Fábio André Machado Porto

Estratégias para a execução paralela de consultas em bancos de dados científicos distribuídos

Tese de Doutor em Informática: Ciência da Computação

Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 16 de Abril de 2001.

Fábio André Machado Porto

Estratégias para a execução paralela de consultas em bancos de dados científicos distribuídos

Tese apresentada ao Departamento de Informática da PUC-Rio, como parte dos requisitos para a obtenção do título de Doutor em Informática: Ciência da Computação.

Orientador: Prof. Rubens Nascimento Melo

Co-orientador: Prof. Patrick Valduriez

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 16 de Abril de 2001.

"A Patrícia, Tiago e Daniel, por serem a razão maior da minha vida e motivo principal daquilo que faço. Agradeço pela compreensão com que compartilharam sua vidas e pelo amor que sempre me dedicaram durante os bons e difíceis momentos. "

“ Aos meus pais, João André e Edmê, por terem sempre me incentivado e apoiado com amor e compreensão. O exemplo que recebi deles é parte do que sou hoje. Espero que esta tese sirva como conquista do esforço na minha formação e como resultado de um grande exemplo. ”

“ Aos meus irmãos Dionísio, Márcio *in memorium*, Cássio e Filipe, por serem parte de mim e por terem compartilhado os momentos mais importantes de minha vida. ”

Agradecimentos

Em um trabalho tão árduo e longo repleto de percalços inimagináveis e caminhos quase desconhecidos, somente com o apoio daqueles que nos emprestam um pouco de suas vidas podemos chegar ao fim com êxito. Gostaria de nessa página lembrar e agradecer a todos vocês que caminharam comigo: ensinando, aconselhando, ajudando , ouvindo, direcionando, amando, surpreendendo, guiando, discutindo, clareando, criticando, parabenizando, convivendo... A lista que se segue registra apenas uma lembrança mais forte, possivelmente influenciada pelo dia em que escrevo estas palavras. Certamente não comportará todos aos quais me referir anteriormente e para quem deixo minha profunda gratidão.

Ao Prof. Rubens Nascimento Melo por ter me acolhido como aluno ainda durante o curso de mestrado e ter me enriquecido com conhecimento e sabedoria durante estes tantos anos de convívio.

Ao Prof. Patrick Valduriez por ter me recebido durante o período de sanduíche na França. Sua constante preocupação com o meu bem estar e o de minha família revelou uma pessoa admirável por suas qualidades como professor, orientador e chefe. A ele devo, em grande parte, o sucesso deste trabalho.

Aos Profs. Luc Bouganim e Françoise Fabret pela paciência em me ensinar os mistérios da pesquisa em avaliação de consultas. A dedicação deles ao meu trabalho foi essencial para chegarmos a este resultado. Meus mais profundos agradecimento.

Ao Prof. Sérgio Carvalho *in memorium* pelo apoio sincero e amigo de um mestre quase pai. Sua prematura partida deixa orfãos de uma forma única de incentivo e direcionamento. Queira Deus que eu poça, ao menos um pouco, repetir com meus alunos o que fez conosco.

Ao meu amigo Alberto Lerner que comigo dividiu as angustias e felicidades do início de nossas vidas como pesquisadores. Esperando que o *peso* daquele começo se traduza em uma amizade e cooperação duradouras.

Ao meu amigo Álvaro Barbosa que me abriu as portas em meu retorno dando o privilégio de contribuir com seu trabalho e participar de sua amizade.

Aos colegas de pesquisa Khaled Yagoub, Helena Galhardas e Luiz Rivera que me deram o prazer de compartilhar diferentes momentos desta jornada.

A todos os demais colegas da PUC-Rio e do Projeto Caravel com quem compartilhei essa experiência.

A Elisabeth Baqué que sempre esteve pronta a ajudar nos inúmeros problemas que tivemos que enfrentar em nossa estadia no exterior.

A CAPES pelo apoio financeiro ao meu curso de doutorado sanduíche.

A Sandra Abreu e demais colegas do P-15 pelo apoio incondicional em todos os momentos que precisei.

Aos Professores e funcionários da PUC-Rio que me auxiliaram ao longo deste período.

Resumo

Este trabalho foi concebido a partir da arquitetura de integração proposta pelo sistema Le Select que é um *middleware* de integração desenvolvido para apoiar a publicação de recursos científicos.

O objetivo do trabalho é propor uma estratégia de execução de consultas científicas distribuídas que minimize o tempo total de execução e maximize a utilização de recursos.

Propõe-se uma estratégia original para geração de planos paralelos de execução para consultas científicas. Assumida a premissa de que a execução de programas científicos apresenta um custo por tupla extremamente superior ao de operações relacionais tradicionais, adota-se uma estratégia que privilegia a redução no número de chamadas aos programas. Para isso, é proposto um conjunto de técnicas de execução. O resultado da aplicação das técnicas flexibiliza o posicionamento de programas, no plano de execução de consultas. O que viabiliza heurísticas para geração de planos paralelos; bem como a estratégia híbrida, para a avaliação de predicados caros, que combina os modelos de paralelismo independente e de fluxo de dados. Essa estratégia mostrou-se mais adequada do que as propostas na literatura, frente às variações, em tempo de execução, de parâmetros estimados estaticamente.

Como alternativa à estratégia híbrida, propõe-se a abordagem *Picking Cherry* para execução de predicados caros, baseada na exploração de dependências funcionais entre valores de atributos de entradas de programas.

Abstract

This work was conceived based on the data integration architecture proposed by the Le Select system which is an integration middleware developed to support cientific applications.

The goal of this work is to propose an execution strategy for distributed scientific queries that minimize their elapsed-time and maximize resource utilization.

As a result of this research, we propose an original strategy for the generation of parallel execution plans. Considering the assumption that the execution of scientific programs present a per tuple cost extremely higher than that of traditional relational operations, we adopt a strategy that aims into reducing the number of programs invocation during the query execution. To that end we conceive some special execution techniques. The result of applying such techniques are reflected in the query execution plan where the placement of the expensive program invocation is flexibilized, making it possible to adopt heuristics for parallel query plans generation, as well as the hybrid strategy for expensive predicates. This strategy combines pipelined and independent inter-operator parallelism providing an execution profile that is adpated to variations of estimated parameters during query execution.

As an alternative to the hybrid strategy, we propose the Picking Cherry approach for the execution of expensive predicates based on the analysis of input data dependency.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

1.1 Motivação

Após duas décadas de intenso desenvolvimento tecnológico, sistemas de gerência de bancos de dados relacionais (SGBDR) apresentam tecnologia estável e eficiente para o armazenamento e acesso a dados convencionais. Seu grande sucesso é, em parte, devido ao isolamento entre os dados e suas aplicações proporcionado pelo sistema. Além de tornar-se independente de variações nos detalhes de armazenamento, aplicações beneficiam-se de uma linguagem declarativa para acesso e atualização de seus dados. Essas características combinadas a procedimentos de segurança confiáveis fez de SGBDRs uma escolha incontestável para aplicações empresariais.

Nos últimos anos, alguns domínios de aplicação, como o de aplicações científicas, que até então processavam informações de maneira rudimentar, desenvolveram-se no uso da informática, fomentados pelo aumento de capacidade nos computadores, barateamento de custo das máquinas, disponibilidade de acesso remoto via rede, e também devido a facilidades trazidas por novas técnicas de desenvolvimento de *software*. Em pouco tempo o volume de dados processado por essas aplicações cresceu ao ponto de não condizerem com procedimentos manuais de gerência. O processo que já havia ocorrido com sistemas empresariais, na década de 80, iniciou-se para essas novas aplicações carentes de sistemas que as ofereça visões de alto nível sobre dados e programas a serem acessados.

Com esta meta, o grupo de pesquisa em banco de dados da PUC-Rio - TecBD vem, desde 1997, pesquisando técnicas para flexibilizar a construção de SGBDR [MPLB98]. Em 1999, assinou-se um convênio com o INRIA, França, e outras instituições acadêmicas brasileiras, com o objetivo de desenvolvimento de tecnologia de banco de dados para suporte à aplicações ambientais.

Aplicações ambientais surgem como um bom exemplo de novos requisitos à SGBDRs. Do ponto de vista de dados, as informações analisadas combinam estruturas de dados simples como cidade, região, dados sócio-econômicos, com objetos complexos: matrizes de resultados experimentais, dados meteorológicos, imagens de satélite etc. Do ponto de vista arquitetural, a localização de grupos de pesquisa em diversas regiões requer suporte sistêmico que torne viável o compartilhamento de dados e programas distribuídos.

Tem-se nesse exemplo um excelente material de pesquisa, compreendendo heterogeneidade e distribuição de dados e programas.

1.1.1 Integração do aspecto comportamental de aplicações ambientais em BDs relacionais

Este trabalho sintetiza a pesquisa em alguns problemas identificados em aplicações ambientais. Inicialmente, analisaram-se problemas de integração do modelo Orientado a Objetos (OO) com sistemas de banco de dados relacionais em aplicações ambientais.

Em um primeiro trabalho nesta área, propoem-se técnicas para integração comportamental dos modelos. Esse problema foi motivado pela verificação da importância da observação dos estados de alguns objetos ambientais, para tomadas de decisão. A aplicação, da Secretaria de Meio-Ambiente do Rio de Janeiro, gerencia os índices de poluentes eliminados por indústrias, na Baía de Guanabara, e correlaciona-os com dados sobre marés e informações meteorológicas. A combinação desses parâmetros, identificado como Índice de Qualidade da Baía da Guanabara (IQBG), determina ações a serem tomadas por diversos setores da Secretaria. As variações do IQBG, e as respectivas ações a serem tomadas, foram modeladas através de um Diagrama de Transição de Estados (DTE) [Har88]. Transições no DTE representam a mudança do IQBG para um intervalo onde determinadas ações devem ser tomadas. Essas, por sua vez, correspondem a pós-condições no DTE, enquanto cenários como fortes ventos, que podem reduzir efeitos de derramamento de poluentes na Baía, são modelados como pré- condições, inibindo transições para um estado mais grave.

A proposta, apresentada no artigo [PVsC98], modela o armazenamento dos estados do DTE como tabelas em um banco de dados Relacional. Adicionalmente, eventos motivadores de transições de estados de objetos, como a detecção de derramamento de poluentes na Baía, são representados por *procedimentos armazenados* no banco de dados. Ao serem acionados, esses procedimentos efetuam as transições entre os estados, modelados como tabelas. Gatilhos [1], associados às tabelas de estados, implementam verificações e ações, representadas no DTE, por pré e pós-condições, respectivamente.

O objetivo do trabalho foi estender a integração entre aplicações OO e bancos de dados relacionais, valendo-se de mecanismos ativos de banco de dados, para representação comportamental de objetos da aplicação.

Um segundo trabalho ampliou a integração entre os modelos OO e relacional, com o tratamento de sincronismo entre as visões de dados nos dois modelos. Retornando-se à aplicação ambiental, descrita anteriormente, verifica-se que sua implementação compreende dois módulos: a aplicação OO e o SGBD relacional. O primeiro trata do processamento não persistente dos objetos, enquanto o segundo, implementa persistência, além de aspectos do comportamento dinâmico de objetos. Dessa forma, têm-se duas visões dos dados: uma Orientada a Objetos, implementada pela linguagem de programação, e outra Relacional. Tais visões precisam ser consistentemente mantidas. Porém, atualizações no banco de dados efetuadas por procedimentos ativos (como transições de estado) e correspondentes (pós-condições) modificam o estado de tabelas, tornando suas visões Orientadas a Objetos inconsistentes.

O trabalho apresentado em [PCVsM99] propõe uma arquitetura para manutenção do sincronismo entre as visões Orientadas a Objetos, processadas por programas em uma linguagem de programação OO, e suas correspondentes visões relacionais, persistidas em SGBDs relacionais ativos.

[1] triggers

1.1.2 Otimização de consultas científicas distribuídas

Duas outras aplicações ambientais, os projetos Thetis [Thet] e DECAIR [Deca], motivaram a pesquisa central deste trabalho. O autor participou desses projetos [2] como membro do grupo Caravel, do *Institute National de Recherche en Informatic et Automation (INRIA-França).* O intercâmbio com o projeto Caravel é parte da cooperação estipulada no convênio ECOBASE, financiado pelo CNPq, entre a PUC-Rio, a UNI-Rio, a COPPE e o próprio INRIA.

O trabalho desenvolvido encontra sua motivação inicial na integração de dados e programas via Internet para apoio ao trabalho de equipes de cientistas ambientais. Unidos por projetos de pesquisa regionais ou globais, cientistas almejam compartilhar dados amostrais e imagens (imagens de satélite, índices de poluição, tabelas de temperatura, entre outros) capturados por sensores e satélites. O compartilhamento de programas é também muito relevante, pois homogeniza procedimentos científicos além de reduzir custos de pesquisa. Infelizmente, a execução remota desses programas não é simples. Em sua maioria, são escritos em linguagem de 3^a geração, sem suporte nativo a ativação remota, além de, frequentemente, requisitarem bibliotecas específicas, o que dificulta sua execução em um outro sítio. Por outro lado, sua execução local compartilhada é igualmente complexa devido ao alto custo de localização, extração e uso.

A solução vislumbrada é permitir que cientistas publiquem seus recursos (dados e programas) para a comunidade através de um sistema mediador comum [Wied92]. Publicar um recurso é registrá-lo no sistema mediador, facilitando seu grau de acessibilidade. Assim, usuários de recursos publicados são capazes de combinar, de forma transparente, recursos locais e recursos remotos.

Le Select [LeSe] é um sistema mediador desenvolvido no Inria que suporta a integração de dados e programas disponibilizados na Internet. Sua arquitetura permite a configuração de servidores Le Select em cada um dos locais que sediem uma equipe de cientistas. Cada sítio participante tem um servidor Le Select completo, capaz de publicar e acessar recursos, bem como de processar consultas escritas na linguagem SQL.

Tendo como base a estratégia de integração de dados e programas distribuídos

[2] financiado pela CAPES - MEC, Brasília Brasil

via um sistema de mediação como o Le Select, este trabalho propõe estratégias para execução otimizada de consultas de integração, sobre dados e programas científicos publicados na Internet. Considera-se como premissas a possibilidade das consultas requererem a transferência de objetos complexos longos (Blobs) pela rede (como execução de programas que processam imagens) e o fato dos programas científicos apresentarem um custo de execução, medido em tempo por tupla, significativamente maior que o tempo de execução de operadores relacionais tradicionais (i.e. junção, seleção e projeção).

1.2 Formulação do problema

1.2.1 Um cenário típico do problema

Nesta seção, apresenta-se um cenário de aplicação ambiental que motivará os problemas abordados.

Considere-se uma aplicação ambiental que tenha por objetivo correlacionar medidas mensais de poluição com mapas regionais de umidade, além de outras informações ambientais, obtidas por equipes de cientistas do Estado do Rio de Janeiro. Os mapas de umidade são capturados por satélites, na forma de imagens. Uma vez processadas, essas imagens possibilitam a identificação, em diversas cores, de índices de umidade do solo em uma dada região. As medidas de poluição são armazenadas em uma matriz de 7 índices, capturados três vezes ao dia por sensores. Esses dados armazenam-se em dois arquivos: *Precipitação por Região* (PrecReg) e *Poluição por Região* (PolReg), além do arquivo *Informação Ambiental* (Env) que armazena dados diversos sobre características da região. A aplicação conta, adicionalmente, com a colaboração de pesquisadores franceses e de um grupo de São Paulo, que disponibilizam dois programas para o tratamento de dados ambientais. O primeiro deles, o programa *Índice de Precipitação*, calcula o valor do respectivo índice baseado em uma imagem *raster* de satélite. Este programa, disponibilizado por uma equipe de cientistas localizada em Paris, foi escrito em FORTRAN. O segundo programa foi desenvolvido pela equipe paulista para produzir um índice de poluição a partir de matriz de valores amostrais. O programa foi escrito em C e conta com bibliotecas desenvolvidas localmente.

De maneira a integrar dados e programas, as equipes de cientistas do Rio, São

Paulo, Paris e Brasília utilizam o sistema de mediadores Le Select. Cada sítio conta com uma cópia do sistema, formando uma rede *peer-to-peer*. A integração propiciada pelo sistema possibilita que cientistas obtenham resultados da aplicação de programas sobre os dados, disponibilizados remotamente, a partir de consultas SQL.

Como exemplo de uma consulta, considere-se a necessidade de se identificar regiões onde índices de poluição correlacionam-se, segundo um dado critério, ao fator de umidade local. Os arquivos estão publicados no sistema Le Select via adaptadores de dados, construídos pelas respectivas equipes de pesquisa. O esquema exportado dos dados encontra-se descrito a seguir:

Exemplo 1.2.1
Mapa umidade - PrecReg (nome: string, prec: blob)
Indices Poluição - PolReg (regnome: string, mês: int, medidas: blob)
Info Ambiental - Env (região: string, tipo-solo: int, vegetação: int, ecosistema: int)

Os dois programas científicos, Prec *e* Pol, *são publicados por meio de adaptadores de função e apresentam as seguintes assinaturas:*

Prec ((mapHumidity: blob);(indPrec: float));
Pol ((matrix: blob);(polInd: float));

Com base no esquema acima, um pesquisador da equipe localizada em Brasília formularia, ao sistema Le Select, a seguinte consulta:

Select *pr.nome,e.tipo-solo,e.ecosistema*
    From *PrecReg pr, PolReg po, Env e*
    Where *pr.nome=po.regnome and*
        pr.nome= e.região and
        Prec(pr.prec) > 1.5 and
        Pol(po.medidas) < 0.6

□

Este trabalho trata do problema de otimização de execução de consultas científicas como a expressa no exemplo 1.2.1, tendo como objetivo o desenvolvimento de técnicas, estratégias e algoritmos que minimizem o tempo de resposta a consultas desse tipo. Para alcançar seu objetivo, este trabalho concentra-se em quatro metas principais: reduzir o número de invocações a programas, reduzir o volume de bytes transferidos pela rede - resultantes da transferência de *Blob*s, minimizar o impacto de variações no fluxo de dados em tempo de execução e maximizar o paralelismo na avaliação de predicados caros.

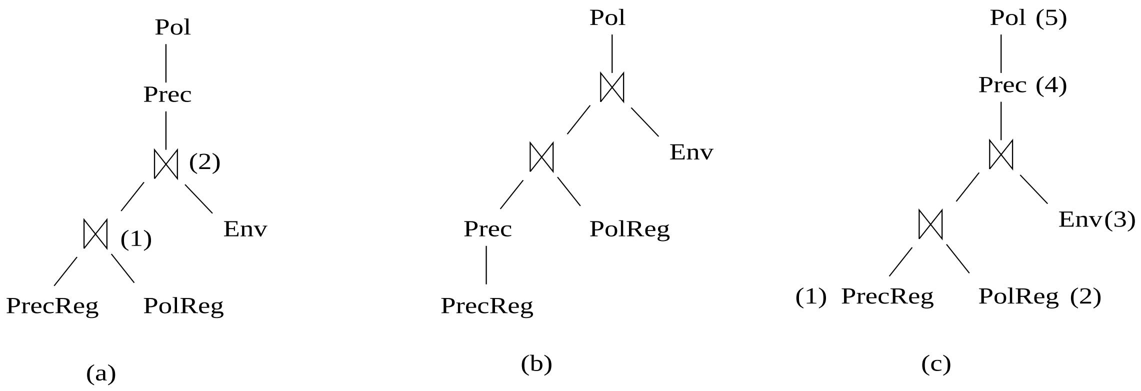


Figura 1.1: *Planos de execução de consultas distribuídas*

As quatro metas definidas no parágrafo anterior podem ser refinadas nos seguintes sub-problemas:

- Eliminar invocação repetitiva a programas científicos

  Suponha-se que o plano de execução escolhido para a consulta em 1.2.1 seja o representado na figura 1.2.1(a). A junção *1* associa uma tupla de *PrecReg* eventualmente com várias de PolReg (i.e. amostras de vários meses de uma mesma região). Seguindo o plano de execução, o programa *Prec* é executado repetidamente sobre o mesmo *Blob*, duplicado nas tuplas da relação resultante da junção em *1*. Em um outro exemplo, o próprio arquivo de entrada (i.e. PrecReg ou PolReg) pode apresentar o mesmo *Blob* em diferentes tuplas, induzindo a repetidas invocações de programas sobre a mesma entrada.

- Eliminar invocações desnecessárias de programas científicos

  Considere-se o plano de execução representado na figura 1.2.1(b). O programa *Prec* é executado sobre todas as imagens contidas no arquivo *PrecReg*. Considere-se também que algumas das regiões em *PrecReg* não tiveram amostras de poluição coletadas e, consequentemente, não aparecem no arquivo *PolReg*. Neste cenário, o programa *PrecReg* estará sendo desnecessariamente invocado sobre *Blob*s de tuplas que teriam sido eliminadas caso a junção entre *PrecReg* e *PolReg* houvesse precedido à invocação do programa.

- Minimizar, durante a execução, o impacto de variações no fluxo de dados

  Em um plano de execução como o da figura 1.2.1(a), onde *Prec* e *Pol* estão localizados em sítios distintos, a ocorrência de variações em tempo de execução em um dos sítios pode levar a grandes desbalanceamentos de carga. Alguns eventos motivadores de variações de fluxo de dados seriam: alteração no fator de seletividade previsto, ocorrência de valores distintos nas entradas para os programas, alterações nas condições do ambiente de execução, etc. Como exemplo, veja-se novamente o plano mostrado na figura 1.2.1(a) e a relação intermediária resultante da junção *2* apresentando cardinalidade igual a **100**. Suponha-se ainda que o tempo médio de execução, por invocação dos programas *Prec* e *Pol*, seja, respectivamente, **1** e **5** segundos. Caso a execução do predicado P_1 resulte em falso para as 95 primeiras tuplas da relação intermediária, o processamento de *Prec* e *Pol* se daria quase que de forma sequencial. Em outras palavras, durante os 95 seg. iniciais o programa *Pol* não seria executado. A partir da 96^a tupla, *Pol* começaria a ser executado, enquanto sua fila ficaria preenchida com as 4 últimas tuplas processadas por *Prec*, para as quais o predicado teria sido avaliado verdadeiro. Nesta etapa final *Prec* teria terminado sua tarefa e se manteria inativo.

- Eliminar transferência repetitiva de *Blob*s pela rede

  O cenário descrito no primeiro item acima é útil também para apresentação deste sub-problema. Uma vez que a junção em *1* produza tuplas duplicadas na relação *PrecReg*, por exemplo, os Blobs que façam parte do atributo *prec* serão, correspondentemente, duplicados. O resultado dessa duplicação, é a transferência repetida para o sítio da relação *Env*, e, posteriormente, ao sítio de *Prec*, onde, finalmente, seriam processados.

- Eliminar transferência desnecessária de *Blob*s pela rede

  Tome-se o plano de execução em 1.2.1(c). As tuplas da tabela *PolReg* são enviadas para os sítios *2, 3* e *4* antes de chegarem ao sítio onde se encontra o programa *Pol*. Os *Blob*s referentes ao atributo *medidas* teriam realizado uma enorme e custosa trajetória desnecessariamente, uma vez que apenas são úteis no sítio final. Adicionalmente, tuplas podem ser excluídas por junções anteriores à invocação do programa, fazendo com que os respectivos *Blob*s não precisem ser transferidos.

Conforme exemplificado nos cenários explorados acima, execuções ingênuas de aplicações científicas distribuídas, produzem trabalho desnecessário. Considerando-se o alto custo de execução de programas científicos, que pode variar, entre alguns segundos, a até horas de processamento, tais custos adicionais podem inviabilizar o trabalho integrado, via Internet, de grupos remotos. É a partir dessa constatação verificada pelas equipes dos projetos Thetis e Decair, durante o uso inicial do sistema Le Select, que este trabalho desenvolve técnicas para melhor integrar aplicações científicas, ao contexto de execução de consultas distribuídas de integração.

1.2.2 Resumo da proposta

Neste trabalho propõe-se uma estratégia para execução eficiente de consultas científicas distribuídas. Inicialmente, integra-se a invocação de programas ao processo de avaliação de consultas por meio de dois operadores *Apply* e *GetBlob*. Um conjunto de técnicas de execução implementa esses operadores com o objetivo de reduzir o número de chamadas de programas e o custo de transferência de objetos longos, (ex. imagens) pela rede.

Assume-se como premissa que o custo de programas científicos é significativamente superior ao custo de operações relacionais tradicionais. Com isso adota-se a heurística *Pull-Up* para o posicionamento de programas no plano de execução de consultas. Tal heurística propõe a alocação de programas, no plano, o mais tardiamente possível, de maneira a beneficiar-se de reduções realizadas por predicados simples (junção e seleção).

Tendo-se adotado a heurística *Pull-Up*, deriva-se uma estratégia para geração de planos de execução de consultas distribuídas onde predicados simples são otimizados segundo estratégia tradicional [SA80].Para execução de programas científicos produz-se um plano com paralelismo inter-operador segundo a relação de dependência de dados entre os programas.

Finalmente, para a execução de predicados caros, uma proposta *híbrida* para geração de planos paralelos, combina os modelos de paralelismo independente e de fluxo de dados. Essa estratégia se mostrou eficiente, frente às encontradas na literatura, no tratamento de variações, durante a execução, de previsões realizadas estaticamente. Adicionalmente, apresenta-se uma abordagem para execução de predicados caros explorando-se as dependências funcionais entre atributos de entrada a programas científicos.

1.3 Contribuições

Neste trabalho analisa-se o problema de otimização e execução de consultas distribuídas, que incluam invocação de programas científicos, em sistemas de mediadores, com arquitetura distribuída, e modelo relacional de integração. As principais contribuições deste trabalho encontram-se listadas abaixo e foram resumidamente descritas em [BFPV01]:

- Inicialmente, integra-se programas científicos, do tipo tupla-tupla, ao processo de avaliação de consultas. São especificados dois operadores, *Apply* e *GetBlob* responsáveis pelas operações de execução de programas e captura de dados do tipo Blob, respectivamente. O par de operadores estende, formalmente, o conjunto de operadores da álgebra relacional.
- Em seguida, reduz-se o custo de execução de programas científicos, em consultas, eliminando-se custos adicionais, produzidos por estratégias ingênuas de execução. Nesse aspecto, integram-se ao processo de execução de consultas, as técnicas de *transferência postergada*, *caching* e *transferência em segundo plano.*
- Adicionalmente, adota-se a heurística *Pull-Up*, para alocação da operação *Apply*, no plano de execução de consultas(PEC). A estratégia posiciona o operador no PEC, sucedendo a todos os predicados simples, de modo a beneficiar-se das reduções efetuadas, o que minimiza o número de chamadas à programas científicos.
- Um dos resultados obtidos na validação experimental das técnicas de execução para avaliação de consultas científicas aponta que seu custo é minimizado quando se aplicam todas as técnicas, em conjunto, independentemente do plano de

execução gerado. O emprego, em conjunto das técnicas, reduz o problema de avaliação de consultas científicas, a um processo de avaliação de consultas, segundo estratégia convencional, seguido de tratamento específico para a execução distribuída de programas.

- A execução otimizada de programas distribuídos é proporcionada por planos que combinam estratégias de paralelismo independente e de fluxo de dados, com o objetivo de reduzir o tempo total da consulta, maximizando o uso de recursos distribuídos.

- Para consultas com mais de um predicado caro, desenvolveu-se a estratégia híbrida de execução. A estratégia produz uma árvore de execução com ramos independentes, compostos de cadeias de execução de predicados caros, segundo estratégia paralela de fluxo de dados. A estratégia reduz o efeito, durante a execução, de variações nas condições previstas estaticamente.

- Finalmente, propõe-se a abordagem de execução de predicados caros, baseada na exploração de dependências funcionais entre valores de atributos em entradas de predicados caros. Uma estratégia de execução, adaptada ao modelo paralelo independente, é proposta, além de uma estratégia de execução gulosa, baseado no cálculo do *fanout* entre valores.

Todas as propostas apresentadas foram validadas através de experimentação e simulação. Os resultados confirmam a intuição de que a combinação das técnicas de execução com as estratégias e algoritmos propostos produzem planos de execução de consultas científicas distribuídas mais eficientes que os popostos na literatura.

1.4 Estrutura do Documento

Este trabalho está estruturado em sete capítulos. O capítulo 2 prepara o leitor para a compreensão do trabalho. Inicialmente apresentam-se sistemas de bancos de dados heterogêneos, nas suas formas, fortemente e fracamente acoplados. Dentre esses, os sistemas de mediadores, fracamente acoplados, serviram de base para este trabalho, e são discutidos na seqüência. Uma vez que se tenha apresentado a arquitetura distribuída de sistemas de mediadores, discute-se o processo de avaliação de consultas, incluindo as estratégias paralelas independentes e de fluxo de dados. Finalmente, o capítulo apresenta um modelo para representação de programas em consultas e os principais trabalhos que tratam de otimização de consultas com predicados caros.

No capítulo 3, apresenta-se o sistema de mediadores Le Select. Este trabalho teve como objetivo propor técnicas, implementáveis no sistema Le Select, que garantissem o processamento ótimo de consultas distribuídas, envolvendo a execução de programas científicos. Discute-se a integração de dados e serviços ao sistema, além de uma proposta para execução de consultas com apenas um programa científico. Finalmente, quanto ao processo de otimização de consultas, apresenta-se o tratamento dado à execução de funções, além da localização de funções e tabelas, no plano de execução distribuído.

O quarto e quinto capítulos propõem a integração de programas científicos em consultas a sistemas de BD distribuídos. No capítulo 4, estende-se a álgebra relacional com operadores que implementam o tratamento de execução de programas. Utilizando-se esses operadores, concebem-se técnicas de execução que minimizam o número de chamadas de programas e de transferência de dados. O capítulo cinco explora técnicas de paralelismo para geração de planos distribuídos de execução que minimizem o tempo de consulta e maximizam o uso de recursos.

Tendo apresentado as propostas desenvolvidas neste trabalho, o sexto capítulo demonstra a eficácia segundo validação experimental. Um *benchmark* informal direciona a validação das técnicas de execução. Adicionalmente, é implementado um simulador de execução que possibilita explorar 200.000 cenários de avaliação da estratégia híbrida, frente as estratégias baseadas em paralelismo de fluxo de dados. Por último, verifica-se a valia da abordagem de execução baseada na dependência funcional entre atributos, chamada de *Picking Cherry*, comparando-a com as estratégias híbrida e ingênua.

Finalmente, no capítulo 7, são apresentadas as conclusões do trabalho e as perspectivas de trabalhos futuros.

Capítulo 2

Trabalhos relacionados

2.1 Introdução

A integração de aplicações científicas por sistemas de mediadores, como o Le Select, recorre a problemas tratados na área de banco de dados heterogêneos. Sistemas de bancos de dados são qualificados de heterogêneos quando seus componentes apresentam serviços ou modelos de bancos de dados, implementados diferentemente. Assim, um sistema que integre *softwares* de banco de dados, desenvolvidos por diferentes fabricantes; modelos de dados distintos; formatos e expressões de dados, segundo visões particulares, é considerado um sistema de banco de dados heterogêneo. Em outras palavras, toda e qualquer discrepância, entre dados, ou sistemas de acesso a dados, que suscite um tratamento específico para integração de dados e serviços, torna o sistema heterogêneo. Por oposição, sistemas de integração que não apresentam diferenças entre seus componentes são ditos homogêneos.

Nesse sentido, a integração de aplicações científicas, para grupos de pesquisadores distribuídos pela Internet, caracteriza-se como uma aplicação heterogênea. Essas aplicações integram dados de tipos não convencionais, como imagens e matrizes de dados amostrais, e serviços, na forma de programas científicos, a dados convencionais, como temperatura, pressão, e dados sócio econômicos.

Sistemas de integração de dados heterogêneos são classificados em dois grupos: sistemas fortemente acoplados e sistemas fracamente acoplados [SL90]. As seções seguintes apresentam estes dois modelos.

2.1.1 Sistema fortemente acoplados

Um sistema federado [SL90] controla e coordena a manipulação de uma coleção de sistemas de gerência de base de dados autônomos. Sistemas são considerados autônomos, uma vez que continuam a ser utilizados por aplicações desenvolvidas localmente. Um sistema federado integra as bases de dados locais, em um ou vários sistemas federados, e recebe consultas e atualização sobre esse esquema. Os objetivos de sistemas federados, segundo [K95] são:

- eliminar a necessidade de migrar dados de uma base para outra;
- respeitar a autonomia de base de dados locais. O sistema federado acessa as bases locais com a mesma prioridade que sistemas locais;
- permitir manipulação uniforme dos dados integrados, a partir de uma linguagem comum;
- mascarar a heterogeneidade dos ambientes locais de cada base de dados (ex. máquina, redes);
- suportar transações contendo operações de leitura e atualização sobre o conjunto de dados da federação;
- fornecer todos os serviços de um SGBD: definição de um esquema, gerência de transações, avaliação de consultas, interface interativa e de aplicação, além de utilitários de administração;
- oferecer desempenho que o aproxime aos encontrados em sistemas de bancos de dados distribuídos e homogêneos.

Sistemas fortemente acoplados são adequados à integração de sistemas que formam um domínio, relativamente pequeno, como a integração de sistemas de uma mesma empresa.

2.1.2 Sistema fracamente acoplado

Os sistemas fracamente acoplados são concebidos para atender ao problema de integração em grande escala, como ocorre com fontes de dados distribuídos na internet.

De modo a facilitar a participação de uma nova fonte de dados, os sistemas adotam a arquitetura de sistemas de mediadores [Wied92], conforme mostrado na figura 2.1.

Contrariamente à sistemas fortemente acoplados, o esquema global aparece como um entrave à evolução de sistemas de mediação [TRV98]. Dessa forma, esses sistemas possibilitam que fontes de dados sejam acessadas, sem requerer um esquema global de integração.

A arquitetura de sistemas mediadores pode ser particionada em três camadas: fontes de dados, adaptador e mediador, ver figura 2.1. A camada de *fonte de dados* representa a base de dados local a ser acessada. Essa pode ser disponibilizada por um SGBD relacional completo ou ser apenas um conjunto de arquivos em um sistema de arquivos. Cada *fonte de dados* é feita disponível ao mediador por meio de um adaptador. Adaptadores são programas especificamente desenhados para cada tipo de *fonte de dados* e configurados para exportação, e integração, do modelo de dados local da *fonte de dados*, segundo o modelo global de integração do mediador. O processo de exportação inclue o tratamento para homogeneização de dados, segundo semânticas globais. Não há um consenso, na literatura, sobre os limites de funcionalidade de adaptadores [ÖV99], que podem ser simples roteadores de pedidos SQL a SGBDs relacionais, a até poderosas máquinas de execução, sobre fontes sem capacidade relacionais, como sítios *Web*. Finalmente, a camada de *mediador* oferece uma interface acessível por aplicações clientes, além de oferecer componentes para o processamento de consultas de integração.

A arquitetura em três camadas proporciona grande flexibilidade, uma vez que o sistema central (i.e. camada mediador) não requer alteração para que seja incluído um novo tipo de *fonte de dados*, cujas particularidades são tratadas pelo adaptador. Esse é concebido e construído de forma independente, bastando atender ao contrato estipulado na interface entre ele e o mediador. Alguns exemplos de sistemas mediadores são: Garlic [CHS+95], Disco[TRV98], TSIMMIS [GPQ+97], MOCHA [MR00] e LeSelect [LeSe].

À luz dessas definições, este trabalho esta baseado na arquitetura de sistemas de mediadores. A integração de dados e programas científicos, publicados por grupos de pesquisadores, é realizada pelo sistema mediador Le Select. O sistema adota o modelo Relacional, como modelo de integração. Dessa forma, dados e programas são processados por adaptadores, que os transformam para uma visão Relacional.

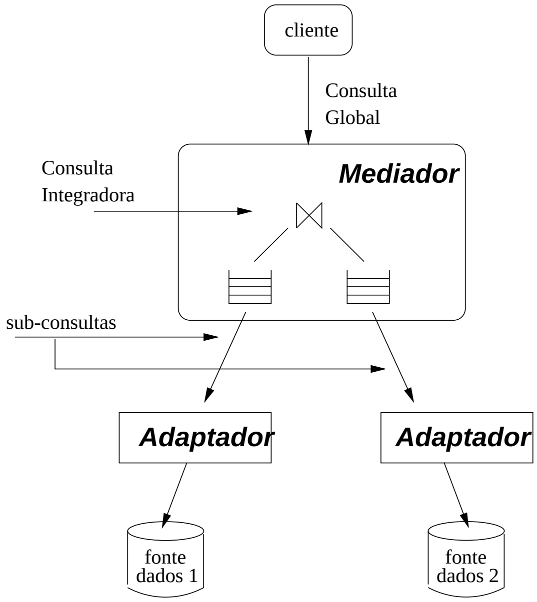


Figura 2.1: *Execução de consultas em Mediadores*

O acesso ao sistema mediador processa-se pela submissão de consultas em linguagem declarativa (SQL), que combinam acesso a dados e chamadas a programas. A avaliação dessas consultas, pelo sistema mediador, combina técnicas tradicionais, e específicas à sistemas distribuídos na internet.

Na seção seguinte, apresenta-se uma visão geral do processo de avaliação de consultas distribuídas, necessárias à compreensão do restante do trabalho.

2.2 Visão geral do processo de avaliação de consultas

2.2.1 Preliminares

As aplicações científicas suportadas por este trabalho acessam dados e programas, a partir de consultas de integração formuladas com a linguagem SQL, como a apresentada no exemplo 1.2.1.

Consultas em SQL apresentam a forma:

Select $a_i, b_i, \cdots, a_n$

From $R_1, R_2, \cdots, R_n$

Where $p_1 \wedge p_2 \wedge \cdots \wedge p_n$

Onde R_i são relações, e p_i são predicados.

