Universidade Federal do Ceará
Mestrado em Ciência da Computação

Pré-Cálculo de Junções para o Processamento de Consultas em Ambientes com Recursos Computacionais Limitados

Tathianne Moreira Paulino

Dissertação apresentada ao Mestrado em Ciência da Computação como requisito parcial para a obtenção do título de Mestre em Ciência da Computação pela Universidade Federal do Ceará (UFC). Área de Concentração: Banco de Dados.

FORTALEZA
Ceará - Brasil
13 de outubro de 2004

Universidade Federal do Ceará
Mestrado em Ciência da Computação

Pré-Cálculo de Junções para o Processamento de Consultas em Ambientes com Recursos Computacionais Limitados

Tathianne Moreira Paulino

Orientador: Prof. Dr. Angelo Brayner

Dissertação apresentada ao Mestrado em Ciência da Computação como requisito parcial para a obtenção do título de Mestre em Ciência da Computação pela Universidade Federal do Ceará (UFC). Área de Concentração: Banco de Dados.

FORTALEZA
Ceará - Brasil
13 de outubro de 2004

Pré-Cálculo de Junções para o Processamento de Consultas em Ambientes com Recursos Computacionais Limitados

Tathianne Moreira Paulino

Dissertação aprovada em 30 de agosto de 2004

Prof. Dr. Angelo Brayner
Universidade de Fortaleza
Orientador e Componente da Banca Examinadora

Prof. Dr. Javam Machado
Universidade Federal do Ceará
Componente da Banca Examinadora

Prof. Dr. Sérgio Lifschitz
Pontifícia Universidade Católica do Rio de Janeiro
Componente da Banca Examinadora

FORTALEZA
Ceará - Brasil
13 de outubro de 2004

© Tathianne Moreira Paulino, 2004.

Todos os direitos reservados.

A Deus,
a quem tudo devemos.

À minha mãe Constancia,
pelo amor a mim dedicado.

Ao meu noivo Fabio,
pelo incentivo, colaboração, compreensão e
paciência.

Agradecimentos

Ao meu orientador Angelo pelo incentivo, paciência, ensinamento, dedicação e tempo dispensados no criterioso trabalho de acompanhamento e orientação deste trabalho. Agradeço também pelos seus valiosos conselhos nos momentos difíceis por que passei durante o desenvolvimento do mesmo.

Ao Professor Dr. Sérgio Lifschitz e ao Professor Dr. Javam Machado, por terem aceitado o convite para participar da banca examinadora e por colaborarem para a melhoria deste trabalho.

Um enorme agradecimento à minha mãe, a maior responsável por eu ter chegado aonde cheguei. Por ser a mulher mais forte, mais corajosa e mais batalhadora que eu já conheci. Por sempre ter feito tudo por mim. Por ter me dado tudo que estava ao seu alcance. Por ser minha "pãe"(minha mãe e meu pai em uma única pessoa) e por representar a pessoa mais importante da minha vida.

Um agradecimento muito especial, ao meu noivo Fabio, que tanto me incentivou e apoiou durante todo este trabalho. Pelo enorme e constante amor, paciência, compreensão e companheirismo. Por ter estudado comigo, por ter tirado minhas dúvidas, pelas sugestões, pelas constantes revisões no decorrer deste trabalho, enfim, por ter colaborado diretamente na execução desta dissertação.

À minha afilhada Nikole, por me fazer esquecer todos os problemas sempre que me dá um abraço bem apertado, um sorriso bem gostoso e me chama de "Madinha", aquietando meu coração.

Aos meus familiares que sempre estiveram ao meu lado, entendendo as minhas ausências nas comemorações pertinentes à nossa família.

Ao meu amigo Lineu, por ter suportado meus momentos de *stress*. Por ter me incentivado, mesmo quando não se sentia motivado a concluir o seu próprio trabalho.

Enfim, pela amizade, apoio e atenção dispensados comigo.

Ao meu amigo Joney, vulgo Joneco, por ter me apresentado ao meu orientador Angelo. Por ter me ajudado nos momentos de dificuldade quando eu achava que seria impossível ter o Angelo como meu orientador. Joney, devo a você ter conseguido o orientador que todo aluno sonha em ter. Sem sua ajuda, não sei se teria conseguido.

Ao meu amigo Rodrigo, por ter me ajudado na revisão e na validação deste trabalho, inclusive por ter me emprestado seu celular para desempenhar meus testes. Pela amizade e ajuda nos momentos difíceis. A você, Rodrigo, minha eterna gratidão.

Ao meu amigo Emanuel, mais conhecido como "Manu", por também ter me ajudado na revisão deste trabalho, por ter dado várias sugestões de melhoria e por sua amizade.

Ao amigo Marcelo, por ter me motivado a concluir este trabalho. Por você, a minha profunda consideração, respeito e amizade.

Aos amigos Vando, Sara e Auler pela amizade construída nesse último ano.

Aos colegas e amigos da turma de mestrado de 2001, pela amizade, companheirismo e apoio durante o curso. Pelo tempo dispensado nas tardes e noites de estudos.

Aos amigos que nos últimos três anos compartilharam comigo momentos de alegria e de tristeza, dentre eles, Aninha, Victória, Valdiana, Daniel, Wládia, Ana Lúcia, Alexandra, Waryson e Walyson.

Agradeço também ao meu amigo João Fernando, que mesmo estando longe, nunca esqueceu de mim.

À Universidade Federal do Ceará pela oportunidade do grande crescimento profissional.

Aos professores e funcionários do Mestrado em Ciências da Computação da Universidade Federal do Ceará pelo apoio à realização do curso.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), pelo auxílio financeiro que possibilitou a realização deste trabalho.

À Universidade de Fortaleza, por ter permitido que o Prof. Dr. Angelo Brayner continuasse a ser meu orientador, mesmo eu não possuindo vínculos com a mesma.

Ao Instituto Atlântico, empresa onde trabalho, por ter me apoiado na realização deste mestrado, por saber reconhecer o valor que este curso representa para mim e pela oportunidade de crescimento profissional e pessoal.

Aos meus colegas de trabalho do Instituto Atlântico, que de alguma forma me ajudaram.

Agradeço, especialmente, a Deus, infinita luz do saber, por sua bondade e generosidade para comigo. Por sempre ter estado presente na minha vida e por ter me dado forças para concluir esta dissertação.

Enfim, agradeço a todos aqueles que contribuíram direta ou indiretamente para a conclusão desta dissertação.

Resumo

PAULINO, Tathianne. Uma Abordagem para Otimizar o Processamento de Consultas em Ambientes com Recursos Computacionais Limitados. Orientador: Angelo Brayner. Fortaleza-CE, UFC, 2004. Diss.

Atualmente, carregar dados em dispositivos computacionais portáteis, tais como *smartcards*, *palmtops* e telefones celulares, está se tornando comum. Isso porque usuários portando esses dispositivos são capazes de acessar serviços, a qualquer hora e em qualquer lugar, mesmo que eles não estejam fisicamente conectados a uma rede, independente de sua localização física e de seus movimentos. Entretanto, dispositivos portáteis possuem severas limitações de recursos computacionais, tais como capacidades de processamento e memória restritas para o gerenciamento dos dados. O presente trabalho enfoca o problema do processamento eficiente de consultas em ambientes com recursos computacionais limitados. Duas estruturas de armazenamento, chamadas *MSJoin* e *MSJoin*$^+$, são propostas com o objetivo de otimizar a execução do operador relacional de junção. As estruturas propostas garantem a execução do operador de junção através do algoritmo *Merge-Join*, o qual apresenta a menor estimativa de custo para uma operação de junção. Além disso, a estrutura *MSJoin*$^+$ garante a compactação de dados, um aspecto crucial para ambientes com recursos limitados.

Palavras-chave: *Smartcards, Processamento de Consultas, Otimização de Consultas, Bancos de Dados Ubíquos, Ambientes com Recursos Computacionais Limitados.*

Abstract

PAULINO, Tathianne. An Approach for Optimizing Query Processing in Environments with Limited Computational Resources. Advisor: Angelo Brayner. Fortaleza-CE, UFC, 2004. Diss.

Nowadays, carrying data on portable computing devices, such as smartcards, palmtops and cell phones, is becoming common. This is because users holding these devices are able to access services, anytime and anyplace, even though they are not physically connected to a network, regardless of their physical location or movement patterns. However, portable devices have severe computational resource limitations, such as restricted processing and memory capacities for data management. The present work focuses the problem of efficient query processing in computational environments with limited resources. Two storage structures, called MSJoin and MSJoin$^+$, are proposed for optimizing the execution of the join relational operator. The proposed structures ensure the execution of the join operator through the Merge-Join algorithm, which presents the lowest estimation cost for the join operation. Furthermore, the MSJoin$^+$ structure ensures data compactness, a crucial feature for environments with limited resources.

Keywords: *Smartcards, Query Processing, Query Optimization, Ubiquitous Database, Environments with Limited Computational Resources.*

Sumário

Resumo ix

Abstract x

Lista de Figuras xiii

1 Introdução 1

1.1 Motivação 1

1.2 Objetivos da Dissertação 3

1.3 Estrutura da Dissertação 4

2 Ambientes com Recursos Computacionais Limitados 5

2.1 Introdução 5

2.2 Ambientes Computacionais Limitados 5

2.3 Um Ambiente Computacional Armazenável em *Smartcards* 6

2.4 Bancos de Dados Ubíquos 11

3 Processamento de Consultas 14

3.1 Introdução 14

3.2 Processamento de Consultas Estático 15

3.2.1 Fases do Processamento de Consultas 15

3.2.2 Implementação dos Operadores Relacionais 18

3.2.2.1 Operador de Seleção 19

3.2.2.2 Operador de Junção 21

3.2.2.3 Pré-Cálculo de Junção 27

3.3 Processamento de Consultas Adaptativo 29

3.3.1 Ripple Joins 31
3.3.2 XJoin 33
3.4 Processamento de Consultas em Ambientes Computacionais Limitados 35
3.4.1 Principais Limitações 35
3.4.2 Processamento de Consultas em *Smartcards* 37
3.4.3 Modelo de Armazenamento de Domínio 40

4 MSJoin e MSJoin$^+$: Garantindo Suporte para Otimização do Operador de Junção 42
4.1 Introdução 42
4.2 Árvores B$^+$ 43
4.3 Estrutura MSJoin 46
4.4 Estrutura MSJoin$^+$ 51

5 Validação 56
5.1 Introdução 56
5.2 Inserção em árvores B$^+$ 56
5.3 Remoção em árvores B$^+$ 57
5.4 Validação da Estrutura MSJoin 61
5.4.1 Ambiente de Simulação 61
5.4.2 Resultados da Simulação 64

6 Conclusão 66
6.1 Considerações Sobre as Estruturas Propostas 66
6.2 Trabalhos Futuros 67
6.3 Considerações Finais 68

Referências Bibliográficas 69

Lista de Figuras

3.1 Fases do Processamento de Consultas. . . . 17
3.2 Algoritmo Nested-Loop Join. . . . 22
3.3 Algoritmo Block Nested-Loop Join. . . . 24
3.4 Algoritmo Merge-Join. . . . 24
3.5 Algoritmo Hash-Join. . . . 26
3.6 Representação Gráfica de Índice de Junção. . . . 29
3.7 Modelo de Armazenamento de Domínio 40

4.1 Estrutura de um nó interno de uma árvore B^+. . . . 45
4.2 Estrutura de um nó folha de uma árvore B^+. . . . 46
4.3 Estrutura MSJoin. . . . 48
4.4 Nó folha de uma árvore B^+ estendida. . . . 49
4.5 Representação gráfica da Estrutura MSJoin. . . . 50
4.6 Representação da estrutura MSJoin usando a árvore B^+ estendida. . . 51
4.7 Estrutura $MSJoin^+$. . . . 52
4.8 Representação da estrutura $MSJoin^+$. . . . 54
4.9 Representação da estrutura $MSJoin^+$ usando a árvore B^+ estendida. . 55

5.1 Chamada ao procedimento inserirEntrada. . . . 56
5.2 Algoritmo de inserção de uma entrada em uma árvore B^+ . . . 58
5.3 Chamada ao procedimento removerEntrada . . . 59
5.4 Algoritmo de remoção de uma entrada em uma árvore B^+ . . . 60
5.5 Estimativa de custo . . . 64
5.6 Estimativa de custo em escala logarítmica . . . 64

1 Introdução

1.1 Motivação

Nos últimos anos, os dispositivos computacionais portáteis incluindo *smartcards*[1], Assistentes Digitais Pessoais, do inglês *Personal Digital Assistants (PDAs)*, telefones celulares, *iButtons* [10], dentre outros sistemas embutidos, vêm despertando grande interesse de pessoas das mais diversas áreas, das mais diversas empresas, em todo o mundo. Isto é um fato compreensível tendo em vista que usuários carregando estes dispositivos são capazes de acessar serviços, independente da sua localização física ou de seus movimentos, mesmo que eles não estejam fisicamente conectados a uma rede. Entretanto, dispositivos portáteis possuem severas limitações computacionais, principalmente devido ao seu reduzido tamanho e por terem sido projetados para fornecer mobilidade. Como exemplo dessas limitações, podem ser citadas: pequena capacidade de armazenamento e de processamento, escrita muito lenta, memória RAM muito pequena, baixa autonomia de bateria, taxas de transmissão de dados caras e limitadas [4].

Dentre os dispositivos computacionais portáteis que possuem mais limitações estão os *smartcards* (cartões de plástico do tamanho de um cartão de crédito convencional com microchips embutidos capazes de armazenar e processar dados eletrônicos). Atualmente, *smartcards* estão entre os mais modernos e seguros dispositivos computacionais portáteis existentes [4]. Conseqüentemente, um número cada vez maior de aplicações estão sendo desenvolvidas utilizando a tecnologia

[1] O termo *smartcard* é bastante intuitivo. Entretanto, é um termo ambíguo que pode ser utilizado de diferentes formas. A ISO (*International Standard Organization*) usa o termo Cartão de Circuito Integrado (*Integrated Circuit Card - ICC*) para se relacionar a todos os dispositivos onde um cartão de identificação de plástico ISO ID1 contém um circuito integrado [12].

de *smartcards* e obtendo excelentes resultados. Além de aplicações bancárias e aplicações na área de telefonia celular podem ser mencionadas ainda aplicações de *smartcards* nas áreas de saúde, identificação pessoal, transportes (passagens eletrônicas) e finanças (porta moedas eletrônico).

Tais aplicações armazenam dados críticos, como por exemplo, o saldo de uma conta (e os registros de *log* refletindo operações de débitos e créditos sobre a conta), dados pessoais de saúde e senhas. Conseqüentemente, um gerenciamento eficiente desses dados é uma necessidade estratégica para essas aplicações. A tecnologia de banco de dados existente foi consolidada como solução para o gerenciamento de dados. Processamento de consultas, controle de concorrência, mecanismos de recuperação e armazenamento de dados são as principais funcionalidades de Sistemas de Bancos de Dados que garantem que os dados sejam gerenciados de forma eficiente. Portanto, é importante introduzir técnicas e funcionalidades de bancos de dados para gerenciar os dados armazenados em ambientes com recursos computacionais limitados de maneira eficiente. Uma vez que esses ambientes possuem severas limitações computacionais, as técnicas de bancos de dados tradicionais não podem ser diretamente aplicadas a eles [4].

Estratégias de processamento de consultas relacionais tradicionais podem requisitar uma grande quantidade de memória e de acessos a disco para processar as operações relacionais. Por exemplo, a operação de junção é uma das operações que mais consome tempo e recursos no processamento de consultas relacionais. Logo, adaptar técnicas de bancos de dados convencionais para gerenciar dados em ambientes com recursos computacionais limitados tem surgido como um grande desafio para a comunidade de banco de dados.

Enquanto existem excelentes trabalhos desenvolvidos para aprimorar técnicas existentes que lidam com grandes bancos de dados (VLDB - *Very Large Database*), pouca atenção tem sido dada à adaptação de técnicas para lidar com bancos de dados desenvolvidos para ambientes computacionais limitados. No entanto, desenvolver tais técnicas é um tanto quanto difícil e por isso merece atenção.

Assim, a presente dissertação enfoca o problema do processamento eficiente de consultas em ambientes com recursos computacionais limitados. *Smartcards* são utilizados como ambientes computacionais para validar a abordagem proposta, desde que tais ambientes representam o pior caso para o problema de limitação de recursos computacionais.

1.2 Objetivos da Dissertação

Diante da motivação exposta, teve-se como objetivo otimizar a execução de operações de junção em ambientes computacionais com recursos limitados, a fim de garantir o processamento eficiente de consultas em tais ambientes. Na realidade, o que se propõe é uma estratégia de pré-cálculo de junções para o processamento de consultas em ambientes com recursos computacionais limitados.

Duas estruturas de armazenamento, chamadas *MSJoin* e *MSJoin*$^{+}$, são propostas com o objetivo de otimizar a execução do operador relacional de junção. A idéia chave das estruturas de armazenamento propostas é permitir a execução do algoritmo *Merge-Join* para processar operações de junção. Desse modo, a execução do operador de junção é otimizada uma vez que tal algoritmo garante a menor estimativa de custo para realização de uma operação de junção. Além disso, a estrutura *MSJoin*$^{+}$ garante a compactação de dados, um aspecto crucial para ambientes com recursos limitados.

Para validar e avaliar a *performance* da estrutura *MSJoin*, foi feita uma simulação para comparar a execução de junçao utilizando a estrutura *MSJoin* com a execução de junção utilizando o algoritmo *Nested-Loop Join*. Para completar a validação, a implementação da estrutura *MSJoin* e da simulação realizada sobre a mesma foram migradas da plataforma J2SE 1.4.2 [25] para a plataforma J2ME (CLDC 1.1/MIDP 2.0) [27, 28].

1.3 Estrutura da Dissertação

Esta dissertação apresenta uma abordagem que tem por objetivo otimizar o operador relacional de junção em ambientes com recursos computacionais limitados. Para tal, o seu conteúdo foi dividido nos seguintes capítulos:

Capítulo 2: Descreve os ambientes com recursos computacionais limitados, apresenta com mais detalhes as características de um ambiente computacional armazenável em *smartcards* e também algumas propostas de bancos de dados ubíquos existentes no mercado.

Capítulo 3: Fornece uma revisão bibliográfica sobre processamento de consultas de uma forma geral, englobando processamento de consultas estático, passando pelo processamento de consultas adaptativo e finalizando no processamento de consultas em ambientes com recursos computacionais limitados.

Capítulo 4: Descreve a estrutura de índice baseada em árvores B^+ e apresenta as estruturas de armazenamento propostas, chamadas de *MSJoin* e *MSJoin*$^+$, que têm como principal objetivo otimizar a execução do operador relacional de junção, um dos operadores que mais consome tempo e recursos no processamento de consultas relacionais.

