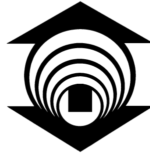


FUNDAÇÃO EDSON QUEIROZ
UNIVERSIDADE DE FORTALEZA – UNIFOR
VICE – REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
MESTRADO EM INFORMÁTICA APLICADA - MIA

MOBIJOIN: UM OPERADOR DE JUNÇÃO PARA BANCOS DE DADOS MÓVEIS

EVELINE VASCONCELOS CAMPOS

FORTALEZA – CE

2005

Livros Grátis

http://www.livrosgratis.com.br

Milhares de livros grátis para download.

EVELINE VASCONCELOS CAMPOS

MOBIJOIN: UM OPERADOR DE JUNÇÃO PARA BANCOS DE DADOS MÓVEIS

Dissertação apresentada ao Curso de Mestrado em Informática Aplicada da Universidade de Fortaleza – UNIFOR como requisito parcial para a obtenção do grau de mestre em Informática.

Orientador: Prof. Ângelo Roncalli Alencar Brayner, Dr-Ing.

Fortaleza – CE

2005

EVELINE VASCONCELOS CAMPOS

MOBIJOIN: UM OPERADOR DE JUNÇÃO PARA BANCOS DE DADOS MÓVEIS

Data de Aprovação: 06/05/2005

Banca Examinadora:

Prof. Dr. Ângelo Roncalli de Alencar Brayner

(Orientador – UNIFOR)

Prof. Dr. Nabor das Chagas Mendonça

(Membro – UNIFOR)

Profª. Dra. Marta Lima de Queiros Mattoso

(Membro – UFRJ)

CAMPOS, Eveline Vasconcelos. **MobiJoin**: um operador de junção para bancos de dados móveis. Fortaleza: UNIFOR, Dissertação (Mestrado em Informática Aplicada), 2005.

xl, 112 p., 210 x 297 mm (MIA/UNIFOR, M.Sc., Ciência da Computação, 2003)

Dedico este trabalho aos meus pais, pessoas de quem muito me orgulho e que são exemplos de vida para mim, e a meu marido, pelo seu amor, companheirismo e paciência.

AGRADECIMENTOS

A Deus e São Judas Tadeu, por me fortalecerem e iluminarem meu caminho.

Ao professor Ângelo Brayner, pela sua orientação firme, dedicação e apoio durante todo o mestrado.

A Marlus Saraiva, responsável pela implementação do protótipo aqui apresentado, por sua colaboração e disponibilidade.

Ao coordenador do mestrado Prof. Plácido, pela compreensão e paciência.

A funcionária do mestrado Tânia, pela sua gentileza e presteza.

A todos aqueles que contribuíram de alguma forma para que este momento se concretizasse.

RESUMO

Uma Comunidade de Bancos de Dados Móveis (MDBC) representa uma coleção de bancos de dados móveis e autônomos. Em tal contexto, cada usuário de um sistema de banco de dados participante da comunidade pode acessar os outros bancos de dados através de uma infra-estrutura de comunicação sem fio. A utilização de técnicas tradicionais para processamento de consultas no acesso a banco de dados em uma MDBC tem se mostrado ineficiente devido à imprevisibilidade na taxa de acesso a bancos de dados da comunidade e a limitação de memória disponível nos dispositivos móveis para a execução de certos operadores, como a junção, por exemplo. A imprevisibilidade é provocada por quedas constantes na comunicação entre os membros de uma MDBC (cenário comum em redes sem fio) e limitação de processamento de certas unidades móveis, atrasando a entrega de tuplas. Para reagir a estes eventos, é apresentado nesta dissertação um operador de junção para processamento de consultas em bancos de dados móveis, denominado MobiJoin. O operador proposto garante as seguintes propriedades: (i) produção incremental de resultados à medida que os dados são disponibilizados; (ii) continuidade no processamento da consulta mesmo que a entrega dos dados esteja bloqueada, e (iii) reação a situações de limitação de memória durante a execução do operador.

Palavras-Chave: Bancos de Dados, Bancos de Dados Móveis, Processamento Adaptativo de Consultas.

ABSTRACT

A Mobile Database Community (MDBC) is a dynamic collection of autonomous mobile databases. In such a context, each database user can access each other's database through a wireless communication infrastructure. Traditional query processing techniques fail to support the access to databases in an MDBC, since data access (or arrival) in such an environment becomes unpredictable and mobiles hosts (in which mobile databases reside) may suffer from limited available main memory to process some query operators (e.g., join). The data access unpredictability arises due to the constants communication disruption, which mat occur frequently in wireless networks, and limited processing capability of some mobile hosts, which may cause a bursty arrival of tuples. To react to those events, it is proposed in this thesis, a join operator for query processing in mobile databases, called MobiJoin. The proposed operator ensures the following properties: (i) incremental production of results as soon as the data become available; (ii) progress of the query processing even when the delivery of data is blocked, and; (iii) reaction to situations of memory limitation on the execution of operator.

Keywords: Databases, Mobile Databases, Adaptative Query Processing.

LISTA DE FIGURAS

Figura 1 - Arquitetura para computação móvel 18
Figura 2 - Configuração de uma MDBC 23
Figura 3 - Arquitetura AMDB 24
Figura 4 - Query Engine da AMDB 26
Figura 5 - Representação gráfica de uma consulta 35
Figura 6 - Adaptative Symmetric Hash Join 41
Figura 7 - Buckets do XJoin 43
Figura 8 - Cenário de execução do MobiJoin 52
Figura 9 - Estado de execução antes da transferência de bucket para disco 55
Figura 10 - Estado de execução após transferência do bucket para disco 56
Figura 11 - Pseudocódigo da primeira fase do algoritmo MobiJoin 57
Figura 12 - Pseudocódigo da segunda fase do algoritmo MobiJoin 58
Figura 13 - Pseudocódigo da terceira fase do algoritmo MobiJoin 60
Figura 14 - Estado de execução antes da chegada de ta5 60
Figura 15 - Estado de execução após a transferência de M_{A1} 62
Figura 16 - Estado de execução até o processamento de tB6 62
Figura 17 - Estado de execução após alocação da tupla t_{B7} 63
Figura 18 - Estado de execução após o processamento da 2a. fase 63
Figura 19 - Estado final da execução da 1a fase 64
Figura 20 - Diagrama de classes do MobiJoin .... **Erro! Indicador não definido.**

LISTA DE GRÁFICOS

Gráfico 1 – Performance da execução do operador com 16 MB....................................74
Gráfico 2 – Performance da execução do operador com 4MB.......................................74
Gráfico 3 - Variações de performance do MobiJoin 75

LISTA DE TABELAS

Tabela 1 – Diferenças do Gerenciamento de Dados na Computação Móvel.................21
Tabela 2 – Comparativo de operadores adaptativos..50
Tabela 3 – Comparativo de operadores adaptativos e do MobiJoin...............................67

ÍNDICE

1. INTRODUÇÃO 14
1.1 Motivação 14
1.2 Objetivo e Escopo 15
1.3 Estrutura 16
2. BANCOS DE DADOS MÓVEIS 17
2.1 Introdução 17
2.2 Mobilidade 17
2.3 Mobilidade e Bancos de Dados 19
2.4 Arquitetura AMDB 22
2.4.1 Processamento de Consultas na Arquitetura AMDB 25
2.4.2 Características Adaptativas 27
2.4.3 Operadores Físicos Adaptativos 28
3. OPERADORES DE JUNÇÃO ADAPTATIVOS 29
3.1 Introdução 29
3.2 Processamento Adaptativo de Consultas 30
3.3 Técnica de *Pipelining* 34
3.4 Operadores de Junção Adaptativos 36
3.4.1 Double Pipelined Hash Join 36
3.4.2 Adaptative Symmetric Hash Join 39
3.4.3 Xjoin 42
3.5 Análise Comparativa 46
3.5.1 Critérios de Comparação 46
3.5.2 Análise dos operadores de junção apresentados com relação aos critérios identificados 48
4. MOBIJOIN 51
4.1 Introdução 51
4.2 Cenário de Execução do Operador 52
4.3 Funcionamento do Operador 53
4.4 Exemplo de Execução do Operador 60

4.5 Estimativa de Custo 65
4.6 Análise Comparativa 66
5. IMPLEMENTAÇÃO DO MOBIJOIN 70
CONCLUSÃO 77
REFERÊNCIAS 80
APÊNDICE A – DIAGRAMA DE CLASSES DO MOBIJOIN 86
APÊNDICE B – CÓDIGO FONTE DA IMPLEMENTAÇÃO DO MOBIJOIN 87

1. INTRODUÇÃO

1.1 Motivação

Com o desenvolvimento das tecnologias de sistemas de comunicação sem fio e de computadores portáteis, a computação móvel tornou-se realidade no ambiente computacional moderno. A integração destas duas tecnologias possibilita que computadores portáteis (*laptops*, *notebooks*, *palms*, etc...) conectem-se, como componentes, a um ambiente distribuído e disponibilizem seus recursos computacionais, mesmo que mudem constantemente sua localização geográfica. Portanto, modelos, arquiteturas e tecnologias para computação móvel se fazem necessários. Novos desafios estão surgindo devido à complexidade gerada pela mobilidade. As características da comunicação sem fio, como baixa largura de banda e desconexão freqüente, favorecem um estilo de computação que deve prosseguir mesmo na presença de desconexão e deve também explorar a conectividade quando a mesma estiver disponível. Os componentes de um ambiente móvel devem estar continuamente habilitados para o acesso aos recursos recentemente disponíveis e para interagir com outros componentes, móveis ou não, que estejam conectados [14, 23, 31, 32].

Atualmente, o compartilhamento de informações entre múltiplas fontes de dados heterogêneas, autônomas, distribuídas e móveis tem emergido como um requerimento estratégico que deve ser suportado pela tecnologia de bancos de dados. Usuários que carregam equipamentos portáteis podem acessar ou disponibilizar serviços de bancos de dados independentemente da sua localização física ou padrão de movimentação. De acordo com este novo cenário, comunidades de bancos de dados móveis podem ser formadas. Brayner e Aguiar [13] definem uma Comunidade de Bancos de Dados Móveis (MDBC) como uma coleção dinâmica de bancos de dados móveis, distribuídos e autônomos, na qual cada usuário de

banco de dados pode acessar cada um dos outros bancos de dados através de uma infra-estrutura de comunicação sem fio.

Considerando comunidades de bancos de dados autônomos, heterogêneos e móveis que são formadas dinamicamente e que estão conectadas através de uma infra-estrutura de comunicação sem fio, Brayner e Aguiar [11] propuseram uma arquitetura, chamada AMDB, que provê um mecanismo de processamento de consultas baseado no conceito de agentes, suportando a formação oportuna de MDBCs e habilitando o compartilhamento de dados entre os membros das MDBCs.

1.2 Objetivo e Escopo

Com o objetivo de minimizar o impacto da mobilidade no processamento das consultas aos bancos de dados móveis participantes em uma MDBC na AMDB, e prover uma adaptação dinâmica do plano de execução das consultas em resposta a problemas relacionados à mobilidade, apresentamos nesta dissertação um operador de junção para processamento de consultas em bancos de dados móveis chamado MobiJoin. A idéia é que o MobiJoin comporte-se como um operador de consulta adaptativo, reagindo a eventos, como o atraso na entrega de tuplas, comuns em Comunidades de Bancos de Dados Móveis, e que podem aumentar substancialmente o tempo de resposta no acesso a bancos de dados móveis.

O MobiJoin está baseado em três princípios de funcionamento: *(i)* produção incremental de resultados, à medida que os dados são disponibilizados, *(ii)* continuidade no processamento da consulta, mesmo que a entrega dos dados esteja bloqueada, e *(iii)* reação a situações de limitação de memória durante a execução do operador.

O MobiJoin é baseado nos operadores *Symmetric Hash Join* (SHJ) [40]. Conforme originalmente proposto, o SHJ requer que as tabelas *hash* das duas relações de entrada da junção sejam mantidas em memória durante a execução da consulta. Em conseqüência disso, o SHJ não pode ser utilizado para executar a junção entre relações de entrada quando há limitação de memória. O MobiJoin

estende o SHJ, adicionando um mecanismo de transferência dos dados que estão na memória para o disco quando necessário. Porém, esta extensão não é suficiente para mascarar tempos de espera significantes no recebimento dos dados das fontes remotas. Para resolver este problema, foi desenvolvida uma estratégia que oportunamente utiliza os tempos de espera para produzir resultados mais cedo. Com esta propriedade, o MobiJoin reduz a taxa de transferências de *buckets* em memória para o disco, além de permitir uma melhor utilização da memória disponível.

Para demonstrar a viabilidade do operador MobiJoin, foi implementado um protótipo que permite a execução do operador em um ambiente simulado de computação móvel, onde estão previstas situações como limitação de memória e bloqueio e atraso no recebimento dos dados. As simulações servem com base para uma análise mais profunda do nível de adaptabilidade deste operador e futuramente também serão utilizadas para testes comparativos com outros operadores propostos.

1.3 Estrutura

Esta dissertação está organizada como descrito a seguir. No capítulo 2 apresentaremos uma visão geral sobre mobilidade e seu impacto na área de Banco de Dados, e descreveremos a arquitetura AMDB. O capítulo 3 provê uma visão geral de processamento adaptativo e apresenta algumas propostas de operadores de junção adaptativos para processamento de consultas em ambientes distribuídos, além de uma análise comparativa destas propostas. No capítulo 4, descreveremos o operador de junção proposto, seu cenário e fases de execução, além de uma discussão dos trabalhos relacionados. A implementação do protótipo do MobiJoin e o resultado dos testes efetuados é descrito no capítulo 5. Finalmente apresenta-se a conclusão e aponta-se trabalhos futuros.

2. BANCOS DE DADOS MÓVEIS

2.1 Introdução

Este capítulo apresenta uma visão geral sobre mobilidade e o seu impacto na área de Bancos de Dados, introduzindo o conceito de Comunidades de Bancos de Dados Móveis e apresentando a arquitetura AMDB, que provê processamento de consultas em ambientes de computação móvel.

Este capítulo possui a seguinte estrutura: a seção 2 descreve as características de um ambiente de computação móvel e introduz o conceito de Comunidades de Bancos de Dados Móveis (MDBC); a seção 3 descreve o impacto da mobilidade na área de bancos de dados em vários aspectos e principalmente no processamento de consultas e a seção 4 descreve a arquitetura AMDB (*Accessing Mobile Databases*), a funcionalidade de seus componentes e seu suporte a processamento adaptativo de consultas.

2.2 Mobilidade

O modelo de um sistema de computação móvel consiste em componentes fixos e móveis, conforme descrito em [19] e como mostra a Figura 1. O componente móvel é chamado de unidade móvel. Uma unidade móvel é um computador capaz de se conectar em uma rede fixa através de um link de comunicação sem fio. *Hosts* fixos se mantêm conectados através de uma rede fixa. Os componentes desta rede são classificados como *hosts* fixos ou estações de base. Um *host* fixo é um computador fixo na rede fixa que não é capaz de se conectar com uma unidade móvel. Uma estação de base é capaz de se conectar com uma unidade móvel e possui um dispositivo de comunicação sem fio. Elas também são conhecidas como estações de suporte a mobilidade (ESM) ou estações base (EB). As estações base atuam como a interface entre os computadores móveis e as redes fixas, dando

suporte às células. A célula determina a área de abrangência da estação de base e é constituída por um conjunto de computadores móveis.

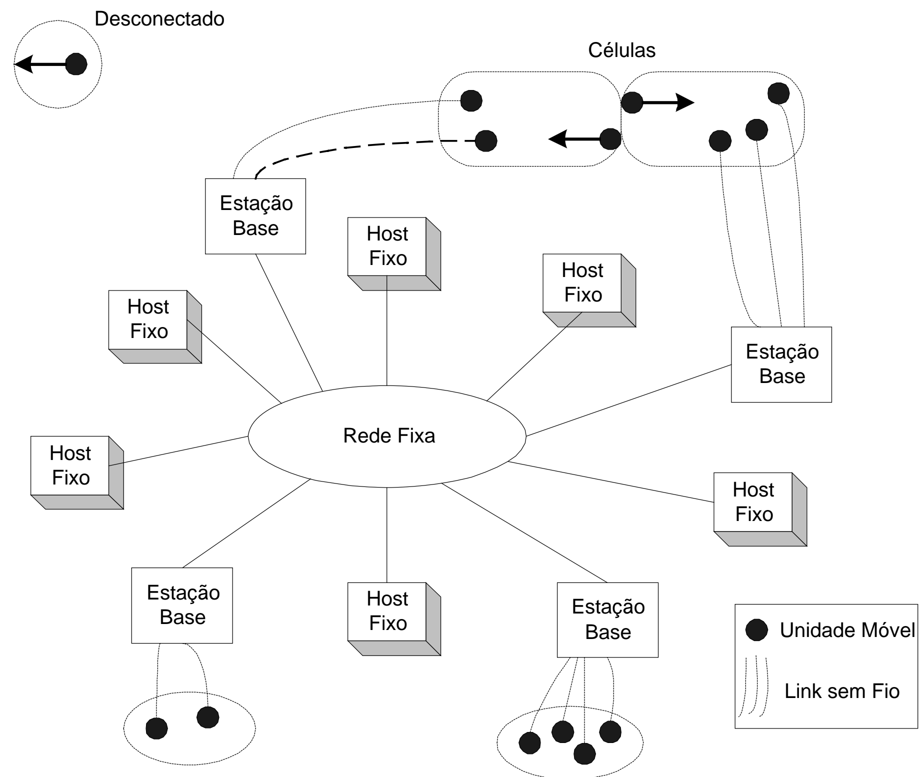


Figura 1 - Arquitetura para computação móvel

Segundo Murphy, Picco e Roman [36], a computação móvel abrange dois cenários. No primeiro, como mostrado na Figura 1, a disponibilidade da rede fixa é assumida e suas facilidades são exploradas pela infra-estrutura móvel. O segundo cenário surgiu com os avanços de desempenho dos computadores portáteis e da capacidade de comunicação sem fio, que possibilitou que alguns tipos de dispositivos móveis pudessem separar-se completamente de uma rede fixa, formando estruturas dinâmicas que podem se desenvolver independentemente de uma infra-estrutura física de comunicação. Estas estruturas dinâmicas são chamadas "redes *ad hoc*" e são formadas exclusivamente por *hosts* móveis que podem conectar-se entre si desde que estejam dentro de uma área de abrangência de uma rede de comunicação sem fio. As redes *ad hoc* favorecem um estilo de

computação de oportunidades, que explora a conectividade enquanto a mesma estiver disponível, demandando, dessa forma, uma nova prática no desenvolvimento de aplicações distribuídas. Em contraste aos ambientes tradicionais de computação móvel, onde a localização dos provedores de informação assim como as conexões de rede são relativamente estáveis, em redes *ad hoc* a migração dos dispositivos e as falhas de conexão são a norma [11, 32].

2.3 Mobilidade e Bancos de Dados

Na área de sistemas de banco de dados, a mobilidade oferece acesso à informação em qualquer lugar a qualquer hora. O paradigma de computação móvel afeta a área de banco de dados uma vez que conceitos e mecanismos aplicados para banco de dados tradicionais precisam ser revistos.

Imielinski e Badrinath [30] agrupam os maiores desafios trazidos pela computação móvel no gerenciamento de dados nas seguintes categorias:

1. Mobilidade, que apresenta os desafios de localização de usuários móveis e distribuição de informação para usuários;
2. Desconexão, distinguindo os seus vários graus (*weak* e *strong*) e tratando-a diferentemente de falhas que ocorrem nos ambientes de sistemas distribuídos tradicionais;
3. Novos modos de acesso aos dados, uma vez que é preciso considerar a infra-estrutura de comunicação e a autonomia dos computadores portáteis;
4. Escala, considerando o tamanho do conjunto de usuários potenciais, a distribuição dos serviços, a organização de mediadores e o particionamento de conhecimento entre os diferentes componentes do sistema.

Dois tipos de gerenciamentos de dados em ambientes de computação móvel são também identificados por Imielinski e Badrinath [30]: gerenciamento de dados globais, que soluciona problemas em nível de rede, incluindo localização, endereçamento, replicação e *broadcasting*, e gerenciamento de dados locais, que soluciona problemas relacionados com o usuário final, como gerenciamento dos graus de desconexão e processamento de consultas.

Dunham e Helal, em [19] identificaram que apesar dos sistemas móveis possuírem muitos dos problemas que os sistemas distribuídos possuem, existem algumas diferenças, descritas na tabela 1, e por isto muitas soluções para problemas de bancos de dados distribuídos não são soluções aceitáveis para o ambiente de computação móvel.

Dunham e Ren [21] estudam o gerenciamento de dados de objetos móveis e a informação de localização dos mesmos. Definem dado dependente de localização como o dado cujo valor é determinado pela localização com a qual ele está relacionado. Exemplos deste tipo de dados são: páginas amarelas, relatórios de tráfego, informações climáticas e outros. Consultas dependentes de localização são consultas processadas sobre dados dependentes de localização e cujo resultado depende do critério de localização explicitamente ou implicitamente especificado. Os resultados destas consultas podem ser alterados à medida que os usuários alteram as suas localizações.

Com relação ao banco de dados, segundo Brayner e M. Filho [11], a mobilidade pode ser caracterizada em dois tipos:

i) mobilidade física: deslocamento físico de servidores e clientes de bancos de dados entre diferentes regiões;

ii) mobilidade lógica: migração de código entre vários clientes móveis e servidores de bancos de dados. Com o objetivo de prover mobilidade lógica, computadores móveis devem estar aptos para gerar códigos de acesso a bancos de dados (consultas SQL, *stored procedures* ou métodos), que podem migrar de forma autônoma para vários servidores de bancos de dados. Quando este código chega em um servidor de banco de dados, ele deve ser localmente executado, e pode retornar à origem com o resultado originado pelo código de acesso carregado por ele.

Alonso e Korth [3] apontam o impacto da computação móvel em vários aspectos dos sistemas de bancos de dados convencionais. O processamento de consultas deve considerar o modelo do ambiente de rede da computação móvel, o custo de comunicação e a autonomia dos equipamentos portáteis.

Tabela 1 – Diferenças do Gerenciamento de Dados na Computação Móvel

Aspectos de Gerenciamento de Dados	Diferenças na Computação Móvel
Aplicações	• Podem ser dependentes de localização • Necessidade de adaptação às alterações do contexto do sistema
Transações	• Novos modelos que capturem a mobilidade
Recuperação	• Particionamento freqüente da rede • Desconexão voluntária não é uma falha do sistema • Técnicas para recuperação de desconexão
Replicação	• Restrições de consistência diferentes • Novas técnicas para atualização do cache das unidades móveis devido às desconexões freqüentes
Processamento de Consultas	• Podem ser dependentes de localização • Diferentes fatores de custo • Resultado de consulta retornado para diferentes localizações • Técnicas adaptativas necessárias
Resolução de Nomes	• Nova estratégia de nomeação global devido à mobilidade e às desconexões

Ainda com relação ao processamento de consultas em ambientes de computação móvel, a utilização de técnicas tradicionais de processamento de consultas tem se mostrado ineficiente por várias razões, entre elas, as mais críticas são:

i) Ausência de estatísticas globais;
ii) Imprevisibilidade do tempo de resposta dos bancos de dados envolvidos em uma consulta;
iii) Limitação de memória disponível nos dispositivos móveis;
iv) Utilização de operadores que possuem mais de uma fase de execução e que não apresentam suporte para a técnica de *pipelining*[1].

[1] Operadores de consulta que suportam a técnica de *pipelining* geram tuplas resultado tão logo tuplas de entrada do operador são recebidas (lidas).

