem que os algoritmos sejam paralelizados sobre o atributo de junção. Ou seja, que os dois planos de entrada estejam particionados sobre o atributo de junção. Se os planos não satisfizerem essa

condição, eles deverão ser reparticionados. Assim, essas regras utilizam a função *required_partition* (regra 3 linhas 4 e 7; regra 4 linhas 4 e 7; regra 5 linhas 4 e 7) para enviar aos dois sub-planos de entrada (*x* e *y*) o particionamento de dados necessário para estratégia.

Os sub-planos são otimizados antes da verificação da condição e da execução do corpo da regra de forma *top-down*. Por exemplo, se os sub-planos representarem operadores de leitura de dados (*get*), o otimizador pode selecionar um TABLE_SCAN, INDEX_SCAN, EXCHANGE(TABLE_SCAN), EXCHANGE(INDEX_SCAN) para implementar cada um deles. Essa seleção depende de fatores como a existência de índices nas extensões dos objetos, ou se essas extensões já estão particionadas pelo atributo requisitado pelo predicado da junção. Todos os sub-planos gerados são consumidos pela regra, que os utiliza para a verificação de condições e para gerar planos de junção alternativos. As propriedades físicas são estimadas de forma *Bottom-up* usando as propriedades físicas, como tamanho e custo, estimada para esses sub-planos. Por exemplo, na linha 15 da regra 3 a propriedade física de tamanho (*size*) do plano é estimada com base no tamanho estimado nos sub-planos (*size = int(^x.size*^y.size*selectivity(pred));*)

Regra 6

Padrão: Regra Livre

```
1.  { order=order(); groups=groups(); }
2.  reduce(`M,`e <-{ datapartition=datapartition(); },`v,`head,`pred)
3.     = REDUCE(`M,`e,`v,`head,`pred)
4.       : {     size = (collectionp(M)) ? ^e.size : 1;
5.               cost = ^e.cost+(^e.size)/
6.                           get_parallelism_degree(^e.datapartition);
7.               order = ^e.order;
8.               groups = ^e.groups;
9.               datapartition = ^e.datapartition;
10.            };
```

A regra 6 transforma o operador lógico *reduce* no operador físico REDUCE. Essa regra implementa o padrão regra física livre. Assim, a partição vazia *datapartition(),* na regra 6 linha 2, é enviada para o sub-plano de *e* através da

linha de cabeçalho *`e <-{ datapartition=datapartition(); }*. Essa partição sinaliza ao otimizador que o operador existente no nível superior aceita qualquer particionamento para seu processamento. Dessa forma, o otimizador gera planos alternativos com as possíveis configurações de particionamento existentes para as estratégias aplicáveis ao operador lógico existente no sub-plano *e*. Essa reescrita é feita com o uso de regras de garantia livre devido à classificação do operador REDUCE na Tabela 4.3. A estratégias e partições são então utilizadas por essa regra no passo *Bottom-up* para o cálculo das propriedades físicas e geração de árvores de consulta alternativas. A linha *datapartition = ^e.datapartition,* na regra 6 linha 9, mostra que o operador REDUCE é executado sobre a partição selecionada no sub-plano (nível inferior de otimização).

O operador REDUCE constitui a raiz da árvore de execução. Ele é o operador de disponibilização dos dados para o cliente. Assim, um operador EXCHANGE deve ser inserido sobre ele para que os dados possam ser integrados a partir dos nós de processamento e disponibilizados para a aplicação cliente. Esse processo deve ser feito automaticamente pelo otimizador. As regras 11 e 12, mostradas mais adiante são responsáveis pela inserção do EXCHANGE e o encapsulamento dos custos de reparticionamento de dados, separando esses detalhes do restante das regras.

Regra 7

Padrão: Regra Livre

```
1.  unnest( `M, `e<-{ datapartition=datapartition(); },
2.          `v, `path, `pred, `keep )
3.   = UNNEST( `e, `v,`path, `pred, `keep )
4.      : {
5.         size = ^e.size;
6.         cost=^e.size+(^e.size)/get_parallelism_degree(^e.datapartition);
7.         order = ^e.order;
8.         datapartition = ^e.datapartition;
9.        };
```

A regra 7 implementa o padrão de regra livre. O operador UNNEST é utilizado para desaninhar sub-consultas e avaliar expressões de caminho. Por ser

um operador livre, o UNNEST apresenta-se, no contexto do paralelismo, como uma solução alternativa para a técnica de *pointer-joins*. As junções necessitam do particionamento dos dados de acordo com o predicado, incorrendo em custos de comunicação referentes ao reparticionamento dos dados. Já o UNNEST, que é um operador livre (Tabela 4.3), não requer o reparticionamento. Ele realiza a leitura dos dados correspondentes a corrente de entrada *e* e recupera os dados relativos a coleção aninhada através da função de expressão de caminho *path*.

Regra 8

Padrão: Regra Base

```
1.  { order=order(); groups=groups();}
2.   get(`M,`ext,`name,`pred)
3.    = TABLE_SCAN(`M,`ext,`name,`pred)
4.    : {
5.         size = int(table_cardinality(ext)*selectivity(pred));
6.         cost = table_size(ext)/ get_parallelism_degree(
7.                                      get_extent_partition(ext));
8.         order = #<order(OID(`name))>;
9.         groups = #<groups(`name)>;
10.        datapartition = get_extent_partition(ext);
11.  };
```

Regra 9

Padrão: Regra Base

```
1.  { order=order(`o,...os); groups=groups(); }
2.  get(`M,`ext,`name,`pred)
3.  = #case find_index(ext,#<order(`o,...os)>)
4.    |  index(`idx,order(...io))
5.          => INDEX_SCAN(`ext,`name,`pred,`idx,order(...io))
6.            : {
7.               size = int(table_cardinality(ext)*selectivity(pred));
8.                cost = index_access_cost(ext,pred,io) /
9.                                  get_parallelism_degree(
10.                                    get_extent_partition(ext));
11.               order = #<order(...io)>;
12.               groups = #<groups(...io)>;
13.               datapartition = get_extent_partition(ext); };
14.   #end;
```

Regra 10

Padrão: Regra Base

```
1.  { order=order(); groups=groups(); }
2.  get(`M,`ext,`name,`pred)
3.  : ext->variablep()
4.  = #forall r in all_table_indexes(ext,name) do
5.      #case r
6.       |  index(`idx,order(...io))
7.              => INDEX_SCAN(`ext,`name,`pred,`idx,order(...io))
8.                 : {
9.                  size = int(table_cardinality(ext)*selectivity(pred));
10.                 cost = index_access_cost(ext,pred,io) /
11.                                    get_parallelism_degree(
12.                                       get_extent_partition(ext));
13.                order = #<order(...io)>; };
14.                groups = #<groups(...io)>;
15.                datapartition = get_extent_partition(ext); };
16.     #end;
17.   #end;
```

As regras 8, 9 e 10 selecionam o operador de acesso aos dados armazenados nos discos (*extent*). Elas geram um TABLE_SCAN, INDEX_SCAN ou vários INDEX_SCAN (um para cada índice disponível) de acordo com condições como a existência de um índice que satisfaça uma ordem esperada para o plano, ou a avaliação de planos para cada índice (caso um ordem não seja especificada). Todas elas, no entanto, levam em consideração a condição adicional da Tabela 4.2, que requer que o operador seja paralelizado sobre a partição física existente no *extent*.

Regra 11

Padrão: Regra de Garantia Condicionada

```
1.  { datapartition=datapartition(`a); }
2.   (`x<-{ datapartition=`(function(#<nonepart>, Nil));})
3.    : is_part_plan(x) && differs(^x.datapartition,#<datapartition(`a)>)
4.    = EXCHANGE(`x,datapartition(`a))
5.      : { size   = ^x.size;
6.          cost   = ^x.cost + repartition_cost(^x.size,
7.                                    ^x.datapartition, #<datapartition(`a)>);
8.          order  = ^x.order;
9.          groups = ^x.groups;
10.         datapartition = #<datapartition(`a)>; };
```

Regra 12

Padrão: Regra de Garantia Livre

```
1.  { datapartition=datapartition(); }
2.  (`x<-{ datapartition=`(function(#<nonepart>, Nil));})
3.    : is_part_plan(x)
4.    = #forall outpart in all_partition(x) do
5.        #case expr_differs(^x.datapartition, outpart)
6.            | true =>
7.          EXCHANGE(`x,  `outpart)
8.          : {
9.              size = ^x.size;
10.           cost = ^x.cost + repartition_cost(^x.size,
11.                                ^x.datapartition, outpart);
12.           order  = ^x.order;
13.           groups = ^x.groups;
14.           datapartition = outpart;
15.         };
16.       #end;
17.     #end;
```

As regras 11 e 12 implementam, respectivamente, os padrões de garantia condicionada e livre. Essas regras encapsulam o custo do paralelismo com reparticionamento e a seleção da configuração de particionamento.

A regra 11 é verificada quando o nível de otimização possui no contexto um particionamento requisitado pelo nível superior de otimização. Essa partição é indicada no cabeçalho através da expressão *{ datapartition=datapartition(`a); }* na regra 11 linha 1. A regra envia para o sub-plano uma partição nula de forma a evitar a reaplicação da regra. A condição da regra indicada na regra 11 linha 3 (*:is_part_plan(x) && differs(^x.datapartition,#<datapartition(`a)>)*) verifica se a expressão em *x* é um operador físico paralelo e se a sua partição difere da partição exigida pelo contexto de otimização. Se a condição é satisfeita, a regra é aplicada. Observe que o custo de transmissão é estimado através da função *repartition_cost*. Para isso, a função utiliza o tamanho da corrente de dados, a partição original e a partição indicada na restrição de saída. Essa função, também, pode utilizar estatísticas acerca da configuração de particionamento para a estimativa de custos. Por fim, a partição da restrição de saída é atribuída à

propriedade física *datapartition* do plano encabeçado pelo operador EXCHANGE.

A regra 12 é verificada quando o nível de otimização possui no contexto uma partição com atributos vazios. Isso indica que o otimizador está livre para selecionar a melhor partição. Assim, a regra utiliza a função de suporte *all_partitions* para percorrer as possíveis partições para o plano de entrada. Para cada uma das partições possíveis a condição é verificada. Se a condição é satisfeita, a regra gera um plano com o EXCHANGE no topo. Esse operador reparticiona os dados a partir dos nós indicados na partição original para os nós indicados na nova partição. A regra também utiliza a função *repartition_cost* para estimar o custo de reparticionamento de cada configuração possível.

5.3 Especificação de regras com paralelismo inter-operador

Os sistemas de construção de otimizadores permitem a declaração de regras físicas para a transformação de operadores lógicos em operadores físicos. No entanto, o problema de otimização para sistemas paralelos exige um ambiente no qual as transformações referentes a cada tipo de paralelismo possam ser especificadas de forma transparente. Ou seja, espera-se ser possível definir regras de seleção de operadores físicos com paralelismo intra-operador e regras que possam refinar os operadores físicos, permitindo o escalonamento do paralelismo inter-operador. Como dito anteriormente, as regras para extração de paralelismo apresentadas em [Sam98] foram implementadas como regras físicas, misturando as etapas de geração de plano físico e extração do paralelismo. Esta seção apresenta a contribuição deste trabalho com relação à geração de otimizadores para sistemas paralelos no que diz respeito à extração de paralelismo. Assim, além de módulos lógicos e módulos físicos, o usuário poderá declarar um módulo para extração do paralelismo e definir suas regras de fatoração de operadores, tomando como cabeçalho da regra o operador físico, ou seja, a regra transforma um operador físico em um ou mais operadores físicos paralelos de forma clara e modular.

Para ilustrar a separação dos tipos de paralelismo em sistemas de regras, são apresentadas a seguir a regra original do Lambda-DB para o operador

NESTED_LOOP, a regra física com fatoração de operador definida em [Sam98] e a regra exclusiva para extração de paralelismo:

REGRA FÍSICA ORIGINAL

```
{ order=`ord; }
join( `M, `x<-{ order=`ord; },
      `y<-{ order=order(); },
      `pred, `keep )
   = NESTED_LOOP(`M,`x,`y,`pred,`keep)
     : { size = int(^x.size*^y.size*selectivity(pred));
         cost = ^x.cost+^y.cost+(^x.size*^y.size);
         order = ^x.order; };
```

REGRA FÍSICA DE FATORAÇÃO [Sam98]

```
{ order=order(); }
join('M, 'x, 'y, 'pred, 'keep)
     = MATCH(READ(`x),`y,`pred,`keep)
        : { size = int((^x.size+^y.size)*selectivity(pred));
            cost = ^x.cost+^y.cost+(^x.size+^y.size);
            order = ^x.order; };
```

REGRA FÍSICA DO MÓDULO DE EXTRAÇÃO DE PARALELISMO

```
NESTED_LOOP('M, 'x, 'y, 'pred, 'keep)
     = MATCH(READ(`x),`y,`pred,`keep);
```

A regra física original apresentada seleciona o método *NESTED_LOOP* para o operador *join* e estima o custo de execução. Essa regra foi alterada em [Sam98], de tal forma que o operador *join* é mapeado diretamente nos operadores físicos paralelos *MATCH* e *READ*. Essa regra modela ao mesmo tempo a seleção de operadores e a fatoração da álgebra. A terceira regra é exclusiva para extração do paralelismo. Essa regra fatora o operador físico *NESTED_LOOP* nos operadores *MATCH* e *READ* de forma clara e simples, e a regra física original ainda é mantida. Note que essas regras não fazem referência às propriedades físicas do plano. Isso ocorre dado que a extração de paralelismo apenas fatora a álgebra física sem alterar custo ou outra propriedade física já calculada pelas regras físicas de seleção de operadores. Essa abordagem apresenta os seguintes benefícios:

- Facilidade para extensão do otimizador, já que as regras originais podem ficar intactas e novas regras de extração de paralelismo podem ser adicionadas.
- Maior modularização dos otimizadores construídos através de um módulo de regras específico para extração do paralelismo.
- Transparência entre seleção de métodos e extração de paralelismo, já que as duas tarefas são modeladas por regras distintas que podem ser modificadas independentemente.