Capítulo 5: Apresenta os algoritmos de inserção e remoção em uma árvore B^+, já que as estruturas propostas são baseadas em árvores B^+. Descreve como foi realizada a implementação para validar a estrutura *MSJoin* e apresenta os resultados da simulação feita para comparar a execução de uma junção utilizando a estrutura *MSJoin* com a execução utilizando o algoritmo *Nested-Loop Join*.

Capítulo 6: Apresenta as conclusões e sugestões para trabalhos futuros.

2 Ambientes com Recursos Computacionais Limitados

2.1 Introdução

Este capítulo descreve os ambientes com recursos computacionais limitados. Além disso, são apresentadas com mais detalhes as características de um ambiente computacional armazenável em *smartcards*. Por fim, são mostradas algumas propostas de bancos de dados ubíquos existentes no mercado.

2.2 Ambientes Computacionais Limitados

Um ambiente com recursos computacionais limitados caracteriza-se por ser um ambiente computacional que possui severas restrições de *hardware* (principalmente por causa do seu reduzido tamanho), como por exemplo, pequena capacidade de armazenamento e de processamento, escrita muito lenta, memória RAM muito pequena, baixa autonomia de bateria, taxas de transmissão de dados caras e limitadas. Conseqüentemente, esses ambientes são capazes de processar apenas uma pequena quantidade de dados. Neste trabalho, define-se um ambiente com recursos computacionais limitados como sendo aquele que apresenta as mesmas características que um *smartcard*, visto que na literatura *smartcards* são considerados ambientes com recursos computacionais limitados.

Como ambientes computacionais limitados, podem ser citados os Assistentes Digitais Pessoais (PDAs), os telefones celulares, os *iButtons* e os *smartcards*.

Um Assistente Digital Pessoal é um pequeno dispositivo computacional móvel que permite ao usuário acessar, armazenar e organizar informações. Alguns PDAs possuem tela sensível ao toque (*touchscreen*) e reconhecimento de escrita, enquanto

outros possuem pequenos teclados integrados. PDAs são também conhecidos como *handhelds* ou *palmtops*.

Dentre os dispositivos com recursos computacionais limitados, os que possuem mais restrições são os *smartcards* e os *iButtons*. Um *smartcard* é um cartão de plástico do tamanho de um cartão de crédito convencional que possui um *microchip* embutido capaz de armazenar e processar dados eletrônicos. Um *iButton* é um *microchip* embutido em uma pequena caixa de aço [10]. *iButtons* são pequenos e portáveis como *smartcards*, inclusive possuindo as mesmas limitações, porém estão mais próximos do usuário pelo simples fato de que eles podem ser facilmente anexados a itens pessoais, como relógios ou anéis, tornando sua utilização bem mais cômoda.

Enquanto ainda existem poucas aplicações utilizando a tecnologia de *iButtons* (por exemplo, aplicações de controle de acesso a prédios e computadores), um número cada vez maior de aplicações estão sendo desenvolvidas utilizando a tecnologia de *smartcards*, dentre elas aplicações na área de telefonia celular, aplicações bancárias, aplicações financeiras, aplicações de identificação pessoal, dentre outras.

Apesar de *iButtons* e *smartcards* possuírem as mesmas limitações, *smartcards* serão utilizados como ambientes computacionais para validar a abordagem proposta. Por ser uma tecnologia mais recente, existe pouca informação disponível sobre a tecnologia de *iButtons*.

O ambiente de *smartcards* será descrito com mais detalhes na seção a seguir.

2.3 Um Ambiente Computacional Armazenável em *Smartcards*

De uma forma genérica, pode-se definir um *smartcard* como sendo um cartão de plástico do tamanho de um cartão de crédito convencional com um *chip* de computador nele embutido capaz de armazenar e processar dados eletrônicos, que pode oferecer diversos serviços, tais como serviços sofisticados de criptografia, além de

possuir um custo relativamente baixo [5, 7, 9, 12, 15]. Dentre algumas outras vantagens dos *smartcards* podem ser mencionadas:

- ▹ São bastante flexíveis, isto é, adaptam-se a inúmeros tipos de aplicações;
- ▹ São recarregáveis, isto é, os dados podem ser carregados diversas vezes no mesmo cartão;
- ▹ São multioperacionais, ou seja, podem oferecer vários serviços ao mesmo tempo em um só cartão.

Smartcards interagem com um dispositivo conhecido como leitora. Quanto à forma de comunicação com a leitora de cartões, os *smartcards* podem ser de dois tipos:

- ▹ Cartões com contato (*Contact Cards*), onde a comunicação é feita através de um contato físico direto;
- ▹ Cartões sem contato (*Contactless Cards*), onde a comunicação é feita remotamente através de uma interface eletromagnética sem contato.

Os cartões com contato são os mais comuns por causa do seu amplo uso na Europa em aplicações de telefonia pré-paga. São fáceis de serem identificados porque possuem um conector dourado visível em uma das suas bordas. Um cartão com contato precisa ser inserido em uma leitora de cartão para que possa haver a transmissão de dados e de comandos da leitora para o cartão e vice-versa. Essa transmissão ocorre através dos pontos físicos de contato que os cartões possuem.

Os cartões sem contato são ideais para transações rápidas, como por exemplo, a coleta de taxas de forma eletrônica em aplicações na área de transportes. Entretanto, ainda não atingem uma fatia de mercado significativa. Um cartão sem contato necessita apenas estar próximo a uma leitora de cartão para que a comunicação ocorra, pois tanto a leitora como o cartão possuem uma antena e é por meio dessas antenas que ocorre a comunicação entre ambos sem nenhum contato físico. Essa

comunicação se dá através do uso de uma avançada transmissão de sinal de rádio. Muitos cartões sem contato contêm sua própria bateria, suprida através dos sinais eletromagnéticos [7].

Existem ainda dois tipos especiais de *smartcards* derivados dos dois tipos descritos acima [7]:

- ▹ Cartões Híbridos, que possuem dois *chips*, sendo que um possui uma interface com contato e o outro possui uma interface sem contato;
- ▹ Cartões *Combi*, que possuem um único *chip* contendo uma interface com contato e sem contato ao mesmo tempo.

Além dessa classificação dos *smartcards* quanto à forma de comunicação, pode-se ainda classificá-los quanto ao seu conteúdo, podendo ser de três tipos [12]:

- ▹ Cartões de Memória;
- ▹ Cartões de Memória com Lógica de Segurança;
- ▹ Cartões Inteligentes.

Os cartões de memória foram projetados especialmente para armazenar informações e oferecem uma grande vantagem com relação aos cartões magnéticos tradicionais. Eles podem ter seu conteúdo gravado e atualizado muitas vezes durante seu ciclo de vida, enquanto os cartões magnéticos tradicionais só podem ter seu conteúdo gravado uma única vez, apesar desse conteúdo poder ser acessado diversas vezes [5].

Os cartões de memória com lógica de segurança são similares aos cartões de memória. A diferença é que eles contêm uma lógica de controle para garantir que os cartões não possam ser violados. Essa lógica de segurança é utilizada para controlar o acesso a memória, usualmente através de um código de segurança. Esse tipo de cartão também pode utilizar criptografia a fim de garantir características adicionais

de segurança. Tanto os cartões de memória quanto os cartões de memória com lógica de segurança são bastante utilizados em aplicações de telefonia pré-paga [5].

Os cartões inteligentes podem ser vistos como computadores sem os dispositivos de entrada e saída anexados a eles. Eles possuem uma unidade de microprocessamento, uma memória volátil (RAM), uma memória não volátil (ROM), uma memória para armazenar dados persistentes (EEPROM) e o seu próprio sistema operacional. Seu microprocessador utiliza um conjunto de instruções que podem processar operações matemáticas, tais como, soma, subtração, multiplicação e divisão. Além disso, também possuem uma lógica para controlar os vários elementos do dispositivo, o que em alguns casos, inclui um sistema de controle de interrupção, que pode ser usado, por exemplo, quando a leitora quiser enviar um sinal para o cartão informando que já terminou de processar alguns dados e que já está pronta para receber mais dados do cartão [5].

Segundo Everett [12], o componente fundamental de um *smartcard* é o módulo de memória. Por isso, serão descritos a seguir os tipos de memória mais utilizados nesses dispositivos. Os principais são:

- RAM (*Random Access Memory*);
- ROM (*Read Only Memory*);
- PROM (*Programmable Read Only Memory*);
- EPROM (*Erasable Programmable Read Only Memory*);
- EEPROM (*Electrically Erasable Programmable Read Only Memory*);
- *Flash*-EEPROM;
- FeRAM (*Ferroeletric* RAM).

Estes tipos de memória possuem características próprias que controlam seu método de uso. A seguir serão mostradas algumas dessas características.

A memória RAM é usada como a memória de trabalho do *smartcard*, a fim de calcular resultados. É o tipo de memória mais rápido para esse propósito.

A memória ROM é uma memória permanente, não pode ser alterada e, apesar de seu baixo custo, leva meses para ser produzida. Em um *smartcard*, ela é usada para armazenar seu sistema operacional, dados fixos e rotinas padrões [4].

A memória PROM, como o próprio nome já diz, tem a habilidade de ser programada pelo usuário. Entretanto, a dificuldade de programação desse tipo de memória está fazendo com que ela não seja mais tão utilizada em *smartcards*.

A memória EPROM já foi amplamente utilizada no passado, mas o nome desse tipo de memória é mal denominado, pois geralmente seu conteúdo era programado uma única vez. Isto porque, para apagá-la é necessário utilizar luz ultravioleta, a fim de possibilitar uma nova escrita. Logo, o uso dessa memória em *smartcards* tornou-se quase obsoleto.

Já a memória EEPROM pode ser considerada realmente apagável pelo usuário, pois ela é apagada eletronicamente, e sua reescrita é possível muitas vezes. Esse tipo de memória é utilizado para armazenar informações persistentes no *smartcard*. Ela possui um tempo de leitura muito rápido se comparada com a memória RAM, mas um tempo de escrita muito lento [4].

Uma possível alternativa para substituir a memória EEPROM é a memória *Flash*-EEPROM [14]. Ela é mais compacta que a memória EEPROM e é uma boa candidata para *smartcards* de alta capacidade. A principal diferença entre esses dois tipos de memória é a velocidade no processo de escrita. Mesmo sendo necessário que os bancos da memória *Flash*-EEPROM sejam apagados antes da escrita (um processo extremamente lento), a memória *Flash*-EEPROM ainda é mais rápida que a memória EEPROM. Seu tempo de escrita é de 10μs, enquanto o tempo de escrita da memória EEPROM varia entre 3 e 10ms.

Outra alternativa para substituir este tipo de memória seria a memória FeRAM, que possui tempos de leitura e escrita bastante rápidos. Porém, é muito cara e menos segura do que a memória EEPROM e do que a memória *Flash*-EEPROM.

É possível notar que uma das maiores restrições dos *smartcards* está relacionada com os tipos de memória neles utilizados, como por exemplo, a sua limitada capacidade de armazenamento, o tempo de escrita muito lento e o tamanho reduzido desses tipos de memória.

Apesar de *smartcards* possuírem severas restrições de *hardware*, o seu uso está se tornando cada vez maior no mercado e o número de aplicações envolvendo essa tecnologia aumenta a cada dia. Um dos principais fatores que está causando esse crescimento é o fato de existirem sistemas operacionais avançados para *smartcards* que permitem a existência de múltiplas aplicações em um mesmo cartão.

2.4 Bancos de Dados Ubíquos

A necessidade por bancos de dados ubíquos aumenta a cada dia. Isto é devido principalmente ao fato de que esses tipos de bancos de dados permitem ao usuário acessar informação a qualquer hora e em qualquer lugar, independente de sua localização física [20].

Algumas propostas foram feitas no sentido de desenvolver pequenos bancos de dados para dispositivos computacionais portáteis com pouco poder de processamento e pouca memória, tais como telefones celulares, *palmtops*, Assistentes Digitais Pessoais, aplicações inteligentes e outros tipos de sistemas embutidos [20]. Os principais projetos incluem:

- *Oracle9i Lite* da *Oracle* [22];
- *Adaptive Server Anywhere* da *Sybase* [29];
- *DB2 Everyplace* da *IBM* [19];
- *SQL Server CE* da *Microsoft* [21].

Embora estes Sistemas de Gerenciamento de Bancos de Dados tenham identificado e resolvido parte do problema de adaptação das técnicas de bancos de dados já consolidadas, eles foram primordialmente desenvolvidos para computadores

portáteis e para PDAs. Mesmo possuindo um tamanho relativamente pequeno, variando entre 50 KB e 1 MB, as operações de leitura e escrita desses SGBDs utilizam uma maior quantidade de memória RAM e de memória persistente que um SGBD para *smartcard* pode utilizar. Logo, eles não resolvem as limitações mais severas desse ambiente.

A primeira tentativa em direção à construção de um Sistema de Gerenciamento de Banco de Dados para *smartcards* foi o *SQLJava Machine* [8]. Foi o primeiro Sistema de Gerenciamento de Bancos de Dados Relacional para Java a realmente caber em um *smartcard*. O *SQLJava Machine* é um pequeno mas completo SGBD (menor que 8KB) que fornece uma Linguagem de Definição de Dados (*Data Definition Language - DDL*) e uma Linguagem de Manipulação de Dados (*Data Manipulation Language - DML*) com o objetivo de definir e acessar dados relacionais persistentes em *smartcards*. Além disso, soluciona problemas de integridade referencial, suporta o conceito de transação em banco de dados, gerencia de forma econômica o espaço no *smartcard* e possui avançadas características de segurança.

Uma outra tentativa de desenvolver um SGBD para *smartcards* foi o *PicoDBMS* [4]. Atualmente, existe um protótipo do *PicoDBMS* [1] que foi desenvolvido na linguagem *JavaCard 2.1* [24] e que é executado em um simulador de *smartcard*. Ele se baseia em estruturas de dados altamente compactas e na execução de consultas sem a utilização de memória RAM. O protótipo do *PicoDBMS* é maior que o *SQLJava Machine* e possui cerca de 30KB. Ele inclui:

- ▹ Um gerenciador de armazenamento, responsável por gerenciar o armazenamento do banco de dados e dos índices associados;
- ▹ Um gerenciador de consulta, responsável por processar planos de consultas compostos de seleção, projeção, junção e agregação;
- ▹ Um gerenciador de direito de acesso, responsável por prover direitos de acesso ao banco de dados e às visões complexas definidas pelo usuário;

▹ Um gerenciador de visão, responsável por gerenciar as visões definidas pelo usuário.

Um outro sistema de banco de dados embutido para *smartcards*, chamado *GnatDB*, é proposto em [32]. Este sistema de banco de dados provê proteção tanto contra corrupção acidental de dados como contra corrupção maliciosa de dados. A principal meta do *GnatDB* é prevenir corrupção acidental de dados utilizando atualizações duráveis e atômicas. A proteção contra corrupção maliciosa de dados é feita através de técnicas que utilizam criptografia. Uma vez que seu foco principal é a segurança de dados, o *GnatDB* não implementa o processamento de consultas.

Apesar da existência dos Sistemas de Gerenciamento de Bancos de Dados descritos anteriormente, o problema de adaptar funcionalidades de banco de dados para ambientes com recursos computacionais limitados, especialmente para *smartcards*, ainda está no início de uma longa caminhada em direção ao desenvolvimento de outras implementações de SGBDs apropriados para esse tipo de ambiente e que provavelmente deverão aparecer no mercado muito em breve.

3 Processamento de Consultas

3.1 Introdução

Este capítulo faz uma revisão bibliográfica sobre processamento de consultas de uma forma geral, englobando o processamento de consultas tradicional (processamento de consultas estático) [11, 13, 16, 23], passando pelo processamento de consultas adaptativo [3, 17, 18, 30] e finalizando no processamento de consultas em ambientes computacionais com recursos limitados [1, 2, 4, 8, 32].

Sobre processamento de consultas estático, serão mostradas as fases do processamento de consultas que incluem *Parsing*, Tradução, Otimização e Avaliação, e a implementação dos operadores relacionais de seleção e junção, ou seja, os principais algoritmos que implementam esses dois operadores. Dentre os algoritmos de seleção, são apresentados os algoritmos *File Scan* e os algoritmos *Index Scan*. Dentre os algoritmos de junção, são apresentados os algoritmos *Nested-Loop Join*, *Indexed Nested-Loop Join*, *Block Nested-Loop Join*, *Merge-Join* e *Hash-Join*.

Sobre processamento de consultas adaptativo, será apresentada sua definição e serão fornecidos dois exemplos de algoritmos adaptativos: *Ripple Joins* e *XJoin*. Como o próprio nome já diz, o processamento de consultas adaptativo adapta-se ao ambiente em que a consulta está sendo executada.

Por fim, na seção sobre processamento de consultas em ambientes computacionais com recursos limitados, serão discutidas as principais limitações existentes em ambientes de *smartcards*. Será mostrada também a solução proposta por Bobineau *et al* [4] para o problema de processamento de consultas em ambientes de *smartcards*. Essa solução é baseada em estruturas de dados altamente compactas e na execução de consultas sem a utilização de memória RAM.

3.2 Processamento de Consultas Estático

3.2.1 Fases do Processamento de Consultas

Entende-se por processamento de consultas o conjunto de atividades responsáveis por realizar a extração de dados de um banco de dados. Dentre essas atividades podem ser incluídos a tradução de consultas expressas em linguagens de alto nível para outras formas de representação (como por exemplo, expressões da álgebra relacional), a otimização e a avaliação dessas consultas.

De acordo com Silberschatz *et al* [23], as principais fases envolvidas no processamento de uma consulta são:

- ▹ *Parsing* (Análise);
- ▹ Tradução;
- ▹ Otimização;
- ▹ Avaliação.

Durante a fase de *parsing*, a consulta escrita em uma linguagem de alto nível, tal como SQL, é primeiramente analisada. O *parser* analisa a sintaxe da consulta do usuário, verificando, por exemplo, se os nomes dos atributos e das relações são válidos e semanticamente significativos no esquema do banco de dados que está sendo consultado e se as referências para os atributos estão corretas quando se têm relações com o mesmo nome de atributo. Além disso, o *parser* verifica se a sintaxe da consulta do usuário está formulada de acordo com as regras de gramática (ou regras de sintaxe) da linguagem de consulta em questão. A consulta é então convertida em uma árvore de análise (*parse-tree*). Esta, por sua vez, é traduzida para uma forma de representação interna do sistema. Para sistemas de bancos de dados relacionais, geralmente essa forma de representação é uma expressão da álgebra relacional.

A partir de então, o SGBD deve planejar uma estratégia de execução para obter o resultado da consulta, isto é, selecionar um plano de avaliação (execução)

de consulta para executar a mesma. Existe uma série de estratégias de avaliação possíveis para executar uma determinada consulta. O processo de seleção da melhor estratégia de execução (do plano de avaliação mais eficiente) para processar a consulta é conhecido como otimização de consultas [11]. Planos de avaliação diferentes podem ter custos diferentes para uma dada consulta. É de inteira responsabilidade do sistema construir um plano de avaliação de consulta que minimize o custo de executar tal consulta. Portanto, a meta da otimização de consultas é exatamente fazer a seleção da melhor estratégia para processar uma consulta, ou seja, achar um plano de execução de consulta[1] que minimize a média de desempenho mais relevante, que geralmente é o número de acessos a disco. Para isso, é necessário, por exemplo, achar uma expressão da álgebra relacional equivalente à consulta dada e que conduza ao algoritmo mais eficiente.