Segundo Alonso e Korth [3], o controle de transações deve estar preparado para as desconexões dos dispositivos móveis, que são comuns quando equipamentos portáteis são utilizados. Diferentemente das transações em ambientes distribuídos, transações móveis podem não ter início e fim no mesmo *site*. O movimento das transações neste tipo de ambiente faz com que os conceitos tradicionais sejam revistos de forma que consigam capturar o comportamento do movimento. O modelo de transações para ambientes de computação móvel deve suportar concorrência, atomicidade e recuperação, além de gerenciar as freqüentes desconexões que podem ocorrer e manter a consistência mútua entre os dados replicados [18]. Para uma leitura em profundidade sobre processamento de transações móveis, recomenda-se a leitura de [11,12,13].

2.4 Arquitetura AMDB

Atualmente, o compartilhamento de informações entre múltiplas fontes de dados heterogêneas, autônomas, distribuídas e móveis tem emergido como um requerimento estratégico que deve ser suportado pela tecnologia de bancos de dados. Usuários que carregam equipamentos portáteis estão habilitados para acessar serviços de bancos de dados independentemente da sua localização física ou padrão de movimentação. De acordo com este novo cenário, comunidades de bancos de dados móveis podem ser formadas.

Uma Comunidade de Bancos de Dados Móveis (MDBC) é uma coleção dinâmica de banco de dados móveis, distribuídos e autônomos, na qual cada usuário de banco de dados pode acessar cada um dos outros bancos de dados através de uma infra-estrutura de comunicação sem fio (rede móvel *ad hoc*). A Figura 2 ilustra uma possível configuração de uma MDBC. Em um determinado momento, um variado número de computadores móveis pode está participando de uma MDBC. Neste caso, os membros da MDBC compartilham os bancos de dados que desejam com a comunidade. Desta forma, existem dois tipos de membros em uma MDBC: servidores de bancos de dados e clientes de bancos de dados. Os bancos de dados armazenados nos servidores de bancos de dados podem ser

acessados pelos clientes de bancos de dados, que não têm ou não desejam compartilhar seus bancos de dados entre os membros da comunidade.

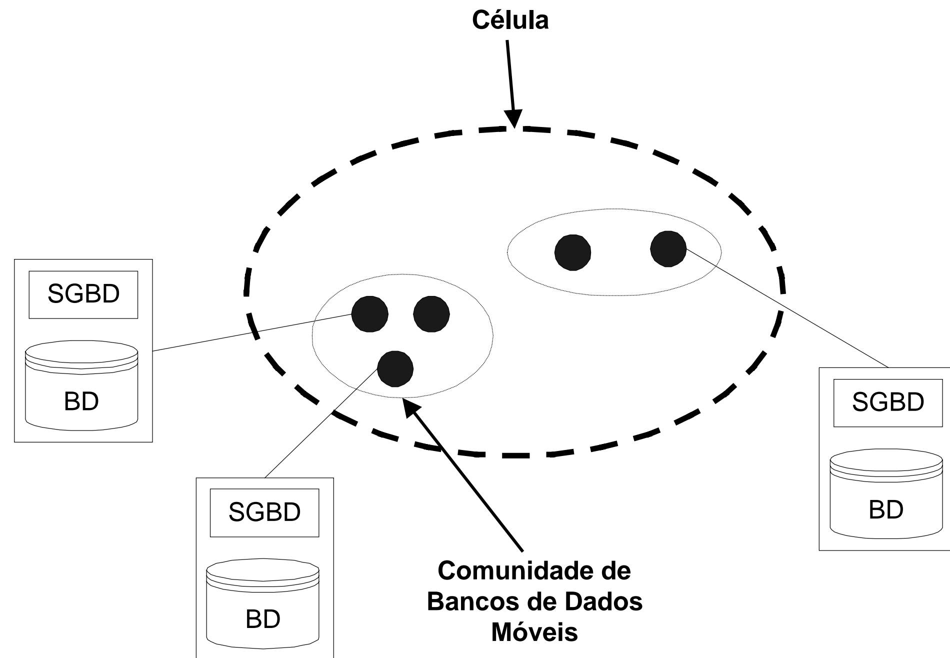


Figura 2 - Configuração de uma MDBC

As técnicas tradicionais para processamento de consultas, processamento de transações e distribuição de dados devem ser revistas para suportar acesso a bancos de dados móveis. Considerando comunidades de bancos de dados autônomos, heterogêneos e móveis que são formadas dinamicamente e que estão conectados através de uma infra-estrutura de comunicação sem fio, Brayner e M. Filho [11] propuseram uma arquitetura, chamada AMDB (*Accessing Mobile Databases*), que provê um mecanismo de processamento de consultas baseado no conceito de agentes, suportando a formação oportunística de MDBCs e habilitando o compartilhamento de dados entre os membros das MDBCs.

Na AMDB, como mostra a Figura 3, existem dois tipos de agentes: fixos e móveis. Os agentes fixos podem ser Administradores, responsáveis pelos recursos computacionais locais do computador móvel, ou Mantenedores, que provêm uma interface entre os usuários das unidades móveis e a plataforma AMDB e uma representação de dados comum para dados armazenados nos computadores

móveis. Agentes móveis são habilitados para transportar-se de um computador móvel para outro, levando código de acesso ao banco de dados e os resultados das operações efetuadas através deste código, enquanto migram entre os vários membros da MDBC. Existem três tipos de agentes móveis: Executor, Carregador e Promotor. Os agentes do tipo Executor são responsáveis pela execução de tarefas requeridas pelos usuários das unidades móveis em bancos de dados remotos pertencentes à MDBC. A função dos agentes do tipo Carregador é levar o resultado da consulta de banco de dados para a unidade móvel que requereu a consulta. Agentes do tipo Promotor obtêm os esquemas dos bancos de dados móveis da MDBC quando uma unidade móvel se une à comunidade, e definem uma unidade móvel como unidade de armazenamento temporário para armazenar resultados parciais de consultas quando necessário.

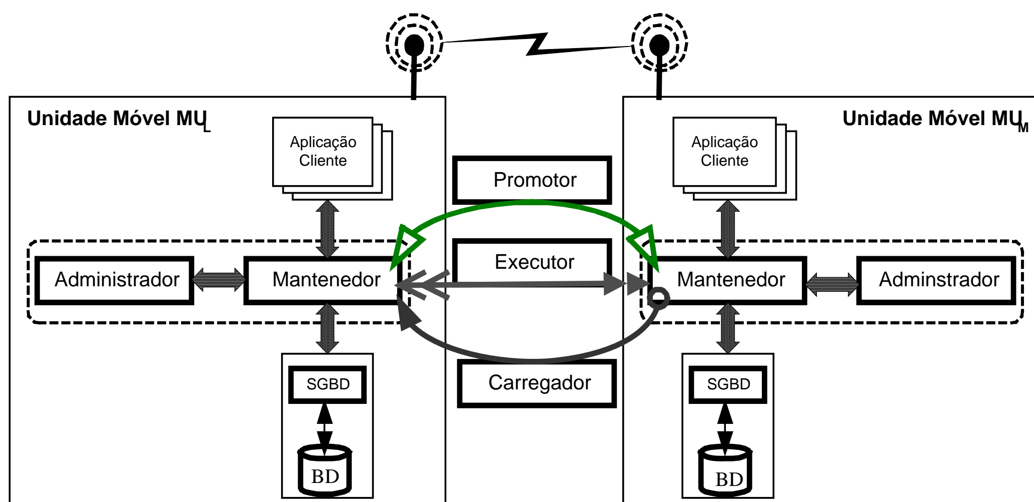


Figura 3 - Arquitetura AMDB

Para formar uma MDBC é necessário que a primeira unidade móvel, que deseja formar a comunidade, declare-se coordenadora inicial para o seu agente Mantenedor, que enviará a informação sobre a nova comunidade para todos os computadores da área coberta pela rede de comunicação sem fio na qual esta unidade móvel está conectada. Após a chegada de novos membros à comunidade o papel de coordenador não é mais necessário, sendo executada local e

colaborativamente pelos agentes Mantenedores em cada unidade móvel que é membro da comunidade.

Para que uma unidade móvel visualize os esquemas de bancos de dados disponibilizados pelos outros membros da comunidade, o agente Mantenedor local cria um agente Promotor, que percorre todas as unidades móveis e coleta todos os esquemas disponíveis, que são providos pelo agente Mantenedor local de cada unidade móvel. Após a coleta o agente Promotor entrega os esquemas, no formato XML Schema, para o agente Mantenedor da unidade móvel que o criou, que provê uma interface que habilita os usuários à submissão de consultas.

Usuários ou aplicações submetem consultas na arquitetura AMBD no formato *MXQuery*. Estas consultas podem ser locais, globais (distribuídas) ou remotas. As consultas locais ou remotas acessam um único banco de dados que podem estar localizados local ou remotamente, respectivamente. Consultas globais envolvem operações em vários bancos de dados móveis da comunidade. Definimos Coordenador de uma consulta global a unidade móvel que requereu a consulta e como Participante uma unidade móvel da comunidade que é responsável pela execução de operações locais de uma consulta global ou remota.

2.4.1 Processamento de Consultas na Arquitetura AMDB

Uma consulta é processada por seis módulos distintos: Pré-Processamento, Decomposição, Reconhecimento de Contexto, Geração de Clones, Execução de Fragmentos e Pós-Processamento, mostrados na Figura 4. O processamento é totalmente distribuído, considerando que as consultas são processadas por diferentes agentes em diferentes unidades móveis. O processo de otimização utiliza metadados sobre os esquemas locais, o número de *hosts* nos quais a consulta será executada e informações estatísticas disponíveis sobre os dados locais.

As atividades do módulo Pré-processamento são executadas pelos agentes Mantenedor e Executor. Após receber a consulta *MXQuery*, o Mantenedor faz a avaliação sintática e semântica da consulta e gera o seu plano de execução global (PEG), que é representado por uma árvore de operadores, onde cada operador é anotado com regras que executam ações quando eventos específicos ocorrem, com

o objetivo de adaptar o PEG inicial ao novo contexto de execução. Estatísticas incorretas, reformulação pelo coordenador e fonte de dados não disponível são alguns dos eventos que podem ser especificados. As ações correspondentes podem ser: reformulação do plano e rota, execução de novo fragmento e reformulação da rota.

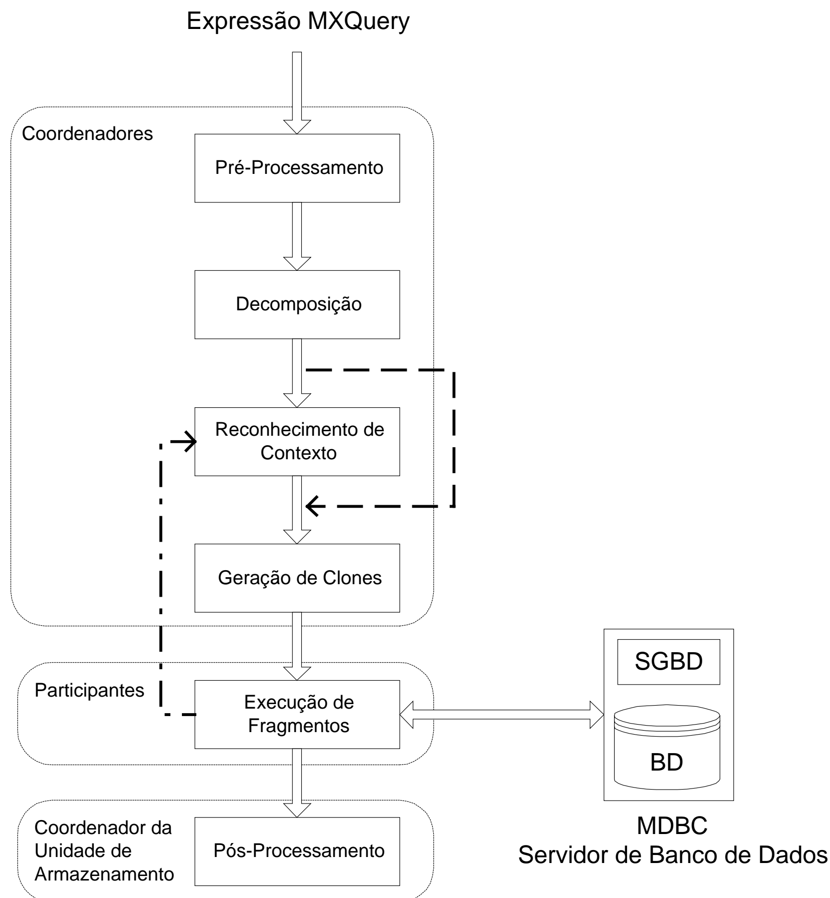


Figura 4 - Query Engine da AMDB

O módulo Decomposição tem como objetivo a decomposição da consulta em fragmentos, conforme o princípio da localidade. Um fragmento de consulta representa uma sub-árvore do PEG gerado pelo módulo Pré-processamento.

Esta rota é definida através de uma heurística baseada na cardinalidade dos resultados intermediários de cada unidade móvel, onde as cardinalidades mais baixas devem estar no início da rota. A execução paralela consiste em execução paralela e assíncrona de fragmentos de consultas, e neste caso, o módulo Geração de Clones deve ser ativado. A decisão do tipo de execução mais adequado é

baseada em uma outra heurística, que determina o custo de execução dos dois tipos de execução escolhendo o de mais baixo custo.

Considerando que a execução será paralela, o módulo Geração de Clones iniciará sua execução com o agente Executor criando um clone no *host* de origem para cada fragmento de consulta, e enviando-os para as unidades móveis correspondentes para que os fragmentos sejam executados.

O módulo Execução de Fragmentos é executado quando um agente Executor ou seu clone migra para uma unidade móvel na qual o fragmento de consulta deve ser executado. Na chegada do fragmento, o agente Executor submete o fragmento *MXQuery* ao agente Mantenedor local que o traduz para a linguagem nativa do sistema gerenciador de banco de dados local, submete-o para execução e retorna o resultado para o agente Executor. Durante a execução do fragmento o agente Executor também coleta estatísticas que são entregues na unidade móvel de origem ao agente Mantenedor para atualização do esquema correspondente local. Regras do PEG também podem ser executadas no módulo Execução de Fragmentos, durante a execução de um fragmento e na presença de alguns eventos como: fonte de dados não disponível e número de tuplas esperado diferente do número de tuplas recuperado. A ocorrência destes eventos dispara uma ação reativa, cuja execução é de responsabilidade do agente Executor e que pode modificar o fragmento local ou outros fragmentos, ou mesmo o plano inteiro.

O módulo Pós-processamento é iniciado quando o agente Executor, ou seu clone, chega na unidade de origem ou na unidade de armazenamento temporário e inicia a execução das operações que não podem ser executadas por nenhum sistema gerenciador de banco de dados local. O agente Executor utiliza operadores adaptativos para execução das operações globais que, após finalizadas, correspondem ao resultado final da consulta e são enviadas para o usuário. Informações estatísticas também são acumuladas durante este módulo.

2.4.2 Características Adaptativas

Em ambientes de configuração dinâmica com infra-estrutura de comunicação sem fio, como uma MDBC, desconexões dos participantes enquanto consultas são

processadas podem ocorrer freqüentemente. Técnicas de processamento adaptativo de consultas [1,6,11,22,26,40] são adequadas para estes tipos de ambiente e podem ser aplicadas para que mesmo com a ocorrência de determinados eventos, o processamento da consulta continue ocorrendo de forma útil. A arquitetura AMDB integra operadores adaptativos e *triggers* para atender os requesitos do usuário e as características de uma MDBC, como freqüentes desconexões, baixa largura de banda e baixa capacidade de processamento dos participantes.

2.4.3 Operadores Físicos Adaptativos

Devido à autonomia das unidades móveis pertencentes a uma MDBC, algumas operações do plano de execução global da consulta podem ser executadas pelo agente Executor, quando um sistema gerenciador de banco de dados pertencente a MDBC não tem capacidade de processar uma determinada operação. Como estas operações podem requerer dados de unidades móveis distintas que podem ser canalizadas no plano de execução global da consulta, operadores adaptativos são utilizados na execução das mesmas.

Quando o usuário submete uma consulta, o agente Executor tenta executar as operações globais o mais cedo possível, conforme a estratégia descrita em seguida. Para cada nó do plano de execução global da consulta, existe um valor indicando o número de tuplas esperadas que será entregue por unidade de tempo. Baseado neste valor, o agente Executor pode escolher o operador físico para executar a operação global. Uma vez que o agente Executor pode ser *multi-thread*, ele pode continuamente monitorar eventos, e reagir à ocorrência destes eventos alterando o operador escolhido.

3. OPERADORES DE JUNÇÃO ADAPTATIVOS

3.1 Introdução

Com o objetivo de suportar o acesso à banco de dados móveis participantes em uma MDBC utilizando a arquitetura AMDB, as técnicas tradicionais para processamento de consultas precisam ser revistas, pois, neste tipo de ambiente, o uso de tais técnicas tem se mostrado ineficiente por várias razões, entre elas, as mais críticas são as seguintes:

- Ausência de estatísticas globais. Considerando o número variado de bancos de dados móveis autônomos que entram na comunidade há pouca ou nenhuma estatística sobre os mesmos;
- Imprevisibilidade do tempo de resposta dos bancos de dados da comunidade. Há pouco conhecimento a respeito das taxas de entrega de dados dos bancos de dados envolvidos em uma consulta. Desta forma, mesmo que o otimizador de consulta tenha escolhido o melhor plano em um determinado instante, baixas taxas de entrega dos dados (causadas por desconexões constantes na rede sem fio, comuns neste tipo de cenário) no momento da execução da consulta podem tornar o plano ineficiente. Para contornar tal problema é necessária a utilização de operadores que se adaptem ao novo cenário de execução da consulta, para reagir a latências não previstas;
- Limitação de memória disponível nos dispositivos móveis para execução de certos operadores de junção;
- Utilização de operadores que possuem mais de uma fase de execução e que não apresentam suporte para a técnica de *pipelining*. Por exemplo, o operador *hash join* possui duas fases de execução (construção e comparação). A fase de comparação só pode ser executada, após a execução da fase de construção por completo (ou seja, para todas as tuplas das duas relações da operação de

junção). O desempenho deste operador é afetado caso ocorra atraso na entrega dos dados, fazendo com que a primeira fase forneça seu resultado para a fase de comparação a taxas muito baixas de entrega de tuplas.

Neste capítulo descrevemos processamento adaptativo de consultas e a técnica de *pipelining*. Além disso, discutimos pesquisas com foco em operadores de junção, que apresentam a propriedade de adaptabilidade durante a execução de consultas em ambientes distribuídos. Estas propostas são interessantes porque algumas técnicas utilizadas também podem ser aplicadas no contexto das consultas em ambientes móveis, e conforme [33], estes ambientes têm algumas características comuns, como:

- Adaptação às capacidades de processamento das distintas fontes de dados;
- Otimização das consultas durante a execução de acordo com o contexto das fontes de dados acessadas;
- Obtenção do menor tempo de resposta na chegada da primeira tupla de resposta na execução da consulta, mesmo que seja necessário um aumento do tempo total de execução da consulta;
- Provimento de resultados parciais, os dados são disponibilizados para o usuário à medida que são processados.

Este capítulo possui a estrutura em seguida descrita. A seção 2 introduz o conceito de processamento adaptativo de consulta e a seção 3 descreve a técnica de *pipelining*. Operadores de junção adaptativos propostos para ambientes distribuídos são apresentados na seção 4. A seção 5 apresenta uma análise comparativa entre os operadores descritos na seção 4.

3.2 Processamento Adaptativo de Consultas

Nos últimos anos, muitas áreas da Ciência da Computação têm explorado a arquitetura de sistemas que se adaptam a seus ambientes de execução. Na área de banco de dados, as técnicas tradicionais de processamento de consultas estenderam-se para o processamento adaptativo de consultas, uma vez que muitas

máquinas de consultas operam em ambientes imprevisíveis e em constante alteração de configuração. Em sistemas de ambientes *wide-area*, como a Internet, variações de performance são normalmente observadas nos servidores e na rede [4]. Estes sistemas normalmente atendem a uma grande comunidade de usuários, cujo comportamento é de difícil previsão e cujo hardware envolvido é muito heterogêneo. Os dados envolvidos nas consultas não possuem estatísticas atualizadas e as estimativas de seletividade são quase sempre incorretas ou imprecisas.

Hellerstein et al [28] definem que um sistema de processamento de consultas é adaptativo se:

i. Receber informações do ambiente de execução (no momento da execução da consulta), informações estas relativas às relações envolvidas no processamento da consulta (cardinalidade, estimativa de seletividade, etc.);
ii. Usar as informações recebidas para determinar o seu comportamento e;
iii. Garantir um processo de interação ao longo do tempo, gerando um ciclo de respostas entre o ambiente e o comportamento do sistema de processamento de consultas.

Esta interação é a chave para que o sistema tome múltiplas decisões e observe os resultados das mesmas em tempo de execução. Os sistemas de processamento de consultas tradicionais possuem as duas primeiras características, mas não a terceira. Já os sistemas de processamento de consultas adaptativos utilizam estratégias dinâmicas, de acordo com as informações recebidas e, caso necessário, para prover adaptação do plano de execução das consultas às características do ambiente de execução. Desta forma, o plano de execução da consulta pode ser alterado, determinando um novo comportamento para o sistema e objetivando a diminuição do impacto de fatores relacionados ao ambiente de execução no processamento da consulta.

A necessidade de adaptatividade cresce conforme o aumento da imprevisibilidade associada ao ambiente de execução das consultas. Muitos projetos de pesquisa focados em ambientes, cujo comportamento é de difícil previsão, têm explorado a adaptatividade do plano de execução da consulta com relação à fase de

otimização, na escolha dos métodos de acesso aos dados, dos operadores de junção e sua ordem de execução. Uma prática utilizada em alguns sistemas distribuídos é enviar subconsultas para sites remotos e, então, utilizar a chegada dos resultados das subconsultas para guiar o escalonamento da execução das partes remanescentes da consulta.

Bouganim et al. [7] propõem uma estratégia dinâmica de escalonamento na execução de operadores em um plano de execução de consultas para resolver problemas de performance devido à imprevisibilidade da taxa de chegada dos dados em ambientes de integração de dados. Esta estratégia supõe que atrasos acontecerão durante a execução e inclui este fator na sua estratégia de execução. Para fazer isto, monitora constantemente a taxa de chegada dos dados de cada fonte de dados participante e a memória disponível. Este conhecimento é usado na elaboração de planos de execução que são revisados sempre que as taxas de chegadas de dados alteram significantemente. Desta forma, intercala fases de planejamento, onde planeja o futuro próximo, e fases de execução, onde reage instantaneamente aos atrasos.

Contudo, a adaptatividade também pode ser inserida dentro dos operadores de consulta, como por exemplo, a junção. Desta forma, os operadores passam a adaptar-se ao contexto de execução, mesmo fazendo parte de um plano de execução fixo.

Mais especificamente, podemos afirmar que algoritmos de operadores de junção tradicionais têm características indesejáveis quando executados para processamento de consultas sobre MDBCs. Como exemplos para este fato, podemos citar os operadores *Sort-Merge Join* e *Hash Join*. No caso do *Sort-Merge Join,* com exceção quando as relações envolvidas já possuem dados classificados, as tuplas já recebidas não podem ser imediatamente comparadas e canalizadas para as operações superiores do plano de execução global da consulta, pois requerem um passo inicial de ordenação. Desta forma, a comparação das tuplas apenas é iniciada após a chegada de todas as tuplas envolvidas no processamento da consulta. Caso ocorra atraso na entrega das tuplas, o que é comum em ambientes de computação móvel, a performance do operador será comprometida, tornando assim o plano de execução da consulta ineficiente.