Abaixo são especificadas algumas regras para o módulo de extração de paralelismo para a álgebra paralela abordada neste trabalho. Os operadores TABLE_SCAN, INDEX_SCAN, MERGE_JOIN, UNNEST e MAP, devido à sua natureza atômica, não são particionados, portanto não há necessidade de especificar regras para eles. O otimizador utiliza os operadores originais para esses casos.

```
SORT(‘x, ‘o)
     = MERGE_RUNS(FORM_RUNS(`x, `o ),`o);
```

Essa regra fatora o operador físico SORT nos operadores físicos paralelos de granulidade fina FORM_RUNS que gera uma corrente (*stream*) a partir das tuplas do plano de entrada *`x* de acordo com a ordem *`o*, e no operador MERGE_RUNS que realiza a união da corrente gerada pelo FORM_RUNS de acordo com a ordem *`o*.

```
NESTED_LOOP(‘M, ‘x, ‘y, ‘pred, ‘keep)
     = MATCH(READ(`x),`y,`pred,`keep);
```

Essa regra fatora o operador físico NESTED_LOOP nos operadores MATCH e READ. O operador READ lê as tuplas a partir do plano de menor cardinalidade (*`x*) para a memória. O operador MATCH concatena as tuplas carregadas na memória por READ com as tuplas do plano da direita (*`y*) de acordo com o predicado da junção (*`pred*) e com *`keep*, que indica se o

algoritmo deve implementar um *left-outer-join*, ou um *right-outer-join* ou um *regular-join*.

```
NEST(`M,`e,`v,`hd,`gv,`vars(...r),`apred)
= let list<Expr>* other_nestvars = r->append(all_nesting_vars(e)) in
 REDUCING(`M,
      EVALUATE_HEAD(
              NESTING(
                 GROUP(`e, `gv),
                 vars(...other_nestvars)), `hd, `apred), `v, `hd);
```

Essa regra fatora o operador NEST da seguinte forma:

- GROUP agrupa as tuplas da corrente de entrada do plano *`e* de acordo com as variáveis de agrupamento (*`gv*).
- NESTING cria uma tupla (para cada grupo gerado por GROUP) estendida com os valores das variáveis definidas em *`vars(...r)* mais as variáveis de aninhamento do plano de entrada.
- EVALUATE_HEAD e REDUCING representam a aplicação do operador REDUCE (já fatorado) para estruturação da saída de cada tupla (correspondente aos diferentes grupos) gerada em NESTING.

```
REDUCE(‘M, ‘e, ‘v, ‘head, ‘pread)
     = REDUCING(`M, EVALUATE_HEAD(‘e, ‘head, `pred), `v, `head);
```

Essa regra fatora o operador REDUCE através do operador EVALUATE_HEAD e REDUCING. O primeiro avalia a expressão *`head* para cada tupla da corrente de entrada do plano *`e* de acordo com o predicado (*`pred*). O REDUCING reduz a corrente de tuplas a um valor ligado a variável *`v* ou a uma coleção de acordo com o *monoid `M*.

```
MERGE(‘M, ‘x, ‘y)
     = PROBE(`M, BUILD(`x),`y);
```

Essa regra fatora o operador MERGE em BUILD, que constrói uma tabela *hash* com o plano da esquerda (*`x*), seguido do operador PROBE, que compara as tuplas na tabela *hash* com as tuplas no outro plano (*`x*), de forma que as tuplas

que já estão na tabela sejam descartadas e as outras, unidas às que estão em memória e formatadas para saída de acordo com o *monoid `M* especificado.

5.4 Exemplos e Experiementos

Essa seção apresenta os resultados obtidos com o otimizador para sistemas paralelos construído com a aplicação da técnica sobre o sistema OPTGEN. O tempo de otimização para cada consulta no novo otimizador foi obtido através da função *clock()* do C++. No entanto, não foi possível obter o tempo de execução do plano gerado pelo fato do Lambda-DB não ser um SGBDOO paralelo e, assim, não ter capacidade para executar os planos obtidos. Assim, o tempo de execução paralela foi estimado com base no custo do plano paralelo, no custo do plano seqüencial e no seu respectivo tempo de execução, obtendo o tempo gasto por cada unidade de custo através da execução do plano seqüencial. Para que esse método de simulação seja válido, o trabalho assume que:

- O número de objetos produzidos por cada nó é o mesmo com relação ao atributo de particionamento. Ou seja, o sistema está com a carga bem balanceada.
- O tempo para produção de um objeto é constante.
- Não existem conflitos de acesso na rede.
- O tempo de comunicação de um objeto é constante e proporcional ao seu tamanho.

As consultas utilizadas nos *benchmarks* são as mesmas consultas que foram utilizadas nos *benchmarks* apresentados em [FM00].

Cada consulta é acompanhada pelo plano paralelo gerado e no final da seção apresenta-se um quadro comparativo entre o tempo de CPU obtido com o otimizador original e o tempo de CPU estimado para o otimizador modificado.

As extensões das classes estão particionadas como descrito abaixo. O esquema de objetos utilizado está definido na Figura 2.3. O grau de paralelismo utilizado é de três nós de processamento.

Instructors : datapartition(dept);

Departments: datapartition(head);

Courses : datapartition(taught_by);

5.4.2 Consultas e planos com paralelismo intra-operador

Nessa seção, apresentam-se as consultas usadas nos testes e os respectivos planos "ótimos" selecionados pelo otimizador. Em *Detalhe*, apresenta-se uma breve explicação sobre a semântica da consulta e uma explicação detalhada sobre o plano de execução. O plano é apresentado de forma textual, de acordo com a saída produzida pelo otimizador. Uma figura com uma árvore anotada de operadores é apresentada para cada plano. Nessa figuração a notação *epart* representa a partição esperada por um operador, e a notação *p* indica a partição utilizada na corrente de dados. O operador EXCHANGE é destacado no plano com o uso de negrito.

Consulta 1:

```
select x: e.name, y: (select c.name from c in e.teaches)
from e in Instructors
```

Detalhe 1:

Essa consulta seleciona o nome de todos os instrutores e o nome dos cursos ensinados por cada instrutor.

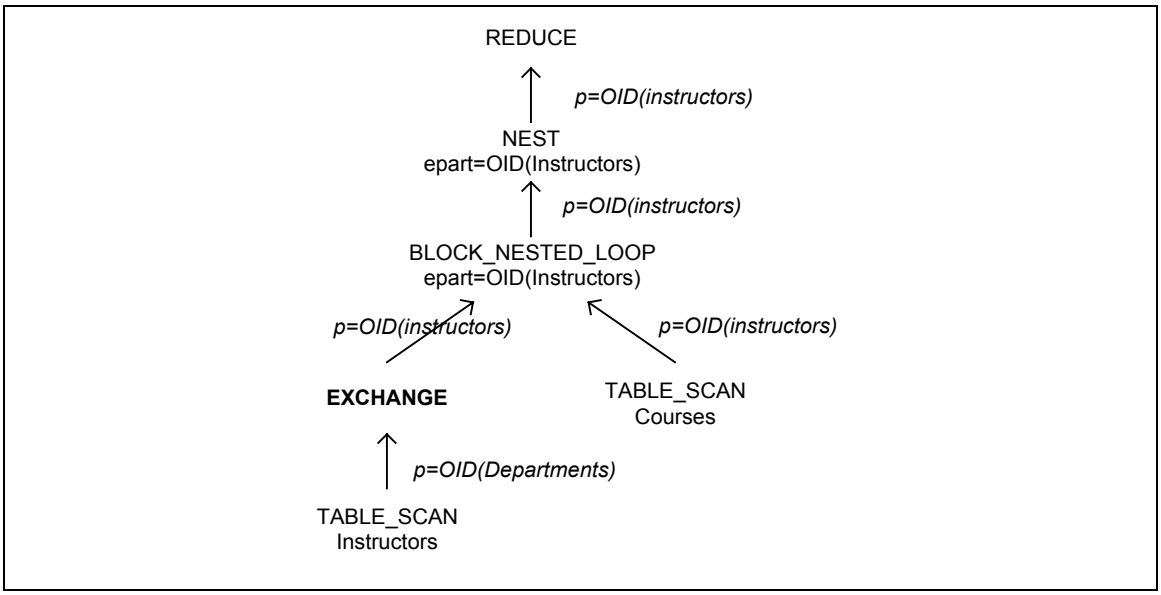
O plano obtido utiliza um *pointer-join* para avaliar a expressão de caminho *e.teaches*. Os dados obtidos por TABLE_SCAN a partir da extensão *Instructors* são reparticionados da sua partição original, *datapartition(OID(Departments))*, para a partição esperada pelo BLOCK_NESTED_LOOP (*Datapartition(OID(_X35))*. Note que os dados produzidos na operação de leitura sobre *Instructors* são *ligados* à variável *_X35*. O reparticionamento é realizado através do operador de garantia EXCHANGE. Os cursos obtidos pela junção com ponteiros são aninhados, através do operador NEST, nos objetos *instrutores* representados pela variável *_X35*. Finalmente o operador REDUCE estrutura a saída através de um coleção do tipo monóide *bag*. Essa coleção é composta por uma estrutura com o nome do instrutor

estendido com uma *bag* com os nomes dos cursos associados (veja parâmetro *struct(bind(x,project(_X35,name, string)),bind(y,_X45))* do operador REDUCE).

Plano 1:

```
REDUCE(bag,
   NEST(bag,
        BLOCK_NESTED_LOOP(bag,
                EXCHANGE(
                   TABLE_SCAN(bag,
                              Instructors,
                              _X35,
                              and()),
                   datapartition(OID(_X35))),
                TABLE_SCAN(bag,
                           Courses,
                           _X36,
                           and()),
                and(eq(project(_X36,taught_by,Instructor), OID(_X35))),
                _X35),
         _X45,
         project(_X36,name,string),
         _X35,
         and(),
         and()),
   _X34,
   struct(bind(x,project(_X35,name,string)),bind(y,_X45)),
   and())
```

Árvore de Operadores 1:



Consulta 2:

```
select x: e.name, y: count(e.dept.instructors)
from e in Instructors
```

Detalhe 2:

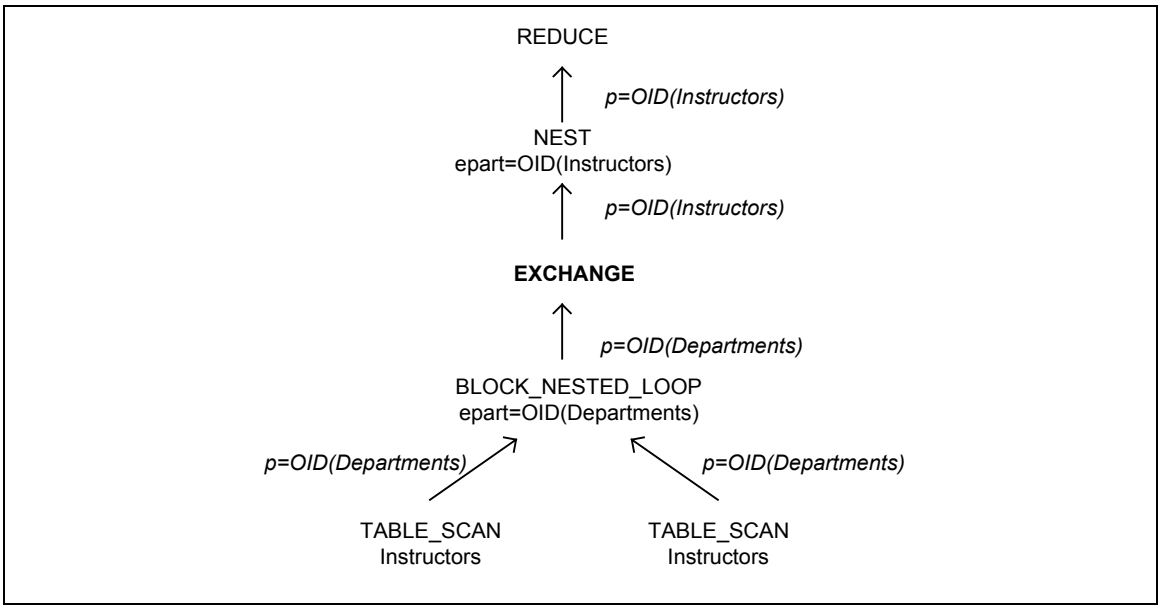
Essa consulta seleciona o nome de todos os instrutores e a quantidade de instrutores que trabalham no mesmo departamento de cada um deles.

No plano gerado, a expressão de caminho *e.dept.instructors* é avaliada através de uma junção com ponteiros através do identificador do departamento agregado em *Intructor*. O resultado do BLOCK_NESTED_LOOP está particionado pelo identificador do departamento. O operador NEST utiliza o monóide *sum* para contar o resultado da avaliação da expressão de caminho agrupando pelo objeto *Intructor*. Esse operador espera um particionamento sobre o objeto de agrupamento. Assim o operador EXCHANGE é inserido para reparticionar os dados de acordo com *OID(Instructors*).

Plano 2:

```
REDUCE(bag,
       NEST(sum,
          EXCHANGE(
            BLOCK_NESTED_LOOP(bag,
                     TABLE_SCAN(bag,
                                Instructors,
                                _X36,
                                and()),
                     TABLE_SCAN(bag,
                                Instructors,
                                _X37,
                                and()),
                     and(eq(project(_X37,dept,Department),
                                OID(project(_X36,dept,Department)))),
                     _X36),
            datapartition(OID(project(_X36,dept,Department))),
          _X46,
          1,
          _X36,
          and(),
          and()),
       _X35,
       struct(bind(x,project(_X36,name,string)),bind(y,_X46)),
       and())
```

Árvore de Operadores 2:


REDUCE
p=OID(Instructors)
NEST
epart=OID(Instructors)
p=OID(Instructors)
EXCHANGE
p=OID(Departments)
BLOCK_NESTED_LOOP
epart=OID(Departments)
p=OID(Departments)
p=OID(Departments)
TABLE_SCAN
Instructors
TABLE_SCAN
Instructors

Consulta 3:

```
select x: e.name,
       y: (select x: c.name, y: count(c.has_prerequisites)
          from c in e.teaches)
from e in Instructors
```

Detalhe 3:

Essa consulta seleciona o nome de todos os instrutores juntamente com o nome dos cursos ensinados por eles e, para cada um dos cursos, a quantidade de cursos pré-requisitos.

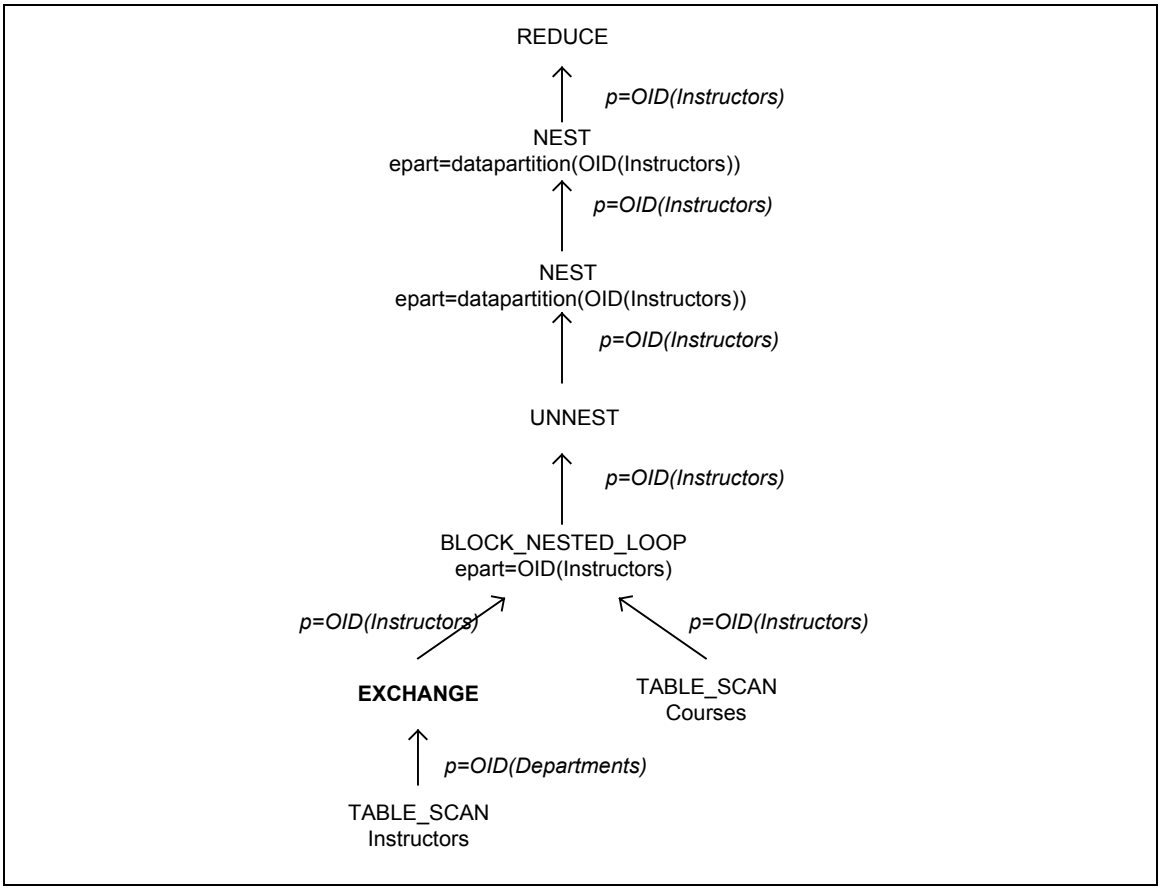
No plano gerado, a expressão de caminho *c.has_prerequisites* é avaliada através de uma junção com ponteiros utilizando o atributo *taught_by* em cada objeto *Course*. Os dados produzidos na leitura de *Instructors*, particionados originalmente por *OID(Departments*), são reparticionados de acordo com *OID(Instructors*). Depois a coleção de pré-requisitos é desaninhada (UNNEST) de cada objeto *Course* e depois aninhada novamente (operador NEST interno) com um quantificador agrupado por instrutor. O NEST mais externo aninha os cursos e contagem de pré-requisitos dentro dos instrutores e finalmente o operador REDUCE estrutura a saída com um monóide *bag*.

Plano 3:

```
REDUCE(bag,
       NEST(bag,
            NEST(sum,
               UNNEST(bag,
                        BLOCK_NESTED_LOOP(bag,
                          EXCHANGE(
                            TABLE_SCAN(bag,
                                       Instructors,
                                       _X82,
                                       and()),
                          datapartition(OID(_X82)),
                          TABLE_SCAN(bag,
                                     Courses,
                                     _X83,
                                     and()),
                          and(eq(project(_X83,taught_by,Instructor),
                          OID(_X82))),
                          _X82),
                        _X84,
                        project(_X83,has_prerequisites,set(Course)),
                        and(),
                        pair(_X82,_X83)),
               _X103,
               1,
               pair(_X82,_X83),
               and(),
```

```
        and()),
      _X93,
      struct(bind(x,project(_X83,name,string)),bind(y,_X103)),
      _X82,
      and(),
      and()),
_X81,
struct(bind(x,project(_X82,name,string)),bind(y,_X93)),
and())
```

Árvore de Operadores 3:



Consulta 4:

```
select x: dn, y: count(partition)
from e in Instructors
group by dn: e.rank
```

Detalhe 4:

Essa consulta seleciona a descrição de todas as posições cabíveis a instrutores e quantas ocorrências de cada posição existem na instituição.

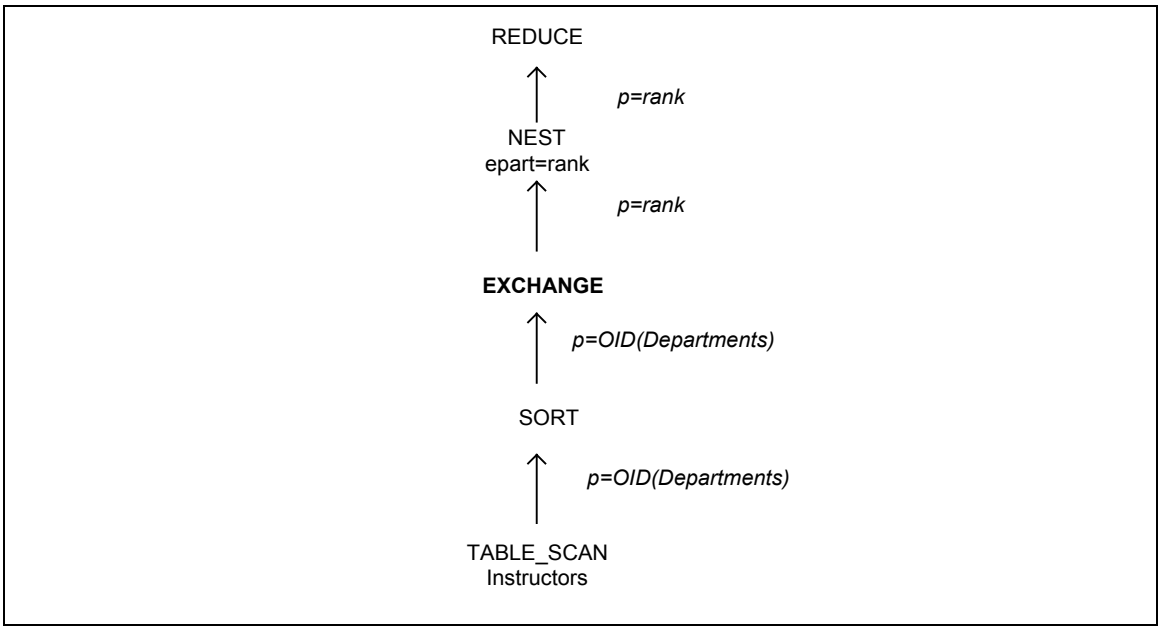
No plano gerado, os dados lidos da extensão *Instructors* são reparticionados de acordo com o valor do atributo *rank* (posição). Esse reparticionamento é exigido

pelo operador NEST. Esse operador conta (monóide *sum*) quantas ocorrências existem por posição.

Plano 4:

```
REDUCE(bag,
       NEST(sum,
          EXCHANGE
            SORT(
               TABLE_SCAN(bag,
                          Instructors,
                          _X46,
                          and()),
               order(project(_X46,rank,string))),
          datapartition(project(_X46,rank,string))
          _X59,
          1,
          project(_X46,rank,string),
          and(),
          and()),
       _X45,
       struct(bind(x,project(_X46,rank,string)),bind(y,_X59)),
       and())
```

Árvore de Operadores 4:



Consulta 5:

```
select d.name,
       c: count(select * from e in d.instructors
              where e.rank = "professor")
from d in Departments
order by count(select * from e in d.instructors
             where e.rank = "professor")
```

Detalhe 5:

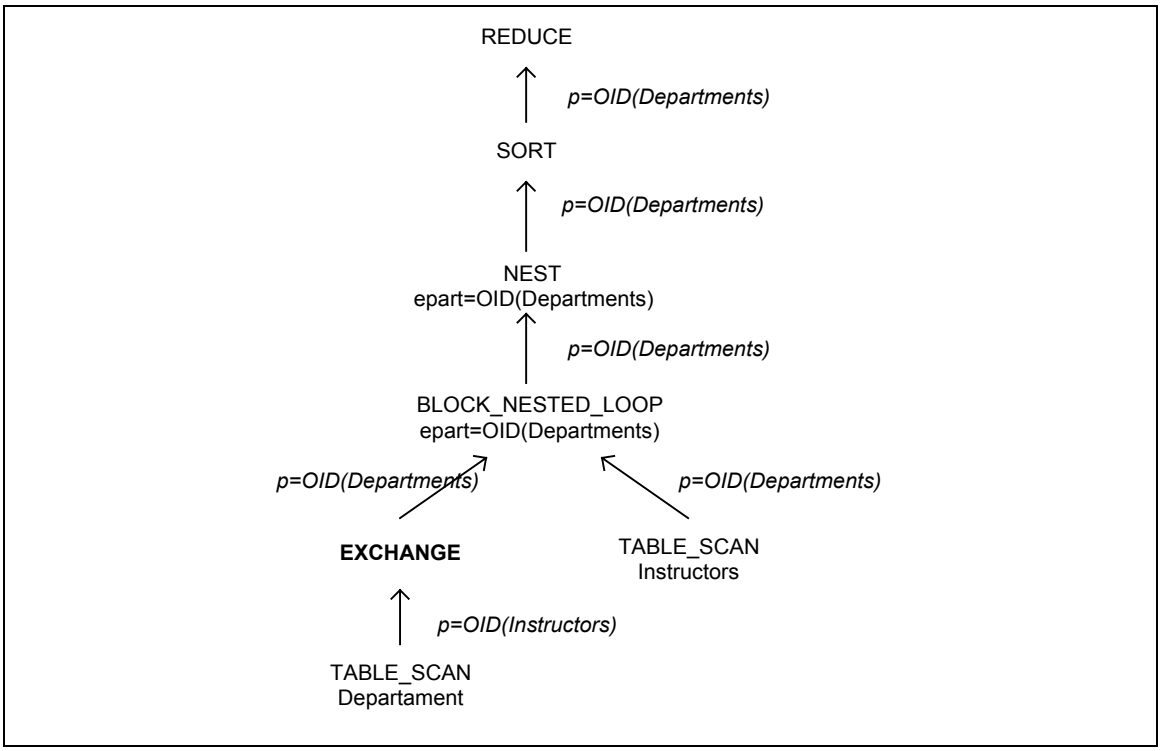
Essa consulta seleciona o nome de todos os departamentos juntamente com a quantidade de professores em cada departamento. O resultado é ordenado por essa quantidade.

No plano gerado, a expressão de caminho *d.instructors* é avaliada através do operador BLOCK_NESTED_LOOP. Devido ao predicado desse operador (*and(eq(project(_X85,dept,Department),OID(_X83)))*), o resultado da leitura sobre a extensão *Departments* é reparticionado pelo EXCHANGE. A leitura sobre a extensão *Instructors* utiliza o predicado *and(eq(project(_X85,rank,string), "professor")))* para selecionar os instrutores com posição de professor. O operador NEST conta quantos professores existem por departamento e o operador SORT ordena o resultado por essa quantidade. Finalmente o resultado é estruturado pelo REDUCE com o monóide *list*.

Plano 5:

```
REDUCE(list,
       SORT(NEST(sum,
                 BLOCK_NESTED_LOOP(bag,
                       EXCHANGE(
                          TABLE_SCAN(list,
                                     Departments,
                                     _X83,
                                     and()),
                          datapartition(OID(_X83))),
                       TABLE_SCAN(bag,
                                  Instructors,
                                  _X85,
                                  and(eq(project(_X85,rank,string),
                                         "professor"))),
                       and(eq(project(_X85,dept,Department),OID(_X83))),
                       _X83),
                 _X94,
                 1,
                 _X83,
                 and(),
                 and()),
            order(_X94)),
       _X82,
       struct(bind(name,project(_X83,name,string)),bind(c,_X94)),
       and())
```

Árvore de Operadores 5:



Consulta 6:

```
select e.name, c: count(e.teaches)
from e in Instructors
where count(e.teaches) >= 4
```

Detalhe 6:

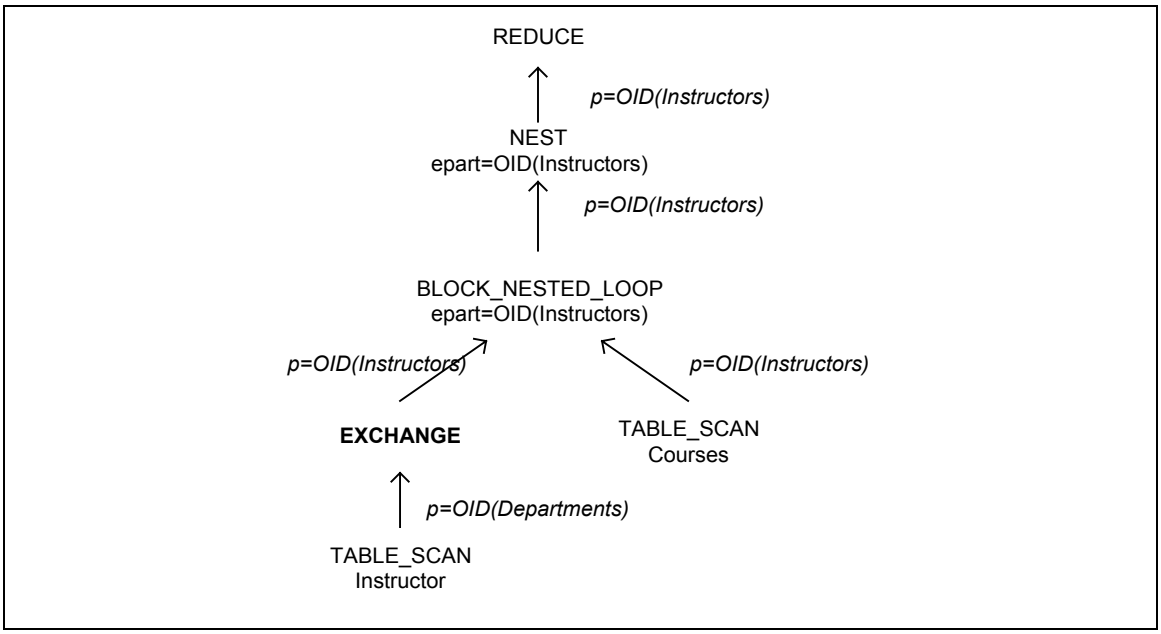
Essa consulta seleciona o nome de todos os instrutores que ensinam mais de quatro cursos, juntamente com a quantidade de cursos ensinados.

No plano gerado, os dados produzidos na leitura de *Instructors* são reparticionados para possibilitar a avaliação da expressão de caminho *e.*teaches através de uma junção. A contagem dos cursos por instrutor é realizada pelo operador NEST. Finalmente, o operador REDUCE disponibiliza os resultados utilizando o predicado *and(geq(_X78,4)))* para selecionar os instrutores que ensinam mais de quatro cursos. Note que a variável *_X78* representa o resultado do NEST, ou seja, a quantidade de cursos por instrutor.

Plano 6:

```
REDUCE(bag,
       NEST(sum,
            BLOCK_NESTED_LOOP(bag,
                  EXCHANGE(
                        TABLE_SCAN(bag,
                                    Instructors,
                                    _X67,
                                    and()),
                        datapartition(OID(_X67))),
                  TABLE_SCAN(bag,
                              Courses,
                              _X68,
                              and()),
                  and(eq(project(_X68,taught_by,Instructor), OID(_X67))),
                  _X67),
            _X78,
            1,
            _X67,
            and(),
            and()),
       _X66,
       struct(bind(name,project(_X67,name,string)),bind(c,_X78)),
       and(geq(_X78,4)))
```

Árvore de Operadores 6:



Consulta 7:

```
select x, y, c: count(c)
from e in Instructors,
    c in e.teaches
group by x: e.ssn, y: c.name
having x > 60 and y > "Fundamentals39"
```

Detalhe 7:

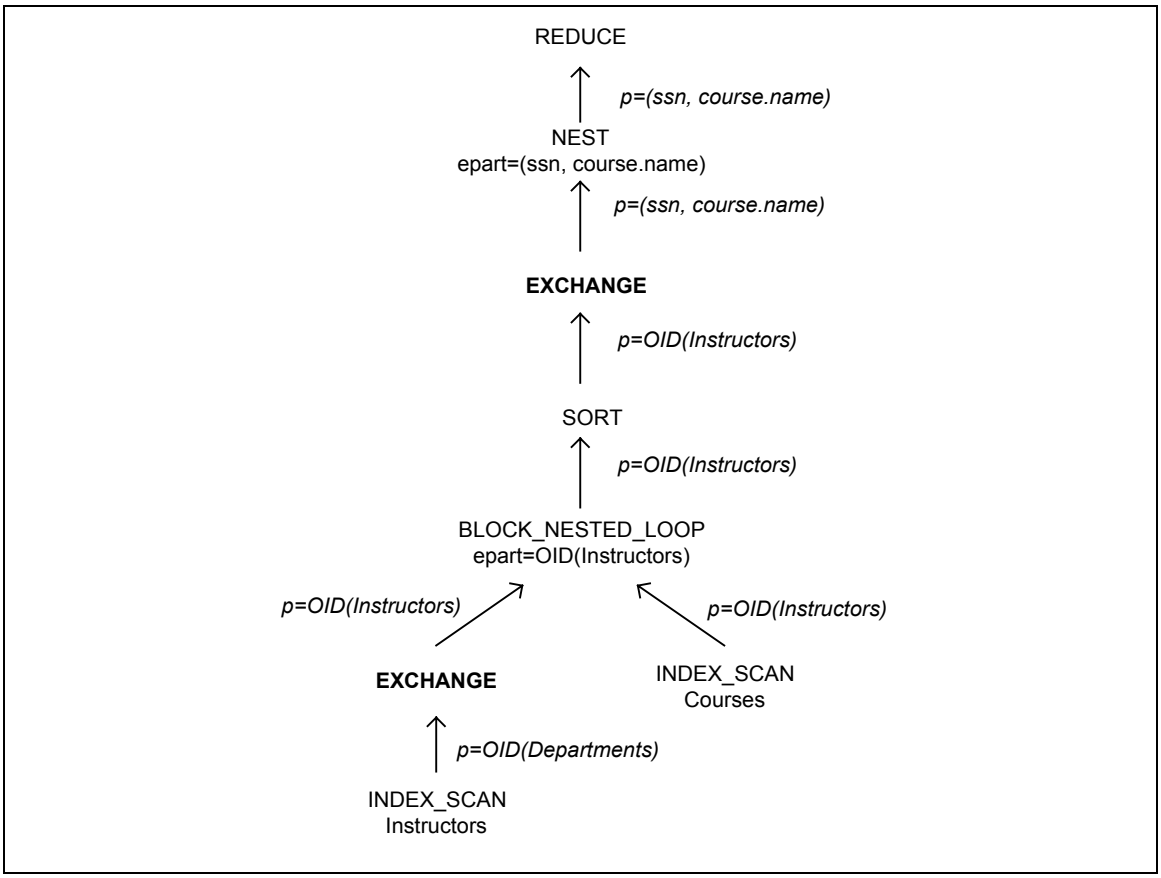
Essa consulta recupera o número de seguro social de instrutores, dos cursos ensinados por eles e a quantidade de cursos ensinados. O resultado é agrupado por nome de instrutor e curso. Só são selecionados instrutores com seguro social maior que sessenta e cursos superiores ao “Fundamentals39”.

No plano gerado, a expressão de caminho *e.teaches* é avaliada com uma junção de ponteiros. Nesse plano, duas operações de reparticionamento de dados são executadas (EXCHANGE). A primeira reparticiona os instrutores selecionados pelo INDEX_SCAN utilizando o valor de *OID(Instructors)*. Depois, o resultado da junção é reparticionado por uma partição composta sobre *Instructor::ssn* e *Course::name*.

Plano 7:

```
REDUCE(bag,
       NEST(sum,
          EXCHANGE(
            SORT(BLOCK_NESTED_LOOP(bag,
                   EXCHANGE(
                      INDEX_SCAN(bag,
                                 Instructors,
                                 _X116,
                                 and(gt(project(_X116,ssn,long),60)),
                                 _index_Instructor_0,
                                 60,
                                 none),
                   datapartition(OID(_X116))),
                   INDEX_SCAN(bag,
                              Courses,
                              _X117,
                              and(gt(project(_X117,name,string),
                                             "Fundamentals39")),
                              _index_Course_1,
                              "Fundamentals39",
                              none),
                   and(eq(project(_X117,taught_by,Instructor),OID(_X116))),
                   none),
               order(project(_X116,ssn,long),project(_X117,name,string))),
            dapartition(project(_X116,ssn,long),project(_X117,name,string))),
          _X141, 1,
          pair(project(_X116,ssn,long),project(_X117,name,string)),
          and(), and()),
       _X115,  struct(bind(x,project(_X116,ssn,long)),
                      bind(y,project(_X117,name,string)),bind(c,_X141)),
       and())
```

Árvore de Operadores 7:



Consulta 8:

```
select x: x, y: count(e)
from e in Instructors
group by x: count(e.teaches)
having x>0
```

Detalhe 8:

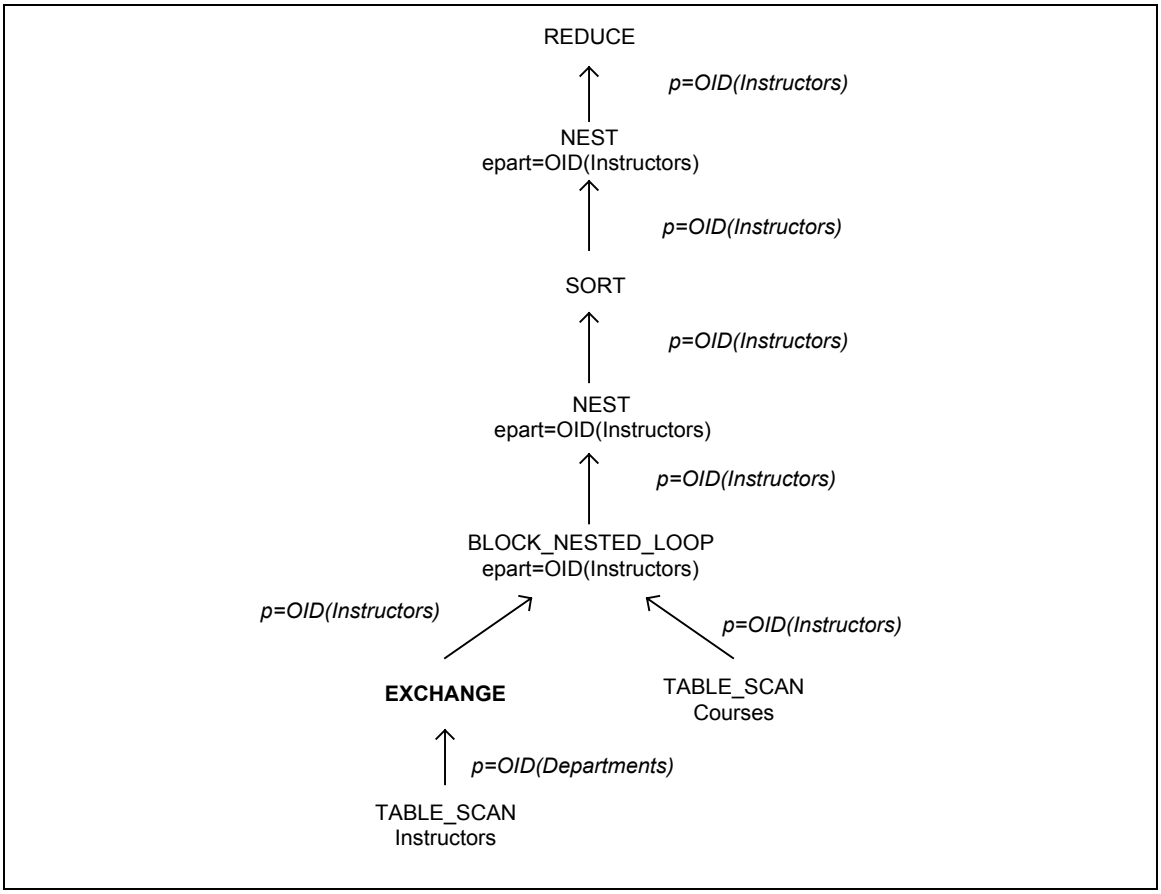
Essa consulta recupera a quantidade de cursos ensinados por cada instrutor e a quantidade de instrutores. O resultado é agrupado pela quantidade de cursos de cada instrutor. Apenas instrutores que estão a ensinar cursos são selecionados.

No plano gerado, a expressão de caminho *e.teaches* é avaliada com uma junção de ponteiros com reparticionado sobre a leitura de *Instructors*. O resultado é agrupado por instrutor (variável *_X290*) e a quantidade de cursos ensinados é sumarizada pelo operador NEST mais interno. Depois, o NEST mais externo agrupa os dados pela quantidade de cursos de cada instrutor e conta os instrutores. O resultado final é um *bag* gerado pelo REDUCE.

Plano 8:

```
REDUCE(bag,
       NEST(sum,
            SORT(
              NEST(sum,
                 BLOCK_NESTED_LOOP(bag,
                      EXCHANGE(
                           TABLE_SCAN(bag,
                                      Instructors,
                                      _X290,
                                      and()),
                           datapartition(OID(_X290))),
                      TABLE_SCAN(bag,
                                 Courses,
                                 _X291,
                                 and()),
                      and(eq(project(_X291,taught_by,Instructor),OID(_X290))),
                      _X290),
                 _X305,
                 1,
                 _X290,
                 and(),
                 and()),
               order(_X305)),
            _X330,
            1,
            _X305,
            and(),
            and()),
       _X289,
       struct(bind(x,_X305),bind(y,_X330)),
       and(gt(_X305,0)))
```

Árvore de Operadores 8:



Consulta 9:

```
select x, y, c: count(e)
from e in Instructors
group by
       x: count(e.teaches),
      y: (exists c in e.teaches: c.name="Fundamentals39")
having x>0
```

Detalhe 9:

Essa consulta recupera a quantidade de cursos ensinados por cada instrutor que ensine algum curso, juntamente com um indicativo se um desses cursos é o *Fundamentals39* e a quantidade de instrutores nessas condições. O resultado é agrupado pela quantidade de cursos de cada instrutor e pelo indicativo da existência do *Fundamentals39*.

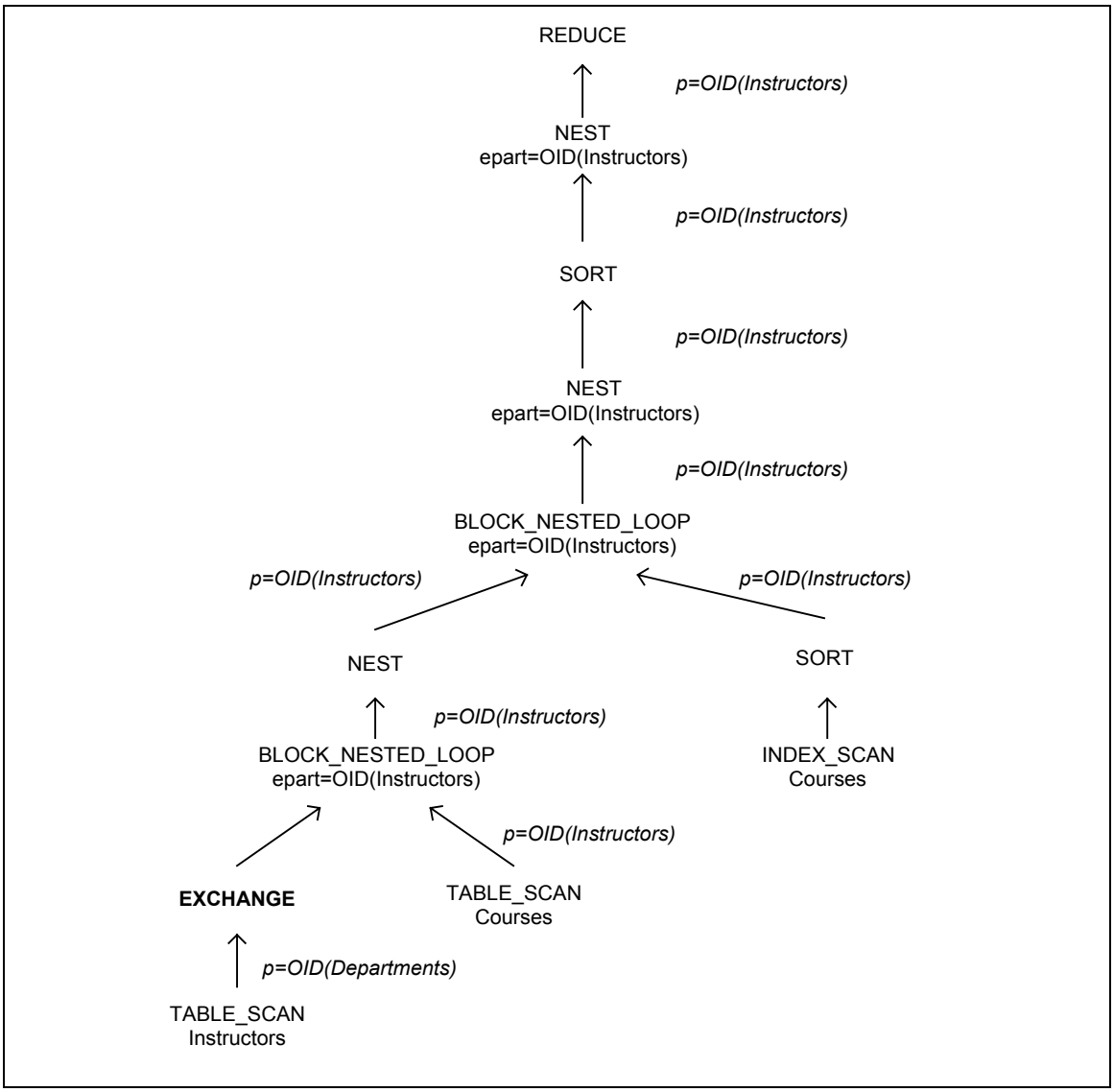
No plano gerado, a expressão de caminho *e.teaches* é avaliada com uma junção de ponteiros com reparticionado sobre a leitura de *Instructors*. A quantidade de cursos por instrutor é sumarizada pelo operador NEST mais interno. Esse resultado é unido aos objetos em *Courses* para verificação da existência do curso *Fundamentals39* pelo operador NEST do meio. Essa verificação é feita com o uso do monóide *some*. O resultado desse NEST é ordenado e depois utilizado pelo NEST mais externo para contagem dos instrutores nessas condições.

Note que apesar da quantidade de operadores, o plano utiliza apenas um EXCHANGE, minimizando a comunicação de dados.

Plano 9:

```
REDUCE(bag,
       NEST(sum,
            SORT(
              NEST(some,
                   BLOCK_NESTED_LOOP(set,
                     NEST(sum,
                          BLOCK_NESTED_LOOP(bag,
                             EXCHANGE(
                                TABLE_SCAN(bag,
                                           Instructors,
                                           _X516,
                                           and()),
                                datapartition(OID(_X516))),
                             TABLE_SCAN(bag,
                                        Courses,
                                        _X517,
                                        and()),
                             and(eq(project(_X517,taught_by,Instructor),
                             OID(_X516))),
                             _X516),
                          _X534,
                          1,
                          _X516,
                          and(),
                          and()),
                     SORT(
                        INDEX_SCAN(set,
                                   Courses,
                                   _X519,
                                   and(eq(project(_X519,name,string),
                                                  "Fundamentals39")),
                                   _index_Course_1,
                                   "Fundamentals39",
                                   "Fundamentals39"),
                        order(_X516,_X534)),
                     and(eq(project(_X519,taught_by,Instructor),OID(_X516))),
                     pair(_X516,_X534)),
                   _X553,
                   true,
                   pair(_X516,_X534),
                   and(),
                   and()),
                order(_X534,_X553)),
            _X579,
            1,
            pair(_X534,_X553),
            and(),
            and()),
       _X515,
       struct(bind(x,_X534),bind(y,_X553),bind(c,_X579)),
       and(gt(_X534,0)))
```

Árvore de Operadores 9:



Consulta 10:

```
select x: x, y: count(e)
from d in Departments,
     e in d.instructors
group by x: count(e.teaches)
having x>0
```

Detalhe 10:

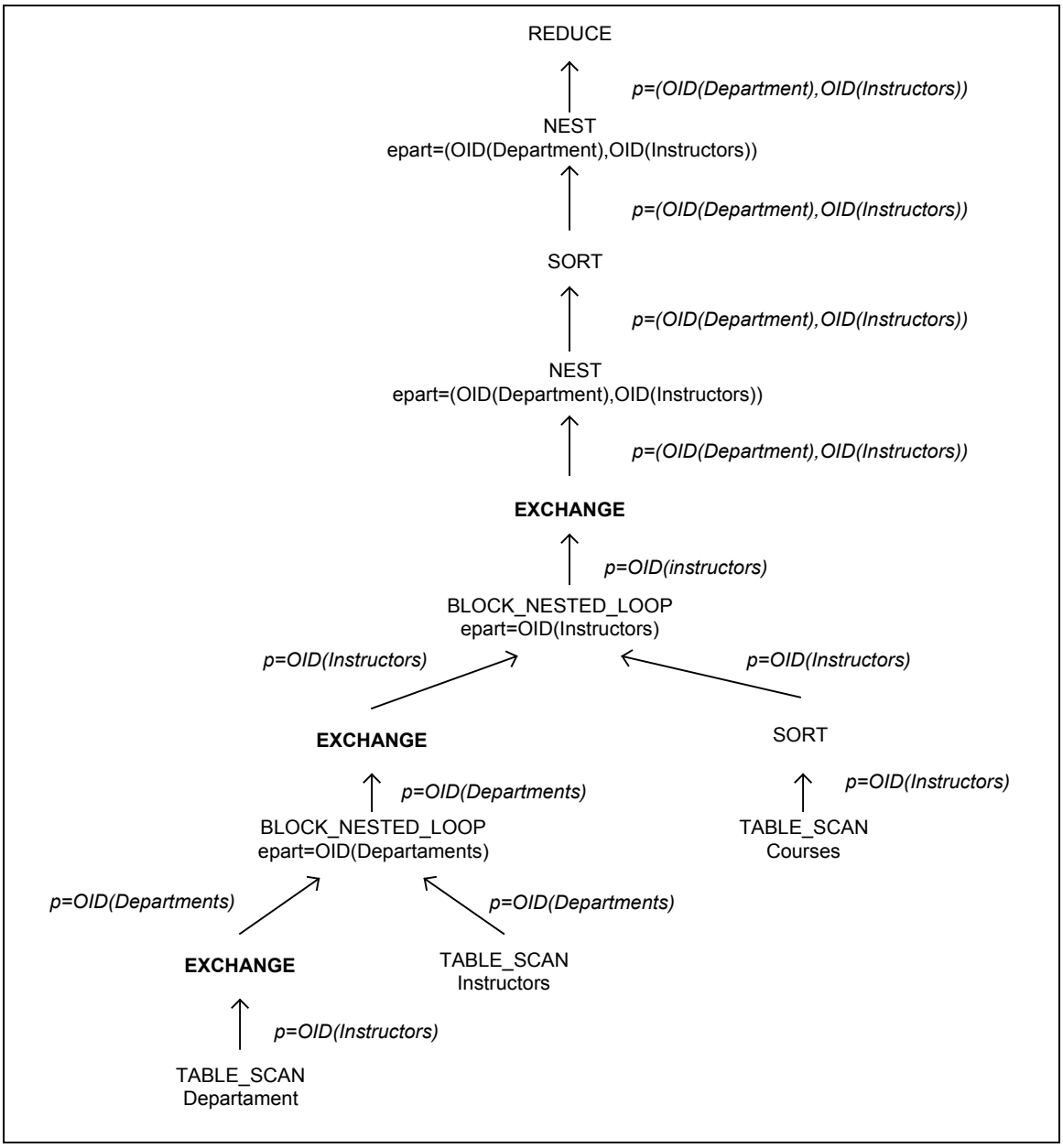
Essa consulta recupera a quantidade de cursos ensinados por cada instrutor e a desses instrutores que estão subordinados a um departamento. O resultado é agrupado pela quantidade de cursos de cada instrutor. Apenas instrutores que estão a ensinar cursos são selecionados.

No plano gerado, a expressão de caminho *d.instructors* é avaliada com uma junção de ponteiros paralelizada nos nós indicados pelo *OID(Departments)*. Para isso, os dados oriundos de *Department* são reparticionados com o EXCHANGE. O resultado dessa junção é novamente reparticionado, mas desta vez pelo *OID(Instructors)*. Isso ocorre para viabilizar a junção com *Courses* pelo atributo *Course::taught_by*. Novamente, um reparticionamento é necessário devido ao agrupamento do resultado da Segunda junção por departamento e instrutor. Esse agrupamento e soma dos respectivos cursos é feito pelo NEST mais interno. O NEST mais externo sumariza os instrutores nessas condições.

Plano 10:

```
REDUCE(bag,
       NEST(sum,
            SORT(
              NEST(sum,
                EXCHANGE(
                   BLOCK_NESTED_LOOP(bag,
                      EXCHANGE(
                         BLOCK_NESTED_LOOP(bag,
                             EXCHANGE(
                                TABLE_SCAN(bag,
                                          Departments,
                                          _X386,
                                          and()),
                                datapartition(OID(_X386))),
                             TABLE_SCAN(bag,
                                       Instructors,
                                       _X387,
                                       and()),
                             and(eq(project(_X387,dept,Department),
                                                                  OID(_X386))),
                             none),
                         datapartition(OID(_X387))),
                      SORT(
                         TABLE_SCAN(bag,
                                    Courses,
                                    _X388,
                                    and()),
                         order(_X386,_X387)),
                      and(eq(project(_X388,taught_by,Instructor),
                                                                  OID(_X387))),
                      pair(_X386,_X387)),
                datapartition(_X386,_X387)
                _X411,
                1,
                pair(_X386,_X387),
                and(),
                and()),
              order(_X411)),
            _X436,  1,  _X411,
            and(),
            and()),
       _X385,
       struct(bind(x,_X411),bind(y,_X436)),
       and(gt(_X411,0)))
```

Árvore de Operadores 10:



Consulta 11:

```
sum(select sum(select e.salary from e in d.instructors)
    from d in Departments);
```

Detalhe 11:

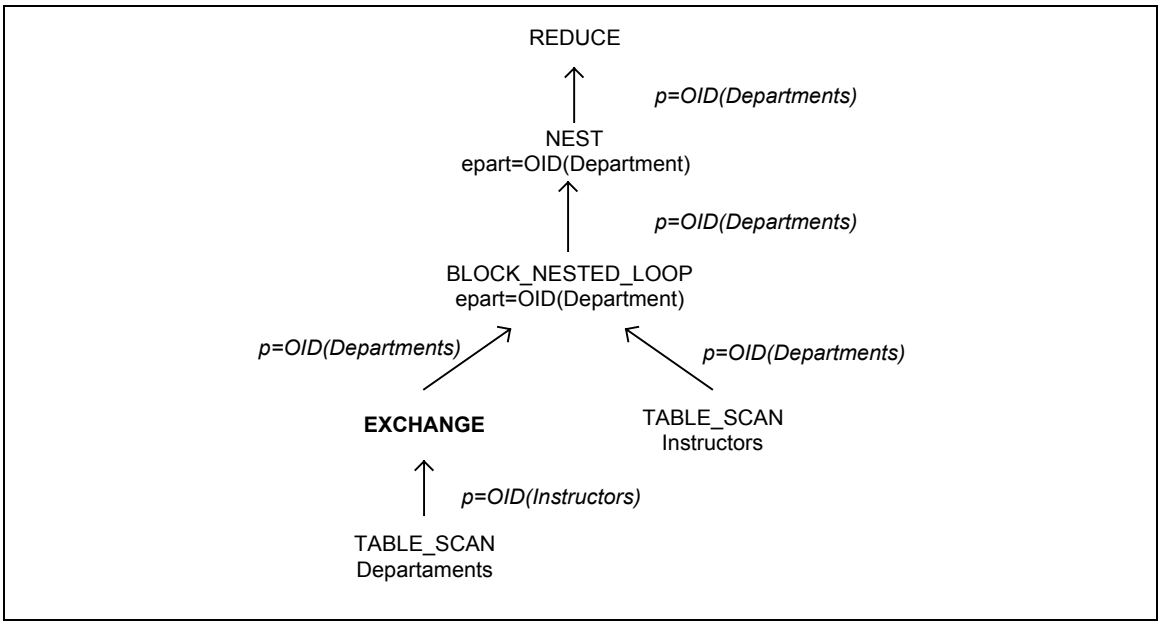
Essa consulta recupera a soma dos salários de instrutores subordinados a departamentos.

No plano gerado, a expressão de caminho *d.instructors* é avaliada com uma junção com ponteiros. Os dados oriundos de *Departments* são reparticionados de acordo com *OID(Deparments*). O resultado da junção é somado pelo NEST com agrupamento sobre o departamento.

Plano 11:

```
REDUCE(bag,
       NEST(sum,
            BLOCK_NESTED_LOOP(bag,
                      EXCHANGE(
                         TABLE_SCAN(bag,
                                   Departments,
                                   _X37,
                                   and()),
                         datapartition(OID(_X37))),
                      TABLE_SCAN(bag,
                                 Instructors,
                                 _X38, and()),
                      and(eq(project(_X38,dept,Department),OID(_X37))),
                      _X37),
            _X47,  project(_X38,salary,long),
            _X37,  and(), and()),
       _X36,_X47,and())
```

Árvore de Operadores 11:



Consulta 12:

```
select e.name,
       X: (select x
           from c in e.teaches
           group by x: count(c.has_prerequisites))
from d in Departments,
    e in d.instructors
```

Detalhe 12:

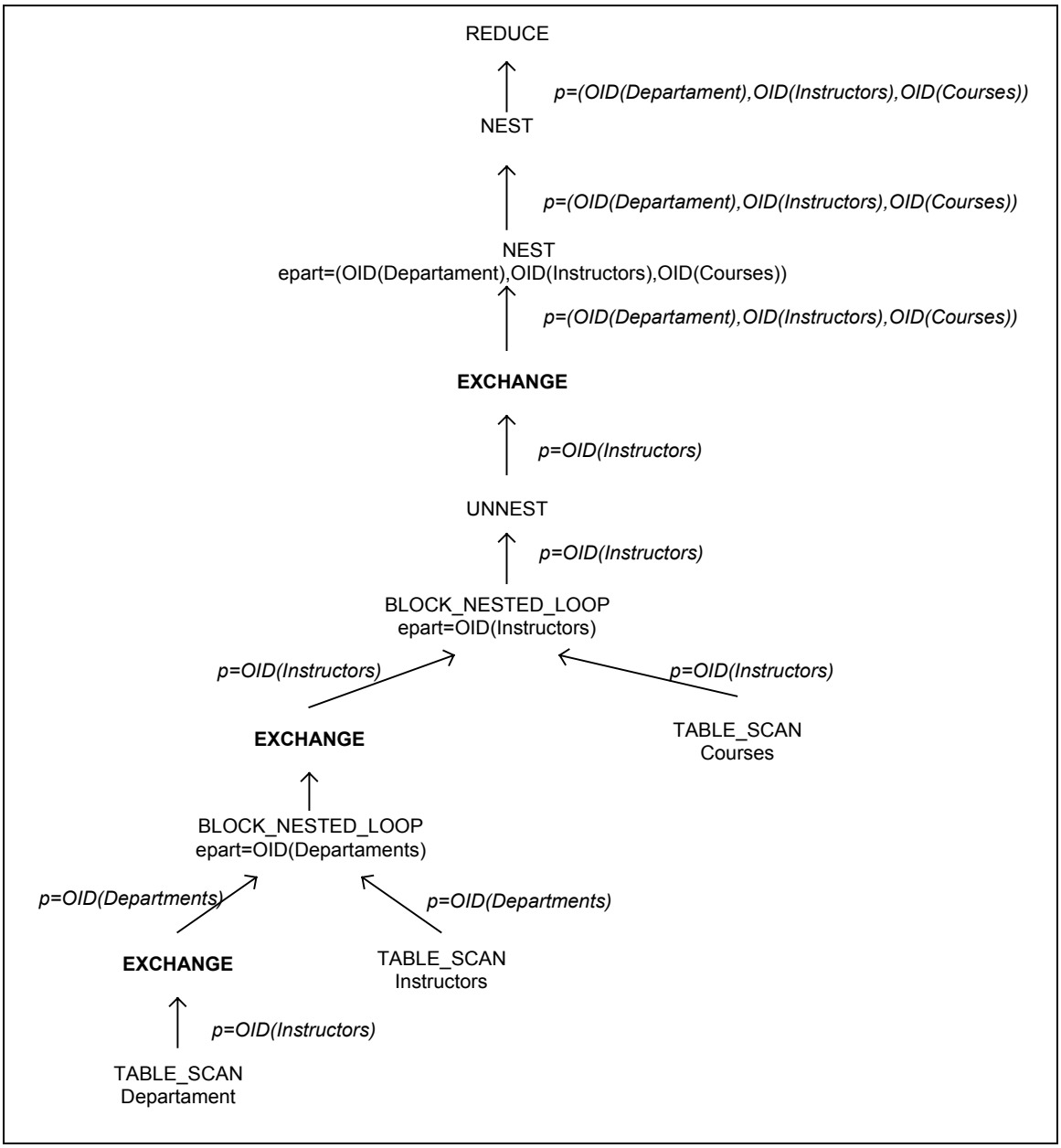
Essa consulta recupera o nome dos empregados subordinados a departamentos, e uma lista de cursos ensinados por esses departamentos. Essa lista é agrupada pela quantidade de pré-requisitos para esses cursos.

No plano gerado, a expressão de caminho *d.instructors* é avaliada com uma junção com ponteiros. Os dados oriundos de *Departments* são reparticionados de acordo com *OID(Deparments*). O resultado da junção é novamente reparticionado para viabilizar a avaliação da expressão de caminho *e.teaches*. Os cursos obtidos nessa expressão, têm seu atributo *has_prerequisites* desaninhado pelo operador UNNEST, que logo a seguir é reparticionado para viabilizar a contagem de pré-requisitos pelo operador NEST mais interno. Depois, os cursos obtidos nessas condições são aninhados, no formato de *bag*, por departamentos, empregados e curso com o uso do NEST mais externo.

Plano 12:

```
REDUCE(bag,
       NEST(bag,
            NEST(sum,
              EXCHANGE(
                 UNNEST(bag,
                     BLOCK_NESTED_LOOP(bag,
                        EXCHANGE(
                          BLOCK_NESTED_LOOP(bag,
                             EXCHANGE(
                                TABLE_SCAN(bag,
                                           Departments,
                                           _X192,
                                           and()),
                                datapartition(OID(_X192))),
                             TABLE_SCAN(bag,
                                        Instructors,
                                        _X193,
                                        and()),
                             and(eq(project(_X193,dept,Department),
                                                                  OID(_X192))),
                             none),
                          datapartition(OID(_X193))
                        TABLE_SCAN(bag,
                                   Courses,
                                   _X194,
                                   and()),
                        and(eq(project(_X194,taught_by,Instructor),
                                                                  OID(_X193))),
                        pair(_X192,_X193)),
                     _X195,
                     project(_X194,has_prerequisites,set(Course)),
                     and(),
                     pair(pair(_X192,_X193),_X194)),
                 datapartition(pair(pair(_X192,_X193),_X194)))
                 _X221,
                 1,
                 pair(pair(_X192,_X193),_X194),
                 and(),
                 and()),
            _X211,
            _X221,
            pair(_X192,_X193),
            and(),
            and()),
       _X191,
       struct(bind(name,project(_X193,name,string)),bind(X,_X211)),
       and())
```

Árvore de Operadores 12:



Consulta 13:

```
select e
from e in Instructors
where for all c in e.teaches:
             exists d in c.is_prerequisite_for:
                           d.name = "Fundamentals10"
```

Detalhe 13:

Essa consulta recupera os instrutores que ensinam cursos que tenham o curso *Fundamentals10* como pré-requisito.

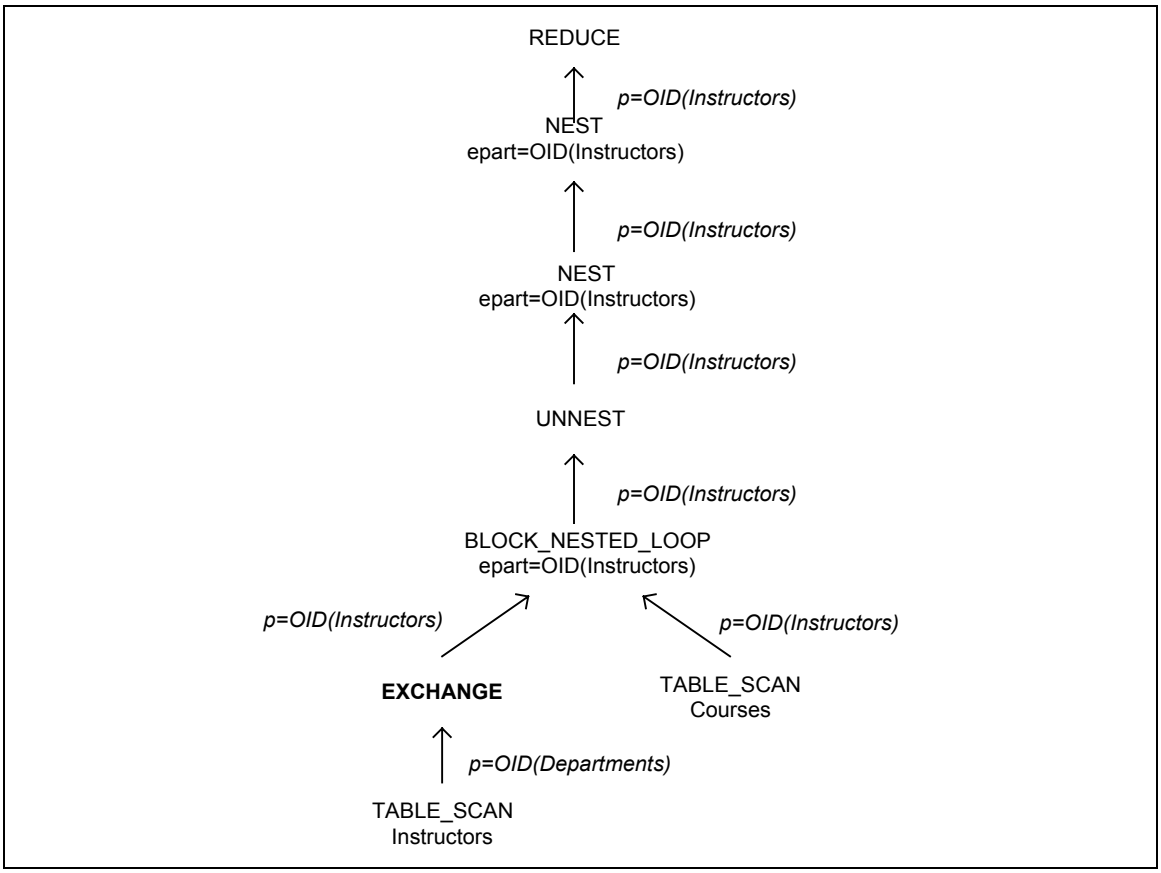
No plano gerado, a expressão de caminho *e.teaches* é avaliada com uma junção com ponteiros. Os dados oriundos de *Instructors* são reparticionados pelo EXCHANGE de acordo com *OID(Instructors*). A coleção de pré-requisitos é desaninhada do resultado da junção pelo UNNEST e um monóide *set* é criado. Os dois operadores NEST em seqüência verificam a existência de algum curso que tenha *Fundamentals10* como pré-requisito, e a condição para todos os cursos ensinados pelo instrutor.

Plano 13:

```
REDUCE(bag,
       NEST(all,
            NEST(all,
                 UNNEST(set,
                        BLOCK_NESTED_LOOP(set,
                           EXCHANGE(
                              TABLE_SCAN(bag,
                                         Instructors,
                                         _X85,
                                         and()),
                              datapartition(OID(_X85))),
                           TABLE_SCAN(set,
                                      Courses,
                                      _X86,
                                      and()),
                           and(eq(project(_X86,taught_by,Instructor),
                                                              OID(_X85))),
                              _X85),
                        _X87,
                        project(_X86,is_prerequisite_for,set(Course)),
                        and(eq(project(_X87,name,string),"Fundamentals10")),
                        pair(_X85,_X86)),
                 _X106,
                 false,
                 pair(_X85,_X86),
                 and(),
                 and()),
            _X96,
            false,
```

```
      _X85,
      and(_X106),
      and()),
_X84,
_X85,
and(_X96))
```

Árvore de Operadores 13:



5.4.2 Plano com paralelismo intra-operador e inter-operador

Essa seção apresenta o plano de execução para a consulta 1 com a utilização da fase de otimização com paralelismo intra-operador e inter-operador. Note que a fase inter-operador não modifica as propriedades físicas do plano e assim propriedades como custo e tamanho permanecem as mesmas que foram estimadas somente com o paralelismo intra-operador.

Nesse plano, indica-se a ocorrência de restrição de paralelismo [PaC] e restrição de precedência [PrC]. Se entre dois operadores existe uma restrição de paralelismo, esses operadores podem ser executados em paralelo através de *pipelining*. É o caso, por exemplo, dos operadores GROUP e MATCH. A medida que os dados são unidos através da operação de MATCH, o operador GROUP já realiza o agrupamento das informações produzidas e evita a materialização da junção.

Já a restrição de precedência indica que o operador consumidor deve esperar a finalização do operador produtor. O reparticionamento de dados por vezes necessário ao paralelismo intra-operador é indicado pela presença do operador EXCHANGE.

```
REDUCING(bag,
   [PrC] EVALUATE_HEAD(
        [PaC] REDUCING(bag,
            [PrC] EVALUATE_HEAD(
                [PaC] NESTING(
                    [PrC] GROUP(
                      [PaC] MATCH(bag,
                          [PrC] READ_CHUNK(
                                    EXCHANGE(
                               [PaC] TABLE_SCAN(bag,
                                                Instructors,
                                                _X35,
                                                and()),
                                     datapartition(OID(_X35)))),
                          [PaC] TABLE_SCAN(bag,
                                           Courses,
                                           _X36,
                                           and()),
                                and(eq(project(_X36, taught_by,Instructor),
                                                                 OID(_X35))),
                                _X35),
                             _X35),
                          and()),
                    project(_X36,name,string),
                    and()),
                _X45,
                project(_X36,name,string)),
```

```
                                struct(bind(x,project(_X35,name,string)),
                                bind(y,_X45))),
_X34,
struct(bind(x,project(_X35,name,string)),bind(y,_X45)))
```

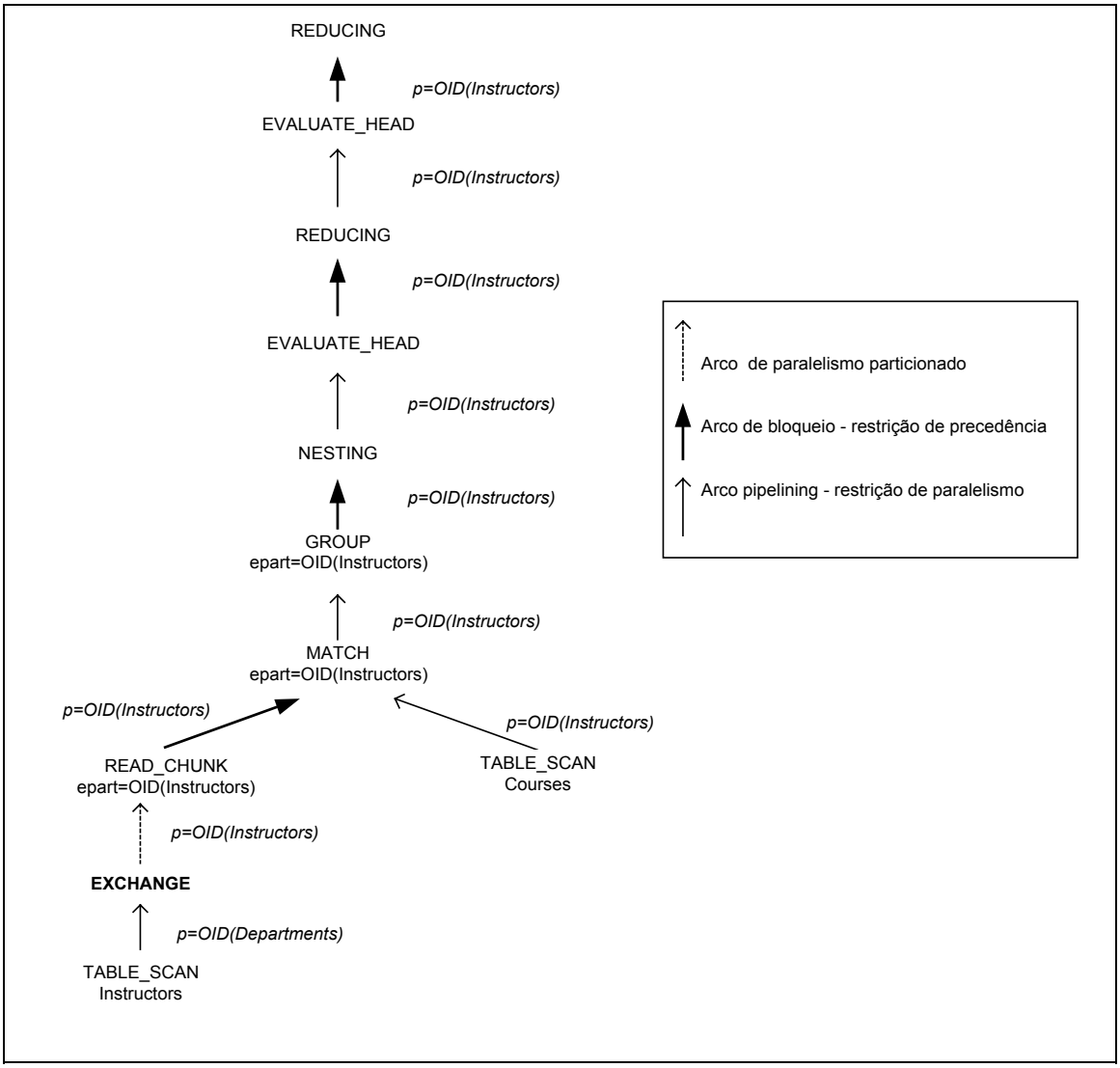


Figura 5.2 - Plano com paralelismo intra-operador e inter-operador

5.4.3 Análise de desempenho

Para avaliar resultados da proposta quanto ao desempenho, os tempos de execução e otimização proporcionados pelo novo otimizador foram comparados ao otimizador original do Lambda-DB. No entanto, esse SGBDOO não fornece suporte para a execução paralela de consultas. Assim, os tempos de execução dos planos paralelos foram estimados com base no tempo de execução seqüencial, no custo do plano seqüencial e no custo do plano paralelo.

Os testes foram realizados em uma máquina paralela com 2 processadores Pentium III 750 MHz, 512 MB de memória- HD de 18 GB scsi ultra. O Lambda-DB está instalado sobre o sistema operacional Linux Red Hat 6.2 e utiliza o gerenciador de objetos SHORE. Os tempos foram obtidos com a função *clock* do C++, que obtém o tempo de CPU. Os valores finais considerados representam a média aritmética de três execuções distintas para cada consulta. As consultas utilizadas nos testes foram apresentadas na seção 5.4.1.

Os resultados são apresentados em duas tabelas. As Tabela 5.2 e Tabela 5.3 apresentam os resultados em diferentes massas de dados. A primeira coluna das tabelas identifica a consulta realizada sobre as extensões *Departments*, *Instructors* e *Courses*. A segunda coluna apresenta os tempos de otimização, execução e tempo de resposta total para o otimizador seqüencial. A quantidade de dados para cada extensão é apresentada no cabeçalho da coluna. Por exemplo, a Tabela 5.2 indica, na segunda coluna, que as extensões *Departments*, *Instructors* e *Courses* estavam carregadas com 10, 100 e 50 objetos respectivamente. A terceira coluna apresenta o mesmo tipo de informação para o otimizador paralelo. A última coluna apresenta o percentual de cada um dos tempos do otimizador paralelo em relação ao otimizador seqüencial.

Department/ Instructors/ Courses	Otimizador Sequencial 10/100/50	Otimizador Paralelo 10/100/50	% Paralelo-Seqüencial
	Otimização/ Execução/ Total	**Otimização/ Execução/ Total**	**Otimização/ Execução/ Total (%)**
Consulta 1:	0,01 / 0,65 / 0,66	0,03 / 0,25 / 0,28	300/38/42%
Consulta 2:	0,03 / 1,81 / 1,84	0,09 / 0,60 / 0,69	300/33/37%
Consulta 3:	0,02 / 0,76 / 0,78	0,11 / 0,46 / 0,57	550/60/73%
Consulta 4:	0,02 / 0,10 / 0,12	0,02 / 0,07 / 0,09	100/70/75%
Consulta 5:	0,02 / 0,16 / 0,18	0,05 / 0,06 / 0,11	250/37/61%
Consulta 6:	0,01 / 0,65 / 0,66	0,03 / 0,25 / 0,28	300/38/42%
Consulta 7:	0,01 / 0,19 / 0,20	0,07 / 0,13 / 0,20	700/68/100%
Consulta 8:	0,01 / 0,70 / 0,71	0,08 / 0,25 / 0,33	800/35/46%
Consulta 9:	0,03 / 1,00 / 1,03	0,20 / 0,68 / 0,88	667/68/85%
Consulta 10:	0,09 / 0,90 / 0,99	0,21 / 0,42 / 0,63	233/46/63%
Consulta 11:	0,01 / 0,24 / 0,25	0,05 / 0,09 / 0,14	500/37/56%
Consulta 12:	0,03 / 1,00 / 1,03	0,15 / 0,47 / 0,62	500/47/60%
Consulta 13:	0,03 / 0,73 / 0,76	0,08 / 0,26 / 0,34	260/35/44%

Tabela 5.2 - Tempos massa de dados 1

Através da Tabela 5.2 pode-se observar que os tempos de otimização para planos paralelos são em média 4 vezes maiores do que os tempos de otimização para ambientes seqüenciais.

Já os tempos de execução para planos paralelos são em média a metade dos tempos de planos seqüenciais. Apesar das simulações utilizarem um ambiente com grau de paralelismo igual a 3, os tempos não reduziram para um terço do original. Isso é devido a ocorrência do operador EXCHANGE nos planos para realizar a repartição dinâmica dos dados, onerando assim os resultados com o tempo necessário para a comunicação dos dados.

É importante observar que alguns dos tempos totais paralelos ficaram quase equivalentes aos seqüenciais. Isso ocorre devido ao baixo volume de dados (10, 100 e 50 para *Departments, Instructors* e *Courses*)existente na configuração de testes usada na Tabela 5.2. Veja que os tempos de otimização paralela representam um percentual maior do tempo total quando comparados ao tempo total do otimizador seqüencial. A consulta 7, por exemplo, apresenta um tempo de otimização paralelo bem maior que o seqüencial. Como a massa de dados é pequena, o *overhead* do processo de otimização se torna mais significativo, aproximando os tempos totais dos otimizadores seqüencial e paralelo.

Departments/ Instructors/ Courses	Otimizador seqüêncial 10/1000/500	Otimizador Paralelo 10/1000/500	% Paralelo-Seqüencial
	Otimização/ Execução/ Total	**Otimização/ Execução/ Total**	**Otimização/ Execução/ Total (%)**
Consulta 1:	0,01 / 53,43 / 53,44	0,04 / 17,81 / 17,85	400/33/33 %
Consulta 2:	0,03 / 237,65 / 237,68	0,09 / 79,31 / 79,40	300/32/33 %
Consulta 3:	0,02 / 58,92 / 58,94	0,10 / 19,64 / 19,74	500/33/33 %
Consulta 4:	0,02 / 01,14 / 01,16	0,03 / 00,51 / 00,54	150/44/46 %
Consulta 5:	0,02 / 00,80 / 00,82	0,04 / 00,28 / 00,32	200/35/39 %
Consulta 6:	0,01 / 54,07 / 54,08	0,03 / 18,01 / 18,04	300/33/33 %
Consulta 7:	0,01 / 18,68 / 18,69	0,07 / 07,21 / 07,28	700/38/39 %
Consulta 8:	0,01 / 57,07 / 57,08	0,10 / 19,02 / 19,12	1000/33/33 %
Consulta 9:	0,04 / 54,50 / 54,54	0,19 / 17,31 / 17,50	475/31/32 %
Consulta 10:	0,08 / 55,80 / 55,88	0,21 / 29,00 / 29,21	263/51/52 %
Consulta 11:	0,01 / 55,81 / 55,82	0,05 / 18,78 / 18,83	500/33/33 %
Consulta 12:	0,03 / 57,01 / 57,04	0,17 / 30,02 / 30,19	566/52/53 %
Consulta 13:	0,03 / 53,75 / 53,78	0,09 / 17,91 / 18,00	300/33/33 %

Tabela 5.3 - Tempos massa de dados 2

A Tabela 5.3 repete os testes realizados na Tabela 5.2 e utiliza um maior volume de dados. Dessa vez os ganhos com o paralelismo são mais representativos. A consulta 7, por exemplo, havia fornecido tempos equivalentes em ambientes seqüencial e paralelo no teste anterior. Na nova configuração, a utilização do paralelismo nessa consulta apresenta bons ganhos. Isso é devido ao fato de os tempos de otimização não apresentarem a mesma ordem de grandeza da configuração anterior, e esses tempos não representam um percentual significativo do tempo de execução.

5.5 Conclusões do capítulo

Este capítulo apresentou a fase de especificação da metodologia juntamente com exemplos e resultados obtidos na utilização da técnica no sistema OPTGEN e no otimizador para consultas com paralelismo construído a partir do otimizador do SGBDOO Lambda-DB. Observou-se pelos experimentos que o paralelismo pode reduzir o tempo de resposta das consultas proporcionalmente ao grau de paralelismo utilizado. No entanto, se o plano apresentar operadores de reparticionamento (EXCHANGE) os ganhos podem ser reduzidos e em alguns casos deixar até de haver ganhos com relação à execução seqüencial. A técnica proposta evita a utilização de operadores EXCHANGE ou utiliza esse operador sobre particionamentos que gerem um menor custo de comunicação de dados.

Apesar dos tempos de otimização com paralelismo serem maiores que os tempos seqüenciais, verificou-se que a perda é compensada pelo menor tempo de execução da consulta em paralelo.

O capítulo seguinte apresenta as conclusões do trabalho e os próximos passos a serem realizados.

Capítulo 6

Conclusões e Trabalhos Futuros

Este capítulo apresenta um resumo das contribuições fornecidas por este trabalho. Além disso, são analisadas as dificuldades encontradas e as possíveis abordagens para trabalhos futuros.

Os principais resultados desta dissertação foram aceitos para apresentação em [PM01].

6.1 Contribuições

Este trabalho apresentou uma técnica para construção de otimizadores para SGBDOO's paralelos. Essa técnica caracteriza-se por:

1. Estender o modelo de otimização paralela em duas fases proposto em [Has96]
2. Fornecer uma abordagem sobre uma abstração do modelo de otimização *Top-down* conduzido por regras e baseado em custos.
3. Fornecer um algoritmo de reordenação de operadores para SGBDOO's paralelos.

O trabalho apresentado em [Has96] forneceu algoritmos para a implementação da fase de seleção de operadores físicos e reordenação de operadores no contexto de bancos dados relacionais. Foi sugerido que a incorporação dos algoritmos aos otimizadores convencionais fosse realizada através da: a) Substituição das fases equivalentes pelos algoritmos propostos b) Adição dos algoritmos como uma fase "post-past" com relação as fases convencionais.

Este trabalho estende o modelo proposto de três formas. Primeiro, propõe que a incorporação da técnica não seja feita ao nível do otimizador, e sim ao nível de sistemas de construção de otimizadores (meta-nível). Para isso, a técnica

procedimental teve que ser estendida para o modelo de regras de reescrita de termos. Depois, o trabalho utilizou o conceito de otimização *Top-down* e passagem de propriedades físicas esperadas para reduzir o número de planos alternativos avaliados. Por fim, este trabalho aplicou o modelo de otimização no contexto de bancos de dados orientados a objetos e forneceu a análise de uma álgebra orientada a objetos com relação às restrições do paralelismo intra-operador.

Por outro lado, a proposta não se limitou à utilização do modelo de otimização com paralelismo em um *framework* específico. Ao invés disso o trabalho fornece uma abstração do modelo de regras e estratégia *Top-down* utilizada em sistemas como Cascades, OPTGEN e Columbia [SMB01] para especificação do modelo de otimização em duas fases estendido. Essa abstração fornece padrões de regras que funcionam como moldes para facilitar a especificação de um otimizador em qualquer um desses sistemas. Esses padrões são baseados na classificação dos operadores da álgebra física com relação às restrições do paralelismo intra-operador e da fragmentação desses operadores com relação ao paralelismo inter-operador.

Este trabalho também propõe o algoritmo GOO-OOP para reordenação de operadores em SGBDOO's paralelos. Esse algoritmo combina as propostas apresentadas em [Has96] e [Feg98] com a utilização dos custos de reparticionamento como base para avaliar a reordenação de operadores.

Para validação da técnica proposta, uma especificação de otimizador foi criada para o *framework* OPTGEN. Essa especificação partiu da especificação de um otimizador seqüencial para o SGBDOO Lambda-DB e a estendeu com o uso da técnica proposta. Depois, foram realizados experimentos sobre o otimizador gerado a partir da especificação com paralelismo. Os tempos de execução em ambiente paralelo foram estimados a partir dos custos dos planos gerados pelo otimizador original, custos dos planos gerados pelo otimizador com paralelismo, informações sobre o dados e configuração inicial de particionamento, e tempos de execução em ambiente seqüencial. O trabalho fornece tabelas com resultados dos experimentos. Essas tabelas mostram que apesar de o tempo de otimização em ambiente paralelo, o tempo total de resposta

é realmente melhorado quando a base de dados apresenta um volume de dados significativo.

Apesar de a técnica ter sido apresentada no contexto de sistemas orientados a objetos, ela pode ser facilmente aplicada a outros modelos como o relacional e objeto-relacional, como também para diferentes álgebras.

6.2 Trabalhos Futuros

Esta seção apresenta algumas possibilidades de trabalhos futuros. Primeiramente será necessário avançar para a construção de um SGBDOO paralelo, ou integrar o Lambda-DB com um gerenciador de objetos com suporte ao paralelismo. Com base no SGBDOO paralelo, estudos sobre o processamento paralelo de consultas com XML e suporte para objetos distribuídos poderão ser realizados.

6.2.1 SGBDOO Paralelo

Este trabalho concentrou-se na construção de otimizadores para SGBDOO's paralelos abordando o paralelismo inter-operador e intra-operador. No entanto, para a construção de um SGBDOO paralelo ainda faz-se necessária a implementação de técnicas de escalonamento do plano de execução. O escalonamento visa à distribuição eficiente dos operadores através dos nós de processamento com base nas anotações geradas no PEC pelo otimizador e em informações dinâmicas sobre a carga do sistema. Em [Has96] alguns algoritmos para escalonamento foram apresentados. Esses algoritmos deverão ser avaliados e adaptados para SGBDOO's. No entanto, devido ao seu caráter dinâmico, o modelo de regras não poderá ser usado para essa tarefa.

O SGBDOO paralelo poderia ser construído a partir do Lambda-DB com o uso de uma instância do SHORE por nó de processamento em uma arquitetura sem compartilhamento.

6.2.2 Otimização para consultas XML em SGBDOO paralelo

As aplicações para Web têm sido uma grande motivação para as pesquisas sobre processamento de consultas e linguagens de consultas para recuperação de documentos. A idéia básica é considerar a Internet como um grande banco de dados distribuído que permite a execução de consultas em paralelo através dos *sites* espalhados ao redor do mundo. Uma abordagem é utilizar linguagens baseadas em extensões do OQL para suportar consultas em XML. Entre algumas dessas propostas pode-se citar WebOQL [AM98] e XML-OQL [FE01]. Essa última utiliza regras para converter uma consulta em XML-OQL para OQL e deixa que o otimizador do sistema Lambda-DB realize a otimização sobre a consulta OQL gerada.

Uma consulta XML pode requisitar dados de documentos que estão em nós remotos e esses documentos, por sua vez, podem referenciar documentos em outros nós. Esse problema é similar às questões discutidas neste trabalho no que se refere às expressões de caminho e comunicação de dados necessária para o processamento dos operadores. No entanto, os custos de reparticionamento são muito mais críticos devido ao desempenho de uma rede tão grande como a internet. A técnica aqui apresentada poderia ser utilizada para a otimização de consultas em tal ambiente distribuído e paralelo com o uso de linguagens de consulta XML baseadas em OQL. No entanto, seria necessário o estudo mais aprofundado sobre questões relativas à falta de informações de meta-dado e estatísticas sobre as consultas e a própria estrutura de documentos Web.

Referências Bibliográficas

[AM98] G. Arocena, A.Meldenzon. *WebOQL: Restructuring Documents, Databases and Webs*. Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Flórida, Fevereiro, 1988.

[BB96] G. Brassard, P. Bratley. *Fundamentals of Algorithmics*. Prentice Hall, 1996.

[Ber94] E. Bertino. *Index configuration in object-oriented databases*. The International Journal on Very Large Data Bases, 1994.

[BMG93] J. A. Blakeley, W. McKenna, G. Graefe. *Experiences Building the Open OODB Query Optimizer*. ACM SIGMOD, Washington, DC , 1993.

[Bor88] H. Boral. *Parallelism in Bubba. Version 1.0*. Proceedings of the International Symp. on Database in in Parallel and Distributed Systems, Austin, Texas, página 68 , Dezembro 1988.

[BRJ97] G. Booch, J. Rumbaugh e I. Jacobson. *Unified Modeling Language, Version 1.0*. Rational Software Corporation, Janeiro 1997.

[CB97] R. Cattel, D. Barry, editors. *The object database standard: ODMG 93*. Morgan Kaufman, 1997.

[DGS88] J. DU, J.Y-T. Leung. *A Performance Analysis of the GAMMA Database Machine*. Proceedings of ACM Sigmod Conference. Chicago, IL, páginas 350-360, Junho 1988.

[FE01] L. Fegaras, R. Elmasri. *Query Engines for Web-Accessible XML Data*. Para publicação no VLDB 2001.

[Feg97a] L. Fegaras. *An experimental optimizer for OQL*. Techinical Report TR-CSE-97-007, CSE, University of Texas at Arlington, 1997.

[Feg97b] L. Fegaras. *The OPTGEN Optimizer Generator. Department of Computer Science and Engineering*. The University of Texas at Arlington, http://ranger.uta.edu/~fegaras/optimizer, 1997.

[Feg98] L. Fegaras. *Query unnesting in object-oriented databases*. ACM SIGMOD International Conference on Management of Data, Seattle, Washington, 1998.

[FM00] L. Fegaras, D. Maier. *Optimizing Object Queries Using an Effective Calculus*. ACM Transactions on Database Systems, Dezembro 2000.

[FSRM00] L. Fegaras, C. Srinivasan, A. Rajendran, D. Maier. *Lambda-DB: An ODMG-Based Object-Oriented DBMS*. ACM SIGMOD International Conference on Management of Data, Dallas, Texas, 2000.

[GD87] G. Graefe, D.J. DeWitt. *The EXODUS Optimizer Generator*. ACM SIGMOD International Conference on Management of Data, San Francisco, California, 1987.

[GHLZ94] S. Ghandeharizadeh, D. Wilhite, K. Lin, X. Zhao. *Object Placement in Parallel Object-Oriented Database Systems*. ICDE, páginas 253-262, 1994.

[GI96] M. N. Garofalakis, Y. E. Ioannidis. *Multi-dimensional Resource Scheduling for Parallel Queries*. ACM SIGMOD International Conference on Management of Data, Montréal, Canadá, 1996.

[GM93] G. Graefe, W. McKenna. *The Volcano Optimizer Generator*. IEEE Conference on Data Engineering, Viena, Austria, 1993.

[Gra90] G. Graefe. *Encapsulation of Parallelism in the Volcano Query Processing System*. ACM SIGMOD International Conference on Management of Data, páginas 102-111, 1990.

[Gra95] G. Graefe. *The Cascades Framework for Query Optimization*. Bulletin of the IEEE Technical Committee on Data Engineering, 18(3), Pages 19-29, Setembro 1995.

[Has96] W. Hasan. *Optimization of SQL Queries for Parallel Machines.* PhD thesis, Stanford University, Department of Computer Science, 1996.

[HL91] K. A. Hua, C. Lee. *Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning*. Proceedings of the 17th International Conference on Very Large Databases, 1991.

[Hol88] R. Holbrook. *NonStop SQL - A Distributed Relational DBMS for OLTP*. Proceedings IEEE Compcon '88 Conf, páginas 418-421, San Francisco, California, Fevereiro 1988.

[HS91] W. Hong, M. Stonebraker. *Optimization of Parallel Query Execution Plans in Xprs*. Proc. 1st Int. PDIS Conference, páginas 218-225, Miami, FL, Dezembro 1991.

[ISO96a] ISO/IEC JTC1/SC21 N10489, ISO//IEC 9075, Part 2, Committee Draft (CD), *Database Language SQL - Part 2: SQL/Foundation,*
ftp://speckle.ncsl.nist.gov/isowg3/dbl/BASEdocs/cd-found.pdf , Julho, 1996.

[ISO96b] ISO/IEC JTC1/SC21 N10491, ISO//IEC 9075, Part 8, Committee Draft (CD), *Database Language SQL -Part 8: SQL/Object,*
ftp://speckle.ncsl.nist.gov/isowg3/dbl/BASEdocs/cd-objct.pdf , Julho, 1996.

[KGM91] T. Keller, G. Graefe, D. Maier. *Efficient Assembly of Complex Objects*. ACM SIGMOD International Conference on Management of Data, Denver, Co, 1991.

[LDM93] D.F. Liewen, D.J. DeWitt, M. Mehta. *Parallel Pointer-Based Join Techniques for Object-Oriented Databases.* Sencond International Conference on Parallel and Distributed Information System, San Diego, California, 1993.

[Loh88] G. M. Lohman. *Grammar-Like Functional Rules for Representing Query Optimization Alternatives*. ACM SIGMOD International Conference on Management of Data, Chicago, 1988.

[MC97] J. C. Machado, C. Collet. *A Parallel Execution Model for Databases Transactions*. Proceedings of the Fifth International Conference on Database Systems for Advanced Applications, Melbourne, Austrália, 1997.

[MD97] M. Mehta, D. J. DeWitt. *Data placement in shared-nothing parallel database systems*. VLDB Journal: Very Large Data Bases, volume 6, número 1, páginas 53-72, 1997.

[OMG91] OMG. *The Commom Object Request Broker: Architecture and Specification*. Object Management Group and X/Open, 1991.

[OHE96] R. Orfali, D. Harkey, J. Edwards. *The Essential Distributed Objects Survival Guide*. Wiley, 1996.

[OV99] M. T. Özsu, P. Valduriez. *Principles of Distributed Databases Systems*. Second Edition, Prentice Hall, 1999.

[PM01] C. G. Pires, J. C. Machado. *Applying Rules for Partitioned Parallelism in OODBMS within an Optimizer Generator Framework.* XVI Simpósio Brasileiro de Banco de Dados, Sociedade Braileira de Computação, Rio de Janeiro, 2001.

[RM93] E. Rahm, R. Marek. *Analysis Of Dynamic Load Balancing Strategies For Parallel Shared Nothing Systems*. Proceedings of the 19th Very Large DataBases Conference, Dublin, Irlanda, 1993.

[SAC79] P. Selinger, M. Astahan, D. Chamberlin, R. Lorie e T. Price. *Access Path Selection In A Relational Database Management System*. Proceedings of SIGMOD, Boston, páginas 82-93, Maio, 1979.

[Sam98] S. F. M. Sampaio. *Extração De Paralelismo No Processamento De Consultas Declarativas Em SGBOO*. Dissertação de Mestrado, Universidade Federal do Ceará, Departamento de Computação, 1998

[SMB01] L. Shapiro, D. Maier, P. Benninghoff, K. Billings, Y. Fan, K. Hatwal, Q. Wang, Y. Zhang, H. Wu, B. Vance. Exploiting Upper and Lower Bounds In Top-Down Query Optimization, IDEAS 2001, Grenoble, França, *2001*.

[Ste95] H. J. Steenhagen. *Optimization Of Object Query Languages*. PhD thesis, Universiteit Twente, 1995

[SZ90] G.M Shaw, S. B. Zdonik. *A Query Algebra For Object-Oriented Databases*. Proceeding of the 6ht International Conference on Data Engenniering, IEEE, Los Angeles, CA, Fevereiro, 1990.

[TN92] M. Tsangaris, J. F. Naughton. *On the performance of object clustering techniques*. ACM SIGMOD Int. Conf. on Management of Data, páginas 144-153, San Diego, CA, junho 1992.

[Van93] S. Vandenberg. *Algebras for Object-Oriented Query Languages*. PhD thesis, U. Wisconsin, 1993.

[W3C] World Wide Web Consortium. *Extensible Markup Language (XML)*. http://www.w3.org/XML/.

[XH94] Z. Xie, J. Han. *Join Index Hierarchies For Supporting Efficient Navigations In Object-Oriented Databases*. Proceedings of the 20th Very Large DataBases Conference, Santiago, Chile, 1994.

[YM98] C. T. Yu, W. Meng. *Principles Of Database Query Processing For Advanced Applications*. Morgan Kaufmann Publishers, 1998.