Para decidir qual a melhor estratégia, qual o melhor plano de avaliação de consulta, o otimizador precisa estimar o custo de cada um dos possíveis planos existentes. Para isso, os otimizadores necessitam de certas informações estatísticas sobre as relações, tais como, a quantidade de tuplas da relação, o tamanho de uma tupla em *bytes*, o número de valores distintos de um certo atributo que aparece em uma relação, a altura de um índice dentre outras informações. Essas informações são fundamentais para se fazer uma boa estimativa do custo de um plano de execução e fazem parte do catálogo de dados estatísticos do banco de dados.

Existem basicamente duas técnicas principais para implementar a otimização de consultas. A primeira é baseada em regras heurísticas para ordenar as operações em uma estratégia de execução de consulta. Essas regras basicamente reordenam as operações em uma árvore de consulta. A segunda técnica envolve estimar sistematicamente o custo de estratégias de execução diferentes e escolher o plano de execução com a menor estimativa de custo. Essas duas técnicas são geralmente combinadas

[1]A saída do otimizador de consultas é chamada de plano de execução de consulta ou plano de avaliação de consulta [16].

em um otimizador de consulta [11].

Após a escolha do melhor plano de execução ter sido realizada, a consulta é avaliada com o plano escolhido e o seu resultado é então devolvido para o usuário que entrou com tal consulta. Todo este processo que foi descrito anteriormente pode ser melhor compreendido através da visualização da figura 3.1.

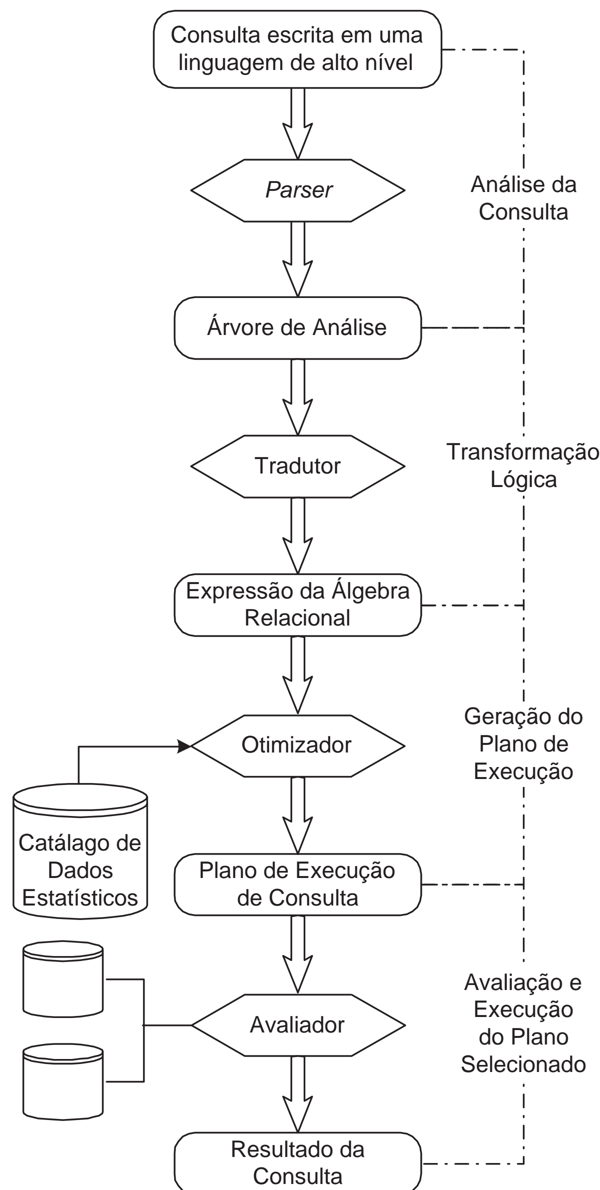

Figura 3.1. *Fases do Processamento de Consultas.*

3.2.2 Implementação dos Operadores Relacionais

Para implementar os diferentes tipos de operadores relacionais, um Sistema de Gerenciamento de Bancos de Dados deve incluir um ou mais algoritmos que geralmente são avaliados e podem aparecer na estratégia de execução da consulta [11]. Nesta seção, serão descritos os principais algoritmos usados para implementar as operações de seleção e junção. Serão mostradas também suas respectivas estimativas de custo. A medida de custo aqui utilizada é o número de páginas transferidas do disco, pois o número de acessos a disco é geralmente a medida mais importante, desde que operações de acesso a disco são bastante lentas se comparadas com operações realizadas na memória principal. Para representar a estimativa de custo desses algoritmos será utilizada a sigla **EC**.

Para fazer o cálculo de estimativas de custo, os otimizadores necessitam de certas informações estatísticas que são armazenadas no catálogo do SGBD. As informações mais relevantes sobre relações e índices para o cálculo dessas estimativas de custos incluem [23]:

- ▹ n_r (número de tuplas de uma relação r);
- ▹ p_r (número de páginas contendo as tuplas de uma relação r);
- ▹ s_r (tamanho de uma tupla da relação r em *bytes*);
- ▹ f_r (número de tuplas de uma relação r que cabem em uma página, ou seja, fator de página de uma relação r);
- ▹ $V(A, r)$ (número de valores distintos que aparecem em uma relação r para um determinado atributo A);
- ▹ $SC(A, r)$ (número médio de tuplas que satisfazem uma condição de igualdade sobre o atributo A de uma relação r, dado que pelo menos uma tupla satisfaz a condição de igualdade);
- ▹ HT_i (número de níveis da estrutura de índice i).

Para manter estas informações estatísticas precisas é necessário atualizá-las, cada vez que uma relação for modificada. Essa atualização causa uma sobrecarga substancial ao sistema e por isso muitos sistemas, ao invés de atualizarem as informações estatísticas a cada modificação nas relações, só fazem as atualizações durante períodos de pouca utilização do sistema. Como resultado, as estatísticas utilizadas para escolher uma estratégia de processamento de consulta podem não ser completamente exatas. Entretanto, se não ocorrerem muitas modificações nos intervalos entre as atualizações das informações estatísticas, essas estarão suficientemente precisas para fornecer uma boa estimativa de custo dos diferentes planos.

3.2.2.1 Operador de Seleção

Algoritmos de Seleção Básicos: Existem diversos algoritmos capazes de executar a operação de seleção. Dentre os algoritmos mais básicos podem ser destacados os algoritmos *File Scan*, algoritmos de busca que localizam e recuperam tuplas que satisfazem uma condição de seleção. Considerando uma operação de seleção onde as tuplas pertencem a uma relação armazenada em um único arquivo, dois algoritmos de busca implementam a operação de seleção. São eles:

- ▹ Busca linear;
- ▹ Busca binária.

No algoritmo de busca linear, o arquivo é lido de forma seqüencial e todas as tuplas são testadas para verificar se elas satisfazem a condição de seleção. Esse algoritmo é aplicável a qualquer arquivo, sem pressupor a existência de índices e nem a ordenação do arquivo. Como todas as páginas devem ser lidas, sua estimativa de custo é $EC = p_r$. Para uma operação de seleção sobre um atributo chave, assume-se que apenas metade das páginas será recuperada antes da tupla em questão ser encontrada. É nesse ponto que a busca pode terminar. Nesse caso, a estimativa de custo desse algoritmo cai para $EC = \frac{p_r}{2}$.

No algoritmo de busca binária, para localizar as tuplas que satisfazem uma condição de igualdade sobre um determinado atributo, o arquivo deve estar ordenado por esse atributo. Esse algoritmo é mais eficiente que o algoritmo de busca linear. Sua estimativa de custo é $EC = \lceil \log_2(p_r) \rceil + \lceil \frac{SC(A,r)}{f_r} \rceil - 1$. O primeiro termo, $\lceil \log_2(p_r) \rceil$, representa o custo de localizar a primeira página através de uma busca binária contendo tuplas que satisfazem a condição de seleção. O termo $SC(A, r)$ representa o número total de tuplas que irá satisfazer a condição de seleção. Essas tuplas irão ocupar $\lceil \frac{SC(A,r)}{f_r} \rceil$ páginas, onde uma das páginas já foi previamente recuperada. Se a condição de igualdade for sobre um atributo chave, então $SC(A, r) = 1$ e a estimativa de custo, nesse caso, cai para $EC = \lceil \log_2(p_r) \rceil$.

A estimativa de custo de uma busca binária baseia-se na suposição de que as páginas da relação estão armazenadas de forma adjacente no disco. Do contrário, o custo para consultar as estruturas de acesso ao arquivo para localizar o endereço físico de uma página deve ser adicionado à estimativa. A estimativa de custo da busca binária também depende do tamanho do resultado da seleção.

Algoritmos de Seleção que Utilizam Índices: É possível também processar consultas envolvendo seleções simples utilizando índices. Índices são caminhos através dos quais os dados podem ser localizados e acessados. É válido lembrar que um índice é dito primário quando ele possibilita que a leitura dos registros de um arquivo seja correspondente à ordem física em que eles estão armazenados no arquivo. Caso contrário ele é dito secundário.

Os algoritmos de busca que usam índices são ditos *Index Scan* [23]. Dentre eles podem ser destacados:

- ▹ Algoritmos baseados em índice primário;
- ▹ Algoritmos baseados em índice secundário.

Os algoritmos baseados em índice primário podem ter a condição de igualdade sobre um atributo do tipo chave primária (definido como índice primário) ou sobre um

atributo que não é do tipo chave primária (também definido como índice primário). No primeiro caso, o índice é usado para recuperar uma única tupla que satisfaça a condição de igualdade. Para buscar uma única tupla, é necessário recuperar uma página mais o número de níveis da estrutura de índice. Logo, sua estimativa de custo é $EC = HT_i + 1$. No segundo caso, o índice é usado para recuperar várias tuplas que satisfaçam a condição de igualdade. $SC(A, r)$ tuplas satisfarão a condição de igualdade e $\lceil \frac{SC(A,r)}{f_r} \rceil$ páginas serão acessadas. Logo, a estimativa de custo desse algoritmo é $EC = HT_i + \lceil \frac{SC(A,r)}{f_r} \rceil$. É válido lembrar que, como o índice é primário, as tuplas estão armazenadas em páginas contíguas.

Os algoritmos baseados em índice secundário também podem ter a condição de igualdade sobre um atributo do tipo chave primária (definido como índice secundário) ou sobre um atributo que não é do tipo chave primária (também definido como índice secundário). No primeiro caso, apenas uma tupla é recuperada e sua estimativa de custo é $EC = HT_i + 1$ (a mesma estimativa de custo dos algoritmos baseados em índice primário onde a condição de igualdade é sobre um atributo chave definido como índice primário). Já no segundo caso, várias tuplas podem ser recuperadas. Sendo assim, sua estimativa de custo, no pior caso, onde as tuplas estão em páginas diferentes é $EC = HT_i + SC(A, r)$.

3.2.2.2 Operador de Junção

A operação de junção é uma das operações que mais consome tempo no processamento de consultas. A seguir serão descritos os principais algoritmos que implementam a operação de junção (considerando apenas junções do tipo *equijoin*) [23].

Nested-Loop Join: Este algoritmo consiste basicamente de um par de laços *for* aninhados, onde para cada tupla t_r de uma relação r (considerando r como a relação mais externa), o algoritmo recupera cada tupla t_s de uma relação s (considerando s a relação mais interna) e verifica se as duas tuplas satisfazem a condição de junção e, caso satisfaça, inclui $t_r.t_s$ no resultado ($t_r.t_s$ denota a tupla construída pela con-

catenação dos valores dos atributos de t_r e t_s), como pode ser visto no procedimento da figura 3.2.

```
1. para cada tupla t_r em r faça
2. início
3.     para cada tupla t_s em s faça
4.     início
5.         se o par (t_r, t_s) satisfizer a condição de junção então
6.             adicione t_r.t_s ao resultado
7.     fim
8. fim
```

Figura 3.2. *Algoritmo Nested-Loop Join.*

Assim como o algoritmo de busca linear, o algoritmo *Nested-Loop Join* não pressupõe a utilização de índices e pode ser usado independente da condição de junção. Entretanto, o algoritmo *Nested-Loop Join* é um algoritmo caro, uma vez que ele examina cada par de tuplas das duas relações. O número total de pares de tuplas examinados equivale a $(n_r * n_s)$ e para cada tupla em r é necessário realizar acesso a todas as tuplas de s. Considerando que no pior caso o *buffer* armazena somente uma única página de cada relação, sua estimativa de custo será dada por $EC = p_r + (n_r * p_s)$. Assim, recomenda-se que a maior relação seja a mais externa, pois dessa forma ela só precisará ser lida uma vez. Considerando que no melhor caso existirá espaço suficiente na memória para ambas as relações, cada página será lida somente uma vez. Logo, sua estimativa de custo passará a ser dada por $EC = p_r + p_s$.

Indexed Nested-Loop Join: Em um algoritmo *Nested Loop-Join* se existir um índice no atributo de junção da relação mais interna (laço mais interno do algoritmo), as buscas seqüenciais poderão ser substituídas por buscas através desse índice. Ou seja, para cada tupla t_r de uma relação r (considerando r como a relação mais externa), o índice é usado para buscar as tuplas de uma relação s (considerando s a

relação mais interna) que satisfaçam a condição de junção com a tupla t_r. Esse algoritmo pode ser usado tanto com índices já existentes como com índices temporários criados com o único propósito de executar a junção.

A estimativa de custo desse algoritmo pode ser calculada como descrito a seguir. Para cada tupla da relação mais externa r, uma busca é desempenhada sobre o índice de s e as tuplas relevantes são recuperadas. Considerando que no pior caso só existirá espaço no *buffer* para uma página de r e para uma página de índice, p_r acessos a disco serão necessários para ler a relação r e para cada tupla t_r em r, será realizada uma busca de índice sobre s. Logo, a estimativa de custo desse algoritmo será $EC = p_r + (n_r * c)$, onde c representa o custo de uma seleção simples sobre s usando a condição de junção. Nesse algoritmo, geralmente é mais eficiente utilizar a relação menor como a relação mais externa, quando existem índices sobre as relações r e s.

Block Nested-Loop Join: Esse algoritmo é considerado uma variação do algoritmo *Nested-Loop Join*. Ele pode ser utilizado quando o *buffer* é muito pequeno para conter ambas as relações por completo na memória principal. Utilizando esse algoritmo, ao invés das relações serem processadas por tuplas, elas são processadas por blocos, tornando-o mais eficiente do que o algoritmo *Nested-Loop Join*.

Como pode ser visto no procedimento da figura 3.3, cada bloco da relação s (considerando s como a relação mais interna) é comparado com cada bloco da relação r (considerando r como a relação mais externa). Dentro de cada par de blocos, cada tupla de um bloco é comparada com cada tupla do outro bloco para gerar todos os pares de tuplas. Se o par de tuplas satisfizer a condição de junção, então ele é adicionado ao resultado. No pior caso deste algoritmo, cada bloco da relação mais interna s é lido apenas uma vez para cada bloco da relação mais externa r. Assim, a estimativa de custo é dada pela expressão $EC = p_r + (p_r * p_s)$. No melhor caso, a estimativa de custo será $EC = p_r + p_s$. Neste algoritmo, recomenda-se que a menor relação seja a mais externa.

```
1.  para cada bloco B_r da tabela r faça
2.  início
3.      para cada bloco B_s da tabela s faça
4.      início
5.          para cada tupla t_r em B_r faça
6.          início
7.              para cada tupla t_s em B_s faça
8.              início
9.                  se o par (t_r, t_s) satisfizer a condição de junção então
10.                     adicione t_r.t_s ao resultado
11.             fim
12.         fim
13.     fim
14. fim
```

Figura 3.3. *Algoritmo Block Nested-Loop Join.*

Merge-Join: Nesse algoritmo, as relações são lidas seqüencialmente e devem estar ordenadas pelo atributo de junção. Dessa forma, tuplas que possuem o mesmo valor nos atributos de junção estão em ordem consecutiva. Logo, cada tupla ordenada precisa ser lida apenas uma vez, e conseqüentemente, cada bloco também só precisa ser lido uma vez. O algoritmo *Merge-Join* pode ser visto na figura 3.4.

```
1.  ler primeira tupla de r
2.  para cada tupla t_s em s faça
3.  início
4.    enquanto (t_s[atr-juncao] >= t_r[atr-juncao]) e r não chegou ao fim faça
5.    início
6.      se o par (t_r, t_s) satisfizer a condição de junção então
7.        adicione t_r.t_s ao resultado
8.      ler próxima tupla t_r em r
9.    fim
10. fim
```

Figura 3.4. *Algoritmo Merge-Join.*

O algoritmo lê a primeira tupla t_r de uma relação r (considerando r a relação mais externa) e verifica para cada tupla t_s de uma relação s (considerando s a relação mais interna) se o atributo de junção de t_s é maior ou igual ao atributo de junção de

t_r. Se o par de tuplas satisfizer a condição de junção, ele inclui t_r e t_s no resultado. Depois lê a próxima tupla de r e repete os passos anteriormente descritos até que se tenham todas as tuplas que satisfaçam a condição de junção. A estimativa de custo deste algoritmo é dada por $EC = p_r + p_s$.

No caso em que uma das relações de entrada não esteja ordenada sobre o atributo de junção, essa relação pode ser primeiramente ordenada e posteriormente o algoritmo *Merge-Join* poderá ser aplicado. Além disso, também é possível utilizar uma variação do algoritmo *Merge-Join* sobre tuplas que não estão ordenadas se existirem índices secundários sobre os atributos de junção. As tuplas são pesquisadas através dos índices e como resultado, elas são recuperadas de forma ordenada. No entanto, essa variação acarreta um grande prejuízo na estimativa de custo do algoritmo *Merge-Join*, visto que as tuplas podem estar completamente dispersas nos blocos do arquivo. Como conseqüência, cada acesso a disco poderia envolver o acesso a um bloco de disco, o que seria extremamente caro.

Uma outra técnica chamada *Hybrid Merge-Join* combina o uso de índices com o algoritmo *Merge-Join*. Nessa técnica, se uma das relações estiver ordenada e a outra estiver desordenada, mas possuir um índice secundário baseado em árvore B^+ sobre o atributo de junção, o algoritmo *Hybrid Merge-Join* combina a relação ordenada com as entradas das folhas do índice secundário B^+. O arquivo resultante conterá tuplas da relação ordenada e endereços para tuplas da relação desordenada. Posteriormente, esse arquivo é ordenado baseado nos endereços da relação desordenada, o que permite uma recuperação eficiente das tuplas correspondentes na ordem de armazenamento física, o que completa a junção.