No caso do operador de junção *Hash Join*, este possui duas fases de execução: construção e comparação. A fase de comparação só pode ser executada, após a execução da fase de construção por completo (ou seja, para todas as tuplas das duas relações da operação de junção). Esta característica pode degradar a performance na execução do operador (por exemplo, se as duas relações apresentam grandes cardinalidades). A performance deste operador também é afetada caso ocorra atraso na entrega dos dados, fazendo com que a primeira fase forneça seu resultado para a fase de comparação a taxas muito baixas de entrega de tuplas.

Ainda com relação ao *Hash Join*, pode-se observar que este operador apresenta importantes parâmetros que podem ser configurados pelo otimizador, baseado no conhecimento das cardinalidades das relações envolvidas. A decisão mais importante é a respeito da definição da relação de entrada, que deve ser a menor das duas relações, para que caiba totalmente na memória. Outros parâmetros são o número de *buckets* usado, o número de *buckets* escrito em disco e a quantidade de memória alocada para o operador.

Em sistemas de bancos de dados tradicionais, onde o otimizador tem conhecimento das cardinalidades e o custo de leitura e escrita em disco para qualquer fonte de dados é o mesmo, os parâmetros do *Hash Join* podem ser configurados com eficácia. Porém, em ambientes de computação móvel encontramos vários desafios como o não conhecimento pelo otimizador do tamanho das relações envolvidas. Como deseja-se um plano de execução da consulta que obtenha o menor tempo de resposta para a consulta e não o de menor custo, é possível que em determinado momento seja mais indicado eleger a maior relação como relação de entrada, se a fonte de dados de origem da maior relação apresentar uma taxa de entrega de tuplas mais alta.

O sistema *Tukwilla* [42] é um bom exemplo para ilustrar a aplicabilidade do conceito de processamento adaptativo de consultas. O *Tukwilla* é um sistema de integração de dados, que utiliza operadores adaptativos com o objetivo de minimizar o tempo requerido para obter os primeiros resultados da consulta. Especificamente, este sistema implementa uma extensão do operador de junção *Double Pipeline*

Hash Join [42] com técnicas para adaptação de sua execução quando a memória for insuficiente. Este operador será descrito mais detalhadamente na seção 3.4.

3.3 Técnica de *Pipelining*

Podemos melhorar a eficiência das consultas reduzindo o número de tabelas temporárias produzidas. Alcançamos essa redução combinando várias operações relacionais em um encadeamento (*pipeline)* de operações em que os resultados de uma operação são passados para a próxima operação no *pipeline*, eliminando assim o custo de escrita e leitura de relações temporárias. Em um plano de execução de consultas, a técnica de *pipelining* pode ser aplicada em dois níveis: inter-operadores e intra-operadores. A seguir será ilustrada a utilização da técnica de *pipelining*, caracterizando-se seus dois níveis de aplicação.

Considere uma junção em um par de relações seguida de uma projeção, como mostra a árvore de operadores apresentado na Figura 5. Quando a operação de junção gera uma tupla do seu resultado, aquela tupla é passada imediatamente à operação de projeção para processamento. Combinando a execução da operação de junção com a operação de projeção, evitamos criar um resultado intermediário e, em vez disso, criamos o resultado final diretamente. Neste caso, tem-se um *pipelining* inter-operadores.

Para a consulta apresentada na Figura 5, pode-se utilizar um operador de junção que processe a junção à medida que as tuplas da relação cliente e do resultado da operação de seleção são recebidas por este operador. Um exemplo clássico deste tipo de operador de junção é o *Merge Join*. Neste caso, tem-se um *pipelining* inter-operador. Portanto, no exemplo da Figura 5 todas as três operações podem ser colocadas em um *pipeline*, no qual os resultados da seleção são passados para a junção assim que gerados. Por sua vez, os resultados da junção são passados para a projeção assim que gerados, caracterizando dessa forma um *pipelining* inter-operadores. As exigências de memória são baixas, pois os resultados de uma operação não são armazenados por muito tempo. Por outro lado, como

resultado do *pipelining*, as entradas para as operações não precisam estar todas disponíveis de uma só vez para o processamento da consulta.

Agora, considere uma operação de junção cuja entrada à esquerda está no *pipeline*. Como ela está no *pipeline*, a entrada não se encontra disponível de uma vez para o processamento da operação de junção. Essa indisponibilidade limita a escolha do algoritmo de junção para ser usado. Por exemplo, o algoritmo *Sort-Merge Join* não pode ser usado, caso a entrada não esteja ordenada, já que não é possível ordenar uma relação até que todas as tuplas estejam disponíveis. Entretanto, o Index *Nested-Loop Join* pode ser usado: conforme as tuplas da relação operando à esquerda são recebidas, elas podem ser usadas para acessar as tuplas da relação operando à direita, supondo-se que esta relação esteja indexada pelo atributo de junção. Esse exemplo ilustra que as escolhas relativas ao algoritmo usado para uma operação e as escolhas relativas ao *pipelining* não são independentes.

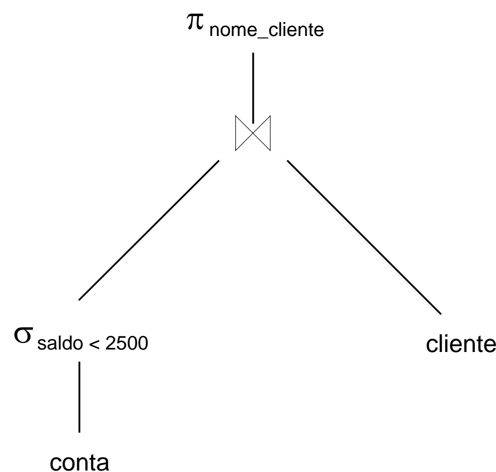


Figura 5 - Representação gráfica de uma consulta

No caso da junção, o uso efetivo de *pipelining* requer o uso de operadores que possam gerar tuplas resultado enquanto as tuplas estão sendo recebidas pelas entradas da operação. Em outras palavras, o operador de junção tem que garantir um *pipelining* intra-operador. Este é o caso do *Symmetric Hash Join* (SHJ) [29,42] que foi designado para permitir um alto grau de *pipelining* em bancos de dados

paralelos. O SHJ constrói duas tabelas *hash* em memória, uma para cada relação envolvida na operação de junção, através da aplicação de uma função *hash* no atributo de junção. Quando uma tupla chega da sua fonte de dados, ela é primeiramente inserida na tabela *hash* da sua relação, e imediatamente comparada com as tuplas da tabela *hash* da outra relação. As tuplas de resultado são produzidas quando tuplas coincidentes são encontradas e podem ser canalizadas para a operação superior logo após a comparação.

3.4 Operadores de Junção Adaptativos

Nesta seção apresentamos os principais operadores de junção adaptativos propostos para ambientes distribuídos. Ao final da seção será apresentada uma análise comparativa dos operadores apresentados, baseada nos critérios identificados na seção 3.4.

3.4.1 Double Pipelined Hash Join

O *Double Pipelined Hash Join* (DPHJ) é um operador de junção simétrico e incremental, que produz tuplas, imediatamente, mascarando, dessa forma, baixas taxas de transmissão de dados. Este operador foi integrado ao projeto Tukwila [42]. Como originalmente proposto, o DPHJ é executado com a chegada dos dados das relações envolvidas na junção. Para tanto, mantém duas tabelas *hash* em memória, uma para cada relação. Cada uma das relações envia tuplas para o operador de junção o mais rápido possível. Assim que o operador obtém uma tupla de uma relação operando, adiciona a tupla à tabela *hash* correspondente e compara-a com a tabela *hash* da relação oposta.

O DPHJ apresenta algumas vantagens com relação ao *Hash Join* convencional:

- As tuplas resultado são produzidas mais rapidamente;
- O operador é simétrico não sendo necessário, portanto, a definição de uma relação interna e externa na execução do operador.

Por outro lado, o DPHJ requer memória suficiente para manter as duas relações envolvidas na operação de junção em memória. Por isso, foi necessária a identificação de estratégias que solucionassem de forma eficiente o problema de limitação de memória (*overflow* de memória) durante a execução deste operador. O projeto Tukwilla [42] propôs e implementou dois algoritmos, descritos a seguir, com o objetivo de solucionar este problema durante a execução do DPHJ.

3.4.1.1 Incremental Left Flush

Este operador de junção estende o DPHJ, através da inclusão de um mecanismo para gerenciamento dos *overflows* de memória. Ao ocorrer *overflow* em qualquer um dos buckets das tabelas *hash* em memória, o algoritmo utiliza a estratégia de ler apenas as tuplas da relação à direita. Um *bucket* da tabela *hash* da relação à esquerda é transferido para o disco cada vez que ocorre um *overflow*. Esta estratégia resume a leitura e o processamento das tuplas da relação operando do lado esquerdo, permitindo que o operador gradualmente se transforme no *Hybrid Hash Join*.

Desta forma, considerando uma operação de junção envolvendo duas relações não ordenadas, A (relação à esquerda) e B (relação à direita), caso a memória seja completamente utilizada antes que a operação de junção seja finalizada, os seguintes passos devem ser executados:

1) Suspender a leitura das tuplas da relação A;
2) Transferir alguns *buckets* da tabela *hash* da relação A para o disco;
3) Continuar lendo as tuplas da relação B, gravando-as na tabela *hash* da relação B e comparando-as com as tuplas do *bucket* correspondente da tabela *hash* de A (parte das tuplas da relação A). Se uma tupla da relação B deve ser gravada em um *bucket* da tabela *hash* de B cujo *bucket* correspondente da tabela *hash* da relação A foi transferido para o disco, então marcar a tupla para processar depois;
4) Se ocorrer um *overflow* após a relação A já ter sido totalmente transferida para o disco, então um ou mais *buckets* da tabela *hash* da relação B são transferidos para disco;
5) Quando todas as tuplas de B tiverem sido lidas, retomar o processamento das tuplas da relação A. Se as tuplas da relação A devem ser gravadas em *buckets*

da tabela *hash* da relação A que foram transferidos para disco, então gravar as tuplas diretamente nos *buckets* em disco, senão comparar estas tuplas com as tuplas do *bucket* correspondente da tabela *hash* da relação B;

6) Uma vez que todas as tuplas dos *buckets* em memória foram processadas, executar um *Recursive Hybrid Hash Join* para unir os *buckets* transferidos para o disco na ocorrência dos *overflows*. A marcação da tupla no item 3 é utilizada para indicar que a tupla foi lida e armazenada no *bucket* em memória após a transferência para disco do *bucket* em memória correspondente da tabela *hash* da relação oposta. Desta forma, pode-se garantir que as tuplas marcadas não foram comparadas com nenhuma tupla do *bucket* correspondente da tabela *hash* da relação oposta. Assim sendo, as tuplas marcadas de um *bucket* em disco devem ser comparadas com todas as tuplas (marcadas ou não) do *bucket* em disco correspondente da relação oposta. A tupla que não possui marcação, já foi comparada com todas as tuplas do *bucket* correspondente da relação oposta que tinham sido gravadas no *bucket* antes da transferência para disco. Portanto, as tuplas desmarcadas de um *bucket* em disco devem apenas ser comparadas com as tuplas marcadas do *bucket* correspondente da relação oposta.

3.4.1.2 Incremental Symmetric Flush

Assim como o *Incremental Left Flush*, este operador de junção estende o DPHJ, através da inclusão de um mecanismo para gerenciamento dos *overflows* de memória. No *Incremental Symmetric Flush* quando ocorre *overflow* de memória, o algoritmo seleciona um par de *buckets* para transferir para o disco, transferindo os *buckets* correspondentes de ambas as relações.

Considerando uma operação de junção envolvendo duas relações não ordenadas, caso ocorra um *overflow* de memória antes que a operação de junção seja finalizada, os seguintes passos devem ser executados:

1) Selecionar um *bucket* da tabela *hash* de uma das relações e transferí-lo para disco juntamente com o *bucket* correspondente da tabela *hash* da relação oposta;

2) Continuar lendo as tuplas de ambas as relações, gravando-as no bucket apropriado da tabela *hash* de cada relação e comparando-as com as tuplas do *bucket* correspondente da tabela *hash* da relação oposta. Se uma nova tupla pertencer a um dos *buckets* que foi transferido para disco, marcar a tupla como nova e gravá-la no disco;

3) Uma vez que ambas as relações foram processadas, fazer um *Recursive Hybrid Hash Join* para processar as tuplas dos *buckets* que estão em disco. A marcação das tuplas do item 2 deve ser considerada a fim de evitar duplicações no resultado. As tuplas marcadas, ou seja, as tuplas que foram lidas e gravadas após a transferência do bucket para disco, devem ser processadas com todas as tuplas, marcadas ou não, do bucket em disco correspondente, pois ainda não foram comparadas com nenhuma delas, uma vez que o bucket a qual pertencem já havia sido transferido para disco. Já as tuplas não marcadas, ou seja, as tuplas que foram lidas e gravadas antes da transferência do bucket para disco, devem ser comparadas apenas com as tuplas marcadas, pois já foram comparadas com as tuplas não marcadas antes da transferência do bucket a qual pertencem para o disco.

3.4.2 Adaptative Symmetric Hash Join

O *Parallel Hash Ripple Join* (PHRJ) foi proposto por Haas et al. [27] com o objetivo de investigar a combinação de paralelismo com amostragem e sua aplicação em agregação *online*.

O *Parallel Hybrid Hash Join* (PHHJ) [16] é executado em duas fases. Primeiro o algoritmo redistribui as tuplas da relação A (relação de construção) para os sites (nós) onde a operação de junção será executada. As tuplas que chegam nos sites da redistribuição são adicionadas à tabela *hash* em memória enquanto outras são gravadas em disco. Então, as tuplas da relação B (relação de comparação) são redistribuídas para os sites correspondentes, e as tabelas *hash* construídas na primeira fase são comparadas com algumas tuplas de B, enquanto algumas tuplas remanescentes de B são também gravadas em disco. Finalmente, as porções de A e B residentes em disco são comparadas.

Desta forma, nenhuma tupla de resultado é produzida até que a relação de construção seja completamente redistribuída e então a segunda fase inicie o seu processamento. No PHRJ, a redistribuição das tuplas deve ocorrer simultaneamente com a junção. Assim, o algoritmo do PHRJ executa simultaneamente as três ações seguintes em cada site:

1. Redistribui as tuplas de A;
2. Redistribui as tuplas de B;
3. Executa a junção local das tuplas de A e B que estão chegando da redistribuição usando o algoritmo *Adaptative Symmetric Hash Join* (ASHJ).

O ASHJ é o algoritmo usado pelo PHRJ para solucionar os problemas de *overflow* de memória que podem ocorrer durante a execução de uma operação de junção através do PHRJ. Como citado anteriormente, o *Symmetric Hash Join* mantêm uma tabela *hash* para cada relação da operação de junção, conforme aplicação de uma função *hash* H nos atributos da junção, e requer que ambas as tabelas *hash* estejam em memória. O ASHJ estende o SHJ controlando os *overflows* de memória que possam ocorrer durante a sua execução.

O ASHJ funciona como descrito a seguir. Durante o processamento de uma junção entre duas relações A e B, as tuplas são armazenadas nas tabelas *hash* H_A e H_B, respectivamente, conforme Figura 6. As tabelas *hash* são divididas em *buckets*, onde cada *bucket* (E_A e E_B) tem uma parte em memória (MP_A e MP_B) e uma parte em disco (DP_A e DP_B). Considere que a memória disponível para execução da operação de junção está organizada em páginas. Por razões de performance, para cada parte em disco (DP_A e DP_B) é mantida uma página (P_A e P_B) em memória como um *buffer* de escrita.

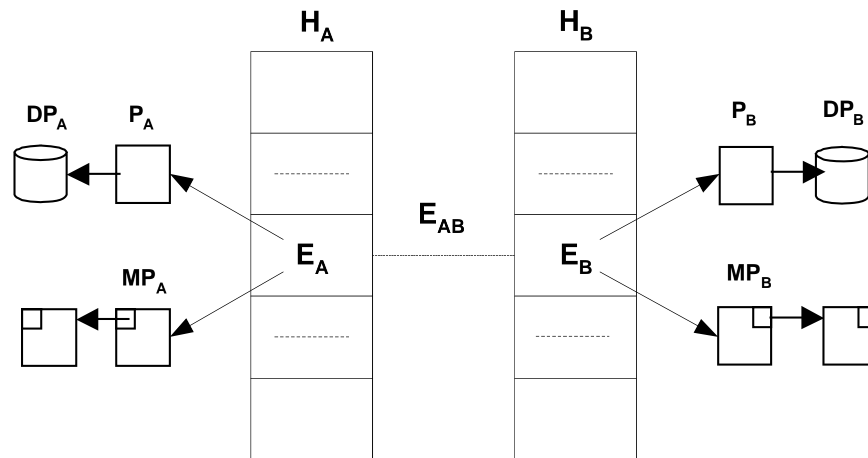


Figura 6 - Adaptative Symmetric Hash Join

A execução do ASHJ em cada nó é composta de duas fases:

1ª. Fase

O objetivo da primeira fase é comparar o maior número de tuplas possível com a memória disponível. Esta fase é completada quando todas as tuplas de ambas as relações forem recebidas. Inicialmente, cada par de *buckets* é formado por duas páginas de memória, associadas às partes em memória (MP_A e MP_B) das duas relações A e B. As páginas de memória remanescentes que serão alocadas durante a execução da operação de junção são organizadas em um *buffer pool* (BP).

Quando uma tupla de A ou B chega, uma função *hash* é aplicada para que o *bucket* que ela deve ser armazenada seja identificado. Se a tupla é de A, é inserida na parte em memória do *bucket* da tabela *hash* da relação A (MP_A) identificado com a aplicação da função *hash*, e comparada com as tuplas da parte em memória (MP_B) do *bucket* correspondente da relação B. Caso a parte em memória (MP_A) do *bucket* da tabela *hash* da relação A esteja cheia, o algoritmo verifica se existe alguma página de memória no BP, para que a mesma seja alocada para MP_A. Caso o BP esteja vazio, e não seja possível alocar uma página de memória para MP_A, a tupla é inserida no *buffer* de escrita (P_A). Sempre que o *buffer* de escrita (P_A) encher, o algoritmo transfere as suas tuplas para a parte em disco (DP_A) do *bucket*, e assim P_A pode aceitar tuplas novamente.

Em um caso especial, para um dado par de buckets E_{AB}, caso ocorra *overflow* em MP_A, se todas as tuplas de B já tiverem chegado e DP_B estiver vazio, as tuplas de A não precisam ser gravadas em disco, evitando trabalho desnecessário na segunda fase.

2ª. Fase

Após todas as tuplas de A e B terem sido recebidas pelo operador, a segunda fase é iniciada. Isto significa que apesar de chegarem em diferentes momentos na primeira fase, todas as tuplas entram na segunda fase no mesmo momento. E neste momento, para um determinado par de *bucket* E_{AB}, se ocorreu *overflow* primeiro em um bucket em memória da relação A (MP_A) deste par de *buckets*, podemos considerar que todas as tuplas de MP_A já foram comparadas com todas as tuplas de MP_B, e só é necessário comparar as tuplas de DP_A com as tuplas de MP_B.

A segunda fase funciona da seguinte forma: pares de *buckets* são selecionados aleatoriamente em um dado momento e as seguintes operações são executadas para cada um deles. Inicialmente uma tabela hash H_{DP}, criada através da aplicação de uma função *hash* H', é iniciada em memória. Considerando que na execução da primeira fase ocorreu *overflow* primeiro em MP_A, então as tuplas de DP_A (incluindo as tuplas de P_A) são lidas do disco para a memória. Ao mesmo tempo elas são comparadas com as tuplas de MP_B e inseridas em HD_P de acordo com os valores hash de H'.

As tuplas de DP_B (inclusive as tuplas de P_B) são lidas do disco para a memória. Ao mesmo tempo, são comparadas com as tuplas de DP_A através da função hash H' e da tabela hash HD_P. Ao final, a tabela hash H_{DP} é descartada, liberando a memória por ela utilizada.

3.4.3 Xjoin

O *Xjoin* [25] foi originalmente proposto com objetivo de obter bom tempo de resposta na presença de atraso no recebimento de tuplas em uma operação de junção. O *XJoin* foi desenvolvido baseado em dois princípios fundamentais:

produção incremental de resultados e continuidade da execução da consulta independente do bloqueio da entrega de tuplas de uma ou das duas fontes envolvidas na consulta.

O *XJoin* é uma extensão do SHJ que necessita de menos memória para processar a junção, pois permite que *buckets* de memória sejam transferidos para uma unidade de armazenamento secundário. Assim como o SHJ, o *XJoin* constrói duas tabelas *hash*, uma para cada relação. Estas tabelas são compostas por buckets, que por sua vez possuem uma parte em memória e outra em disco. Quando uma tupla de uma das relações envolvidas na operação de junção é recebida pelo operador, é automaticamente inserida em um *bucket* da tabela *hash* correspondente à relação a que pertence, através da aplicação de uma função *hash* no atributo da junção, e comparada com as tuplas do *bucket* correspondente da tabela *hash* da relação oposta, conforme mostra a Figura 7. Desta forma, mesmo que uma das relações esteja bloqueada o operador continua produzindo resultados.

O *Xjoin* possui três fases de execução descritas a seguir. A primeira compara as tuplas residentes em memória, agindo de forma similar ao SHJ. A segunda fase compara as tuplas transferidas para os *buckets* em disco, com as tuplas dos *buckets* em memória que ainda não foram transferidas para o disco, e a terceira fase compara as tuplas que ainda não foram comparadas, obtendo assim o resultado final da consulta.

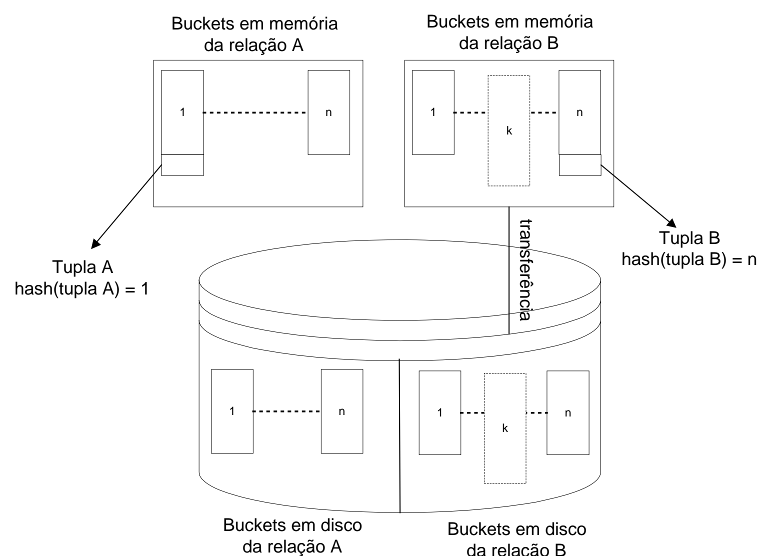


Figura 7 - Buckets do XJoin

1ª Fase

Organiza as tuplas em *buckets* localizados pela aplicação de uma função *hash* nos atributos do predicado da junção. Quando a memória enche, as tuplas do *bucket* com maior quantidade de tuplas são transferidas para o disco e o espaço é então liberado. Este processo é repetido sempre que a memória encher.

2ª Fase

Iniciada quando o recebimento das duas relações está bloqueado. Um *bucket* de disco é selecionado e suas tuplas são comparadas com as tuplas do *bucket* de memória correspondente da outra relação. As tuplas coincidentes são apresentadas como resultado.