$\square$

Um predicado é uma função *booleana* de seus operandos, definido no contexto do cálculo de predicado de primeira-ordem [Stol63]. A operação de seleção é definida com base em uma relação e um predicado de seleção do tipo $A\ \theta\ c$, onde A é um atributo de uma relação R, θ é um operador de comparação aritmético $<, >, =, \neq, \leq, \geq$ e c é uma constante. A operação de junção apresenta um predicado de junção do tipo $A\ \theta\ B$, onde A e B são atributos das relações operandas.

Consultas consideradas no trabalho, compreendem o sub-conjunto das possíveis expressões SQL, formado por expressões do tipo *SPJ*, ou seja, consultas compostas, exclusivamente, de operações de *seleção, projeção,* e *junção.* Predicados de junção e seleção são combinados em expressões conjuntivas.

Considerando-se consultas conforme definido, apresenta-se, a seguir, o processo que os avalia para geração de um plano de execução de consultas (PEC) distribuído.

2.2.2 Fases de avaliação de consultas

O processo de avaliação de consultas pode ser dividido em duas fases, *otimização lógica* e *otimização física.* Na primeira fase, parte-se de uma consulta, expressa em uma linguagem de alto nível, e efetuam-se transformações, com o objetivo de gerar uma representação simplificada, da consulta inicial. Produz-se uma estrutura de dados que representa a consulta, segundo um formato canônico, adequado a seu processamento. Em geral, adota-se a representação através de grafo de consulta (ver seção 2.3.3). Na segunda fase, com base em um modelo de custo e um algoritmo de exploração do universo de planos possíveis de execução de consultas (espaço de busca), determina-se o plano de menor custo e se o envia para execução.

Pode-se sintetizar as etapas do processo de avaliação de consultas, conforme apresentado em [Koss00]:

Análise

Durante a fase de análise, a consulta global é validada sintática e semanticamente, frente à linguagem de consulta e ao modelo global de integração, respectivamente. A consulta é transformada em uma forma intermediária, sofrendo modificações, que a tornam mais simples (e.g. resolvendo redundâncias e simplificando expressões). A estrutura intermediária, representa a consulta em um formato canônico (por exemplo, grafo de consulta [Ullm82]) permitindo a expressão dos componentes da consulta (tabelas e operações lógicas relacionais) sem pré-definição de uma ordem de execução.

Reescritura

A fase de reescritura efetua transformações algébricas sobre a estrutura em formato intermediário apoiadas em regras de equivalência entre operações da álgebra relacional, sem alterar a semântica da consulta. Adicionalmente, a consulta global é sub-dividida em sub-consultas, a serem enviadas às fontes de dados; e em uma consulta integradora dos resultados intermediários.

Otimização

Na fase de otimização, analisam-se planos de execução de consulta gerados a partir de um plano lógico inicial e modificados, sucessivamente, através de transformações algébricas e físicas. As primeiras, fazem uso das propriedades comutativa e associativa das junções, e distributiva das seleções sobre as junções, para modificar o plano inicial.

Em otimizadores baseados em modelo de custo [NGT98], os planos equivalentes são custeados e o de custo mais baixo é selecionado para execução. A determinação do plano ótimo pode ser realizada, exaustivamente, pela geração e cômputo do custo de todos os possíveis planos equivalentes à consulta (i.e. espaço de busca da consulta). Considerando-se o tamanho do espaço de busca produzido por uma análise exaustiva dos possíveis planos de execução, otimizadores adotam heurísticas que limitam esse espaço a um sub-conjunto dos planos possíveis.

O custo de cada plano é calculado com base nos custos de execução de cada operação, na árvore de consulta, que, por sua vez, depende das cardinalidades [1] das relações envolvidas e de características de implementação da operação.

Execução

Uma vez que o otimizador tenha determinado o PEC de menor custo, esse é enviado para processamento pela máquina de execução. Essa última compreende a implementação dos operadores, lógicos e físicos, suportados pelo sistema. Ela é responsável por ativar a execução, na cadeia de operações explicitadas no PEC, e retornar o conjunto resposta à aplicação cliente.

Tendo-se apresentado, sucintamente, as etapas que compreendem o processo de avaliação de consultas, nas próximas seções discutem-se alguns conceitos e técnicas que participam do processo de avaliação e são importantes para a apresentação dos capítulos seguintes.

2.2.3 Modelos de representação de consultas

A consulta, expressa em linguagem declarativa pelo usuário, será representada por meio de duas estruturas de dados, durante o processo de sua avaliação. Inicialmente, conforme descrito na etapa de análise, ela é modelada em um grafo de consulta. Em seguida, a partir de relações de dependência de dados entre as arestas do grafo, deriva-se um plano de execução de consulta (PEC), representado por uma árvore de execução. Dadas as características distribuídas da aplicação, o PEC gerado é adaptado para suporte ao paralelismo entre os sítios, transformando-o em um plano de execução de consulta distribuído (PECD).

Nas seções seguintes apresentam-se estas estruturas de dados e discutem-se alternativas de implementação de paralelismo nos PECs.

Grafo de Consulta

O grafo de consulta (GC) [Daya87] modela consultas declarativas, em um formato canônico. Dada uma consulta q, o grafo de consulta GC(q)=(V,E,P,PF) modela q,

[1] número de tuplas

onde V é um conjunto de vértices, representando as tabelas referenciadas em q; E é um conjunto de arestas, representando os predicados de junção; P é um conjunto de predicados, unários e binários, na consulta; e PH é o conjunto de atributos listados na projeção final. Na figura 2.2, apresenta-se o grafo de consulta correspondente à expressão em SQL abaixo.

Select p.região, p.indprec, rp.map
From PrecReg p, RegPolui rp
Where p.região = rp.região

□

Este trabalho restringe-se ao estudo de consultas modeláveis por um grafo em forma de árvore (i.e. *acíclico*) e apresentando ramos conexos (i.e. sem produto cartesiano).

O GC é utilizado pelo otimizador como fundamento para geração da árvore de execução inicial. As relações de dependência de dados entre os componentes de consulta, expressos no CG, são considerados ao se projetarem planos paralelos de execução.

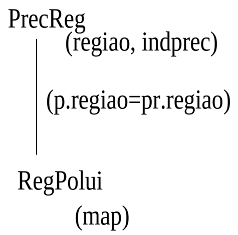


Figura 2.2: *Grafo de Consulta*

Árvore de consulta

Uma árvore de consulta (ou árvore de processamento) é uma abstração do plano de execução de consultas, na forma de uma árvore binária, onde os nós folhas representam relações de base, e os demais nós representam operações lógicas ou físicas [Grae93]. Entre operações, arestas representam a relação intermediária, produzida pela operação anterior e processada pela operação seguinte.

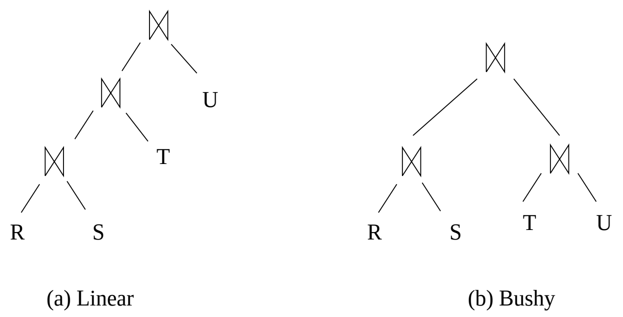


Figura 2.3: *Topologia de árvores de execução de consultas*

Encontram-se, na literatura, estudos que analisam a topologia de árvores de consulta versus o espaço de busca e o processo de otimização de consultas [LVZ93, SDeW90, FMLS99]. As principais topologias a serem consideradas são a Linear e a *Bushy* (ver figura 2.3). Na primeira, o espaço de busca compreende os planos gerados pela permutação das n relações. Árvores lineares são bastante interessantes, do ponto de vista de execução de consultas com predicados, na medida em que cada operação de redução (seleção ou junção) beneficia a operação subsequente, reduzindo o número de tuplas da relação intermediária, o que acarreta a minimização do custo de execução. Planos de topologia *Bushy* apresentam vários programas independentes, que podem ser computados em paralelo. Reduções exercidas por predicados em um ramo da árvore não influem no tamanho dos operandos de operações em outros programas; o que pode produzir um custo total mais elevado que um equivalente linear, porém, apresentando custo, mensurado quanto ao tempo de execução da consulta, possivelmente menor. O espaço de busca coberto por árvores *Bushy* é extremamente maior que o produzido por árvores lineares, da ordem de $\frac{(2n-2)!}{(n-1)!}$ [LVZ93], sem produto cartesiano, na medida em que todas as associações parentéticas são possíveis. Enquanto o espaço maior de planos aumenta a probabilidade de obtenção do plano ótimo, o tempo gasto no processo de otimização pode tornar-se inviável, frente ao tempo de execução de consultas não muito complexas (i.e. menos de 5 ou 6 relações em um ambiente centralizado).

2.2.4 O plano de execução de consultas distribuído

O PECD é produzido a partir da árvore de execução de consultas, com a introdução de operações físicas, que implementem tratamento de acesso e transferência de dados, além de controle sobre o sincronismo de execução em paralelo.

Planos lógicos e físicos

A representação do plano de execução de consultas, conforme apresentado, expressa uma combinação dos operadores da álgebra relacional segundo as regras de transformação e reescritura de consulta. A árvore de execução, composta apenas por esses operadores, é denominada plano lógico.

Para completar o plano lógico, e gerar um plano de execução, é necessário incluir, na árvore correspondente, operadores físicos. Esses últimos isolam os operadores lógicos de detalhes de implementação, tais como: leitura de disco, transferência pela rede, materialização de dados, comunicação entre processos etc. Além dos operadores estritamente físicos, o plano de execução é complementado pelos algoritmos de implementação das operações lógicas (principalmente para a operação de junção) e anotações, com meta-dados sobre os operandos e sobre o ambiente de execução.

A inclusão de operadores físicos proporciona, aos algoritmos de operações relacionais, independência de implementação, podendo ser concebidos sem preocupação quanto ao ambiente de execução [Grae93]. Em particular, em ambiente de execução distribuído, como em sistemas de mediadores Le Select, podem-se projetar planos lógicos de execução, como em ambiente centralizado, que se valham de operadores especiais para implementar uma estratégia de execução distribuída.

Paralelismo inter-operador em consultas distribuídas

Em um ambiente distribuído pode-se reduzir o tempo de resposta em consultas, e aumentar o uso de recursos distribuídos, paralelizando-se a execução de seus operadores entre os sítios participantes. A execução de diferentes operadores de um PECD, em paralelo, é chamada de *paralelismo inter-operador* [DeWG92]. Podem-se considerar dois tipos de paralelismo inter-operador: *independente* e *de fluxo de dados*. Paralelismo independente provê um mecanismo para se processar sub-planos, independentemente. Um exemplo, bastante oportuno de paralelismo independente, pode ser observado no processo de avaliação de consultas, em sistemas de mediadores como [TRV98, MR00, GPQ+97]. Nesses sistemas, consultas são expressas em árvores com topologia *Bushy*, compostas de consultas globais de integração e sub-consultas. Cada sub-consulta, a ser processada por uma fonte de dados, representa um programa independente, paralelizável em relação aos demais.

A adoção da estratégia de paralelismo independente minimiza, em alguns casos, o tempo de duração da consulta [2], em detrimento do tempo total de execução [3]. Paralelismo independente entre operadores de redução (i.e seleção ou junção) aumentam o tempo total, na medida em que um predicado não se beneficia das reduções daquele executado em paralelo.

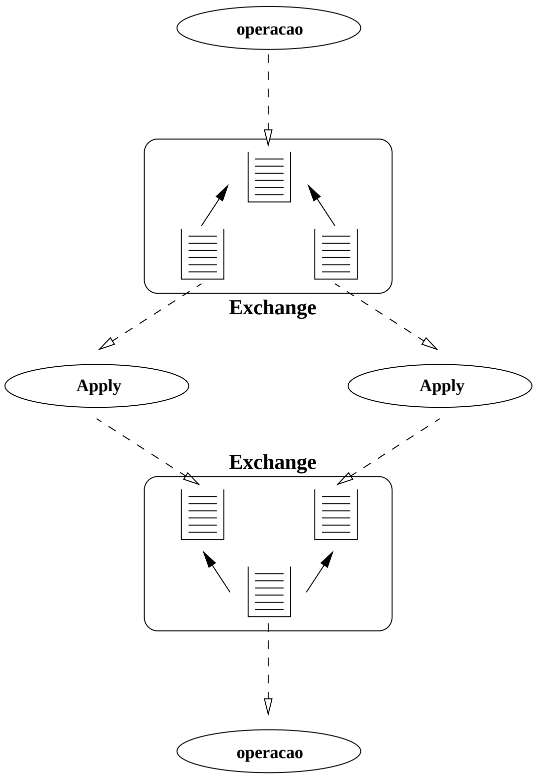


Figura 2.4: *Operador Exchange integrando paralelismo independente e de fluxo de dados*

Uma outra estratégia de paralelismo inter-operador é denominada *fluxo de dados*

[2]diferença entre o instante de término e o instante de início da consulta

[3]tempo considerado como a soma dos tempos gastos em cada operador

ou *pipeline*. Nessa estratégia, utiliza-se uma árvore de topologia linear e aloca-se, cada operador, em um processador diferente. Tuplas, que atendem ao predicado de uma operação, passam ao próximo operador, que inicia seu processamento. Dessa forma, a partir do instante em que uma tupla passa por todas as operações, o conjunto de processadores alocados mantém-se em carga constante, processando as operações em paralelo. A estratégia é benéfica na medida em que operações, em nós mais altos da árvore de execução, processam apenas o sub-conjunto de tuplas produzido pela sub-expressão à esquerda (resp. à direita) da árvore profunda à esquerda (resp. à direita), reduzindo o custo de execução. Por outro lado, a dependência de execução entre os operadores pode apresentar desequilíbrio na distribuição de carga entre os processadores, na medida em que variações em tempo de execução (i.e. seletividades, disponibilidade de fontes de dados [AFTU96], etc.) impeçam a progressão uniforme das tuplas pela árvore de execução.

O modelo de paralelismo independente, implementado em árvore *Bushy*, adequa-se bem ao modelo iterador [Grae93] de execução. Nesse caso, a operação de junção integra, ao processo *pipeline*, os ramos com programas independentes. Alternativamente, pode-se conceber topologias mais complexas, onde ramos independentes ocorrem em partes do PECD e são conectados, em ambos os nós terminais, com ramos *pipeline* (ver figura 2.4) via o operador *Exchange* [Grae93].

O operador *Exchange* é o responsável pela integração das duas estratégias paralelas: independente e de fluxo de dados. A comunicação com a estratégia de fluxo de dados dá-se na interface com os operadores segundo o modelo *iterador*. Por outro lado, a comunicação entre os componentes produtores e consumidores do operador *Exchange* é estendida para que ambos possam se comunicar com mais de um operador, possibilitando a execução entre ramos independentes do PECD.

O otimizador de consultas do sistema Le Select, por exemplo, diferentemente da maioria dos sistemas de mediadores, produz um PECD que combina as topologias *Bushy* e linear. A topologia linear implementa uma arquitetura paralela de fluxo de dados entre os servidores Le Select. Complementarmente em cada sítio, um sub-plano formado pela consulta a ser resolvida localmente pelo servidor Le Select, ou pela fonte de dados, implementa uma árvore de topologia *Bushy*.

2.2.5 Estimativa das características de relações intermediárias

Para completar a discussão sobre a formulação do PECD é preciso apresentar o processo de captura de meta-informações sobre os objetos participantes do PECD que servirão de base para o cômputo do custo dos planos.

O custo de avaliação de um PECD, em um sistema de mediadores, depende do tamanho dos resultados das sub-consultas enviadas às fontes de dados. Para estimar esses valores o modelo de custo precisa conhecer a cardinalidade das relações de base, bem como o tamanho médio de cada tupla. Precisa também do número estimado de valores distintos, para atributos que participam de predicados de igualdade, além dos valores mínimos e máximos, em atributo, que participem de predicados de desigualdade [NGT98]. Esses valores são exportados pelos adaptadores e podem ser armazenados em catálogo do mediador, ou serem capturados dinamicamente, em tempo de otimização.

A partir dos valores estatísticos capturados, o otimizador computa o tamanho de relações intermediárias através de expressões que modelam o fator de seletividade dos predicados. Nas próximas seções apresenta-se o modelo de estimativa de seletividade utilizado neste trabalho.

Seletividade de predicados

A seletividade de um predicado pode ser definida como o potencial de redução de um predicado sobre uma relação de entrada, e é expresso por um valor entre 0 e 1. Uma seletividade próxima a 1 (resp. a 0) é dita baixa (resp. alta), uma vez que apresenta um baixo (resp. alto) potencial de redução.

Sejam R e S relações compostas, entre outros, pelos atributos a e b, respectivamente. $||R||$ representa a cardinalidade da relação R, Dom(R.a) representa o domínio dos valores do atributo a e $||R.a||$ o número de valores distintos do atributo $R.a$. Em [SACL+79], predicados de seleção tem seletividade calculada como abaixo:

- Sel(R.a $= valor) = \frac{1}{||R.a||}$
- Sel(R.a $> valor) = \frac{max(R.a) - valor}{max(R.a) - min(R.a)}$
- Sel(R.a $< valor) = \frac{valor - min(R.a)}{max(R.a) - min(R.a)}$

O predicado de junção requer um pouco mais de cuidado, uma vez que o potencial de redução deve ser verificado em função do percentual redutor de cada relação participante, tal como no *semi-join* [HY79]. Assim, dada a junção ($R \bowtie S$) calcula-se o percentual redutor do *semi-join* ($R \propto_{a=b} S$) e ($S \propto_{b=a} R$). Onde,

- $\text{Sel}(R \propto_{a=b} S) = \frac{||S.b||}{||Dom(b)||}$
- $\text{Sel}(S \propto_{b=a} R) = \frac{||R.a||}{||Dom(a)||}$

O problema de estimativa de seletividade tem sido extensamente pesquisado como em [LNS90, PI97]; não obstante, Ioannidis mostra em [IC91] a impossibilidade de determinação da cardinalidade de uma relação intermediária após a terceira junção, em virtude da sucessão de aproximações nos passos anteriores. Adicionalmente, o cálculo do fator de seletividade, baseado no conhecimento do domínio de valores de atributos, pode não ser possível, em um ambiente de integração de grande escala, como na internet.

Em R^* ?? a seletividade de junção é computada como a razão, entre a cardinalidade da relação resultante e a cardinalidade do produto cartesiano das relações de entrada. Esse modelo tem característica pessimista por considerar o universo de tuplas no produto cartesiano entre as relações, e requer o conhecimento da cardinalidade do resultado da junção que poderia ser armazenado a partir de execuções anteriores.

Estimativa sobre relações intermediárias

Tendo-se estimado o fator de seletividade de um predicado, pode-se computar as cardinalidades de relações intermediárias. Esta seção apresenta o modelo, utilizado no protótipo de otimizador implemento (ver validação), para cálculo da cardinalidade de relações intermediárias, bem como, do número de valores distintos em seus atributos.

Quanto a esses, considera-se a premissa tradicional de distribuição homogênea dos valores em atributos.

Para a operação de seleção, aplica-se, diretamente, o fator de seletividade sobre os valores da cardinalidade da relação e sobre o número de valores distintos. Sejam $R = \{a_1, a_2, \cdots, a_n\}$, uma relação, e *S(R)*, uma operação de seleção sobre *R*, com fator de seletividade *sf*, calculado como apresentado na seção anterior. Considere-se, ainda, *R'* como a relação gerada pela avaliação de *S(R)*. Define-se a cardinalidade de *R'*, $\mid R' \mid$, e o número de valores distintos do atributo $R'.a_i$, $dist(R'.a_i)$, como:

$\mid R' \mid = \mid R \mid *sf$ e $dist(R'.a_i) = dist(R.a_i) * sf$;

Para a operação de junção, com fator de seletividade *sh*, entre relações R e *S*, onde $S = \{b_1, b_2, \cdots, b_n\}$, adota-se o cômputo da cardinalidade da relação resultante *RS*, $\mid RS \mid$, como:

$\mid RS \mid = max(\mid R \mid *sf, \mid S \mid *sf)$ (1) , se $\mid RS \mid \subset [0.5*min(\mid R \mid, \mid S \mid), 1.5*max(\mid R \mid, \mid S \mid)]$ (2), senão, a cardinalidade é considerada como o valor limite do intervalo em (2) mais próximo ao calculado em (1).

Por fim, para computar o número de valores distintos em um atributo $a_i \in RS$, $dist(RS.a_i)$, gerada por uma operação de junção entre R e *S*, calcula-se o fator de seletividade da junção, sobre *R*, sj_R, e, respectivamente, sobre *S*, sj_R. Em seguida, efetua-se como em operações de seleção:

$sj_R = \frac{|RS|}{|R|} \Rightarrow dist(RS.a_i) = dist(R.a_i) * sj_R$.

Em casos onde $\mid RS \mid > \mid R \mid$ adota-se, para o valor de distintos em $\mid RS.a_i \mid$, a redução direta do predicado de junção sobre a_i, $\mid R.a_i \mid$.

A apresentação das expressões para o cálculo da cardinalidade e do número de valores distintos de atributos completa a discussão sobre os conceitos de base sobre avaliação de consultas. Na seção seguinte, estendem-se as consultas apresentadas, para representação de chamadas a programas.

2.3 Programas de usuários em consultas

Um importante aspecto de adequação de SGBDs e Sistemas de Mediadores [Wied92] às necessidades de aplicações científicas é a possibilidade de extensão de funcionalidade do sistema através de programas desenvolvidos pelos próprios usuários. Alguns exemplos destes tipos de programas incluem: processamento de imagens, implementação de modelos matemáticos sobre matrizes de dados amostrais, etc. A extensão de SGBDs nesta direção permite que sejam incluídas, em consultas a banco de dados, invocações a programas científicos, integrando programas aos dados analisados.

2.3.1 Classificação de funções de usuário

De forma a modelar a participação de funções de usuário em consultas Jaedick, em [JM99], define quatro tipos do que chama de *rotinas definidas pelo usuário*, baseado na variação da estrutura de dados de entrada e de saída de funções em consulta. O tipo de função *UDSF*, *função escalar definida pelo usuário*, compreende todas as funções escalares que recebem como entrada uma tupla e retornam, da mesma forma, uma tupla [4]. Uma segunda categoria de funções, *UDAF*, *funções de agregação definidas por usuários* engloba todas aquelas que recebem como entrada um grupamento de tuplas e retornam uma única tupla. Simetricamente, *UDTF*, *funções de tabelas definidas por usuários* recebem uma tupla de entrada e produzem uma tabela na saída. Sendo menos comum que as anteriores, sua apresentação merece ser ilustrada. Considere-se uma função que recebe o mapa de humidade da relação *PrecReg*, apresentada no exemplo 1.2.1, e gere uma tupla para cada região da imagem, segundo um intervalo de valores de humidade. O resultado da aplicação da função seria uma tabela contendo tuplas por graus de humidade. Finalmente, a categoria *UDTO*, *operadores de tabelas definidos por usuário*, recebem como entrada uma ou mais tabelas e geram, como saída, uma tabela. *UDTO* foi proposto em [JM99] com o intuito de proporcionar extensões dos operadores relacionais binários (i.e. *join*) para tratamento de dados não estruturados. A motivação seria obter ganhos de performance através do fornecimento de algoritmos específicos para operações binárias com predicados sobre dados não estruturados.

2.3.2 Um modelo para representação de programas

Na seção anterior, apresentou-se uma classificação de funções quanto a seus parâmetros de entrada e de saída. À luz daquela classificação, este trabalho trata, exclusivamente, de funções definidas segundo um sub-conjunto das funções de tipo *UDSF*. Esse conjunto é definido pelas funções cujo custo de execução, por chamada, varia de alguns segundos a até horas, como é o caso de programas científicos. Parâmetros de entrada e de saída são estruturados em tuplas, e a composição da tupla de entrada foi restringida a conter atributos de uma única relação de base [5]. Com respeito à

[4] onde a tupla de saída pode ser formada por qualquer projeção do produto cartesiano das relação de entrada, combinada com o resultado da avaliação da função

[5] elimina-se a possibilidade de considerar-se o produto cartesiano entre relações.

terminologia, consideram-se, indistintamente, os termos *função*, *função de usuário*, *programa* e *programa científico*, apesar de, em sua maior parte, o texto referir-se à expressão *programa*. A razão dessa opção é refletir o fato de que esses objetos não foram construídos com a inteção de serem utilizados em consultas; muito pelo contrário, sua integração ao sistema de consulta requer tratamentos específicos, a serem fornecidos por adaptadores de programas.

2.3.3 Sintaxe para publicação de programas

Programas científicos do tipo *UDSFs*, são modelados, utilizando-se a representação BNF estendida, como:

<Programa> ⟶ <nomePrograma> (<param> in ,<param> out)
<nomePrograma> ⟶ string
<param> ⟶ <tupla>
<tupla> ⟶ <listaAtributo>
<ListaAtributo> ⟶ <atributo> {<atributo>}
<atributo> ⟶ <nome de atributo> <tipo dado > || <constante>
<nome de atributo> ⟶ string
<constante> ⟶ literal
<tipo dado> ⟶ {int, float, string, Blob, etc }

Adotando-se o modelo de dados acima, a publicação de um esquema de banco de dados através de um sistema de mediadores, incluindo a especificação de um programa, é dada conforme abaixo:

Exemplo 2.3.1 *Esquema de uma aplicação científica*

- ***Tabela***: *PolReg (regnome String, estação int, mapa Blob)*
- ***Tabela***: *PrecReg (nome String, temp float, prec Blob)*
- ***Programa***: *Pol (<mapaPoluição Blob> in, <polIndex float > out)*

A representação da chamada de programas em consultas é feita associando-se os atributos de entrada com seus parâmetros e alocando-se sua chamada em uma posição adequada na consulta.

A seguir, mostra-se o programa Pol sendo chamado em duas consultas. Na consulta (a), o programa participa da lista de projeções e na consulta (b), aparece como parte da avaliação dos predicados.

Select *regnome, Pol(p.mapa)*
    From *PolReg p;*
        (a)
Select *p.estação, pr.temp*
    From *PolReg p, PrecReg pr*
    Where *Pol(p.mapa) > 1.5 and*
    p.regnome=pr.nome
        (b)

□

Semântica da execução de consultas com programas do tipo UDSF

Programas do tipo *UDSF* podem ser invocados em consultas SQL [6] em duas das suas três cláusulas, *SFW* (i.e Select, From, Where). Podem ser invocados na lista de projeções da consulta (S) ou como parte de predicados da cláusula *Where (W)* (ver os exemplos acima). Do ponto de vista do modelo de execução, programas são operadores que iteram sobre a variável de tupla associada à expressão relacional subjacente.

Cada iteração do programa procede da seguinte forma: dada uma relação R e um programa P, considerem-se as projeções *PJ(R)* e *PJ'(R)*, composta dos atributos de R associados à entrada de P e de seu complemento referente ao esquema de R, respectivamente. Para cada tupla de R, aplique-se P sobre o valor dos atributos de *PJ(R)*, gerando a tupla *t'*, conforme o tipo definido na tupla de saída. Produz-se uma nova tupla com estrutura formada pela concatenação de *t'* com a projeção *PJ'(R)*.

[6] consultas do tipo select/project/join

Este modelo de execução corresponde a uma visão local e ingênua, da execução de programas no contexto maior do plano de execução de uma consulta completa. Para integrar a execução de programas ao processo de avaliação de consultas, alguns trabalhos propõem alterações nas estratégias conhecidas de otimização e execução.

Representando consultas com invocação de programas por meio de um grafo de consulta

Uma das ferramentas de auxílio ao processo de avaliação de consultas é o grafo de consulta. Em particular, a estratégia de execução apresentada neste trabalho, usa o GC como base para a geração de planos paralelos. Nesta seção , define-se uma extensão para o GC que permite a modelagem de chamadas de programas em consultas.

Define-se o grafo de consulta **G=(V,E,P,PF)**, estendendo-se a proposta de representação de funções de agregação em grafos de consultas em [Daya87], onde V é um conjunto de nós e E representa um conjunto de arestas. $V = R \cup P$, onde R é o conjunto de relações da consulta e P é o conjunto de programas na consulta. As arestas em E são de dois tipos: não direcionadas e direcionadas. O primeiro tipo define operações de junção, associando duas relações. O segundo tipo define uma ordem de precedência entre nós, onde o nó de origem da seta será computado antes do nó de destino. Assim, para se processar a invocação de um programa, têm-se que obter as tuplas da relação de entrada.

O conjunto de predicados em P é composto por predicados unários (seleções simples [7] e predicados caros [8]) ou predicados binários (junções).

Na figura 2.5 o GC, referente ao exemplo 2.3.1, é representado. O predicado caro é expresso como uma restrição ao resultado da execução do programa *Pol*. Obedecendo-se a regra de precedência, o acesso a relação *PolReg* deve preceder à execução do programa.

A definição da ordem de precedência no GC possibilita a expressão de diversas hipóteses de invocação de programas em consultas. Vejam-se os exemplos da figura 2.6. O GC em *(a)*, representa uma invocação aninhada de programas. O programa *cut* reduz uma imagem segundo dimensões estipuladas. De posse da imagem em

[7] seleções sobre valores de atributos

[8] seleções sobre resultados da execução de programas

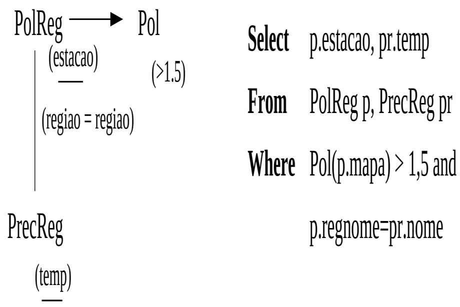


Figura 2.5: *Grafo de consultas com invocação de programas*

tamanho reduzido, o programa *Prec* analisa precipitações pluviométricas, apenas na área significante da imagem.

Discussão

Note-se a importância da restrição de precedência para o processo de avaliação de consultas. Nesse exemplo, a ordem de invocação entre programas, no plano de execução de consultas, precisa ser preservada. No exemplo em *(b)*, vê-se o programa *P1* associado a dois vértices. A representação denota uma operação de junção entre as duas relações, implementada pelo programa, já que não há uma aresta não direcionada entre as relações. Por outro lado, em *(c)*, a aresta ligando R e S modela uma junção, entre as relações. O predicado aplicado sobre o resultado de *P1* sugere uma simples restrição no resultado da junção. Em seguida, no GC em *(d)*, representa-se uma junção, entre o resultado da avaliação do programa *P1*, sobre a expressão relacional presente na origem da aresta direcionada, e a expressão associada à extremidade oposta da aresta. O grafo retrata a característica assimétrica da junção, onde uma das relações operandas apresenta restrições de acesso sobre suas tuplas (ver 3.3.2). Na figura *(e)*, fica clara a independência entre as invocações dos programas listados na projeção final da consulta. O otimizador poderia, por exemplo, valer-se de um plano paralelo, usando uma estratégia independente para cada avaliação de programa. Em *(f)*, o resultado da invocação do programa *P1* é utilizado tanto no predicado caro quanto na projeção final. O grafo, nesse cenário, auxilia o processo de normalização da consulta, detectando redundâncias. Finalmente, o GC em *(g)* apresenta o caso que mais interessa a este trabalho, onde consultas compreendem o tratamento de mais de um predicado caro.

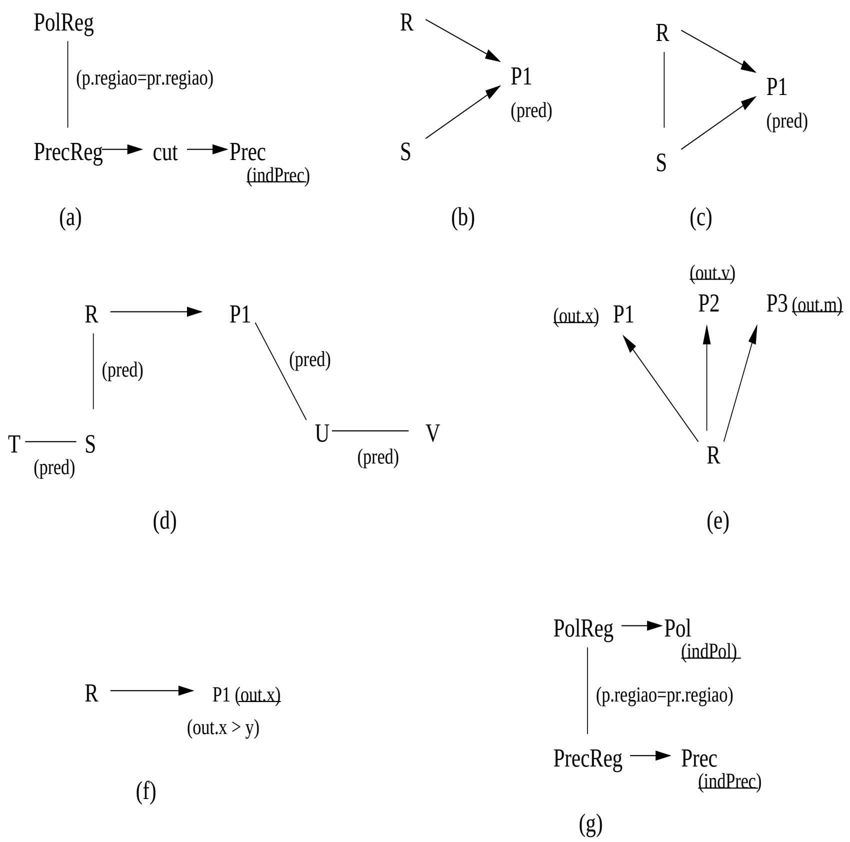


Figura 2.6: *Potencial analítico de grafos de consulta para programas*

Conforme foi mostrado, a representação da consulta através de um grafo provê uma estrutura de dados passível de avaliação pelo processador de consultas. Algumas transformações e otimizações podem ser derivadas automaticamente, percorrendo-se o grafo.

2.4 Avaliação de consultas com programas

Nesta seção, apresentam-se os mais importantes trabalhos que tratam da integração de programas ao processo de avaliação de consultas.

2.4.1 A aborgadem LDL

Nesta abordagem, predicados sobre o resultado da execução de programas definidos pelo usuário são tratados como uma relação virtual, do ponto de vista de otimização. Como exemplo, considere-se o programa *Pol*, apresentado em 2.3.1 *(b)*. Durante o processo de otimização, o programa é representado por uma relação virtual, com esquema, $Pol\prime = \{mapa\ Blob, polIndex\ float\}$. O benefício de se modelar predicados caros através de relações, está no aproveitamento direto de estratégias de otimização de consultas tradicionais, baseada em *Programação Dinâmica*, [SACL+79]. Logo, se e é um predicado caro e R_1 e R_2 são relações, então o otimizador irá analisar o espaço de busca dos planos formados pelas *relações* R_1, R_2 e $e\prime$, onde a última corresponde a relação virtual representando o predicado caro e.

Um problema na proposta *LDL* resulta da restrição do projeto ao uso exclusivo de árvores lineares. Nesse caso, planos, onde o predicado sobre o resultado de programas pudesse ser aplicado sobre a relação, antes da aplicação dessa em uma junção, não são considerados. Retome-se o exemplo dado no parágrafo anterior; um plano possível seria $R_1 \bowtie \sigma_{e\prime}(R_2)$. No entanto, uma vez que planos gerados restringem-se a topologia linear, o plano mostrado nunca seria produzido. Nesse caso, a estratégia *LDL* pode produzir planos que sejam piores que os gerados pela heurística tradicional de se executar seleções próximas às relações.

Por outro lado, a estratégia de modelagem de predicados caros como relações, aumenta, consideravelmente, o espaço de busca a ser analisado. Em uma consulta com n relações e k predicados, gera-se 2^{n+k} planos [CS99].

Outras propostas consideram, igualmente, a modelagem de programas como relações. Em [MS99], apresenta-se uma estratégia de execução para programas distribuídos em uma arquitetura cliente/servidor. A estratégia minimiza o custo de transferência entre o cliente e o servidor, conectados através de ligações de redes assimétricas, com alta taxa de transferência no sentido servidor-cliente, e baixa no sentido inverso. A solução proposta emprega os algoritmos de *semi-join* e, uma versão da operação de junção, chamada *client-site* join, para execução de programas. Nessa última, uma máquina de execução, localizada no cliente, opera a junção entre tuplas enviadas pelo servidor e a relação virtual. A estratégia é integrada ao algoritmo tradicional de programação dinâmica, que é estendido para considerar a localização do resultado da aplicação do programa. Tal otimização objetiva atender a programas localizados no mesmo sítio. Os autores do trabalho alegam que a característica distribuída da aplicação torna ineficiente a estratégia *rank order* [HS93]; dado que a proximidade em um mesmo sítio de predicados, pode determinar uma ordem de execução com custo menor que a sugerida pelo *rank order*.

2.4.2 O algoritmo Predicate Migration - rank order

O algoritmo *Predicate Migration* tem como objetivo a alocação de predicados caros, em pontos *ótimos* da árvore de execução, em relação as operações de junção. A proposta considera a árvore de execução segundo *streams*, compostos de operações entre a raiz da árvore e cada uma de suas folhas, onde árvores apresentam topologia linear sem restrição (i.e. um predicado de seleção pode preceder à participação da relação na junção [9]).

Para cada predicado, caro ou de junção, determina-se o custo de sua avaliação por tupla. Para predicados de junção, o custo por tupla é considerado como o custo diferencial da avaliação da operação, sobre n tuplas, com respeito à sua execução sobre uma relação apresentando cardinalidade de $n+1$ tuplas. Adicionalmente, o algoritmo considera conhecidos os fatores de seletividade dos predicados caros. Para predicados de junção, calcula-se a seletividade da operação, em cada *stream*, segundo a expressão de seletividade para operações de semi-join. Ou seja, para a junção $R \bowtie S$, considera-se a seletividade sobre R, S_{jR}, como $S_{jR} = \frac{|O_j|}{|R|}$, onde $\mid O_j \mid$ é a cardinalidade da relação produzida pela junção. Similarmente, computa-se a seletividade sobre S, S_{jS}.

Tendo-se computado a seletividade em ambos ramos, para cada predicado, e seu custo por tupla, calcula-se o *ranking* do predicado, dado pela expressão $\frac{1-seletividade}{custoportupla}$. O algoritmo usa o resultado em [MS79], onde mostra-se que, em um *stream*, a ordenação de operações, sujeitas a restrições semânticas, que produz o menor custo de execução é aquela que obedece à ordenação descendente de *rank*.

A adequação da estratégia de ordenação, baseada em *rank* de predicados, ao processo de otimização de consultas, dá-se segundo uma variação do algoritmo de exploração do espaço de busca dos planos, usando a heurística de programação dinâmica,

[9] predicate migration não modela funções como relações virtuais.

adotada no sistema R [SACL+79]. Inicialmente, todos os predicados caros são empurrados para junto das relações de base, utilizando-se da estratégia tradicional quanto aos predicados de seleção. Em seguida, a cada iteração do algoritmo, onde se inclui uma nova relação, analisa-se os *streams*, à direita e à esquerda da operação de junção. Caso existam predicados caros abaixo da junção, comparam-se seus *rankings* com o do predicado de junção. Os predicados com *rank* menor que o da operação de junção migram para acima da mesma no *PEC*.

Para exemplificar essa etapa, considere-se o plano parcial P ,$R \bowtie e_1(S)$, onde R e S são relações e e_1 é um predicado caro. Considere-se, também, o plano parcial *P'*, $e_1(R \bowtie S)$. Se $rank\ e_1(S) < rank\ e_1(R \bowtie S)$, então elimina-se o plano *P*, e guarda-se *P'*, que continuará o processo para sua complementação. Por outro lado, em caso contrário, ambos os planos são mantidos. O plano *P'* continua o processo de otimização, enquanto o plano P é guardado para pós processamento. Ao final do algoritmo de programação dinâmica, todos os sub-planos em que não foi possível migrar um predicado caro, e com isso foram mantidos, são exaustivamente complementados e têm seus custos comparados com o plano "ótimo" encontrado até então.

O algoritmo apresenta um avanço, com relação a estratégia *LDL*, na medida em que considera planos formados pela árvore linear sem restrição. Sua convergência para um plano ótimo ocorre em tempo polinomial, no número de predicados.

Chaudhuri [CS96, CS99] apresenta uma falha na estratégia de otimização empregada no algoritmo *Predicate Migration*. O cálculo do *rank* das operações de junção depende do custo de sua avaliação, que por sua vez é função da cardinalidade das relações de entrada. No entanto, essa última tem seu valor alterado, mediante a aplicação, ou não, do predicado caro antes da avaliação da junção. Logo, essa dependência cíclica pode levar a cômputos de *rankings* incorretos e, consequentemente, a planos de execução sub-ótimos. Adicionalmente, no pior caso, o número de planos guardados para análise posterior pode ser tal que leve a enumeração de todo o espaço de busca.

Com as considerações no último parágrafo, Chaudhuri,[CS96], apresenta uma variação do algoritmo *Predicate Migration*, melhor adaptada à heurística de exploração do espaço de busca, baseada em *Programação Dinâmica*. Sua estratégia consta de ordenar (descendentemente) predicados caros, segundo a estratégia *rank order*. Em seguida a cada etapa do algoritmo de programação dinâmica, o espaço de busca é acrescido de todas as seqüências de predicados caros, cuja alocação não fira a restrição semântica[10]. Cada sub-plano ótimo é enriquecido com anotações, informando os predicados de mais baixo *rank*, sendo avaliado até aquele ponto, em cada relação participante do plano. Essa estratégia corresponde a uma extensão às *ordens interessantes*, apresentada no sistema R [SACL+79].

Planos com o mesmo conjunto de predicados caros avaliados, sobre o mesmo conjunto de relações, são considerados equivalentes, e são todos, menos o de menor custo, eliminados. Adicionalmente, a heurística *pushdown* reduz o espaço de busca com a seguinte regra: se dois planos P e P' avaliam o mesmo conjunto de relações e $S \subset S\prime$ corresponde a lista de predicados caros avaliados em P, com $Custo(P) > Custo(P\prime)$, então o plano P pode ser eliminado.

2.4.3 Outros trabalhos

Em [SLR97], apresenta-se uma proposta para o uso de informações semânticas sobre funções no processo de otimização de consultas. Nesse trabalho, o sistema *Predator* é estendido para suportar a definição de *tipos abstratos de dados estendidos*, onde esses são TADs estendidos justamente para exportar a semântica de suas funções. Um exemplo de sua aplicabilidade pode ser observada em consultas com funções compostas, i.e $f(g(x))$, onde a semântica das funções indique uma ordem mais eficiente para avaliação do conjunto, por exemplo $g(f(x))$.

Um outro trabalho que estende os resultados do algoritmo *Predicate Migration*, com respeito à otimização de predicados caros, é o sistema MOCHA.

O *middleware* MOCHA [MR00] é um sistema mediador distribuído, para integração de grande número de fontes de dados na internet. Sua característica diferenciadora é a implementação de estratégia de execução de consultas que combina *data shipping* e *code shipping*. A implementação de *code shipping* é defendida considerando-se dois motivos principais. Primeiramente, a estratégia provê escalabilidade ao sistema, quanto a extensões de novos tipos de dados. Classes, implementando novos tipos de dados, são desenvolvidas em um sítio central, e enviadas para os sítios próximos às fontes de dados, em tempo de execução. Junto às fontes, instala-se um módulo do sistema responsável pelo acesso aos dados (adaptador) e por execução de operadores

[10] dependência de dados entre os predicados e as relações de base

de usuário. Em segundo lugar, argumenta-se que, em sistemas de integração de dados distribuídos pela internet, o custo de transferência pela rede é um fator importante para o desempenho das aplicações científicas suportadas pelo sistema.

Nesse contexto, a estratégia *code shipping* pode minimizar o custo de execução de consultas em cenários, onde a operação enviada para o cliente reduza o volume de dados a ser transferido ao sítio central, onde encontra-se o sistema integrador. Para tal, o algoritmo tradicional de avaliação de planos (i.e. programação dinâmica) é modificado para incluir a localização de operações do usuário.

O algoritmo inicia determinando a localização, próxima à fonte de dados ou no servidor, dos operadores de usuário que atuam sobre cada relação. Em seguida, os operadores localizados em cada fonte de dados são ordenados, com base em seu *ranking*, com em [HS93]. O algoritmo de programação dinâmica, adaptado segundo a estratégia proposta em [CS96], posiciona eficientemente os predicados caros executados no mediador e explora o espaço de busca de junções entre relações intermediárias.

2.5 Sumário

Neste capítulo apresentaram-se os conceitos fundamentais relativos ao processo de avaliação de consultas em sistemas de mediadores. O objetivo do capítulo é formular um panorama da área para que se possa acompanhar as propostas contidas nos capítulos 4 e 5.3.

Inicialmente, apresentam-se sistemas de mediadores como parte de uma família de sistemas de integração denominada sistemas de banco de dados heterogêneos. São apresentadas as duas principais classes desses sistemas, fortemente e fracamente acoplados, dentre as quais os sistemas ditos mediadores são classificados no segundo grupo. Em seguida, discute-se o processo de avaliação de consultas. Inicia-se com a apresentação das diferentes fases componentes do processo em sistemas de integração. Como componentes básicos do processo, discutem-se as estruturas intermediárias de representação de consultas. Apresentam-se o grafo de consultas, utilizado como suporte à geração de planos de execução, e as árvores de execução, estruturadas segundo as topologias, linear e *Bushy*. O plano de execução de consultas distribuído é descrito por meio de seus operadores, lógicos e físicos, alocados na árvore de execução.

Discutem-se, adicionalmente, os modelos, independente e de fluxo e dados, para construção de planos paralelos de execução de consultas. Por fim, prepara-se a discussão sobre modelo de custo paralelo, que será apresentado no capítulo 6, com as fórmulas para a estimativa do fator de seletividade de predicados e cardinalidades de relações intermediárias.

Tendo discutido o processo de avaliação de consultas em sistemas de mediadores, na segunda parte do capítulo discute-se a integração de programas em consultas à sistemas de bancos de dados. Inicialmente, especifica-se um modelo para representação de programas em sistemas relacionais. Utiliza-se a classificação de funções, proposta em [JM99], segunda a qual os programas tratados neste trabalho enquadram-se como funções tupla-a-tupla. Adicionalmente, propõe-se uma sintaxe, segundo o padrão *BNF*, para especificação de programas em um esquema de banco de dados. Por fim, descreve-se a semântica de execução de programas do tipo *UDSF* em consultas relacionais.

A última seção deste capítulo apresenta os principais trabalhos associados ao processo de otimização de consultas com predicados caros.

Capítulo 3

O Sistema Le Select

3.1 Introdução

A proposta de integração de aplicações científicas deste trabalho baseia-se no sistema *middleware* Le Select. O objetivo do sistema é integrar dados e programas heterogêneos, distribuídos e autônomos, possibilitando seu compartilhamento remoto por equipes de cientistas. Neste capítulo, apresentam-se as principais características do sistema que nortearam o desenvolvimento das estratégias formuladas.

No sistema Le Select, o ato de disponibilizar um item, dado ou programa, para acesso compartilhado é denominado publicação do item, e corresponde a seu registro no sistema, segundo uma visão Relacional. O registro de uma publicação se processa através da inclusão no sistema do respectivo *adaptador*, que são módulos de *software* responsáveis pela transformação do objeto a ser publicado, para o modelo Relacional. Uma vez registrado no sistema de integração, o item publicado pode ser acessado, a partir de consultas escritas na linguagem SQL. Adicionalmente, Le Select oferece uma interface particular dirigida à execução assíncrona de um programa científico.

Le Select executa consultas distribuídas a partir de um plano de execução com topologia *Bushy*. Programas, nesses planos, são modelados como relações virtuais, conforme em [CGK89], o que permite a adoção de estratégia de otimização baseada na proposta do SistemaR* [SA80].

Neste capítulo, apresentam-se, resumidamente, os seguintes aspectos do sistema:

sua arquitetura; os mecanismos de publicação de itens do acervo; o processo de avaliação de consultas.

3.1.1 Abordagem proposta

Le Select pode ser comparado com outros sistemas de mediadores como [TRV98, CHS+95, GPQ+97], porém apresenta três pontos importantes de distinção. Primeiramente, o sistema não oferece uma visão integradora global; o acesso a dados é realizado a partir da referência explícita ao identificador das tabelas. Em segundo lugar, além da integração de dados, possibilita também a integração de serviços remotos. E, finalmente, incorpora novas técnicas de otimização para atender à essa integração.

Uma fonte de dados para ser disponibilizada pelo sistema tem sua estrutura adequada à sua representação no Modelo relacional via código de programa do respectivo *adaptador*. Esses, por sua vez, são programas que apresentam uma interface muito próxima à adotada pelo padrão JDBC, para acesso aos dados e meta-dados por ele publicados. Sua construção e disponibilização em um servidor Le Select realizam a publicação de um item de acervo no sistema. Entretanto, dado que existem alguns tipos disponíveis de *adaptador* (como aqueles para acesso a base de dados relacionais, arquivos ASCII, etc.), em alguns casos não há necessidade de codificação para que se possa efetuar uma publicação.

Uma característica original do sistema é a capacidade de compartilhar serviços (programas científicos). No contexto de aplicações ambientais, essa característica é bastante importante, uma vez que a gerência da execução de programas complexos sobre dados distribuídos requer controles e técnicas avançadas, difíceis sem o auxílio de um *middleware*.

Com respeito à distribuição, um ambiente Le Select é composto de vários servidores Le Select que se comunicam para atender a consultas distribuídas, formando uma rede *peer-to-peer*. Cada servidor Le Select é munido de uma máquina de execução de consultas completa capaz de avaliar consultas locais, além de executar sub-planos enviados por outro componente do ambiente. Durante a execução de um plano distribuído entre vários servidores Le Select, os sub-planos comunicam-se entre si, formando uma árvore de execução de fluxo de dados entre sítios.

3.2 Arquitetura geral

Le Select é um *middleware* para integração de dados e serviços distribuídos pela rede. Na figura 3.1 apresenta-se a arquitetura geral de um ambiente integrado. A arquitetura provê uma máquina virtual homogênea de avaliação de consultas distribuídas baseadas na integração de diversos servidores Le Select. Cada participante é responsável pela publicação e acesso às suas fontes de dados contendo uma máquina de avaliação de consultas completa.

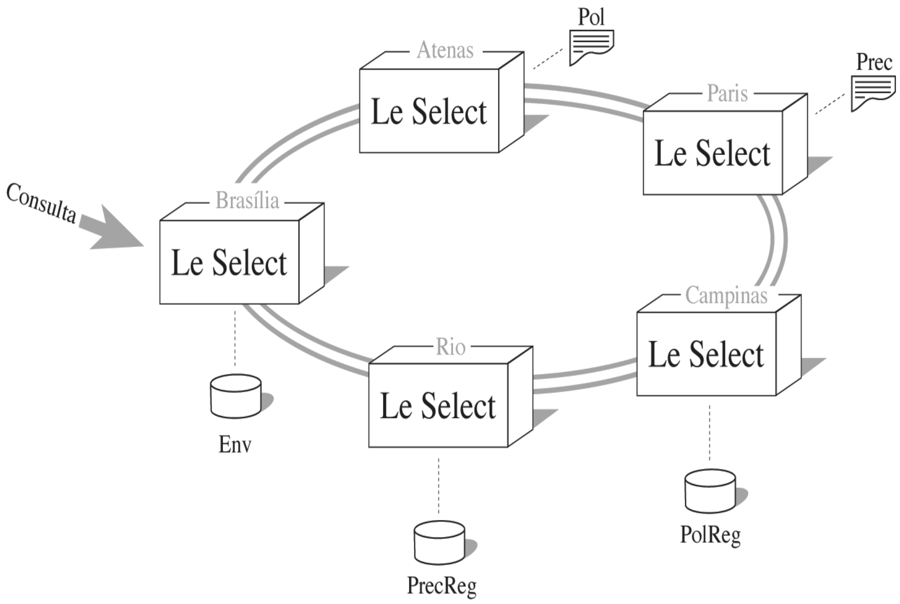


Figura 3.1: *Arquitetura de Publicação Le Select*

Le Select foi concebido com forte inspiração na arquitetura de sistemas de mediadores, que consiste no desenvolvimento de uma camada de software de integração, complementada por uma camada de acesso e homogenização da fonte de dados, conhecida como camada de adaptadores. Aplicações consumidoras de dados integrados acessam o sistema a partir de uma camada de comunicação e por meio de consultas escritas na linguagem SQL ou na linguagem de execução de serviços.

Um servidor Le Select pode participar da avaliação de consultas enviadas por clientes locais. Nesse caso, poderá resolver localmente a consulta, caso as tabelas participantes sejam todas publicadas pelo servidor local ou, de uma maneira mais geral, poderá colaborar com outros servidores na resolução de pedidos distribuídos.

3.2.1 Arquitetura interna

Um servidor Le Select recebe solicitações de consultas, a partir de aplicações, e subplanos de consultas globais. Os principais componentes de sua arquitetura interna são: a máquina de avaliação de consultas, o módulo de acesso a metadados e o gerenciador de adaptadores.

A máquina de avaliação de consultas provê um ambiente para processamento de operações lógicas e físicas; nesse caso, para acesso e transferência de dados.

O módulo de acesso à metadados exporta meta-informações sobre os objetos publicados localmente. Meta-informações são obtidas a partir dos arquivos de configuração associados aos adaptadores, e complementadas, quando disponíveis, por informações obtidas das próprias fontes de dados, como, por exemplo, dados estatísticos sobre as tabelas em fontes de dados do tipo SGBD.

O gerenciador de *adaptadores* é responsável pela criação local de *adaptadores* e de *proxies* para acesso à adaptadores remotos. *Proxies* [GHJV95] implementam, do ponto de vista da máquina de execução, independência de acesso à adaptadores remotos. Apresentam a mesma interface de acesso a objetos locais, porém implementam comunicação remota com o servidor do *adaptador* remoto.

3.2.2 Publicação de dados

No sistema Le Select, considera-se como publicação a definição de uma fonte de dados (ou de um serviço) e sua disponibilização para acesso em consultas SQL ou em pedidos de execução de programas. A publicação de um novo tipo de fonte de dados é realizada através da criação, ou reaproveitamento, de um adaptador de dados e de seu arquivo de configuração. A funcionalidade implementada em um adaptador é atendida por um conjunto de interfaces Java, a serem implementadas pelo publicador, provendo acesso à fonte de dados e às suas respectivas meta-informações. Quanto ao seu papel no proceso de integração de dados heterogêneos, realiza o mapeamento da fonte de dados para o formato de tabela, caso a fonte não implemente o modelo relacional, bem como qualquer outro tipo de tratamento de homogeneização de dados.

Algumas decisões importantes devem ser tomadas durante a criação de um adaptador. Inicialmente, precisa-se definir o esquema da tabela relacional a ser publicada. Os atributos podem ser definidos com base em um dos domínios pré-definidos pelo Le Select ou um dos tipos de dado definido no padrão MIME. Em seguida, o publicador implementa os métodos *executeQuery(string)*, da classe *Adaptador*, e o método *next()*, na classe *WrapperResultSet*. A combinação da especificação desses dois métodos determina o modelo de acesso a fonte de dados. Em alguns casos (ex. banco de dados), os metadados podem ser obtidos diretamente a partir da fonte de dados. Em outros casos, o arquivo de configuração contém definições com meta-informações das fontes de dados correspondentes.

3.2.3 Publicação de serviços

Uma das características originais oferecidas pelo sistema Le Select é a possibilidade de publicação de serviços. Serviços, no contexto do sistema Le Select, são implementados por programas ou funções desenvolvidos pelo usuário em uma linguagem de programação tal como C ou FORTRAN. Existem dois modos de acionamento de serviços no sistema. O primeiro modo foi concebido para execução de serviços lentos[1] e para os quais pode-se adotar uma estratégia de execução assíncrona. A publicação de serviços assíncronos é obtida pela definição de um *adaptador de programa*. Sua interface implementa os seguintes controles de execução sobre o serviço: início da execução; armazenamento de estados intermediários; divulgação do estado da execução e obtenção de resultados. Serviços podem receber, como parâmetros, expressões relacionais e objetos de tipo básico, sendo responsabilidade do *adaptador de programa* adequar os parâmetros passados ao formato de dados de entrada do programa. O resultado da execução de um programa é publicado via um conjunto de *adaptadores de dados* temporários, criados pelo *adaptador de programa*.

O exemplo abaixo mostra um pedido de execução de serviço que recebe como parâmetro o resultado da execução de uma consulta. A execução dessa solicitação processa-se da seguinte forma: inicialmente, o Le Select recebedor da solicitação gera um identificador global de serviço, retornando-o ao cliente. Em seguida, a solicitação

[1] segundo critério do usuário

é redirecionada ao servidor de publicação do serviço requisitado. Uma vez acionado, o gerente de execução de serviços cria uma nova instância do objeto *serviço*, em uma nova linha de processamento *(thread)*, responsável por executar o pedido. Esse objeto solicita a execução da consulta fornecida como parâmetro de entrada ao programa, retornando o *conjunto resposta* ao *adaptador de programa*. O adaptador executa o programa iterando-o sobre as tuplas contidas no conjunto resposta.

Exemplo 3.2.1

JOB EXECUTE */programs/materialize*
input data set is
Select *pr.name,e.tipo-solo,e.ecosistema*
    From *www.inf.puc-rio.br/wrapprec/PrecReg pr,*
        www.inf.puc-rio.br/wrappenv/Env e
    Where *pr.name= e.region*

□

Arquivo de configuração

Um dos objetivos do sistema Le Select é fornecer um ambiente flexível para publicações. Parte dessa flexibilização é obtida utilizando-se arquivos de configuração de adaptadores. Arquivos de configuração são documentos, no formato XML, passados como parâmetros, durante a instanciação de adaptadores, e que contêm meta-informações sobre as fontes de dados.

O exemplo abaixo apresenta um arquivo de configuração para uma fonte de dados do tipo ASCII.

Exemplo 3.2.2

<Wrapper WrapperClass="LeSelect.Wrappers.Text.TableWrapperFactory">
    <Parameters>
        <Tablename="table1" file="examples/data/data1.txt">
            <Column name="Date" type="timestamp" size="8" / >
            <Column name="Value" type="string" size="9" / >
        < / Table>

< /*Parameters*>
< /*Wrapper*>

□

3.3 Avaliação de consultas

Nesta seção são apresentados os principais aspectos envolvidos no processo de avaliação de consultas pelo sistema Le Select. Após uma breve introdução sobre a linguagem de consultas adotada, prossegue-se descrevendo o processo de otimização. Por fim, discute-se alguns aspectos do mecanismo de execução.

Uma aplicação usuária do sistema Le Select submete consultas escritas em um sub-conjunto da linguagem SQL. Esse sub-conjunto inclue a cláusula *select-from-where* e exclue tratamento de agrupamento. Tabelas são referenciadas por identificadores formados pela composição *//nome do host/nome do adaptador/nome tabela*, onde *nome do host* é o nome da máquina hospedando o servidor Le Select publicador da fonte de dados; *nome do adaptador* especifica a classe implementando o acesso à fonte de dados e *nome tabela* identifica a tabela a ser acessada.

Exemplo 3.3.1 *(Consulta no Le Select)*

Select *date,value*
  From *//neemias.inf.puc-rio.br/exemplo/data1*

□

3.3.1 Otimização

Le Select adota a heuística de exploração do espaço de busca proposta pelo sistema R* [SA80], com extensões para o tratamento de árvores de topologia Bushy [CS99] e anotações sobre restrições de acesso às tabelas. Nas seções seguintes apresentam-se alguns aspectos importantes no processo de avaliação de consultas no sistema. Inicialmente, discute-se a estratégia de localização de predicados sobre objetos distribuídos.

Em seguida, apresentam-se algumas técnicas adotadas para suportar a integração de fontes de dados heterogêneas ao sistema de avaliação de consultas.

Localização de predicados

A localização de predicados binários baseia-se na estratégia adotada pelo sistema R^*. Dados dois sub-planos, R e S, e um predicado P, analisa-se o custo de avaliação de P nos seguintes sítios:

- no local onde se encontra R - ou seja, no local onde o resultado da sub-expressão R foi gerado;
- se R é uma relação de base (sob controle de um adaptador) - no sítio de R;
- no local onde se encontra S - ou seja, no local onde o resultado da sub-expressão S foi gerado;
- se S é uma relação de base (sob controle de um adaptador) - no sítio de S;
- no local da próxima operação - temporariamente indeterminado;

Planos que contenham predicados com localização indeterminada não podem ser eliminados, mas são mantidos até que seu custo possa ser avaliado; uma vez que a localização da próxima operação tenha sido determinada. A localização indeterminada de predicados aumenta o número de planos a serem mantidos, em benefício de uma cobertura mais completa do espaço de busca. Finalmente, a localização de predicados segue as restrições de capacidades exportadas pelos adaptadores. Em outras palavras, no caso de uma fonte não estar capacitada a executar junções, a localização de predicados em seu sítio não é avaliada.

Integração de fontes heterogêneas

Nesta seção apresentam-se dois aspectos do modelo de avaliação de consultas no sistema Le Select, especialmente adequados a consultas sobre fontes de dados heterogêneas. Primeiramente, apresenta-se o conceito de *capacidade relacional de fontes de dados*, e *padrões de acesso* à tabelas. Em seguida, apresenta-se o modelo de integração para funções do usuário.

Capacidades de fontes de dados

Fontes de dados heterogêneas apresentam diferentes níveis de implementação de operadores relacionais. SGBDs, por exemplo, implementam todas as operações relacionais empregadas em planos de execução gerados pelo Le Select. Por outro lado, uma fonte de dados correspondendo a um arquivo ASCII requer que a execução dos operadores relacionais sejam executados por um dos servidores Le Select, participantes da avaliação da consulta.

O conjunto de operações relacionais que uma fonte de dados está habilitada a executar é chamado de *capacidade relacional* da fonte de dados. No sistema Le Select os adaptadores de dados são responsáveis por *exportar* ao sistema, como meta-informação, a capacidade relacional da fonte de dados correspondente. A informação é divulgada a partir de um arquivo de configuração onde se define a habilidade da fonte de dados em executar as seguintes operações: filtros sobre os atributos das relações, junção entre tabelas, união de tabelas e avaliação de fórmulas aritméticas em projeções.

Com base nas habilidades relacionais das fontes de dados, consultas são reescritas em consultas globais de integração e sub-consultas enviadas às fontes de dados. Adicionalmente, contribuem para a determinação dos sub-planos a serem executados por cada servidor participante da avaliação da consulta.

Padrões de Acesso

Uma outra técnica de avaliação de consultas em sistemas heterogêneos, particularmente adequada ao acesso a dados em páginas *Web* e à execução de funções de usuário, é a modelagem de acesso a tabelas com base em padrões. Padrões de acesso [FMLS99] foram inicialmente criados para modelar o acesso a formulários HTML.

No ambiente *Web*, pesquisas sobre o conteúdo de sítios são direcionadas ao fornecimento de uma chave de busca. Assim, por exemplo, em um sítio como *YAHOO*, fornece-se uma palavra-chave para a realização da busca desejada. Como resposta, tem-se uma lista das ocorrências que atendem à expressão regular proposta como consulta. Verifica-se que o acesso aos dados é restrito a um modelo de fornecimento de palavras-chave, assim como o conteúdo do resultado restringe-se ao disponibilizado pelo ambiente de busca utilizado. Em suma, o acesso à base de dados acontece segundo restrições.

Nesse modelo, $T = ((A_1, \cdots, A_n), (B_1, \cdots, B_n))$ descreve o padrão de acesso de uma tabela T, onde o conjunto $(A_1, \cdots, A_n)$ determina os parâmetros obrigatórios para acesso a T e $(B_1, \cdots, B_n)$ correspondem aos atributos disponíveis ao acesso, uma vez que os obrigatórios tenham sido fornecidos.

Considere a tabela LIVRO=(autor, ano, título, editora) onde os valores dos atributos (ano, título, editora) são acessíveis mediante o fornecimento de um valor para o atributo ($autor$). A definição de padrão de acesso seria especificada como LIVRO= $((autor), (ano, título, editora))$. Uma consulta sobre essa tabela tomaria a seguinte forma:

Select ano, titulo, editora
From LIVRO
Where autor="Joao"

□

O modelo de representação de acesso à tabelas segundo um padrão foi estendido no sistema Le Select, incluindo também o acesso restrito à publicações de programas.

Funções de usuário

Le Select adota o modelo introduzido em [CGK89] para representação de funções de usuário em consultas. Naquele trabalho, funções definidas pelo usuário são modeladas como relações virtuais (ver 2.4.1). Retomando a discussão sobre padrões de acesso, a publicação de relações virtuais apresenta padrão de acesso definido pelos atributos de entrada da função, obrigatórios, e os produzidos por ela, considerados disponíveis.

Como exemplo, considere-se novamente a função de usuário *Pol*, com esquema definido por Pol(<mapaPoluição Blob> in, <polIndex float > out). A função *Pol* seria publicada como uma relação virtual segundo o esquema: Pol((mapaPoluição), (polIndex)). A consulta científica correspondente à apresentada no exemplo 2.3.1*(a)*, seria formulada como:

Select regnome,p.polIndex
 From PolReg pr, Pol p;
 Where p.mapaPoluicaof = pr.mapa

□

A modelagem de funções como tabelas traz um importante benefício ao processo de otimização de consultas, uma vez que homogeniza o tratamento de tabelas e funções de usuário.

Considerando-se a avaliação do espaço de busca de consultas utilizando-se o algoritmo de programação dinâmica [SACL+79], pode-se incluir o tratamento de funções de usuário no processo de avaliação de consultas sem modificação, onde o custo de execução da função é computado como o custo de acesso à tabela via um índice, por exemplo. Adicionalmente, uma vez que o processo de avaliação de consultas no sistema Le Select gera árvore de topologia *Bushy*, predicados sobre os reultados da avaliação de funções do usuário podem ser avaliados sobre as relações de base, antes de sua participação em junções, o que não era possível no sistema *LDL*, por considerar apenas árvore de topologia linear.

A modelagem de funções de usuário como relações aumenta o espaço de busca dos planos analisados de até $(m - n + 1)!/n!$ planos, onde m é o número de funções na consulta e n o número de relações. Esse número é reduzido considerando-se a assimetria da junção com relações virtuais e de planos semanticamente incorretos [FMLS99].

3.3.2 Execução de Planos

O plano distribuído de execução de consultas, gerado pelo otimizador do Le Select, é formado pela combinação de sub-planos de topologia *bushy*, enviados aos adaptadores, com um plano linear paralelo de fluxo de dados, que integra os resultados intermediários produzidos pelos sub-planos. A comunicação entre os operadores do plano linear se dá por meio da interface no modelo *iterador* [Grae93]. Adicionalmente, a comunicação entre sub-planos enviados a sítios distribuídos é realizada por uma versão do operador *Exchange* [Grae93].

Bind-Join

Uma das características inovadoras quanto a avaliação de consultas por sistemas de mediadores implementadas pelo Le Select é o operador de *Bind Join*, [FMLS99]. O operador modela uma variação da operação de junção onde o operando à esquerda é uma expressão relacional e à direita, uma relação com restrição de acesso.

Uma relação $R = a_1, a_2, \ldots, a_n$ édefinida como apresentando um conjunto $S = a_1, a_2, \ldots, a_m$ de atributos necessários ao acesso a seus dados e seu complemento $S\prime = a_{m+1}, \ldots, a_n$ de atributos livres que podem ser recuperados diretamente. Este modelo representa bem sítios de busca na Internet onde o usuário não dispõe de acesso indiscriminado a base de busca, precisando enviar algumas palavras chave *bind* para obter como resposta os atributos livres. Funções de usuário podem também ser modeladas através do operador bind-join, tendo os atributos de entrada a função como *necessários* e os atributos resposta como os de livre acesso.

O operador não apresenta as propriedades simétrica e associativa, encontradas na operação de junção.

A execução do operador processa-se segundo o modelo tupla-a-tupla, onde as tuplas da relação à esquerda (*outer*) são lidas e os valores correspondentes aos atributos obrigatórios da relação à direita (*inner*) são a ela enviados, um a um, juntamente com um identificador de tupla. A relação *inner* é acessada com o valor obrigatório, e as tuplas associadas, caso existam, são retornadas ao operador. Para cada uma das tuplas retornadas, a tupla correspondente ao valor avaliado compõe uma tupla resposta da junção. A figura 3.2 ilustra a execução do operador.

3.4 Sumário

Neste capítulo apresentou-se o sistema Le Select que é um sistema de intergação de dados e programas distribuídos. O sistema utiliza o modelo Relacional para integração de fontes de dados heterogêneas. Aplicações acessam os itens do acervo, publicados através de consultas na linguagem SQL. A publicação de funções de usuário no sistema pode ser realizada de duas maneiras: como relações virtuais, ou como serviços assíncronos. A primeira permite que se adote a estratégia de otimização de consultas distribuídas segundo a proposta do sistema R*. A operação de junção entre relações virtuais é realizada pelo operador *Bind Join*.

Este trabalho otimiza a avaliação de consultas com programas científicos propondo técnicas para implementação eficiente de chamadas à programas e estratégias para geração de planos paralelos para execução de programas distribuídos.

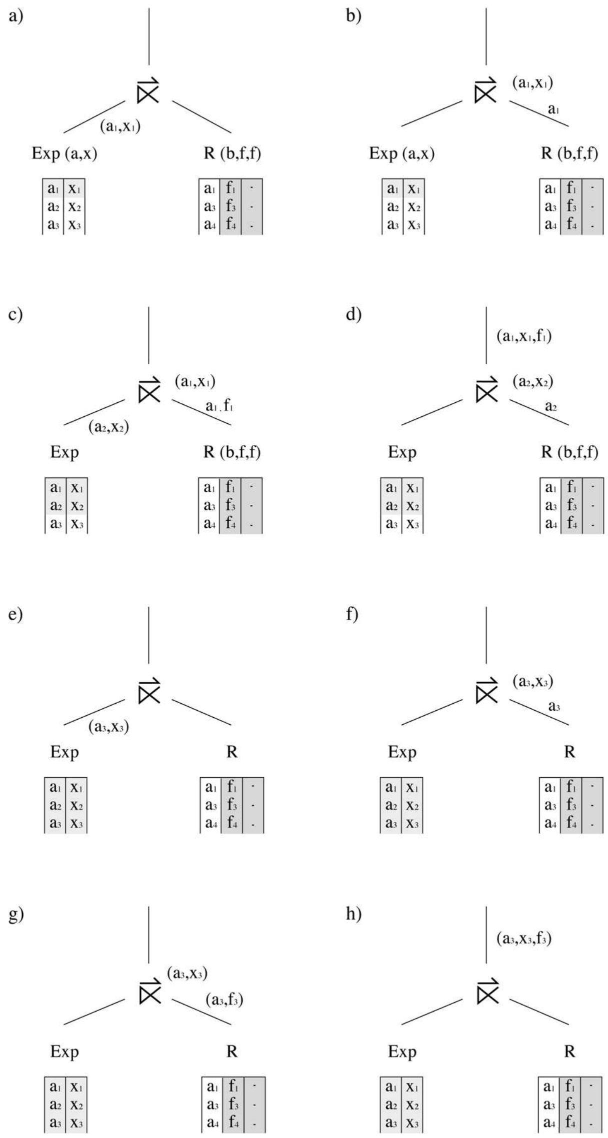


Figura 3.2: *Execução do operador Bind-Join*

Capítulo 4

Técnicas para execução eficiente de consultas científicas

4.1 Introdução

O Le Select foi concebido com o intuito de integrar aplicações científicas remotas, desenvolvidas autonomamente. Um de seus aspectos singulares está em lidar com tais programas dentro do contexto das próprias consultas. Atualmente, porém, isso é feito através de uma estratégia de execução ingênua.

Por exemplo, não raro um programa científico é aplicado sobre dados de uma relação intermediária gerada por uma junção. É sabido que a operação de junção pode gerar valores duplicados para atributos que, nas relações de entrada, eram únicos. Se isso ocorrer em um atributo *Blob*, o programa científico que o processa será invocado desnecessariamente sobre as duplicatas. Além disso, o LeSelect manipula *Blobs* exatamente da mesma forma como o faz com tipos de dados simples. Um *Blob* pode ter que atravessar vários sítios antes de chegar aquele responsável pela execução do programa.

Para evitar esses problemas, este trabalho desenvolve um conjunto de técnicas de execução que minimiza o número de chamadas a programas científicos, e reduz, ao mínimo, o custo de transferência de objetos do tipo *Blob*.

Este capítulo apresenta tais técnicas da seguinte forma. Inicialmente, propõe-se estender a álgebra relacional com operadores para execução de programas. Em seguida, discutem-se técnicas para utilização de tais operadores em consultas do tipo *SPJ* que também contenham chamadas a programas científicos. Por fim, sugere-se uma estratégia de implementação dessas técnicas no sistema Le Select.

4.2 Integração de programas em consultas

Uma maneira natural de invocar programas, no contexto de execução de consultas relacionais, é o de criar a possibilidade de fazê-lo através de operadores de álgebra específicos. No caso dos programas científicos, propõe-se dois operadores que atuam sempre de forma conjunta: o *Apply* e *GetBlob*. Enquanto o último se encarrega de obter, onde quer que esteja, o *Blob* a ser utilizado como parâmetro de entrada, o primeiro é responsável por invocar o programa científico sobre o *Blob* obtido.

4.2.1 O operador Apply

Para que se possa integrar a invocação de programas em consultas a seu processo de avaliação, propõe-se, inicialmente, estender a álgebra relacional com o operador *APPLY*. Formalmente, a execução de um programa P, sobre atributos de uma expressão relacional R, em uma consulta q, é modelada pelo operador **APPLY (R;X;Y;Z;P)**[1]; onde X é uma tupla produzida pela projeção de R sobre os atributos associados à entrada de P, em q; Y é uma tupla do mesmo tipo que a tupla produzida por P (i.e. *out*); e Z é uma lista de atributos de R propagados para a saída [2].

Seguindo-se o exemplo 2.3.1, a invocação do programa *Pol*, na consulta *(b)*, seria expressa pelo operador:

APPLY(Exp, <mapa Blob>;<polIndex float>;<estação int,temp float>; Pol), onde Exp é uma expressão relacional.

Graficamente, quando o operador aparecer em planos de execução de consulta, optar-se-á por repesentá-lo apenas com os parâmetros: nome do programa invocado

[1] A notação do operador segue a empregada na representação de funções de agregação em [Daya87]

[2] os valores dos atributos da variável de tupla, que não estão associados à entrada do programa, são copiados, sem alteração, para saída e concatenados à Y.

e lista de atributos em sua entrada; ficando implícitos os demais parâmetros. Quanto aos predicados sobre resultados da execução de programas, *Pol(p.mapa) > 1.5* por exemplo, não serão diferentes de outros predicados, sendo representados por seleções ou junções sobre o valor de um dos componentes da variável de resultado do programa.

4.2.2 O operador GetBlob

Invariavelmente, os programas científicos invocados pelo operador *Apply* processam imagens, matrizes e outros objetos, modelados como *Blobs*. A operação de captura desses Blobs é representada, no plano de execução de consultas distribuídas, pelo operador físico *GetBlob*. A operação é alocada no plano de execução, junto ao operador *APPLY*, cujo programa consome o *Blob* capturado. Semanticamente, o operador atua segundo o mesmo procedimento utilizado pelo conhecido programa FTP, *File Transfer Program*, transferindo o *Blob* do seu sítio de origem ao de execução do operador.

Para atender às particularidades de Blobs, propõe-se modelá-los como arquivos do sistema operacional, portando identificador único (*URI*) fornecido pelo adaptador de encapsulamento da fonte de dados. O *URI* é formado pelo endereço da máquina onde o Blob se encontra armazenado, seguido de uma referência a ser fornecida pelo adaptador. Fontes de dados compostas por atributos de tipo básico e Blobs, são mapeadas pelo adaptador correspondente em uma tabela cujo esquema compreende tipos de dados básicos e URIs (ver figura 4.1).

O tratamento dedicado a dados de tipo Blob, em consultas distribuídas, distingue-se daquele adotado para dados convencionais[3]. Verifica-se que, quanto à estrutura, Blobs são tipos *opacos*, do ponto de vista do processador de consultas. O que significa dizer que o processamento de valores de tipo Blob não se beneficia de características semânticas de seu conteúdo. Adicionalmente, na medida em que sua dimensão, em média, é extremamente maior que a de atributos de tipo básico, seu armazenamento ocorre externamente às tuplas a que pertence. Finalmente, o custo de transferência de Blobs pela rede, considerando-se consultas distribuídas, apresenta-se em ordem de grandeza superior aos dados de tipos básicos.

O operador *GetBlob* apresenta a interface: *GetBlob (URI,URI)*, onde os parâmetros

[3]de tipo de dado simples.

referenciam o identificador do Blob remoto, a ser capturado, e o respectivo identificador do arquivo local, gerado pelo armazenamento do Blob transferido.

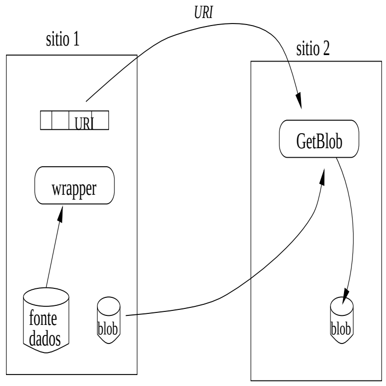


Figura 4.1: *Modelo de execução do operador GetBlob*

A figura 4.2 mostra um plano parcial, expresso através de uma árvore de execução que contém os operadores *Apply* e *GetBlob*.

Mediante esses operadores, nas seções seguintes, elaboram-se técnicas eficientes que os integram à estratégia de avaliação de consultas científicas distribuídas.

4.3 Técnicas para execução de consultas científicas

Uma implementação ingênua dos operadores *Apply* e *GetBlob* pode ser desastrosa para execução de consultas distribuídas. Tanto a chamada de programas quanto a transferência de *Blobs* são sujeitos aos problemas descritos a seguir.

4.3.1 Motivação

No exemplo apresentado em 1.2.1, têm-se três relações *PrecReg*, *PolReg* e *Env* contendo, respectivamente, dados de precipitações pluviométricas por região, dados de poluição regional e características ambientais. A primeira encontra-se publicada por um servidor Le Select localizado na PUC-Rio, Rio de Janeiro. A segunda foi publicada por um servidor Le Select situado na UNICAMP, em Campinas. A tabela

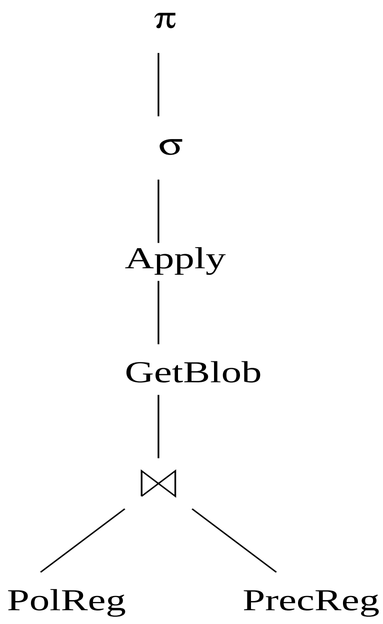


Figura 4.2: *Árvore de execução com APPLY e GetBlob*

Env está disponível em um servidor em Brasília. Adicionalmente, estão disponíveis dois programas científicos, *Prec* e *Pol*, publicados respectivamente, em Paris e em Atenas. A figura 4.3 traz uma representação da arquitetura de publicação, composta de servidores Le Select, e um cliente que, a partir de Brasília, deseja saber *quais são as regiões onde há uma correlação interessante entre índices de poluição e índices de umidade* (exemplo 4.3.1).

Exemplo 4.3.1

Select pr.nome, Prec(pr.prec),Pol(po.medidas),e.ecosistema
from PrecReg pr, PolReg po, Env e
where pr.nome=po.regnome and
pr.nome = e.região and
Prec(pr.prec) > 1.5 and
Pol(po.medidas) < 0.6

O esquema dos objetos componentes da aplicação encontra-se descrito a seguir:

Mapa Umidade - PrecReg(nome:string, prec: blob)
Indices Poluição - PolReg(regnome:string, mês:int, medidas:blob)

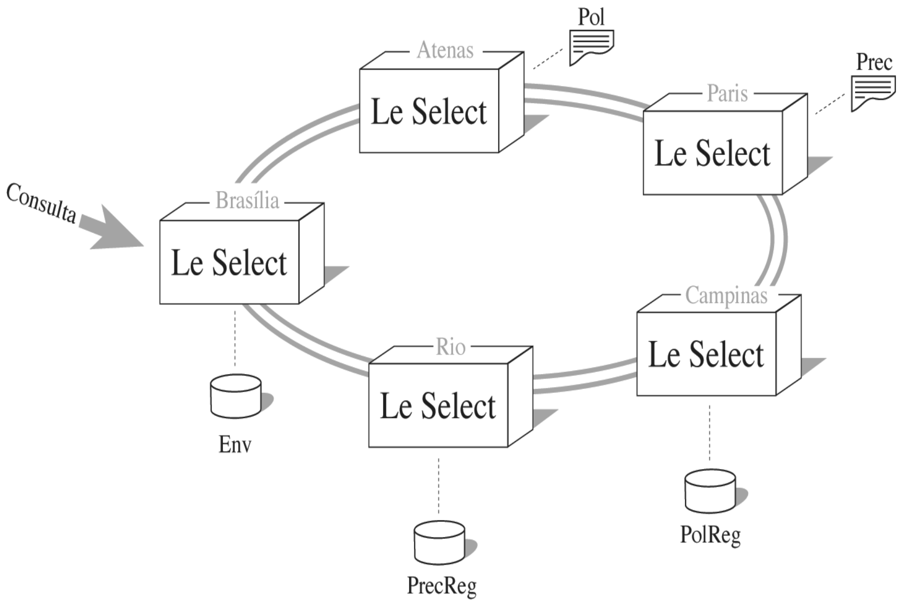


Figura 4.3: *Arquitetura de Publicação Le Select*

Info ambiental - Env(região:string, tipo-solo:int,vegetação:int, ecosistema:int)

Os dois programas científicos, Prec *e* Pol*, são publicados por meio de adaptadores de programas e apresentam as seguintes assinaturas:*

Prec((mapHumidity:blob);(indPrec:float));
Pol((matrix:blob);(polInd:float));

□

A figura 4.3.1 exprime possíveis estratégias para a consulta do exemplo acima. No que se segue, apresentam-se alguns problemas causados pela execução desses planos, que o desviariam do caminho para um custo minimal.

1. Invocação repetitiva de programas científicos

   Suponha-se que o plano de execução escolhido para a consulta em 4.3.1 seja o representado na figura 4.3.1 (a). A junção *1* associa uma tupla de *PrecReg* eventualmente com várias de PolReg (i.e. amostras de vários meses de uma

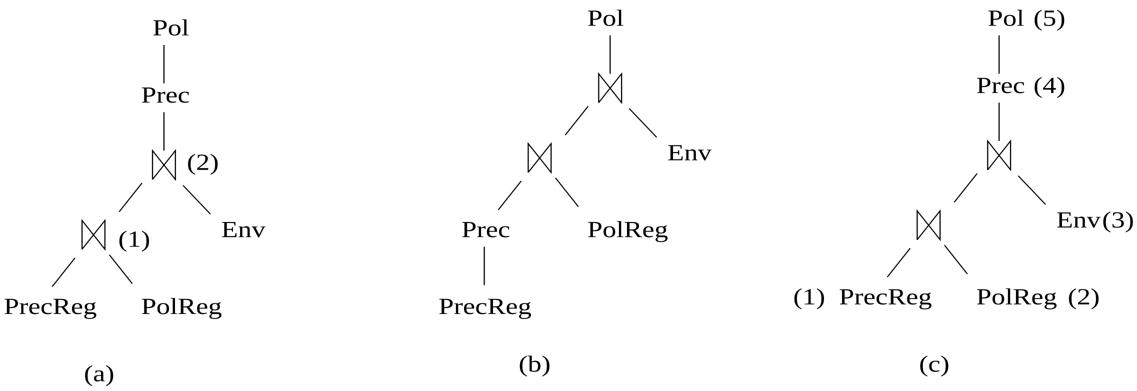


Figura 4.4: *Planos de execução de consultas distribuídas*

mesma região). Seguindo-se o plano de execução, o programa *Prec* é executado repetidamente sobre o mesmo Blob duplicado nas tuplas da relação resultante da junção em *1*. Em um outro exemplo, o próprio arquivo de entrada (i.e. PrecReg ou PolReg) pode apresentar o mesmo Blob, em diferentes tuplas, induzindo a repetidas invocações de programas sobre a mesma entrada.

2. Invocações desnecessárias de programas científicos

   Considere-se o plano de execução representado na figura 4.3.1(b). O programa *Prec* é executado sobre todas as imagens contidas no arquivo *PrecReg*. Considere-se, também, que algumas das regiões em *PrecReg* não tiveram amostras de poluição coletadas e, conseqüentemente, não aparecem no arquivo *PolReg*. Nessa hipótese, o programa *PrecReg* estará sendo, desnecessariamente, invocado sobre Blobs de tuplas que teriam sido eliminadas, caso a junção entre *PrecReg* e *PolReg* houvesse precedido a invocação do programa.

3. Transferência repetitiva de Blobs pela rede

   Considere-se, novamente, o plano de execução representado na figura 4.3.1 (c), onde a junção, entre *PrecReg e PolReg*, duplique as tuplas da primeira, como fruto do relacionamento n-ário entre as relações. Blobs referentes ao atributo *prec* sãoigualmente duplicados, sendo transferidos, repetitivamente, pela rede, durante sua trajetória no plano entre os sítios (1) e (4).

4. Transferência desnecessária de Blobs pela rede

Tome-se o plano de execução em 4.3.1(c). As tuplas da tabela *PolReg* são enviadas para os sítios *3 e 4*, antes de chegarem ao sítio *5*, onde se encontra o programa *Pol*. Os Blobs referentes ao atributo *medidas* teriam realizado uma enorme, e custosa, trajetória, desnecessariamente, uma vez que apenas são úteis no sítio final. Adicionalmente, tuplas podem ser excluídas por junções anteriores à invocação do programa, o que tornaria inútil a transferência dos respectivos Blobs.

A minimização do custo de consultas, que podem apresentar os problemas acima, é obtida pela combinação de técnicas de execução de programas científicos e de transferência de Blobs, conforme será apresentado no que se segue.

4.3.2 Premissas

Antes de se prosseguir na discussão, é importante definir alguns limites ao problema que o tornem tratável, embora sem distanciá-lo da realidade. Assumem-se, assim, as seguintes premissas:

- tratamento exclusivo de consultas SPJ (seleção, projeção e junção) conjuntivas;
- ligações de rede de mesma velocidade entre servidores Le Select distribuídos;
- ambientes computacionais com capacidade de processamento equivalentes, suportando os sistemas Le Select;
- custo de execução de programa científico em ordem de grandeza superior ao de operações relacionais (i.e. junção e seleção);
- custo de transferência de Blobs em ordem de grandeza superior ao referente a atributos de tipo básico;
- servidores Le Select com mesma capacidade de execução de operações relacionais;
- característica estática de programas científicos (i.e. não há *code shipping*);

Tendo-se estabelecido essas premissas, apresentam-se, nas próximas seções, as técnicas de execução.

4.3.3 Otimizando as transferências de Blobs

A primeira técnica discutida atende ao problema de *Transferência desnecessária de Blobs pela rede*, descrito anteriormente. Considerando-se um dos custos preponderantes na execução de consultas, aquele referente à transferência de Blobs, pretende-se conceber uma estratégia que postergue sua ativação para o ponto do plano de execução onde o Blob será consumido. Voltando ao exemplo da figura 4.3.1(c), a transferência do arquivo de medições *medidas* deve ser realizada apenas quando a respectiva tupla alcançar o sítio onde se encontra o operador *APPLY* que ativará o programa *Pol*.

De modo a evitar o acesso prematuro aos Blobs, os adaptadores "exportam", em seu lugar, os correspondentes URIs, como componentes das tuplas em cada tabela. Essa técnica é chamada de *Transferência Postergada*. Dessa forma, a tupla é processada como se contivesse apenas atributos de tipo simples, cujo custo de tranferência pela rede é considerado nulo [4]. As transferências de tuplas, entre sítios, para avaliação de predicados simples, apresenta custo desprezível em função dos tamanhos de atributos de tipos básicos [5]. Ao atingir o sítio de publicação, onde o respectivo operador GetBlob se encontra, o URI é utilizado pelo operador, para efetuar a transferência direta, entre o sítio fonte e o de seu consumo. Ao processá-lo, GetBlob recupera o Blob correspondente e o armazena localmente. A cópia local do Blob é identificada por um URI local, que servirá de referência para seu consumo. Vale enfatizar que, caso o programa científico espere ler sua entrada de um arquivo, o custo associado à materialização local torna-se nulo, sendo obrigatoriamente realizada, independentemente de sua captura.

A técnica de transferência postergada é importante, na medida em que influe no processo de otimização. Note-se que, sem o emprego dessa técnica, o plano de execução apresentado em 4.3.1 (c) pode não ser escolhido como o melhor pelo otimizador, uma vez que, ao seu custo, seria adicionado o da transferência repetitiva entre os sítios de Rio-Campinas, Campinas-Brasília e Brasília-Paris. Um segundo aspecto importante é a redução no custo de transferências de Blobs proporcionado pelos predicados simples precedentes. Os Blobs que fazem parte das tuplas não selecionadas, não chegam a ser transferidos.

[4] em verdade com custo desprezível em relação ao custo de transferência de Blobs.

[5] URI é de tipo string

4.3.4 Evitando processamento redundante

Analisando-se o esquema das relações *PolReg* e *PrecReg*, e o plano de execução de consulta em 4.3.1 *(c)*, verifica-se que a junção *1* pode apresentar seletividade parcial maior que 1, referente à tabela PrecReg , calculada, conforme [Hel98], como $\frac{||Out||}{||PrecReg||}$, onde *Out* é a relação resultante da junção. Visto que, para cada região em *PrecReg*, podem existir registros de poluição relativos a diversos meses. O reflexo imediato desse fato é o aumento no tempo de execução da consulta em virtude de transferências redundantes do mesmo Blob e da invocação, também redundante, da função *Prec*, sobre valores repetidos de URIs. Em outras palavras, planos de execução de consulta que apresentem predicados binários precedendo à invocação de programas podem produzir um custo maior de execução. Isso ocorre na medida em que valores distintos de atributos de entrada para programas podem ser duplicados após a avaliação do predicado de junção .

Esta situação tende a ocorrer, corriqueiramente, em junções entre tabelas com relacionamento de chave estrangeira. Além disso, atributos podem apresentar redundância natural. É o caso da mesma imagem referenciada em diferentes tuplas o que sugeriria o processamento desnecessário sobre as ocorrências múltiplas do valor repetido.

Na literatura encontram-se três técnicas para a solução do problema de redundância de dados na execução de procedimentos. Em [SACL+79], é proposto o uso de ordenaçãopara impedir a recomputação de sub-consultas correlacionadas. A idéia seria aproveitar a ordenação do atributo de referência na consulta principal, eliminando reexecuções de sub-consultas correlacionadas. Caso o atributo não estivesse ordenado, uma operação de ordenação poderia ser incluída pelo otimizador. A partir da entrada ordenada, cria-se um *cache* para armazenar a primeira ocorrência de cada valor distinto, bem como do resultado da avaliação da sub-consulta correlacionada. Valores redundantes não requerem computação da sub-consulta, acessando diretamente o resultado em *cache*.

Uma outra estratégia, vinda da área de linguagens de programação, é conhecida como *Memoization* [Mic68]. De modo a evitar a recomputação de expressões sobre um mesmo valor, uma tabela *hash*, em memória, armazena os parâmetros de entrada e o resultado da expressão. Obviamente, sendo limitado o espaço de memória, a estratégia apresentará problemas de performance, quando aquele for ultrapassado pelo volume formado pelos parâmetros de entrada e pelos resultados da expressão.

Para solucionar o problema de preenchimento de memória no algoritmo *Memoization*, [HN96] propôs-se o algoritmo denominado *Hybrid Cache*. A proposta deriva do algoritmo de *hybrid hash join* [DeWKOS+84]. O algoritmo opera em duas fases. Na primeira fase, tuplas são processadas sem materialização. Tuplas com valores correspondentes a chaves, armazenados na tabela *hashing*, capturam o valor associado (resultado da execução das operações GetBlob, ou Apply, sobre os parâmetros chave) e são enviadas para a saída. Novas tuplas têm seus valores-chave armazenados na tabela, e seguem para serem processadas por um dos operadores. A partir do momento em que a partição em memória torna-se completamente preenchida, tuplas com novas chaves são alocadas a um *buffer*, com entradas do tamanho da tupla, correspondendo a uma das partições da entrada. Uma vez que não haja mais tuplas a serem processadas, uma a uma, as partições são processadas, de forma idêntica à partição inicial.

O algoritmo apresenta resultados melhores que o algoritmo baseado em ordenação para funções com resultados de tipo de dado básico. O ganho de desempenho do algoritmo *Hybrid Cache* é explicado por utilizar espaço de memória proporcional ao número de valores distintos na entrada da função, enquanto a ordenação requer memória disponível em tamanho proporcional ao número de tuplas. Para funções que produzem Blobs, o algoritmo é menos eficaz que o de ordenação, na medida em que é obrigado a incluir o Blob na tabela *hash*, diminuindo o número de valores armazenáveis em memória, e aumentando o número de partições em disco.

Este trabalho adota uma variação do algoritmo *Hybrid Cache* para o armazenamento dos parâmetros de entrada e saída dos operadores *APPLY* e *GetBlob*, eliminando o processamento redundante sobre ambos.

O algoritmo geral de execução é :

```
for each r in input
    verifica (cache);
    if (not found)
       execute operator over r;
```

```
        update cache;
    endif
    generate output;
endfor
```

A adoção da estratégia de *caching*, no contexto de técnicas propostas, viabiliza várias otimizações sobre o algoritmo inicialmente proposto. Primeiramente, a combinação dos operadores GetBlob e Apply resolve a questão da alocação de memória na tabela *hash* para objetos do tipo Blob. Através da técnica de transferência postergada, Blobs capturados são armazenados localmente no sistema de arquivo e apenas o URI local é referenciado na tabela *hash*. O custo extra de gravação de Blobs é, na realidade, ofuscado pelo custo mais significativo incorrido na tranferência do Blob pela rede. Ainda que assim não o seja, muitos programas teriam sido preparados para ler arquivos do sistema operacional o que obrigaria sua materialização incondicional.

Adicionalmente, em alguns casos, obtêm-se ganhos com o compartilhamento de *cache* entre os operadores. Considerem-se, para esta análise, dois cenários de programas que processam Blobs: (I) programas com um parâmetro de entrada, sendo esse do tipo Blob; e (II) programas com parâmetros de tipo básico, além de um ou mais de tipo Blob. No primeiro caso, um único *cache* é necessário, armazenando o par (URI remoto, resultado do programa). O Blob capturado pode ser eliminado assim que for processado pelo programa; uma vez que tuplas com o mesmo URI não induzirão transferência, nem invocação de programas. Ao serem recebidas, as tuplas são imediatamente propagadas para o resultado da operação *APPLY*, obtendo-se o resultado do programa a partir do acesso ao *cache*. Note-se que a eliminação do Blob pode apresentar economia significativa, considerando-se o tamanho potencial desses arquivos. No segundo caso, há a necessidade de dois *caches* distintos, um para o GetBlob e outro para o Apply. O primeiro armazenará o par (URI remoto, URI local), enquanto o segundo armazenará a tupla de entrada, o URI remoto [6], e a tupla resultado (< ...,URI remoto,... >,<resultado>). Nesse último cenário, o Blob capturado precisa ser mantido em disco até o processamento da última tupla, em função da combinção com outros parâmetros de entrada do programa.

[6] o uso do URI remoto possibilita o processamento de tuplas duplicadas diretamente, por meio de acesso ao *cache*

A adoção conjunta das técnicas de transferência postergada e de *caching* permite otimizações interessantes em consultas com invocação composta de programas, localizados em sítios distintos. Considerem-se, por exemplo dois programas f e g, onde o Blob produzido por f é consumido por g. Uma vez que se dota os respectivos operadores *APPLY* de controle de *caching*, Blobs duplicados produzidos por f podem ser descartados imediatamente, enviando-se apenas a g o URI associado.

Finalmente, com relação ao plano de avaliação de consultas, a invocação de programas pode ocorrer tardiamente, beneficiando-a de possíveis reduções sobre valores distintos aplicados por predicados simples precedentes. A eventual duplicidade de valores, gerada pela avaliação anterior do predicado de junção, torna-se inteiramente inócua à transferência de Blobs e à invocação de programas, frente a técnica de *caching*.

4.3.5 Paralelizando a captura de Blobs com a execução de programas

O custo total de execução de programas é composto do custo de captura dos parâmetros de entrada e do custo de sua própria execução. Analisando-se os operadores *GetBlob* e *Apply*, verifica-se que apresentam características distintas quanto ao uso de recursos computacionais. O primeiro está baseado no processamento de Entrada e Saída, tanto na captura do Blob pela rede, quanto em seu armazenamento local; enquanto o segundo apresenta processamento baseado na utilização de CPU. A observação de tal circunstância motiva a exploração concorrente dos recursos computacionais, através da execução paralela entre a captura dos dados e a avaliação do programa.

A figura 4.5 apresenta o modelo de execução paralela, nos moldes do operador *Exchange* apresentado em [Grae93]. Linhas de execução separadas ativam a captura de Blobs pelos operadores GetBlob e a execução de programas pelo operador Apply. Comunicação entre os operadores é realizada por intermédio de uma fila, onde são armazenados os parâmetros de entrada do programa, incluindo-se os URIs locais. O ganho potencial desta estratégia é o mesmo obtido com processamento de E/S assíncrono, em uma máquina monoprocessada [MPT+94].

É interessante notar a simplicidade de implementação de paralelismo utilizando-se o operador *Getblob*, se comparada a outras estratégias como, por exemplo, o semi-join

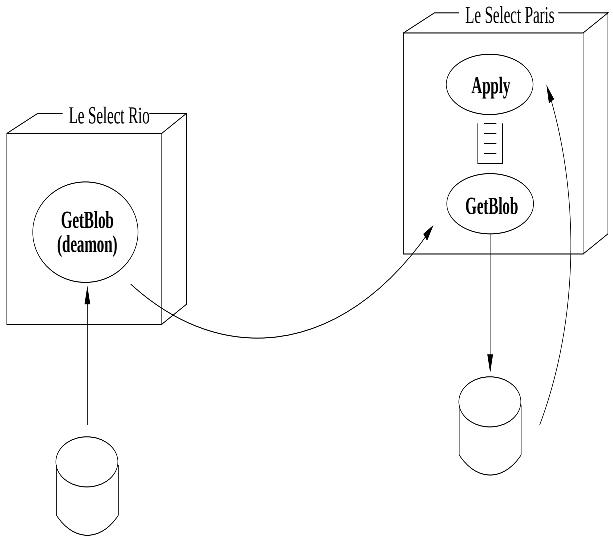


Figura 4.5: *Paralelismo entre a captura e ativação de programas*

que requereria complexa otimização paralela.

4.4 Integrando-se as técnicas - Adaptador Configurado de Programa

O sistema Le Select disponibiliza serviços publicados por meio de adaptadores de programas, como apresentado na seção 3.2.3. Seu projeto foi desenvolvido com o objetivo de suportar a execução de um programa lento por vez, assincronamente. Para sua execuçãomaterializa-se a expressão relacional que forneça os dados necessários, e processa-se o programa sobre cada uma de suas tuplas. Do ponto de vista de otimização de consultas, a proposta implementa a estratégia *Pull-Up*, com materialização de resultados intermediários.

Quanto à execução de funções[7], adaptadores de dados modelam funções como relações virtuais, segundo a proposta em [CGK89]. Dessa forma, funções participam do processo de otimização tradicional, com custo de acesso sendo expresso por seu custo médio de execuçãopor tupla.

[7] no Le Select, refere-se a programas rápidos

Nesta seção, apresenta-se o *wrapper configurado de programas* (WCP). Sua arquitetura compreende módulos que implementam as técnicas de execução descritas nas seções precedentes que garantem uma execução otimizada de programas científicos.

Na figura 4.6, apresenta-se a arquitetura geral do WCP. Elipses, na figura, representam componentes necessários na integração das técnicas de execução. As operações indexadas de D executam como *deamons*, consumindo sua entrada, assincronamente, a partir de uma fila. Os demais componentes são indexados com a letra I, indicando que implementam a interface *iterador*:(open, next, close) [Grae93].

O operador *APPLY* é o único componente do WCP a ser implementado pelo publicador. Sua função é invocar o programa, adaptando a tupla de entrada à interface do programa. Ao término do programa, WCP constrói a tupla de saída, composta dos atributos propagados e a tupla resultado.

O componente *GB* implementa a funcionalidade do operador *GetBlob*. A arquitetura permite a criação de mais de uma instância do operador, obtendo a transferência, em paralelo de Blobs a partir de sítios distintos. Cada instância acessa uma fila contendo os Blobs de sua fonte de dados. O *caching* é compartilhado pelas instâncias do operador, onde o controle de concorrência é assegurado utilizando-se de procedimentos sincronizados [Tane95].

Além das operações *Apply* e *GetBlob*, os seguintes componentes completam a arquitetura:

- Check Cache e Update Cache implementam a técnica de *caching* na invocação de programas. O componente *Check Cache*, sendo o primeiro a processar as tuplas de entrada, garante a eficiência no tratamento de duplicados, caso em que a tupla resultado é rapidamente depositada na fila de saída. O componente *Update Cache* recebe a tupla resultante da avaliação do programa e atualiza o *cache* correspondente. Note-se que o componente é obrigado a acessar o *cache* de Blobs, uma vez que precisa das URIs remotas, enquanto a tupla resultado contém apenas URIs locais. Essa é a razão pela qual optou-se por ter uma única estrutura de *cache* compartilhada pelas diversas instâncias do operador *GetBlob*.

- Split e Merge são meta-operadores [Grae93] responsáveis pela partição (respectivamente composição) das tuplas em filas de Blobs e de atributos propagados.

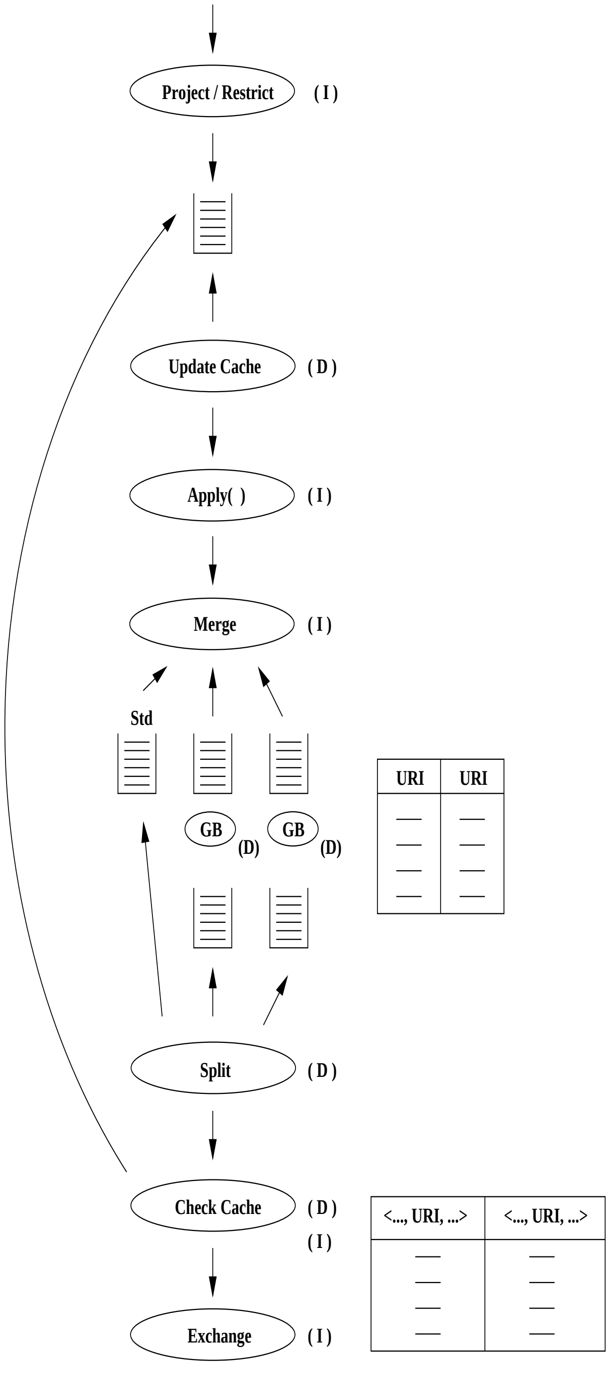


Figura 4.6: *Wrapper Configurado de Programa*

As entrada nas filas armazenam a identificação da tupla a que pertencem e, para cada valor, o atributo correspondente.

- Project e Restrict permitem ganhos de desempenho da arquitetura executando restrições e projeções sobre o resultado da avaliação do programa, na medida em que resultados não selecionados podem ser imediatamente desprezados, sendo importante para resultados do tipo Blob.

O WCP provê uma arquitetura otimizada de execução de programas ao sistema Le Select a partir da implementação das técnicas de execução propostas. A combinação dessas técnicas flexibiliza a alocação do oprador *Appy* no PECD, de tal forma que podem-se adotar heurísticas que reduzam ao mínimo o custo de sua avaliação.

4.5 Sumário

Neste capítulo, apresentaram-se as propostas de um conjunto de técnicas para a otimização e avaliação de consultas com programas científicos, que objetivam minimizar o tempo de resposta a consultas distribuídas, ao mesmo tempo em que maximizam a utilização de recursos distribuídos.

Inicialmente, apresentaram-se dois novos operadores, *APPLY* e *GetBlob*, responsáveis pela representação, no plano de avaliação de consultas, de invocações de programas e de transferências de Blobs pela rede, respectivamente. Em seguida, apresentaram-se três técnicas de execução: Transferência Postergada, *Caching*, Paralelismo de captura de Blobs com invocação de programas. Essas três técnicas foram reunidas para compor a arquiteura do *Wrapper Configurado de Programa*.

No capítulo seguinte apresenta-se a estratégia para geração de planos para a execução de programas científicos, em paralelo.

Capítulo 5

Estratégias para a execução paralela de consultas em bancos de dados científicos distribuídos

5.1 Introdução

No capítulo anterior apresentou-se um conjunto de técnicas que minimizam o custo de avaliação de consultas distribuídas com programas científicos, na medida em que eliminam custos extras na transferência de *Blobs* e no acionamento de programas. O conjunto de técnicas foi integrado no módulo *wrapper configurado de programa*, garantindo a sua adoção em publicações realizadas pelos usuários do sistema mediador.

Neste capítulo, discutem-se e apresentam-se estratégias para geração de planos de execução de consultas que minimizam seu tempo total e maximizam a alocação de recursos distribuídos. As estratégias consideram que o posicionamento do operador *Apply* no plano de execução pode ser realizado de maneira a proporcionar o melhor desempenho possível, na medida em que as técnicas de execução eliminam os efeitos de eventuais custos adicionais.

5.1.1 Benefício das técnicas de execução

Além de minimizarem o custo de execução , as técnicas propostas no capítulo anterior são, também, extremamente importantes ao processo de otimização de consultas, uma vez que eliminam as restrições à localização do operador *APPLY* no plano de execução. Lembrando-se que a meta a alcançar está centrada na redução do número de acionamentos de programas e, igualmente, na minimização do custo de transferência de *Blobs*, a localização do operador *APPLY* na árvore de execução está diretamente relacionada à variação no custo do plano de execuçãoe, consequentemente, no alcance da meta estabelecida. Alguns exemplos de aplicação das técnicas ilustram seus benefícios para o processo de otimização:

$*$ a adoção de *caching* possibilita a localização do operador *APPLY* após a avaliação de predicados simples (seleções e junções), beneficiando a execução da consulta com a possível eliminação de valores distintos, ao mesmo tempo que evita acionamentos repetitivos frutos de valores de entrada duplicados por junções. Logo, pode-se alocar as operações *Apply* acima de operações de junção sem aumento no custo do plano.

$*$ a técnica de *Transferência Postergada* elimina do processo de otimização de consultas a avaliação de planos quanto à distribuição de *Blobs*. Note-se que a geração do espaço de busca conforme realizado tradicionalmente, como no *System R**, analisa três possibilidades de localização para uma operação binária, respectivamente, considerando sua avaliação nos sítios de cada operando, e, também, no sítio da operação seguinte. Com a estratégia de *transferência postergada*, *Blobs* são diretamente transferidos entre seu sítio de origem e o de destino, minimizando totalmente o custo de transferência. Logo, o espaço de busca de planos a serem analisados pelo otimizador não é alterado pela necessidade de transferência de *Blobs*. Adicionalmente, a técnica *TP* faz com que o custo de execução de um plano seja independente da distância, no plano de execução, entre a fonte e o destino de *Blobs*.

Com base nessa discussão, verifica-se que as técnicas de execução são vantajosas, na medida em que reduzem o custo operacional associado à execução de programas científicos e, adicionalmente, permitem que o operador *Apply* seja alocado o mais tardiamente possível no plano de execução, sem que isso acarrete em custo adicional.

5.1.2 A heurística Pull-Up

Retomando a questão do custo principal de avaliação de consultas com programas científicos, centrado no custo de transferência de *Blobs* pela rede e no de acionamento de programas, repetidamente, sobre as tuplas da relação, deseja-se determinar uma estratégia de alocação para programas científicos, no *PECD*, que minimize o custo total de execução de consultas.

Ora, quanto à transferência de *Blobs*, considerando-se a adoção da técnica de *transferência postergada*, e a premissa de taxas de transferência iguais entre sítios, mostrou-se que obtém-se um custo fixo para sua execução, que é independente da posição de sua alocação no *PECD*. Logo, considerando-se a alocação da operação GetBlob localmente à chamada do programa consumidor dos *Blobs*, resta determinar a alocação desse último no *PECD*.

Minimizar o custo de iteração de programas científicos sobre um conjunto de tuplas, durante a execução de uma consulta, considerando-se programas do tipo *caixa-preta*, corresponde a minimizar o número de iterações. Como mostrou-se, a técnica de *caching* reduz a invocação do programa aos valores distintos de sua entrada. Logo, a minimização da execução total do programa, durante o processamento de consultas, dá-se com a redução máxima no número de valores distintos de sua entrada.

A estratégia de alocação de programas no *PECD*, chamada *Pull-Up*, posiciona-os após todos os predicados simples (seleção, e junção) da consulta.

Note-se que a heurística *Pull-Up* é análoga à adotada em sistemas como INGRES, onde as operações de junção são sempre alocadas acima das operações de seleção, localizadas, heuristicamente, próximas às respectivas relações. Naquele sistema, considera-se que o custo de avaliação de operações de junção é extremamente superior ao respectivo custo de seleções, o que torna inútil a análise de ordenação entre as operações de junção e seleção. No contexto científico, o custo por tupla de acionamento de programas é enormemente superior ao custo por tupla de execução de junções, donde deriva-se heurística análoga a adotado no sistema INGRES.

A heurística *Pull-Up* foi apresentada, inicialmente, por [Hel98] para o problema de consultas com predicados caros em ambiente centralizado. Este trabalho adota a heurística para consultas com programas caros que participem de predicados ou da projeção final. Adicionalmente, incorpora-se o tratamento de distribui ção e de processamento de *Blobs*.

Do ponto de vista de execução de consultas científicas essa estratégia usa os predicados simples para reduzirem, ao máximo, o número de valores distintos de atributos nas entradas de programas, minimizando o número de suas invocações. Do ponto de vista de otimização de consultas, a heurística divide o espaço de busca em duas partes. A primeira, composta de predicados simples, é explorada segundo algoritmos tradicionais [SACL+79]. A segunda parte compreende chamadas de programas e projeções. A geração de plano de execução para essa parte será tratada na próxima seção.

5.2 Estratégia para geração de PEC para consultas com um programa científico

A geração de um *PECD* para consultas científicas com apenas um programa é trivial. Seguindo-se a heurística *Pull-Up*, o acionamento do programa deve ocorrer após avaliação de todos os predicados simples. Sendo assim, toma-se o grafo representativo da consulta científica e retira-se a invocação do programa. Otimiza-se o grafo reduzido utilizando-se estratégia tradicional de otimização de consultas distribuídas [SA80]. Obtido o *PECD*, aloca-se o operador *APPLY* acima do último predicado barato. O plano gerado garante a minimização do custo de avaliação de programas, uma vez que minimiza o número de valores distintos sobre os quais o programa científico será iterado.

Nesta seção apresentou-se a estratégia de geração de planos de execução distribuídos para consultas científicas compostas de um único programa. Na próxima seção, estende-se a estratégia para o tratamento de mais de um programa distribuído pelos sítios Le Select.

5.3 Gerando *PECD* com mais de um programa

Antes de continuar a discussão sobre a geração de planos de execução para consultas distribuídas com mais de um programa científico, é importante relembrar a arquitetura de distribuição considerada neste trabalho.

Supõe-se uma arquitetura distribuída de sistemas de mediadores, como a implementada pelo sistema Le Select (ver seção 3.2). Cada sítio, participante de um ambiente de publicação Le Select, possui uma cópia do servidor com capacidade de execução completa quanto aos operadores relacionais.

Um plano de consulta distribuído é particionado em diversos sub-planos. Cada sub-plano é enviado ao sistema Le Select publicador dos objetos nele (sub-plano) referenciados. A execução global segue o modelo *iterador* [Grae93], realizando paralelismo de fluxo de dados entre os sítios participantes, conforme ilustrado na figura 5.1.

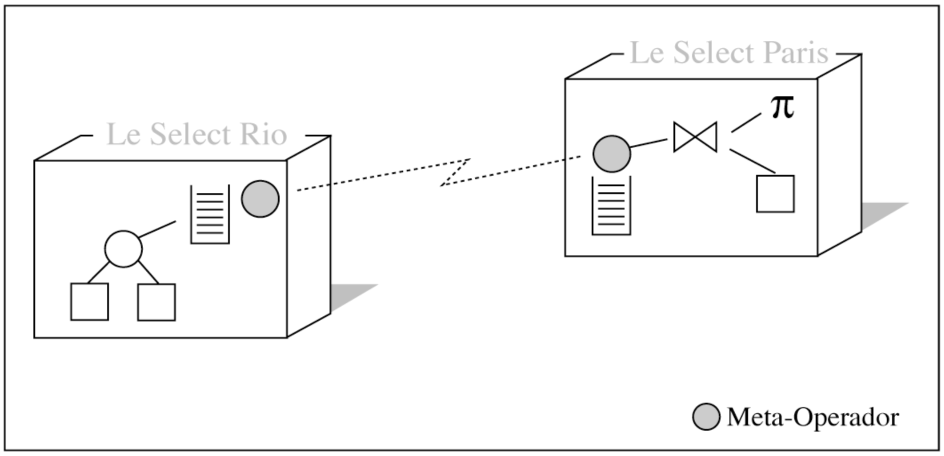


Figura 5.1: *Execução distribuída de consultas no Le Select*

Tendo-se estabelecido a arquitetura distribuída de execução, implementada pelo sistema Le Select, passa-se à discussão sobre consultas com invocação de mais de um programa científico.

O problema a ser tratado é definir uma estratégia de execução paralela para programas distribuídos que obedeça às restrições impostas por dependências de dados, expressas no grafo de consulta [1] e minimize o tempo total de sua execução . Três aspectos deste problema são determinantes: *distribuição, dependência de dados e restrições sobre resultados.*

Exemplo 5.3.1

Select *Prec(Cut(pr.prec))*
    From *PrecReg pr, PolReg p;*
    Where *pr.nome =p.regnome*
                (I)

Select *Prec(pr.prec), Pol(p.medidas)*
    From *PrecReg pr,Pol p*
    Where *pr.nome =p.regnome*
                (II)

Select *pr.nome, e.ecosistema*
    From *PrecReg pr, PolReg p, Env e*
    Where *pr.nome=p.regnome and*
        pr.nome=e.região and
        Pol(p.medidas) $<$ *0.6 and*
        Prec(pr.prec) $>$ 1.5
                (III)

□

O exemplo 5.3.1 apresenta 3 consultas onde, em cada uma, um cenário para análise de estratégias de execução paralela de programas é representado.

5.3.1 Programas dependentes

A consulta (I) do exemplo 5.3.1 apresenta dois programas, *Cut* e *Prec*, onde o resultado do primeiro é consumido pelo segundo. Esse modelo de consultas, representado pelo grafo de consulta da figura 2.6 *(a)*, determina uma seqüência de avaliação entre os dois programas baseada em sua inter-dependência de dados. Nesse contexto, uma

[1] ver 2.3.3

estratégia de execução puramente sequencial[2] atenderia ao requisito de dependência de dados, porém não proporcionaria utilização máxima dos recursos computacionais distribuídos. Uma estratégia de execução baseada no paralelismo independente, por sua vez, fere a relação de dependência. Logo, no cenário de programas aninhados distribuídos, a estratégia de execução que minimiza o tempo total da consulta e atende à relação de dependência de dados é o paralelismo de fluxo de dados entre *Cut* e *Prec*, ver um possível plano de execução na figura 5.3 *II*. O custo de execução do plano é estimado pela seguinte função de custo [3]:

$* \ Custo_{//pipeline} = t_0 + t_1 + n * MAX(cp_1, cp_2)$, onde $t_1 = cp_1$, para $cp_1 <= cp_2$, ou $t_1 = cp_2$, para $cp_1 > cp_2$, e cp_1 e cp_2 iguais aos custos de execução por tupla de *Cut* e *Prec*, respectivamente. Finalmente, t_0 é o tempo gasto na consulta para que uma tupla chegue ao primeiro programa.

5.3.2 Programas independentes

Na consulta apresentada em 5.3.1 (II), os programas *Prec* e *Pol*, localizados nos sítios de Paris e Atenas, participam da lista de projeção final. Os programas recuperam seus dados de entrada, independentemente um do outro, a partir da expressão relacional que os precede, no *PECD*.

Nesse cenário, uma estratégia de execução linear causaria sincronismos desnecessários entre suas execuções. Na figura 5.2, representa-se o custo de três execuções paralelas dos programas P1 e P2, onde o eixo dos X representa o tempo e o eixo dos Y, o custo total de execução. Na coluna da esquerda, modela-se a execução com paralelismo de fluxo de dados, enquanto, na coluna à direita, a mesma execução é representada adotando-se uma estratégia de paralelismo independente. As áreas hachuradas determinam o acréscimo do custo total da consulta introduzido pela linearização da execução de programas independentes. Dependendo da razão de custo entre os programas, a diferença no custo total de avaliação da consulta corresponde ao tempo de *subida* da primeira tupla, entre o primeiro e o segundo operador *APPLY*, ou ao tempo de *descida* da última tupla.

[2]com materializações totais entre cada operador

[3]nestes exemplos considera-se o custo de captura de Blobs compondo o custo de execução dos programas

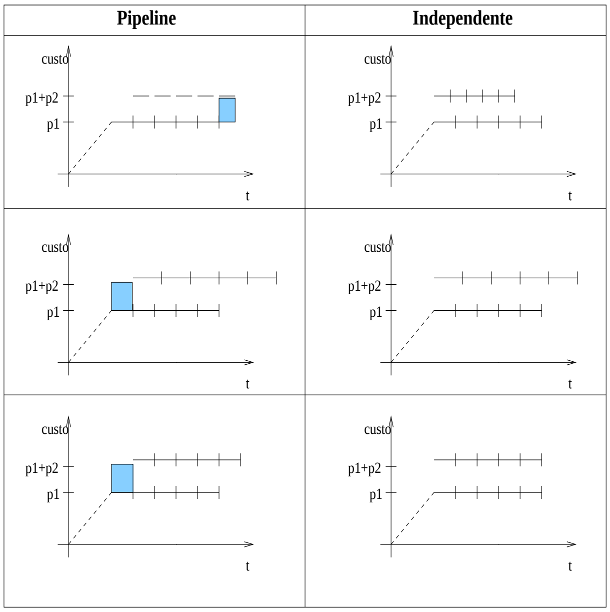


Figura 5.2: *Modelos para execução paralela de programas independentes*

Supondo-se a independência entre as execuções de programas, sua localização distribuída e obedecendo-se à estratégia *Pull-Up*[4], deriva-se a heurística de geração do *PECD* com avaliação paralela independente dos programas. Executando-se os programas de forma independente elimina-se o custo de sincronismo entre seus resultados. O custo estimado do plano, considerando-se um fluxo constante de dados entre os operadores (i.e não há inatividade de operadores), pode ser resumido pela expressão:

$* \ Custo_{//indep.} = t_0 + n * MAX(cp_1, cp_2)$, onde t_0 é o tempo gasto na consulta para que uma tupla chegue ao primeiro programa, cp_1 *e* cp_2 são os custos de avaliação de cada um dos programas e n é o número de tuplas na expressão relacional após a avaliação dos predicados. A figura 5.3 *(I)* mostra um possível plano de execução para a consulta.

[4]uma estratégia inversa seria propor o paralelismo independente junto às fontes de dados, o que incorreria em execuções desnecessárias, ver 4.3

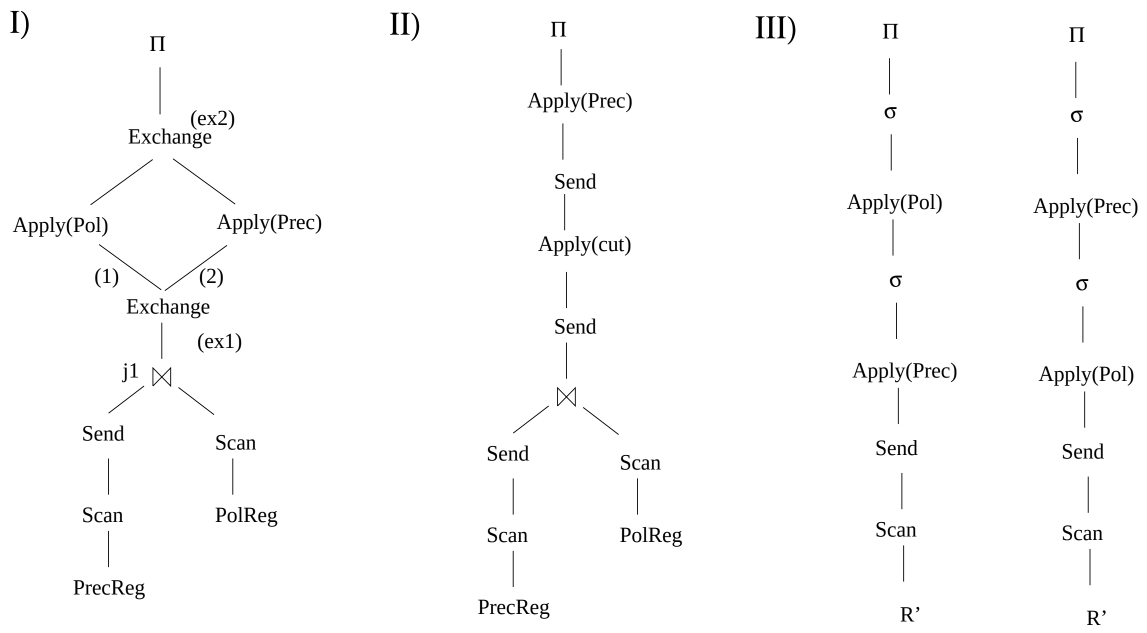


Figura 5.3: *Execução Paralela de Programas Científicos*

5.3.3 Predicados caros

O terceiro cenário a ser analisado é o de consultas com predicados sobre os resultados de programas, como na consulta do exemplo 5.3.1 (III). Nesse caso, apesar de não haver uma ordem de execução explícita, como ocorre em consultas com programas aninhados, a execução dos programas seguindo uma estratégia de paralelismo de fluxo de dados atende à estratégia global de minimização do número de invocações de programas. Usando essa heurística, predicados caros beneficiam-se das reduções infligidas pelo predicados que o precedem no fluxo de dados. No entanto, a seqüência de avaliação de predicados que proporcionará o menor custo total precisa ser definida.

Importância da ordem de avaliação de predicados caros

A importância da sequenciação da avaliação de programas com predicados pode ser verificada detalhando-se a execução da consulta 5.3.1 (III). Considere-se que os predicados simples foram avaliados e tem-se uma relação intermediária $R\prime$, fruto da expressão relacional correspondente. Considerem-se, ainda, os custos de avaliação de programas, $cp_{Prec} = 1$ e $cp_{Pol} = 5$, e que, adicionalmente, são estimadas, com base

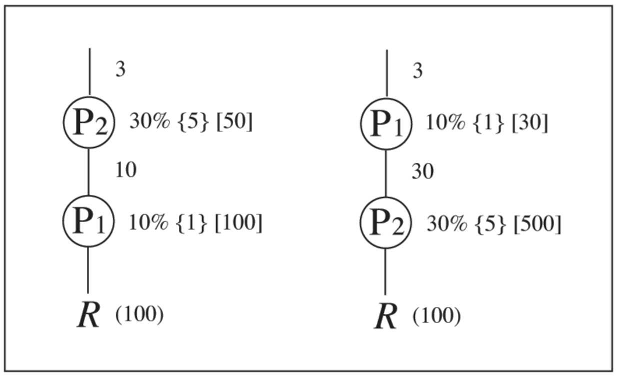


Figura 5.4: *Custo da execução em paralelo de predicados caros*

em execuções passadas, as seletividades dos predicados, $s_{Prec} = 0.1$ e $s_{Pol} = 0.3$. Por fim, seja *100* a cardinalidade de *R'*. O plano de execução paralela, com estratégia de fluxo de dados, tendo como ordem *Prec* $\Rightarrow$ *Pol*, tem custo 105 segundos, sendo gastos 100 seg. no processamento das 100 tupla por *Prec* e 5 seg. no processamento da última tupla por *Pol*. Já a seqüência inversa apresentaria custo de 501 seg. (ver figura 5.4). Portanto, apesar de serem equivalentes, os dois planos apresentam custos distintos, devendo-se identificar aquele cuja execução irá minimizar a função de custo estabelecida.

Na seção seguinte, com base nas heurísticas para alocação de programas científicos em plano de execução de consultas distribuídas, complementa-se a estratégia de geração do *PECD*.

5.3.4 Estratégia para geração do plano de execução de consultas científicas distribuídas

Com base na discussão apresentada na seção anterior, pode-se definir o plano ideal paralelo, para consultas distribuídas com programas científicos, como aquele que adota os diferentes modelos de execução paralela inter-operador, obedecendo à dependência de dados entre programas, e que *maximize a utilização de recursos* e *minimize o tempo total de execução da consulta*. Logo, a adoção de paralelismo independente, por si só, não é ideal, na medida em que reduções promovidas por predicados sobre resultados de programas não são aproveitadas. Tão pouco é suficiente a utilização de paralelismo de fluxo de dados, uma vez que produz um custo adicional para programas sem dependência de dados, além de permitir a inatividade de recursos em função de variações nas taxas de fluxo de dados.

A solução para este problema está centrada na combinação adequada de cada uma das técnicas discutidas. Nesse sentido, a estratégia para geração de um *PECD* que atende aos objetivos acima é obtida aplicando-se as seguintes heurísticas:

1. Otimizar a consulta, subtraída das invocações de programas científicos, por meio de algoritmos tradicionais.
2. Posicionar os operadores *APPLY*, no PEC, o mais tardiamente possível (*Pull Up*).
3. Utilizar a estratégia de paralelismo independente, para avaliação de programas sem predicados sobre seus resultados [5].
4. Adotar paralelismo de fluxo de dados para programas aninhados.
5. Aplicar paralelismo de fluxo de dados entre programas com predicados sobre seus resultados, onde a seqüência ótima de execução dos predicados precisa ser estabelecida.

Os quatro primeiros itens podem ser implementados em otimizadores distribuídos tradicionais sem muita dificuldade. Na fase de reescrita, o grafo de consulta é particionado, formando um sub-grafo composto de relações de base associadas através de seus predicados simples, e um segundo sub-grafo contendo a relação intermediária, fruto da avaliação dos predicados simples, e os nós de programas, associados através de arestas direcionadas [6]. O primeiro sub-grafo é enviado ao otimizador para geração do *PECD*. O plano gerado ordena e localiza a avaliação de predicados simples. Note-se que, nessa fase, *Blobs* são substituídos por seus URIs o que, somado à extração de programas, transforma a consulta científica em uma consulta distribuída tradicional. O próximo passo complementa o *PECD* com tratamento de paralelismo entre programas.

[5] Note-se que a execução de programas independentes deve seguir ao último predicado caro - benefício de reduções.

[6] lembrando que neste trabalho não se está analisando consultas com programas participando como operandos de junções

Na próxima sub-seção detalha-se o procedimento para complementação do *PECD* com a inclusão de paralelismo entre programas sem predicados sobre seus resultados. Em seguida, na sub-seção 5.3.6 discute-se a ordenação estática de programas com predicados. Por fim, apresenta-se as estratégias de execução *Híbrida e Picking Cherry*, que exploram características dinâmicas de execuções de consulta para minimizar o seu tempo total de avaliação.

5.3.5 Paralelismo entre programas sem predicados

Programas sem predicados podem apresentar dois tipos de relacionamentos, expressos claramente no grafo de consulta: independentes e dependentes.

Programas dependentes

A geração de plano para programas com dependência de dados segue estritamente a estratégia de paralelismo de fluxo de dados. Sua inserção no *PECD* parcial resume-se na alocação dos respectivos *Wrapper Configurados de Programas* na seqüência especificada pelo grafo de consulta e de operadores *Exchange* para realização de transferências entre sítios. [7]

Programas Independentes

Programas sem relacionamento de dependência de dados podem ser executados inteiramente em paralelo. O desafio, nesse cenário, é integrar a estratégia de paralelismo independente entre os programas com a estratégia de fluxo de dados entre os demais operadores da *PECD*.

A figura 5.3 (I) mostra um possível *PECD* para a consulta do exemplo 5.3.1 *(I)*. Note-se que a árvore de execução conta com dois ramos independentes, *1* e *2*, limitados por meta-operadores *Exchange*.

Sendo assim, o operador identificado como *ex1* na figura 5.3 (I), comunica-se através da interface *iterador* com os operadores *GetBlob* (representado no plano) e

[7] note-se que, mesmo na parte do *PECD* composta de predicados simples, a transferência entre sítios é também realizada pelo meta-operador *Exchange*.

com a junção em *j1*, enquanto, internamente, o módulo produtor comunica-se com dois consumidores distintos, um em cada ramo. Reciprocamente, o módulo consumidor em *ex2*, comunica-se com dois produtores, advindos dos ramos independentes, enquanto oferece a interface no modelo *iterador* à operação de projeção final e a cada um dos adapatadores que implementam o operador *APPLY*.

Duas funcionalidades adicionais são introduzidas no meta-operador. Para o operador indentificado na figura como *ex1*, inclui-se a tarefa de particionamento da tupla, de acordo com a interface dos programas. Simetricamente, o operados *ex2*, realiza a recomposição de tuplas.

A relação entre o caching e o paralelismo

Na sub-seção 4.3.4 apresentou-se três algoritmos de *caching* capazes de eliminar o processamento redundante de programas e de transferência de *Blobs*. A estratégia nesse trabalho adota mecanismos de *caching* implementados como parte do *wrapper configurado de programa.*

Para *PECDs* inteiramente baseados em fluxo de dados, o algoritmo de ordenação, para tratamento de duplicados, produz uma indesejável materialização total de resultados intermediários, quebrando o fluxo de dados e afetando o sincronismo natural tupla-a-tupla entre os operadores. Nesse caso, o algoritmo *Hybrid cache* apresenta uma estratégia, em duas fases, mais atraente, na medida em que simula o processamento tupla-a-tupla, apesar de ocorrerem materilializações das partições cujas tuplas não tenham cabido na partição *0*.

Para execuções que exijam a materialização de partições, o algoritmo *hybrid cache* produz tuplas em sua saída segundo uma ordenação que pode não ser a mesma que a de entrada, igualando-se, nesse aspecto, ao algoritmo de ordenação. Voltando ao plano com avaliação independente de programas, em ramos diferentes do *PECD*, a mudança na ordem das entradas, em cada ramo, impacta na reformulação da tupla após a avaliação dos programas. Note-se que o operador *Exchange* (ex2) recebe parte da tupla de cada ramo recompondo-as em uma tupla de saída.

O problema a tratar é o de recompor tuplas cujas partições são recebidas desordenadamente pelo operador *Exchange* ao final do ramo de paralelismo independente.

Adota-se a seguinte solução para o problema. O operador *Exchange* em *ex1* insere um atributo extra em cada partição de tupla contendo o identificador de tupla. O operador *Exchange* em *ex2* executa uma junção entre os resultados em cada ramo independente. Como particularidade, tem-se uma junção com fator de seletividade igual a *1*. Na implementação desse trabalho optou-se pelo algoritmo de *double-pipelined hash join* [IFFLW99] para essa junção, sabendo-se que tuplas juntadas podem ser eliminadas das tabela *hash*, uma vez que o relacionamento entre as pseudo-relações é de cardinalidade $1 - 1$.

É interessante notar que o tratamento de paralelismo está inteiramente encapsulado nos meta-operadores *Exchange*, o que simplifica bastante a implementação dos demais operadores. Em particular, os *wrappers configurados de programa* podem ser gerados uma única vez e combinados em planos paralelos independentes, dependentes e de fluxo de dados.

Nesta seção, discutiu-se a implementação de estratégias paralelas entre programas sem predicados. Na próxima seção, apresenta-se o tratamento de ordenação de predicados caros.

5.3.6 Programas com predicados sobre seus resultados

Para completar a geração do PEC distribuído precisa-se definir um algoritmo eficiente para ordenação de predicados caros. Na literatura encontram-se algoritmos que estendem a técnica tradicional de geração de planos estáticos através de programação dinâmica. O algoritmo *Predicate Migration*, proposto por Hellerstein em [Hel94], e um aperfeiçoamento desenvolvido por Chaudhuri em [CS99], além do projeto *LDL* [CGK89] formam o conjunto dos trabalhos mais importantes quanto à extensão do algoritmo de otimização estática tradicional para suportar execução de predicados caros em sistemas centralizados. Adicionalmente, em uma fase inicial deste trabalho, propôs-se a ordenação baseada num algoritmo exaustivo de varredura do espaço de busca formado pelos predicados caros. Esses algoritmos foram apresentados em detalhe na seção 2.4. Nessa sub-seção fazem-se algumas considerações que atestam sua ineficácia no contexto distribuído de publicações na Internet.

Fragilidade da ordenação estática de predicados caros

O processo de otimização estática de consultas, baseada em custo, objetiva a geração de um plano de execução de consultas que minimize uma dada função de custo. A instanciação da função de custo em uma expressão de custo para um dado plano é produzida a partir de meta-informações estatísticas sobre os dados e sobre o sistema alvos da consulta. Quanto aos dados, estimam-se as cardinalidades de tabelas envolvidas, número de valores distintos em atributos, valores mínimo e máximo por domínio, seletividades de predicados e, no contexto científico, o custo de execução de programas por tupla, entre outros. Quanto ao sistema, incluem-se diferentes custos de entrada/saída, disponibilidade de memória e CPU, além de custos associados à tranferência de objetos pela rede, no caso distribuído. Esses parâmetros são sabidamente difíceis de serem apurados [AH00, AFTU96, BFMV00] em ambientes distribuídos e heterogêneos. Porém, para o caso de consultas científicas, as conseqüências de suas variações em tempo de execução são graves; uma vez que afetam a alocação de recursos, com eventual inatividade de servidores, o que produz aumento do tempo total de execução de consultas. No que se segue, apresentam-se algumas considerações sobre a variação na estimativa dos parâmetros relevantes em consultas científicas:

- Custo de execução de programas por tupla pode ser definido pelo publicador ou calculado pelo adaptador, com base no histórico de execuções. O custo real de execução, porém, depende da carga na máquina em tempo de execução. Adicionalmente, algoritmos de ordenação de predicados assumem custo constante de execução de programas o que não ocorre com programas com custo dependente da entrada, como é o caso de alguns programas de tratamento de imagem.

- Seletividade de predicado é normalmente computada a partir de estatísticas detalhadas sobre as relações envolvidas (por ex. histogramas). Em predicados sobre resultado de programas, porém, os valores de comparação não existem pois ainda serão computados. Ainda que se queira armazenar os resultados, o que coincidiria com a materialização dos resultados do programa, a variedade dos dados de entrada a serem enviados para o processamento, uma vez que o programa tenha sido publicado na internet, torna o domínio de valores de entrada completamente desconhecido. Apenas em alguns caso, estimativas grosseiras podem ser feitas para programas que produzam valores sobre um domínio bem estabelecido. Como exemplo, suponha-se um programa que identifique a cor com maior percentual de presença em uma imagem. Neste caso, caso se considere um universo de 10 cores, teria-se uma seletividade de 10% para o predicado de igualdade, *PercCor(imagem)='Azul'*. De uma maneira geral, a estimativa de seletividade de predicados sobre o resultado de programas são, hoje em dia, extremamente imprecisas, comprometendo processos em cujo valor sejam baseados.

- Número de valores distintos em atributos. Uma vez que se utiliza *caching*, o custo de execução de programas é proporcional ao número de valores distintos nas tuplas a servirem de entrada para o programa. Porém, na medida em que junções anteriores possam produzir valores duplicados e/ou, eliminar valores distintos, a estimativa do número de valores distintos na relação de entrada ao programa tende à imprecisão.

Essas considerações apontam contrariamente à ordenação estática de predicados caros, definida como função do custo de execução de programa e de seu fator de seletividade, como é o caso dos algoritmos baseados no *rank order* e no modelo de custo para uma busca exaustiva. Observe-se que a predição incorreta da melhor ordem de predicados pode aumentar dramaticamente o tempo de resposta da consulta (ver o exemplo 5.3.3).

Influência de valores distintos e seletividade

Ainda que se obtenha uma ordem correta de avaliação dos predicados caros, variações no fluxo de dados entre programas podem aumentar, consideravelmente, o tempo de resposta. Como exemplo desse problema, suponham-se dois predicados caros P e Q, com custo por tupla de *50* e *10* segundos, respectivamente. Uma vez que P é mais lento que Q, sua avaliação na ordem $P \Rightarrow Q$ resulta na inatividade de Q. Porém, a ocorrência de valores duplicados em tuplas da entrada de P podem minimizar o impacto sobre Q, na medida em que, para esses valores, o custo da avaliação de P é mínimo (apenas uma consulta ao *cache*). O que se observa nesse exemplo mostra que a utilização de recursos para o problema P-Q depende da distribuição dos valores de entrada pelas tuplas e da seletividade de P.

Tais variações no fluxo de dados entre predicados são bastante difíceis de serem observadas teoricamente, de modo que implementou-se um gerador de dados e um sistema distribuído paralelo de fluxo de dados, com o qual pôde-se comprovar a importância dessas anomalias sobre o tempo de resposta e a utilização de recursos computacionais. No que se segue apresenta-se a plataforma do experimento.

Quantificando o efeito de variações nas estimativas estáticas

Deseja-se medir o tempo de execução de uma cadeia de predicados executados segundo a estratégia paralela de fluxo de dados e se observar a influência de valores duplicados e da seletividade de predicados sobre o tempo de resposta. Um gerador de dados produz uma relação, composta de valores randomicamente computados que obedecem a uma estimativa do número de valores distintos por atributo e de sua cardinalidade. A relação produzida apresenta as seguintes propriedades: (i) atributos são independentes (não há dependência funcional); (ii) número de valores distintos é gerado randomicamente; (iii) a distribuição de dados é randômica. A relação gerada apresenta *n+1* atributos, onde n coincide com o número de predicados e um atributo é chave. Para o paralelismo de fluxo de dados entre P *e* Q, gerou-se uma relação R com 3-uplas $(\underline{K}, A_p, A_q)$, onde A_p é o parâmetro de entrada para P e A_q, o para Q. Para simular o processamento paralelo de fluxo de dados entre P e Q, implementaram-se dois processos executando em máquinas distintas. O primeiro processo recebe tuplas de R, simula o tratamento de *caching* e o processamento de programa e envia tuplas selecionadas para o predicado seguinte.

Para se avaliar os resultados da execução, tome-se uma das execuções onde a relação de entrada contém 200 tuplas. Atributo A_p apresenta 40 valores distintos enquanto A_q 160 valores distintos. As seletividades de P e Q são estimadas em 40% e 60%, respectivamente. Executou-se o fluxo de dados $P \Rightarrow Q$ com vários arquivos, gerados segundos os mesmos parâmetros. Uma vez que os custo de P e Q são de 50 e 10 segundos (respectivamente), uma estimativa para o tempo de resposta total, assumindo a inexistência de problemas de sincronismo, seria de 2010 segundos, correspondendo a execução de 40 tuplas por P, mais a última tupla processada por Q (uma vez que a avaliação por Q das demais tuplas selecionadas por P seriam encobertas pela avaliação de P).

O tempo de execução conferido nos experimentos é muito maior, 2520 segundos. A figura 5.5 mostra, para cada tupla processada, o tempo em que P foi avaliado (t_{pi}), e o mesmo para Q, onde i é o número da tupla (no eixo X). A explicação para a diferença

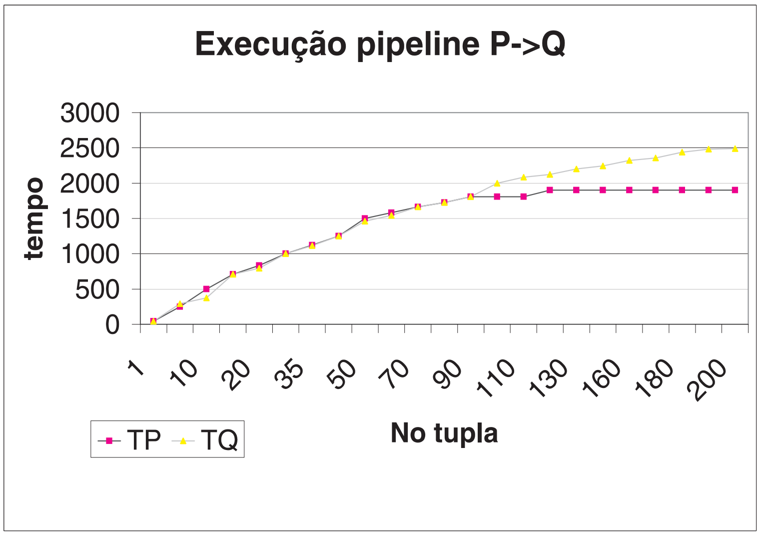


Figura 5.5: *Efeitos da seletividade e valores distintos sobre pipeline*

constatada entre o tempo estimado e o tempo real computado está, exatamante, na existência de valores duplicados. Ao iniciar-se a execução, há uma propabilidade de 100% de que P processe um valor novo (o *cache* encontra-se vazio!!!). Na medida em que o processamento progride, a probabilidade de precessar um valor distinto diminui. No gráfico, verifica-se que, após processar um terço das tuplas, 90% dos valores distintos para P foram processados. Logo, na primeira parte do gráfico, Q está sempre esperando por P ($t_{qi} = t_{pi} + 10, para\ i < 90$). A partir do momento em que a probabilidade de se obter um valor distinto torna-se baixa, Q é sobrecarregado por tuplas processadas muito rapidamente por P, uma vez que correspondem a valores duplicados.

Esse cenário, observado para vários arquivos gerados, ilustra o problema de sincronismo que pode ocorrer durante a execução paralela com fluxo de dados. No pior caso, a execução paralela pode apresentar um tempo de resposta próximo ao de uma execução sequencial, no caso em que, na primeira parte da execução as tuplas apresentam apenas valores distintos para o primeiro predicado e duplicados para o segundo e, inversamente, na segunda parte da execução.

5.4 A estratégia híbrida

Considerando-se as observações na seção precedente, pode-se indagar se a estratégia paralela de fluxo de dados é a mais adequada para predicados caros. A ordem de execução de predicados que minimiza o custo de execução da consulta é difícil de ser prevista, uma vez que os custos de execução de programas e as seletividades de predicados são, imprecisamente, estimados. Por outro lado, o sincronismo entre as avaliações dos predicados é sujeito às variações, durante a execução, tanto no número de valores duplicados em sua entrada, quanto no fator seletividade previsto. A adoção de paralelismo independente, por sua vez, poderia maximizar a utilização de recursos, uma vez que não requeriria sincronismo entre a avaliação dos predicados. Porém, segundo o modelo de execução apresentado, o tempo de resposta da consulta tenderia a aumentar, na medida em que não se beneficiaria de possíveis reduções efetuadas entre predicados caros.

Em vista desse contexto, deseja-se definir uma estratégia de execução para predicados caros que apresente as seguintes qualidades:

- Maximize o uso de recursos distribuídos.
- Beneficie-se de reduções entre predicados caros.
- Adapte-se, dinamicamente, às variações de estimativas computadas estaticamente.
- Integre-se ao *PECD*.

Este trabalha apresenta a estratégia *híbrida*, onde se executam, segundo o modelo de paralelismo independente, ramos da árvore de execução compostos por uma cadeia de predicados executados segundo modelo de paralelismo de fluxo de dados. Em cada ramo, uma ordem de execução de predicados é exercitada, sendo que o ramo apresentando a ordem escolhida como a mais benéfica estaticamente (mediante o emprego das técnicas baseadas em ordenação por *rank* ou busca exaustiva) é dotado de prioridade maior que os demais ramos concorrentes.

A proposta é ilustrada por meio de uma consulta q que apresenta os predicados P e Q sobre atributos de uma relação R. Considere-se que a seqüência de predicados que proporciona o menor tempo de resposta, segundo uma análise estática, foi computada como $P \Rightarrow Q$. A estratégia híbrida produz um plano de execução paralelo para q cujo custo corresponde à $Maximo \mid PQ, QP \mid$.

A implementação da estratégia híbrida dá-se pela criação de duas instâncias de P, P_1 e P_2, no sítio de P, e duas instâncias de Q, Q_1 e Q_2, no sítio de Q, ver figura 5.6. Tuplas de R são consumidas por P_1 ou Q_2. As tuplas consumidas por P_1 são enviadas a Q_1, e as consumidas por Q_2 são enviadas a P_2. Uma operação de *União* recupera os resultados de cada um dos ramos. Adicionalmente, define-se P_1 com maior prioridade que P_2, no sítio de P, e Q_1 com maior prioridade que Q_2, no sítio de Q, fazendo com que o fluxo $P_1 \Rightarrow Q_1$ tenha prioridade sobre o fluxo $Q_2 \Rightarrow P_2$.

O ganho proporcionado pela estratégia híbrida está na ocupação total de recursos distribuídos frente aos problemas de sincronismo entre os predicados caros sequenciados segundo um modelo de execução paralelo de fluxo de dados. Quando P_1 deixa de passar tuplas à Q_1, por simples regra de escalonamento de linhas de execução, Q_2 torna-se ativo lendo tuplas de R e acionando o ramo $Q_2 \Rightarrow P_2$. Dessa forma, a distribuição de tuplas de R pelos ramos paralelos independentes processa-se dinamicamente, em função da carga alocada a cada predicado.

Uma vez que P_1 e P_2 (resp. Q_1 e Q_2) compartilham o *cache*, a execução de duas instâncias de cada predicado caro, na mesma máquina, não proporciona aumento de custo significativo. Ao contrário, a estratégia elimina inatividade no sistema, maximizando o uso de recursos distribuídos e reduzindo o tempo de resposta total de consultas, beneficiando-se de eliminações de tuplas entre predicados caros.

As variações durante a execução, das estimativas computadas estaticamente têm seu efeito reduzido sobre o tempo total da consulta, visto que não induzem à inatividade de componentes do sistema.

Esta abordagem assemelha-se ao *Competition Model* [AZ96] que propõe a execução concorrente de vários PECs referentes à mesma consulta. Nesse modelo, após um determinado intervalo de tempo, todos os planos são eliminados, exceto aquele que tenha se mostrado o melhor. Na estratégia híbrida, nenhum ramo é eliminado, na medida em que todos realizam trabalhos úteis.

A generalização da estratégia para consultas com n predicados caros é obtida gerando o plano híbrido com n ramos concorrentes. A maximização de recursos é

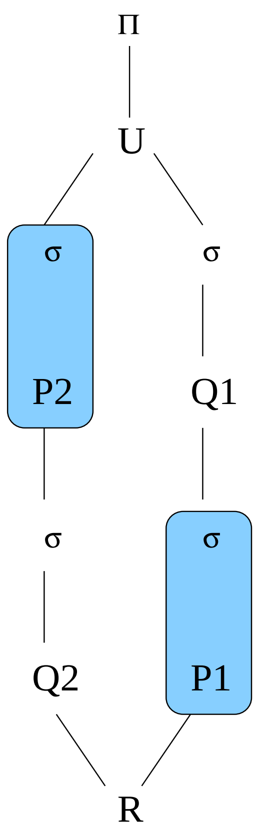


Figura 5.6: *Paralelismo Híbrido*

garantida com cada um dos diferentes predicados aparecendo em um ramo como o primeiro a ser avaliado. O ramo suportando a seqüência escolhida como a mais benéfica tem as instâncias participantes recebendo a mais alta prioridade. Nos demais ramos, a partir do primeiro predicado, a ordem não é importante, uma vez que a utilização de recursos já está garantida e a ordenação de predicados, apesar de poder ser realizada, não trará benefícios significativos. É importante mencionar que dificilmente uma consulta científica envolverá mais que três predicados caros.

A estratégia para geração do plano paralelo, conforme descrito na seção 5.3.4, é complementada no item *5* com o seguinte procedimento:

1. Gerar estaticamente, com base em busca exaustiva ou em *rank order*, a seqüência de predicados mais benéfica;

2. Permutar a ordem de predicados de maneira a obter ramos de fluxo de dados concorrentes. Sujeito à restrição de que as cadeias de predicados, em cada ramo paralelo independente, iniciem-se por diferentes predicados;

3. Definir prioridade mais alta à melhor seqüência e prioridade igual para os demais ramos [8];

Nesta seção apresentou-se a estratégia de execução híbrida que proporciona estabilidade à execução de predicados caros, segundo o modelo paralelo de fluxo de dados. A proposta utiliza um princípio original para geração do *PECD* que combina as estratégias paralelas independente e de fluxo de dados. Tal combinação garante o benefício de reduções entre predicados caros, ao mesmo tempo que atenua significativamente o impacto de variações dos parâmetros estimados estaticamente sobre o tempo de execução de consultas. A estratégia é particularmente importante para aplicações distribuídas na internet onde tais variações ocorrem frequentemente.

Na próxima seção, apresenta-se uma abordagem diferente para avaliação de predicados caros, baseada no aproveitamento de relacionamentos de dependência funcional entre os atributos de entrada de predicados caros.

[8]tratamentos mais precisos de prioridade para os ramos secundários podem ser alvo de outros estudos

5.5 A abordagem Picking Cherry

Nas seções precedentes apresentou-se uma estratégia para execução paralela de programas científicos distribuídos. Para a execução de predicados caros, desenvolveu-se a estratégia híbrida que reduz significativamente o impacto sobre o tempo de resposta de consultas, causado por variações, durante a execução, das previsões estáticas. Nessa seção será apresentado a abordagem *Picking Cherry* (PC) para execução de predicados caros. A abordagem explora a dependência de dados entre atributos, normalmente ignorada por algoritmos de otimização de consulta tradicionais.

5.5.1 Beneficiando-se da dependência de dados

A intuição da abordagem PC está na observação de que o tempo de resposta total de consultas com predicados caros é minimizado, se predicados forem avaliados sobre valores identificados que promovam a maior redução no número de tuplas, caso o resultado de sua avaliação seja falso.

Para ilustrar a intuição motivadora da abordagem *PC*, considere-se novamente a consulta do exemplo 5.3.1 (III), reproduzida abaixo.

Exemplo 5.5.1

Select *pr.nome, e.ecosistema*
    From *PrecReg pr, PolReg p, Env e*
    Where *pr.nome=p.regnome and*
        pr.nome=e.região and
        Pol(p.medidas) $<$ *0.6 and*
        Prec(pr.prec) $>$ 1.5

□

As relações *PrecReg* e *PolReg* apresentam relacionamentos n-ário, expresso pelos atributos *nome* e *regnome*, respectivamente. Seja uma relação *R'*, fruto da junção entre as duas relações, conforme a tabela no exemplo 5.5.2 . O relacionamento entre os atributos é expresso na ocorrência de valores duplicados dos atributos de *PrecReg*, nome e prec, em *R'*. A região *R1*, por exemplo, aparece associada a duas medições de poluição, enquanto que a região *R2* é duplicada em sua associação com cinco medições de índices de poluição.

Suponha-se que, em uma execução da consulta 5.5.1, os predicados sobre os resultados de *Prec* e *Pol* apresentam seletividade diferente de 1 e custo de execução por tupla iguais a 2 e 1 segundos, respectivamente.

Observando-se a relação *R'*, verifica-se que a validação do predicado *Prec* tende a proporcionar resultado mais eficiente que o de *Pol*.

Exemplo 5.5.2

nome	prec	medidas
R1	*b1*	*r1m1*
R1	*b1*	*r1m4*
R2	*b2*	*r2m1*
R2	*b2*	*r2m2*
R2	*b2*	*r2m3*
R2	*b2*	*r2m4*
R2	*b2*	*r2m5*

□

Considere-se, por exemplo, o caso em que *Prec(b1)=falso* e *Prec(b2)=falso*. O custo de avaliação da consulta seria de 4 segundos, ao passo que a execução de *Pol* sobre todos os 5 valores de *medidas* custaria 5 segundos. Adicionalmente, observa-se que a avaliação de *Prec* sobre *b2* é mais benéfica à redução do tempo de resposta do que sobre *b1*, uma vez que, para o caso em que avaliar *falso*, o primeiro eliminaria 5 tuplas, contra 2 da avaliação de *b1*. Ainda que as tuplas em que *b1* participe tenham que ser avaliadas obrigatoriamente, pode-se minimizar o tempo total prevalecendo-se da execução de *b2* sobre a de *b1*. Pode-se, neste caso, optar-se por executar em paralelo *Pol* sobre *r1m1* e *r1m4*. O custo mínimo deste escalonamento paralelo seria de 2 segundos.

Tais observações sugerem que os valores distintos de atributos de entrada a predicados caros contribuem de forma desigual para a redução do tempo de resposta total da consulta.

5.5.2 Definição do Problema

Nessa seção, define-se formalmente o problema tratado na abordagem *PC*. Suponha-se uma relação R, contendo os atributos [a,b], onde os valores em $R.a$ e $R.b$ não sejam necessariamente únicos. Têm-se dois programas científicos *F1 e F2* que recebem, como parâmetro de entrada, em uma consulta q, os valores dos atributos $R.a$ e $R.b$, respectivamente. Considerem-se o conjunto $S1 = \{a_1, a_2, ..., a_n\}$ composto dos valores distintos de $R.a$ e o conjunto $S2 = \{b_1, b_2, ..., b_m\}$, compreendendo os valores distintos em $R.b$. Define-se um grafo bi-partido $BG = \{VS1, VS2, E\}$, onde *VSn* contem vértices correspondendo a elementos em *Sn*, e E representa arestas ligando, nas duas direções, elementos de $VS1$ com elementos de $VS2$. Um par de valores (a_i, b_j) é ligado por uma aresta, se ele ocorre em uma tupla de R. Processar um vértice significa executar um programa sobre o valor representado pelo vértice, e obter-se um valor *booleano*. Se o valor obtido é falso, eliminam-se todas as arestas associadas ao vértice. Caso seja verdadeiro, eliminam-se as arestas saindo do vértice. Adicionalmente, o custo de se processar vértices em VS1 não é o mesmo de processar um vértice em VS2. Supondo-se esse cenário, deseja-se construir um algoritmo ótimo que elimine todas as arestas do grafo bi-partido.

Modelo funcional de apoio

Especifica-se, nesta seção , um modelo funcional que servirá de apoio às definições propostas na seqüência do capítulo.

- $P_i(a_i)$ - avalia um predicado (P_i) sobre um valor (a_i).
- $a_i.tuplas()$ - retorna o conjunto de tuplas em que a_i participa.
- $a_i.arestas()$ - retorna o conjunto de arestas partindo de a_i.
- $a_i.pares()$ - retorna o conjunto de $b_js\ distintostaisquea_i.tuplas \cap b_j.tuplas \neq \emptyset$.
- $Prob(P_i).true()$ - probabilidade de que a avaliação do predicado P_i resulte em verdadeiro, $(\equiv fatordeseletividade)$.
- $Prob(P_i).false()$ - probabilidade de que a avaliação do predicado P_i resulte em falso, $(\equiv (1 - fatordeseletividade)$.

- $P_i.custo()$ - custo de execução, por valor, do predicado P_i.
- $a_i.naresta()$ - número de arestas partindo de a_i [9];

O Fanout de valores de entrada para programas

Deseja-se conceber uma estratégia de execução de programas caros que se beneficie de dependências entre os dados a serem processados. Nesse trabalho, tal dependência de dados é expressa pelas arestas do grafo bi-partido, que ligam vértices se os valores associados compartilham uma tupla na relação de entrada dos predicados caros. Dessa forma, quanto maior for o número de arestas a que determinado vértice estiver associado (correspondente no grafo ao grau do vértice associado) maior é a dependência de dados entre o vértice e seus valores associados.

Chama-se de *Fanout* de um valor a_i em $R[a]$, $F(a_i)$, ao número de arestas ligadas a a_i no grafo bi-partido formado a partir da relação$R(a, b)$, com as respectivas projeções: $R[a]$ e $R[b]$.

Grau de execução de um predicado P sobre um valor(GEPV)

Nos casos em que os dados a serem processados pelos programas caros são conhecidos, em particular conhece-se a distribuição de valores distintos nas colunas de entrada aos programas, bem como o perfil de execução de cada programa, pode-se estimar um fator de seletividade para os predicados caros. Nesses casos, o conhecimento do *fanout* dos valores pode enriquecer a decisão adotada estaticamente quanto a ordem de avaliação dos programas caros com uma ponderação quanto aos valores de entrada. O GEPV é um grau atribuído a cada um dos valores a serem avaliados, adicionando conhecimento sobre a dependência de dados entre eles e é expresso pela seguinte expressão :

$$GEPV(P_i(a_i)) = \frac{a_i.Fanout()*(1-(Prob(P_i).true()))}{P_i.custo()};$$

[9]tuplas contendo o mesmo par (a_i, b_j) são indicadas com anotações na aresta

5.5.3 Algoritmo guloso para uma arquitetura centralizada

Utilizando-se a relação de dependência de dados entre atributos representada pelo Fanout, deriva-se uma estratégia gulosa para avaliação do grafo bi-partido *BG*, em um ambiente monoprocessável. A intuição da estratégia está em utilizar o Fanout dos valores, associados ao *rank* dos predicados, para determinar a operação a ser executada.

O pseudo-código abaixo modela a implementação da estratégia gulosa para execução de predicados caros em ambiente centralizado. O algoritmo executa os $P_i(a_i)$, por ordem de GEPV.

Enquanto houver arestas
  calcule o $GEPV(P_i(a_i))$ que retorne o maior grau de execução onde $a_i.arestas() \neq \emptyset$
  execute $GEPV(P_i(a_i))$
  elimine as arestas segundo a avaliação de $P_i(a_i)$
Fim

5.5.4 Algoritmo Picking Cherry para arquitetura distribuída

Em uma arquitetura distribuída otimiza-se o algoritmo anterior processando-se, em paralelo, os predicados caros, sobre valores ordenados segundo seus *Fanouts*.

Nesta seção, apresenta-se uma estratégia paralela de avaliação de predicados caros que explora dependências de dados entre valores de atributos de entrada aos programas. A proposta apresenta três aspectos distintos e igualmente importantes: a abordagem de otimização da execução de predicados caros, utilizando-se de dependências de dados entre os atributos; o algoritmo de execução, que explora, de maneira gulosa, o *fanout* entre os valores e, por fim, o modelo de execução paralelo independente para predicados caros.

Definição do problema

Deseja-se determinar o sub-conjunto *U* de vértices em *VS1* e *VS2* que maximize a eliminação de arestas do gráfo bi-partido *BG*.

Solução

Teorema 5.5.1 *Seja um grafo bi-partido* $BG = (VS1, VS2, E)$*, como descrito em 5.5.2 para uma relação* R(a,b)*. O custo total,* CP*, da avaliação dos valores* $v_i \in U$*, que maximizam a eliminação de arestas em* E*, é o mesmo, independentemente da ordem de suas avaliações.*

Proof:

Sejam $a_i, a_j \in R[a]$ e $b_k \in R[b] tais\ que\ b_k \in (a_i.pares() \cap a_j.pares())$.

Suponha-se que $P_i(a_i) = falso \Rightarrow (\exists\ e \in\ E\ tal\ que\ e \in (a_j.arestas() \cap b_k.arestas()))$

$\Rightarrow CP = P_i(a_i).custo() + P_i(a_j).custo()$.

Correspondentemente, se $P_i(a_j)$ é processado inicialmente e $P_i(a_j) = falso \Rightarrow (\exists\ e \in E\ tal\ que\ e \in (a_i.arestas()\ \cap b_k.arestas()))$

$\Rightarrow CP = P_i(a_j).custo() + P_i(a_i).custo()$.

c.q.d.

Considerando-se o teorema 5.5.1, pode-se basear a escolha dos vértices a processar em $V = VS_1 \cup VS_2$ que maximizem a eliminação de arestas em BG sem consideração com a ordem entre as avaliações. Uma heurística possível é a que escolhe os vértices a processar segundo a ordem descendente de fanout dos valores em cada coluna.

O fanout efetivo

Até aqui, discutiu-se, exclusivamente, sobre relações com dependência de dados entre valores de atributos com cardinalidade (1-N) (i.e. para cada valor de região ocorrem n valores de cidade). Para atender ao caso mais geral de relacionamentos (muitos-muitos), é necessário estender o conceito de *Fanout* definido anteriormente. Define-se *Fanout efetivo*, ao potencial redutor da avaliação de um predicado sobre um valor, onde considera-se a probabilidade de que essa avaliação elimine valores distintos, associados ao valor processado através de arestas no grafo bi-partido. Considere-se, então, o conjunto dos pares $(a_i, b_j) \mid a_i \in VS1 \wedge b_j \in VS2 \wedge b_j \in a_i.pares()$, tem-se, então:

$FanoutEfetivo(P_u(a_i)) = \sum_{j=1}^{n}(Prob.a_i.false)^k$, onde n é o número de elementos em $a_i.pares()$ e $k = Fanout(b_j)$.

A expressão pode ser compreendida da seguinte forma. Para se eliminar um valor b_j, sem processá-lo, é preciso que todos os k valores distintos, aos quais esteja associado, sejam avaliados como falso, ou seja $(Prob.a_i.false * \ldots * Prob.a_t.false) \equiv (Prob.a_i.false)^k$. O somatório exprime a redução sobre todos os elemento em $a_i.pares()$.

O algoritmo PC paralelo

O algoritmo guloso *Picking Cherry*, para arquitetura distribuída, é , resumidamente, apresentado abaixo, para uma consulta q com dois predicados caros P_1 e P_2 e predicados simples pr:

calcular os *Fanouts* de (a_i) e (b_j);
inicie P_1;
inicie P_2;
Enquanto $a_i.arestas() \cup b_j.arestas() \neq \emptyset$ faça
    forneça o $a_i(resp.b_j)$ de maior *fanout* a P_1 e processe $P_1(a_i)$ (resp.$P_2 b_j$) ;
    atualize novo *Fanout* para $b_j(resp.a_i) \in a_i.pares()(resp.b_j.pares())$;
    ...
    ao final da avaliação de $P_1(a_i) \vee P_2(b_j)$
        recalcular os *fanouts* dos valores $\in a_i.pares()(resp.b_j.pares())$
        se falso
            $Prob(P_1(a_i)).true = 0(resp.b_j)$
            eliminar todas as arestas $\in a_i.arestas()(resp.b_j.arestas())$
        senão
            $Prob(P_1(a_i)).true = 1(resp.b_j)$
            eliminar as arestas partindo de $a_i(resp.b_j)$
            se $\exists b_j \in a_i.pares() \mid Prob(P_2(b_j)).true = 1(resp.a_i)$
            gravar $a_i.tuplas() \cap b_j.tuplas()$
fimEnquanto
termine $P_1(a_i)$;
termine $P_2(b_j)$;

A estratégia de execução, baseada no cálculo do *fanout*, exige que se materialize a relação intermediária, produzida pela avaliação de predicados simples. O processamento da relação materializada, produz duas estruturas de dados: uma tabela *hashing*, contendo cada um dos valores $v_i \in \{VS1 \cup VS2\}$, como chave de acesso, associados a uma lista dos valores em $v_i.pares()$, e uma segunda lista, apresentando os elementos de $v_i.tuplas()$; além de uma árvore B^+ para cada conjunto VSn, indexada pelo valor do *fanout* de cada v_i.

Arquitetura

A proposta de processamento de predicados caros, mediante a exploração de dependências de dados entre valores, introduz uma nova estratégia de execução de predicados caros. Nesta nova estratégia ao invés de se executar de maneira *pipeline* os predicados caros, executa-se os inteiramente em paralelo. Cada Predicado recebe um valor a processar e retorna o resultado de sua avaliação a um módulo controlador. Este último é responsável por atualizar a relação de arestas associadas e distribuir os valores aos predicados.

Esta estratégia libera a execução da restrição de sequencialidade entre os predicados favorecendo, entre outras coisas, a utilização de recursos distribuídos.

Para introduzir o processamento paralelo de valores de tuplas em predicados, é preciso combinar a estratégia de execução paralela de fluxo de dados, sobre a qual executam-se os demais operadores no plano de execução paralelo, com a estratégia independente de execução de predicados caros. Para tal, adapta-se o operador *Exchange*.

O operador *Exchange* é composto de dois módulos, um produtor e um segundo consumidor. Ambos oferecem uma interface, segundo o modelo iterador, para comunicação com os operadores vizinhos. O processamento de predicados, segundo o modelo de paralelismo independente, é implementado internamente ao operador (ver figura 5.7).

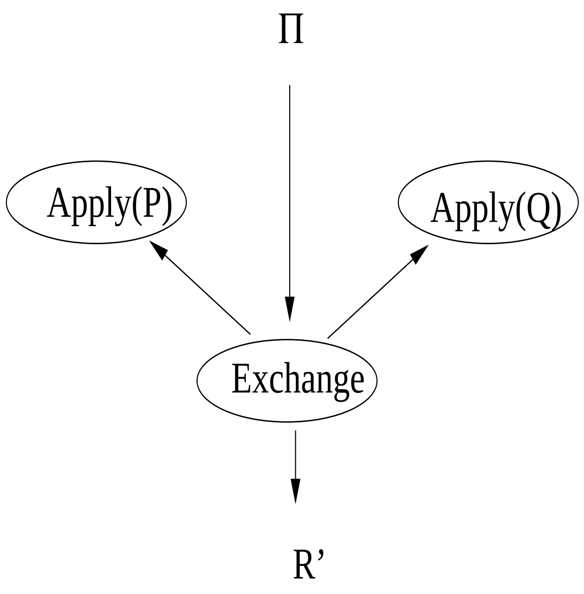


Figura 5.7: *Estratégia paralela independente no PC*

5.5.5 Discussão

A abordagem *PC* explora dependências de dados entre parâmeros de predicados caros, de modo a minimizar o número de invocações a programas. Considera-se que predicados baratos anteriores foram executados, segundo a estratégia *Pull-Up*. Em especial, junções executadas podem reagrupar os dados em relacionamentos (M-N) e (1-N). Tais relacionamentos são bastante comuns em aplicações cujos dados tenham sido modelados de forma normalizada em diversas relações.

Algumas características da abordagem *PC* merecem ser realçadas frente aos demais algoritmos para execução de predicados caros conhecidos na literatura.

Inicialmente, não há nenhuma proposta que o autor deste trabalho conheça, que aproveite relacionamentos de dependência de dados para o processamento de predicados caros.

Em segundo lugar, por estar baseado no processamento de valores distintos, o algoritmo não requer o tratamento de *caching* pelo *wrapper configurado de programa*, o que simplifica sua geração automática. Uma conhecida solução, que também não requer *caching*, é a adoção de semi-join para execução de programas [MS99] distribuídos. Essa solução, não é compatível com a estratégia *PC*, uma vez que na última o resultado do processamento de cada valor altera o estado do grafo bi-partido.

Adicionalmente, a estratégia independente provê alocação total de recursos, independentemente de variações em tempo de execução, e de características de dados (seletividade e número de duplicados). *PC*, em ambiente distribuído, elimina a necessidade de ordenação de predicados, proporcionando o benefício, no tempo total de execução, de reduções mútuas entre predicados.

Por fim, a necessidade de materialização do resultado da avaliação de predicados baratos, implica na quebra do fluxo de dados, durante a execução do *PECD*. Porém, considerando-se a premissa assumida neste trabalho, do custo de predicados caros e transferência de *Blobs* sendo extremamente maior que o de execuções de predicados simples, o custo de materialização torna-se desprezível.

5.6 Sumário

Neste captítulo propôs-se uma estratégia para a geração de planos de execução de consulta distribuída com programas científicos. Inicialmente, argumenta-se que, frente às premissas adotadas, a estratégia *Pull-Up*, para programas caros, deve ser sempre utilizada, desde que apoiada pelas técnicas de execução propostas.

Produz-se, com uso dessas técnicas, um plano de execução de consultas onde a alocação de programas científicos pode ser livremente determinada. Uma vez que considera-se um ambiente de avaliação de consultas distribuído, define-se uma estratégia para geração de planos paralelos para programas sem predicados sobre seus resultados, a partir das relações de dependência de dados entre os programas.

Para execução de programas com avaliação de predicados sobre seus resultados, propõe-se uma nova estratégia híbrida que combina os modelos de paralelismo independente e de fluxo de dados. Essa estratégia objetiva reduzir os impactos, no tempo de resposta da consulta, de variações no fluxo de dados detectados em tempo de execução.

Por fim, apresentou-se a abordagem *Picking Cherry*. *PC* propõe uma estratégia original para execução ótima de consultas distribuídas compreendendo a invocação de predicados caros. A intuição principal do algoritmo é identificar os valores que possam prover maior benefício para a execução da consulta. O potencial redutor de cada valor, chamado de *fanout efetivo*, exprime as dependências funcionais entre valores de atributos, processados por predicados caros. A abordagem PC define uma estratégia de avaliação de predicados baseada no processamento de valores distintos, que permite sua avaliação segundo o modelo de paralelismo independente. Com isso, além de não requerer *caching* para o processamento de programas, os recursos distribuídos apresentam utilização total.

No capítulo seguinte apresenta-se as validações que comprovam as qualidades das propostas apresentadas neste capítulo.

Capítulo 6

Validação da estratégia

6.1 Introdução

Nos capítulos 4 e 5 foi apresentada uma estratégia de execução paralela de consulta científica que minimiza seu tempo de resposta e maximiza a alocação de recursos distribuídos. Este capítulo descreve os experimentos realizados e apresenta as conclusões sobre a validade das propostas.

A estratégia foi validada segundo modelos de experimentação e simulação. Para isso, desenvolveram-se um protótipo de otimizador e uma máquina de execução de consultas distribuídas. Os protótipos foram implementados em Java, usando JDK 1.1.7, rodando em máquinas Pentium III, com 128 MBytes de memória RAM, e utilizando o sistema operacional LINUX *Red Hat 6.2*.

6.1.1 O otimizador exaustivo

O protótipo de otimizador gera planos de execução distribuídos para consultas que incluam chamadas de programas e transferência de *Blobs*. O otimizador recebe, como entrada, um grafo de consulta, expresso como um documento XML, e produz um plano físico de execução. O grafo de consulta é enriquecido com meta-informações sobre seus vértices e arestas. As meta-informações sobre uma relação compreendem sua cardinalidade, a definição de seus atributos e da respectiva quantidade de valores distintos, além de sua localização. Para atributos do tipo *Blob*, seu tamanho médio é fornecido. Programas, por sua vez, são descritos segundo a associação entre seus parâmetros de entrada e os atributos da relação que fornecem os valores para processamento, bem como segundo a estimativa de custo de execução por tupla. Finalmente, cada predicado participa com uma estimativa de seu fator de seletividade. Planos de execução seguem a topologia de árvores lineares, profundas à esquerda. As operações lógicas empregadas são as de *seleção, junção, projeção, produto cartesiano e apply*. Além dessas, as operações físicas empregadas são : *scan, send, materilize, read e getblob*.

A determinação do plano ótimo de execução de consulta distribuída baseia-se num modelo de custo paralelo, apresentado na seção 6.1.3, e num algoritmo, exaustivo, de permutação na ordem de predicados caros. O modelo de custo considera a execução dos operadores em paralelo, segundo o modelo de fluxo de dados.

Antes de iniciar a apresentação dos experimentos, nas seções seguintes, descreve-se o contexto em que se utilizam validações. Em seguida, apresenta-se o modelo de custo paralelo adotado pelo otimizador. Finalmente, apresenta-se o *benchmark* utilizado.

6.1.2 As validações

Os experimentos podem ser divididos em duas partes: validação das técnicas de execução, apresentadas no capítulo 4, e validação das estratégias híbrida e *Picking Cherry*.

A validação das técnicas de execução foi realizada a partir de simulações produzidas pela geração de planos parciais de execução. Os *PECD*s foram avaliados em função do modelo de custo considerando-se suas operações e as informações estatísticas sobre os operandos.

O desempenho da estratégia híbrida é analisado através de simulação e experimentação, enquanto o *Picking Cherry* é experimentalmente comparado com outras estratégias. Para tais experiências, supõe-se uma máquina de execução distribuída, com comunicação entre processos remotos. O tempo de resposta total de execução da consulta é utilizado na comparação entre os algoritmos. A fim de se obter um universo de teste abrangente em função do número de execuções, utilizou-se simulações da execução real. Para tanto, construiu-se um simulador que modela execução de planos paralelos segundo o modelo de fluxo de dados e a execução com emprego da estratégia híbrida. A simulação de execução de consulta é obtida através de contadores que simulem os tempos de execução e inatividade para cada programa. Assegurando a correção dos resultados, comparam-se algumas simulações com resultados de experimentações reais.

Na validação das estratégias paralelas de execução de predicados caros, considerou-se onde os predicados de junção simples foram avaliados e uma relação contendo os atributos necessários é produzida. Dessa forma, analisam-se especificamente o problema do impacto das variações, em tempo de execução, e a ordem de execução de predicados caros.

As próximas seções estão organizadas conforme se segue. Inicialmente, apresenta-se o modelo de custo utilizado pelo otimizador. Em seguida, descrevem-se as validações de cada uma das técnicas de execução empregadas. Por fim, avalia-se a estratégia híbrida de execução e a abordagem *Picking Cherry*.

6.1.3 Modelo de Custo

Nesta seção, apresenta-se o modelo de custo adotado pelo otimizador. O objetivo do processo de otimização, entenda-se, é produzir um plano de execução de consulta distribuído que minimize o *tempo de resposta total* de consultas. Dessa forma, os custos das operações, no plano de execução de consulta, são expressos em unidades de tempo.

Um plano de execução de consulta é expresso como uma seqüência de operações executáveis. Cada operação apresenta um modelo de custo próprio, dependente de parâmetros externos (i.e taxa de transferência na rede, taxa de transferência de entrada, saída, etc.), do tamanho da entrada (i.e. cardinalidade da relação, número de valores distintos ou tamanho da tupla) e do custo de execução do algoritmo associado.

O custo de um plano de execução de consultas completo é computado considerando-se as estratégias paralelas, independente e de fluxo de dados.

Modelo de custo para estratégia paralela de fluxo de dados

Consideram-se as premissas tradicionais quanto à relações no que se refere à distribuição uniforme de valores de atributos pelas tuplas e à independência entre valores de atributos distintos. Considera-se, ainda, que o custo de avaliação de programas científicos é independente da entrada e de variações na utilização de CPU, e se conhece uma estimativa do fator de seletividade para predicados caros. Quanto aos *Blobs*, utiliza-se, no modelo de custo, uma estimativa de seu tamanho médio por atributo. Por fim, assume-se constante a taxa de transferência na rede.

O modelo de custo adotado é uma adaptação ao proposto em [GHK92]. Seja $R = \{R_1, R_2, \ldots, R_n\}$ o conjunto de relações em uma consulta q, onde $t \in R_i$ é uma tupla. Seja ainda $P = \{P_1, P_2, ..., Pn\}$ uma seqüência de predicados em um plano *PE*, gerado para a consulta q. Define-se:

1. Caminho crítico de PE - o predicado, $P_i \in P$, cuja estimativa de custo de execução, segundo um plano *PE*, é da forma $\{P_i \mid custo(P_i) \geq custo(P_k), 1 \leq k \leq n\}$.

2. Custo de predicado - o custo, $custo(P_k) = \sum_{j=1}^{m}$ custo da operação j, onde operação *1* corresponde a primeira operação após o predicado P_{k-1}, e m corresponde a operação de restrição, ou junção, que implementa o predicado P_k.

3. Custo do caminho crítico - o custo, $Custodocaminhocr\acute{i}tico = custo(P_k)$, onde (P_k) é o predicado no caminho crítico.

4. Custo da primeira tupla - o custo $custoprimtupla = \sum_{k=i-1}^{1}(\frac{custo(P_k)}{card}) * selacum_k$. Onde, P_{i-1} corresponde ao predicado anterior ao caminho crítico, $seleacum_k$ corresponde ao número de tuplas necessárias para se produzir um resultado verdadeiro pelo predicado P_k, e card é a cardinalidade da relação processada por P_k.

5. Custo da última tupla - o custo $custo\acute{u}ltimatupla = (\sum_{k=i+1}^{n}(\frac{custo(P_k)}{card}))+$ custo de projeção, onde P_{i+1} corresponde ao predicado seguinte ao caminho crítico.

No que se segue, apresentam-se duas tabelas utilizadas para o cômputo do custo de cada operação no *PECD*. A primeira tabela define os parâmetros utilizados, enquanto a segunda apresenta as expressões de custo propriamente ditas.

parâmetro	descrição
cscan	taxa de E/S
ctransf	taxa de transferência
cpred	cte de comparação
read	$\equiv$ a scan para *Blobs*
tmédio	tamanho médio do *Blob*
mcost	custo de materialização
cardi	cardinalidade do operando i
jcost	cte
concost	custo de conexão
cachecost	custo de caching
custo pgm	custo de chamada do programa, por tupla
k	tamanho do *Blob*

A tabela a seguir mostra as expressões de custo para cada operação. Em algumas dessas, aparece a opcionalidade entre *dist e card*, referindo-se a presença, ou ausência, do emprego de *caching*.

operação	expressão custo
scan	$tamanhotupla * card * cscan$
send	$(tamanhotupla * card) \div ctrans$
restrict	$cpred * card$
read	$tamanhotupla * card * cscan$
project	$zero$
mat	$tmédio * [dist \mid card] * mcost$
junção	$(card1 + card2) * jcost$
getblob	$(tmédio * [dist \mid card]) + concost + [cachecost]$
apply	$custopgm * [dist \mid card] + [cachecost]$

Os parâmetros externos de custo considerados encontram-se na tabela a seguir.

descrição	valor
taxa de transferência na rede	70 KB/s
taxa de transferência em disco	10 MB/s
acesso à memória	1000 MB/s
custo de conexão	0.1 s
tamanho URI	60 bytes

Com base nas definições acimа, o custo do plano paralelo é expresso como:

$$CustodoPlano = Custodaprimeiratupla + Custodocaminhocrítico + custodaúltimatupla. \tag{6.1}$$

A expressão em 6.1 exprime o modelo de custo utilizado pelo otimizador no cômputo do custo dos *PECDs* com modelo de paralelismo de fluxo de dados. Na seção seguinte, apresenta-se o *benchmark* desenvolvido, que simula uma aplicação ambiental distribuída.

6.2 Benchmark

A validação das técnicas de execução seguem um *benchmark*, composto de duas relações *T1* e *T2*, e dois programas, *P1* e *P2*, cada um localizado em um sítio distinto. Adicionalmente, representam-se duas funções, *f1* e *f2*, que processam valores de tipos simples e encontram-se localizadas em sítios também distintos. A seguir apresenta-se o esquema dos objetos participantes do *benchmark*.

Esquema:

T1 (a:int, b1:blob, c1:string)
T2 (a:int, b2:blob, c2:string)
P1 (<c:blob> in,<r:float> out)
P2 (<c:blob> in, <r:float> out)
f1(<d:int> in , <r:int> out)
f2(<e:string> in, <r:int> out)

Foram desenvolvidos vários casos de teste, onde, em cada um, utilizaram-se versões das seguintes consultas:

Exemplo 6.2.1

Select *T1.p1*
  From *T1, T2*
  Where *T1.a = T2.a and*
    P1(T1.b1) = 10 and
    P2(T2.b2) = 5
    (a)

Select *P1(T1.b1)*
  From *T1, T2*
  Where *T1.a = T2.a*
    (b)

Select ***
  From *T1*
  Where *f1(a)=3 and*
    f2(c) = 10 and
    P1(b1) = 5
    (c)

Select *T2.p2*
  From *T1, T2*
  Where *T1.a = T2.a and*
    P1(T1.b1) = 10
    (d)

□

6.3 Validação das técnicas de execução

Para validar e concluir sobre a adoção das técnicas de execução, foram realizadas simulações de execuções, a partir de planos parciais gerados pelo otimizador. A estratégia de validação apresenta um perfil incremental de complexidade. Inicialmente, verifica-se a valia de cada técnica, individualmente. Em seguida, combinam-se, duas a duas, técnicas complementares, até que se chegue na simulação que inclua as quatro técnicas: transferência postergada, *caching*, paralelismo entre captura e execução de programa e, finalmente, *pull-up*.

6.3.1 Transferência em segundo plano

Apresenta-se, nessa seção, as validações da técnica de transferência postergada. Discute-se o efeito da combinação dessa técnica com a técnica de *caching* e *Pull-Up*.

GetBlob

Para se validar esta técnica, utiliza-se a consulta *(b)* do *benchmark*. Inicialmente, compara-se a execução de um plano que implementa o operador *GetBlob*, contra um plano equivalente, onde o *Blob* é transferido como parte da tupla. O plano gerado contém apenas as operações associadas a E/S e transferência na rede, além da operação *GetBlob*. Considera-se que o *PECD* determina duas transferências, pela rede, de atributos de *T1*. A primeira envia tuplas do sítio de *T1* para o sítio de *T2*. O envio para o sítio de *T2* é usado apenas para aumentar o percurso das tuplas. Em seguida, tuplas são enviadas para o sítio da projeção final, onde se encontra o programa P_1. Neste teste, a implementação do operador *GetBlob* efetua uma nova conexão ao sítio de *T1* para cada *Blob* a ser capturado. O experimento computa 10 execuções para cada estratégia, com e sem *GetBlob*, onde, em cada execução, se varia o tamanho médio dos *Blobs* em *T1.b1*. Os planos de execução são ilustrados abaixo:

- com GetBlob - T1-Scan-Send-Send-Scan-GetBlob-Mat

  Nesse plano, a relação *T1* é acessada para leitura a partir de disco. Suas tuplas, após o tratamento pelo adaptador do atributo de tipo *Blob*, apresentam os atributos *(a,b,c)*, onde valores de *b* são substituídos pelos correspondentes *URIs*, formando uma tupla de 94 bytes. As tuplas são sucessivamente transmitidas até chegarem ao sítio de *P1*, onde o processamento da operação *GetBlob* efetua o acesso aos *Blobs* a partir do disco, e transmite-os pela rede, armazenado-o localmente.

- sem GetBlob - T1-Scan-Read-Send-Mat-Scan-Send-Mat

Na execução sem *GetBlob*, a relação *T1* é lida do disco e, para cada tupla, o *Blob* correspondente é acessado. A tupla resultante apresenta os atributos *(a,b,c)*, onde b representa os valores de tipo *Blob*. As tuplas são transferidas para os sítios de *T2* e *P1*, respectivamente, onde são materializadas.

Discussão

Os tempos de resposta de 10 execuções de cada um dos planos são, comparativamente, apresentados no gráfico da figura 6.1. Nessas execuções, a relação *T1* tem cardinalidade *40* e o atributo *b* conta com 10 valores distintos. O gráfico mostra a execução dos planos com *caching*. O plano *sem GetBlob* apresenta tempo de resposta total dominado pelo custo de transferência das tuplas, que é da ordem de 10^2 superior ao custo de leitura em disco. Por sua vez, na estratégia com *GetBlob*, verifica-se que a conexão ao sítio de origem da transferência, também é decisória no desempenho do plano para *Blobs* pequenos. Verifica-se que, para *Blobs* com tamanho menor que *1 KB*, o custo de conexão entre os sítios destino e origem supera o custo de transferência de *Blobs*, e varia segundo o número de *Blobs* a serem capturados (influência do *caching*). Em casos como esse, deve-se manter a conexão ativa durante todo o processamento da consulta.

Para *Blobs* maiores que *1 KB*, os custos de ambos os planos passam a ser proporcionais ao seu tamanho. Em tal cenário, a razão entre esses, dada por ($semGetBlob \div comGetBlob$), tende a *(s * r)*, onde s é o número de transferências e r é a razão entre a cardinalidade da relação e o número de valores distintos do atributo *(b)*. Nas execuções representadas na figura 6.1, tem-se r=4 e s=2, o que está de acordo com a tendência do gráfico para a razão 8.

A figura 6.2, mostra a tendência da razão $semGetBlob \div comGetBlob$ apontando para 2, quando não se implementa *caching* associado ao operador *GetBlob*. O acréscimo de 4x no custo da execução com a estratégia *GetBlob* corresponde ao processamento de *Blobs* duplicados.

Em suma, o custo de execução de planos, com a técnica de transferência postergada, é independente da distância entre a origem e o destino do *Blob*, no *PECD*, e é proporcional a seu tamanho e ao número de *Blobs* transferidos.

Custo aproximado de execução de planos com a estratégia *GetBlob*, e uma conexão:

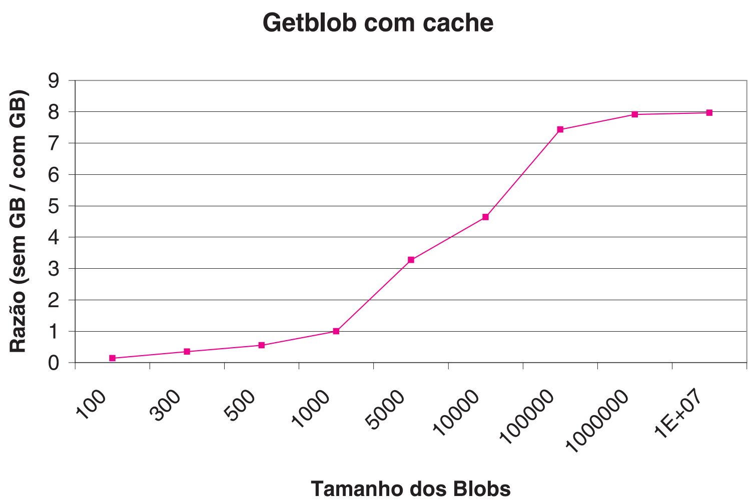


Figura 6.1: *GetBlob e cache*

Com *caching* $\Rightarrow\approx (dist * k)$

Sem *caching* $\Rightarrow\approx (card * k)$

Sem *GetBlob* $\Rightarrow\approx (card * (ns\acute{\imath}tio - 1) * k)$

Na próxima seção, analisam-se planos onde *Blobs* são afastados de suas fontes pela adoção da heurística *Pull-Up*.

GetBlob e Pull-Up

O objetivo deste experimento é constatar o benefício da técnica *GetBlob*, quando associada à técnica *Pull-Up*.

Para esta avaliação, utilizou-se a consulta do *benchmark* 6.2.1 *(c)*. De modo a forçar a validação de predicados em sítios distintos, utilizou-se dois predicados sobre os resultados de funções do usuário, *f1* e *f2*, sobre os atributos de *T1*, *T1.a* e *T1.c*. O operador *GetBlob* é alocado, junto ao programa P1, como parte do último predicado a ser avaliado, conforme a heurística *Pull-Up*.

No experimento, a relação *T1* tem 500 tuplas e o atributo *b* apresenta 350 valores distintos. Os predicados sobre $f1$ e $f2$, tem fator de seletividade estimado em *0.4*

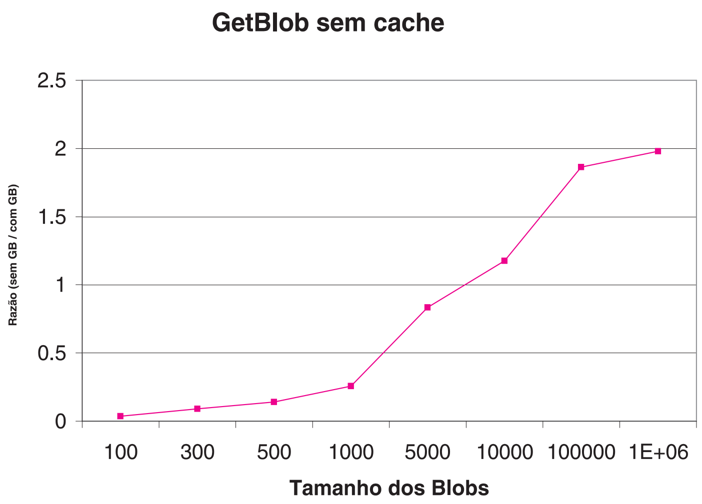


Figura 6.2: *GetBlob sem cache*

e *0.2*, respectivamente. Finalmente, considera-se que as reduções induzidas pelas avaliações de predicados sobre os resultados de f1(a) e f2(c), proporcionam redução correspondente sobre os valores distintos de *b*.

Os planos gerados cobrem o custo até a captura do *Blob*, no sítio do programa P1, sendo sua avaliação inibida para simplificação da validação.

- com GetBlob - T1-Scan-Apply(f1)-Restrict(f1)-Send-Apply(f2)-Restrict(f2)-Send-Read-GetBlob-Mat
- sem GetBlob - T1-Scan-Read-Apply(f1)-Restrict(f1)-Send-Mat-Read-Apply(f2)-Restrict(f2)-Read-Send-Mat

No plano sem a adoção do operador *GetBlob* é preciso materializar a relação transferida, em cada sítio, uma vez que o tamanho do *Blob* pode ser maior que a quantidade de memória disponível.

Discussão

Esta experiência foi realizada seguindo procedimento similar ao anterior, com a diferença que a conexão ao sítio de origem foi mantida durante toda a execução.

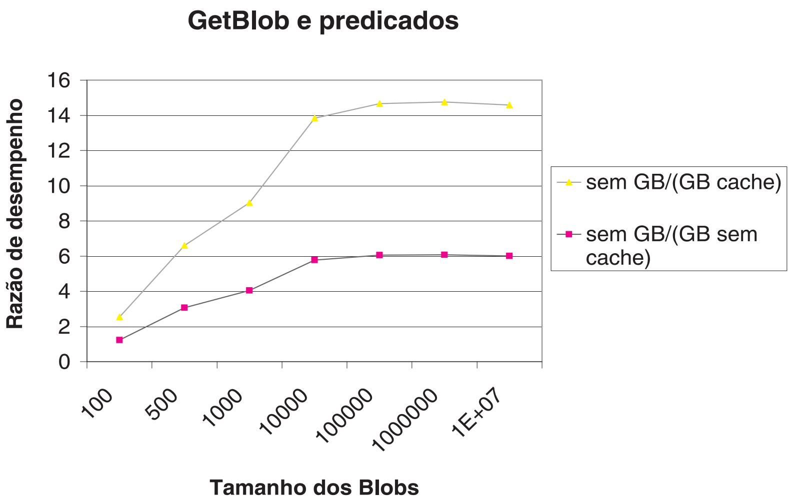


Figura 6.3: *GetBlob & Pull-Up*

Nota-se, para *Blobs* menores que *10Kbytes*, o custo de transferência de tuplas com atributos básicos e URIs é comparável ao custo de transferência dos próprios *Blobs*, principalmente pelas reduções no número de tuplas que diminuem ainda mais o custo dos *Blobs* transferidos. Apesar de, em tal hipótese, o benefício da técnica de *GetBlob* ser reduzido, ainda assim é vantajoso, pois reduz o número de transferências. Para *Blobs* maiores que *10Kbytes*, o custo dos planos, utilizando-se a técnica de *GetBlob*, é dominado pelo custo de transferência, que pode ser aproximado pela expressão:

custo com *GetBlob* $\approx (n * (\prod_{i=1}^{p} sel_i) * k) \div 7e4$, onde sel_i é a seletividade do $i_{ésimo}$ predicado, n vale *card* ou *dist*, dependendo da adoção ou não de *caching*, e k é o tamanho médio do *Blob*.

custo sem *GetBlob* $\approx ((\sum_{i=1}^{p} card * (\prod_{j=1}^{i} sel_j) * k) \div 7e4$.

A razão entre os custos das estratégias *sem GetBlob* e *com GetBlob*, considerando-se s sítios e p predicados é dada pela expressão:

$\frac{card}{n} * (\sum_{i=0}^{s-2} \frac{1}{\prod_{j=i+2}^{p} sel_j}) + 1$.

A expressão acima representa, como esperado, o custo extra da transferência de *Blobs* até o sítio do penúltimo predicado, uma vez que a última transferência tem custo correspondente em ambos os planos.

Conclusão

Os experimentos mostram a validade da aplicação da técnica de *transferência postergada*. A incidência do custo de conexão sobre o resultado das transferências aconselhou a manutenção da conexão durante o maior tempo possível. Note-se que, apesar de estarem referenciados em uma mesma relação, *Blobs* podem estar armazenados em outros sítios, desde que exista um servidor Le Select no sítio referenciado. Nesse caso, haveria necessidade de se manter uma conexão com cada um dos servidores, que, no pior caso, pode ocorrer para cada valor distinto. Esse problema aparece para *Blobs* pequenos, menores que *1 KByte*, quando se utiliza *caching*; e é inteiramente tolerável, uma vez que imagens e outras estruturas científicas são, em geral, mais volumosas. Finalmente, a inclusão da estratégia *Pull-Up* apresenta o resultado esperado, que beneficia a execução com a composição de reduções. Deve-se atentar que, apesar de em consultas predicados caros aparecerem em pequeno número, transferências de *Blobs* podem ser induzidas por avaliação de predicados simples distribuídos, executados antes do consumo de *Blobs*, justificando e estendendo o benefício da técnica.

6.3.2 Caching

O uso de *caching* elimina o efeito de valores duplicados sobre a avaliação de programas e a transferência de *Blobs*. A validação da técnica teve como objetivo mensurar o custo versus o benefício de sua adoção.

Caching versus alocação de memória

No primeiro experimento, comparou-se o custo de acesso à tabela *hashing*, considerando-se um número crescente de valores distintos, e o correspondente acréscimo de custo de execução da consulta. Para a validação, executou-se a consulta *(b)* do *benchmark*. A relação *T1* apresenta cardinalidade de 10000 tuplas e o custo de execução por invocação de *P1* é estimado em 10000 segundos. O experimento é executado 10 vezes, variando-se o número de valores distintos de b entre 1000 e 9000, e se registrando o custo de execução contra o de uso de memória.

Custo de memória

A valoração do custo de acesso a memória foi obtida com base no número de partições necessárias para armazenar, na tabela *hashing*, as chaves de acesso e os valores associados. No experimento, considerou-se *1 MByte* de espaço disponível para alocação das partições em memória. Dado que os *Blobs*, remotos e locais, têm apenas seus URIs armazenados na tabela *hashing*, o espaço disponível é suficiente para armazenar até 9000 valores distintos, que é o número máximo analisado.

Discussão

O custo, por chamada, da execução de programas científicos varia entre alguns segundos a algumas horas. Proporcionalmente, o custo de uma eventual paginação entre partições da tabela *hashing*, implementando o *caching*, torna-se desprezível. Adicionalmente, as filas de armazenamento temporário das tuplas a serem processadas, tanto à frente do operador *GetBlob*, quanto à frente do operador *Apply*, diluem o custo adicional de uma eventual paginação. Soma-se ainda a esse cenário o fato de que, nas aplicações ambientais que se analisaram consultas não apresentam característica de processamento de grande número de tuplas. Por outro lado, a função de custo adotada apresenta-o com variação linear, frente ao somatório do custo das operações no caminho crítico. Em consultas com programas, o caminho crítico encontra-se na avaliação de um dos programas, que, por sua vez, tem custo proporcional ao número de entradas. Sendo assim, tem-se o custo do plano linear em função do tamanho da entrada, enquanto o custo de uso de memória se mantém constante (ver figura 6.4).

6.3.3 Paralelismo entre a captura de Blobs e a execução de programas

Este experimento analisa a valia da paralelização entre a captura de *Blobs*, e a execução de programas. Utilizou-se uma adaptação da consulta *(b)* do *benchmark*, de onde se retirou a junção, reduzindo a consulta à chamada de *P1* sobre a tabela *T1*. O atributo *(b)* apresenta tamanho médio de $5 * 10^5$ bytes. Efetuam-se 12 execuções, em que variou o custo de avaliação de *P1* no intervalo $[350, 3500000]$ segundos.

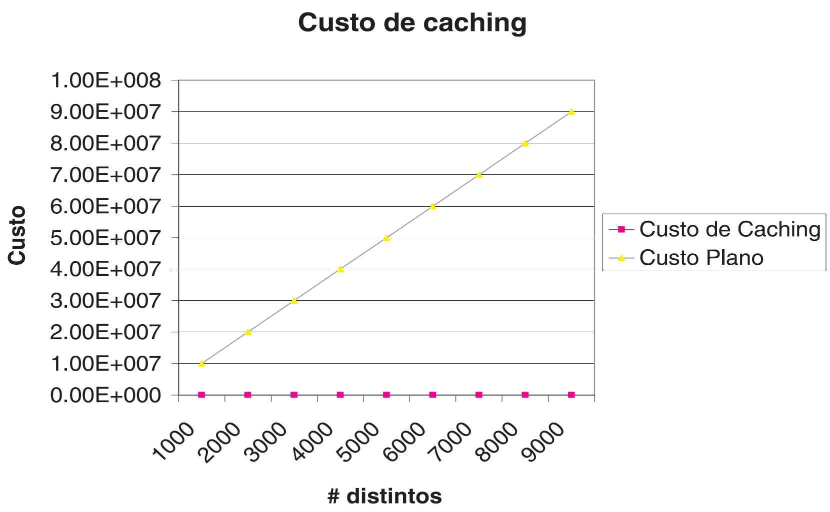


Figura 6.4: *Custo de Caching*

Discussão

O processamento do programas científicos depende da captura de seus dados de entrada. Considerando-se que esses dados podem estar armazenados remotamente e que podem incluir dados do tipo *Blob*, o custo de captura da entrada pode ser tão ou mais significativo, do ponto de vista do tempo de resposta da consulta, quanto a própria execução do programa. Do ponto de vista do consumo de recursos computacionais, a execução de programas faz uso intensivo de CPU, enquanto a captura de dados de entrada opera sobre os dispositivos de E
S. Segundo esse modelo, a paralelização entre os dois processos parece oferecer ganhos no tempo de resposta total.

Os dois processos não podem, no entanto, executar de forma inteiramente independente, na medida em que, a entrada precisa estar disponível para que o programa execute, mais ainda, a ordem de avaliação é rígida, com a produção dos dados precedendo seu consumo. Dessa forma, adotou-se, para implementação dessa estratégia, o modelo de paralelismo de fluxo de dados.

A análise do benefício trazido pela estratégia paralela de fluxo de dados foi feita analitica e experimentalmente. A primeira utiliza o modelo de custo paralelo, apresentado em 6.1.3, para custear planos com as operações *GetBlob* e Apply executadas em paralelo. A segunda, serviu para confirmar a verificação analítica.

Na validação analítica, supõe-se *CC*, como o custo de captura dos n valores distintos de b e *CP* o custo de avaliação do programa *P1* sobre os mesmos n *Blobs*, tem-se que, simplificadamente, o custo de avaliação do plano será $\approx max \mid CC, CP \mid$. Logo, o paralelismo proporcionará benefícios para valores próximos de *CC* e *CP*, com ganho máximo quando $CC = CP$. Inversamente, para valores muito distantes, o paralelismo não trará benefícios, podendo até introduzir custos adicionais de gerência de sincronismo.

O gráfico 6.5 reflete a análise apresentada no parágrafo anterior. Na segunda execução, considerando-se o custo de execução do programa *P1* igual a *7* segundos e o tamanho de *Blob 500 KB*, tem-se $CC = 2535$ e $CP = 2450$. Esse cenário, proporciona um ganho de, aproximadamente, 1.9 para o processamento paralelo. Para programas mais baratos, o caminho crítico passa a ser a transferência de *Blobs*, o que fará com que o programa sempre espere por um novo valor a processar. Correspondentemente, quando o custo de execução de programa torna-se significativamente mais caro que a captura de seus dados, o benefício da paralelização dos processos é reduzido.

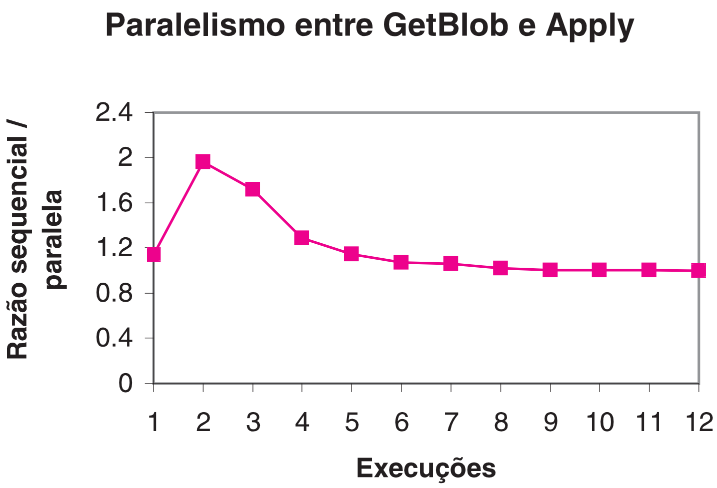


Figura 6.5: *Captura de Blobs com paralelismo*

6.3.4 Pull-Up

A heurística *Pull-Up* foi proposta, neste trabalho, para alocação de predicados caros no plano de execução de consultas. Considerando-se um custo de execução de programas, por chamada, extremamente maior que o custo associado ao processamento de uma tupla por um algoritmo de junção. Neste experimento, valida-se a heurística, utilizando-se a consulta *(d)* do *benchmark*.

Pull-Up para programas científicos

Analisam-se dois planos de execução,*(a)* e *(b)*, como ilustrado na figura 6.6. Inicialmente, estima-se o custo de avaliação da junção, como constante, e valendo 10^{-6} segundos, por tupla [1]. O custo de avaliação de *P1* é variado entre *1 e 1000* segundos. A relação *T1* apresenta cardinalidade igual a 500 tuplas, e 200 valores distintos para o atributo *b1*. A relação T2 apresenta cardinalidade de 100 tuplas. Cada *Blob* em *(b)* apresenta tamanho médio de 10KBytes. Finalmente, os fatores de seletividade dos predicados p_1, de junção, e p_2, sobre o resultado do programa *P1*, são, respectivamente, 0.3 e 0.1 %.

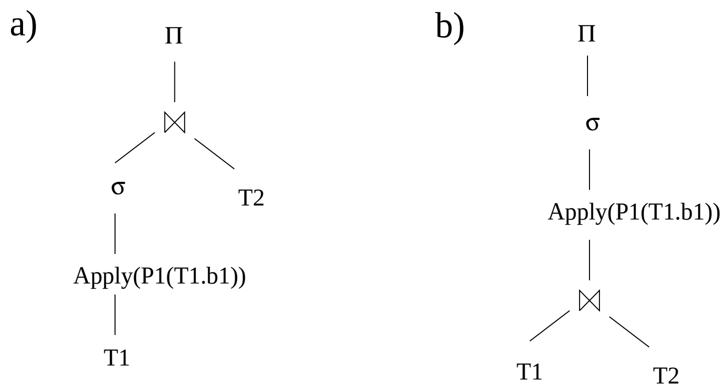


Figura 6.6: *Comparação Pull-Up, Push-Down*

Computa-se o tempo total de execução para 10 cenários dos planos *(a)* e *(b)*, um para cada valor estimado de *P1*. Sendo que o plano *(b)* é executado, para cada

[1]esse custo foi baseado no tempo de comparação entre dois valores inteiros, por CPU de 400 MHertz

cenário, duas vezes, considerando-se a adoção, ou não, da técnica de *GetBlob*. Tem-se, em resumo, três planos de execução para cada cenário: *(a)*, *Push-down*, com *cache* e sem *GetBlob* [2];*(b) Pull-Up*, com *cache* e *GetBlob*; e *(c) Pull-Up*, sem *cache* e sem *GetBlob*.

Discussão

O caminho crítico dos três planos analisados está na avaliação dos predicados caros. Nesse caso, mesmo o plano *(c)*, que implementa apenas a estratégia *Pull-Up*, traz resultados melhores que o plano baseado na estratégia *Pull-Down*, com *cache*. Verifica-se que, nesse cenário, onde a razão entre a cardinalidade e o número de valores distintos de *(b)* provê um ganho de 0.4%, no uso de *caching*, *Pull-Up* se beneficia de uma redução mais significativa de 0.3%, produzida pela junção. Esse resultado, que será melhor explorado na próxima seção, demonstra a correlação entre as técnicas, uma vez que, para quantidade de valores distintos menores, o resultado da comparação favoreceria o plano *(a)* frente ao plano *(c)*. Por sua vez, o plano *(*b), estando independente das variações em número de valores distintos, é sempre mais eficiente que os outros dois planos, segundo um fator de 3.3 que representa, exatamente, o ganho obtido com a redução imposta pelo predicado de junção.

O trabalho realizado por Hellerstein [HS93, Hel98] demonstra que uma seqüência ótima de execução de predicados caros é obtida ordenando-se descendentemente os predicados, com base em seus *rankings*. O *rank* de um predicado é calculado estaticamente como $\frac{1-sel}{custo}$, onde *sel* corresponde ao fator de seletividade estimado do predicado, e custo é seu custo de execução por tupla. A heurística *Pull-Up* é apresentada como alternativa à ordenação baseada em *rank*, a ser adotada com o uso de *cache*, em predicados com custo muito mais caros que os de predicados simples.

Este trabalho adota a heurística *Pull-Up*, incondicionalmente, dadas as características de custo dos programas científicos.

[2]inútil para estratégia Push-down

6.3.5 Análise conjunta das técnicas de execução

Nas seções anteriores apresentaram-se alguns experimentos que ilustram e comprovam a eficácia, individual, das técnicas de execução propostas. Nesta seção, analisa-se a adoção conjunta das quatro técnicas: transferência postergada*(TP)*; *caching (C)*; paralelismo entre a captura de *Blobs* e a execução de programas científicos *(P)*; e, por fim, a estratégia *Pull-Up (PU)*. Para isso, analisou-se o emprego das técnicas, em cinco diferentes planos de execução, gerados a partir da consulta *(a)* do *benchmark*. Os cinco planos podem ser observado na figura 6.7, e os meta-dados associados encontram-se exibidos na tabela abaixo.

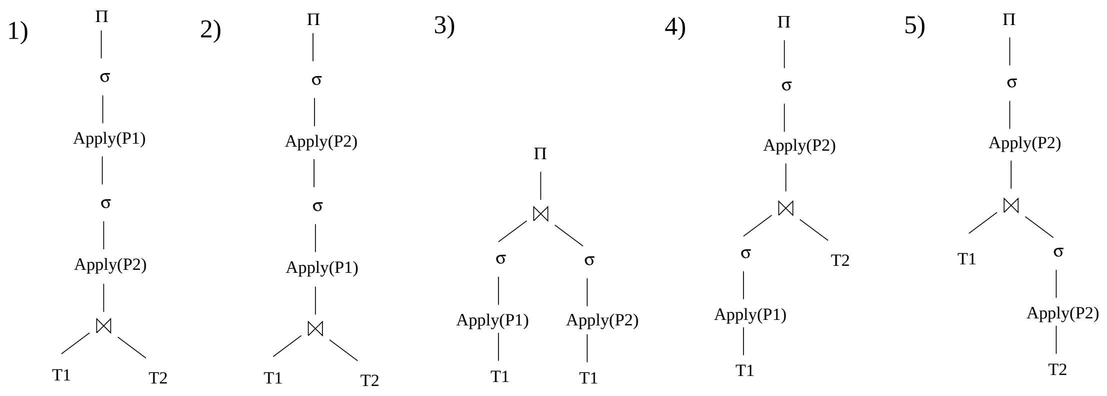


Figura 6.7: *Planos para comparação do emprego das técnicas de execução*

cardinalidade de T1	50 tuplas
cardinalidade de T2	300 tuplas
número de distintos de T1.b1	50
número de distintos de T2.b2	150
seletividade de T1 $\bowtie$ T2	0.6
seletividade de pred. P1	0.8
seletividade de pred. P2	0.5
custo, por tupla, P1	200 s
custo, por tupla, P2	30 s
tempo médio de transferência de T1.b1	100 s
tempo médio de transferência de T2.b2	1 s

Para realizar a comparação entre as técnicas, foram definidos cinco cenários de execução. No primeiro, os planos são executados obedecendo a uma estratégia *ingênua*. Todos os operadores processam sequencialmente e não há tratamento para valores duplicados. Adicionalmente, *Blobs* são transferidos repetidas vezes, por sítios intermediários, antes de chegarem ao destino de processamento. No segundo cenário, executam-se os planos utilizando-se apenas a técnica *TP*. Nesse cenário, os planos são executados com repetidas chamadas de programas e transferências de *Blobs*, caso a entrada para os respectivos operadores contenha valores duplicados. O benefício da técnica *TP* aparece, quando os consumidores de *Blobs* encontram-se distantes das fontes e, ainda assim, obtém-se custo constante de transferência. O próximo cenário, *C*, adota apenas a técnica de *caching*, com a qual execuções dos planos geram transferências desnecessárias entre sítios intermediários, e a única redução obtida ocorre no número de chamadas aos programas. No quarto cenário, *TPC*, implementa-se as duas técnicas anteriores. Os planos de execução, nesse cenário, apresentam resultados melhores, porém ainda não contam com paralelismo entre a captura de *Blobs* e a execução de programas. O último cenário, *Todas*, implementa todas as quatro técnicas.

Discussão

A figura 6.8 apresenta o gráfico comparativo entre os resultados das execuções dos cinco planos analisados. Cada barra, no gráfico, representa a execução de um plano, segundo uma das estratégias; e está identificado com a sigla da estratégia e o número do plano. Verifica-se que as técnicas são sempre benéficas, independentemente do plano avaliado. O cenário, onde todas as técnicas são empregadas em conjunto, é sempre o melhor em cada um dos planos analisados; o que é razoável, dado o baixo custo adicional introduzido na sua adoção.

O plano *1* aparece como o melhor em todas os cenários, salvo o baseado na estratégia *TP*. O tempo de execução é minimizado, na medida em que o caminho crítico, *P1*, tem seu custo inteiramente reduzido. Por outro lado, seu custo, sobre o cenário *TP*, apresenta pior resultado que os verificados nos planos *3 e 4*, onde a avaliação desses ocorre antes da junção, geradora de valores duplicados.

O plano *3* é contrário à estratégia *Pull-Up*. Seus melhores resultados servem para reforçar a adoção das estratégias em conjunto, uma vez que se apresenta adequado, justamente, quando as técnicas que possibilitam o *Pull-Up* não são aplicadas em conjunto.

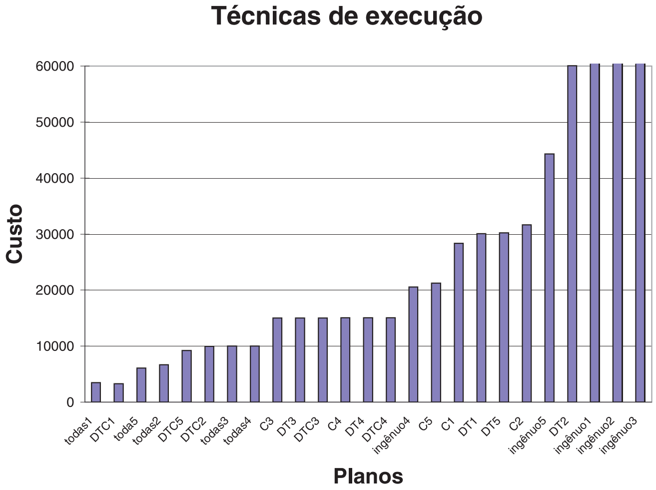


Figura 6.8: *Comparação entre as técnicas*

6.3.6 Conclusão

Nas seções anteriores descreveram-se os experimentos realizados para validação das técnicas de consultas com programas científicos. O principal resultado das avaliações mostra a importância da adoção das técnicas em conjunto. O baixo custo adicional introduzido por sua implementação reforça a motivação para adoção, em conjunto, das técnicas em um *adaptador configurado de programa*. Mecanismos mais finos podem ser desenvolvidos para que detectem situações onde, por exemplo, a captura de *Blobs* em paralelo não precise de ser implementada. Note-se, porém, que o ganho eventualmente obtido com essa decisão pode não apresentar grandes benefícios, uma vez que os *Blobs* capturados, em geral, devem ser gravados em disco, para o processamento do programa científico. Quanto à técnica de *transferência postergada*, ela traz sempre benefícios para transferências entre mais de um sítio. O custo de conexão deve ser mantido em níveis mínimos, principalmente para *Blobs* de tamanho próximo a *1 KByte*. O ganho obtido com a técnica de *caching* é muito significativo, considerando-se que age diretamente sobre o caminho crítico, e gera um custo adicional muito pequeno, limitado ao armazenamento, em memória, de dados de tipos básicos. Adicionalmente, para programas com apenas um parâmetro de entrada, de tipo *Blob*, simplifica-se a arquitetura, compartilhando-se um só *cache* entre o operador *Apply* e o operador *GetBlob*. Finalmente, a estratégia *Pull-Up* mostrou-se vantajosa para programas científicos, desde que apoiada pela implementação dos demais operadores. Note-se que sem a técnica de transferência postergada com *cache*, os resultados apresentados em [Hel98] não gerariam um plano eficiente para a execução de consultas científicas, visto que omitida a transferência de *Blobs*, muito comum nesse domínio. Adicionalmente, por não considerar paralelismo, as capturas de *Blobs* são realizadas sequencialmente à execução dos programas que os consome.

6.4 Validação dos algoritmos para paralelismo entre predicados caros

Nesta seção, apresentam-se as validações das abordagens Híbrida e *Picking Cherry*, que visam a minimizar o tempo de avaliação de predicados caros distribuídos. Inicialmente, descrevem-se os cenários de experimentação para, em seguida, discutirem-se os resultados obtidos.

6.4.1 Validação da estratégia Híbrida

Conforme apresentado na seção 5.4, a estratégia Híbrida flexibiliza planos com predicados caros, oferecendo ramos independentes com diferentes seqüências *pipeline*. As diferentes ordens de avaliação de predicados proporcionam adequação dinâmica da execução, frente a variações nas estimativas da seletividade de predicados e do número de valores distintos em atributos.

De modo a validar a estratégia híbrida, implementou-se um protótipo de máquina de execução para predicados distribuídos e um simulador. Inicialmente, descreve-se o ambiente de experimentação.

O cenário para validação

O cenário avaliado apresenta uma relação $R = \{\underline{k}, a, b\}$, e dois programas científicos P e Q. Uma consulta q avalia os predicados pr_p e pr_q sobre, respectivamente, os resultados de *P(a)* e *Q(b)*. Têm-se estimados os custos CP e CQ de execução de P e Q, por tupla, além dos fatores de seletividade $sf(pr_p)$ e $sf(pr_q)$. Finalmente, são estimados a cardinalidade de R, e o número de valores distintos de $R.a$ e $R.b$.

A relação R é produzida por um algoritmo de geração aleatória de valores, segundo parâmetros que definem a cardinalidade, o número de valores distintos em atributos, e o fator de seletividade dos predicados caros. O algoritmo para geração de valores distintos é resumidamente apresentado no anexo A.1.

A adequação do arquivo produzido pelo algoritmo descrito acima, aos fatores de seletividade, é proporcionada pela geração de valores em uma coluna adicional,com domínio {*0,1*}, indicativos do resultado da avaliação do predicado sobre o valor na respectiva tupla. Uma versão resumida do algoritmo encontra-se no anexo A.2.

A arquitetura

A máquina de execução foi construída utilizando-se uma arquitetura distribuída, composta de três PC Pentium III, cada um com 128 MBytes de memória RAM. As diferenças de desempenho, entre as máquinas foram compensadas com o devido balanceamento no tempo de execução dos programas, de forma que, relativamente, às máquinas apresentam o mesmo poder de processamento.

A relação R encontra-se alocada na máquina *m1*. Nas máquinas *m2* e *m3* implementam-se os adaptadores de programas P e Q.

Em cada máquina, executam-se duas instâncias de um dos programas. Na máquina *m2*, executam-se as duas instâncias de P, enquanto na máquina *m3* encontram-se as instâncias de Q.

A comunicação entre os processo dá-se por meio de gravação de filas no NFS. As instâncias *P1* e *Q2*, acessam diretamente a relação R e gravam, na fila de saída, as tuplas selecionadas. Essas tuplas são, então, processadas pelas instâncias *Q1* e *P2*, respectivamente. Para cada valor processado por um programa, um arquivo com o mesmo nome é gerado e usado como um *caching*. O *cache* é compartilhado por ambas as instâncias do mesmo programa.

A execução dos programas científicos foi simulada por um *loop*.

O problema com a experimentação

A validação por meio de experimentação mostrou-se difícil, uma vez que a execução de cada cenário requeria a preparação do ambiente distribuído, o que inclui geração dos arquivos de dados, limpeza dos diretórios temporários e inicialização dos processos remotos. Além disso, cada experimento despendia muito tempo (vários minutos). Esses fatores motivaram o desenvolvimento de um simulador de execução.

O simulador processa relações, que foram geradas segundo o mesmo método apresentado anteriormente. Ao invés de processar programas, sua execução é simulada por meio de incremento de contadores. A alteração de escalonamento entre *threads* *P1* e *P2*, onde *P1* apresenta maior prioridade, foi modelada a partir de um valor Φ, computado como função do número de valores distintos processados, e do número de avaliações de predicados com resultado falso. Inversamente, a *thread* secundária retorna o controle à primária, assim que sua instância correspondente, de maior prioridade, recebe uma tupla.

A estratégia mostrou-se bastante eficaz, na medida em que, obteve-se uma taxa de 0,1 segundos por execução de cenário. Adicionalmente, verificou-se a correção do modelo do simulador calibrando-o, frente a execuções reais do protótipo de máquina de execução. Em média as variações entre os resultados apontados pelo simulador e pela máquina de execução, foram de apenas 0,1%.

Resultados

Apresentam-se nesta seção, as comparações de desempenho entre a execução paralela de fluxo de dados e a estratégia híbrida, baseadas na execução do simulador. Considera-se a relação R, com cardinalidade de 200 tuplas. Variam-se os valores de cinco parâmetros: número de valores distintos de $R.a$ e $R.b$, seletividade de pr_p e pr_q, e custo relativo de avaliação dos predicados, CP e CQ. Para cada parâmetro, computa-se um valor randômico, compreendido entre os limites máximos e mínimos da tabela abaixo. A combinação de cinco valores gerados forma um cenário de avaliação.

Para garantir a estabilidade dos resultados, foram computados 10.000 cenários; para cada cenário geraram-se 20 arquivos. Cada arquivo apresenta um perfil de execução, segundo os algoritmos de geração de dados e seletividade. Para cada um dos 200.000 arquivos, efetuam-se quatro execuções: a execução segundo a "melhor" e a "pior" seqüência de predicados, para cada uma das estratégia, pipeline e híbrida.

Descrição	valor mínimo	valor máximo
número de distintos de R.a	20	200
número de distintos de R.b	20	200
seletividade de pred. P	0.1	0.9
seletividade de pred. Q	0.1	0.9
custo relativo	1	5

Baseado nos resultados obtidos, apresentam-se as seguintes observações. Primeiramente, verifica-se que muito embora estejam disponíveis os parâmetros para determinação da ordem mais eficaz para avaliação de predicados, esta pode não ser de fato a melhor. Observou-se que, em cerca de 1/3 dos cenários, a melhor ordem para o processamento em *pipeline* varia entre arquivos. Como se havia previsto, distribuições não homogêneas dos valores distintos e variações da seletividade prevista alteram as premissas assumidas durante a análise estática da consulta.

Em segundo lugar, a execução da estratégia híbrida, para uma dada ordem de predicados, acarreta em tempo de resposta inferior ao do *pipeline*, em 89% dos casos; ou igual, nos 11% restantes. Ao analisarem-se as execuções, onde a ordem adotada como melhor estaticamente, não se reflete em tempo de execução, a estratégia *pipeline* reporta tempo maior, em 53% dos casos. Tomando-se as execuções, onde a ordem decidida estaticamente se confirma correta, a perda é de apenas 5%.

Nos casos mais extremos de execução da estratégia *pipeline*, 14% do total de execuções, o tempo de resposta varia entre o 2 e até 13 vezes o correspondente tempo na estratégia híbrida. Em tais hipóteses, ou o fator de seletividade é muito alto, para o predicado mais alto da seqüência (entre 10% e 20%), ou o número de valores distintos é próximo a 200. Nesses casos, alterando-se a ordem de execução de predicados, produz-se uma execução significativamente mais eficiente, na medida em que as tuplas serão quase todas eliminadas pelo predicado altamente seletivo.

Finalmente, o aspecto mais forte da estratégia híbrida é que o tempo de execução não varia significativamente com alterações de prioridades entre os ramos. Em média, as execuções da estratégia híbrida, considerando-se a ordem errada de predicados, ficam 12% piores que as correspondentes, seguindo-se a ordem correta. Por outro lado, nesta mesma situação, a estratégia *pipeline* aumenta o tempo de execução em 62%. Os pontos extremos são ainda mais graves na estratégia *pipeline*, onde se chegou a obter execuções até 14 vezes mais lentas que na ordem correta; enquanto a degradação de execução do mesmo plano, pela estratégia híbrida, foi de 30%. Na pior execução da estratégia híbrida, obteve-se degradação de 72% no tempo de resposta, em relação ao melhor plano.

A figura 6.9 resume as medições. Os gráficos mostram a razão entre os tempos de execução dos planos nas estratégias *híbrida* e *pipeline*, considerando-se, em *(a)*, planos em que a ordem correta foi escolhida e, em *(b)*, planos onde se optou por uma ordem que não se verificou como sendo a melhor, em tempo de execução.

6.4.2 Validação do algoritmo Picking Cherry

O objetivo da validação é verificar, experimentalmente, os ganhos obtidos no tempo de resposta de consultas, mediante a abordagem de avaliação de consultas com predicados caros baseada na exploração de dependências funcionais entre os atributos de relações intermediárias. Os experimentos validam, adicionalmente, a adoção do algoritmo guloso, baseada no aproveitamento do *fanout específico* de valores consumidos por predicados caros, e da estratégia de avaliação desses, segundo o modelo de paralelismo independente.

Estratégias de experimentação

A experimentação utilizou os mesmos procedimentos, para geração de cenários de teste, que os apresentados na experimentação da estratégia híbrida. Considerou-se a mesma consulta que a anterior, compreendendo dois predicados caros, avaliados a partir de uma relação $R = \{\underline{k}, a, b\}$. Foram gerados 20 arquivos, com 200 tuplas cada um. Os parâmetros de número de valores distintos, de seletividade estimada de predicados e de custos de predicados, foram gerados aleatoriamente, segundo os intervalos especificados na tabela apresentada na validação da estratégia híbrida.

Comparou-se o tempo de resposta de três estratégias diferentes de execução: *híbrido*, *ingênuo* e *PC*. Adicionalmente, implementou-se um algoritmo de exploração exaustiva da ordem de avaliação dos valores em a e b. A seguir, descrevem-se as estratégias ingênua e exaustiva.

A estratégia ingênua utiliza as estruturas de dados preparadas pelo algoritmo PC, porém, diferentemente desse, a escolha do valor a ser processado, realiza-se aleatoriamente. A implementação armazena valores distintos de um atributo em um vetor, sem qualquer ordem. Inicialmente, escolhe-se ao acaso uma posição do vetor e se envia o valor correspondente ao respectivo predicado. O retorno da avaliação atualiza o estado das arestas do grafo bi-partido e retorna, ao predicado, o valor fisicamente mais próximo, ainda não processado (ou eliminado).

O objetivo do algoritmo exaustivo é identificar as ordens de avaliação, por P, dos valores de *(a)*, e, por Q, dos valores de *(b)*, que produzam o menor custo de execução paralela. O algoritmo recebe, como entrada, as tabelas *hashing*, que contêm os valores distintos de *(a)* e de *(b)*. Define-se, como bloco de avaliação (BA), um conjunto de n valores distintos de b, tais que $n = CP/CQ$. O algoritmo é apresentado no anexo **??**.

O algoritmo exaustivo é utilizado como parâmetro de qualidade da estratégia gulosa adotada. Infelizmente, dada a explosão combinatória no número de planos analisados, conseguiu-se executá-lo com, no máximo, 20 tuplas.

Resultados

Nessa seção apresentam-se os resultados iniciais da avaliação da abordagem *Picking Cherry*. Os resultados da execução das três estratégias estão representadas no gráfico da figura 6.10. Os pontos, no gráfico, representam valores médios, de tempo de resposta, de execuções sobre os 20 cenários gerados.

Analisaram-se dois aspectos da abordagem PC: a estratégia de execução independente e o algoritmo guloso baseado no *fanout*.

O desempenho da abordagem PC equipara-se ao obtido com a estratégia híbrida em 80% dos cenários.

Em 20% dos cenários, PC mostra-se, em torno de 14%, melhor que a estratégia híbrida. Nesse casos, verifica-se o benefício da combinação da estratégia paralela independente, com o algoritmo guloso baseado no *fanout*. As execuções, nesses cenários, apresentam fatores de seletividade próximos e diferença de custo entre os programas que é anulada pelo benefício do *fanout*. O resultado dessa combinação é uma execução que, na estratégia híbrida, não é suficiente para direcionar o fluxo de dados para o ramo de menor prioridade, produzindo um ligeiro acréscimo no tempo de resposta, frente à estratégia de execução independente com algoritmo guloso.

O algoritmo ingênuo mostrou-se inferior ao PC, em 99,5% dos casos. Em 30% dos cenários, o tempo de resposta do primeiro fica entre 10 e 30% maior que do segundo e, em apenas 1 cenário, se mostrou 1,2% melhor que o do PC. Retorna-se ao problema inicial, para se encontrar resposta a essa diferença. Cumpre é determinar o menor número de valores n que, processados, eliminem todas as arestas do grafo bi-partido. Um algoritmo qualquer tem probabilidade n/k de processar um desses valores, sendo k o número de valores distintos em ambos os predicados. O algoritmo PC utiliza a estratégia gulosa para se aproximar da escolha dos n melhores valores. A estratégia ingênua depende da diferença $k - n$; quanto maior, menor é a chance de se obter um desempenho equiparável ao obtido pela estratégia PC.

Por fim, comparou-se o algoritmo PC com a estratégia exaustiva. Executou-se 10 cenários, mantendo-se a relação R com 20 tuplas e gerando-se, randomicamente, o número de valores distintos em a e b. Os resultados mostraram que em 6 casos a estratégia gulosa processou os valores na cobertura mínima do grafo.

6.5 Sumário

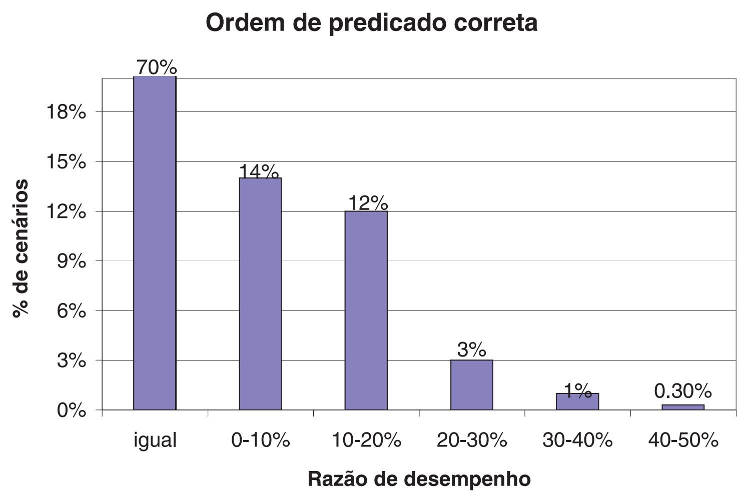
Neste capítulo apresentaram-se as validações das propostas formuladas para integração otimizada de programas científicos em consultas distribuídas.

Inicialmente, validaram-se as otimizações para implementação dos operadores de execução de programas. O objetivo da validação é verificar o benefício da implementação dessas otimizações e identificar a importância de sua adoção para o contexto de execução de consultas científicas distribuídas.

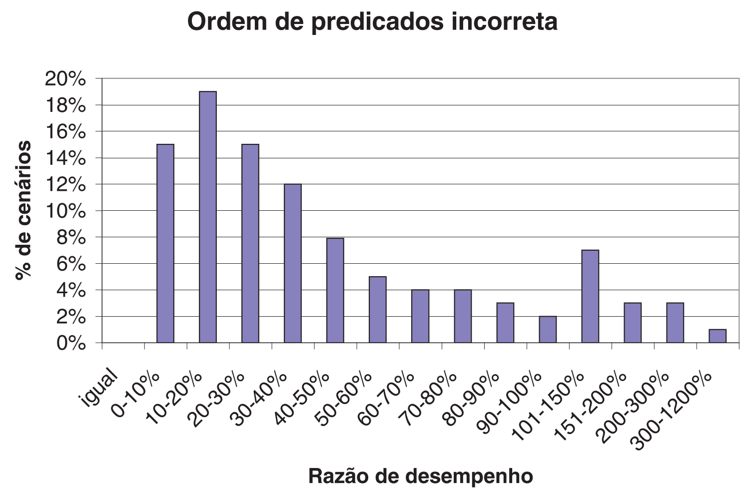
O principal resultado das avaliações mostra a importância da adoção das técnicas em conjunto. O baixo custo adicional introduzido por sua implementação reforça a motivação para adoção, em conjunto, das técnicas em um *adaptador configurado* *de programa*. Mecanismos mais finos podem ser desenvolvidos para que detectem situações onde, por exemplo, a captura de *Blobs* em paralelo não precise ser implementada. Note-se, porém, que o ganho eventualmente obtido com essa decisão, pode não apresentar grandes benefícios, uma vez que os *Blobs* capturados, em geral, devem ser gravados em disco, para o processamento do programa científico. Quanto à técnica de *trasferência postergada*, ela traz sempre benefícios para transferências entre mais de um sítio. O custo de conexão deve ser mantido em níveis mínimos, principalmente para *Blobs* de tamanho próximo a *1 KByte*. O ganho obtido com a técnica de *caching* é muito significativo, considerando-se que age diretamente sobre o caminho crítico, e gera um custo adicional muito pequeno, limitado ao armazenamento, em memória, de dados de tipos básicos. Adicionalmente, para programas com apenas um parâmetro de entrada, de tipo *Blob*, simplifica-se a arquitetura, compartilhando-se um só *cache* entre o operador *Apply* e o operador *GetBlob*. Finalmente, a estratégia *Pull-Up* mostrou-se vantajosa para programas científicos, desde que apoiada pela implementação dos demais operadores. Note-se que sem a técnica de transferência postergada com *cache*, os resultados apresentados em [Hel98] não gerariam um plano eficiente para a execução de consultas científicas, visto que omitida a transferência de *Blobs*, muito comum nesse domínio. Adicionalmente, por não considerar paralelismo, as capturas de *Blobs* são realizadas sequencialmente à execução dos programas que os consome.

Em seguida avaliaram-se as técnicas de execução paralela híbrida e *Picking Cherry*. A validação da estratégia híbrida foi realizada por meio de simulação . Compararam-se os tempos de resposta de consultas em 200.000 cenários distintos. Os resultados reforçam dois aspectos importantes: primeiramente, verifica-se que mesmo quando se conhecem os valores de meta-dados sobre os objetos da consulta, a melhor ordem de execução pode não ser a indicada estaticamente, uma vez que essas estimativas podem variar, em tempo de execução . O efeito de variações desse tipo sobre o plano de execução de consultas, que implementem paralelismo de fluxo de dados, pode levar a um custo adicional de até 14 vezes superior ao custo de um plano que executasse os predicados caros segundo a ordem que se revela a mais eficiente. Por outro lado, no pior cenário de execução da estratégia híbrida, tem-se custo 72% superior, que o custo correspondente, adotando-se a ordem de avaliação dos predicados mais eficiente. Essa observação leva ao segundo aspecto importante da técnica que é sua estabilidade frente à variações entre a melhor ordem a ser avaliada.

Finalmente, verifica-se a valia da abordagem *PC*, frente à uma estratégia ingênua de execução e à estratégia híbrida. A abordagem *PC* foi implementada adotando-se a estratégia gulosa de execução , onde os valores são processados em ordem descendente de *fanout efetivo*. A validação realizou-se com a implementação das estratégias em uma máquina de execução distribuída. Compararam-se, experimentalmente, a execução de 20 cenários. Os resultados iniciais apontam para boas perspectivas dessa abordagem. O algoritmo guloso apresentou resultado, no mínimo igual ao obtido pela estratégia híbrida, com ganhos de até 14%. Os resultados, porém não são conclusivos, em função do pequeno número de cenários avaliados. Porém, na medida em que todas essas estratégias encontram-se implementadas, validações mais extensivas podem ser realizadas, sem muita dificuldade.



(a)



(b)

Figura 6.9: *Comparação entre a estratégia Híbrida e pipeline*

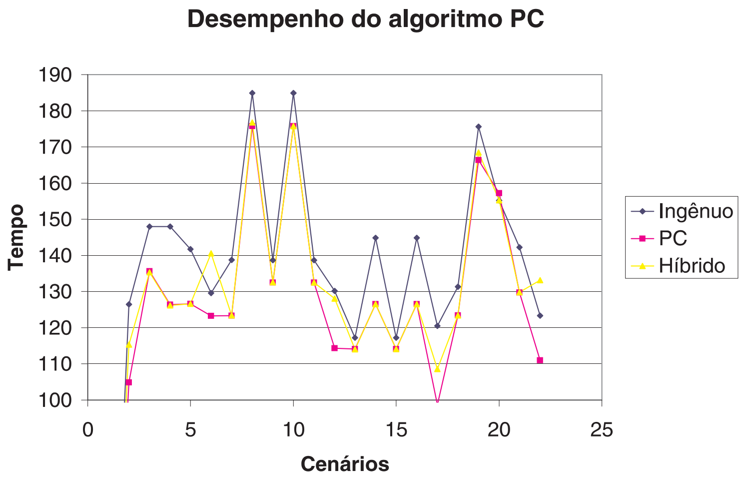


Figura 6.10: *Desempenho do algoritmo PC frente a estratégia híbrida e ingênua*

Capítulo 7

Conclusão

7.1 Introdução

Este trabalho propõe uma estratégia para execução otimizada de consulta científicas sobre sistemas de bancos de dados distribuídos. A motivação para o trabalho surgiu nos projetos Thetis e Decair que buscam desenvolver tecnologia para apoio ao trabalho integrado entre grupos de cientistas ambientais. Dentre as linhas de pesquisa exploradas nesses projetos, encontra-se o desenvolvimento do sistema *middleware* Le Select para a integração de dados e programas científicos. O sistema foi projetado com as características de sistemas heterogêneos com acoplamento fraco. Sua proposta de integração assemelha-se ao sistemas ditos mediadores, na literatura, onde a homogeneização de modelos das fontes de dados para o Modelo relacional, o Modelo de integração do sistema, é realizado via adaptadores.

No sistema Le Select, consultas científicas integram acesso a dados à execução de programas, ambos distribuídos na internet.

Assim como ocorre na publicação de dados, para a publicação de programas requer-se o desenvolvimento de um adaptador, que implementará mecanismos necessários para sua integração ao processo de avaliação de consultas.

Programas científicos, no contexto deste trabalho, apresentam características importantes para sua integração ao sistema de avaliação de consultas. Em primeiro lugar, os programas são vistos como caixas-pretas pelo sistema de avaliação de consultas. As únicas informações disponíveis sobre programas científicos constam de sua interface e de uma estimativa do tempo médio de execução por tupla, além de uma estimativa do fator de seletividade, para predicados sobre seus resultados. Em segundo lugar, esses programas são, em geral, implementados em linguagens de programação como C e FORTRAN, e fazem uso de bibliotecas específicas, como as que realizam certos tratamentos em imagens de satélite. Dessa forma, sua avaliação precisa ser realizada no sítio de publicação que conta com infra-estrutura necessária para seu processamento. Adicionalmente, programas científicos processam dados de tipos simples e complexo. Como exemplo de dados de tipo complexo, consideram-se imagens capturadas por satélites, matrizes com dados amostrais de vários índices ambientais etc. Finalmente, com respeito aos operadores tradicionais da álgebra relacional, a execução de programas apresenta um custo, avaliado no tempo médio de cada execução, extremamente alto. O processamento sobre uma imagem de alta resolução pode variar de alguns segundos a até algumas horas.

7.2 Um exemplo de problema

Um exemplo de consulta científica que integra programas com dados, incluindo dados de tipo complexo (BLOBs), é apresentado abaixo:

Select pr.nome,e.tipo-solo,e.ecosistema
  From PrecReg pr, PolReg po, Env e
  Where pr.nome=po.regnome and
    pr.nome= e.região and
    Prec(pr.prec) $>$ 1.5 and
    Pol(po.medidas) $<$ 0.6

□

Nesse exemplo, empregam-se dois programas científicos, *Prec* e *Pol*. Ambos recebem como parâmetro de entrada dados de tipo Blob, *prec* e *medidas*, respectivamente. Consideram-se as tabelas e os programas publicados por adaptadores localizados em sítios distintos e integrados por servidores Le Select.

O problema tratado por este trabalho é o de conceber uma estratégia que minimize o tempo total de execução e maximize a utilização de recursos distribuídos, para consultas científicas como a representada acima.

7.3 A solução proposta

A solução propõe a geração de planos de execução de consultas que paralelizam a execução de programas distribuídos. A estratégia adotada é apoiada por um conjunto de técnicas de execução que flexibiliza o posicionamento de programas científicos no plano de execução de consultas distribuído. Em virtude disso, a execução de consultas científicas é otimizada pela combinação de estratégias de paralelismo independente e de fluxo de dados.

Inicialmente, esse trabalho estende a álgebra relacional com os operadores *Apply* e *GetBlob*. O primeiro representa a execução de programas científicos em um plano de execução de consulta, enquanto o segundo é um operador físico para transferência de Blobs.

O conjunto de técnicas de execução para programas científicos proposto nesse trabalho encontra-se apresentado abaixo:

- Transferência Postergada - seu objetivo é evitar transferências desnecessárias de dados de tipo Blob pela rede. A técnica compreende a implementação do operador GetBlob, e o mapeamento de valores de tipo Blob a seus identificadores únicos, URIs, que os substituem, durante o processamento de predicados simples.

- Caching - esta técnica, proposta inicialmente em [HN96] para evitar chamadas repetitivas de programas em consultas, foi estendida e combinada com o processamento de transferência de dados de tipo Blobs.

- Paralelismo entre execução de programas e transferência de Blobs - considerando-se o custo alto de ambas as operações e a dependência de dados entre elas, essa técnica propõe um plano paralelo de fluxo de dados compreendendo a execução de programas e a captura de seus dados de tipo Blob.

A combinação dessas três técnicas flexibiliza a alocação da operação *Apply* no plano de execução de consulta distribuída ($PECD$), uma vez que: a captura dos dados de entrada torna-se independente da distância, no plano, entre as fontes de dados e o operador *Apply*; e a chamada de programas não é afetada por duplicidades de valores produzidas por operações de junções que antecedam sua execução. Com essa observação e considerando-se o custo de execução de programas científicos que inclue a captura de seus dados de entrada, adota-se a heurística *Pull-Up* para alocação de programas científicos no *PECD*. Segundo essa heurística, as operações *Apply* sucedem todos os predicados simples, seleções e junções, no *PECD*. A heurística tem como objetivo beneficiar-se de todas as reduções sobre os valores distintos de entradas para os programas, minimizando o número de chamadas aos programas.

Com a adoção da heurística *Pull-Up*, resta definir a estratégia de execução de programas que minimize o custo total da consulta e maximize a utilização de recursos. Propõe-se as seguintes heurísticas para geração de planos paralelos:

- Paralelismo independente - para programas sem relação de dependência de dados.
- Paralelismo de fluxo de dados - para programas com dependência de dados (i.e. funções compostas).
- Estratégia híbrida - para programas com avaliação de predicados sobre o resultado de sua execução.

A estratégia híbrida é uma abordagem original para a execução de predicados caros distribuídos proposta nesse trabalho. Durante as experimentações, observou-se a influência de características do momento da execução de consultas, que afastavam planos gerados segundo estimativas estáticas de uma execução com uso total de recursos. Tais características eram principalmente variações de estimativa no fator de seletividade e no número de valores distintos em atributos de entrada de programas. Variações nos valores estimados desses parâmetros produzem alterações no fluxo de dados entre predicados alocados em cadeias, fazendo com que aqueles alocados em posições mais altas não recebam dados para processamento. A estratégia híbrida soluciona esse problema propondo um *PECD*, onde ramos independentes são compostos por cadeias de predicados caros. Em cada ramo, aloca-se um dos predicados caros como o primeiro da cadeia de execução. Dessa forma, garante-se a utilização total de recursos. Mecanismos de prioridades entre linhas de execução definem cadeias cuja ordem seja considerada prioritária.

Adicionalmente, propõe-se uma abordagem para a execução de predicados caros, baseada nas relações de dependência funcional entre os valores de atributos de entrada em programas científicos. Nessa nova abordagem, chamada de *Picking Cherry*, computa-se a relação intermediária produzida pelos predicados simples da consulta. A partir dessa relação, supondo-se uma consulta composta de dois predicados caros, constrói-se um grafo bi-partido, contendo em cada parte o conjunto dos valores distintos de entrada a cada um dos programas. Programas são executados segundo o modelo de paralelismo independente, que garante o uso total de recursos distribuídos. Propõe-se uma estratégia de execução que permite beneficiar-se de reduções entre predicados caros. Para o processamento de cada um dos conjuntos, define-se uma estratégia gulosa de execução baseada no *fanout* de cada um dos valores.

Para validar as técnicas de execução e as estratégias híbrida e *Picking Cherry* foram construídos um protótipo de otimizador de consultas exaustivo, com modelo de custo paralelo, além de uma máquina de execução distribuída para processamento das estratégias paralelas híbridas e de fluxo de dados. Adicionalamente, a validação da estratégia híbrida contou com a implementação de um simulador que propiciou a avaliação de 200.000 cenários diferentes para comparação entre as estratégias.

7.4 Trabalhos futuros

Como trabalhos futuros considera-se a integração de outros tipos de programas científicos, incluindo os que apresentam interface do tipo (tabela - tupla), conhecidos, como funções de agregação. Adicionalmente, deseja-se analisar a geração de planos que considere a existência de programas replicados em diferentes sítios. Ainda com relação a integração de programas em consultas, é interessante explorar mecanismos que enriqueçam a descrição das interfaces dos adaptadores, possibilitando validações mais consistentes dos dados de entrada à programas.

Com relação à estratégia híbrida, seria interessante verificar a validade de alguns aspectos da abordagem *Picking Cherry* em sua implementação. Por exemplo, poder-se-ia considerar o *fanout* médio dos valores de atributos nas entradas de programas como parte do cálculo do *ranking* de cada predicado. Adicionalmente, pode-se verificar a adequação da estratégia independente de execução, baseada em valores, proposta na abordagem *Picking Cherry*. A abordagem *Picking Cherry* apresenta ainda várias questões não exploradas nesse trabalho. Pode-se analisar outras estratégias de cobertura do grafo bi-partido, além de definir formalmente limites de ganhos para a estratégia.

Bibliografia

[AZ96] G. Antoshenkov e M. Ziauddin, "Query Procesing and Optimization in Oracle Rdb", em *VLDB Journal*, Vol. 5, Num. 4, 1996.

[ACPS96] S. Adali, K. Candan, Y. Papakonstantinou e V. Subrahmanian, "Query Caching and Optimization in Distributed Mediator Systems", em *Proc. Intl. Conf. on Management of Data*, Montreal, Canadá, 1996.

[AFTU96] L. Amsaleg, M. J. Franklin, A. Tomasic e T. Urhan, "Scrambing Query Plans to Cope with Unexpected Delays", em *PDIS Conf.*, Miami, USA, 1996.

[AH00] R. Avnur e J. Hellerstein, "Eddies: Continuously Adaptive Query Processing", em *Proc. Intl. Conf. on Management of Data*, Dallas, Texas, 2000.

[BC81] P. Bernstein e D. Chiu, "Using Semi-Join to Solve Relational Queries", em *Journal of the Association for Computing Machinery*, 28(1), Janeiro, 1981.

[BFMV00] L. Bouganim, F. Fabret, C. Mohan e P. Valduriez, "Dynamic Query Scheduling in Data Integration Systems", em *Proc.* 16^{th} *International Conference on Data Engineering*, São Diego, CA, Março, 2000.

[BT98] P. Bonnet e A. Tomasic, "Partial Answers for Unavaiable Data Sources", em *Conf. on Flexible Query Answering Systems*, Roskilde, Dinamarca, 1998.

[BFPV01] L. Bouganim, F. Fabret, F. Porto e P. Valduriez, "Processing Queries with Expensive Functions and Large Objects in Distributed Mediator

Systems", em *Proc. 17th Intl. Conf. on Data Engineering (ICDE)*, Heidelberg, Alemanha, Abril 2001.

[BW96] K. Brown e B. Whitenack, "Crossing Chasm: A Pattern Language for Object-RDBMS Integration", em *Pattern Languages of Program Design 2*, Addison-Wesley, 1996, pp. 227-238.

[Brow96] K. Brown, "Crossing Chasm: The Architectural Patterns", em *Pattern Languages of Programs (PloP)*, Monticello, Illinois, 1996.

[Cat94] R. G. G. Cattell, "Object Data Management", em *Addison Wesley, Reading*, ed. revisada, Massachusetts, USA, 1994.

[CDSK98] S. Cluet, C. Delobel, J. Simeon, e K. Smaga, "Your Mediators Need Data conversion!", em *Proc. Intl. Conf. on Management of Data*, Seattle, Washington, 1998.

[CBBC+89] S. Chakravarthy, B. Blaustein, A. P. Buchman, M. J. Carey, U. Dayal, et all, "HIPAC: A research project in active, time constrained database management", em *Relatório técnico XAIT-89-02*, XEROX Advanced Information Technology, Cambridge, Massachusetts, USA, Julho, 1989.

[CGK89] D. Chimenti, R. Gamboa e R. Krishnamurthy, "Towards an open architecture for LDL", em *Proc. 15th Intl. Conf. on Very Large Data Bases*, Amsterdã, Holanda, Agosto, 1993.

[GBD94] F. Gargouri, F. Biufars, C. Ducateau, "Relational Implementation of Object Oriented Information Systems Design Using a Generic Model", em *Proc. of Object Oriented Information Systems*, London, 1994, pp 115-129.

[CHS+95] M. Carey, L. Haas, P. Schearz, M. Arya, W. Cody, R. Fagin, M. Flickner, A. Luniewski, W. Niblack, D. Petkovic, J. Thomas II, J. Willians e E. Wimmers, "Towards Heterogeneous Multimedia Information Systems: The Garlic Approach", em *IEEE RIDE Workshop*, Taipei, 1995.

[Che76] P. Chen, "The Entity-Relationship Model-Toward a unified model of data", em *ACM. Transaction Database Systems*, 1, Março, 1976.

[CD94] S. Cook e J. Daniels, "Designing Object Systems: Object-Oriented Modelling with Syntropy", em *Prentice Hall*, 1994.

[CS93] S. Chaudhuri e K. Shim, "Optimization in the Presence of Foreign Functions", em *Proc.* 19^{th} *Intl. Conf. on Very Large Data Bases*, Dublin, Ireland, Agosto, 1993.

[CS96] S. Chaudhuri e K. Shim, "Optimization of Queries with User-Defined Predicates", em *Proc.* 22^{th} *Intl. Conf. on Very Large Data Bases*, Bombay, India, Setembro, 1996.

[CS99] S. Chaudhuri e K. Shim, "Optimization of Queries with User-Defined Predicates", em *ACM Transactions on Database Systems*, 24(2), Junho, 1999.

[DD98] C. J. Date, H. Darwen, "Foundation for Object/Relational Databases - The Third Maniphesto", em *Addison-Wesley*, 1998.

[Deca] Projeto Caravel, "Aplicação do domínio da ciência ambiental lidando com modelos de qualidade do ar", INRIA, França, em *http//caravel.inria.fr/Fcontract_DECAIR.html*

[Daya87] U. Dayal, "Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates and Quantifiers", em *Proc.* 13^{th} *Intl. Conf. on Very Large Data Bases*, Brighton, Inglaterra, Agosto, 1987.

[DeWG92] D. J. DeWitt e J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems", em *Communications of the ACM*, Vol. 35, Num. 6, pp.85-98, Junho, 1992.

[DeWKOS+84] D. J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker e D. Wood, "Implementation Techniques for Main Memory Database Systems", em *Proc. Intl. Conf. on Management of Data*, Boston, Junho, 1984.

[EN94] R. Elmasri e S. Navathe, "Fundamentals of Database Systems", em *Addison-Wesley*, 2^a ed., 1994.

[FMLS99] D. Florescu, I. Monolescu, A. Levy e D. Suciu, "Query optimization in the Presence of Limited Access Patterns", em *Proc. Intl. Conf. on Management of Data*, Filadélfia, USA, Junho, 1999.

[GHJV95] E. Gamma, R. Helm, R. Johnson e J. Vlissides, "Design Patterns", em *Addison-Wesley*, 1995.

[GPQ+97] H. Garcia-Molia, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagin, J. Ullman and J. Widom, "The TSIMMIS Project: Integration of Heterogeneous Information Sources", em *Journal of Intelligent Information Systems*, 8(2), pp 177-132, Março, 1997.

[GHK92] S. Ganguly, W. Hasan e R. Krshnamurthy, "Query Optimization for Parallel Execution", em *Proc. Intl. Conf. on Management of Data*,São Diego, CA, Junho, 1992.

[Grae93] G. Graef, "Query Evaluation Techniques for Large Databases", em *ACM Computing Surveys*, 25(2), Junho, 1993.

[Har88] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", em *Science Comput. Prog.*, 8, pp 231-274, 1988.

[Harel+96] D. Harel e E. Gery, "Executable Object modeling with Statecharts", em *Proc.* 18^{th} *International Conference on Software Engineering*, 1996.

[HKWY97] L. M. Haas, D. Kossmann, E. L. Wimmers e J. Yang, "Optimizing Queries Across Diverse Data Sources", em *Proc.* 23^{rd} *Intl. Conf. on Very Large Data Bases*, Atenas, Grécia, Agosto, 1997.

[HY79] A. R. Hevner e S. B. Yao, "Query Processing in Distributed Database Systems", em *IEEE Trans. Software Eng.*, Vol.5(3), pp. 177-182, Março, 1979.

[HS93] J. Hellerstein e M. Stonebraker, "Predicate Migration: Optimizing Queries with Expensive Predicates", em *Proc.* 12^{th} *Intl. Conf. on Management of Data*, Washington, USA, Junho, 1993.

[Hel94] J. Hellerstein, "Predicate Migration Placement", em *Proc.* 13^{th} *Intl. Conf. on Management of Data*, Mineápolis, USA, Maio, 1994.

[HN96] J. Hellerstein e J. Naughton, "Query Execution Techniques for Caching Expensive Methods", em *Proc. 15th Intl. Conf. on Management of Data*, Montreal, Canadá, Junho, 1996.

[Hel98] J. Hellerstein, "Optimization Techniques For Queries with Expensive Methodos", em *ACM Transactions on Database Systems*, 23(2), Setembro, 1998.

[Hir96] R. Hirschfeld, "Three-Tier Distribution Architecture", em *Pattern Languages of Programs (PloP)*, Monticello, Illinois, 1996.

[IC91] Y. E. Ioannidis e S. Christodoulakis, "On the Propagation of Errors in the Size of Join Results", em *Proc. 11th Intl. Conf. on Management of Data*, Denver, Colorado, Junho, 1991.

[IFFLW99] Z. G. Ives, D. Florescu, M. Friedman, A. Levy e D. Weld, "An Adaptive Query Execution System for Data Integration", em *Proc. Intl. Conf. on Management of Data*, Filadélfia, USA, Junho, 1999.

[JK84] M. Jarke, J. Koch, "Query optimization in Database Systems", em *ACM Computing Surveys*, Vol.16, No.2, Junho, 1984.

[JM99] M. Jaedick e B. Mitschang, "User-Defined Table Operators: Enhancing Extensibility for ORDBMS", em *Proc. 25th Intl. Conf. on Very Large Data Bases*, Edinburgh, Escócia, Agosto, 1999.

[KJA93] A. Keller, R. Jensen, and S. Agarwal, "Persistence Software: Bridging Object-Oriented Programming and Relational Databases", em *Proceedings of 12th International Conf. on Management of Data*, Washington, USA, junho 1993.

[Kel98] W. Keller, "Object/Relational Access Layers A Roadmap, Missing Links and More Patterns", em *Proc. of 3rd European Conference on Pattern Languages of Programming (Europlop98)*, Bad Irsee, Alemanha, 1998.

[KC96] W. Keller and J. Coldewey, "Relational Database Access Layer", em *Pattern Languages of Programs (PloP)*, Monticello, Illinois, 1996.

[K95] W. Kim, "Modern Database Systems: The Object Model, Interoperability, and Beyond.", em *Addison-Wesley*, 1995.

[Koss00] D. Kossmann, "The State of the Art in Distributed Query Processing", em *ACM Computing Surveys*, Setembro, 2000.

[LNS90] R. J. Lipton, J. F. Naughton e D. A. Schneider, "Practical selectivity estimation through adaptative sampling", em *Proc. Intl. Conf. on Management of Data*, pp 1-11, Atlantic City, USA, Maio, 1990.

[LVZ93] R. S. G. Lanzelotte, P. Valduriez e M. Zaït, "On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces", em *Proc. 19^{th} Intl. Conf. on Very Large Data Bases*, pp. 493-504, Dublin, Irlanda, Agosto, 1993.

[LeSe] "A Mediator System Developed in the Caravel Project", INRIA, França, em *http://caravel.inria.fr/Fprototype_LeSelect.html.*

[LMSS95] A. Y. Levy, A. O. Mendelzon, Yehoshua Sagiv e Divesh Srvivastava, "Answering queries using views", em *Proc. of the ACM SIGMOD Symposium on Principles of Database Systems (PODS)*, São José, CA, 1995.

[Mic68] D. Michie, "*Memo* Functions and Machine Learning", em *Nature*, (218): 19-22, Abril, 1968.

[MPT+94] C. Mohan, H. Pirahesh, W. Tang e Y. Wang, "Parallelism in Relational Database Management Systems", em *IBM System Journal*, 33(2), 1994.

[MR00] M. Martinez e N. Roussopoulos, "MOCHA: A Self-Extensible Database Middleware System for Distributed Data Sources", em *Proc. 19^{th} Intl. Conf. on Management of Data*, Dallas, USA, Maio, 2000.

[MS99] T. Mayr e P. Seshadri, "Client-Site Query Extensions", em *Proc. 18^{th} Intl. Conf. on Management of Data*, Filadélfia, USA, Junho, 1999.

[MS79] C. L. Monma e J. Sidney, "Sequencing with Series-Paralel Precedence Constraints", em *Mathematics of Operations Research*, N. 4, 1979.

[MPLB98] R. N. Melo, F. Porto, F. Lima e A. Barbosa, "ECOHOOD: Constructing Configured DMBSs Based on Frameworks", em *XIII Simpósio Brasileiro de Banco de Dados*, Maringá, PR, Brasil, 1998.

[NGT98] H. Naacke, G. Gardarin e A. Tomasic, "Leveraging Mediator Cost Models with Heterogeneous Data Sources", em *Proc. 12^{th} Intl. Conf. on Data Engineering*, Orlando, Flórida, 1998.

[ÖV99] M. T. Özsu e P. Valduriez, "Principles of Distributed Database Systems", em *Prentice Hall*, Nova Jersey, USA, 1999.

[Papa97] Y. Papakonstantinou, "Query Processing in Heterogeneous Information Sources", em *Tese PHD, Universidade de Stanford*, 1997.

[PI97] V. Poosala e Y. Ioannidis, "Selectivity estimation without the attribute value independence assumption", em *Proc. 23^{rd} Intl. Conf. on Very Large Data Bases*, Atenas, Grécia, Agosto, 1997.

[PVsC98] F. Porto, M. Vianna e Silva e S. R. Carvalho, "Object Life-Cycles in Active Relational Databases", em *Proc. Techniques of Object_Oriented Languages and Systems (TOOLS-USA)*, Santa Bárbara, CA, Agosto, 1998.

[PCVsM99] F. Porto, S. R. Carvalho, M. Vianna e Silva e R. N. Melo, "Persistent Object Synchronization with Active Relational Databases", em *Proc. Techniques of Object_Oriented Languages and Systems (TOOLS-USA)* Santa Bárbara, CA, Agosto, 1999.

[SA80] P. Selinger e M. Adiba, "Access Path Selection in Distributed Database Management Systems", em *Proc. First Intl. Conf. on Data Bases*, pp. 204–215, 1980.

[SACL+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie e T. Price, "Access path selection in a relation database management system", em *Proc. Intl. Conf. on Management of Data*, Boston, USA, Junho, 1979.

[SL90] A. Sheth e J. Larson, "Federated database systems for managing distributed, heterogeneous and autonomous databases", em *ACM Computing Surveys*", 22(3):183-236, Setembro, 1990.

[SLR97] P. Seshadri, M. Livny e R. Ramakrishnan, "The Case for Enhanced Abstract Data Types", em *Proc. 23^{rd} Intl. Conf. on Very Large Data Bases*, Atenas, Grécia, Agosto, 1997.

[SDeW90] D. A. Schneider, D. J. DeWitt, "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocesor Database Machines", em *Proc. 16^{th} Intl. Conf. on Very Large Data Bases*, pp 469-480, Brisbane, Austrália, Agosto, 1990.

[SKS96] A. Silberschatz, H. Korth e S. Sudarshan, "Database System Concepts", em *McGraw-Hill*, 3^a ed., 1996.

[Stol63] R. R. Stoll, "Set Theory and Logic", em *W.H. Freeman*, São Francisco, CA, 1963.

[Tane95] A. Tanenbaum, "Distributed Operating Systems", em *Prentice Hall*, 1995.

[Thet] Projeto Caravel, "Aplicação do domínio da ciência ambiental lidando com gerenciamento da zona costeira", INRIA, França, em *http://caravel.inria.fr/Fcontract_THETIS.html*

[TRV98] A. Tomasic, L. Raschid e P. Valduriez, "Scaling Access to Heteregeneous Data Sorces with DISCO", em *IEEE Trans. on Knowledge and Data Engineering*, 10(5), 1998.

[Ullm82] J. D. Ullman, "Principles of Database Systems - 2^a edição", em *Computer Science Press*, Rockville, USA, 1982.

[Vald82] P. Valduriez, "Semi-join Algorithms for Multiprocessor Systems", em *Proc. Intl. Conf. on Management of Data*, Orlando, Flórida, Junho, 1982.

[WG93] R. Wolniewicz e G. Graefe, "Algebraic Optimization of Computations over Scientific Databases", em *Proc. 19^{th} Intl. Conf. on Very Large Data Bases*, Dublin, Irlanda, 1993.

[WC96] J. Widom e S. Ceri, "Active Database Systems: Triggers and Rules for Advanced Database Processing", em *Morgan Kauffman*, 1996.

[Wied92] G. Wiederhold, "Mediators in the Architecture of Future Information Systems", em *IEEE Computer*, 1992.

[YC84] C.T. Yu e C.C. Chang, "Distributed Query Processing", em *ACM Computing Surveys*, Vol.16, No.4, Dezembro, 1984.

Apêndice A

Algoritmos para geração de dados

A.1 algoritmo para geração de valores distintos de atributos, com distribuição aleatória em tuplas

```
parametros de entrada: card, dist1, dist2;
para cada atributo i

   gerar dist(i) valores, numerados de 1 a dist(i)
   para cada valor de dist(i)

      gerar, aleatoriamente, um numero entre 0 e card-1
      alocar dist(i);   // simplificado, sem tratamento de colisoes
   fimpara
   para cada tupla sem valor de i preenchido

gerar, aleatoriamente, um numero entre 1 e dist(i);
   fimpara
fimpara
```

A.2 Algoritmo para geração de indicadores de valores selecionados em predicados

```
parametros de entrada: card, sel1, sel2;
para cada atributo de seletividade si

    limite = sel(si)*100;
    para cada tupla t de R

       gerar um numero n, aleatoriamente, entre 1 e 100;
       se n ja computado
         t.si= n.valor
       senao, se  n <= limite
                  t.si= 1;
               senao
                  t.si=0;
               fimse
        fimse
       fimpara
 fimpara
```

Apêndice B

Validação da abordagem Picking Cherry

B.1 Algoritmo exaustivo de geração de planos de avaliação de valores em predicados caros

principal

    Sejam $A = a_1, a_2, \cdots, a_n eB = b_1, b_2, \cdots, b_n eCP > CQ$;
    $S = \emptyset, S' = A$;
    custoMelhorPlano=∞;
    melhoresPlanos=$\perp$;
    gerar $T = subconjuntosdeBcomC_m^c$ elementos;
    planos=gerarPlanos(S,S',T);
    imprimir melhoresPlanos;

fimprincipal

gerarPlanos(S ,S',T)

    para cada $a_i \in S'$
        U=S;V=S';W=T;
        $U = +a_i$;
        $V = -a_i$;
        para cada $t_i \in T$
            $U = +t_i$;

```
        W = -t_i;
        gerarPlanos(U,V,W);
    fimpara
  fimpara
  Se (S.custo <= custoMelhorPlano
    melhoresPlanos=+S;
    custoMelhorPlano=S.custo;
  fimse
fimgerarPlanos
```