Hash-Join: O algoritmo *Hash-Join* utiliza uma função *hash* h para particionar as tuplas das relações, formando conjuntos com o mesmo valor *hash* para os atributos de junção. A estratégia consiste em comparar tuplas das relações r e s que pertençam a partições que possuam o mesmo valor *hash*. Após o particionamento das relações, o algoritmo realiza separadamente o algoritmo *Indexed Nested-Loop Join*, descrito

anteriormente, para cada par de partições, como pode ser visto no algoritmo da figura 3.5.

```
1.  /* Partição da tabela s */
2.  para cada tupla t_s em s faça
3.  início
4.    i := h(t_s[atr_juncao]);
5.    H_{s_i} := H_{s_i} ∪ {t_s};
6.  fim
7.  /* Partição da tabela r */
8.  para cada tupla t_r em r faça
9.  início
10.   i := h(t_r[atr_juncao])
11.   H_{r_i} := H_{r_i} ∪ {t_r}
12. fim
13. /* Junção sobre cada partição */
14. para cada i := 0 até max faça
15. início
16.   ler H_{s_i} e construir um índice hash para tuplas de H_{s_i} na memória principal
17.   para cada tupla t_r em H_{r_i} faça
18.   início
19.     utilize o índice hash sobre H_{s_i} para localizar todas as tuplas t_s
20.      onde t_s[atr_juncao] = t_r[atr_juncao]
21.     para cada tupla t_s correspondente em H_{s_i} faça
22.     início
23.       adicione t_r.t_s ao resultado
24.     fim
25.   fim
26. fim
```

Figura 3.5. *Algoritmo Hash-Join.*

Neste algoritmo, a função *hash* mapeia os valores *atr_juncao* de 0, 1, ..., até *max*. *atr_juncao* representa os atributos comuns de r e s usados na junção. H_{r_0}, H_{r_1}, ..., $H_{r_{max}}$ representam as partições das tuplas da relação r, inicialmente vazias. Cada tupla $t_r \in r$ é colocada na partição H_{r_i}, onde $i = h(t_r[atr_juncao])$. O mesmo vale para s, isto é, H_{s_0}, H_{s_1}, ..., $H_{s_{max}}$ representam as partições das tuplas da relação s, inicialmente vazias. Cada tupla $t_s \in s$ é colocada na partição H_{s_i}, onde $i = h(t_s[atr_juncao])$.

O algoritmo *Hash-Join* funciona da seguinte forma: se uma tupla t_r de r e uma tupla t_s de s satisfazem a condição de junção, então essas tuplas terão o mesmo valor

para os atributos de junção. Se a função *hash* h for executada sobre esse valor para algum valor de i, então a tupla t_r deverá está em H_{r_i} e a tupla t_s em H_{s_i}. Como conseqüência, as tuplas t_r em H_{r_i} só precisam ser comparadas com as tuplas t_s em H_{s_i}. Ou seja, as tuplas t_r não precisam ser comparadas com as tuplas t_s das outras partições.

O particionamento das duas relações r e s necessita que ambas as relações sejam completamente lidas e escritas. Logo, o custo para esse particionamento é dado por $2 * (p_r + p_s)$. Além disso, é necessário ler as partições uma vez mais para construí-las, requerendo mais $(p_r + p_s)$ acessos. O número de páginas ocupadas pelas partições pode ser um pouco maior que $(p_r + p_s)$, devido às páginas preenchidas parcialmente. O acesso a essas páginas pode levar a um custo adicional de pelo menos $(2 * max)$, onde max representa o número total de partições. Isso porque cada uma das partições pode ter uma página preenchida parcialmente que deve ser escrita e lida novamente. Assim, a estimativa de custo desse algoritmo é dada por $EC = 3 * (p_r + p_s) + (2 * max)$.

3.2.2.3 Pré-Cálculo de Junção

Levando em consideração que a operação de junção é uma das operações mais caras no processamento de consultas, a otimização do operador de junção torna-se importante nesse contexto.

Uma forma de otimizar o operador relacional de junção é pré-calcular junções entre relações armazenando cada domínio separadamente, onde cada valor do domínio associa-se com a lista de identificadores das tuplas que satisfazem uma determinada condição de junção. Esse modelo de armazenamento favorece as operações de junção em detrimento das outras operações do processamento de consultas.

Um trabalho que propõe a otimização do operador de junção utilizando pré-cálculo de junções pode ser encontrado em [31]. Para realizar o pré-cálculo de junções, [31] propõe uma estrutura de índices, chamada índice de junção, que será descrita a seguir.

Um índice de junção é uma implementação particular do conceito de *link*. *Links* são implementados através do encadeamento das tuplas das relações envolvidas na operação de junção, usando para isso os identificadores das tuplas combinados com os dados, ao invés de ponteiros. Ou seja, um índice de junção é uma relação pré-calculada armazenada separadamente dos operandos. Essa é a principal razão para o aumento da performance das junções.

Em nível de implementação, um índice de junção é uma relação de aridade 2, criado através da junção de duas relações A e B, onde cada tupla de A e cada tupla de B são unicamente identificadas por um identificador gerado pelo sistema que nunca muda. Logo, um índice de junção contém apenas pares de identificadores das relações envolvidas nas junções.

Um índice de junção deve estar fisicamente ordenado. Uma vez que pode ser necessário o acesso rápido as tuplas da relação contendo o índice de junção através dos valores de A ou dos valores de B, a relação contendo o índice de junção deve estar fisicamente ordenada sobre A e sobre B. Uma solução para esse problema é manter duas cópias da relação do índice de junção: uma ordenada sobre A e outra ordenada sobre B. Cada cópia do índice de junção é implementada como uma árvore B^+. A relação de índice de junção ordenada sobre A faz dos acessos à junção de A para B eficiente. O mesmo vale para a relação de índice de junção ordenada sobre B, que faz dos acessos à junção de B para A eficiente.

Considere a figura 3.6 que ilustra um exemplo de índice de junção. A Relação A possui um identificador único chamado A_ID, um índice sobre esse identificador e um atributo de nome ATR_AB. Já a Relação B possui um identificador único chamado B_ID, um índice sobre esse identificador e um atributo de nome ATR_AB. Para uma operação de junção entre as relações A e B sobre o atributo ATR_AB, considerando que a junção seja realizada de A para B, cria-se uma relação de índice de junção, IJ_A_ID, onde essa relação deve estar ordenada sobre o identificador A_ID. Para uma operação de junção entre as relações A e B sobre o atributo ATR_AB, considerando que a junção seja realizada de B para A, cria-se uma relação de índice

de junção, IJ_B_ID, onde essa relação deve estar ordenada sobre o identificador B_ID. Os valores dos identificadores A_ID e B_ID armazenados em IJ_A_ID e em IJ_B_ID devem satisfazer a condição de junção $A.ATR_AB = B.ATR_AB$.

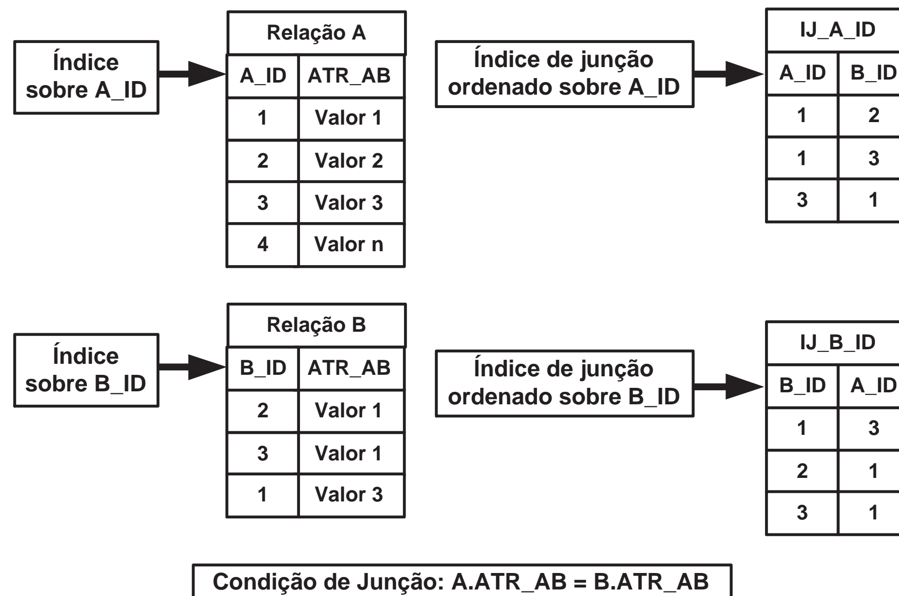

Figura 3.6. *Representação Gráfica de Índice de Junção.*

Para recuperar as tuplas da relação que satisfazem a condição de junção da Relação A em direção a Relação B, realiza-se uma junção entre a Relação A e a relação de índice de junção IJ_A_ID. Tendo o resultado dessa junção, como na relação IJ_A_ID encontram-se apenas os identificadores da relação B que satisfazem a condição de junção $A.ATR_AB = B.ATR_AB$, é necessário apenas recuperar as tuplas da Relação B contendo esses identificadores. O mesmo vale para a junção da Relação B em direção a Relação A. Mas nesse último caso, a relação de índice de junção utilizada é a relação IJ_B_ID.

3.3 Processamento de Consultas Adaptativo

Nos dias de hoje, aplicações acessam dados das mais diversificadas fontes. Essas, por sua vez, possuem tempos de resposta imprevisíveis e podem estar amplamente

distribuídas. Na Internet, por exemplo, existem muitas fontes e métodos de acesso com velocidades variáveis competindo para obter os mesmos dados. O ambiente de execução das consultas também é altamente volátil, com seus recursos variando de acordo com a carga do sistema. As condições existentes no momento de submissão de uma consulta dificilmente serão as mesmas no decorrer de seu processamento. Como resultado, as tradicionais técnicas estáticas de execução e otimização de consultas não são efetivas em tais situações [3]. Com isso, vários sistemas estão sendo projetados para se adaptarem aos seus ambientes operacionais e, com o crescimento desses sistemas, as técnicas tradicionais de processamento de consultas estão se tornando mais flexíveis e, portanto, mais adaptativas.

Segundo Hellerstein [18], um mecanismo de processamento de consultas é dito adaptativo quando ele:

- Recebe informação do ambiente. Essa informação contextualiza tal ambiente no momento de execução da consulta;
- Usa a informação do ambiente para determinar seu comportamento;
- Repete esse processo ao longo do tempo, gerando uma relação de *feedback* entre ambiente e comportamento. Uma relação de feedback pode ser definida como um retorno que se obtém a partir de uma informação fornecida. No contexto em questão, o ambiente fornece informação que é utilizada pelo sistema para alterar o seu comportamento. A alteração do comportamento é o retorno obtido com a utilização da informação fornecida pelo ambiente. O *feedback* envolvido em um sistema adaptativo é a chave para a sua eficácia pois permite ao sistema tomar múltiplas decisões e observar o resultado das mesmas.

Em outras palavras, um sistema de processamento de consultas adaptativo é um sistema que muda seu comportamento em resposta a uma mudança do ambiente a fim de atingir uma determinada meta. Quanto mais imprevisível for o ambiente maior é a necessidade de se utilizar técnicas de processamento de consultas adaptativo.

É importante salientar as diferenças existentes entre processamento de consulta estático e processamento de consulta adaptativo. No primeiro, o objetivo é achar um plano de consulta ótimo e depois executá-lo de forma estática, ao passo que no segundo, planos de consultas podem ser modificados em tempo de execução, em resposta a mudanças do ambiente. Uma outra diferença é que no processamento de consultas estático não existe mecanismo de *feedback*, enquanto que no processamento de consultas adaptativo esse mecanismo existe e é usado durante a execução da consulta.

Quando se fala em adaptabilidade em sistemas adaptativos, existem três aspectos importantes que devem ser levados em consideração [18]: a freqüência, os efeitos e a extensão da adaptabilidade. A freqüência da adaptabilidade é relativa à freqüência com que o sistema pode receber informação e mudar seu comportamento. Os efeitos da adaptabilidade estão relacionados com qual comportamento o sistema pode mudar. A extensão da adaptabilidade está relacionada com a duração da relação de *feedback*, ou seja, o tempo que essa relação pode ser mantida.

A noção de adaptabilidade chegou a ser incorporada em alguns processadores de consultas mais antigos, entretanto ela ainda é considerada um desafio para a comunidade de banco de dados. A seguir, serão descritos dois exemplos de algoritmos de processamento de consultas que implementam a operação de junção, utilizando a técnica de *pipelining* e que são adaptativos: o *Ripple Join* e o *Xjoin.*

3.3.1 Ripple Joins

Embora consultas de suporte à decisão (consultas *ad hoc*) estejam se tornando extremamente importantes em diversas aplicações, os Sistemas de Gerenciamento de Bancos de Dados Relacionais atuais não possuem um tratamento especial para tais consultas. Esses sistemas processam consultas *ad hoc* de forma que os usuários são forçados a esperar durante muito tempo, sem qualquer tipo de *feedback*, até que uma resposta precisa seja retornada. Uma solução para esse problema seria desempenhar agregações *online*, onde resultados parciais da consulta em execução são refinados

progressivamente e continuamente mostrados para o usuário.

Nesse contexto, surgiram os algoritmos *Ripple Joins*. Eles constituem uma nova família de algoritmos de junção e foram projetados inicialmente para o processamento de consultas de agregação *online* sobre múltiplas tabelas em SGBDs Relacionais [17]. Por serem adaptativos, sua aplicabilidade no contexto de *smartcards* deve ser levada em consideração.

Enquanto os algoritmos de junção tradicionais são projetados para minimizar o tempo de finalização da consulta, os algoritmos *Ripple Joins* são projetados para minimizar o tempo necessário até que uma estimativa precisa e aceitável do resultado da consulta esteja disponível.

Ripple Joins ajustam seu comportamento de acordo com as propriedades estatísticas dos dados durante o processamento da consulta. Eles permitem que as estimativas de uma consulta em execução sejam atualizadas de forma estável e contínua. Além disso, eles também permitem que o usuário controle a taxa com que essas atualizações ocorrem.

Um algoritmo *Ripple Join* pode ser visto como uma generalização do algoritmo *Nested-Loop Join*, onde os papéis tradicionais da relação mais interna e da relação mais externa são continuamente trocados durante o processamento da consulta [17].

Assim como o algoritmo *Nested-Loop Join*, o algoritmo *Ripple Join* também possui algumas variantes, sendo elas:

- ▹ *Block Ripple Join*, onde ao invés de ser lida uma tupla ou uma página de disco da relação, são lidos grandes blocos de páginas;
- ▹ *Index Ripple Join*, onde a junção é feita através de índices. Esse algoritmo é idêntico ao *Index Nested-Loop Join*;
- ▹ *Hash Ripple Join*, onde duas tabelas *hash* são materializadas na memória (uma para cada relação envolvida na junção). Esse algoritmo falha quando as tabelas *hash* não cabem mais na memória. Quando isso ocorre, ele pode ser convertido para o algoritmo *Block Ripple Join*.

A grande desvantagem dos algoritmos *Ripple Joins* é que eles utilizam muitos recursos de sistema, o que impossibilita sua adaptação no contexto de *smartcards*.

3.3.2 XJoin

A existência de múltiplas fontes de dados em áreas geograficamente distribuídas causa sérios problemas de *performance* às tradicionais técnicas de processamento de consultas. O acesso a dados em ambientes como a Internet envolve um grande número de fontes remotas, *sites* intermediários e *links* de comunicação, os quais são vulneráveis a sobrecargas, congestionamentos e falhas. Tais problemas podem causar atrasos significativos e imprevisíveis no acesso às informações das fontes remotas.

Para lidar com esses atrasos na chegada dos dados e produzir resultados iniciais mais rapidamente, foi criado um operador de junção conhecido como *XJoin*. O *XJoin* se baseia em dois princípios fundamentais [30]:

- ▹ É projetado para produzir resultados incrementalmente à medida que eles se tornam disponíveis;
- ▹ Permite que algum tipo de progresso na execução seja feito mesmo quando uma ou mais fontes de dados atrasam.

Além disso, o *XJoin* é baseado no algoritmo *Symmetric Hash-Join*. Esse, por sua vez, foi inicialmente projetado para permitir um alto grau de *pipelining* em sistemas de bancos de dados paralelos. Entretanto, durante a maior parte da execução da consulta, o *Symmetric Hash-Join* requer que as tabelas *hash* de ambas as suas entradas sejam mantidas na memória principal. Conseqüentemente, ele não pode ser usado na execução de junções que possuam entradas grandes. Pelo mesmo motivo, sua habilidade para executar múltiplas junções também é bastante restrita. Logo, o *XJoin* veio tentar resolver tais limitações. Ele estende o algoritmo *Symmetric Hash-Join*, permitindo que partes das tabelas *hash* sejam movidas para a memória secundária, utilizando dessa forma, uma menor quantidade de memória principal. Ele faz isso através do particionamento de suas entradas.

Um componente fundamental do algoritmo *XJoin* é um processo de *background* iniciado quando as entradas estão atrasadas. Esse processo utiliza convenientemente os atrasos no recebimento dos dados das fontes remotas para produzir uma maior quantidade de tuplas mais rapidamente. Além desse processo de *background*, o *XJoin* possui três estágios, cada qual sendo executado por um processo separado [30].

O primeiro e o segundo estágios são executados de forma intercalada. O primeiro estágio faz a junção das tuplas residentes na memória, agindo de forma similar ao algoritmo *Symmetric Hash-Join*. A principal diferença é que no *XJoin* as tuplas são organizadas em partições. Em geral, cada partição consiste de uma porção residente na memória e uma porção residente no disco. A porção que reside na memória armazena as tuplas que chegaram recentemente naquela partição. A porção que reside no disco contém as tuplas da partição que foram enviadas para o disco devido a restrições de memória. Se existir memória disponível para uma tupla de entrada que acabe de chegar de uma fonte de dados, então ela é colocada em uma partição e usada para testar a porção residente na memória da partição correspondente da outra fonte. Do contrário, uma das partições é escolhida e suas tuplas residentes na memória são enviadas para o disco. Se o primeiro estágio atingir o limite de tempo em ambas as suas entradas por causa de atrasos inesperados, por exemplo, ele é bloqueado e o segundo estágio começa a ser executado. O primeiro estágio termina quando todas as tuplas de suas entradas forem recebidas.

O segundo estágio faz a junção das tuplas que foram enviadas para o disco devido a restrições de memória. Ele é ativado sempre que o primeiro estágio ficar bloqueado devido à falta de dados de entrada. Ele testa as tuplas da porção residente em disco de uma partição com as tuplas residentes em memória da partição correspondente da outra fonte. Se alguma correspondência for encontrada, as tuplas correspondentes são colocadas no resultado final. Após uma porção residente no disco ter sido completamente processada, o operador verifica se uma das entradas voltou a produzir tuplas. Se sim, o segundo estágio pára e o primeiro estágio é reiniciado. Caso contrário, uma porção residente no disco diferente é escolhida e o segundo estágio

continua.