Após o processamento completo de um *bucket* de disco, o operador verifica se o bloqueio do recebimento dos dados continua. Se positivo, processa outro *bucket* de disco, senão esta fase pára e a primeira fase é reiniciada.

3ª Fase

Esta fase é iniciada após o recebimento de todas as tuplas. É necessária porque a 1ª. e a 2ª. fase podem computar apenas resultados parciais. A primeira fase falha na união das tuplas que não tiveram na memória no mesmo momento, ou seja, quando duas tuplas correspondentes foram recebidas em tempos diferentes e uma delas já tenha migrado para o disco quando a outra chega. Estes pares de tuplas não foram unidos na primeira fase. A segunda fase falha na união de duas tuplas se uma delas não está na memória quando outra é trazida do disco e comparada com as tuplas em memória.

Esta fase une todos os *buckets* (memória+disco) das duas relações. Faz distinção entre a menor e a maior relação de entrada quando está operando. Após identificar a menor relação, as tuplas deste *bucket* de disco são colocadas em memória e uma tabela *hash* é criada para elas. A tabela *hash* é então comparada com os *buckets* de disco e memória correspondentes da outra relação. Este

processo é repetido para todas os *buckets* remanescentes que estão em disco. O processamento é similar ao *Hybrid Hash Join*, e esta fase trabalha pouco porque produz apenas o que não foi produzido na primeira e segunda fase.

3.4.3.1 Gerenciamento de Duplicações no Resultado e Resultados Incompletos

Os múltiplos estágios do Xjoin podem criar duplicações de tuplas no resultado durante a execução da segunda e terceira fases. O gerenciamento destas duplicações é feito através de um mecanismo de prevenção baseado em *timestamps*, que são mantidos pelo Xjoin. O Xjoin aumenta a estrutura das tuplas, que passam a conter dois *timestamps*, cujos valores são definidos durante a execução da primeira fase. Um *timestamp* chamado *Arrival Timestamp* (ATS) é atualizado na primeira vez que a tupla é recebida da sua fonte na memória, e o segundo *timestamp* chamado *Departure Timestamp* (DTS) é atualizado quando a tupla é transferida da memória para o disco. O ATS e DTS juntos descrevem o intervalo de tempo no qual a tupla ficou em memória, e estes valores depois de atualizados não mudam mais, sendo atualizados uma única vez. Como citado anteriormente, a segunda e a terceira fase podem gerar resultados duplicados. Tuplas comparadas durante a execução destas fases podem já ter sido comparadas na primeira fase ou por uma execução anterior da segunda fase. Para obter as tuplas que já foram comparadas no primeiro estágio, o XJoin identifica as tuplas que estiveram na memória ao mesmo tempo, ou seja, tuplas cujos intervalos de tempo entre o DTS e o ATS se sobrepõem, e não as considera na execução da segunda e terceira fases.

Estes *timestamps* evitam a produção de tuplas duplicadas na execução da primeira fase, mas não evitam estas duplicações nas múltiplas execuções da segunda fase. A segunda fase compara as tuplas que já foram transferidas da memória para o disco com as tuplas da outra relação que estão em memória. Se um *bucket* em disco é usado em várias execuções da segunda fase, tuplas duplicadas podem ser criadas.

Este problema é resolvido pelo Xjoin através do uso de *timestamps* que guardam o momento no qual a segunda fase utilizou cada *bucket* em disco. Estes

timestamps adicionais são gravados como uma lista encadeada associada a cada *bucket* de disco. Esta lista contém uma entrada para cada vez que a segunda fase utiliza o *bucket* em disco na sua execução. As entradas da lista são formadas por dois *timestamps* na forma {DTSlast, ProbeTS}, onde DTSlast é o valor do *timestamp* DTS da última tupla do *bucket* em disco que foi comparada com as tuplas residentes em memória da relação anterior, e ProbeTS o momento em que a segunda fase foi executada.

Para identificar se uma tupla de um *bucket* em disco já foi comparada com uma tupla do bucket em memória correspondente da relação oposta na execução da segunda fase, o Xjoin lê a lista encadeada construída para o *bucket* em disco e identifica se esta tupla já foi comparada com uma tupla do *bucket* em memória. Então verifica se a tupla do *bucket* em memória já estava na memória durante alguma das comparações identificadas no passo anterior, através dos valores dos *timestamps* ATS e DTS. Se for deduzido que a tupla do *bucket* em disco já foi comparada com a tupla do *bucket* em memória, estas tuplas não são mais comparadas. Esta mesma prática pode ser usada para identificar a produção de tuplas duplicadas na terceira fase.

3.5 Análise Comparativa

Nesta seção será feita uma análise comparativa entre os operadores de junção adaptativos apresentados na seção anterior. Esta análise esta baseada em três critérios distintos, identificados como fundamentais por representar fatores que degradam a performance da consulta em ambientes de computação móvel.

3.5.1 Critérios de Comparação

3.5.1.1 Tuplas processadas assim que chegam ao operador de junção

A adoção deste critério tem como objetivo verificar a quantidade de tuplas (pertencentes ao resultado da operação de junção) que são produzidas pelo

operador em um dado intervalo de tempo, ou seja, verificar a “*taxa de pipelining*” do operador.

Este critério é altamente influenciado pela estratégia utilizada pelo operador de junção no recebimento de tuplas pertencentes a *buckets* de memória que sofreram pelo menos um *overflow* de memória, ou seja, que já tiveram tuplas transferidas para o disco. Muitos operadores de junção, quando recebem tuplas que devem ser alocadas em *buckets* de memória que já foram transferidos para disco, transferem-as diretamente para disco, adiando o processamento destas tuplas para uma fase final da execução do operador e conseqüentemente diminuindo a “*taxa de pipelining*” do operador.

3.5.1.2 Continuação do processamento durante o bloqueio no recebimento de tuplas de ambas as relações

A escolha deste critério tem como objetivo identificar interrupções no processamento do operador de junção quando as tuplas das relações envolvidas deixam de ser recebidas pelo operador. Interrupções no processamento da operação de junção podem impactar a performance da consulta, o que justifica a adoção deste critério.

3.5.1.3 Verificação da necessidade de armazenamento da tupla em memória

A escolha deste critério é justificada pela otimização do uso da memória disponível para a execução da operação de junção que ocorre quando o mesmo é uma característica de um operador de junção. Esta otimização é gerada pela verificação da real necessidade de armazenamento da tupla em memória, o que pode, dependendo das características (quantidade de tuplas, por exemplo) e das relações envolvidas na operação de junção, evitar muitos *overflows*, minimizando o número de transferências de *buckets* de memória para disco, e otimizando não apenas o uso da memória disponível, como também a performance do operador, que fará menos operações de leitura e escrita em disco.

3.5.2 Análise dos operadores de junção apresentados com relação aos critérios identificados

Apresenta-se a seguir uma análise comparativa detalhada de cada operador de junção face aos critérios adotados. Esta análise foi resumida na tabela 2.

3.5.2.1 Tuplas processadas assim que chegam no operador de junção

Conforme citado anteriormente, este critério é determinado pela estratégia utilizada no recebimento de tuplas pertencentes a *buckets* de memória que sofreram pelo menos um *overflow* de memória, ou seja, que já tiveram tuplas transferidos para disco. Quanto a esta característica observou-se o seguinte:

- *Adaptative Symmetric Hash Join*

As tuplas, que chegam para um *bucket* em memória que já tem tuplas transferidas para disco, são alocadas diretamente no *bucket* em disco. Portanto, tais tuplas não são comparadas com as tuplas do *bucket* em memória correspondente da relação oposta. Obviamente, tal característica retarda o processamento destas tuplas, que só serão processadas quando todas as tuplas envolvidas na junção tiverem sido recebidas, fazendo com que o operador apresente baixas taxas de entrega de tuplas na ocorrência de *overflows* de memória;

- *Incremental Symmetric Flush*

Armazena diretamente em disco as tuplas que estão chegando para *buckets* em memória que já sofreram *overflow* de memória, apresentando, dessa forma, a mesma restrição que existe no operador *Adaptative Symmetric Hash Join*;

- *Incremental Left Flush*

Apresenta um grave problema, pois na ocorrência do primeiro *overflow* de memória, o operador seleciona uma das relações envolvidas na junção e bloqueia o recebimento de tuplas desta relação, passando a receber somente as tuplas da

relação oposta. Sempre que um *overflow* ocorrer, um *bucket* de memória da relação, cujo recebimento de tuplas foi bloqueado, é transferido para o disco. As tuplas da relação bloqueada só voltam a ser lidas, quando todas as tuplas da outra relação chegarem. Caso a outra relação fique bloqueada ou tenha uma baixa taxa de entrega de tuplas, a performance do processamento da consulta poderá ser muito afetada comprometendo a quantidade de tuplas produzidas;

- *Xjoin*

Não apresenta problemas com relação a este aspecto, pois, independentemente da ocorrência ou não de *overflows* de memória, as novas tuplas que chegam são imediatamente processadas, aumentando a produção de tuplas de resultado.

3.5.2.2 Continuação do processamento durante bloqueio no recebimento de tuplas de ambas as relações

Analisando as propostas com relação à continuação do processamento durante o bloqueio no recebimento de dados das relações envolvidas na junção, observamos que apenas o *XJoin* continua o processamento da consulta na ocorrência de bloqueio na entrega de tuplas de ambas as relações, mascarando o tempo de espera da entrega de tuplas. Nestas situações, o *XJoin* processa as tuplas dos *buckets* que estão em disco, até que novas tuplas cheguem na memória para processamento. Este processamento necessita de um mecanismo de controle adicional para evitar a produção de resultados incompletos e/ou duplicados. O mecanismo implementado pelo *XJoin* baseia-se em *timestamps* que são adicionados à estrutura das tuplas durante a execução da primeira e da segunda fase do operador.

3.5.2.3 Verificação da necessidade de armazenamento da tupla em memória

Analisando as propostas com relação à verificação da necessidade de armazenamento da tupla em memória, observamos que apenas o *Adaptative*

Symmetric Hash Join possui esta característica, otimizando assim o uso da memória disponível para execução do operador. Para um dado par de *buckets*, as tuplas de uma das relações envolvidas na operação de junção não precisam ser inseridas em memória, caso todas as tuplas da relação oposta já tenham sido recebidas e a parte em disco do *bucket* estiver vazia, evitando trabalho desnecessário na segunda fase.

Tabela 2 – Comparativo de operadores adaptativos

Características	**Adaptative Symmetric Hash Join**	**Incremental Symmetric Flush**	**Incremental Left Flush**	**Xjoin**
As tuplas são processadas assim que chegam ao operador de junção	Apenas se não tiver ocorrido *overflow* no *bucket* destino	Apenas se não tiver ocorrido *overflow* no *bucket* destino	Se tiver ocorrido *overflow* em qualquer um dos *buckets* das tabelas *hash* envolvidas, suspende o processamento das tuplas de uma das relações	Sim
Continuação do processamento durante bloqueio no recebimento de tuplas de ambas as relações	Não	Não	Não	Sim
Verificação da necessidade de armazenamento da tupla em memória	Sim	Não	Não	Não

4. MOBIJOIN

4.1 Introdução

Neste capítulo apresentamos um operador de junção para processamento de consultas, chamado MobiJoin, que foi desenvolvido baseado em três princípios fundamentais: *(i)* produção incremental de resultados à medida que os dados são disponibilizados, *(ii)* continuidade no processamento da consulta mesmo que a entrega dos dados esteja bloqueada, e: *(iii)* reação a situações de limitação de memória durante a execução do operador. A idéia é que o MobiJoin comporte-se como um operador de consulta adaptativo, reagindo a eventos que podem aumentar substancialmente o tempo de resposta no acesso a bancos de dados móveis.

O MobiJoin é baseado nos operadores *Symmetric Hash Join* (SHJ) [33,40] e *XJoin* [25]. Conforme originalmente proposto, o SHJ requer que ambas as tabelas *hash* das duas relações de entrada da junção sejam mantidas em memória durante a execução da consulta. Em conseqüência disso, o SHJ não pode ser utilizado para executar uma junção entre relações de entrada quando há limitação de memória, cenário comum no processamento de consultas em MDBCs. O MobiJoin estende o SHJ, adicionando um mecanismo de transferência dos dados que estão na memória para o disco quando necessário. Porém, esta extensão não é suficiente para mascarar tempos de espera significantes no recebimento dos dados das fontes remotas, e para resolver este problema, foi desenvolvida uma estratégia que oportunamente utiliza os tempos de espera para produzir resultados mais cedo.

Este capítulo está organizado como descrito a seguir. Na seção 4.2 apresentamos o cenário de execução do operador de junção MobiJoin. As fases de execução do operador são descritas na seção 4.3. Na seção 4.4 exemplificamos a execução de uma junção utilizando o operador MobiJoin. Na seção 4.5

apresentamos a estimativa de custo de execução do operador. A discussão dos trabalhos relacionados é apresentada na seção 4.6.

4.2 Cenário de Execução do Operador

Nesta dissertação assumimos o ambiente de execução descrito a seguir e ilustrado através da Figura 8. Considere uma comunidade de banco de dados móveis M. Suponha que os bancos de dados pertencentes a M residem em computadores interconectados através de uma rede *ad hoc*. Agora considere um banco de dados DB_1, que reside em um computador móvel C_1, pertencente a M. O banco de dados DB_1 pode ser compartilhado com os outros membros da comunidade, e da mesma forma, o usuário do computador móvel C_1 pode consultar objetos armazenados em outros membros de M. Os computadores móveis C_2 e C_3, que possuem os bancos de dados DB_2 e DB_3 respectivamente, também fazem parte da comunidade e assim como C_1 compartilham seus bancos de dados entre os membros da mesma. Considere a execução de uma junção entre duas tabelas A e B que estão armazenadas em DB_2 e DB_3. Esta operação de junção é o resultado de uma consulta submetida pelo computador móvel C_1 e será resolvida através da execução local em C_2 e C_3 das operações necessárias sobre as tabelas A e B, e envio dos resultados das execuções locais para o computador móvel C_1. Em C_1, será executada a operação de junção, produzindo, assim, o resultado final da consulta.

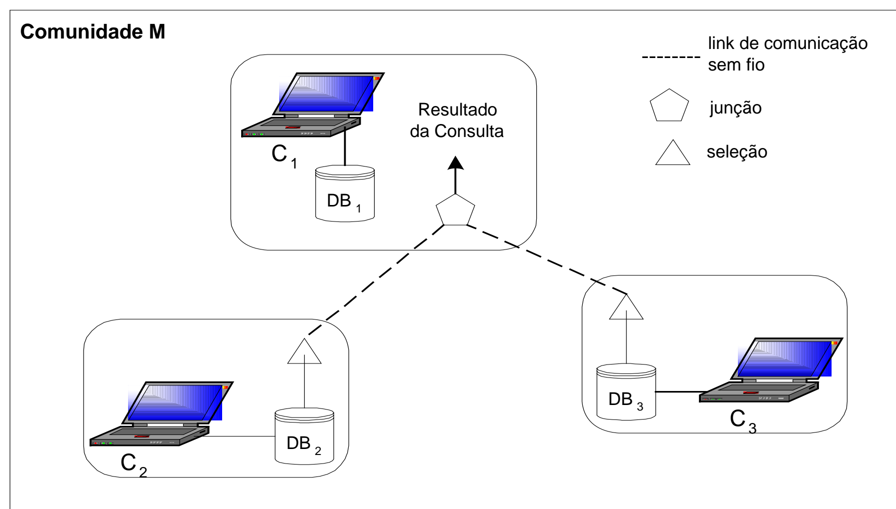


Figura 8 - Cenário de execução do MobiJoin

4.3 Funcionamento do Operador

O MobiJoin possui três fases de execução. A primeira fase garante a execução do operador com base na técnica de *pipelining* (intra-operador), garantindo, desta forma, produção incremental de resultados à medida que tuplas das tabelas envolvidas na junção são disponibilizadas. Nesta técnica há um *pipelining* das tuplas dos resultados parciais da consulta, de forma que o operador mais alto na hierarquia espera pelos resultados que estão abaixo dele.

A primeira fase tem ainda a funcionalidade de reagir ao evento de limitação de memória disponível para a execução do operador. A segunda fase garante continuidade no processamento do operador, quando o recebimento de tuplas (das relações envolvidas na junção) esteja bloqueado. A terceira fase é executada para garantir a corretude nos resultados produzidos pelo operador.

1ª Fase

O objetivo desta fase é comparar a maior quantidade de tuplas possível no espaço de memória disponível no momento da execução da junção. Esta fase é iniciada quando as primeiras tuplas das relações envolvidas na junção começam a chegar. É executada enquanto tuplas de pelo menos uma relação estiverem chegando, e é finalizada quando todas as tuplas de ambas as relações tiverem sido recebidas. Caso haja uma interrupção no recebimento das tuplas de ambas as relações, a execução desta fase é suspensa, uma vez que não haverá tuplas disponíveis para processamento da junção. Esta fase pode ser reiniciada quando novas tuplas voltarem a chegar.

O primeiro passo a ser executado na primeira fase do MobiJoin é a criação de tabelas *hash* (uma para cada relação envolvida na operação de junção) com base na aplicação de uma função *hash* h sobre os atributos da junção. Cada tabela *hash*, conforme mostra a figura 9, é composta por *buckets* em memória que formam pares com os *buckets* correspondentes da relação oposta (P_{AB1} e P_{AB2} na figura 9). Por exemplo, na figura 9, os *buckets* M_{A1} e M_{B1} formam o par P_{AB1} e os *buckets* M_{A2} e M_{B2} compõem o par P_{AB2}.

Cada *bucket* é formado por um conjunto de tuplas, onde cada tupla é identificada através do *identificador único de tupla do bucket* (BTID), que pode ser um número seqüencial incrementado à medida que as tuplas vão chegando no *bucket*. Este identificador é adicionado a todas as tuplas que são alocadas em um *bucket* e representa também a quantidade de tuplas alocadas no *bucket* até o momento.

À medida que vão chegando, as tuplas são alocadas em um *bucket* de acordo com a aplicação da função *hash* sobre o atributo da junção. Por exemplo, na Figura 9, as tuplas da tabela A podem ser alocadas nos *buckets* M_{A1} e M_{A2} e as tuplas da tabela B podem ser alocadas nos *buckets* M_{B1} e M_{B2}. Após a alocação de uma tupla em um *bucket*, as tuplas deste *bucket* são comparadas com as tuplas do *bucket* correspondente da relação oposta. Dois *buckets* de tabelas *hash* distintas são considerados correspondentes se apresentam o mesmo endereço. Em outras palavras, a aplicação da função *hash* sobre os atributos da junção de dois *buckets* correspondentes, por exemplo M_{A1} e M_{B1}, retorna sempre o mesmo valor (endereço).

Os *buckets* armazenados em memória possuem o mesmo tamanho. Quando o espaço de memória de um dado *bucket* M for completamente preenchido e houver a necessidade de alocar novas tuplas em M (evento conhecido com *overflow* de memória), o conteúdo de M é transferido para uma área em disco, denominada de *bucket* em disco. Após esta transferência, a memória é liberada e o *bucket* pode, então, receber novas tuplas. Caso ocorra a transferência de um *bucket* de memória para disco, o seguinte fenômeno poderá ocorrer: tuplas do *bucket* correspondente da relação oposta, ou seja, do *bucket* que não foi transferido para disco, que chegarem após a transferência, não terão sido comparadas com as tuplas que foram transferidas para o disco, o que pode gerar resultados incorretos (incompletos) para a operação de junção. Por exemplo, caso tuplas do *bucket* M_{A1} (figura 9) sejam transferidas para disco, tuplas que chegarem para M_{B1} após esta transferência não serão comparadas com as tuplas de M_{A1} transferidas para disco.

Para evitar o fenômeno supracitado, definimos um mecanismo para identificar quais as tuplas das duas relações que já foram comparadas. Este mecanismo funciona como descrito a seguir. Sempre que um *bucket* for transferido da memória

para o disco, uma tabela de controle é gerada para o par de *buckets* do *bucket* transferido. Por exemplo, caso o *bucket* M_{A1} seja transferido para disco, será gerada, então, uma tabela de controle para o par P_{AB1}.O número de linhas e colunas da tabela de controle é correspondente ao número de tuplas recebidas em cada relação. Na geração da tabela, uma relação envolvida na junção é definida como linha e outra relação como coluna. Cada célula da tabela pode assumir valores 1 ou 0, indicando a ocorrência (1) ou não (0) da comparação entre as tuplas por ela representadas. Portanto, a tabela de controle representa uma tabela de *bits*. As células da tabela apresentam inicialmente o valor 1, já que podemos garantir neste caso que todas as tuplas do par de *buckets* foram comparadas.

Para ilustrar o funcionamento da primeira fase do MobiJoin, considere o cenário descrito na seção 2.1. Assim sendo, o computador C_1 receberá tuplas das relações A e B, selecionadas dos bancos de dados DB_2 e DB_3, para executar uma operação de junção. Neste caso, o operador MobiJoin ao iniciar o recebimento das tuplas em C_1 cria duas tabelas *hash* H_A e H_B, como mostrado na Figura 9. Suponha que a tupla t_{A1} foi a primeira a chegar. Esta tupla deve ser alocada no *bucket* M_{A1} conforme aplicação de função *hash* sobre atributo de junção. Após isto, o BTID é obtido e adicionado à tupla. Neste caso, como t_{A1} é a primeira tupla alocada no *bucket*, o valor de BTID será 1. Outras tuplas da relação A (t_{A2} e t_{A3}) e da relação B (t_{B1} e t_{B2}) também chegam e são processadas pelo MobiJoin da mesma forma como foi processada a tupla t_{A1}. A Figura 9 ilustra o estado de execução do MobiJoin após o processamento destas tuplas.

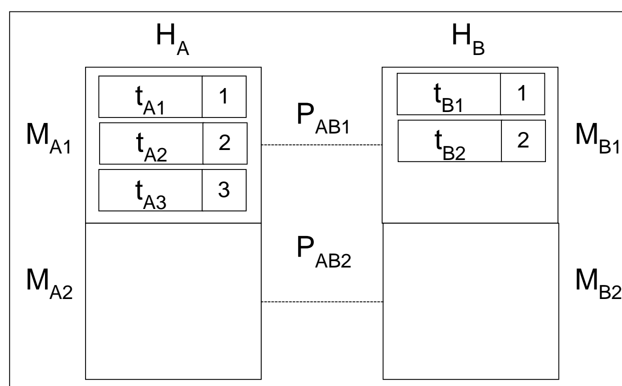


Figura 9 - Estado de execução antes da transferência de bucket para disco

Agora, suponha que uma nova tupla da relação A, t_{A4}, chega e deve ser alocada em M_{A1}. Considere, no entanto, que não há mais espaço disponível em M_{A1}, tornando necessária uma transferência das tuplas de M_{A1} para disco. Após a transferência das tuplas de M_{A1} para o disco, a área de M_{A1} é liberada para alocação de novas tuplas. Com esta transferência, torna-se necessária a geração de uma tabela de controle para o par de *buckets* P_{AB1}, composto pelos *buckets* M_{A1} e M_{B1}. Para a geração da tabela de controle, suponha que a relação A é definida como linha e a relação B como coluna. A tabela possuirá, então, 3 linhas e 2 colunas, de acordo com os BTIDs da última tupla dos *buckets* M_{A1} e M_{B1} (veja Figura 9). A Figura 10 mostra o estado de execução do MobiJoin após a transferência do *bucket* e a tabela de controle gerada por ele. Observe que as células da tabela de controle possuem valor 1. Tal fato indica que as tuplas de M_{A1} (que foram descarregadas em disco) já foram comparadas com as tuplas de M_{B1}.

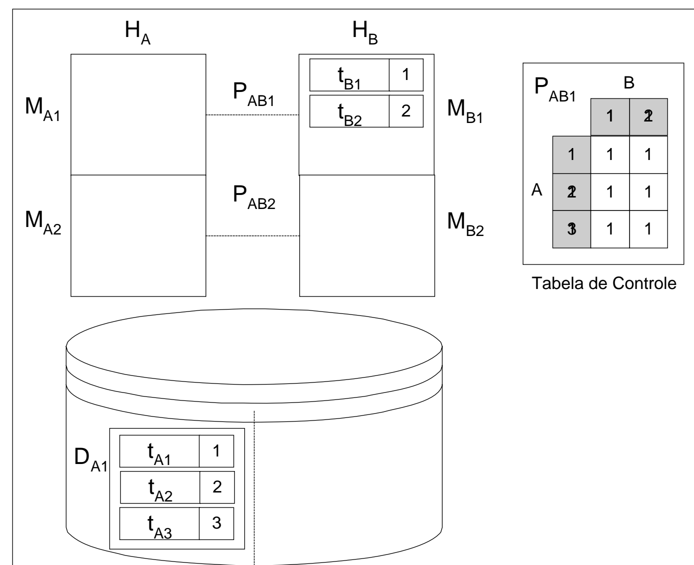


Figura 10 - Estado de execução após transferência do *bucket* para disco

Após a construção da tabela de controle para um par de *buckets*, esta tabela de controle precisa ser redimensionada sempre que uma nova tupla chegar e tiver que ser alocada em um dos *buckets* do par. Em outras palavras, uma nova linha ou coluna deve ser adicionada à tabela quando uma nova tupla for alocada em um dos *buckets* do par.

É importante ressaltar que, antes de armazenar uma tupla no *bucket* em memória, o operador MobiJoin verifica se a relação oposta já foi completamente recebida. Neste caso, apenas é necessário armazenar esta tupla no *bucket* em memória se o *bucket* em memória correspondente da relação oposta possuir *bucket* em disco. Caso contrário, como a relação oposta já foi completamente recebida, podemos garantir que todas as tuplas do *bucket* correspondente já chegaram e estão armazenadas no *bucket* em memória. Portanto, as próximas tuplas que estão chegando só precisam ser comparadas com as tuplas do *bucket* em memória da relação oposta e podem ser descartadas após a comparação, otimizando assim o uso da memória disponível e prevenindo futuros *overflows* de memória. A Figura 11 apresenta o pseudocódigo da primeira fase do algoritmo MobiJoin.

```
Enquanto o evento Bloqueio na Recepção de Dados não ocorrer faça
    Receber tuplas
    Para cada tupla recebida faça
        Aplicar função hash
        Se o evento Overflow no Bucket ocorrer então
            Transferir bucket para o disco
            Se tabela de controle do par de buckets não existe
            então
                Gerar Tabela
            Fim Se
            Adicionar BTID à tupla
            Alocar tupla no bucket em memória
        Senão
            Se o bucket está vazio e não possui partição em disco
                Inicializar BTID
            Senão
                Incrementar BTID
            Fim Se
            Se tabela de controle do par de buckets existe
            então
                Redimensionar Tabela
            Fim Se
            Se o evento "Fim de recepção de uma relação" ocorreu para a relação oposta então
                Se bucket correspondente da relação oposta possui partição em disco então
                    Adicionar BTID à tupla
                    Alocar tupla no bucket em memória
                Fim Se
            Senão
                Adicionar BTID à tupla
                Alocar tupla no bucket em memória
            Fim Se
            Para cada tupla do bucket correspondente na relação oposta faça
                Comparar tuplas
                Se tabela de controle do par de buckets existe
                então
                    Atualizar tabela
                Fim Se
            Fim Para
        Fim Se
    Fim Para
Fim Enquanto
```

Figura 11 - Pseudocódigo da primeira fase do algoritmo MobiJoin

2ª Fase

Esta fase é iniciada quando o recebimento de tuplas das duas relações envolvidas na junção é interrompido. Neste caso, um *bucket*, em disco, é selecionado e suas tuplas são comparadas com as tuplas do *bucket*, em memória, correspondente da relação oposta. As tuplas que satisfazem a condição de junção são inseridas no resultado.

Esta fase utiliza a tabela de controle para evitar a produção de tuplas duplicadas no resultado, uma vez que tuplas que estão em disco já podem ter sido comparadas com algumas tuplas do *bucket* correspondente da relação oposta que estão em memória. Este controle é realizado pela leitura das células da tabela de controle. Antes de ler as tuplas do *bucket* em disco de uma relação, o MobiJoin lê a tabela de controle associada e obtém os BTIDs das tuplas que ainda não foram processadas (com valor 0) desta relação. Após obter esta informação, o MobiJoin lê as tuplas não processadas do *bucket* em disco da relação e compara com as tuplas do *bucket* em memória correspondente da relação oposta.

Após o processamento de todas as tuplas do *bucket* em disco, o MobiJoin verifica se o bloqueio do recebimento dos dados continua. Se o bloqueio continua, outro *bucket* de disco é processado, caso contrário, a execução desta fase é suspensa e a primeira fase é reiniciada. A Figura 12 apresenta o pseudocódigo da segunda fase do algoritmo MobiJoin.

```
Se o evento Bloqueio na Recepção de Dados está ocorrendo faça
   Selecionar uma tabela de controle
   Ler tabela de controle para obter tuplas não processadas das relações A e B
   Ler tuplas não processadas do bucket em disco da relação A associado à tabela de controle
   Para cada tupla lida faça
      Comparar tupla com tuplas do bucket de memória da relação B não processadas
   Fim Para
   Ler tuplas não processadas do bucket em disco da relação B associado à tabela de controle
   Para cada tupla lida faça
      Comparar tupla com tuplas do bucket de memória da relação A não processadas
   Fim Para
Fim Se
```

Figura 12 - Pseudocódigo da segunda fase do algoritmo MobiJoin

3ª Fase

Esta fase é iniciada após o recebimento de todas as tuplas das relações envolvidas na operação de junção. Esta fase se faz necessária, pois a 1ª e a 2ª fases podem computar apenas resultados parciais. A primeira fase falha na junção das tuplas que não estiveram na memória no mesmo momento, ou seja, quando duas tuplas de *buckets* correspondentes foram recebidas em tempos diferentes e uma delas já tenha migrado para o disco quando a outra é alocada no *bucket* em memória. A segunda fase falha na junção das tuplas que ainda não chegaram no *bucket* de memória quando o mesmo é comparado com o *bucket* em disco da relação oposta correspondente.

A terceira fase de execução do MobiJoin tem como funcionalidade realizar a junção de todas as tuplas dos pares de *buckets* correspondentes que já foram descarregados em disco, e que ainda não foram comparadas. Para cada tabela de controle, o MobiJoin executa os seguintes passos. Inicialmente, o MobiJoin lê os valores das células, identificando, dessa forma, as tuplas que ainda não foram processadas. O próximo passo é selecionar as tuplas de um *bucket* de disco de A que não foram processadas e compará-las com as tuplas do *bucket* em memória correspondente da relação B. Para cada tupla lida da relação A, uma função *hash* h' é aplicada sobre o atributo de junção para alocar a tupla em uma tabela *hash* em memória. Após o processamento das tuplas da relação A, as tuplas não processadas do *bucket* em disco de B são lidas e comparadas com as tuplas não processadas do *bucket* de memória correspondente da relação A. As tuplas de A, que estavam em disco, foram alocadas em *buckets* de memória com base na aplicação da função *hash* h'. Portanto, é necessário comparar estas tuplas de A com as da relação B. Para tanto, para cada tupla de B é aplicada a função *hash* h' sobre o atributo de junção, retornando desta forma o endereço do *bucket* que contem tuplas de A e que devem ser comparadas com as tuplas de B. A Figura 13 fornece o pseudocódigo da terceira fase do algoritmo MobiJoin.

```
Se o evento Fim de Recepção de Ambas as Relações ocorreu então
    Para cada tabela de controle faça
        Ler tabela de controle para obter tuplas ainda não processadas das relações A e B
        Ler tuplas não processadas do bucket de disco da relação A
        Para cada tupla lida da relação A faça
            Comparar com tuplas não processadas do bucket de memória correspondente da relação B
            Aplicar função hash H’
            Armazenar tupla em uma tabela hash TH
        Fim Para
        Ler tuplas não processadas do bucket em disco da relação B
        Para cada tupla lida faça
            Comparar com tuplas não processadas do bucket de memória correspondente da relação A
            Aplicar função hash H’
            Comparar com as tuplas da tabela hash TH
        Fim Para
    Fim Para
    Liberar tabela hash da memória
Fim Se
```

Figura 13 - Pseudocódigo da terceira fase do algoritmo MobiJoin

4.4 Exemplo de Execução do Operador

Será apresentado, nesta seção, um exemplo de execução do MobiJoin, considerando o cenário descrito na seção 2.1. Ao se iniciar o recebimento de tuplas de duas relações A e B em C_1, serão criadas duas tabelas *hash* em memória, H_A e H_B. À medida que as tuplas vão chegando a função *hash* h é aplicada para identificar os *buckets*, em memória, onde as tuplas devem ser alocadas, e um BTID é adicionado a tupla. A figura 14 mostra a execução do operador após a chegada das tuplas t_{a1}, t_{a2}, t_{a3}, t_{b1}, t_{b2}, t_{a4}, e t_{b3}.

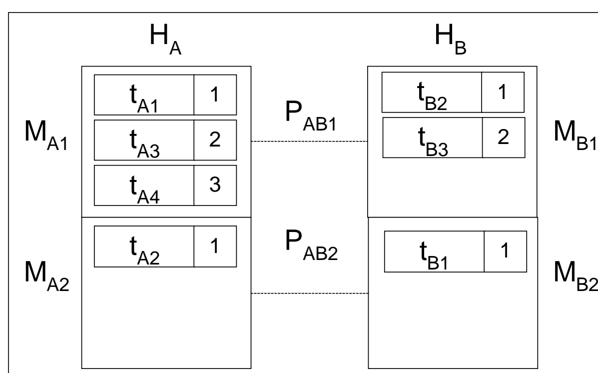


Figura 14 - Estado de execução antes da chegada de ta5

Suponha agora que a tupla t_{a5} é recebida pelo operador MobiJoin. Após a aplicação da função *hash* h sobre t_{a5}, encontra-se o endereço do *bucket* M_{A1}. Contudo, este *bucket* já está com sua capacidade máxima de armazenamento atingida, sendo detectado, portanto, um *overflow* de memória no *bucket* M_{A1}. Neste caso, uma transferência do *bucket* para o disco deve ocorrer. Conforme o algoritmo da primeira fase, apresentado na Figura 11, sempre que uma transferência de *bucket* for executada, uma tabela de controle é gerada para o par de *buckets* correspondentes, neste caso P_{AB1}. A Figura 15 ilustra o estado da execução do operador após a transferência do *bucket* M_{A1} para o disco, a alocação da tupla t_{a5} no *bucket* M_{A1} e a geração e atualização da tabela de controle.

O processamento das tuplas recebidas continua e a Figura 16 apresenta o estado da execução do operador após o processamento das tuplas t_{a6}, t_{b4}, t_{b5} e t_{b6}. Suponha agora que a tupla t_{b7} é a próxima a chegar e foi recebida pelo operador MobiJoin. Ao se aplicar a função *hash* h sobre a tupla t_{b7}, identifica-se que esta tupla deve ser alocada no *bucket* M_{B2} que está com sua capacidade máxima de armazenamento atingida. As tuplas do *bucket* são transferidas para o disco e o espaço em memória é liberado para a recepção da nova tupla no *bucket* M_{B2}. A Figura 17 mostra o estado da execução do operador depois deste *overflow* e da alocação da tupla t_{b7} no *bucket* M_{B2}, bem como as tabelas de controle atualizadas.

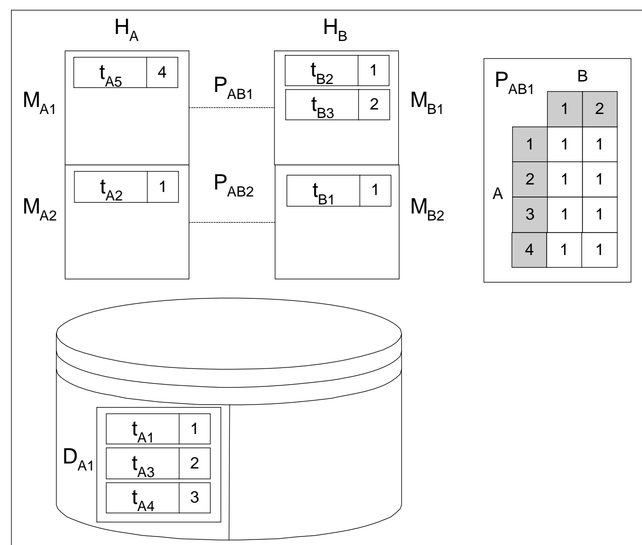
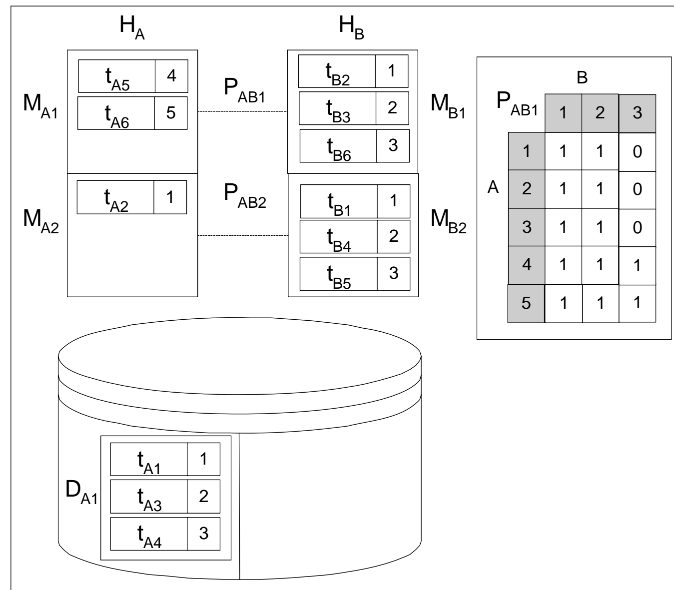
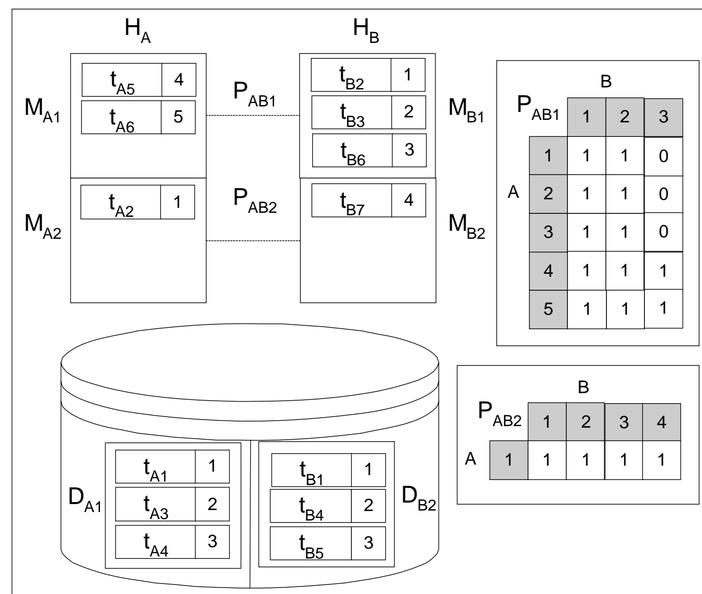


Figura 15 - Estado de execução após a transferência de M_{A1}



P_AB1	1	2	3
1	1	1	0
2	1	1	0
3	1	1	0
4	1	1	1
5	1	1	1

Figura 16 - Estado de execução até o processamento de tB6



P_AB1	1	2	3
1	1	1	0
2	1	1	0
3	1	1	0
4	1	1	1
5	1	1	1

P_AB2	1	2	3	4
1	1	1	1	1

Figura 17 - Estado de execução após alocação da tupla t_{B7}

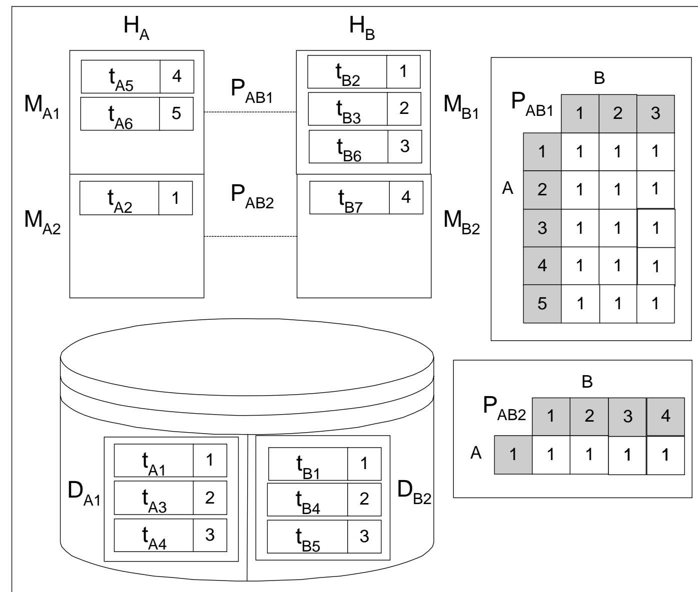


Figura 18 - Estado de execução após o processamento da 2a. fase

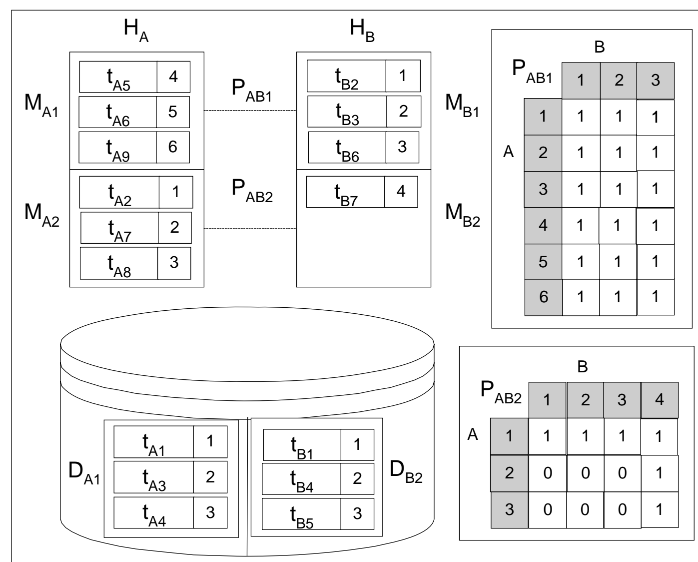


Figura 19 - Estado final da execução da 1a fase

Agora, suponha que após a alocação da tupla t_{b7} no *bucket* M_{B2}, foi constatada uma interrupção no recebimento de tuplas para processamento. A segunda fase de execução do MobiJoin começa selecionando um *bucket* de disco para

processamento. Suponha que o *bucket* selecionado foi o *bucket* D_{A1}, que será comparado com o *bucket* de memória correspondente da relação oposta, neste caso M_{B1}. O operador consultará a tabela de controle para identificar as tuplas de D_{A1} e de M_{B1} que ainda não foram comparadas e executará as comparações atualizando a tabela de controle. A Figura 18 mostra o estado da execução do operador após o processamento da segunda fase e indica que apenas a tabela de controle foi atualizada, de forma a registrar as comparações executadas por esta fase.

Após o fim do processamento do *bucket* em disco D_{A1}, as tuplas t_{A7}, t_{A8}, t_{A9}, e t_{B8} já haviam chegado e a primeira fase é então reiniciada. Estas são as últimas tuplas recebidas e após o processamento das mesmas a execução da primeira fase é finalizada e a terceira fase inicia o seu trabalho. A Figura 19 apresenta o estado final da execução da primeira fase e através dela podemos constatar que a tupla t_{B8} não foi alocada ao *bucket* M_{B2}. Quando a tupla t_{B8} foi processada, todas as tuplas da relação A já haviam sido recebidas. Como o *bucket* correspondente da relação oposta, neste caso, M_{A2}, não possuía *bucket* em disco, podemos considerar que a tupla t_{B8} tinha que ser comparada apenas com as tuplas do *bucket* em memória M_{A2} e por isso após a comparação a tupla t_{B8} foi descartada. As tabelas, ilustradas na Figura 19, são utilizadas no processamento das tuplas que ainda não foram comparadas, conforme pseudocódigo mostrado na Figura 13.

4.5 Estimativa de Custo

Para estimar o custo de execução do operador MobiJoin, consideramos o cenário descrito na seção 4.2. Assim sendo, temos uma operação de junção entre duas tabelas não ordenadas A e B (armazenadas nos bancos de dados móveis DB_2 e DB_3 que residem respectivamente nos computadores móveis C_2 and C_3 respectivamente). Para simplificar consideramos que as tabelas A e B têm a mesma cardinalidade **c**, o mesmo tamanho de tupla e a mesma distribuição de tuplas entre as duas tabelas *hash*. A memória principal disponível em C_1 para a execução do MobiJoin tem capacidade para armazenar s tuplas. Consideramos o custo de acesso a disco, que por questão de simplicidade, é estimado através do número de tuplas

transferidas de/para o disco. O custo de execução do MobiJoin pode ser estimado, de acordo com as seguintes condições:

1. $s \geq 2c$. Neste caso, não ocorre overflow de memória. Então, nenhuma tupla é transferida para disco e a 3ª fase não é executada, sendo o custo de execução igual a ***2c*** (custo de leitura das tabelas A e B);

2. $\frac{s}{2} < c \leq s$. Neste caso, ocorre overflow de até c tuplas, desde que tenhamos assumido a mesma distribuição de tuplas entre as tabelas hash. O custo é dado por $2c + (\frac{c}{2} + \frac{c}{2}) + (\frac{c}{2} + \frac{c}{2})$, ou seja, ***4c***. Note que, um termo $(\frac{c}{2} + \frac{c}{2})$ representa o custo de transferência de tuplas para o disco e o outro representa o custo de leitura de tuplas do disco para a execução da 3ª fase;

3. $s < c$. Nesta situação, ocorre overflow de $(2c - s)$ tuplas, ou seja, um overflow de $(c - \frac{s}{2})$ tuplas para cada tabela (assumindo a distribuição uniforme de tuplas). Então, o custo estimado é $2c + (2c - s) + (2c - s)$, que resulta em $2(3c - s)$.

4.6 Análise Comparativa

Nesta seção, analisamos o MobiJoin com relação aos operadores adaptativos de junção [25,27,42] e os critérios de comparação descritos no capítulo 3. Esta análise foi resumida na tabela 3.

Conforme citado, o critério tuplas processadas assim que chegam no operador de junção é determinado pela estratégia utilizada no recebimento de tuplas pertencentes a *buckets* de memória que sofreram pelo menos um *overflow* de memória, ou seja, que já tiveram tuplas transferidas para disco. O MobiJoin, assim como o *Xjoin,* não apresenta problemas com relação a este aspecto, pois, independente da ocorrência ou não de *overflows* de memória, as novas tuplas que estão chegando continuam sendo processadas, aumentando a produção de tuplas de resultado.

Analisando o MobiJoin com relação à continuação do processamento durante o bloqueio no recebimento de dados das relações envolvidas na junção, observamos, que também como o Xjoin, o MobiJoin possibilita a continuação do processamento da consulta na ocorrência de bloqueio na entrega de tuplas de ambas as relações, processando as tuplas dos *buckets* que estão em disco até que novas tuplas cheguem na memória para processamento e mascarando o tempo de espera da entrega de tuplas.