O terceiro estágio é um estágio de limpeza. Ele começa depois que todas as tuplas de ambas as entradas tiverem sido recebidas. Esse estágio garante que todas as tuplas que devem estar no resultado final foram produzidas. Esse passo é necessário porque o primeiro e o segundo estágios apenas computam o resultado parcialmente.

O segundo e o terceiro estágios do *XJoin* podem gerar tuplas duplicadas. Isso é devido ao fato deles poderem desempenhar trabalho de sobreposição. Para resolver esse problema, o *XJoin* utiliza um mecanismo de prevenção de duplicatas baseado em *timestamps*.

Com base no que foi descrito, é válido salientar que os maiores desafios no desenvolvimento do operador *XJoin* incluem:

- ▹ O gerenciamento do fluxo de tuplas entre memória principal e memória secundária;
- ▹ O controle do processo de *background* que é iniciado quando as entradas estão atrasadas;
- ▹ A garantia de que o resultado completo é produzido, isto é, nenhum resultado parcial deve ser perdido;
- ▹ A garantia de que nenhuma tupla duplicada é produzida inadvertidamente.

3.4 Processamento de Consultas em Ambientes Computacionais Limitados

3.4.1 Principais Limitações

Embora tenham ocorrido muitos avanços tecnológicos na área de *smartcards*, esse tipo de ambiente ainda possui muitas limitações, especialmente limitações de *hardware* devido ao seu pequeno tamanho. Dentre elas:

- ▹ Operações de escrita extremamente lentas;

▹ Limitada capacidade de armazenamento;

▹ Tamanho reduzido de sua memória RAM;

▹ Baixa autonomia de bateria.

Além disso, ainda devem ser levados em consideração a falta de um mecanismo que possa gerenciar os dados de maneira eficiente e o alto nível de segurança que deve ser mantido em todas as situações.

Atualmente o chip de um *smartcard* integra uma unidade de microprocessamento (CPU), uma memória volátil estática (SRAM), uma memória não volátil (ROM), uma memória para armazenar dados persistentes (EEPROM ou *Flash*-EEPROM), um co-processador opcional dedicado à criptografia e um Gerador de Número Randômico (*Random Number Generator* - RNG) para chaves criptográficas [14].

A unidade de microprocessamento, local onde a informação é processada e onde os cálculos são realizados, possui um barramento de dados de 8 bits. Algumas aplicações que utilizam criptografia necessitam de um maior poder computacional e uma maior capacidade de memória. Para isso, unidades de processamentos de 16 e 32 bits já estão sendo introduzidas no mercado.

A memória volátil estática (SRAM) é o local onde o sistema operacional e as aplicações armazenam dados temporários. É a memória de trabalho do *smartcard*. Sua capacidade hoje varia entre 256 bytes e 2 Kbytes. Aplicações com maior poder computacional requerem uma capacidade de até 4 kbytes [14].

A memória ROM é a memória permanente do cartão. Ela contém as camadas de baixo nível do sistema operacional além do seu conjunto de instruções. Sua capacidade varia entre 8 Kbytes e 64 Kbytes. Para sistemas operacionais mais complexos, que suportam múltiplas aplicações em um mesmo cartão, é necessária uma memória ROM com uma capacidade maior que 64 kbytes.

A memória EEPROM é utilizada para armazenar dados persistentes como por exemplo, armazenar dados em uma área protegida do *smartcard* ou, até mesmo, armazenar uma parte do sistema operacional que poderá ser modificada durante a

vida do chip. Sua capacidade hoje varia de 8 kbytes até 32 kbytes. Novas aplicações já estão necessitando de uma maior quantidade de memória EEPROM, que excede 64 kbytes. Pelo fato da memória *Flash*-EEPROM possuir uma velocidade maior nas operações de escrita do que a memória EEPROM, cada vez mais ela está substituindo esse tipo de memória.

Com os dados acima, é possível notar como os recursos de um *smartcard* são reduzidos e como as aplicações que estão surgindo exigem cada vez mais recursos desses ambientes.

3.4.2 Processamento de Consultas em *Smartcards*

As tradicionais técnicas de processamento de consultas geralmente fazem uso de grande quantidade de memória principal para armazenar estruturas de dados temporárias e resultados intermediários. Além disso, essas técnicas materializam os resultados intermediários no disco para prevenir um *overflow* de memória quando não existe espaço suficiente disponível na memória principal para processar uma dada consulta [4]. Tal estratégia não pode ser aplicada para processar consultas em ambientes computacionais com recursos limitados como *smartcards* pois ela consome muitos recursos de armazenamento, além de serem muito complexas para serem utilizadas nesse tipo de ambiente. Em se tratando de *smartcards*, as técnicas de processamento de consultas devem ser simples, seguras e devem consumir a menor quantidade de recursos de armazenamento possível.

A seguir, será discutida a abordagem proposta por Bobineau *et al* [4] para resolver o problema de processamento de consultas em ambientes com recursos computacionais limitados, particularmente em *smartcards*. Similar à abordagem proposta nesta dissertação, a solução apresentada em [4] é baseada em estruturas de dados altamente compactas. Além disso, o seu plano de execução de consulta é baseado em uma árvore de extrema profundidade à direita (*extreme right-deep tree)*, onde todos os operadores utilizam a técnica de *pipelining*. Nesse tipo de árvore, os operadores à esquerda são sempre relações base, as quais já estão materializadas na memória

estável. Os autores afirmam que a estratégia de processamento de consultas proposta para o PicoDBMS evita consumo de memória RAM durante a execução de um plano de consulta. Entretanto, de forma a construir uma árvore de extrema profundidade à direita, o processador de consultas do PicoDBMS viola uma importante heurística para a otimização de consultas relacionais. Essa heurística especifica que os operadores de seleção devem ser executados antes das operações de junção em um plano de execução de consultas. Portanto, a estratégia de processamento de consultas do PicoDBMS tende a ser dispendiosa e a manipular uma maior quantidade de dados, consumindo muito tempo no processamento dos dados, propriedades negativas para o processamento de consultas em *smartcards*.

O processador de consultas do PicoDBMS implementa o Modelo Iterador para processar os operadores relacionais. De acordo com esse modelo, cada operador é modelado como um iterador, que suporta três chamadas de procedimentos:

- ▹ *Open*;
- ▹ *Next*;
- ▹ *Close*.

A chamada de procedimento *open* prepara o operador para produzir um item. A chamada de procedimento *next* produz um item. Finalmente, a chamada de procedimento *close* encerra a iteração. Usando esse modelo, uma execução em *pipeline* pode ser feita da seguinte maneira: um plano de execução de consulta é ativado iniciando na raiz da árvore do operador, prosseguindo em direção às folhas. À medida que os dados são necessários, um operador filho passa uma tupla para o seu nó pai em resposta a uma chamada do procedimento *next* desse nó pai [16].

Nesse contexto, as operações básicas (seleção, projeção e junção) podem ser executadas tanto seqüencialmente como através do uso de índices. O uso de índices nas árvores de extrema profundidade à direita aumenta a *performance* de uma seleção simples sobre a primeira relação base. Entretanto, o uso de índices em outros atributos selecionados que não sejam na primeira relação base pode tornar a execução

da seleção lenta. Como nenhuma materialização ocorre, operadores de projeção são empurrados para o topo da árvore. Já no caso das operações de junção, se não forem utilizados índices, elas são feitas por algoritmos *Nested-Loop Join*, já que nenhuma outra técnica de junção pode ser aplicada sem usar estruturas *ad-hoc* ou sem usar área de trabalho.

Já as operações complexas (agregação, ordenação e remoção de duplicatas) não são compatíveis com a técnica de *pipelining*, uma vez que esses operadores necessitam materializar resultados intermediários. Entretanto, a materialização não pode ocorrer em *smartcards* devido à limitação da memória RAM, nem nas leitoras de *smartcards* devido à questão da segurança. Para resolver esse problema, Bobineau *et al* [4] propôs uma solução baseada em duas propriedades. A primeira propriedade diz que se as tuplas de entrada já estiverem agrupadas por valores distintos, as operações de agregação e remoção de duplicatas podem ser feitas em *pipeline*. A segunda propriedade diz que os operadores em *pipeline* preservam a ordem desde que eles consomem e produzem tuplas na ordem de chegada. Então, se uma ordem de consumo adequada for garantida na folha da árvore de execução, é possível usar a técnica de *pipelining* com os operadores de agregação e remoção de duplicatas. É importante mencionar que a operação de ordenação pode ser feita na leitora de *smartcard* porque a ordem de saída das tuplas resultantes não é importante.

Também é importante mencionar sobre o modelo de armazenamento utilizado no PicoDBMS, desde que está se lidando com ambientes extremamente compactos. Ele usa três modelos de armazenamento. Aqui somente será explicado o modelo de armazenamento de domínio, tendo em vista que o presente trabalho é baseado nessa estrutura. Esse modelo será descrito na seção seguinte.

As técnicas de processamento de consultas propostas por [4] foram avaliadas através do desempenho do modelo de execução proposto, com uma implementação de um processador de consultas (*query engine*) em dois computadores configurados, de forma que ficassem similares à arquitetura de um *smartcard*. Eles mostraram que o desempenho resultante dessa análise se adequou aos requisitos de uma aplicação

de *smartcard*.

3.4.3 Modelo de Armazenamento de Domínio

Devido à capacidade reduzida de recursos existente em ambientes de *smartcards*, a solução proposta em [4] se baseia em estruturas de dados altamente compactas. Para isso, são utilizados três modelos de armazenamento. Um dos modelos utilizados é o modelo de armazenamento de domínio, o qual será explicado com mais detalhes a seguir.

No modelo de armazenamento de domínio ilustrado na figura 3.7, como o próprio nome já diz, os valores são agrupados em domínios (conjuntos de valores únicos). As tuplas referenciam seus valores de atributos por meio de ponteiros. Para diminuir o custo adicional de se ter uma estrutura a mais para armazenar valores agrupados associado com esse modelo, somente são armazenados por domínio os atributos grandes contendo duplicatas. Considere atributos grandes como sendo atributos maiores que o tamanho de um ponteiro. Atributos grandes de tamanho variável mesmo que não contenham duplicatas também podem ser armazenados em domínios de forma eficiente. Nesse caso, todas as tuplas das relações tornam-se de tamanho fixo.

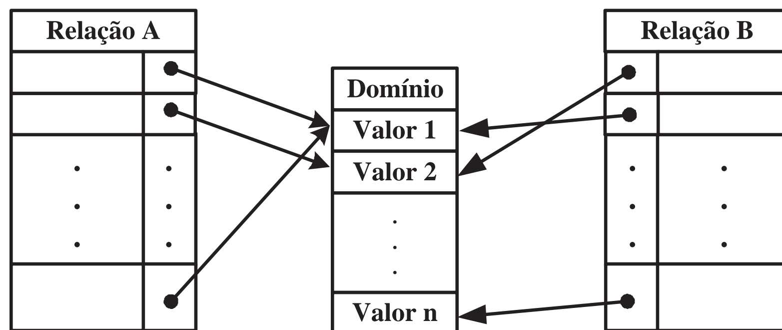

Figura 3.7. *Modelo de Armazenamento de Domínio*

Uma das principais vantagens desse modelo é que atributos distintos podem compartilhar um mesmo domínio. Além disso, esse modelo garante a compactação dos

dados e dos índices. Essas características são importantes no contexto de *smartcards* já que são ambientes com capacidades de processamento e memória reduzidas.

4 MSJoin e MSJoin$^+$: Garantindo Suporte para Otimização do Operador de Junção

4.1 Introdução

Com o objetivo de otimizar a execução do operador relacional de junção em ambientes computacionais com recursos limitados, neste capítulo, propõe-se duas estruturas de armazenamento de dados, chamadas *MSJoin* e *MSJoin*$^+$, baseadas na estrutura de armazenamento de domínio apresentada em [4]. A abordagem de pré-cálculo de junções deste trabalho é diferente da abordagem de pré-cálculo proposta em [31]. A idéia da abordagem proposta é usar essas estruturas para garantir a execução do operador de junção através do algoritmo *Merge-Join*. Esse algoritmo possui a menor estimativa de custo para a execução de operações de junção. A estimativa de custo do algoritmo *Merge-Join* para uma operação de junção sobre duas relações r e s é $p_r + p_s$, onde p_r representa o número de páginas de r e p_s representa o número de páginas de s.

Além de garantir a execução da operação de junção através do algoritmo *Merge-Join*, a estrutura de armazenamento *MSJoin*$^+$ tem a propriedade de otimizar o espaço de armazenamento, mantendo ainda uma execução satisfatória da operação de junção.

Nas próximas seções, as estruturas *MSJoin* e *MSJoin*$^+$ serão descritas e analisadas. Será ilustrada também a aplicabilidade da abordagem proposta através do exemplo de uma aplicação para *smartcards*. Entretanto, antes de apresentar essas duas estruturas, será feita uma descrição da estrutura de índices baseada em árvore B$^+$, visto que a implementação de tais estruturas é baseada nesse tipo de árvore.

4.2 Árvores B$^+$

A estrutura de índices baseada em árvores B$^+$ é uma das estruturas mais utilizadas dentre as diversas estruturas de índices existentes, pelo fato de uma árvore B$^+$ manter sua eficiência independente da freqüência de operações de inserção e remoção. Uma das principais vantagens dessa estrutura é que ela se reorganiza automaticamente através de pequenas operações quando ocorrem inserções e remoções de valores no arquivo. Logo, um arquivo baseado em árvore B$^+$ não precisa ser completamente reorganizado para manter o desempenho da estrutura da árvore B$^+$ após a ocorrência dessas operações (inserção e remoção).

Uma árvore B$^+$ é uma árvore balanceada, onde todos os caminhos de acesso, da raiz até qualquer folha da árvore, são do mesmo tamanho, ou seja, a altura da árvore é constante. Aliás, o fato de uma árvore B$^+$ ser uma árvore balanceada garante um bom desempenho para seus algoritmos de busca, inserção e remoção.

Um típico nó de uma árvore B$^+$ contém n ponteiros Pt_1, Pt_2, ..., Pt_n e $(n-1)$ valores de chave de busca K_1, K_2, ..., K_{n-1}. Dessa forma, a ordem n da árvore é representada pela quantidade de ponteiros que ela pode ter, sejam ponteiros de árvore (ponteiros apontando para alguma sub-árvore da árvore B$^+$), sejam ponteiros de dados (ponteiros que apontam para um registro de arquivo cujo valor de chave de busca é igual a K_i ou para um bloco (página) de ponteiros, cada qual apontando para um registro de arquivo cujo valor de chave de busca é igual a K_i[1]). Ou seja, se uma árvore tem 4 ponteiros de árvore (ou 4 ponteiros de dados), isso significa que a ordem da árvore é 4 e que ela só pode ter 3 valores de chave de busca.

Em uma árvore B$^+$, um nó está em um determinado nível i se existirem i nós internos (excluindo o nó propriamente dito) entre a raiz e o nó. O nó raiz é considerado o nível 0 da árvore B$^+$. A altura de uma árvore B$^+$ é representada pelo nível máximo de nós na árvore mais um. Por exemplo, uma árvore de altura igual a 2

[1]A estrutura de bloco somente é usada se a chave de busca não formar uma chave primária.

possui dois níveis: Nível 0 e nível 1.

Os ponteiros de dados de uma árvore B^+ são armazenados apenas nos nós folhas. Conseqüentemente, a estrutura dos nós folhas é diferente da estrutura dos nós internos. Para permitir o acesso ordenado aos valores de chave de busca, os nós folhas de uma árvore B^+ são ligados por meio de ponteiros. Alguns valores de chave de busca são repetidos nos nós internos da árvore a fim de guiar a busca dos valores que estão nos nós folhas. Serão apresentadas, a seguir, as principais diferenças entre os nós internos e os nós folhas de uma árvore B^+[4, 11].

A estrutura de um nó interno de uma árvore B^+ de ordem n possui as seguintes características:

- ▹ Cada nó interno possui a seguinte estrutura: $< Pt_1$, K_1, Pt_2, K_2, ..., Pt_{m-1}, K_{m-1}, $Pt_m >$, onde Pt_i é um ponteiro de árvore, K_i é um valor de chave de busca e $m \leq n$;

- ▹ Dentro de cada nó interno, $K_1 < K_2 < ... < K_{m-1}$. Isso significa que os valores de chave de busca dos nós internos são mantidos de forma ordenada;

- ▹ Para todo valor X de chave de busca na sub-árvore apontada pelo ponteiro Pt_i, tem-se que: $K_{i-1} < X \leq K_i$, para $1 < i < m$; $X \leq K_i$, para $i = 1$; e $K_{i-1} < X$ para $i = m$. Isto significa que, para $i = 1$, o ponteiro Pt_1 aponta para uma sub-árvore que contém valores de chave de busca X menores ou iguais a K_1. Para $1 < i < m$, o ponteiro Pt_i aponta para uma sub-árvore que contém valores de chave de busca X menores ou iguais a K_i e maiores que K_{i-1}. E por último, para $i = m$, o ponteiro Pt_m aponta para uma sub-árvore que contém valores de chave de busca X maiores que K_{m-1};

- ▹ O número máximo de ponteiros de árvore que um nó interno pode ter é n (ordem da árvore);

- ▹ Cada nó interno, com exceção do nó raiz da árvore, tem pelo menos $\lceil \frac{n}{2} \rceil$ ponteiros de árvore. Se a raiz da árvore for um nó interno, ela tem pelo menos

dois ponteiros de árvore;

▹ Um nó interno com m ponteiros de árvore, onde $m \leq n$, tem $m - 1$ valores de chave de busca.

É possível ver a estrutura de um nó interno de uma árvore B$^+$ na figura 4.1.

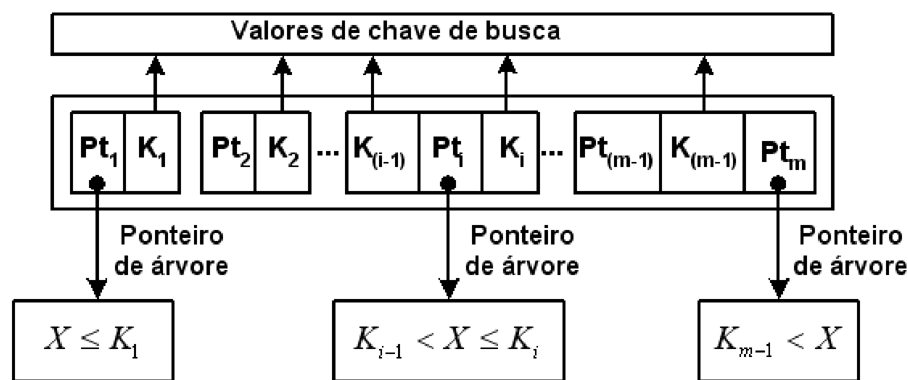

Figura 4.1. *Estrutura de um nó interno de uma árvore B$^+$.*

Já a estrutura de um nó folha de uma árvore B$^+$ possui as seguintes características:

▹ Cada nó folha possui a seguinte forma: $<< K_1, Pt_1 >, < K_2, Pt_2 >, ..., < K_{m-1}, Pt_{m-1} >, < Pt_m >>$, onde Pt_i é um ponteiro de dados, Pt_m representa um ponteiro que aponta para a próxima folha da árvore B$^+$, K_i é um valor de chave de busca e $m \leq n$;

▹ Dentro de cada nó folha, $K_1 < K_2 < ... < K_{m-1}$, $m \leq n$;

▹ Cada nó folha tem pelo menos $\lceil \frac{n-1}{2} \rceil$ valores. Se a raiz da árvore for um nó folha (isto é, se não existirem outros nós na árvore), ela tem entre 0 e $(n - 1)$ valores de chave de busca;

▹ O número máximo de valores de chave de busca que um nó folha pode ter é $(n - 1)$, como já foi dito anteriormente;

▹ Todos os nós folhas estão no mesmo nível.

É possível ver a estrutura de um nó folha de uma árvore B$^+$ na figura 4.2.

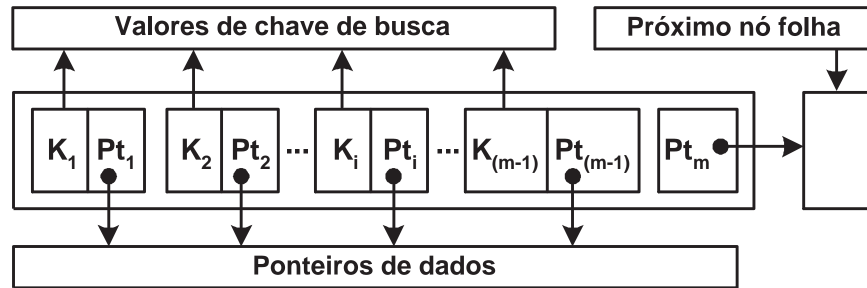

Figura 4.2. *Estrutura de um nó folha de uma árvore B$^+$.*

Como os nós folhas de uma árvore B$^+$ são ordenados pelos valores de chave de busca, a existência de um ponteiro que aponta para o próximo nó folha da árvore permite o acesso seqüencial dos valores armazenados na árvore de maneira eficiente. Conseqüentemente, para recuperar esses valores armazenados é necessário apenas acessar a raiz da árvore, descer até o nó folha mais à esquerda da árvore e retornar os valores fazendo uso do acesso seqüencial. Esses fatores foram determinantes na escolha de árvore B$^+$ como estrutura de indexação a ser utilizada nas estruturas que serão propostas nas próximas seções.

4.3 Estrutura MSJoin

A forma como os dados estão fisicamente organizados em ambientes computacionais com capacidades de processamento e memória restritas para o gerenciamento dos dados é de fundamental importância para se ter um processamento de consultas eficiente nesses ambientes. Como o espaço de armazenamento dos dados é bastante limitado em *smartcards*, mecanismos efetivos de armazenamento de dados podem utilizar o espaço disponível de maneira eficiente. Além disso, como a capacidade de processamento é muito restrita, mecanismos para se ter um processamento de

consultas eficiente também devem ser levados em consideração em ambientes computacionais limitados.

Nesse contexto, propõe-se a estrutura *MSJoin* que estende a estrutura de armazenamento de domínio proposta em [4] (apresentada no capítulo 3 da presente dissertação) e que é a evolução da estrutura de armazenamento apresentada em [6]. Na estrutura de armazenamento proposta, atributos de junção são agrupados em domínios. Dessa forma, a estrutura *MSJoin* evita a ocorrência de valores duplicados para o atributo de junção. Entretanto, enquanto [4] tem como meta otimizar o espaço de armazenamento, além de utilizar um algoritmo *Nested-Loop Join* para executar operações de junção, a estrutura *MSJoin* tem como principal objetivo otimizar a execução de junções, utilizando para isso o algoritmo *Merge-Join*.

Em contraste com a estrutura de armazenamento de domínio, onde as tuplas das relações referenciam seus valores de atributos por meio de ponteiros, a estrutura *MSJoin* possui ponteiros que referenciam as tuplas de relações que satisfazem uma condição de junção. A idéia é ter uma estrutura *MSJoin* para cada operação de junção.

Considere a figura 4.3, que ilustra a estrutura de armazenamento *MSJoin*. A Relação A possui um atributo chamado COD_A. Esse atributo também está presente na Relação B. Para uma operação de junção entre as relações A e B sobre o atributo COD_A, deve-se armazenar o atributo de junção COD_A na estrutura *MSJoin*. Os valores do atributo COD_A armazenados na estrutura *MSJoin* devem satisfazer a seguinte condição de junção: $A.COD_A = B.COD_A$. Se algum valor do atributo COD_A da Relação A ou do atributo COD_A da Relação B não satisfizer a condição de junção, esses valores não estarão armazenados na estrutura *MSJoin*. Dessa forma, garante-se que ao acessar a estrutura *MSJoin*, somente serão recuperadas as tuplas das relações A e B que satisfaçam a condição de junção $A.COD_A = B.COD_A$. Não é permitida a existência de valores repetidos do atributo COD_A na estrutura *MSJoin*. Além disso, a estrutura *MSJoin* possui ponteiros para as tuplas das relações A e B que satisfazem a condição de

junção acima mencionada. Por exemplo, se na estrutura *MSJoin* existir uma tupla onde *COD_A = "Valor 2"*, esta tupla deverá conter ponteiros para todas as tuplas da Relação *A*, onde *A.COD_A = "Valor 2"* e ponteiros para todas as tuplas da Relação *B*, onde *B.COD_A = "Valor 2"*, como pode ser visto na figura 4.3.

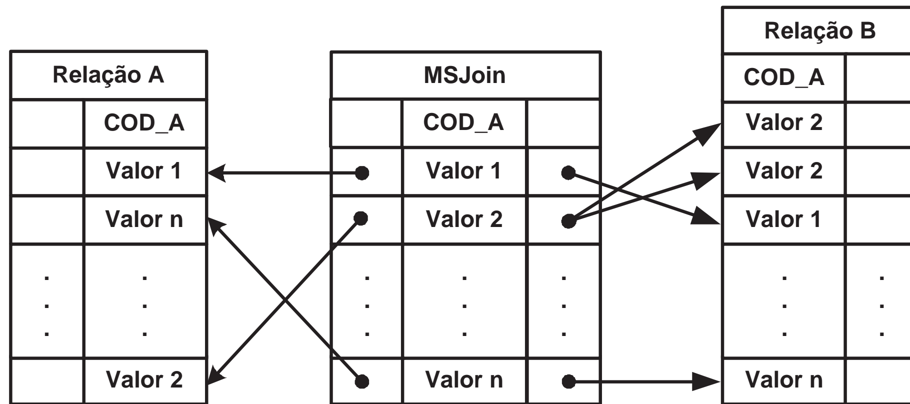

Figura 4.3. *Estrutura MSJoin.*

Para executar uma operação de junção usando a abordagem proposta, o mecanismo de processamento de consultas tem apenas que localizar a estrutura *MSJoin* e, através dos ponteiros para as tuplas que satisfazem a condição de junção, retornar tais tuplas. Se as entradas da estrutura *MSJoin* estiverem ordenadas pelo atributo de junção, o algoritmo *Merge-Join* pode ser aplicado para processar a operação de junção. É válido salientar que o algoritmo *Merge-Join* apresenta a menor estimativa de custo para a operação de junção, além de suportar o uso da técnica de *pipelining*, evitando assim a materialização de resultados intermediários para executar a operação de junção.

De forma a manter as entradas da estrutura *MSJoin* ordenadas pelo atributo de junção, essa estrutura é implementada como uma extensão de árvores B^+. Isto é devido ao fato de que árvores B^+ garantem a ordenação das entradas em cada nó.

Em uma árvore B^+, ponteiros de dados são armazenados apenas nos nós folhas. Por essa razão, a estrutura dos nós folhas é diferente da estrutura dos nós internos

[11]. A folha de uma árvore B^+ tem para cada valor de chave de busca um ponteiro de dados para o registro ou para o bloco que contém o registro. Para garantir o acesso ordenado dos valores de chave de busca para os registros, os nós folhas de uma árvore B^+ são ligados.

A extensão de árvore B^+ proposta trabalha de forma similar às tradicionais árvores B^+. Entretanto, na abordagem proposta, para cada valor de chave de busca existem dois ponteiros de dados, cada qual apontando para as tuplas das relações envolvidas na operação de junção. A chave de busca da árvore B^+ estendida representa o atributo de junção. Por exemplo, suponha que se tenha um valor de chave de busca $K_1 = 7$ (atributo de junção). Na árvore B^+ estendida, tem-se dois ponteiros Pt_{11} e Pt_{12} que referenciam tuplas nas duas relações r e s, respectivamente, que satisfazem uma determinada condição de junção. Em outras palavras, Pt_{11} e Pt_{12} apontam para tuplas nas relações r e s que contém o atributo de junção $K_1 = 7$. A figura 4.4 mostra a extensão introduzida no conceito de árvores B^+ para implementar a estrutura *MSJoin*.

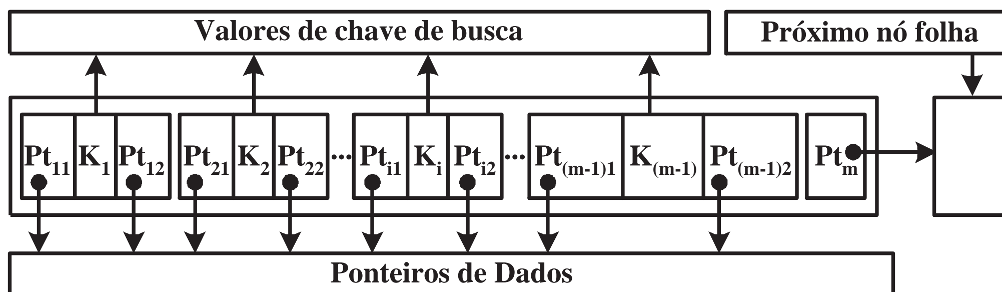

Figura 4.4. *Nó folha de uma árvore B^+ estendida.*

No pior caso, o relacionamento entre duas relações r e s é 1:1. Isso significa que para cada tupla da relação r existe um único correspondente na relação s e para cada tupla da relação s existe um único correspondente na relação r. Dessa forma, o número de tuplas da relação r é igual ao número de tuplas da relação s e, conseqüentemente, igual ao número de entradas armazenadas na estrutura *MSJoin*. Logo, o

custo estimado do algoritmo *Merge-Join* para uma operação de junção sobre duas relações r e s usando a estrutura *MSJoin*, no pior caso, é $EC = p_r + p_s + log_{\frac{n}{2}}m$, onde p_r representa o número de páginas de r, p_s representa o número de páginas de s e $log_{\frac{n}{2}}m$ representa a altura da árvore B$^+$ estendida.

A seguir, será ilustrado o uso da estrutura *MSJoin* através de um exemplo de aplicação para *smartcards*.

Considere uma aplicação de saúde onde cada usuário carrega seus próprios dados através de um *smartcard*. Um sistema de banco de dados para *smartcards* seria responsável por gerenciar os dados no mesmo. Agora suponha que o banco de dados dos *smartcards* de cada usuário tenha as seguintes relações:

▹ *Consultas* (cc, data, ch);

▹ *Médicos* (cd, especialidade, nome);

▹ *Hospitais* (ch, nome);

▹ *Med-Hosp* (cd, ch, hs).

As chaves primárias das relações estão sublinhadas. O relacionamento entre *Médicos* e *Hospitais* é m:n e o relacionamento entre *Hospitais* e *Consultas* é 1:n.

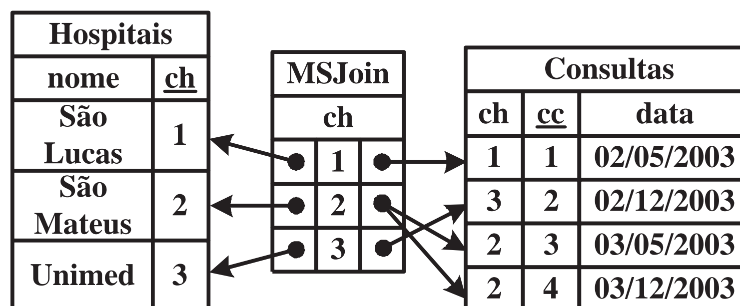

Figura 4.5. *Representação gráfica da Estrutura MSJoin.*

A figura 4.5 ilustra uma representação gráfica da estrutura *MSJoin* usando o relacionamento entre *Consultas* e *Hospitais*. Neste exemplo, o atributo de junção

é a chave primária **ch** da relação *Hospitais*, representada como uma chave estrangeira na relação *Consultas*. Assim, a estrutura *MSJoin* representando esse atributo referencia as tuplas das relações que satisfazem a seguinte condição de junção: $Hospitais.ch = Consultas.ch$.

A figura 4.6 mostra a implementação da estrutura *MSJoin* ilustrada na figura 4.5 usando o conceito de árvore B$^+$ estendida. No exemplo, o atributo de junção **ch** é o valor de chave de busca. Os valores de chave de busca (atributos de junção) são armazenados nas folhas da árvore. Cada valor de chave de busca possui dois ponteiros. No caso onde o valor de chave de busca $K = 1$, por exemplo, tem-se um ponteiro apontando para a tupla da relação *Hospitais*, onde $Hospitais.ch = 1$ e um ponteiro apontando para a tupla da relação *Consultas*, onde $Consultas.ch = 1$. É importante acrescentar que, em casos onde existem múltiplas ocorrências de um valor de chave de busca, o conceito de nível de indireção entre o nó folha e as tuplas é usado, como pode ser visto na figura 4.6, no caso onde o valor de chave de busca $K = 2$.

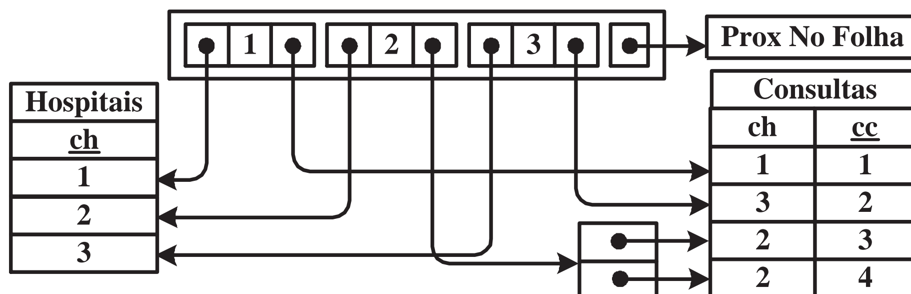

Figura 4.6. *Representação da estrutura MSJoin usando a árvore B$^+$ estendida.*

4.4 Estrutura MSJoin$^+$

A estrutura *MSJoin*$^+$ é uma extensão da estrutura *MSJoin*. A idéia chave por detrás da especificação da estrutura *MSJoin*$^+$ é garantir a compactação dos dados, ou seja, garantir a otimização do espaço de armazenamento. Essa propriedade é alcançada

através do armazenamento dos atributos de junção apenas na estrutura *MSJoin*$^+$. Fazendo isso, atributos de junção não são armazenados nas relações base. Nesse caso, além de existirem ponteiros da estrutura *MSJoin*$^+$ para as tuplas das relações base, existem ponteiros das tuplas das relações base para os valores correspondentes na estrutura *MSJoin*$^+$.

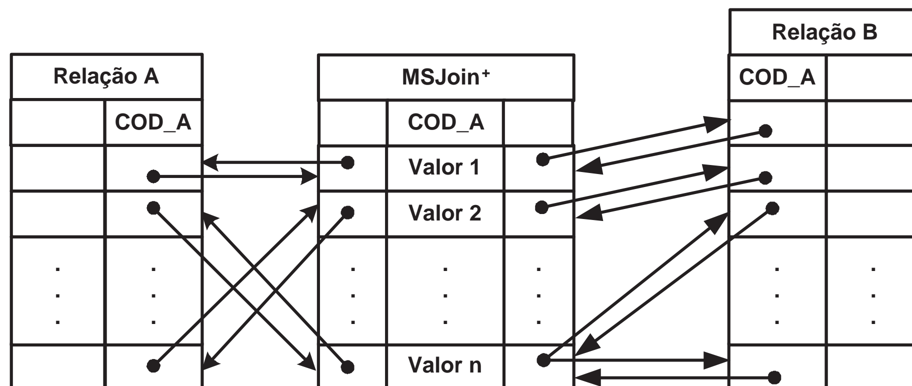

Figura 4.7. *Estrutura MSJoin*$^+$.

Considere a figura 4.7 que ilustra a estrutura de armazenamento *MSJoin*$^+$. Assim como na figura 4.3, as relações A e B possuem um atributo chamado COD_A. Esse é o atributo de junção que deve ser armazenado na estrutura *MSJoin*$^+$. A forma como os dados são armazenados na estrutura *MSJoin*$^+$ é similar à forma como são armazenados na estrutura *MSJoin*. A diferença está na maneira como o atributo COD_A é armazenado nas relações base. Ao invés de se ter os valores do atributo de junção COD_A armazenados também nas relações A e B, tem-se apenas um ponteiro das tuplas das relações base para as tuplas da estrutura *MSJoin*$^+$ que possuem o mesmo valor para o atributo de junção. Por exemplo, se existe uma tupla na Relação A, onde $A.COD_A$ = *"Valor 2"*, ao invés do campo COD_A armazenar este valor, ele possui um ponteiro apontando para a tupla da estrutura *MSJoin*$^+$, onde o atributo de junção $A.COD_A$ = *"Valor 2"*. O mesmo vale para a Relação B. É válido salientar que está se considerando o tamanho do atributo de junção maior

que o tamanho de um ponteiro. Caso o atributo de junção seja menor que o tamanho de um ponteiro, é melhor utilizar a estrutura *MSJoin* proposta anteriormente.

Como as relações estão compartilhando os dados armazenados na estrutura *MSJoin*$^+$, nem todos os valores da estrutura *MSJoin*$^+$ satisfazem uma condição de junção. Portanto, aqui a execução da operação de junção é um pouco diferente da execução que ocorre na estrutura *MSJoin*. Para executar uma operação de junção utilizando a estrutura *MSJoin*, o mecanismo de processamento de consultas precisa apenas localizar a estrutura *MSJoin* e, através dos ponteiros para as tuplas que satisfazem a condição de junção, retornar tais tuplas. Já no caso da estrutura *MSJoin*$^+$, o mecanismo de processamento de consultas além de ter que localizar a estrutura *MSJoin*$^+$, ele precisa identificar os ponteiros para as tuplas das relações que satisfazem a condição de junção. Após essa identificação, só é necessário retornar as tuplas. Essa identificação pode ser feita da seguinte forma: ao acessar a estrutura *MSJoin*$^+$ e encontrar o primeiro valor, o mecanismo de processamento de consultas verifica se um dos ponteiros desse valor está apontando para *nil*. Se estiver, esse valor não satisfaz nenhuma condição de junção e não deve ser retornado. Este processo deve ser seguido até que o último valor seja encontrado.

Mesmo com essa diferença no mecanismo de processamento de consultas, o custo estimado do algoritmo *Merge-Join* para uma operação de junção sobre duas relações r e s usando a estrutura *MSJoin*$^+$, no pior caso, é o mesmo custo da estrutura *MSJoin*, ou seja, $EC = p_r + p_s + log_{\frac{n}{2}}m$, onde p_r representa o número de páginas de r, p_s representa o número de páginas de s e $log_{\frac{n}{2}}m$ representa a altura da árvore B$^+$ estendida.

A figura 4.8 ilustra uma representação gráfica de um exemplo que usa a estrutura *MSJoin*$^+$ para implementar o relacionamento entre as relações *Médicos* e *Hospitais*. Neste exemplo, tem-se dois atributos de junção. O primeiro atributo de junção é a chave primária **cd** da relação *Médicos*, representada como uma chave estrangeira na relação *Med-Hosp*. O segundo atributo de junção é a chave primária **ch** da relação *Hospitais*, representada como uma chave estrangeira na relação *Med-Hosp*.

As chaves estrangeiras **cd** e **ch** formam uma chave primária na relação *Med-Hosp*. A estrutura *MSJoin*$^+$ representando o atributo **cd** referencia as tuplas das relações que satisfazem a seguinte condição de junção: *Medicos.cd = Med-Hosp.cd*. A estrutura *MSJoin*$^+$ representando o atributo **ch** referencia as tuplas das relações que satisfazem a seguinte condição de junção: *Hospitais.ch = Med-Hosp.ch*. Usando a estrutura *MSJoin*$^+$ a compactação dos dados é garantida, pois tanto o atributo **cd**, como o atributo **ch** são armazenados apenas nas estruturas *MSJoin*$^+$ intermediárias.

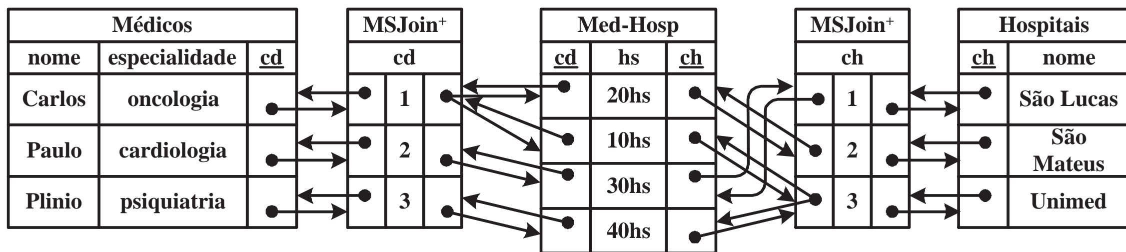

Figura 4.8. *Representação da estrutura MSJoin*$^+$.

Relacionamentos m:n economizarão mais espaço de armazenamento se eles forem representados pela estrutura *MSJoin*$^+$ e não pela *MSJoin*, desde que no primeiro caso, ao invés de se armazenar os atributos de junção na estrutura e nas relações envolvidas na operação de junção, armazenam-se ponteiros que têm a estrutura *MSJoin*$^+$ como origem e as tuplas das relações como destino e ponteiros que têm as tuplas das relações como origem e a estrutura *MSJoin*$^+$ como destino.

A figura 4.9 mostra a implementação da estrutura *MSJoin*$^+$ ilustrada na figura 4.8 usando o conceito de árvore B$^+$ estendida. Neste exemplo, tem-se duas árvores: uma árvore com valor de chave de busca representando o atributo de junção **cd** e outra com valor de chave de busca representando o atributo de junção **ch**. Os valores de chave de busca que representam os atributos de junção são armazenados nas folhas da árvore. Cada valor de chave de busca possui dois ponteiros. No caso do valor de chave de busca representado pelo atributo **cd**, onde **cd** = 2, tem-se um ponteiro apontando para a tupla da relação *Medicos*, onde *Medicos.cd = 2* e um ponteiro

apontando para a tupla da relação *Med-Hosp*, onde *Med-Hosp.cd* = *2*. Além disso, tem-se um ponteiro saindo da tupla da relação *Medicos*, onde *Medicos.cd* = *2* e outro saindo da tupla da relação *Med-Hosp*, onde *Med-Hosp.cd* = *2* em direção ao valor de chave de busca do nó folha, onde K = *2*. No caso de existirem múltiplas ocorrências de um valor de chave de busca, o conceito de nível de indireção é usado entre o nó folha e as tuplas, assim como é usado na implementação da estrutura *MSJoin*. Isto pode ser observado na figura 4.9, no caso onde o valor de chave de busca K = *1*.

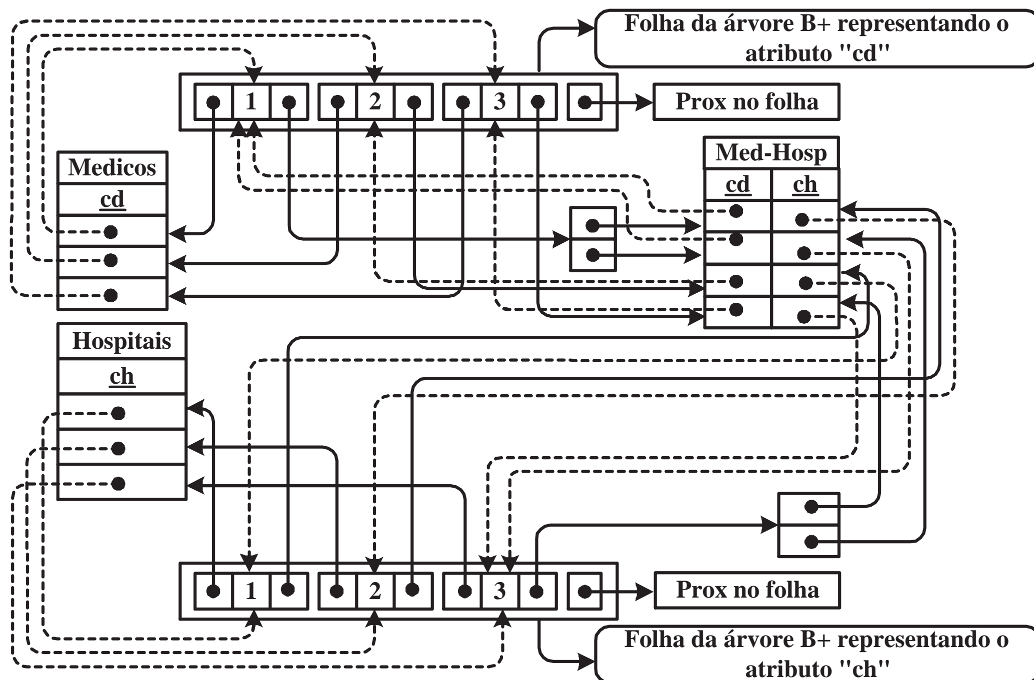

Figura 4.9. *Representação da estrutura MSJoin$^+$ usando a árvore B^+ estendida.*

5 Validação

5.1 Introdução

Neste capítulo será descrito como foi realizada a implementação para validar a estrutura *MSJoin*. Antes de se falar da validação da estrutura *MSJoin*, entretanto, será explicado o funcionamento dos algoritmos de inserção e remoção em uma árvore B^+ já que as estruturas propostas são baseadas em árvores B^+. Os algoritmos aqui citados foram baseados na obra [23]. Além disso, será descrita a forma como foi feita a implementação da estrutura *MSJoin*. Os resultados da simulação que comparou a execução de uma junção utilizando a estrutura *MSJoin* com a execução utilizando o algoritmo *Nested-Loop Join* também serão apresentados.

5.2 Inserção em árvores B^+

O funcionamento do algoritmo de inserção de valores em uma árvore B^+ é descrito a seguir. O algoritmo recebe uma entrada (valor de chave de busca), acha o nó folha onde essa entrada deve ser inserida e insere a entrada nesse nó. O *procedimento inserção* (figura 5.1) acha esse nó folha e faz a chamada ao procedimento para inserir a entrada na árvore B^+. O pseudocódigo para o algoritmo de inserção de uma entrada em uma árvore B^+ é mostrado na figura 5.2.

```
1.  procedimento inserção (valor V, ponteiro P)
2.  início
3.     Ache o nó folha L que deve conter o valor V
4.     inserirEntrada(L, V, P)
5.  fim procedimento
```

Figura 5.1. *Chamada ao procedimento inserirEntrada.*

A idéia chave por trás da inserção em uma árvore B^+ é que a inserção de um novo valor de chave de busca é feita recursivamente através de uma chamada ao algoritmo *inserirEntrada* sobre o nó folha apropriado. Geralmente esse procedimento se resume a descer até o nó folha da árvore onde o novo valor de chave de busca deve ser inserido e colocar o novo valor nessa folha.

No caso do valor de chave de busca a ser inserido já existir no nó folha, o registro é adicionado ao arquivo e um ponteiro é inserido no bloco de ponteiros para o arquivo, caso seja necessário. Se o valor ainda não existir, o registro é adicionado ao arquivo e um bloco de ponteiros é criado. Nesse último caso, onde o valor de chave de busca não existe na árvore, então há duas possibilidades. Se existir espaço no nó folha, o par (*valor*, *ponteiro*) é inserido no nó folha, na posição correta, de forma que os valores de chave de busca continuem ordenados. Caso não haja espaço no nó folha, o nó é dividido em dois nós folhas L e L'. O algoritmo pega os n pares (*valor*, *ponteiro*), incluindo o par a ser inserido, de forma ordenada. Os primeiros $\lceil \frac{n}{2} \rceil$ valores são mantidos no nó original (L) e o restante é colocado no novo nó L'. Sendo K o menor valor existente em L', o algoritmo insere K no pai do nó que foi dividido. Um ponteiro extra para o novo nó é criado no nó pai. Se o nó pai (nó interno) estiver cheio, será necessário dividi-lo também. A divisão de nós procede até que um nó que não esteja cheio seja achado. No pior caso, essa divisão poderá se propagar até a raiz, criando um novo nó raiz e conseqüentemente, criando um novo nível na árvore B^+, aumentando assim a altura da árvore.

5.3 Remoção em árvores B^+

O funcionamento do algoritmo de remoção de valores em uma árvore B^+ é descrito a seguir. O algoritmo recebe um valor de chave de busca, acha o nó folha ao qual esse valor pertence e remove o valor chave desse nó. O *procedimento remoção* (figura 5.3) acha esse nó folha e faz a chamada ao procedimento para remover o valor de chave de busca da árvore B^+.

1. **procedimento** inserirEntrada (*nó L, valor V, ponteiro P*)
2. **início**
3. **se** (L tem espaço para (V, P)) **então**
4. inserir (V, P) em L /* e termina a execução */
5. **senão**
6. **início** /* Divide L */
7. Criar nó L’
8. **se** (L é um nó folha) **então**
9. **início**
10. Seja V' o valor tal que $\lceil \frac{n}{2} \rceil$ dos valores
11. $L.K_1$, $L.K_2$,..., $L.K_{n-1}$, V são menores que V'
12. Seja m o menor valor tal que $L.K_m \geq V'$
13. mova $L.P_m$, $L.K_m$,..., $L.P_{n-1}$, $L.K_{n-1}$ para L'
14. **se** ($V < V'$) **então**
15. inserir (P, V) em L
16. **senão**
17. inserir (P, V) em L'
18. **fim**
19. **senão**
20. **início**
21. Seja V' o valor tal que $\lceil \frac{n}{2} \rceil$ dos valores
22. $L.K_1$, $L.K_2$,..., $L.K_{n-1}$, V são maiores que ou iguais a V'
23. Seja m o menor valor tal que $L.K_m \geq V'$
24. adicione Nil, $L.K_m$,..., $L.P_{n-1}$, $L.K_{n-1}$, $L.P_n$ a L'
25. remova $L.K_m$,..., $L.P_{n-1}$, $L.K_{n-1}$, $L.P_n$ de L
26. **se** ($V < V'$) **então**
27. inserir (V, P) em L
28. **senão**
29. inserir (V, P) em L'
30. remova (Nil, V') de L'
31. **fim**
32. **se** (L não for o nó raiz da árvore) **então**
33. inserirEntrada ($pai(L)$, V', L')
34. **senão**
35. **início**
36. Criar um novo nó R com os nós L e L' e com o valor V'
37. faça de R a nova raiz da árvore
38. **fim**
39. **se** (L é um nó folha) **então**
40. **início** /* Conserte os próximos ponteiros filhos */
41. atribua $L'.P_n = L.P_n$
42. atribua $L.P_n = L'$
43. **fim**
44. **fim**
45. **fim procedimento**

Figura 5.2. *Algoritmo de inserção de uma entrada em uma árvore* B^+

```
1. procedimento remoção (valor V, ponteiro P)
2. início
3.    Ache o nó folha L que contém (V, P)
4.    removerEntrada(L, V, P)
5. fim procedimento
```

Figura 5.3. *Chamada ao procedimento removerEntrada*

O pseudocódigo para o algoritmo de remoção de um valor de chave de busca em uma árvore B^+ é mostrado na figura 5.4.

A idéia chave por trás da remoção em uma árvore B^+ é que a remoção de um valor de chave de busca é feita recursivamente através de uma chamada ao algoritmo *removerEntrada* sobre o nó filho apropriado. Geralmente esse procedimento se resume a descer até o nó folha da árvore onde o valor de chave de busca a ser removido se encontra e removê-lo dessa folha. No caso do valor de chave de busca ser encontrado na árvore, é preciso achar o registro a ser removido, removê-lo do arquivo e do bloco de ponteiros para o arquivo, caso seja necessário.

Depois de achar o valor no nó folha e removê-lo, essa operação pode fazer com que a folha fique com poucas entradas ($\lceil \frac{n-1}{2} \rceil$ valores). Isso pode afetar ou o nó vizinho esquerdo ou o nó vizinho direito da folha, dependendo de qual nó foi escolhido, causando ou uma redistribuição de valores ou a união desses valores em um mesmo nó folha. É válido lembrar que um nó vizinho deve ter o mesmo pai do nó cujo valor de chave de busca foi removido.

Se os valores de chave de busca do nó que continha o valor que foi removido e do seu vizinho (esquerdo ou direito) couberem em um único nó, os dois nós podem ser unidos. Então, todos os valores de chave de busca são inseridos em um único nó e o outro nó é apagado. Sendo o nó uma folha, os *links* entre as folhas são reconectados. O pai dos nós deve ser atualizado para refletir essa mudança através da remoção do valor chave que aponta para o segundo nó.

```
1.  procedimento removerEntrada (nó L, valor V, ponteiro P)
2.  início
3.      remova (V, P) de L
4.      se (L é o nó raiz e L tem apenas um nó filho) então
5.          faça do nó filho de L o novo nó raiz da árvore
6.              e remova L /* termina a execução */
7.      senão se (L tem pouquíssimos valores/ponteiros) então início
8.          /* Redistribuir se possível (não tem senão) */
9.          se (uma entrada puder ser emprestada de um nó vizinho) início
10.             Seja L' o vizinho esquerdo ou o vizinho direito de L /* Escolher */
11.             Seja V' o valor entre L eL' no pai(L)
12.             se (L não for um nó folha) então início
13.                 se (L' é o vizinho direito) início
14.                     remover (P1, K1) de L'
15.                     inserir (V', L'.P1) em L
16.                     substituir V' no pai(L) por L'.K1
17.                 fim
18.                 senão início
19.                     Seja m tal que L'.Pm é o último ponteiro em L'
20.                     remover (Km−1, Pm) de L'
21.                     inserir (L'.Pm, V') em L
22.                     substituir V' no pai(L) por L'.Km−1
23.                 fim
24.             fim
25.             senão início /* L é um nó folha */
26.                 se (L' é o vizinho direito) início
27.                     remover (P1, K1) de L'
28.                     inserir (L'.P1, L'.K1) em L
29.                     substituir V' no pai(L) por L'.K1
30.                 fim
31.                 senão início
32.                     Seja m tal que L'.Km é o último valor em L'
33.                     remover (Pm, Km) de L'
34.                     inserir (L'.Pm, L'.Km) em L
35.                     substituir V' no pai(L) por L'.Km
36.                 fim
37.             fim
38.         fim
39.         senão início /* Una se a redistribuição não for possível */
40.             Seja L' o vizinho esquerdo ou o vizinho direito de L /* Escolher */
41.             Seja V' o valor entre L eL' no pai(L)
42.             se (L não for um nó folha) então
43.                 adicione V' e todos os ponteiros e valores de L em L'
44.             senão início /* L é um nó folha */
45.                 adicione todos os pares (Ki, Pi) de L em L'
46.                 atribua L'.Pn = L.Pn
47.             fim
48.             removerEntrada (pai(L), V', L)
49.             remover nó L
50.         fim
51.     fim
52. fim procedimento
```

Figura 5.4. *Algoritmo de remoção de uma entrada em uma árvore* B^+

Se os valores de chave de busca do nó que continha o valor que foi removido e do seu vizinho (esquerdo ou direito) não couberem em um único nó, os valores devem ser reorganizados através de uma redistribuição dos mesmos. Então, as entradas são redistribuídas entre os dois nós, de forma que ambos os nós tenham pelo menos o número mínimo de entradas permitido. Depois disso, o nó pai precisa ser atualizado para refletir essa mudança pois a redistribuição cria um novo valor inicial em um dos nós. O valor de chave de busca do nó pai que aponta para o segundo nó deve ser alterado para ser o menor valor de chave de busca no segundo nó.

As remoções de nós podem se propagar até que um nó com $\lceil \frac{n-1}{2} \rceil$ valores (ou mais valores para um nó folha) ou $\lceil \frac{n}{2} \rceil$ ponteiros (ou mais ponteiros para um nó interno) seja achado. Se após remoções recursivas, o nó raiz tiver apenas um único filho, esse filho vira o novo nó raiz e o nó raiz anterior é apagado.

5.4 Validação da Estrutura MSJoin

Agora que foi explicado o funcionamento dos algoritmos de inserção e de remoção em uma árvore B^+, será detalhado como foi feito o processo de validação da estrutura *MSJoin*.

De forma a avaliar a *performance* da estrutura *MSJoin* proposta, a execução de uma junção utilizando a estrutura *MSJoin* foi comparada com a execução de uma junção utilizando o algoritmo *Nested-Loop Join*. A *performance* dessas junções foi avaliada baseando-se no custo necessário para executar uma junção usando ambas as idéias, considerando sempre o pior caso.

5.4.1 Ambiente de Simulação

A estrutura *MSJoin* e a simulação realizada sobre essa estrutura foram inicialmente desenvolvidas na linguagem Java utilizando a especificação J2SE 1.4.2 [25]. Os resultados da simulação são baseados nessa implementação. Posteriormente, a implementação da estrutura *MSJoin* e da simulação foram migradas para a plataforma

J2ME (CLDC 1.1/MIDP 2.0) [27, 28], onde foi criada uma pequena aplicação capaz de ser executada em ambientes computacionais limitados. Para completar a validação da estrutura *MSJoin* proposta, a aplicação foi executada em um emulador J2ME e depois em um telefone celular compatível com J2ME. A tecnologia J2ME possui um ambiente de desenvolvimento Java altamente otimizado que soluciona o problema de consumo de espaço de armazenamento. Essa tecnologia abrange os dispositivos extremamente pequenos, tais como, *smartcards*, telefones móveis, PDAs e outros sistemas embutidos. A organização das classes criadas foi baseada na seguinte divisão funcional.

Para simular o comportamento das tuplas foram criadas as classes *Hospital.java* e *Consulta.java*, compostas basicamente pelos atributos das entidades *Hospital* e *Consulta*, respectivamente, além dos métodos de consulta e atualização desses atributos. Essas classes implementam a interface *Tupla.java*.

Para simular o comportamento das tabelas do banco de dados foram criadas as classes *Hospitais.java* e *Consultas.java*, compostas por um atributo que representa uma coleção de objetos das entidades *Hospital* e *Consulta*, respectivamente, além dos métodos de consulta e atualização desse atributo. Essas classes implementam a interface *Tabela.java*.

Para simular o comportamento da estrutura *MSJoin* foram criadas as classes *ArvoreBMais.java*, *NoIntermediario.java*, *NoFolha.java* e *Estrutura.java* e a interface *No.java*. A classe *ArvoreBMais.java* é composta de atributos e métodos necessários para realizar as inserções e remoções em uma árvore B^+. As classes *NoIntermediario.java* e *NoFolha.java* são compostas por atributos e métodos necessários para realizar as operações específicas de um nó intermediário e folha, respectivamente. Essas duas últimas implementam a interface *No.java*. A classe *Estrutura.java* foi criada para representar o nó folha da árvore B^+ estendida.

Uma classe chamada *NestedLoop.java* foi criada para simular o comportamento do algoritmo *Nested-Loop Join*.

Uma classe chamada *Principal.java* foi criada para simular a invocação das

junções. Essa classe realiza basicamente as chamadas aos métodos da estrutura *MSJoin* proposta e da classe que implementa o algoritmo *Nested-Loop Join*.

A simulação ocorre da seguinte forma. Alguns valores são inseridos nas tabelas *Hospitais* e *Tuplas*, em tempo de execução. Os valores que satisfazem a condição de junção são inseridos na estrutura *MSJoin*, utilizando para isso o algoritmo de inserção de árvores B^+.

Para recuperar as tuplas que satisfazem a condição de junção usando a estrutura *MSJoin*, a classe *Principal.java* tem apenas que acessar a classe que representa a estrutura *MSJoin*, ou seja, a classe *ArvoreBMais.java*, onde estão armazenados os valores de chave de busca ou atributos de junção das tabelas. Como os valores estão armazenados de forma ordenada, pois a estrutura de armazenamento baseia-se em árvores B^+, o acesso a esses valores é seqüencial e a estimativa de custo para buscar estes valores no pior caso é $EC = p_r + p_s + log_{\frac{n}{2}} m$.

Para recuperar as tuplas que satisfazem a condição de junção usando o algoritmo *Nested-Loop Join* a classe *Principal.java* tem apenas que acessar a classe *nested-loop*, que irá executar o algoritmo sobre as tabelas *Hospitais* e *Consultas*, e depois retornar os valores que satisfazem a condição de junção. No pior caso, a estimativa de custo para buscar estes valores é dada por $EC = p_r + (n_r * p_s)$.

Para comparar a execução da junção usando a estrutura *MSJoin* e o algoritmo *Nested-Loop Join*, foi fixado que o tamanho de uma página de disco poderia ter no máximo 100 tuplas. A quantidade de valores inseridos nas tabelas variou de 100 a 10000 tuplas. Considerando, por exemplo, que a quantidade de tuplas de uma tabela seja 500, ela ocupará 5 páginas de disco. Além disso, os tamanhos das duas tabelas eram iguais. Se 500 registros fossem inseridos na tabela *Hospitais*, 500 registros também deveriam ser inseridos na tabela *Consultas*. Foi considerado também que todas as tuplas da tabela satisfaziam a condição de junção, pois considerou-se o pior caso.

5.4.2 Resultados da Simulação

A figura 5.5 mostra os resultados experimentais da simulação realizada.

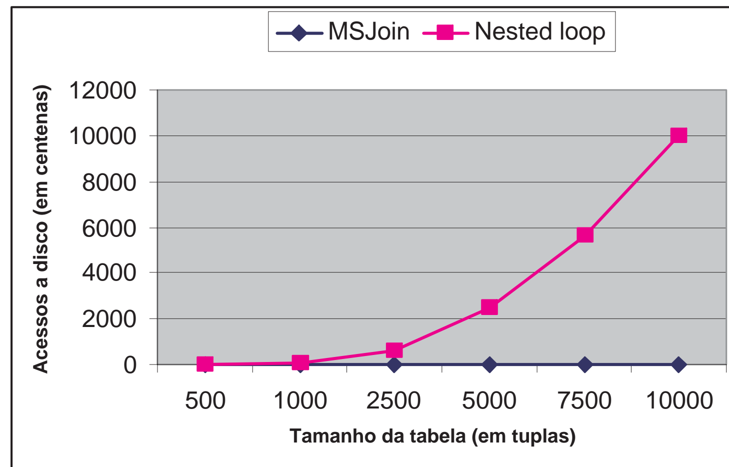

Figura 5.5. *Estimativa de custo*

Nesta figura, é possível verificar que a *performance* da nossa abordagem é melhor do que a *performance* da abordagem do PicoDBMS apresentada em [4], que utiliza o algoritmo *Nested-Loop Join*. Nela também é possível ver que a quantidade de acessos a disco de uma junção que é executada usando a estrutura *MSJoin* é menor do que usando o algoritmo *Nested-Loop Join*.

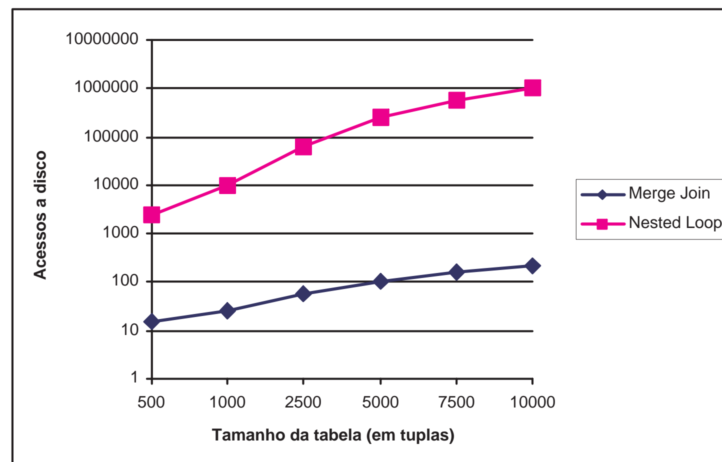

Figura 5.6. *Estimativa de custo em escala logarítmica*

A figura 5.6 mostra os mesmos resultados apresentados na figura 5.5. Entretanto, ele está em escala logarítmica para que seja possível visualizar melhor a diferença de *performance* na execução de uma junção usando a estrutura *MSJoin* e de outra junção usando o algoritmo *Nested-Loop Join*.

6 Conclusão

6.1 Considerações Sobre as Estruturas Propostas

A motivação desta dissertação se deve à carência de trabalhos envolvendo processamento de consultas em ambientes com recursos computacionais limitados. Tais ambientes possuem severas limitações de *hardware*, como por exemplo, a pequena capacidade de armazenamento e processamento. Como conseqüência, ambientes computacionais limitados são capazes de processar apenas uma pequena quantidade de dados e esse processamento tende a ser ineficiente. Logo, é importante introduzir técnicas e funcionalidades de bancos de dados para gerenciar dados de maneira eficiente.

Também devido às limitações computacionais, as técnicas de bancos de dados tradicionais não podem ser diretamente aplicadas a ambientes com recursos computacionais limitados. Adpatar essas técnicas tem surgido como um grande desafio para a comunidade de banco de dados. Pouca atenção tem sido dada à adaptação dessas técnicas. No entanto, adaptá-las é um tanto quanto difícil e por isso merece atenção.

Dessa forma, foram propostas duas estruturas de armazenamento intituladas *MSJoin* e *MSJoin*$^{+}$ com o objetivo de otimizar a execução de junções em ambientes computacionais com recursos limitados, utilizando para isso, estruturas de indexação baseadas em árvores B^{+}. A idéia das estruturas propostas é garantir a execução do operador relacional de junção através do algoritmo *Merge-Join*, o qual possui a menor estimativa de custo para a operação de junção. Além de otimizar a execução de junções, a estrutura *MSJoin*$^{+}$ também otimiza o espaço de armazenamento.

De forma a avaliar a *performance* da estrutura *MSJoin* proposta, foi feita uma simulação para comparar a execução da operação de junção através da estrutura

MSJoin com a execução da operação de junção através do algoritmo que é utilizado para executar operações de junção no protótipo do PicoDBMS [4], ou seja, o algoritmo *Nested-Loop Join*. Tanto a estrutura *MSJoin* como a simulação feita sobre essa estrutura foram implementadas na linguagem Java utilizando a especificação J2SE 1.4.2 [25]. Os resultados da simulação são baseados nessa implementação. Posteriormente, essa implementação foi migrada para a plataforma J2ME (CLDC 1.1/MIDP 2.0) [27, 28], onde foi criada uma pequena aplicação capaz de ser executada em ambientes computacionais limitados. Essa aplicação foi executada em uma emulador J2ME e depois em um telefone celular compatível com J2ME. Os resultados experimentais mostram que a abordagem proposta tem um desempenho melhor que a abordagem do PicoDBMS [4], que usa o algoritmo *Nested-Loop Join*.

O objetivo de otimizar a execução do operador relacional de junção em ambientes com recursos computacionais limitados foi alcançado uma vez que as simulações realizadas comprovaram que as estruturas propostas tem uma estimativa de custo menor do que a estimativa de custo do algoritmo *Nested-Loop Join* e, conseqüentemente, tem um desempenho melhor do que tal algoritmo.

6.2 Trabalhos Futuros

Um dos trabalhos futuros desencadeado por essa dissertação será fazer a migração da implementação da estrutura *MSJoin* da plataforma J2ME para a plataforma JavaCard [26], tecnologia projetada e desenvolvida especificamente para *smartcards*.

Também pode ser apontado como um trabalho futuro o desenvolvimento de um protótipo de um sistema de banco de dados para ambientes computacionais com recursos limitados utilizando as estruturas de armazenamento propostas, a fim de gerenciar os dados de maneira eficiente em tais ambientes. Como as estruturas de armazenamento propostas possuem uma estimativa de custo baixa, a utilização da estrutura *MSJoin* torna o processamento de consultas mais eficiente e a utilização da estrutura *MSJoin*$^{+}$ otimiza o espaço de armazenamento em disco.

6.3 Considerações Finais

Dentre os estudos realizados para tentar atingir a meta de garantir o processamento de consultas eficiente em ambientes computacionais limitados, foram pesquisados alguns algoritmos de processamento de consultas adaptativos a fim de descobrir se tais algoritmos seriam mais eficazes nesses ambientes. Entretanto, eles utilizam muitos recursos de sistema, impossibilitando assim sua adaptação para tais ambientes.

Muito do que foi pesquisado nesse trabalho origina dúvidas e questionamentos sobre o processamento de consultas em ambientes computacionais limitados, especialmente pelo fato das pesquisas envolvendo esses ambientes serem uma área de estudo recente. Entretanto, pode-se considerar que esta dissertação cumpriu seu papel ao mostrar que os resultados encontrados foram satisfatórios.

Espera-se ter contribuído de alguma forma com a comunidade científica da área de banco de dados, tendo provido soluções alternativas para o processamento de consultas e para o armazenamento de dados em dispositivos computacionais com recursos limitados.

Referências Bibliográficas

[1] Anciaux, N., Bobineau, C., Bouganim, L., Pucheral, P., e Valduriez, P. PicoDBMS: Validation and experience. In *Proceedings of the 27th International Conference on Very Large Data Bases* (Roma, Setembro 2001), pp. 709–710.

[2] Arumugam, S., e Nagarajan, K. Survey of small footprint databases. Tech. rep., University of Florida, 2000.

[3] Avnur, R., e Hellerstein, J. M. Eddies: Continuously adaptive query processing. In *Proceedings of ACM-SIGMOD International Conference on Management of Data* (Dallas, Texas, USA, Maio 2000), pp. 261–272.

[4] Bobineau, C., Bouganim, L., Pucheral, P., e Valduriez, P. PicoDBMS: Scaling down database techniques for the smartcard. In *Proceedings of the 26th International Conference on Very Large Data Bases* (Cairo, Setembro 2000), pp. 11–20.

[5] Bradley, P. Implementing airline electronic ticketing using integrated circuit cards (smart cards). Computer Science Degree Project, Dublin Institute of Technology. Disponível em <http://smartcardtutor.future.easyspace.com/proposal.htm>. Acesso em: 28/02/2003, 1998.

[6] Brayner, A., e Paulino, T. M. Processamento eficiente de consultas em ambientes de smartcards. In *Anais do I Workshop de Teses e Dissertações em Banco de Dados* (Gramado, Outubro 2002), pp. 92–96.

[7] Cagliostro, C. Primer on smart cards. Disponível em <http://www.securegroup.it/smartcard/overview.htm>. Acesso em:24/08/2002, 1999.

[8] Carrasco, L. C. RDBMS's for Java Cards? What a senseless idea! Disponível em <http://www.smartcardcentral.com/technical/articles/rdbms/rdbm[1].asp>. Acesso em: 24/08/2002., 1999.

[9] Chen, Z., e Giorgio, R. D. Understanding java card 2.0. *Java Developer* (March 1998). Disponível em <http://www.javaworld.com/javaworld/jw-03-1998/jw-03-javadev.html>. Acesso em: 27/04/2003.

[10] Dallas Semiconductor. *iButtons*, 2003. Disponível em <http://www.ibutton.com/>. Acesso em: 27/04/2003.

[11] Elmasri, R., e Navathe, S. B. *Fundamentals of Database Systems*, 3^a ed. Addison-Wesley, Junho 2000, capítulos 6 e 18.

[12] Everett, D. B. Smart card technology: Introduction to smart cards. Disponível em <http://www.smartcard.co.uk/resources/articles/intro2sc.html>. Acesso em: 24/08/2002, 2001.

[13] Garcia-Molina, H., Ullman, J. D., e Widom, J. *Database System Implementation*. Prentice Hall, 2000, capítulos 6 e 7.

[14] Gemplus. *The Smart Card Engine: a small microcontroller integrated circuit*, 2002. Disponível em <http://www.gemplus.com/techno/chipware>. Acesso em: 20/08/2002.

[15] Giorgio, R. D. Smart cards: A primer. *Java Developer* (December 1997). Disponível em <http://www.javaworld.com/jw-12-1997/jw-12-javadev.html>. Acesso em: 27/04/2003.

[16] GRAEFE, G. Query evaluation techniques for large databases. *ACM Computing Surveys 25*, 2 (Junho 1993), pp. 73–170.

[17] HAAS, P. J., E HELLERSTEIN, J. M. Ripple joins for online aggregation. In *Proceedings of ACM-SIGMOD International Conference on Management of Data* (Philadelphia, Pennsylvania, USA, Junho 1999), pp. 287–298.

[18] HELLERSTEIN, J. M., FRANKLIN, M. J., CHANDRASEKARAN, S., DESHPANDE, A., HILDRUM, K., MADDEN, S., RAMAN, V., E SHAH, M. A. Adaptive query processing: Technology in evolution. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 23*, 2 (Junho 2000), pp. 7–18.

[19] IBM CORPORATION. *DB2 Everyplace*, 2001. Disponível em <http://www-4.ibm.com/software/data/db2/everyplace>. Acesso em: 24/08/2002.

[20] KURAMITSU, K., E SAKAMURA, K. Towards ubiquitous database in mobile commerce. In *Proceedings of the 2nd ACM International Workshop on Data Engineering for Wireless and Mobile Access* (California, Maio 2001), pp. 84–89.

[21] MICROSOFT CORPORATION. *SQL Server CE*, 2001. Disponível em <http://www.microsoft.com/sql/CE/default.asp>. Acesso em: 24/08/2002.

[22] ORACLE CORPORATION. *Oracle9i Lite*, 2002. Disponível em <http://otn.oracle.com/docs/products/lite/content.html>. Acesso em: 24/08/2002.

[23] SILBERSCHATZ, A., KORTH, H. F., E SUDARSHAN, S. *Database System Concepts*, 3ª ed. WCB/McGraw-Hill, 1997, capítulos 11 e 12.

[24] SUN MICROSYSTEMS. *JavaCardTM 2.1 Application Programming Interface Specification*, 1999. JavaSoft Documentation.

[25] SUN MICROSYSTEMS. *Java 2 Platform, Standard Edition Specification 1.4.2*, 2004. JavaSoft Documentation. Disponível em <http://java.sun.com/j2se/1.4.2/index.jsp>. Acesso em: 05/07/2004.

[26] SUN MICROSYSTEMS. *JavaCard^{TM} 2.2.1 Application Programming Interface Specification*, 2004. JavaSoft Documentation. Disponível em <http://java.sun.com/products/javacard/specs.html>. Acesso em: 06/07/2004.

[27] SUN MICROSYSTEMS. *Java^{TM} 2 Platform, Micro Edition Connected Limited Device Configuration (CLDC) Specification 1.1*, 2004. JavaSoft Documentation. Disponível em <http://java.sun.com/products/cldc/index.jsp>. Acesso em: 06/07/2004.

[28] SUN MICROSYSTEMS. *Java^{TM} 2 Platform, Micro Edition Mobile Information Device Profile (MIDP) Specification 2.0*, 2004. JavaSoft Documentation. Disponível em <http://java.sun.com/products/midp/index.jsp>. Acesso em: 06/07/2004.

[29] SYBASE INC. *SQL Anywhere Studio 8*, 2001. Disponível em <http://www.sybase.com/products/anywhere>. Acesso em: 24/08/2002.

[30] URHAN, T., E FRANKLIN, M. J. Xjoin: A reactively-scheduled pipelined join operator. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 23*, 2 (Junho 2000), pp. 27–33.

[31] VALDURIEZ, P. Join indices. *ACM Transactions on Database Systems 12*, 2 (Junho 1987), pp. 218–246.

[32] VINGRALEK, R. GnatDb: A small-footprint, secure database system. In *Proceedings of the 28th International Conference on Very Large Data Bases* (Hong Kong, Agosto 2002), pp. 884–893.