Este processamento, provido por ambos os operadores, necessita de um mecanismo de controle adicional para evitar a produção de resultados incompletos e/ou duplicados. O mecanismo implementado pelo *XJoin* baseia-se em *timestamps* que são adicionados à estrutura das tuplas durante a execução da primeira e da segunda fase do operador. Na primeira fase é adicionado a todas as tuplas um *timestamp* que indica o momento da chegada da tupla no *bucket* em memória. Um segundo *timestamp* é adicionado à tupla quando ocorre um *overflow* e o *bucket* a qual ela pertence é transferido para o disco. Além destes dois *timestamps*, é necessário que seja criada uma lista encadeada para cada *bucket* em disco processado na segunda fase. Cada entrada nesta lista é composta por dois outros *timestamps*: (1) um para marcar quando a última tupla do *bucket* em disco foi descarregada em disco (foi transferida do *bucket* em memória para o *bucket* em disco), e; (2) o outro *timestamp* deve registrar quando a segunda fase foi executada para o *bucket* em disco. Esta lista de *timestamps* adicionais tem a funcionalidade de identificar as tuplas comparadas através da segunda fase do *XJoin*.

Como pode ser observado, o mecanismo de controle do *XJoin* requer várias estruturas de controle, que aumentam a complexidade do operador e seu consumo de memória, recurso escasso em dispositivos móveis. No MobiJoin, o mecanismo de controle para evitar a geração de resultados incompletos e/ou com duplicação de tuplas requer somente que as tuplas sejam acrescidas de um atributo, o BTID. Apenas no caso de ocorrência de transferência de *bucket* em memória para o disco, é necessária a construção de uma estrutura simples para detecção de tuplas já processadas, a tabela de controle (veja seção 2). Portanto, o mecanismo para prevenção de resultados incompletos e/ou duplicados implementado pela MobiJoin, comparado ao operador *XJoin*, utiliza uma estrutura de controle menos complexa,

simplificando o processamento computacional e diminuindo a quantidade de memória necessária para o operador implementar seus mecanismos de controle e executar a operação de junção corretamente.

Tabela 3 – Comparativo de operadores adaptativos e do MobiJoin

Características	**Adaptative Symmetric Hash Join**	**Incremental Symmetric Flush**	**Incremental Left Flush**	**Xjoin**	**MobiJoin**
As tuplas são processadas assim que chegam no operador de junção	Apenas se não tiver ocorrido *overflow* no *bucket* destino	Apenas se não tiver ocorrido *overflow* no *bucket* destino	Se tiver ocorrido *overflow* em qualquer um dos *buckets* das tabelas *hash* envolvidas, suspende o processamento das tuplas de uma das relações	Sim	Sim
Continuação do processamento durante bloqueio no recebimento de tuplas de ambas as relações	Não	Não	Não	Sim	Sim
Verificação da necessidade de armazenamento da tupla em memória	Sim	Não	Não	Não	Sim

Com relação ao terceiro critério de análise, verificação da necessidade de armazenamento da tupla em memória, o MobiJoin, assim como o Adaptative Symmetric Hash Join, possui esta característica e por isso provê uma maior otimização do uso da memória disponível. É uma outra otimização em relação ao *Xjoin*, que não possui este tipo de verificação no seu algoritmo. Suponha a execução de uma junção entre as tabelas A e B e que durante a execução da primeira fase, o MobiJoin recebe uma tupla de A. Neste caso, o MobiJoin verifica se a relação oposta (a relação B) já foi completamente recebida. Se todas as tuplas da relação B já

chegaram, a tupla de A só será alocada no *bucket* em memória se o *bucket* em memória correspondente da relação B já tenha sofrido um *overflow* de memória. Caso contrário, como a relação oposta já foi completamente recebida, podemos garantir que todas as tuplas do *bucket* correspondente já chegaram e estão armazenadas no *bucket* em memória. Portanto, as tuplas de A que chegarem só precisarão se comparadas com as tuplas do *bucket* em memória da relação B e poderão ser descartadas após a comparação. Observe que esta característica do MobiJoin minimiza o número de transferências de *buckets* de memória para disco e garante uma otimização no uso de memória disponível, recurso limitado em dispositivos móveis.

5. IMPLEMENTAÇÃO DO MOBIJOIN

5.1 Introdução

No capítulo anterior foram mostrados os conceitos e o funcionamento do algoritmo MobiJoin. Neste capítulo mostraremos a implementação do algoritmo e as simulações executadas com a implementação, cujo objetivo é demonstrar a viabilidade da proposta apresentada.

O algoritmo foi implementado por Marlus Aguiar Saraiva [38], que utilizou a linguagem de programação Java, escolhida por possuir recursos poderosos que facilitaram o trabalho de desenvolvimento do protótipo, como a fácil manipulação de *threads* (linhas de execução), a presença de um mecanismo embutido de sincronização destas linhas de execução e a implementação nativa de uma função geradora de *hash codes*.

Este capítulo possui a seguinte estrutura: a seção 2 descreve as classes e estruturas de dados utilizadas, abordando com detalhes a implementação de todas as estruturas de dados necessárias para a execução do algoritmo; e os tipos de simulação executados, a configuração do ambiente na qual as simulações foram executadas e os resultados das simulações são descritos na seção 3.

5.2 Classes e estruturas de dados

As classes que foram definidas foram estruturadas buscando uma maior flexibilidade em relação ao funcionamento do protótipo. O diagrama que contém as classes referentes ao operador MobiJoin é apresentado no Apêndice A.

A classe Row define a abstração de uma tupla de uma tabela de banco de dados. Possui métodos para definir e recuperar os valores de seus atributos, além de conter uma propriedade extremamente importante para o funcionamento do algoritmo. Esta propriedade é o BTID (identificador único de tupla do *bucket*).

A classe HashTable define a abstração da tabela *hash* que será utilizada pelo operador MobiJoin. Apesar da linguagem Java fornecer classes que implementam uma tabela *hash*, como é o caso de HashMap e Hashtable do pacote "java.util", preferimos implementar uma classe própria. Caso utilizássemos as classes do Java, não teríamos controle sobre a manipulação dos *buckets*. Além disso, as duas classes citadas possuem um mecanismo dinâmico de redimensionamento chamado *rehash*, que não se adequa às necessidades do operador MobiJoin, que trabalha com tabelas *hash* de tamanho fixo. A classe HashTable é composta por um *array* de objetos do tipo Bucket.

Os objetos do tipo Bucket, citados anteriormente, têm como papel principal armazenar as tuplas da tabela hash a qual eles pertencem. Na verdade, cada objeto do tipo Bucket contém um *array* de objetos do tipo Row. O número de tuplas que um *bucket* pode armazenar é definido no momento da criação do objeto e é imutável durante todo o tempo de vida do objeto. Outro papel importante delegado à classe Bucket é o de definir os BTIDs de suas tuplas.

A classe DiskBucket tem como função abstrair um *bucket* que será serializado (gravado em disco). Quando um *bucket* está com a sua capacidade máxima atingida e tem que ser transferido para o disco, seus dados são copiados para um objeto do tipo DiskBucket que é, efetivamente, serializado.

As classes DynHashTable e DynBucket são referentes à tabela hash utilizada na terceira fase do algoritmo. A classe DynHashTable implementa uma tabela *hash* onde a capacidade dos *buckets* é dinâmica.

A classe ControlTable implementa a tabela de controle do operador MobiJoin. Esta classe é composta, basicamente, por um *array* bidimensional de variáveis do tipo *boolean*.

As classes citadas acima formam o conjunto de classes que podemos definir como as estruturas de dados básicas para a execução do algoritmo. As classes que iremos descrever a seguir são aquelas que tratam da implementação da lógica do algoritmo.

A classe DataSet tem como função básica a conversão das tuplas oriundas de uma fonte JDBC para uma tupla do tipo Row. Este DataSet é instanciado pela classe DataProvider que tem como função buscar as tuplas de uma determinada fonte de dados. A classe DataProvider, por sua vez, herda de Thread que é a classe nativa da linguagem Java para a manipulação de linhas de execução (*threads*). Esta implementação é utilizada devido à necessidade do operador MobiJoin de receber as tuplas das relações envolvidas na junção de maneira simétrica, ou seja, paralelamente.

A classe MobiJoin contém praticamente toda a lógica do algoritmo. Através dela são executadas todas as fases do operador. Por conseqüência, ela instancia as duas tabelas *hash* necessárias para a execução do algoritmo, as tabelas de controle e a tabela *hash* dinâmica utilizada na terceira fase. As duas tabelas *hash* do tipo HashTable são definidas como as propriedades “hashTableA” e “hashTableB”. As tabelas de controle são armazenadas em um *array* de objetos do tipo ControlTable denominado controlTables.

A classe MobiJoin, que herda de Thread, recebe como parâmetros dois DataProviders que irão fornecer as tuplas necessárias para a execução da junção. Toda a comunicação entre os DataProviders e o MobiJoin é realizada utilizando o padrão Observer [24]. Para isso a classe MobiJoin implementa a interface ProviderListener e se adiciona como “observador” dos eventos gerados pelos DataProviders. O DataProvider comunica a todos os seus “observadores” eventos como recebimento de tuplas, bloqueio na recepção, evento de EOF (fim das tuplas), entre outros. Este tipo de implementação resulta em classes mais independentes e sem acoplamento entre si. A própria classe MobiJoin também define uma interface de comunicação para a utilização deste padrão, a MobiJoinListener. A aplicação que instancia um objeto do tipo MobiJoin pode implementar esta interface e conseqüentemente receber notificações dos eventos gerados pelo MobiJoin. É desta maneira que, por exemplo, uma aplicação pode mostrar o resultado da junção ou

gerar *logs,* sem que seja necessário alterar a implementação das classes DataProvider e MobiJoin.

Maiores detalhes da implementação podem ser obtidos em [38]. O código-fonte do algoritmo é apresentado no Apêndice A.

5.3 Simulações

As simulações foram executadas em uma estação Pentium IV com 512 MB de memória RAM e com o sistema operacional Windows XP. As duas tabelas utilizadas nas simulações da execução do operador, foram armazenadas no sistema gerenciador de banco de dados Microsoft SQL Server 8.0, instalado em um servidor com a seguinte configuração: Pentium IV com 4 processadores e 4 GB de memória RAM. As tabelas utilizadas como fonte de dados para a execução do operador possuíam 5,37 MB e 5,29 MB de tamanho, e armazenavam tuplas de mesmo tamanho (115 bytes), a fim de otimizar o uso da memória, uma vez que a implementação do algoritmo considera apenas uma única distribuição de tuplas para as tabelas *hash* criadas durante a execução do operador.

O primeiro tipo de simulação consistiu na execução do MobiJoin com ou sem o disparo da 2ª fase de execução e em ambientes com diferentes quantidades de memória (16MB e 4MB). Estas simulações foram executadas com bloqueios de 5 segundos no recebimento das tuplas de cada relação a cada 5.000 tuplas recebidas, a fim de simular um ambiente com atrasos no recebimento de tuplas, comuns em MDBCs, e que podem aumentar substancialmente o tempo de resposta no acesso aos bancos de dados móveis. Este tipo de simulação tem como objetivo mostrar que em ambientes com limitação de memória a execução da segunda fase do operador realmente faz com que o operador produza tuplas de resultado mais cedo.

Quando executamos o operador em um ambiente simulado de 16 MB de memória, apenas uma pequena diferença entre a execução do operador com a 2ª fase e sem a 2ª fase pode ser notada, conforme mostra o gráfico 1. Este resultado é

justificado pelo tamanho das relações envolvidas na operação de junção, uma vez que ambas cabem completamente na memória, não havendo necessidade de transferência de tuplas para disco. Portanto, mesmo que bloqueios no recebimento de tuplas de ambas as relações ocorram e a execução da 2ª fase seja disparada, não haverá tuplas em disco para serem processadas por ela, e conseqüentemente nenhuma tupla será adicionada ao resultado nos momentos de bloqueio no recebimento de ambas as relações.

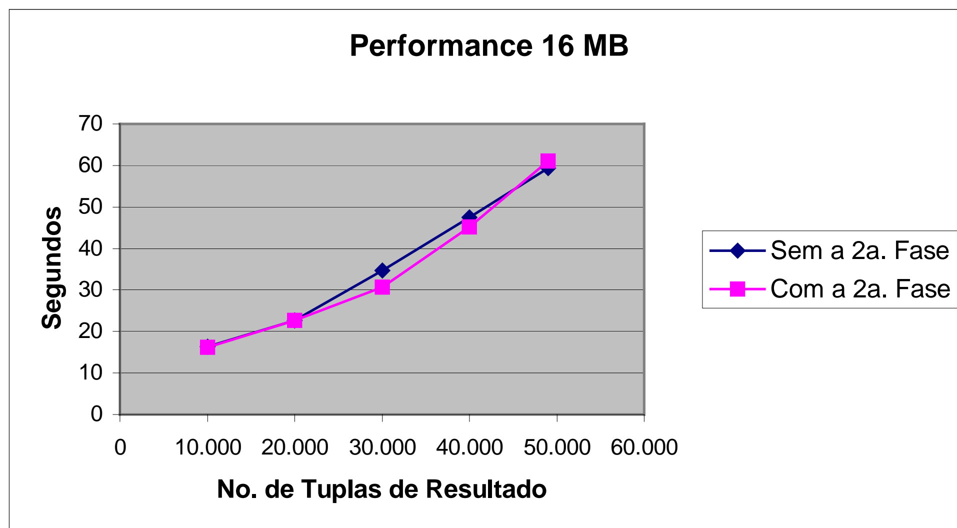


Gráfico 1 – Performance da execução do operador com 16 MB

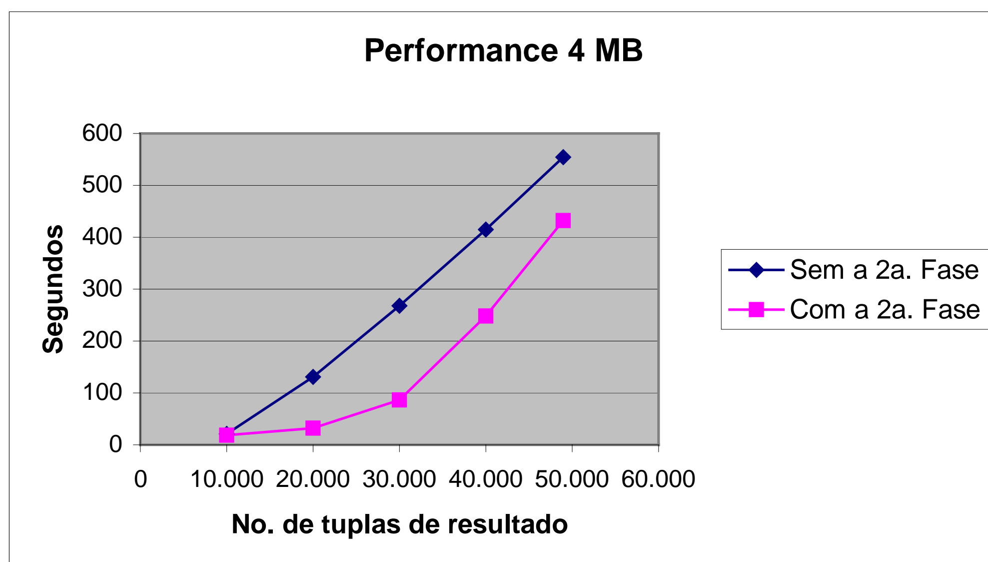


Gráfico 2 – Performance da execução do operador com 4MB

Quando executamos o operador em um ambiente simulado de 4 MB de memória, as relações passam a não caber na memória, sendo necessárias transferências de tuplas para disco. Assim sendo, durante os bloqueios no recebimento de tuplas de ambas as relações, as tuplas já transferidas para disco podem ser processadas, aumentando a quantidade de tuplas de resultado mais cedo. O gráfico 2 mostra a diferença na produção de tuplas de resultado entre a execução do operador com e sem a 2ª fase. Podemos claramente perceber que mais tuplas são produzidas mais cedo. Além disso, o tempo de execução total da consulta também é menor.

O segundo tipo de simulação mostra como o atraso na entrega dos dados e as variações do número de tuplas recebidas impactam a performance do Mobijoin. Mantemos o tamanho da memória constante, 4 MB, e sempre executamos a 2ª fase do operador. Após esta simulação percebemos que quando pequenas quantidades de tuplas (1.000 tuplas) chegam com grandes atrasos (15 segundos), a performance na geração de tuplas de resultado não é muito degradada devido aos atrasos na entrega das tuplas, uma vez que às execuções da 2ª fase fazem com que tuplas de resultado sejam produzidas enquanto o operador aguarda a chegada de novas tuplas.

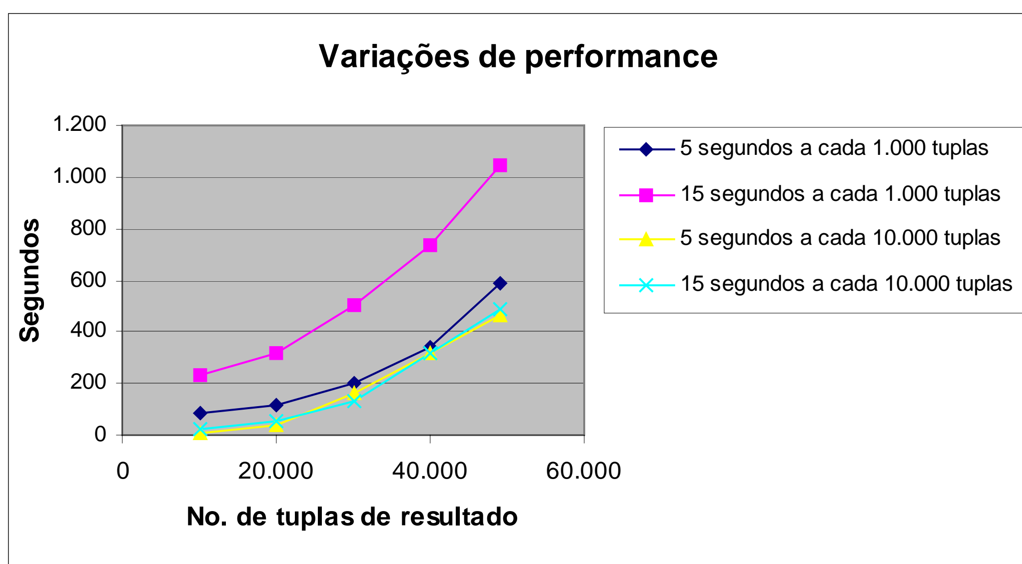


Gráfico 3 - Variações de performance do MobiJoin

Observamos também que a quantidade de tuplas recebidas entre os atrasos na entrega de dados tem grande impacto sobre a performance do operador. Quando grandes quantidades de tuplas, neste caso 10.000 tuplas, são recebidas pelo operador, as tuplas de resultado são geradas mais rapidamente, e a performance do operador é excelente. Isto é muito importante, pois mostra que a complexidade adicionada ao MobiJoin não causa impacto na performance do operador, quando o mesmo é executado com altas taxas de entrega de tuplas. Desta forma, podemos concluir que não estamos sacrificando a performance do operador para que o mesmo possa produzir tuplas de resultado durante os períodos de atraso na entrega das tuplas.

CONCLUSÃO

Nesta dissertação, foram identificados problemas que surgem no processamento de consultas sobre bancos de dados móveis. A imprevisibilidade na taxa de entrega dos dados aos operadores no processamento da consulta e as limitações de memória para processar os operadores podem provocar degradação de desempenho no acesso a dados em comunidade de bancos de dados móveis (MDBCs), caso sejam empregadas técnicas tradicionais de processamento de consultas.

Com o objetivo de minimizar o impacto da mobilidade no processamento das consultas aos bancos de dados móveis participantes de uma Comunidade de Bancos de Dados Móveis, e prover uma adaptação dinâmica do plano de execução das consultas em resposta a problemas relacionados à mobilidade, apresentamos nesta dissertação um operador de junção chamado MobiJoin. O MobiJoin comporta-se como um operador de consulta adaptativo, reagindo a eventos como o atraso na entrega de tuplas, comuns em Comunidades de Bancos de Dados Móveis, e que podem aumentar substancialmente o tempo de resposta no acesso a bancos de dados móveis.

O MobiJoin está baseado em três princípios de funcionamento: *(i)* produção incremental de resultados à medida que os dados são disponibilizados, *(ii)* continuidade no processamento da consulta mesmo que a entrega dos dados esteja bloqueada, e *(iii)* reação a situações de limitação de memória durante a execução do operador.

A realização deste trabalho de dissertação envolveu uma investigação bibliográfica sobre processamento adaptativo, e uma análise dos operadores de junção adaptativos já propostos para ambientes distribuídos, segundo critérios definidos no capítulo 3 e apresentados através de análise comparativa entre os operadores adaptativos estudados e o operador proposto.

Constatamos que com relação ao critério "tuplas processadas assim que chegam no operador de junção", o MobiJoin não apresenta nenhuma restrição, pois, independente da ocorrência ou não de *overflows* de memória, as novas tuplas que estão chegando continuam sendo processadas, aumentando assim a produção de tuplas de resultado pelo operador.

Analisando o MobiJoin com relação à "continuação do processamento durante o bloqueio no recebimento de dados das relações envolvidas na junção", observamos também que o MobiJoin possibilita a continuação do processamento da consulta na ocorrência de bloqueio na entrega de tuplas de ambas as relações, processando as tuplas dos *buckets* que estão em disco até que novas tuplas cheguem na memória para processamento, e mascarando o tempo de espera da entrega de tuplas. Além disso, apresenta um mecanismo de controle para evitar a geração de resultados incompletos e/ou com duplicação de tuplas que requer somente que as tuplas sejam acrescidas de um atributo, o BTID, e apenas no caso de ocorrência de transferência de *bucket* em memória para o disco, é necessária a construção de uma estrutura simples para detecção de tuplas já processadas, a tabela de controle. Desta forma, o processamento computacional é simplificado, diminuindo a quantidade de memória necessária para o operador implementar seus mecanismos de controle e executar a operação de junção corretamente.

Destacamos ainda que o MobiJoin verifica antes de gravar uma tupla no bucket em memória a necessidade de gravação da mesma, verificação que minimiza o número de transferências de *buckets* de memória para disco e garante uma otimização no uso de memória disponível, recurso limitado em dispositivos móveis.

Para demonstrar a viabilidade do operador MobiJoin, foi desenvolvido um protótipo que permite a execução do operador em um ambiente simulado de computação móvel, onde estão previstas situações como limitação de memória e bloqueio e atraso no recebimento dos dados. As simulações serviram de base para uma análise da adaptabilidade deste operador e futuramente também serão utilizadas para testes comparativos com outros operadores propostos.

Os resultados das simulações mostram claramente que a 2ª fase do operador faz com que mais tuplas sejam produzidas mais cedo em ambientes com atrasos na

entrega de tuplas, e o mais importante, sem que a performance do operador seja sacrificada quando estes atrasos não ocorrem. Assim sendo, as simulações confirmaram a adaptabilidade do operador, e mostraram a eficiência do mesmo para ambientes com limitação de memória e atrasos na entrega de tuplas.

Durante a implementação do algoritmo e simulações da execução do operador uma nova otimização foi identificada. A adoção desta otimização terá como objetivo a obtenção um melhor desempenho do operador na presença de grandes quantidades de tuplas. Desta forma, foi definido como trabalho futuro a otimização do algoritmo da 3ª fase, principalmente com relação à utilização de um índice *hash* para comparação das tuplas em disco com as tuplas em memória. Durante as simulações, observamos que problemas referentes à limitação de memória podem ocorrer na execução da 3ª fase caso um bucket em disco possua grande quantidade de tuplas, uma vez que estas tuplas devem ser alocadas em memória para que as devidas comparações sejam feitas.

REFERÊNCIAS

1. ABDALLAH, M., GUERRAOUI, R. and PUCHERAL, P. **Dictatorial Transaction Processing: Atomic Commitment without Veto Right**. Journal of Distributed and Parallel Databases. Vol. 11, No. 3. March, 2002. Pages 239 - 268.

2. AGRAWAL, P. and FAMOLARI, D. **Mobile Computing in Next Generation Wireless Networks**. Proceedings of 3rd International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications. Seattle, WA. August 1999. Pages 32 – 39.

3. ALONSO, R. and KORTH, H. F. **Database system issues in nomadic computing**. Proceedings of the ACM SIGMOD International Conference on Management of Data. Washington, D.C. May 1993. Pages 388 - 392.

4. AMSALEG, L., FRANKLIN, M.J. and URHAN, T. **Cost Based Query Scrambling For Initial Delays**. Proceedings of the ACM SIGMOD International Conference On Management of Data. Seattle, WA. June 1998. Pages 130 -141.

5. AMSALEG, L., FRANKLIN, M.J., TOMASIC, A. and URHAN, T. **Scrambling Query Plans to Cope with Unexpected Delays**. Proceedings of the 4th International Conference on Parallel and Distributed Information Systems (PDIS). Miami Beach, Florida. Deceber 1996. Pages 208 – 219.

6. BOUGANIM, L., FABRET, F., MOHAN, C. And VALDURIEZ, P. **A Dynamic Query Processing Architecture for Data Integration Systems**. IEEE Data Engineering Bulletin, Vol. 23 No. 2. June 2000. Pages 42 - 48.

7. BOUGANIM, L., FABRET, F., MOHAN, C. And VALDURIEZ, P. **A Dynamic Query Scheduling in Data Integration Systems**. Proceedings of the 16th

International Conference on Data Engineering. Washington, D.C. March 2000. Page 425.

8. BOUGANIM, L., KAPITSKAIA, O. and VALDURIEZ, P. **Memory-Adaptative Scheduling For Large Query Execution**. Proceedings of 7th International Conference on Information and Knowledge Management. Bethesda, Maryland. November 1998. Pages 105 – 115.

9. BRASCHE, G. and WALKE, B. **Concepts, Services and Protocols of the New GSM Phase 2+ General Packet Radio Service**. IEEE Comunications Magazine. August 1997. Pages 94 – 104.

10. BRAYNER, A. and CAMPOS, E. **MobiJoin: Um Operador de Junção para Bancos de Dados Móveis**. VI Workshop de Comunicação Sem Fio e Computação Móvel. Fortaleza, Ceará. Outubro 2004.

11. BRAYNER, A. and M. FILHO, José A. **AMDB: An Approach for Sharing Mobile Databases in Dynamically Configuration Environments**. Proceedings of the 17th Brazilian Symposium on Databases (SBBD). Gramado, Rio Grande do Sul. October 2002. Pages 12 – 26.

12. BRAYNER, A. and M. FILHO, José A. **Increasing Mobile Transaction Concurrency in Dynamically Configurable Environments**. Proceedings of the 35rd IEEE Workshop on Mobile Distributed Computing (MDC). Columbus, Ohio. June 2005.

13. BRAYNER, A. and M. FILHO, José A. **Sharing Mobile Databases in Dynamically Configurable Environments**. Proceedings of 15th International Conference of Advanced Information Systems Engineering. Klagenfurt, Austria. June 2003. Pages 724 – 737.

14. DEMERS, K. Petersen et al. **The Bayou Architecture: Support for Data Sharing among Mobile Users**. Proceedings of the IEEE Workshop on Mobile

Computing Systems and Applications. Santa Cruz, California. December 1994. Pages 2 – 7.

15. DEWITT, D.J. and KABRA, N. **Efficient Mid-Query Re-Optimization of Sub-Otimal Query Execution Plans**. Proceedings of ACM SIGMOD Conference on Management of Data. Seattle, WA. June 1998. Pages 106 – 117.

16. DEWITT, D.J. and SCHNEIDER, D.A. **A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment**. Proceedings of ACM SIGMOD Conference on Management of Data. Portland, Oregon. May 1989. Pages 110 – 121.

17. DU, W., SHAN, M. and DAYAL, U. **Reducing Multidatabase Query Response Time By Tree Balancing**. Proceedings of ACM SIGMOD Conference on Management of Data. San Jose, California. May 1995. Pages 293 – 303.

18. DUNHAM, M.H., HELAL, A., and BALAKRISHNAN, S. **A Mobile Transaction Model That Captures Both The Data And Movement Behavior**. Journal on Special Topics in Mobile Networks and Applications, Vol.2, No. 2. October 1997. Pages 149 – 162.

19. DUNHAM, M. H. and HELAL, S. **Mobile Computing And Databases: Anything New?**. SIGMOD Record, Vol. 24, No. 4. December 1995. Pages 5 – 9.

20. DUNHAM, M. H. and KUMAR, V. **Impact of Mobility on Transaction Management**. Proceedings of the ACM International Workshop on Data Engineering for Wireless and Mobile Access. Seattle, WA. August 1999. Pages 14 – 21.

21. DUNHAM, M. H. and REN, Q. **Using semantic caching to manage location dependent data in mobile computing**. Proceedings of 6th Annual International Conference on Mobile Computing and Networking (MobiCom'00). Boston, Massachusetts. August 2000. Pages 210 – 221.

22. ELMAGARMID, A., BOUGUETTAYA, A. and BENATALLAH, B. **Interconnecting Heterogeneous Information Systems**. Kluwer Academic Publishers. 1998.

23. ELMAGARMID, A., JING, J. and HELAL, A. **Client-Server Computing in Mobile Environments**. ACM Computing Surveys, Vol. 31, No. 2. June 1999. Pages 117 – 157.

24. GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-oriented Software**. Addison-Wesley. 1995.

25. FRANKLIN, M. J. and URHAN, T. **X-Join: A Reactively Scheduled-Pipelined Join Operator**. IEEE Data Engineering Bulletin. Vol. 23 No. 2. June 2000. Pages 27 – 33.

26. HAAS, P. J. and HELLERSTEIN, J. M. **Ripple Joins for Online Aggregation**. Proceedings of ACM SIGMOD Conference on Management of Data. Philadelphia, Pennsylvania. June 1999. Pages 287 – 298.

27. HAAS, Peter J. et al. **A Scalable Hash Ripple Join Algorithm**. Proceedings of the ACM SIGMOD International Conference on Management of Data. June 2002. Madison, Wisconsin. Pages 252 – 262.

28. HELLERSTEIN, J. et al. **Adaptative Query Processing: Technology in Evolution**. Bulletin of the IEEE Computer Society Technical Commitee on Data Engineering, Vol. 23, No.2. June 2000. Pages 7 – 18.

29. HONG, W. and STONEBRAKER, M. **Optimization of Parallel Query Execution Plans in XPRS**. Proceedings of 1st International Conference on Parallel and Distributed Information Systems. Miami, Florida. December 1991. Pages 218 - 225.

30. IMIELINKSI, T. and BADRINATH, B. R. **Data Management for Mobile Computing**. SIGMOD Record, Vol. 22, No.1. March 1993. Pages 34 – 39.

31. JOSEPH, D., TAUBER, J. A. and KAASHOEK, M. F. **Mobile Computing with the Rover Toolkit**. IEEE Transactions on Computers, Vol. 46, No. 3. March 1997. Pages 337 – 352.

32. MACKER, J. P. and CORSON, M. S. **Mobile Ad Hoc Networking and the IETF**. ACM SIGMOBILE Mobile Computing and Communications Review, Vol.6, No. 2. April 2002. Pages 1 – 2.

33. M. FILHO, José A. **AMDB – An Approach for Sharing Mobile Databases**. Dissertação de Mestrado, Universidade de Fortaleza, 2003.

34. NOBLE, B. and SATYANARAYANAN, M. **A Research Status Report on Adaptation for Mobile Data Access**. SIGMOD Record, Vol. 24, No. 4. December 1995. Pages 10 – 15.

35. ROMAN, G., PICCO, G. P. and MURPHY, A. L. **Lime: A Middleware for Physical and Logical Mobility**. Proceedings of the 21st International Conference on Distributed Computing Systems. Phoenix, Arizona. April 2001. Pages 524 – 533.

36. ROMAN, G., PICCO, G. P. and MURPHY, A. L. **Lime : Linda Meets Mobility**. Proceedings of the 21st International Conference on Software Engineering. Los Angeles, California. May 1999. Pages 368 – 377.

37. ROMAN, G., PICCO, G. P. and MURPHY, A. L. **Software Engineering for Mobility: A Roadmap**. Proceedings of the Conference on The Future of Software Engineering. Limerick, Ireland. June 2000. Pages 241 – 258.

38. SARAIVA, M. **MobiJoin: Algoritmo de Junção para Bancos de Dados Móveis**. Monografia. Universidade de Fortaleza, 2004.

39. Sun MicroSystems. **Java Database Connection API**. http://java.sun.com.

40. URHAN, T. and FRANKLIN, M. J. **Dynamic Pipeline Scheduling for Improving Interactive Query Performance**. Proceedings of 27th Very Large Database Conference. Rome, Italy. September 2001. Pages 501 – 510.

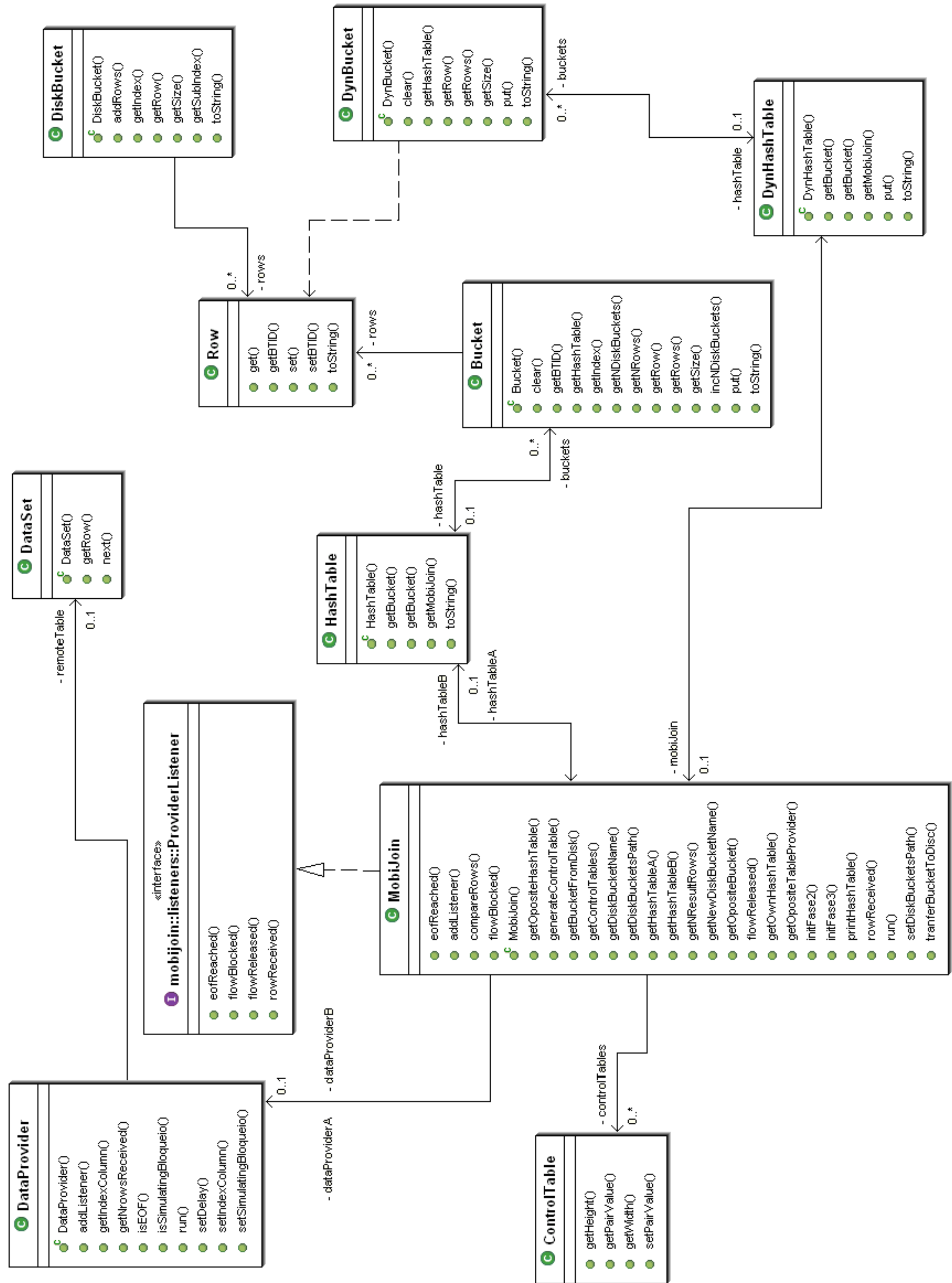
41. VLACH R., LANA J., MAREK J. and NAVARA D. **MDBAS – A Prototype of a Multidatabase Management System Based on Mobile Agents**. Proceedings of SOFSEM. Springer, Verlag. November 2000. Pages 440 – 449.

42. WELD, D. S. et al. **An Adaptive Query Execution System for Data Integration**. Proceedings of ACM SIGMOD International Conference on Management of Data. Philadelphia, Pennsylvania. June 1999. Pages 299 – 310.

43. WHITE S., CATTELL R. And FINKELSTEIN S. **Enterprise Java Platform Data Access**. Proceedings of ACM SIGMOD International Conference on Management of Data. Seattle, WA. June 1998. Pages 504 – 505.

44. WILSCHUT, A. N. and APERS, P. M. G. **Dataflow Query Execution in a Parallel Main-Memory Environment**. Proceedings of 1st Conference on Parallel and Distributed Information Systems. Miami Beach, Florida. December 1991. Pages 68 – 77.

Apêndice A – Diagrama de Classes do MobiJoin



Apêndice B – Código fonte em Java da Implementação do MobiJoin

A.1 A classe Bucket

```
package mobijoin;

public class Bucket
{
    private int size;
    private Row[] rows;
    private int BTID = 0;
    private int nRows = 0;
    private HashTable hashTable;
    private int index;
    private int nDiskBuckets = 0;

    public Bucket(int size, HashTable hashTable, int bucketIndex)
    {
        this.hashTable = hashTable;
        this.size = size;
        this.index = bucketIndex;
        rows = new Row[size];
    }

    public synchronized void put(Row row)
    {
        BTID++;
        row.setBTID(BTID);
        rows[nRows] = row;
        nRows++;
    }

    public synchronized Row[] getRows()
    {
        return (Row[]) rows.clone();
    }

    public synchronized void clear()
    {
        for (int i = 0; i < rows.length; i++)
        {
            rows[i] = null;
        }
        nRows = 0;
    }

    public String toString()
```

```
    {
        StringBuffer buf = new StringBuffer();

        for (int i = 0; i < rows.length; i++)
        {
            buf.append("        ");
            buf.append(rows[i]);
            buf.append("\n");
        }
        return buf.toString();
    }

    public int getIndex()
    {
        return index;
    }

    public synchronized Row getRow(int index)
    {
        return rows[index];
    }

    public int getSize()
    {
        return size;
    }

    public HashTable getHashTable()
    {
        return hashTable;
    }

    public synchronized int getNRows()
    {
        return nRows;
    }

    public synchronized int getBTID()
    {
        return BTID;
    }

    public synchronized int getNDiskBuckets()
    {
        return nDiskBuckets;
    }

    public synchronized void incNDiskBuckets()
    {
        nDiskBuckets++;
    }
}
```

A.2 A classe ControlTable

```
package mobijoin;

public class ControlTable
```

```
{
    private boolean table[][] = new boolean[][]{};
    private int height = 0;
    private int width = 0;

    public void setPairValue(int btidA, int btidB, boolean value)
    {
        if ((btidA > height) || (btidB > width))
        {
            resizeTable(Math.max(btidA, height), Math.max(btidB, width));
        }
        table[btidA-1][btidB-1] = value;
    }

    private void resizeTable(int height, int width)
    {
        boolean newTable[][] = new boolean[height][width];
        for (int i = 0; i < table.length; i++)
        {
            for (int j = 0; j < table[i].length; j++)
            {
                newTable[i][j] = table[i][j];
            }
        }
        table = newTable;
        this.height = height;
        this.width = width;
    }

    public boolean getPairValue(int btidA, int btidB)
    {
        return table[btidA-1][btidB-1];
    }

    public int getHeight()
    {
        return height;
    }

    public int getWidth()
    {
        return width;
    }
}
```

A.3 A classe DataProvider

```
package mobijoin;

import java.sql.Connection;
import java.sql.SQLException;
import java.util.ArrayList;

import mobijoin.listeners.ProviderListener;

public class DataProvider extends Thread
```

```
{
    private DataSet remoteTable;
    private String indexColumn;
    private boolean simulatingBloqueio = false;
    private int delay = 0;
    private boolean EOF = false;
    private int nRowsReceived = 0;
    private ArrayList providerListeners = new ArrayList();

    public DataProvider(Connection conn, String tableName) throws
SQLException
    {
        this.remoteTable = new DataSet(conn, tableName);
    }

    public void run()
    {
        nRowsReceived = 0;
        try
        {
            Row row = null;
            while (remoteTable.next())
            {
                if (simulatingBloqueio)
                {
                    // Esta notificação é feita aqui para garantir que
                    // a notificação de bloqueio venha depois da comparação
                    notifyFlowBlock();
                    while (simulatingBloqueio)
                    {
                        try
                        {
                            sleep(100);
                        } catch (InterruptedException e1)
                        {
                            e1.printStackTrace();
                        }
                    }
                }
                simulateDelay();
                row = remoteTable.getRow();
                nRowsReceived++;
                notifyRowReceived(row);
            }
            EOF = true;
            notifyEofReached();

        } catch (SQLException e)
        {
            e.printStackTrace();
        }
    }

    private void simulateDelay()
    {
        if (delay > 0)
        {
            try
            {
                sleep(delay);
            }
```

```
            catch (InterruptedException e1)
            {
                e1.printStackTrace();
            }
        }
    }

    public boolean isEOF()
    {
        return EOF;
    }

    public String getIndexColumn()
    {
        return indexColumn;
    }

    public void setDelay(int delay)
    {
        this.delay = delay;
    }

    public void addListener(ProviderListener listener)
    {
        providerListeners.add(listener);
    }

    private void notifyRowReceived(Row row)
    {
        for (int i = 0; i < providerListeners.size(); i++)
        {
            ProviderListener listener = (ProviderListener)
providerListeners.get(i);
            listener.rowReceived(row, this);
        }
    }

    private void notifyEofReached()
    {
        for (int i = 0; i < providerListeners.size(); i++)
        {
            ProviderListener listener = (ProviderListener)
providerListeners.get(i);
            listener.eofReached(this);
        }
    }

    private void notifyFlowBlock()
    {
        for (int i = 0; i < providerListeners.size(); i++)
        {
            ProviderListener listener = (ProviderListener)
providerListeners.get(i);
            listener.flowBlocked(this);
        }
    }

    private void notifyFlowRelease()
    {
        for (int i = 0; i < providerListeners.size(); i++)
        {
```

```
            ProviderListener listener = (ProviderListener)
providerListeners.get(i);
            listener.flowReleased(this);
        }
    }

    public void setIndexColumn(String indexColumn)
    {
        this.indexColumn = indexColumn;
    }

    public boolean isSimulatingBloqueio()
    {
        return simulatingBloqueio;
    }

    public void setSimulatingBloqueio(boolean simulatingBloqueio)
    {
        this.simulatingBloqueio = simulatingBloqueio;
        if (simulatingBloqueio)
            ; // notifyFlowBlock() é chamado no método run
        else
            notifyFlowRelease();
    }

    public int getNrowsReceived()
    {
        return nRowsReceived;
    }
}
```

A.4 A classe DataSet

```
package mobijoin;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

public class DataSet
{
    private ResultSet rs;
    private Connection conn = null;

    public DataSet(Connection conn, String tableName) throws SQLException
    {
        rs = conn.createStatement().executeQuery("SELECT * from " +
tableName);
    }

    public boolean next() throws SQLException
    {
        return rs.next();
    }
```

```
    public Row getRow() throws SQLException
    {
        Row row = new Row();
        ResultSetMetaData rsmd = rs.getMetaData();
        String columnName;
        for (int i = 1; i <= rsmd.getColumnCount(); i++)
        {
            columnName = rsmd.getColumnName(i);
            row.set(columnName, rs.getObject(columnName));
        }
        return row;
    }
}
```

A.5 A classe DiskBucket

```
package mobijoin;

import java.io.Serializable;

public class DiskBucket implements Serializable
{
    private Row[] rows = new Row[0];
    private int index;
    private int subIndex;

    public DiskBucket(int index, int subIndex)
    {
        this.index = index;
        this.subIndex = subIndex;
    }
    public void addRows(Bucket bucket)
    {
        Row[] newRows = new Row[rows.length + bucket.getSize()];
        int i = 0;
        for (; i < rows.length; i++)
        {
            newRows[i] = rows[i];
        }
        for (int j = 0; j < bucket.getSize(); j++,i++)
        {
            newRows[i] = bucket.getRow(j);
        }
        rows = newRows;
    }

    public String toString()
    {
        StringBuffer buf = new StringBuffer();

        for (int i = 0; i < rows.length; i++)
        {
            buf.append("\t");
            buf.append(rows[i]);
            buf.append("\n");
        }
```

```
        return buf.toString();
    }

    public int getSize()
    {
        return rows.length;
    }

    public int getIndex()
    {
        return index;
    }

    public int getSubIndex()
    {
        return subIndex;
    }

    public Row getRow(int rowIndex)
    {
        return rows[rowIndex];
    }
}
```

A.6 A classe DynBucket

```
package mobijoin;

import java.util.ArrayList;

public class DynBucket
{
    private ArrayList rows;
    private DynHashTable hashTable;

    public DynBucket(DynHashTable hashTable)
    {
        this.hashTable = hashTable;
        rows = new ArrayList();
    }

    public synchronized void put(Row row)
    {
        rows.add(row);
    }

    public synchronized Row[] getRows()
    {
        return (Row[]) rows.clone();
    }

    public synchronized void clear()
    {
        rows.clear();
    }

    public String toString()
```

```
    {
        StringBuffer buf = new StringBuffer();

        for (int i = 0; i < rows.size(); i++)
        {
            buf.append("        ");
            buf.append(rows.get(i));
            buf.append("\n");
        }
        return buf.toString();
    }

    public synchronized Row getRow(int index)
    {
        return (Row) rows.get(index);
    }

    public int getSize()
    {
        return rows.size();
    }

    public DynHashTable getHashTable()
    {
        return hashTable;
    }
}
```

A.7 A classe DynHashTable

```
package mobijoin;

import java.util.Hashtable;

public class DynHashTable
{
    private Hashtable buckets;

    public DynHashTable(MobiJoin mobiJoin)
    {
        initBuckets();
    }

    private void initBuckets()
    {
        buckets = new Hashtable();
    }

    public synchronized void put(Row row, String indexColumn)
    {
      int hash = row.get(indexColumn).hashCode();
      getBucket(row, indexColumn).put(row);

    }
```

```
    public synchronized DynBucket getBucket(Row row, String indexColumn)
    {
        if (row.get(indexColumn) == null)
            return null;
      int hash = row.get(indexColumn).hashCode();
      return getBucket(new Integer(hash));
    }

    public synchronized DynBucket getBucket(Integer hash)
    {
        DynBucket bucket = (DynBucket) buckets.get(hash);
        if ( bucket == null )
        {
            bucket = new DynBucket(this);
            buckets.put(hash, bucket);
        }
        return bucket;
    }
}
```

A.8 A classe HashTable

```
package mobijoin;

public class HashTable
{
    private int nBuckets;
    private int bucketSize;
    private Bucket[] buckets;
    private MobiJoin mobiJoin;

    public HashTable(int nBuckets, int bucketSize, MobiJoin mobiJoin)
    {
        this.nBuckets = nBuckets;
        this.bucketSize = bucketSize;
        this.mobiJoin = mobiJoin;
        initBuckets();
    }

    private void initBuckets()
    {
        buckets = new Bucket[nBuckets];
        for (int i = 0; i < buckets.length; i++)
        {
            buckets[i] = new Bucket(bucketSize, this, i);
        }
    }

    public synchronized Bucket getBucket(Row row, String indexColumn)
    {
        if (row.get(indexColumn) == null)
            return null;
      int hash = row.get(indexColumn).hashCode();
      int index = (hash & 0x7FFFFFFF) % nBuckets;
        return buckets[index];
    }
```

```
    public synchronized Bucket getBucket(int index)
    {
        return buckets[index];
    }

    public String toString()
    {
        StringBuffer buf = new StringBuffer();
        for (int i = 0; i < buckets.length; i++)
        {
            buf.append("    Bucket ");
            buf.append(i);
            buf.append("\n");
            buf.append(buckets[i]);
            buf.append("\n");
        }
        return buf.toString();
    }

    public MobiJoin getMobiJoin()
    {
        return mobiJoin;
    }
}
```

A.9 A classe MobiJoin

```
package mobijoin;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;

import util.Log;

import mobijoin.listeners.MobiJoinListener;
import mobijoin.listeners.ProviderListener;

public class MobiJoin extends Thread implements ProviderListener
{
    private Log log = new Log("MobiJoin.log");
    private DataProvider dataProviderA;
    private DataProvider dataProviderB;
    private HashTable hashTableA;
    private HashTable hashTableB;
    private int nBucketsPerTable;
    private int nRowsPerBucket;
    private String diskBucketsPath = "c:/temp/";
    private ControlTable[] controlTables;
    private ArrayList mobiJoinListeners = new ArrayList();
    private int currentPhase = 0;
    private int nResultRows = 0;
```

```
    private boolean isProviderABlocked = false;
    private boolean isProviderBBlocked = false;
    private boolean done = false;

    public MobiJoin(DataProvider providerA, String indexColumnA,
DataProvider providerB, String indexColumnB, int nBucketsPerTable, int
nRowsPerBucket)
    {
        super();

        providerA.setIndexColumn(indexColumnA);
        providerA.addListener(this);
        this.dataProviderA = providerA;

        providerB.setIndexColumn(indexColumnB);
        providerB.addListener(this);
        this.dataProviderB = providerB;

        this.nBucketsPerTable = nBucketsPerTable;
        this.nRowsPerBucket = nRowsPerBucket;
        this.controlTables = new ControlTable[nBucketsPerTable];
    }

    public void run()
    {
        nResultRows = 0;
        notifyStarted();
        notifyFaseStarted(1);
        this.hashTableA = new HashTable(nBucketsPerTable, nRowsPerBucket,
this);
        this.hashTableB = new HashTable(nBucketsPerTable, nRowsPerBucket,
this);
        dataProviderA.start();
        dataProviderB.start();
    }

    public HashTable getOwnHashTable(DataProvider tableProvider)
    {
        if (tableProvider == dataProviderA)
            return hashTableA;
        else if (tableProvider == dataProviderB)
            return hashTableB;
        else
            return null;
    }

    public HashTable getOpositeHashTable(DataProvider tableProvider)
    {
        if (tableProvider == dataProviderA)
            return hashTableB;
        else if (tableProvider == dataProviderB)
            return hashTableA;
        else
            return null;
    }

    public DataProvider getOpositeTableProvider(DataProvider tableProvider)
    {
        if (tableProvider == dataProviderA)
            return dataProviderB;
        else
```

```
            return dataProviderA;
    }

    public HashTable getHashTableA()
    {
        return hashTableA;
    }

    public HashTable getHashTableB()
    {
        return hashTableB;
    }

    public synchronized void generateControlTable(Bucket bucket)
    {
        ControlTable controlTable = controlTables[bucket.getIndex()];

        if (controlTable == null)
        {
            controlTable = new ControlTable();
            Row[] rowsA;
            Row[] rowsB;
            if (bucket.getHashTable() == hashTableA)
            {
                rowsA = bucket.getRows();
                rowsB = getOpositeBucket(bucket).getRows();
            } else
            {
                rowsA = getOpositeBucket(bucket).getRows();
                rowsB = bucket.getRows();
            }
            for (int i = 0; i < rowsA.length; i++)
            {
                for (int j = 0; j < rowsB.length; j++)
                {
                    if ((rowsA[i] != null) && (rowsB[j] != null))
                        controlTable.setPairValue(rowsA[i].getBTID(),
rowsB[j].getBTID(), true);
                }
            }
        }
        controlTables[bucket.getIndex()] = controlTable;
    }

    public synchronized Bucket getOpositeBucket(Bucket bucket)
    {
        if (bucket.getHashTable() == hashTableA)
            return hashTableB.getBucket(bucket.getIndex());
        else
            return hashTableA.getBucket(bucket.getIndex());
    }

    public String getDiskBucketsPath()
    {
        return diskBucketsPath;
    }

    public String getNewDiskBucketName(Bucket bucket)
    {
        String tableAlias = ((bucket.getHashTable() ==
bucket.getHashTable().getMobiJoin().getHashTableA()) ? "A" : "B");
```

```
        return bucket.getHashTable().getMobiJoin().getDiskBucketsPath() +
"bucket" + tableAlias + "_" + bucket.getIndex() + "_" +
bucket.getNDiskBuckets();
    }

    public String getDiskBucketName(Bucket bucket, int BTID)
    {
        String tableAlias = ((bucket.getHashTable() ==
bucket.getHashTable().getMobiJoin().getHashTableA()) ? "A" : "B");
        int subIndex = (BTID - 1) / bucket.getSize();
        return bucket.getHashTable().getMobiJoin().getDiskBucketsPath() +
"bucket" + tableAlias + "_" + bucket.getIndex() + "_" + subIndex;
    }

    public synchronized void tranferBucketToDisc(Bucket bucket)
    {
        String filename = getNewDiskBucketName(bucket);
        DiskBucket diskBucket = null;

        FileOutputStream fos = null;
        ObjectOutputStream out = null;
        try
        {
            fos = new FileOutputStream(filename);
            out = new ObjectOutputStream(fos);

            diskBucket = new DiskBucket(bucket.getIndex(),
bucket.getNDiskBuckets());
            diskBucket.addRows(bucket);

            out.writeObject(diskBucket);
            out.flush();
            out.close();
            bucket.incNDiskBuckets();
        } catch (IOException ex)
        {
            ex.printStackTrace();
        }
        bucket.clear();
    }

    public synchronized DiskBucket getBucketFromDisk(Bucket bucket, int
BTID)
    {
        String filename = getDiskBucketName(bucket, BTID);
        DiskBucket diskBucket = null;

        FileInputStream fis = null;
        ObjectInputStream in = null;
        try
        {
            fis = new FileInputStream(filename);
            in = new ObjectInputStream(fis);
            diskBucket = (DiskBucket) in.readObject();
            in.close();
        } catch (IOException ex)
        {
            ex.printStackTrace();
        } catch (ClassNotFoundException ex)
        {
            ex.printStackTrace();
```

```
        }
        return diskBucket;
    }

    public synchronized ControlTable[] getControlTables()
    {
        return controlTables;
    }

    public synchronized void compareRows(Row rowA, String indexColumnA, Row
rowB, String indexColumnB, int bucketIndex)
    {
        if ((rowA != null) && (rowB != null))
        {
            boolean equal =
rowA.get(indexColumnA).equals(rowB.get(indexColumnB));
            if (equal)
                nResultRows++;

            // Se tabela de controle do par de buckets existe
            // e os BTID das tuplas são maiores que zero. Atualiza tabela
de controle.
            // Obs: Quando um BTDI de uma tupla é 0(zero) isto indica que
esta tupla
            // não precisou ir para o bucket de memória, pois a relação
oposta já tinha
            // recebido todos os dados e não havia nehuma partição em
disco.
            if ((getControlTables()[bucketIndex] != null) &&
(rowA.getBTID() != 0) && (rowB.getBTID() != 0))
            {
                controlTables[bucketIndex].setPairValue(rowA.getBTID(),
rowB.getBTID(), true);
            }
            notifyRowsCompared(rowA, rowB, equal);
        }
    }

    public synchronized void logHashTable(Row lastRow, HashTable hashTable)
    {
        String tableAlias = ((hashTable == hashTableA) ? "A" : "B");
        log.append("Recebe tupla " + lastRow + " da Tabela " + tableAlias);
        log.append(hashTable.toString());
    }

    public void addListener(MobiJoinListener listener)
    {
        mobiJoinListeners.add(listener);
    }

    private void notifyRowsCompared(Row rowA, Row rowB, boolean equal)
    {
        for (int i = 0; i < mobiJoinListeners.size(); i++)
        {
            MobiJoinListener listener = (MobiJoinListener)
mobiJoinListeners.get(i);
            listener.rowsCompared(rowA, rowB, equal);
        }
    }

    private void notifyFaseStarted(int fase)
```

```
    {
        for (int i = 0; i < mobiJoinListeners.size(); i++)
        {
            MobiJoinListener listener = (MobiJoinListener)
mobiJoinListeners.get(i);
            listener.phaseStarted(fase);
        }
    }

    private void notifyFaseFinished(int fase)
    {
        for (int i = 0; i < mobiJoinListeners.size(); i++)
        {
            MobiJoinListener listener = (MobiJoinListener)
mobiJoinListeners.get(i);
            listener.phaseStopped(fase);
        }
    }

    private void notifyStarted()
    {
        for (int i = 0; i < mobiJoinListeners.size(); i++)
        {
            MobiJoinListener listener = (MobiJoinListener)
mobiJoinListeners.get(i);
            listener.started();
        }
    }

    private void notifyFinished()
    {
        for (int i = 0; i < mobiJoinListeners.size(); i++)
        {
            MobiJoinListener listener = (MobiJoinListener)
mobiJoinListeners.get(i);
            listener.finished();
        }
    }

    public synchronized void initPhase2()
    {
        currentPhase = 2;
        notifyFaseStarted(2);

        ControlTable controlTable = null;
        DiskBucket bufferBucket = null;
        //Selecionar uma tabela de controle
        for (int i = 0; i < controlTables.length; i++)
        {
            controlTable = controlTables[i];
            if (controlTable != null)
            {
                //Percorrer tabela de controle
                for (int l = 1; l <= controlTable.getHeight(); l++)
                {
                    for (int c = 1; c <= controlTable.getWidth(); c++)
                    {
                        //Se par não foi comparado
                        if (!controlTable.getPairValue(l, c))
                        {
```

```
                            //Se a tupla de A está em disco e a tupla de B
está em memória
                            if ((l <= nRowsPerBucket *
hashTableA.getBucket(i).getNDiskBuckets()) &&
                                (c > nRowsPerBucket *
hashTableB.getBucket(i).getNDiskBuckets()))
                            {
                                //Se o bucket não está no buffer
                                if ((bufferBucket == null) ||
                                    (bufferBucket.getIndex() != i) ||
                                    (bufferBucket.getSubIndex() != ((l-1) /
nRowsPerBucket)))
                                bufferBucket =
getBucketFromDisk(hashTableA.getBucket(i), l);
                                Row rowA = bufferBucket.getRow((l-1) %
nRowsPerBucket);
                                Row rowB =
hashTableB.getBucket(i).getRow((c-1) % nRowsPerBucket);

                                compareRows(rowA,
dataProviderA.getIndexColumn(), rowB, dataProviderB.getIndexColumn(), i);
                            }
                        }
                        //Se não há mais bloqueio em nenhuma das duas
tabelas
                        if (!isProviderABlocked && !isProviderBBlocked)
                        {
                            return;
                        }
                    }
                }
                bufferBucket = null;
                //Percorrer tabela de controle
                for (int l = 1; l <= controlTable.getHeight(); l++)
                {
                    for (int c = 1; c <= controlTable.getWidth(); c++)
                    {
                        //Se par não foi comparado
                        if (!controlTable.getPairValue(l, c))
                        {
                            //Se a tupla de B está em disco e a tupla de A
está em memória
                            if ((c <= nRowsPerBucket *
hashTableB.getBucket(i).getNDiskBuckets()) &&
                                (l > nRowsPerBucket *
hashTableA.getBucket(i).getNDiskBuckets()))
                            {
                                //Se o bucket não está no buffer
                                if ((bufferBucket == null) ||
                                    (bufferBucket.getIndex() != i) ||
                                    (bufferBucket.getSubIndex() != ((c-1) /
nRowsPerBucket)))
                                bufferBucket =
getBucketFromDisk(hashTableB.getBucket(i), c);
                                Row rowB = bufferBucket.getRow((c-1) %
nRowsPerBucket);
                                Row rowA =
hashTableA.getBucket(i).getRow((l-1) % nRowsPerBucket);

                                compareRows(rowA,
dataProviderA.getIndexColumn(), rowB, dataProviderB.getIndexColumn(), i);
```

```
                            }
                        }
                        //Se não há mais bloqueio em nenhuma das duas
tabelas
                        if (!isProviderABlocked && !isProviderBBlocked)
                        {
                            return;
                        }
                    }
                }
            }
        }
        currentPhase = 0;
        notifyFaseFinished(2);
    }

    public synchronized void eofReached(DataProvider provider)
    {
        if (dataProviderA.isEOF() && dataProviderB.isEOF() && !done)
            initPhase3();
    }

    public synchronized void initPhase3()
    {
        currentPhase = 3;
        notifyFaseStarted(3);

        DynHashTable hashTableAux = new DynHashTable(this);
        ControlTable controlTable = null;
        DiskBucket bufferBucket = null;
        //Selecionar uma tabela de controle
        for (int i = 0; i < controlTables.length; i++)
        {
            controlTable = controlTables[i];
            if (controlTable != null)
            {
                //Percorrer tabela de controle
                for (int l = 1; l <= controlTable.getHeight(); l++)
                {
                    for (int c = 1; c <= controlTable.getWidth(); c++)
                    {
                        //Se par não foi comparado
                        if (!controlTable.getPairValue(l, c))
                        {
                            //Se a tupla de A está em disco
                            if (l <= nRowsPerBucket *
hashTableA.getBucket(i).getNDiskBuckets())
                            {
                                //Se o bucket não está no buffer
                                if ((bufferBucket == null) ||
                                    (bufferBucket.getIndex() != i) ||
                                    (bufferBucket.getSubIndex() != ((l-1) /
nRowsPerBucket)))
                                {
                                    bufferBucket =
getBucketFromDisk(hashTableA.getBucket(i), l);
                                }
                                Row rowA = bufferBucket.getRow((l-1) %
nRowsPerBucket);

                                //Se a tupla de B está em memória
```

```
                                if (c > nRowsPerBucket *
hashTableB.getBucket(i).getNDiskBuckets())
                                {
                                      Row rowB =
hashTableB.getBucket(i).getRow((c-1) % nRowsPerBucket);
                                      compareRows(rowA,
dataProviderA.getIndexColumn(), rowB, dataProviderB.getIndexColumn(), i);
                                }
                                hashTableAux.put(rowA,
dataProviderA.getIndexColumn());
                            }
                        }
                    }
                }
                bufferBucket = null;
                //Percorrer tabela de controle
                for (int l = 1; l <= controlTable.getHeight(); l++)
                {
                    for (int c = 1; c <= controlTable.getWidth(); c++)
                    {
                        //Se par não foi comparado
                        if (!controlTable.getPairValue(l, c))
                        {
                            //Se a tupla de B está em disco
                            if (c <= nRowsPerBucket *
hashTableB.getBucket(i).getNDiskBuckets())
                            {
                                //Se o bucket não está no buffer
                                if ((bufferBucket == null) ||
                                    (bufferBucket.getIndex() != i) ||
                                    (bufferBucket.getSubIndex() != ((c-1) /
nRowsPerBucket)))
                                bufferBucket =
getBucketFromDisk(hashTableB.getBucket(i), c);
                                Row rowB = bufferBucket.getRow((c-1) %
nRowsPerBucket);
                                Row rowA = null;
                                //Se a tupla de A está em memória
                                if (l > nRowsPerBucket *
hashTableA.getBucket(i).getNDiskBuckets())
                                {
                                      rowA =
hashTableA.getBucket(i).getRow((l-1) % nRowsPerBucket);
                                }
                                else
                                {
                                    DynBucket bucketAux =
hashTableAux.getBucket(rowB, dataProviderB.getIndexColumn());
                                    for (int j = 0; j <
bucketAux.getSize(); j++)
                                    {
                                        Row rowAux = bucketAux.getRow(j);
                                        if (rowAux.getBTID() == l)
                                        {
                                            rowA = rowAux;
                                            break;
                                        }
                                    }
                                }
                                compareRows(rowA,
dataProviderA.getIndexColumn(), rowB, dataProviderB.getIndexColumn(), i);
```

```
                        }
                    }
                }
            }
        }
    }
    notifyFaseFinished(3);
    done = true;
    notifyFinished();
}

public synchronized void rowReceived(Row row, DataProvider provider)
{
    HashTable hashTable = getOwnHashTable(provider);
    String indexColumn = provider.getIndexColumn();
    //1a. fase
    Bucket bucket = hashTable.getBucket(row, indexColumn);

    if (bucket != null)
    {
        //Se overflow no bucket ocorrer
        if (bucket.getNRows() == bucket.getSize())
        {
            //Se tabela de controle do par de buckets não existe
            if (getControlTables()[bucket.getIndex()] == null)
            {
                generateControlTable(bucket);
            }
            //

            //Só é possível tranferir o bucket depois da criação da
            //tabela de controle
            tranferBucketToDisc(bucket);

            bucket.put(row);
            Row[] opositeBucketRows =
getOpositeBucket(bucket).getRows();

            Row rowA;
            String indexColumnA;
            Row rowB;
            String indexColumnB;
            //Para cada tupla do bucket oposto
            for (int i = 0; i < opositeBucketRows.length; i++)
            {
                if (opositeBucketRows[i] != null)
                {
                    if (bucket.getHashTable() ==
bucket.getHashTable().getMobiJoin().getHashTableA())
                    {
                        rowA = row;
                        indexColumnA = indexColumn;
                        rowB = opositeBucketRows[i];
                        indexColumnB =
getOpositeTableProvider(provider).getIndexColumn();
                    } else
                    {
                        rowA = opositeBucketRows[i];
                        indexColumnA =
getOpositeTableProvider(provider).getIndexColumn();
                        rowB = row;
```

```
                            indexColumnB = indexColumn;
                        }
                        compareRows(rowA, indexColumnA, rowB, indexColumnB,
bucket.getIndex());
                    }
                }
            }
            else
            {
                //Se fim da recepção da relação oposta
                if (getOpositeTableProvider(provider).isEOF())
                {
                    //Se o bucket oposto possui partição em disco
                    if (getOpositeBucket(bucket).getNDiskBuckets() > 0)
                    {
                        bucket.put(row);
                    }
                } else
                {
                    bucket.put(row);
                }
                Row[] opositeBucketRows =
getOpositeBucket(bucket).getRows();

                Row rowA;
                String indexColumnA;
                Row rowB;
                String indexColumnB;
                //Para cada tupla do bucket oposto
                for (int i = 0; i < opositeBucketRows.length; i++)
                {
                    if (opositeBucketRows[i] != null)
                    {
                        if (bucket.getHashTable() ==
bucket.getHashTable().getMobiJoin().getHashTableA())
                        {
                            rowA = row;
                            indexColumnA = indexColumn;
                            rowB = opositeBucketRows[i];
                            indexColumnB =
getOpositeTableProvider(provider).getIndexColumn();
                        } else
                        {
                            rowA = opositeBucketRows[i];
                            indexColumnA =
getOpositeTableProvider(provider).getIndexColumn();
                            rowB = row;
                            indexColumnB = indexColumn;
                        }
                        compareRows(rowA, indexColumnA, rowB, indexColumnB,
bucket.getIndex());
                    }
                }
            }
        }
        //DEBUG
        //logHashTable(row, hashTable);
    }

    public int getNResultRows()
    {
```

```
        return nResultRows;
    }

    public void setDiskBucketsPath(String diskBucketsPath)
    {
        this.diskBucketsPath = diskBucketsPath;
    }

    public synchronized void flowBlocked(DataProvider provider)
    {
        if (provider == dataProviderA)
            isProviderABlocked = true;
        else
            isProviderBBlocked = true;

        if ((currentPhase != 2) && isProviderABlocked &&
isProviderBBlocked)
        {
            notifyFaseFinished(1);
                initPhase2();
        }
    }

    public synchronized void flowReleased(DataProvider provider)
    {
        if (provider == dataProviderA)
            isProviderABlocked = false;
        else
            isProviderBBlocked = false;

        if (currentPhase != 1)
        {
            currentPhase = 1;
            notifyFaseStarted(1);
        }
    }
}
```

A.10 A classe Row

```
package mobijoin;

import java.io.Serializable;
import java.util.LinkedHashMap;
import java.util.Set;

public class Row implements Serializable
{
    private LinkedHashMap columns = new LinkedHashMap();
    private int BTID;

    public Object get(String columnName)
    {
        return columns.get(columnName);
    }
```

```
    public void set(String columnName, Object value)
    {
        columns.put(columnName, value);
    }

    public String toString()
    {
        //DEBUG
        //return "BTID: " + BTID + " = " + columns.toString();

        return columns.toString();
    }

    public void setBTID(int value)
    {
        BTID = value;
    }

    public int getBTID()
    {
        return BTID;
    }

    public Set getColumnsNames()
    {
        return columns.keySet();
    }
}
```

A.11 A classe DefaultMobiJoinListener

```
package mobijoin.listeners;

import mobijoin.Row;

public class DefaultMobiJoinListener implements MobiJoinListener
{
    public void rowsCompared(Row rowA, Row rowB, boolean equal){}
    public void phaseStarted(int phase){}
    public void phaseStopped(int phase){}
    public void started(){}
    public void finished(){}
}
```

A.12 A classe DefaultProviderListener

```
package mobijoin.listeners;

import mobijoin.DataProvider;
import mobijoin.Row;

public class DefaultProviderListener implements ProviderListener
{
```

```
    public void rowReceived(Row row, DataProvider provider){}
    public void eofReached(DataProvider provider){}
    public void flowBlocked(DataProvider provider){}
    public void flowReleased(DataProvider provider){}
}
```

A.13 A interface MobiJoinListener

```
package mobijoin.listeners;

import mobijoin.Row;

public interface MobiJoinListener
{
    public void rowsCompared(Row rowA, Row rowB, boolean equal);
    public void phaseStarted(int phase);
    public void phaseStopped(int phase);
    public void started();
    public void finished();
}
```

A.14 A interface ProviderListener

```
package mobijoin.listeners;

import mobijoin.DataProvider;
import mobijoin.Row;

public interface ProviderListener
{
    public void rowReceived(Row row, DataProvider provider);
    public void eofReached(DataProvider provider);
    public void flowBlocked(DataProvider provider);
    public void flowReleased(DataProvider provider);
}
```

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo