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

MFP: UMA POLÍTICA EFICIENTE DE LIBERAÇÃO DE MEMÓRIA PARA O OPERADOR FÍSICO HASH-MERGE JOIN

DANIELLE DA COSTA FILGUEIRAS ALBUQUERQUE

FORTALEZA - CE
2007

DANIELLE DA COSTA FILGUEIRAS ALBUQUERQUE

MFP: UMA POLÍTICA EFICIENTE DE LIBERAÇÃO DE MEMÓRIA PARA O OPERADOR FÍSICO HASH-MERGE JOIN

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

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

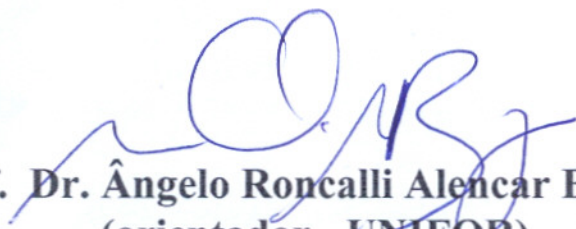
FORTALEZA - CE
2007

Danielle da Costa Filgueiras Albuquerque

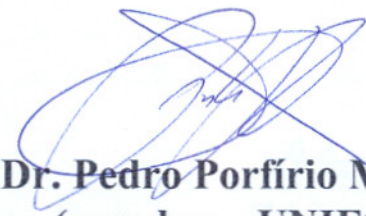
MFP: Uma Política Eficiente de Liberação de Memória para o Operador Físico Hash-Merge Join

Data de Aprovação: 18/12/2007

Banca Examinadora:



Prof. Dr. Ângelo Roncalli Alencar Brayner
(orientador - UNIFOR)



Prof. Dr. Pedro Porfírio Muniz Farias
(membro - UNIFOR)

Rossana Maria de Castro Andrade

Profª. Drª. Rossana Maria de Castro Andrade
(membro – UFC)

Dedico este trabalho aos meus filhos,
Vitor e Davi e ao meu marido
Artur. Eles são os motivos e a força
necessária para eu alcançar meus
objetivos e superar qualquer
obstáculo na vida.

AGRADECIMENTOS

Agradeço a **Deus** por me ter dado força e determinação para alcançar com êxito mais uma meta. Sem Ele eu não teria uma gravidez maravilhosa, filhos maravilhosos e ainda finalizar o mestrado. Sou eternamente grata a **Deus** por me ajudar durante as provas e a me preparar para uma boa defesa, bem como por Ele ter me proporcionado mais esta oportunidade de crescimento profissional.

Agradeço ao meu marido, **Artur**, por compreender todos os momentos que não pude estar ao seu lado, por estar estudando até tarde da noite para concretizar este sonho. Sem ele eu não teria enfrentado todos os obstáculos e bênçãos que apareceram durante o mestrado. Sou grata por ele incentivar meu crescimento profissional e sempre ser o primeiro a acreditar que eu vou alcançar meus objetivos.

Agradeço aos meus filhos, **Vitor** e **Davi**, por terem aparecido em minha vida, rodeados de muitas bênçãos. Por eles e pra eles eu finalizei um mestrado.

Agradeço aos meus pais, **Edna e Dimas**, por me entenderem quando eu não estava presente. Sei que hoje eles estão orgulhosos da "filhinha" deles, pois nem mesmo eu imaginava ser mestre um dia.

Agradeço ao meu irmão, sogros e cunhados por compreenderem a minha falta em momentos importantes.

Agradeço ao meu grande amigo **Milton Escóssia** por me ajudar na implementação do protótipo para avaliação do algoritmo proposto. Sem ele, esta dissertação não teria êxito.

Agradeço ao meu orientador **Ângelo Brayner** por ter paciência e me dar muita força para jamais desistir. Nestes 3 anos de mestrado aprendi muito com ele e vou levar estes ensinamentos aos meus alunos.

Agradeço ao corpo docente do Mestrado em Informática Aplicada – MIA por me terem transmitido valiosos conhecimentos.

Agradeço aos professores **Rossana** e **Porfírio** pelos importantes comentários e sugestões para a melhoria deste trabalho e principalmente por me darem a honra de participar da banca de defesa.

Agradeço aos meus amigos que fizeram mestrado comigo por me ajudarem em todos os momentos nestes últimos 3 anos de curso, durante as disciplinas, durante a fase de preparação da dissertação e preparação para a defesa.

Agradeço aos demais amigos que torciam por mim para que eu concretizasse mais esta meta profissional e entendiam quando eu não podia comparecer a alguns eventos.

RESUMO

As tecnologias de computadores móveis e de comunicação sem fio já se tornaram uma realidade no ambiente computacional moderno, resultando no paradigma da Computação Móvel.

Os dispositivos móveis podem ter BD e os dados destes BDs podem ser compartilhados.

No entanto, os operadores de junção dos operadores de consulta convencional não levam em consideração as limitações de um ambiente que suporta mobilidade, por exemplo: desconexão com a rede de comunicação, baixa largura da banda de comunicação, nível de carga de energia das baterias, dentre outras limitações.

Dessa forma, os algoritmos de junção precisam ser ajustados às limitações da computação móvel para efetuar resultados satisfatórios e em executar em tempo hábil uma consulta solicitada pelo usuário.

As características necessárias para um algoritmo ser executado em um ambiente que suporte mobilidade são: (1) produção incremental de resultados à medida que os dados são disponibilizados, (2) continuidade no processamento da consulta mesmo que a entrega dos dados esteja bloqueada, e (3) reação a situações de limitação de memória durante a execução da consulta[1].

Através dos estudos e de experimentos foi constado que o *Hash-Merge Join*(HMJ) é mais eficiente para garantir as 3 características (propriedades) necessárias para se trabalhar de forma eficiente com junções em um ambiente de computação móvel.

Quando a memória está com sua capacidade máxima de armazenamento, o algoritmo HMJ libera partições da memória de acordo com o estado da mesma. Essa adaptação ao estado da memória determina o melhor par de partições, sendo cada partição de uma fonte diferente, a ser enviado ao disco de forma que maximize o tempo para o próximo *overflow* de memória. *Overflow* de memória acontece quando a memória alcança sua capacidade de armazenamento.

O objetivo deste trabalho é propor uma nova política de liberação de dados da memória, MFP. Essa política propõe uma otimização da política *Adaptive Flushing Policy*(AFP), mantendo o mesmo objetivo principal, enviar pares de partições para o disco, em caso de ocorrência de *overflow* de memória.

A política AFP libera pares correspondentes de partições para o disco com base em uma tabela resumo mantida em memória e dois parâmetros: (1) balanceamento de memória e (2) tamanho mínimo de partições. A tabela resumo em memória contém o tamanho de cada partição, o somatório dos tamanhos de cada par de partições de ambas as relações e o tamanho total de partições de cada relação.

A política MFP também libera pares correspondentes de partições para o disco com base em uma constante de balanceamento de memória e uma tabela resumo diferenciada da tabela resumo utilizada pela política AFP. A tabela resumo da nova política tem uma coluna a mais discriminando a diferença da cardinalidade de cada par de partições das relações de entrada.

O objetivo principal da nova coluna é garantir que o par de partições escolhido para ser enviado ao disco sempre deixará a memória balanceada, além de garantir que sempre haverá no mínimo um par de partições a ser enviado ao disco.

ABSTRACT

Mobile computers and wireless communication technologies are already a reality in the modern IT environment, resulting in the paradigm of Mobile Computing.

These mobile devices may have BD and their data may be shared.

However, the junction operators of the conventional search operators do not take into account the limitations of a mobility-supporting environment, such as a disconnection from the communication network, a narrow communication bandwidth, the battery charge level, etc.

Therefore, the junction algorithms need to be adjusted to the limitations of the mobile computing in order to render satisfactory results and execute a search requested by the user within a reasonable period of time.

The necessary characteristics for an algorithm to be executed in a mobility-supporting environment are: (1) incremental production of results as the data become available; (2) continuous processing of the search, even if the delivery of data is blocked; and (3) reaction to limited memory situations during the execution of the search [1].

It was evidenced through studies and tests that the Hash-Merge Join (HMJ) is more efficient to guarantee these 3 characteristics (properties) needed when working with junctions in an efficient fashion within a mobile computing environment.

When the memory is at its full storage capacity, the HMJ algorithm releases memory partitions according to the memory status. This adaptation to the memory status determines the best pair of partitions, being each partition from a different source, to be sent to the disk in a way that maximizes the time until the next memory overflow. A memory overflow occurs when the memory reaches its storage capacity.

The aim of this work is to propose a new memory data flushing policy – MFP. This policy offers an optimization of the Adaptive Flushing Policy (AFP) while keeping the same main goal, i.e. to send pairs of partitions to the disk in the event of a memory overflow.

The AFP releases corresponding pairs of partitions to the disk based on a summary table kept in memory and on two parameters: (1) memory balance; and (2) minimum partition size. The summary table in memory contains the size of each partition, the sum of the individual sizes of each pair of partitions of both lists and the total size of partitions of each list.

The MFP also releases corresponding pairs of partitions to the disk, based on a constant memory balance and a summary table different from the summary table used by the AFP. The summary table of this new policy has one more column, which states the cardinality difference of each pair of partitions of the input lists.

The main goal of this new column is to guarantee that the pair of partitions chosen to be sent to the disk will always leave a balanced memory, while it guarantees that there will always be at least one pair of partitions to be sent to the disk.

INDICE

1. Introdução 6
1.1. Motivação 6
1.2. Objetivos 8
1.3. Estrutura 9
2. Algoritmos de junção com obrigatoriedade de recebimento de tuplas 10
2.1. Algoritmos de Junção 10
2.1.1. **Algoritmo *Nested-Loop Join* 11**
2.1.2. **Algoritmo *Merge Join* 13**
2.1.3. **Algoritmo *Hash Join* 14**
2.1.4. **Algoritmo *Hybrid Hash Join* 18**
3. Algoritmos de junção sem obrigatoriedade de recebimento de tuplas 22
3.1. **Algoritmo *Double Pipelined Hash Join* 22**
3.2. Algoritmo Progressive Merge Join 24
3.3. Algoritmo *XJoin* 27
3.3.1. Funcionamento do Algoritmo *XJoin* 28
3.4. Algoritmo *MobiJoin* 31
3.4.1. Funcionamento do *MobiJoin* 32
3.5. Algoritmo *Hash-Merge Join* 35
3.5.1. O Funcionamento do algoritmo HMJ 37
3.6. Comparativo entre os algoritmos de junção 40
4. Políticas de liberação de espaço em memória do *Hash-Merge Join* 44
4.1. *Flush All Policy* (FAP) 45
4.2. *Flush Smallest Policy* (FSP) 46
4.3. *Flush Largest Policy* (FLP) 46
4.4. *Adaptive Flushing Policy* (AFP) 47
5. Uma Política Eficiente para Liberação de Memória 50
5.1. *A política* MFP 52
5.2. O algoritmo 59
5.3. Discussão comparativa 61
5.3.1. Memória Desbalanceada 61
5.3.2. Memória Balanceada 63
6. Resultados Experimentais 67
6.1. Introdução 67
6.2. O algoritmo 68
6.3. A arquitetura 68
6.3.1. Apresentação 69
6.3.2. Processamento 72
6.3.3. Classes e estruturas 75
6.4. Simulações 77
6.5. Estimativa de custo do algoritmo 78
7. Conclusão e Trabalhos Futuros 80
8. Referências 83

INDICE DE FIGURAS

Figura 1: Paralisação no recebimento de tuplas....11
Figura 2: Funcionamento do *Nested Loop Join*....12
Figura 3: Funcionamento do *Merge Join*....13
Figura 4: Funcionamento do *Hash Join*....15
Figura 5: Exemplo de funcionamento da estratégia Avoidance....16
Figura 6: Exemplo do funcionamento da estratégia Resolution....18
Figura 7: Resumo do funcionamento do HHJ....19
Figura 8: Resumo do funcionamento do PMJ....25
Figura 9: Funcionamento do PMJ....25
Figura 10: Funcionamento do *XJoin*....30
Figura 11: O funcionamento do *MobiJoin*....33
Figura 12: Estado da Execução após Transferência da partição para o disco....34
Figura 13: Algoritmo Hash-MergeJoin....37
Figura 14: Fase de *Hashing* do HMJ....38
Figura 15: Armazenamento do disco no início da fase de *Merging*....39
Figura 16: Exemplo da fase de *Merging*....40
Figura 17: Exemplo de Políticas de Liberação....45
Figura 18: Tabelas Resumo....50
Figura 19: Tabela resumo resultante de liberação de memória do par (6,12)....52
Figura 20: Tabela resumo proposta....53
Figura 21: Exemplos de novas tabelas resumo....53
Figura 22: Exemplos de Tabelas Resumo....55
Figura 23: Exemplo de partições com valores de diferença iguais....56
Figura 24: Exemplo de memória desbalanceada e pares de partições iguais....57
Figura 25: Exemplo de memória balanceada e pares de tuplas iguais....57
Figura 26: Resultado da aplicação do Algoritmo MFP....58
Figura 27: Várias partições com a menor diferença....58
Figura 28: Exemplo de várias tuplas com o mesmo somatório máximo....59
Figura 29: Algoritmo *Mobile Flushing Policy*....60
Figura 30: Comparativo entre as políticas com memória desbalanceada....62
Figura 31: Tabela resumo após o algoritmo AFP, com memória desbalanceada....62
Figura 32: Tabela resumo após o algoritmo MFP, com memória desbalanceada....63
Figura 33: Comparativo entre as políticas com memória balanceada....64
Figura 34: Tabela resumo após o algoritmo AFP, com memória balanceada....64
Figura 35: Tabela resumo após o algoritmo MFP, com memória balanceada....65
Figura 36: Visão Geral da Arquitetura MFP....69
Figura 37: Interface Gráfica....70
Figura 38: Diagrama de classes do pacote HMJ....73
Figura 39: Diagrama de Pacote modelo e classe Tupla....74
Figura 40: Diagrama de classes do pacote UTIL....76
Figura 41: Comparativo de impacto sobre E/S....78
Figura 42: Comparativo das políticas x tempo....78
Figura 43: Tabela resumo exemplo....79

INDICE DE TABELAS

Tabela 1: Comparativo I entre algoritmos de junção 41
Tabela 2: Comparativo II entre algoritmos de junção 41
Tabela 3: Descrição dos critérios utilizados na comparação dos algoritmos 42
Tabela 4: Comparativo entre as políticas AFP e MFP 65

1. Introdução

1.1. Motivação

Com o avanço da tecnologia de dispositivos portáteis e redes sem fio, a computação móvel tornou-se realidade no ambiente computacional moderno. A integração de dispositivos móveis com as redes sem fio possibilita que computadores móveis se interliguem como componentes, ou seja, os dispositivos móveis pode conter, cada um, seus próprios bancos de dados e solicitarem uns aos outros dados e/ou serviços.

De acordo com esse novo cenário, um grande número de computadores móveis podem se conectar através de uma infra-estrutura de comunicação sem fio estando, cada um com seu próprio sistema de banco de dados, podem formar uma comunidade de sistemas de bancos de dados móveis. Uma Comunidade de Bancos de Dados Móveis(MDBC)[2] funciona como uma coleção dinâmica de bancos de dados móveis, distribuídos e autônomos, na qual cada usuário de banco de dados pode acessar cada um dos bancos de dados através de uma infra-estrutura de comunicação sem fio[2].

Em uma MDBC as técnicas de processamento de consultas devem ser adaptadas para lidarem com a instabilidade do ambiente, assim como limitação de recursos, por exemplo memórias limitadas dos dispositivos móveis. Vejamos um exemplo: as fontes de dados podem ter as taxas de entregas de tuplas previstas pelo otimizador de consultas, no entanto devido a uma desconexão de uma das fontes de dados da rede sem fio, tal fonte de dados ficará desconectada da rede e por conseqüência, não poderá entregar suas tuplas temporariamente, logo o operador de junção ficará parado(bloqueado).

Para processar consultas em uma MDBC, os algoritmos de junção precisam ter as características seguintes: (1) produção incremental de resultados à medida que os dados são disponibilizados, (2) continuidade no processamento da consulta mesmo que a entrega dos dados esteja bloqueada, e (3) reação a situações de limitação de memória durante a execução da consulta[1].

Dentre os diversos algoritmos de junção existentes, alguns deles têm as mesmas características supracitadas, para serem usados em uma MDBC como: *Adaptive Symetric Hash Join*(ASHJ)[9]*, XJoin*[14]*, MobiJoin*[1] e *Hash-Merge Join*(HMJ)[3].

O HMJ, *XJoin* e *MobiJoin* se diferenciam do ASHJ devido a existência das características necessárias para processar consultas em uma MDBC: (1) continuidade no processamento e (2) respostas rápidas.

Dos três algoritmos citados no parágrafo anterior, HMJ, XJoin e MobiJoin, apenas o HMJ utiliza uma política de liberação de tuplas da memória adaptativa, ou seja, quando a memória está com sua capacidade máxima de armazenamento, o algoritmo HMJ libera partições da memória de acordo com o estado da mesma. Partições são conjuntos de tuplas de uma fonte de dados. Fontes de dados são as relações de entrada. Essa adaptação ao estado da memória determina o melhor par de partições, sendo cada partição de uma fonte diferente, a ser enviado ao disco de forma que maximize o tempo para o próximo *overflow* de memória. *Overflow* de memória acontece quando a memória alcança sua capacidade de armazenamento. Cada partição pode conter inúmeras tuplas.

O funcionamento do algoritmo HMJ ocorre em duas fases: fase de *Hashing*, onde as junções são feitas em memória e enviadas ao resultado; e fase de *Merging*, onde as junções são feitas em disco. As fases do algoritmo HMJ se alternam, ou seja, durante a fase de *Hashing* quando as relações deixam de enviar tuplas o algoritmo passa à segunda fase, a fase de *Merging*. Quando as tuplas voltam a ser recebidas, o algoritmo retorna à primeira fase. Quando todas as tuplas já foram enviadas, o algoritmo passa definitivamente à segunda fase, a fase de *Merging*.

Durante a fase de *Hashing*, executada em memória, o algoritmo HMJ usa uma política para liberação de tuplas quando memória alcança seu limite de capacidade de armazenamento, enviando algumas partições para o disco. Esta política é chamada *Adaptive Flushing Policy*(AFP)[3].

O objetivo da política AFP é liberar pares correspondentes de partições para o disco com base em dois parâmetros pré-definidos e uma tabela resumo mantida em memória. Apesar de a política AFP ser eficiente ela permite a ocorrência de dois problemas:

- Necessidade de monitoramento constante dos parâmetros. O algoritmo da política AFP trabalha com dois parâmetros: (1) um parâmetro que determina o tamanho mínimo das partições que são enviadas ao disco e (2) um que parâmetro determina se a memória está balanceada. Estes parâmetros precisam ser monitorados, pois caso o algoritmo de junção não encontre uma partição para enviar ao disco, tais parâmetros precisam ser alterados.

- Necessidade de manutenção do balanceamento constante da memória. Isso ocorre com o monitoramento dos parâmetros supracitados e manutenção da mesma ordem de grandeza da cardinalidade das tabelas.
- Não garante que a memória ficará balanceada após o envio de um par de partições ao disco, pois a escolha desse par de partições pode não ser o mais adequado para deixar a memória balanceada. O par de partições mais adequado é o par que libera o maior espaço em memória e ainda deixa a memória balanceada.

Essa dissertação propõe uma nova política para resolver os problemas supracitados. Com a resolução desses problemas, o algoritmo *Hash-Merge Join* terá um melhor desempenho devido ao fato de escolher o par de partições mais adequado para ser enviado ao disco e ainda liberará mais espaço em memória, evitando futuros *overflows* de memória. Reduzindo o nível de parametrização do algoritmo, o mesmo fica mais independente da intervenção do usuário.

1.2. Objetivos

O objetivo dessa dissertação é propor uma nova política de liberação de dados da memória, chamada *Mobile Flushing Policy*(MFP). A política MFP propõe uma otimização da política *Adaptive Flushing Policy*(AFP), mantendo o mesmo objetivo principal, enviar pares de partições para o disco, em caso de ocorrência de *overflow* de memória.

A política AFP libera pares correspondentes de partições para o disco com base em uma tabela resumo mantida em memória e dois parâmetros: (1) balanceamento de memória e (2) tamanho mínimo de partições. A tabela resumo em memória contém o tamanho de cada partição, o somatório dos tamanhos de cada par de partições de ambas as relações e o tamanho total de partições de cada relação.

A política MFP também libera pares correspondentes de partições para o disco com base em uma constante de balanceamento de memória e uma tabela resumo diferenciada da tabela resumo utilizada pela política AFP. A tabela resumo da nova política tem uma coluna a mais discriminando a diferença da cardinalidade de cada par de partições das relações de entrada.

A nova coluna na tabela resumo inserida na política MFP, determina qual par de partições deve ser enviado, dependendo do estado de balanceamento da memória. Caso

a memória esteja balanceada, o par de partições a ser enviado ao disco será aquele que detém a menor diferença entre as cardinalidades das partições correspondentes de cada relação. Quando a memória está desbalanceada, o par de partições a ser enviado ao disco será aquele que detém a maior diferença entre as cardinalidades das partições correspondentes de cada relação.

O objetivo principal da nova coluna é garantir que o par de partições escolhido para ser enviado ao disco deixará a memória balanceada, além de garantir que sempre haverá no mínimo um par de partições a ser enviado ao disco.

1.3. Estrutura

Essa dissertação está organizada como descrito a seguir. O capítulo 2 apresenta os algoritmos de junção com obrigatoriedade de recebimento de todas as tuplas. No capítulo 3 são apresentados os algoritmos de junção sem obrigatoriedade de recebimento de todas as tuplas. Neste capítulo é feito um comparativo entre os algoritmos de junção descriminados nos capítulos 2 e 3 apresentando os motivos do nosso trabalho estar baseado no algoritmo do *Hash-Merge Join*. No capítulo 4 são apresentadas as políticas de liberação de dados da memória, seus objetivos e o funcionamento. No capítulo 5 apresentamos uma política eficiente de liberação de memória. Neste capitulo apresentamos também um comparativo entre as políticas existentes e a MFP e as vantagens da utilização da nova política, bem como o funcionamento do algoritmo da MFP e suas propriedades. As classes e estruturas, a interface gráfica de implementação da política MFP e os resultados experimentais estão detalhados no Capítulo 6, assim como os resultados comparativos, através de gráficos, dentre as políticas AFP e MFP implementadas sobre o algoritmo *Hash-Merge Join*. Finalmente, o capítulo 7 apresenta a conclusão e os trabalhos futuros e no capítulo 8 estão às referências bibliográficas.

2. Algoritmos de junção com obrigatoriedade de recebimento de tuplas

Os algoritmos de junção que têm obrigatoriedade de recebimento de tuplas são aqueles que somente executam a junção se todas as tuplas estiverem disponíveis das fontes de dados.

Neste capítulo são analisados alguns destes algoritmos de junção como: *Nested-Loop Join*, *Merge Join*, *Hash Join* e *Hybrid Hash Join*. A análise destes algoritmos é feita com base nos critérios de desempenho, custo de processamento, vantagens e desvantagens.

A estrutura deste capítulo está organizada da seguinte forma: a seção 2.1 apresenta os quatro algoritmos de junção com obrigatoriedade de recebimento de tuplas, citados no parágrafo anterior.

2.1. Algoritmos de Junção

Os algoritmos que tem a obrigatoriedade no recebimento de todas as tuplas são aqueles que somente executam a junção se todas as tuplas estiverem disponíveis das fontes de dados. Caso as tuplas não tenham sido recebidas completamente, o algoritmo fica paralisado.

A Figura 1mostra um exemplo do que ocorre quando as tuplas não foram completamente recebidas. Neste exemplo, a fonte R paralisou sua entrega de tuplas por conta de uma falha na conexão. Apenas as tuplas A1, A2, A3 e B1 foram entregues e se encontram em memória. Já a fonte S continua enviando tuplas. Neste caso, mesmo que todas as tuplas da fonte S tenham sido recebidas, o algoritmo fica paralisado à espera das tuplas da fonte R, pois este tipo de algoritmo precisa receber todas as tuplas de ambas as relações para poder efetuar a junção. Mesmo que todas as tuplas da fonte S tenham sido recebidas e algumas da fonte R, o algoritmo só inicia a junção quando todas as tuplas de ambas as fontes chegarem à memória.

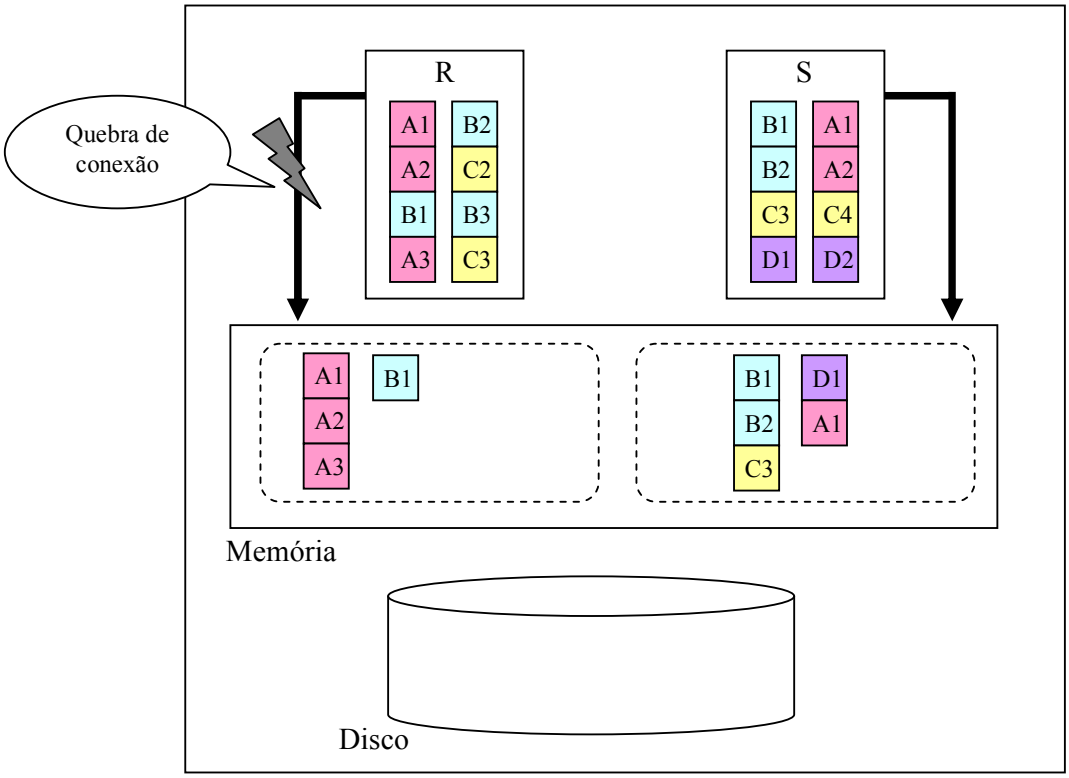


Figura 1: Paralisação no recebimento de tuplas

2.1.1. Algoritmo *Nested-Loop Join*

Para executar a operação de junção r ⋈ s, o *Nested-Loop Join* executa um par de laços aninhados, onde para cada tupla *t*r, da relação *r*, o algoritmo lê todas as tuplas *ts* da relação *s*, considerando que a relação r é a mais externa e a relação s é a mais interna do par de laços aninhados. As tuplas das relações não precisam estar ordenadas pelo atributo de junção. Ao comparar as tuplas, o algoritmo verifica se ambas satisfazem à condição de junção. Caso satisfaça, inclui o par de tuplas no resultado, como pode ser visto no algoritmo descrito na Figura 2.

As informações mais relevantes para o cálculo das estimativas de custo são:

- n_r - número de tuplas de uma relação *r*;
- p_r - número de páginas que contém as tuplas da relação *r*;
- f_r - fator de página de uma relação *r*, ou seja, o número de tuplas da relação *r* que cabe em uma página;

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;

O *Nested-Loop Join* é um algoritmo que tem um processamento caro, devido ao fato da necessidade de haver a leitura de cada par de tuplas das duas relações. Sendo assim, o total de pares de tuplas examinadas equivale a (n_r * n_s). Considerando que no pior caso o buffer armazena somente uma única página de cada relação, a estimativa de custo do algoritmo é dada por $EC = p_r + (n_r * p_s)$. Considerando que no melhor caso o buffer consiga armazenar ambas as relações, de forma que cada relação seja lida apenas uma vez, a estimativa de custo do algoritmo é dada por $EC = p_r + p_s$.

```
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,
6.          então adicione t_r, t_s ao resultado
7.      Fim
8.  Fim
```

Figura 2: Funcionamento do *Nested Loop Join*

Novas propostas surgiram para melhorar a eficiência do *Nested-Loop Join*, como as descritas abaixo:

- *Block Nested-Loop Join*: este algoritmo lê uma página da tabela externa, estando esta página com várias tuplas, enquanto lê a tabela interna. Ele é usado quando o buffer é pequeno para conter ambas as relações por completo na memória. A estimativa de custo deste algoritmo é $EC = p_r + (p_r * p_s)$;
- *Index Nested-Loop Join*: este algoritmo executa a junção através de índices que estão nos atributos de junção de ambas as relações. Ou seja, para cada tupla *tr*, da relação *r*, considerada como a relação mais externa, o índice é usado para pesquisar as tuplas da relação *s*, considerada como a relação mais interna, que satisfaça a condição de junção. A estimativa de custo deste algoritmo é $EC = p_r + (p_r * C)$, onde *C* é o custo de acesso através do índice;

A desvantagem deste algoritmo ocorre quando ele é usado em uma comunidade de bancos de dados móveis, pois há momentos que as relações remotas param de enviar

tuplas, ficando o processo de junção paralisado. Isso ocorre por que o algoritmo só inicia a junção quando todas as tuplas de ambas as fontes chegarem à memória.

2.1.2. Algoritmo *Merge Join*

Para executar a operação de junção r ⋈ s, o *Merge Join* necessita que as tuplas de ambas as fontes de dados estejam ordenadas pelo atributo de junção. Por conseqüência, as tuplas que têm o mesmo valor no atributo de junção estão em ordem consecutiva. Logo, cada tupla só precisa ser lida uma vez. Na Figura 3 está detalhado o funcionamento do algoritmo.

Na Figura 3, o algoritmo lê a primeira tupla da relação *r* e verifica para cada tupla de *s* se o atributo de junção de t_s é maior ou igual ao atributo de junção de t_r. Ao comparar as tuplas, o algoritmo verifica se ambas satisfazem à condição de junção. Caso satisfaça, o algoritmo as inclui no resultado. Em seguida, o algoritmo lê a próxima tupla de *r* e repete os passos anteriormente descritos até que todas as tuplas que satisfaçam a condição de junção estejam no resultado.

```
1.    Ler primeira tupla de r
2.    Para cada tupla ts de s faça
3.    Início
4.        Enquanto (ts [atr-junção] >= tr [atr-junção])
5.        e r não terminou faça
6.        Início
7.            Se o par (tr,ts) satisfizer a condição de junção,
8.            então adicione tr, ts ao resultado
9.            Ler próxima tupla tr de r
10.       Fim
11.   Fim
```

Figura 3: Funcionamento do *Merge Join*

O *Merge Join* é um algoritmo com um processamento mais barato que o *Nested-Loop Join*, devido ao fato da leitura de cada relação apenas uma vez. Por este fato, a estimativa de custo deste algoritmo é $EC = p_r + p_s$, pois a leitura de cada relação só ocorre uma vez.

No entanto ele só funciona com essa estimativa de custo se as relações estiverem ordenadas pelo atributo de junção. Caso uma das relações não esteja ordenada sobre o

atributo de junção, essa relação deve ser ordenada antecipadamente e em seguida executado o algoritmo *Merge Join*. Neste caso, deve-se analisar se a estimativa de custo ainda é melhor que a estimativa do *Nested-Loop Join*, pois a mesma fica alterada para $EC = (P_r * Log_2 P_r) + p_r + p_s$ ou $EC = p_r + (P_s * Log_2 P_s) + p_s$.

Assim como o *Nested Loop Join* a desvantagem deste algoritmo ocorre quando ele é usado em uma comunidade de bancos de dados móveis, pois há momentos que as relações remotas param de enviar tuplas, ficando o processo de junção paralisado.

2.1.3. Algoritmo *Hash Join*

Para executar a operação de junção r ⋈ s, o *Hash Join* não necessita que as tuplas estejam ordenadas pelo atributo de junção, como ocorre com o *Merge Join*, no entanto, este algoritmo mantém uma tabela *hash* em memória principal para cada uma das fontes de dados. A tabela *hash* é criada a partir de uma função *hash* que é usada para particionar as tuplas das relações, formando conjuntos com o mesmo valor *hash* para os atributos de junção. Estes conjuntos são armazenados nas respectivas tabelas *hash* em memória. O particionamento deve ser feito para evitar que todas as comparações, entre os arquivos de entrada, sejam executadas, comparando apenas os itens dos conjuntos com o mesmo valor *hash*. A conseqüência disso é a redução do uso de entrada/saída, haja vista que apenas as partições com o mesmo valor *hash* são comparadas.

Neste algoritmo, a função *hash* mapeia os valores *atr-junção* de 0, 1, ..., até max. f(*atr-junção*) = *end-partição* mapeia um valor do atributo de junção a um endereço de partição. H_{r0}, H_{r1}, ..., Hr_{max} representam as partições das tuplas da relação r, inicialmente vazias. Cada tupla $t_r \in r$ é colocada na partição H_{ri}, onde i = h(tr[*atr-junção*]). O mesmo vale para s, isto é, H_{s0}, H_{s1}, ..., Hs_{max} representam as partições das tuplas da relação s, inicialmente vazias. Cada tupla $t_s \in s$ é colocada na partição H_{si}, onde i = h(ts[*atr-junção*]).

O funcionamento do algoritmo *Hash Join* ocorre 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 de i, então a tupla t_r deverá estar na partição H_{ri} e a tupla ts deverá estar na partição H_{si}. Por conseqüência, as tuplas t_r em H_{ri} só precisam ser comparadas com as

tuplas *ts* em H_{si}. Isto é, as tuplas t_r não precisam ser comparadas com as demais tuplas t_s das outras partições. Este funcionamento está mais detalhado na Figura 4.

O particionamento das relações r e s exige que ambas as relações sejam completamente lidas e escritas, nas respectivas tabelas *hash*. Logo, o custo para esse particionamento é $2 * (p_r + p_s)$. Além disso, as relações precisam ser lidas uma vez mais para serem construídas, 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 não serem preenchidas completamente. 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 $EC = 3 * (p_r + p_s) + (2 * max)$, onde *max* é o numero máximo de tuplas em cada partição.

```
1.  /* particionamento da tabela s */
2.  para cada tupla ts em s faça
3.  Início
4.      i := h(ts [atr-junção]);
5.      Hsi := Hsi ∪ {ts};
6.  Fim
7.  /* particionamento da tabela r */
8.  para cada tupla tr em r faça
9.  Início
10.     i := h(tr [atr-junção]);
11.     Hri := Hri ∪ {tr};
12. Fim
13. /* junção sobre cada partição */
14. para cada i := 0 até max faça
15. Início
16.     ler Hsi e construir um índice hash para tuplas de Hsi na memória principal
17.     para cada tupla tr em Hri faça
18.     Início
19.             utilize o índice hash sobre Hsi para localizar todas as tuplas ts
20.             onde ts [atr-junção] = tr [atr-junção]
21.             para cada tupla ts correspondente em Hsi faça
22.             Ínicio
23.                     adicione tr. ts ao resultado
24.             Fim
25.     Fim
26. Fim
```

Figura 4: Funcionamento do *Hash Join*

Como as partições são trazidas do disco para a memória e comparadas, pode ocorrer *overflow* de memória, haja vista que a memória é limitada.

Para solucionar o *overflow* de memória são sugeridas duas estratégias: Evitar *overflow* (*Avoidance*), quando ocorre a prevenção do *overflow* ou resolver *overflow* (*Resolution*), quando espera que o *overflow* ocorra para solucionar o problema.

Na primeira estratégia, a tabela de entrada é particionada em F partições, em memória principal, antes de ser construída qualquer tabela *hash*. Conseqüentemente ocorre desperdício de uso da memória, pois as partições reservam antecipadamente um espaço da memória mesmo que ainda não tenha necessidade de uso deste espaço. Esta estratégia evita o *overflow* por algum tempo, dependendo do espaço reservado além do necessário inicialmente, pois o espaço reservado em memória é grande o suficiente para as tuplas iniciais da tabela de entrada. Neste caso, como ainda há a possibilidade da ocorrência de *overflow*, após uma grande quantidade de tuplas já recebidas em memória, a solução é a criação de listas dinâmicas em cada partição quando da ocorrência do mesmo. Essa estratégia é muito usada quando se têm grandes arquivos de entrada, evitando o tratamento de *overflow* várias vezes.

Para explicar melhor a estratégia de *Avoidance*, veja a Figura 5. Digamos que o arquivo de entrada tenha 10 tuplas. O algoritmo cria 10 partições na memória com dois espaços para tuplas em cada uma delas, de forma a garantir o retardo da ocorrência de *overflow*. Em seguida, aloca as tuplas nas partições de acordo com suas chaves de busca. Haverá 10 partições sobrando para quando novas tuplas chegarem.

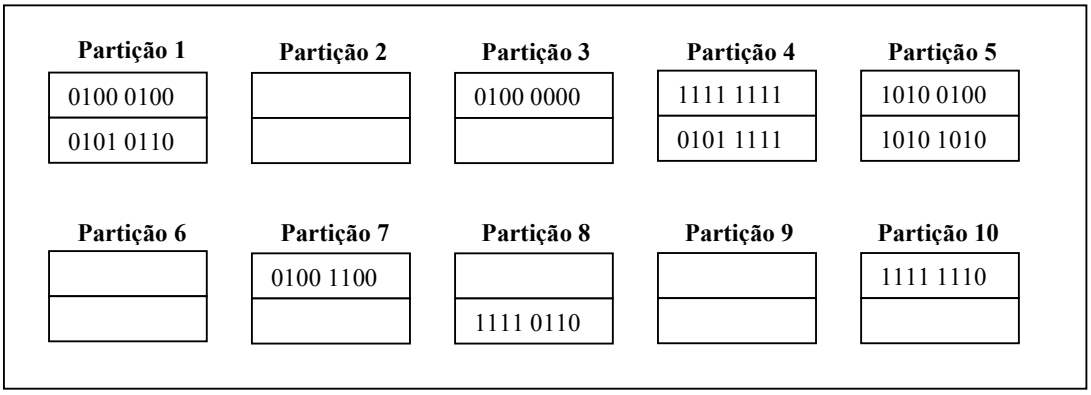


Figura 5: Exemplo de funcionamento da estratégia Avoidance

A segunda estratégia, *Resolution*, tem como premissa a ocorrência de *overflow* para em seguida resolver o problema. Essa estratégia funciona contendo apenas uma partição

com uma quantidade fixa de tuplas inicialmente. Ao ocorrer o *overflow*, esta partição é divida em outras duas partições. Ou seja, à medida que ocorre o *overflow* de cada partição, a mesma é reparticionada para conter as novas tuplas. Essa estratégia é muito usada quando se tem pequenos arquivos de entrada, evitado o desperdício de memória.

Para explicar melhor a estratégia de *Resolution*, vejamos a Figura 6. Digamos que o arquivo de entrada tenha 5 tuplas. As chaves de busca de cada uma delas são: 01000100, 01000111, 01010110, 11111111 e 01010000. Na Figura 6 temos 4 momentos. Inicialmente, a memória está particionada em apenas duas partições. No momento 1 as duas primeiras tuplas com chave de busca 01000100 e 01000111 são recebidas e adicionadas às respectivas partições. No momento 2, quando a terceira tupla é recebida, 01010110, ocorre *overflow* e a memória precisa ser reparticionada. Neste caso o algoritmo reparticiona a memória em 4 partições para conter as duas tuplas já em memória e a nova tupla recém chegada. Para isso, o algoritmo cria um diretório com dois índices, 0 e 1 para determinar onde colocar cada uma das três tuplas. A localização de cada tupla depende do bit mais significativo de cada chave de busca. Assim, as tuplas com chave de busca com o bit mais significativo igual a 0 é apontada pelo índice 0 e as tuplas com chave de busca com o bit mais significativo igual a 1 é apontada pelo índice 1. No nosso exemplo o índice 0 aponta para as tuplas 01000100 e 01010110 e o índice 1 aponta para a tupla 01000111. No momento 3 chega a tupla com chave de busca 11111111. Ela deve ser apontada pelo índice 1. Como há espaço nesta partição, a tupla é apenas adicionada à memória. No momento 4, ocorre a chegada da última tupla 01010000. Ela deve ser apontada pelo índice 0. No entanto, não há espaço na partição, ocorrendo um novo *overflow*. O algoritmo age da mesma forma que anteriormente, executa o reparticionamento da partição que ocorreu o *overflow* e cria um novo diretório de índice. Em seguida, ocorre a recolocação de cada tupla de acordo com seus dois bits mais significativos, haja vista que há dois níveis de diretório. No nosso exemplo da Figura 6, o índice 0 do primeiro diretório aponta o segundo diretório. O índice 0 do segundo diretório aponta para as tuplas 01000100 e 01010000 pois o primeiro e o segundo bit mais significativos são 0 e o índice 1 do segundo diretório aponta para a tupla 01000110, pois o primeiro bit mais significativo é 0 e o segundo bit mais significativo é 1.

Assim como os dois algoritmos anteriores, a desvantagem deste algoritmo ocorre quando ele é usado em uma comunidade de bancos de dados móveis, pois há momentos que as relações remotas param de enviar tuplas, ficando o processo de junção

paralisado. Outra desvantagem do *Hash Join* é a necessidade do conhecimento antecipado da quantidade de tuplas das relações para determinar qual das relações é a de construção e qual das relações é a de busca.

2.1.4. Algoritmo *Hybrid Hash Join*

O *Hybrid Hash Join*(HHJ)[4], é um algoritmo baseado na política de *hash*. Assim como os demais algoritmos baseados nessa política, ele utiliza o *hashing* para melhorar o desempenho da junção das relações de entrada. O *hashing* é utilizado para particionar as relações de entrada, de forma que as tabelas *hash* caibam em memória.

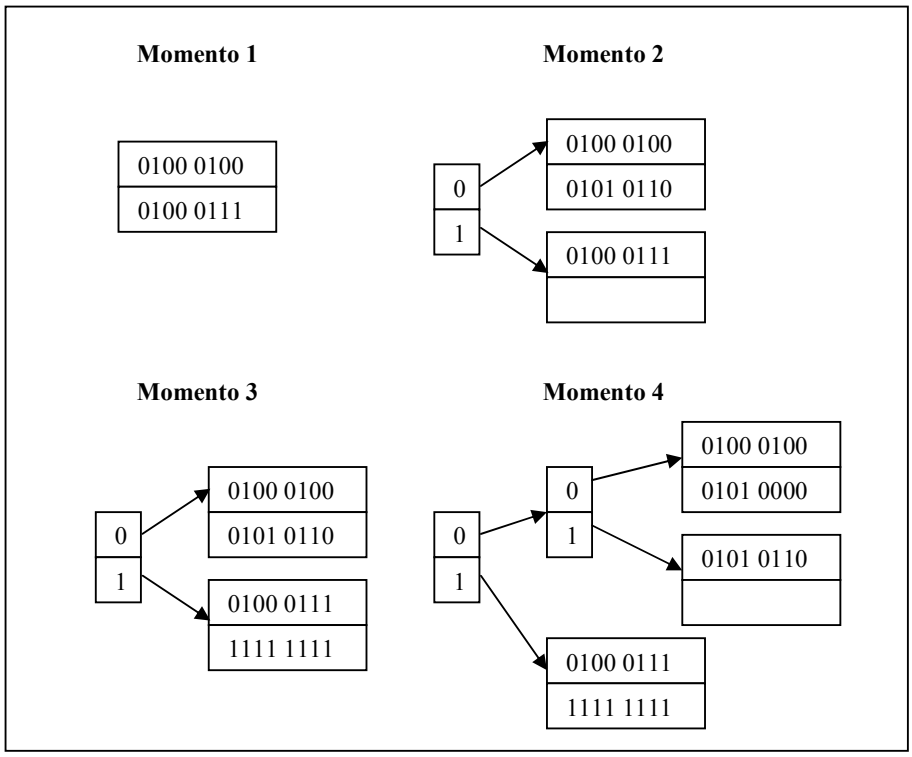


Figura 6: Exemplo do funcionamento da estratégia Resolution

Para executar a operação de junção r $\bowtie$ s, define-se que |r| e |s| são as cardinalidades das relações, considerando que $|r| \leq |s|$. Assim como no *Hash Join*, a menor relação r é chamada de relação de construção, enquanto que a maior relação s é chamada de relação de pesquisa. A junção do HHJ é implementada em dois processos: (1) Processo de Busca e, (2) Processo de Junção, propriamente dito. O Processo de

Busca cria as partições, onde uma delas está em memória e as demais em disco. Em seguida, seleciona as tuplas das relações e aplica a condição de junção sobre as mesmas. Logo depois, o Processo de Junção inicia verificando quais tuplas satisfazem à condição de junção. Similarmente ao algoritmo *Hash Join*, este processo, divide as tuplas das relações dentro das partições já criadas no Processo de Busca, de acordo com a função *hash*, e executa a junção de cada uma delas. Veja um resumo do funcionamento do HHJ na Figura 7 para uma melhor compreensão de suas fases. Note que a parte pontilhada é similar às duas fases do algoritmo *Hash Join*.

Considerando que o número de partições das relações de entrada é B+1, onde B é o número de partições que estão em disco e 1 é a partição em memória. A junção é executada em B+1 fases consecutivas, ou seja, a junção é executada partição a partição. Cada fase do Processo de Junção tem duas operações consecutivas, Construção e Busca. Veja um resumo dos processos que ocorre no HHJ na Figura 7. O algoritmo se divide em dois processos: processo de busca e processo de junção. Este último se sub-divide em outros dois processos: fase de construção e fase de busca.

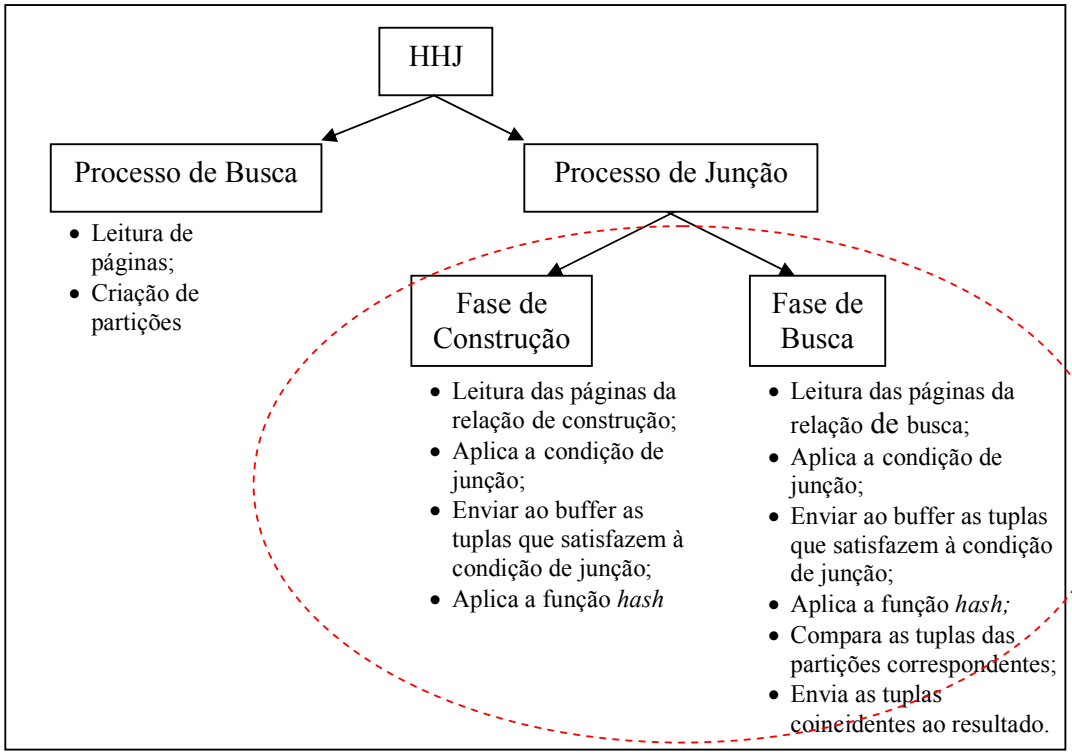


Figura 7: Resumo do funcionamento do HHJ

Analisando o processo de junção, vemos que na fase de construção, o algoritmo lê as páginas da relação de construção r e aplica a condição de junção. As tuplas que satisfazem esta condição são enviadas ao buffer para em seguida passar para a fase de busca. O algoritmo aplica a função *hash* em todas as tuplas do buffer. As tuplas que têm o valor *hash* igual a 0 são inseridas na tabela *hash* da memória. As tuplas com valores *hash* maiores que 0 são inseridas em partições do disco.

Ainda analisando o processo de Junção, vemos que na fase de busca, o algoritmo faz a pesquisa na relação maior, relação s, também chamada relação de busca. Nessa fase o algoritmo busca as páginas da relação s e aplica a condição de junção. As tuplas que satisfazem esta condição são enviadas ao buffer. O algoritmo lê as páginas do buffer e aplica a mesma função *hash* da fase de construção. Se uma tupla tem valor de *hash* igual a 0 o algoritmo busca na tabela *hash* da memória a tupla correspondente da relação *r* e envia o par de tuplas coincidentes para o resultado. Em seguida o algoritmo grava as tuplas com valores de *hash* de 1 até B em uma página em disco.

Cada fase i, onde $0 \leq i \leq B$, tem duas fases, fase i-construção e fase i-busca. Na fase *i-construção*, o algoritmo lê a *i-ésima* partição da relação de construção *r* do disco, uma página por vez, e constrói a tabela *hash* em memória contendo suas tuplas. Em seguida, o algoritmo passa a fase *i-busca*, onde ocorre a leitura da *i-ésima* partição da relação de busca s do disco, uma página por vez. Para cada tupla da *i-ésima* partição, o algoritmo busca, na partição correspondente na tabela *hash* em memória, as tuplas com o mesmo valor *hash* e as inclui no resultado.

Ao final da fase 0, as tuplas das relações r e s da partição 0, ou seja, a partição que está na memória, deve ter passado pelo processo de junção. As demais tuplas, no entanto, estão aguardando os pares correspondentes das partições em disco para efetuar ambas as fases, construção e busca. Em cada uma das fases de 1 a B ocorre a junção das partições correspondentes das relações r e s.

Da mesma forma do *Hash Join* pode ocorrer *overflow* de partições. Neste caso, as duas estratégias sugeridas na sub-seção sobre *Hash Join* podem ser aplicadas à este algoritmo. No entanto, há um modelo desenvolvido em [4] que assume a não ocorrência de *overflow*.

A estimativa de custo do HHJ se divide também em duas fases. A estimativa de custo do Processo de Busca é $EC = 2 * (p_r + p_s)$, pois ocorre a leitura das páginas de ambas as relações e em seguida o particionamento. A estimativa de custo do Processo de Junção é $(p_r + p_s)$, pois as relações precisam ser lidas uma vez mais para serem

construídas, 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 não serem preenchidas completamente. 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 $EC = 3 * (p_r + p_s) + (2 * max)$, similar ao *Hash Join*.

A vantagem deste algoritmo sobre o *Hash Join* é que como as tarefas envolvidas são frequentemente implementadas como uma coleção de processos, estes processos podem ser executados em paralelo. Por exemplo: enquanto o algoritmo executa a leitura das relações no disco (Processo de Busca), o Processo de Junção pode ser executado sobre outras tuplas. Assim, em sistemas de bancos de dados paralelos este algoritmo pode ser implementado para permitir um *pipeline* entre múltiplas operações de junção[17].

A desvantagem deste algoritmo ocorre quando ele é usado em uma comunidade de bancos de dados móveis, pois há momentos que as relações remotas param de enviar tuplas, ficando o processo de junção paralisado. Outra desvantagem do HHJ é a necessidade do conhecimento antecipado da quantidade de tuplas das relações para determinar qual das relações é a de construção e qual das relações é a de busca.

3. Algoritmos de junção sem obrigatoriedade de recebimento de tuplas

Os algoritmos que não tem a obrigatoriedade no recebimento de todas as tuplas são aqueles que executam a junção mesmo que nem todas as tuplas tenham sido recebidas das fontes de dados. Estes tipos de algoritmos iniciam o processo de junção logo que as primeiras tuplas cheguem à memória e assim que ocorra a junção do primeiro par de tuplas este par é enviado ao resultado.

Neste capítulo são analisados alguns destes algoritmos de junção como: *Double Pipelined Hash Join*, *Progressive Merge Join*, *XJoin*, *MobiJoin* e *Hash-Merge Join*.

A análise destes algoritmos é feita com base nos critérios de desempenho, custo de processamento, vantagens e desvantagens.

Este capítulo está estruturado da seguinte forma: a seção 3.1 descreve o funcionamento do algoritmo *Double Pipelined Hash Join*. A seção 3.2 do algoritmo *Progressive Merge Join*. A seção 3.3 descreve o funcionamento do algoritmo *XJoin*. A seção 3.4 seção descreve o algoritmo *MobiJoin*. A seção 3.5 seção descreve o algoritmo *Hash Merge Join* A última seção apresenta um comparativo entre todos os algoritmos, inclusive os algoritmos com obrigatoriedade de recebimento de todas as tuplas para o funcionamento da junção.

3.1. Algoritmo *Double Pipelined Hash Join*

O algoritmo *Double Pipelined Hash Join* (DPHJ)[6] é um algoritmo baseado na política de *hash*. Assim como os demais algoritmos baseados nessa política utiliza o *hashing* para melhorar o desempenho da junção das relações de entrada. No entanto, a grande diferença do DPHJ para o algoritmo de *hash* tradicional é que ele não precisa esperar que todas as tuplas tenham sido recebidas para iniciar a junção.

O DPHJ executa uma junção simétrica e incremental, ou seja, o algoritmo produz resultados à medida que as tuplas são recebidas e não escolhe qual relação deve ser a construção ou busca. O objetivo principal do DPHJ é produzir tuplas quase que imediatamente ao momento que são recebidas e esconder a lentidão na transmissão de tuplas, quando acontece.

A proposta do DPHJ é que ele seja orientado ao comportamento das relações, ou seja, o algoritmo deve executar a junção conforme a entrega de tuplas das relações. Isso significa que se as relações entregarem as tuplas rapidamente, o resultado do algoritmo será enviado para o usuário tão logo ocorram as primeiras junções.

Para executar a operação de junção r $\bowtie$ s, o DPHJ recebe a primeira tupla de uma das fontes de dados, compara esta tupla com as tuplas da tabela *hash* da relação oposta e adiciona esta tupla à tabela *hash* da relação corrente. Os pares de tuplas que tiverem seus valores do atributo de junção coincidentes são enviados imediatamente ao resultado.

A estimativa de custo do DPHJ é melhor que a estimativa do *Hash Join* e do HHJ, $EC = 3 * (p_r + p_s)$ e ainda resolve alguns problemas existentes em ambos os algoritmos:

- As tuplas do resultado são fornecidas aos usuários assim que ocorrem as primeiras junções. Assim, o tempo de saída das primeiras tuplas é minimizado, permitindo ao usuário um resultado parcial rapidamente.
- O operador é simétrico, de forma que não precisa escolher qual é a relação interna e qual é a relação externa, como ocorre, por exemplo, com o HHJ e com o *Hash Join*. Isso é uma vantagem pelo fato desse algoritmo ora lê tuplas da relação r e as comparar com as tuplas da tabela *hash* da relação s, ora lê tuplas da relação s e compará-las com as tuplas da tabela *hash* da relação r. O DPHJ trata ambas as tabelas *hash* como tabela de construção e busca alternadamente. Por este motivo, o algoritmo não fica lento no momento que uma das relações paralisa a entrega de tuplas, pois o algoritmo lê apenas da fonte que está enviando tuplas e as compara com a tabela *hash* da relação paralisada.
- Ocorre uma compensação no processamento quando uma das fontes de dados está lenta, o processamento da outra fonte é executado com mais rapidez, pois o processador fica lendo exclusivamente da fonte que está enviando tuplas. Isso também permite que as consultas usem mais eficientemente a CPU, pois a mesma pode processar os dados de uma relação enquanto espera pelos dados da outra.

Um problema que ocorre no DPHJ é a necessidade de manter duas relações em memória, invés de apenas a menor das relações como ocorre com o *Hash Join* e no HHJ.

Outra desvantagem do DPHJ é quando ocorre a paralisação das tuplas de entrada de ambas as fontes de dados. Neste momento, o algoritmo pára sua execução até que novas

tuplas sejam recebidas. Como conseqüência, o DPHJ não pode ser utilizado em um MDBC.

3.2. Algoritmo Progressive Merge Join

O *Progressive Merge Join*(PMJ)[8,9] é um algoritmo do tipo *Sort-Merge Join*. Os algoritmos Sort-Merge *Join* tem duas fases: fase de ordenação e fase de junção. Estas fases ocorrem em seqüência. Já no PMJ as duas fases podem ocorre em paralelo.

O objetivo do PMJ é a ordenação de ambos os conjuntos de dados de entrada simultaneamente com a criação da primeira execução e progressivamente produzir resultados dos subconjuntos que estão na memória principal.

O PMJ usa uma área de troca, conhecida como *sweep area*, que pode ser usada para suportar todos os algoritmos baseados em ordenação como *Band-Joins*[10], *Temporal Joins*[11], *Spatial Joins*[12] e outros similares.

Para executar a operação de junção r $\bowtie$ s, o PMJ, como os algoritmos baseados no *Sort-Merge Join*, se divide em duas fases: fase de ordenação e fase de junção.

Na primeira fase, o PMJ inicia lendo o máximo de tuplas que conseguir das relações r e s e trazendo para a memória disponível. Ambos os conjuntos de tuplas de r e s, em memória, são ordenados usando um algoritmo como o *Quicksort*[13]. Em seguida ocorre a junção das tuplas ordenadas. Após a junção, os conjuntos de tuplas de *r* e *s* são temporariamente gravados no disco e as tuplas com atributos da junção coincidentes são entregues ao resultado e não mais consideradas.

Em seguida, o PMJ continua a trazer para a memória as tuplas que ainda não foram lidas, ordená-las e executar a junção sobre elas, até que todas as tuplas sejam comparadas.

Na segunda fase, o PMJ executa a junção dos conjuntos de tuplas de *r* e *s* que foram temporariamente gravados no disco na fase anterior. Isso é executado de uma forma que o processo de junção é ativado para cada um dos conjuntos de dados de entrada. As tuplas com atributos de junção coincidentes são entregues ao resultado e as demais tuplas são temporariamente gravadas no disco. Um resumo do funcionamento do PMJ está apresentado na Figura 8.

A Figura 9 mostra um pseudocódigo do algoritmo PMJ. Note que há dois conjuntos de dados de entrada, *r* e *s*. A memória é determinada por M, o número máximo de

conjunto de tuplas que pode ser executado em uma junção é dado por F, RES é o conjunto resultado da junção e Q é um conjunto vazio temporário. Ele servirá para conter todos os conjuntos de tuplas já processadas.

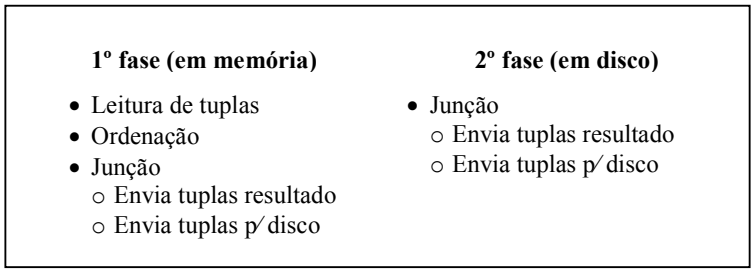


Figura 8: Resumo do funcionamento do PMJ

```
1.    /* fase 1 */
2.    enquanto r ≠ {} e s ≠ {} faça {
3.    Início
4.        ler r'(subconjunto de r); ler s'(subconjunto de s);
5.        onde |r'| + |s'| ≤ M
6         r = r\r'; s = s\s';
7.        Ordenar r' gerando o conjunto r'';
8.        Ordenar s' gerando o conjunto s'';
9.        RES = RES ∪ earlyJoinInitialRuns(r'',s'');
10.       Grava r'' e s'' no disco;
11.       Q = Q ∪ {(r'',s'')};
12.   Fim
13.   /* fase 2 */
14.   Enquanto (|Q| > 1) faça {
15.   Início
16.       /* Escolhe um sub-conjunto de tuplas já processadas na primeira fase */
17.       ler Q'(subconjunto de Q), |Q'| ≤ F/2;
18.       ler (r'',s'');
19.       RES = RES ∪ earlyJoinMergeRuns(Q', (r'',s''));
20.       Grava r'' e s'' no disco;
21.       Q = {Q\Q'} ∪ {(r'',s'')};
22.   Fim
```

Figura 9: Funcionamento do PMJ

Na primeira fase, o PMJ inicia lendo as tuplas das relações *r* e *s* (linha 4). Em seguida, cria sub-conjuntos ordenados (linhas 7 e 8). Após a ordenação, ocorre a junção dos sub-conjuntos através do algoritmo *earlyJoinInitialRuns* (linha 9) e o envio das tuplas coincidentes para o conjunto resultado. As demais tuplas são gravadas no conjunto Q e enviadas para o disco (linha 10 e 11). O algoritmo *earlyJoinInitialRuns*

executa a junção inicial das tuplas em memória. O pseudocódigo deste algoritmo pode ser encontrado em [8] detalhadamente. O PMJ continua a criar pares de seqüências ordenadas até que todas as tuplas de entrada tenham sido processadas (linha 2).

Na segunda fase, o PMJ executa a junção dos sub-conjuntos ordenados. O algoritmo escolhe um sub-conjunto de tuplas já processadas na primeira fase (linha 17). Em seguida lê um par de tuplas deste sub-conjunto (linha18). Com o algoritmo *earlyJoinMergeRuns* ocorre a junção das tuplas deste sub-conjunto (linha 19) e o envio das tuplas coincidentes para o conjunto resultado. As demais tuplas são gravadas no conjunto Q e enviadas para o disco (linha 20 e 21). O algoritmo *earlyJoinMergeRuns* executa a junção das tuplas que foram enviadas ao disco quando as mesmas não cabiam em memória. O pseudocódigo deste algoritmo pode ser encontrado em [8] detalhadamente.

A Estimativa de Custo do PMJ é calculada pelo custo da junção externa, onde cada entrada faz uso de metade da memória disponível. Assim, a estimativa de custo do PMJ é dada pela fórmula $EC = ({}^{N}/_{B} \log {}^{(M-P)}/_{B} {}^{N}/_{B})$, onde B é a quantidade de tuplas que cabe em memória, M é a memória disponível, N é o número de tuplas de entrada e P é o tamanho da área de limpeza.

O PMJ tem a vantagem de satisfazer as seguintes necessidades: entrega rápida dos primeiros resultados. Estes resultados devem ser suficientemente representativos para todo o conjunto de estimativas precisas e necessárias para uma agregação e a manutenção de uma média de velocidade da entrega dos primeiros resultados com as demais entregas.

Uma desvantagem do algoritmo PMJ é que a segunda fase só inicia quando a primeira fase já tiver recebido e ordenado algumas tuplas. Caso ocorra um atraso na entrega inicial de tuplas, o algoritmo pára sua execução e por conseqüência o resultado inicial demora a ser entregue. No entanto, após o recebimento das primeiras tuplas, o algoritmo pode executar a segunda fase com as tuplas em memória e em disco, mesmo que ocorra uma paralisação na entrega de novas tuplas.

Por conta da necessidade de execução da segunda fase após a primeira fase no início do algoritmo, o mesmo não é eficiente quando usado em uma MDBC. Isso também acontece com o algoritmo *Double Pipeline Hash Join*.

3.3. Algoritmo *XJoin*

O acesso aos dados em um ambiente de uma MDBC envolve um grande número de fontes de dados localizadas em áreas remotas, sites intermediários e *links* de comunicação, os quais são vulneráveis a sobrecargas, congestionamento e falhas. Tais problemas podem causar atrasos significativos e imprevisíveis no acesso às informações das fontes remotas.

Para lidar com os atrasos da chegada de dados e produzir resultados iniciais mais rapidamente, surgiu o algoritmo de junção, conhecido como *XJoin*. Esse algoritmo têm duas características importantes [14]: 1) produzir resultados incrementalmente à medida que eles se tornam disponíveis, ou seja, à medida que as tuplas chegam à memória, ocorre o processo de junção e se houver tuplas coincidentes as envia para o resultado; 2) permitir uma execução continua do algoritmo mesmo quando ambas as fontes de dados estiver paralisada ou em atraso.

O *XJoin* é baseado no algoritmo *Symmetric Hash-Join* (SHJ)[9]. Esse, por sua vez foi projetado para permitir um alto grau de *pipelining* em sistemas de bancos de dados paralelos. No entanto, durante quase toda a execução da consulta, o SHJ exige que suas tabelas *hash* sejam mantidas em memória. Por conta dessa restrição o SHJ não pode ser usado quando se tem grandes entradas de dados. Pelo mesmo motivo, sua habilidade para executar múltiplas junções também é bastante restrita. Assim, o *XJoin* surgiu para tentar resolver tais limitações. Ele estende o SHJ permitindo que partes das tabelas *hash* sejam movidas para o disco, utilizando dessa forma, uma menor quantidade de memória. Ele faz isso através do particionamento de suas entradas de dados.

Um processo fundamental do algoritmo *XJoin* é a execução em background de algum processamento quando as entradas estão paralisadas ou atrasadas. Esse processo aproveita os atrasos no recebimento das tuplas das relações remotas para produzir uma maior quantidade de tuplas mais rapidamente. Além desse processo de background, o *XJoin* possui três fases, cada uma delas executadas separadamente[14].

A primeira e a segunda fase do *XJoin* executam alternadamente enquanto ainda há tuplas a serem recebidas das fontes de dados. Já a terceira fase faz uma conferência das tuplas após todo o recebimento. Essa conferência tem dois objetivos: (1) evitar tuplas redundantes, garantindo, no resultado, a presença apenas das tuplas coincidentes e (2) comparar as tuplas ainda não comparadas nas duas primeiras fases.

3.3.1. Funcionamento do Algoritmo *XJoin*

Para executar a operação de junção r $\bowtie$ s, o algoritmo *XJoin* inicia pela primeira fase. Nesta fase ocorre a junção das tuplas de ambas as fontes, *r* e *s*, residentes em memória, agindo de forma similar ao SHJ. O algoritmo nessa fase particiona as tuplas de *r* e *s* em tabelas *hash*. Após o particionamento, as partições correspondentes das tabelas *hash* são comparadas buscando tuplas coincidentes. Se a capacidade máxima de memória para efetuar as junções for alcançada, as tuplas de uma partição são enviadas ao disco deixando espaço para novas tuplas. Para enviar tais tuplas ao disco, o algoritmo escolhe a maior partição entre as partições das fontes *r* e *s*, liberando espaço em memória. Caso apenas uma das fontes de dados pare de enviar tuplas, a primeira fase continua executando as junções comparando os dados já recebidos da fonte paralisada com os dados da fonte ativa. Caso ambas as fontes estejam bloqueadas, o algoritmo passa para a segunda fase. A primeira fase termina quando todas as tuplas de ambas as relações forem recebidas.

A segunda fase faz a junção das tuplas que foram enviadas para o disco por causa de restrições de memória. Essa fase é iniciada sempre que a primeira fase ficar bloqueada devido à falta de tuplas de entrada. Nessa fase uma partição do disco é escolhida e as tuplas dessa partição são enviadas para a memória. Em seguida são comparadas com as tuplas da partição correspondente. Após uma partição do disco ter sido completamente processada, o algoritmo verifica se uma das fontes paralisada voltou a produzir tuplas. Caso positivo, a segunda fase pára e a primeira fase é reiniciada. Caso contrário, outra partição é escolhida do disco para ser enviada à memória e o processo supracitado repetido. Caso não haja tuplas coincidentes entre as tuplas da partição que foi para a memória e as tuplas da partição da relação oposta, as tuplas do disco não podem ser descartadas, pois ainda poderá haver coincidência com as tuplas que ainda não chegaram.

A terceira fase é uma fase de conferência. Ela começa após todas as tuplas de ambas as fontes *r* e *s* terem sido recebidas. Essa fase garante que todas as tuplas que devem estar no resultado final foram produzidas. Essa fase é necessária para garantir que algumas tuplas não comparadas nas duas primeiras fases sejam comparadas, assim como evita tuplas duplicadas no resultado. Para resolver esse problema o *XJoin* utiliza o mecanismo de prevenção de duplicatas baseado em *timestamps*[15].

Nessa fase todas as partições são carregadas do disco para a memória, uma por vez. Cada uma delas é comparada com a partição correspondente da relação oposta. Caso haja tuplas coincidentes, o algoritmo as envia ao resultado e escolhe outra partição do disco. Uma vez que todas as partições tenham sido conferidas a junção está completa.

Timestamps são contadores incrementados toda vez que uma nova tupla é recebida da fonte de entrada ou enviada ao disco. Neste algoritmo são usados dois *timestamps* que são inicializados durante a primeira fase:

- ATS (*Arrival Timestamp*), determina o momento em que a tupla é recebida na memória pela fonte de dados;
- DTS (*Departure Timestamp*), determina o momento em que a tupla é enviada para o disco.

Juntos eles determinam o intervalo de tempo em que cada tupla ficou na memória.

Os *timestamps* ATS e DTS evitam as tuplas duplicadas no resultado produzidas na primeira fase, mas eles não resolvem o problema de tuplas duplicadas na segunda fase. Assim o algoritmo usa mais um *timestamp* para registrar o tempo em que cada partição ficou no disco, durante a segunda fase. Mais detalhes sobre o funcionamento dos *timestamps* ver em [14].

Para ilustrar melhor o funcionamento das fases do *XJoin*, vamos utilizar o cenário da Figura 10. Vamos considerar que as fontes de dados *r* e *s* contém 8 tuplas cada uma. A fonte *r* tem as tuplas: *A1, A2, B1, A3, B2, C2, B3* e *C3*. A fonte s tem as tuplas: *B1, B2, C3, D1, A1, A2, C4* e *D2*. A relação *r* está dividida em três partições: *A, B* e *C*. A relação *s* está dividida em quatro partições: *A, B, C* e *D*. Tais partições são determinadas a partir da aplicação da função *hash* sobre cada atributo de junção de cada tupla.

Na Figura 10, a memória já recebeu algumas tuplas: *A1, A2*, *B1* e *A3* da relação r e *B1* e *B2* da relação s. Por enquanto o disco ainda está vazio, pois a memória ainda não alcançou sua capacidade máxima de armazenamento. Vamos considerar que a capacidade máxima da memória seja de seis tuplas. Foi executada a junção das tuplas *B1* de ambas as relações e as mesmas já estão no resultado.

Caso uma nova tupla chegue para ser processada ocorre um *overflow* de memória. Isso poderá ocorrer a partir do momento que as tuplas *C2* da relação *r* ou *A1* da relação s chegam à memória, pois haveria 7 tuplas em memória e a mesma apenas suporta 6 tuplas. Nesse caso, a maior partição deve ser enviada ao disco. No contexto da Figura 10, a maior partição é a partição *A*, composta por três tuplas, *A1, A2* e *A3*. Em seguida

ao envio, o algoritmo continua a receber novas tuplas das fontes e inserir novamente nas partições da memória.

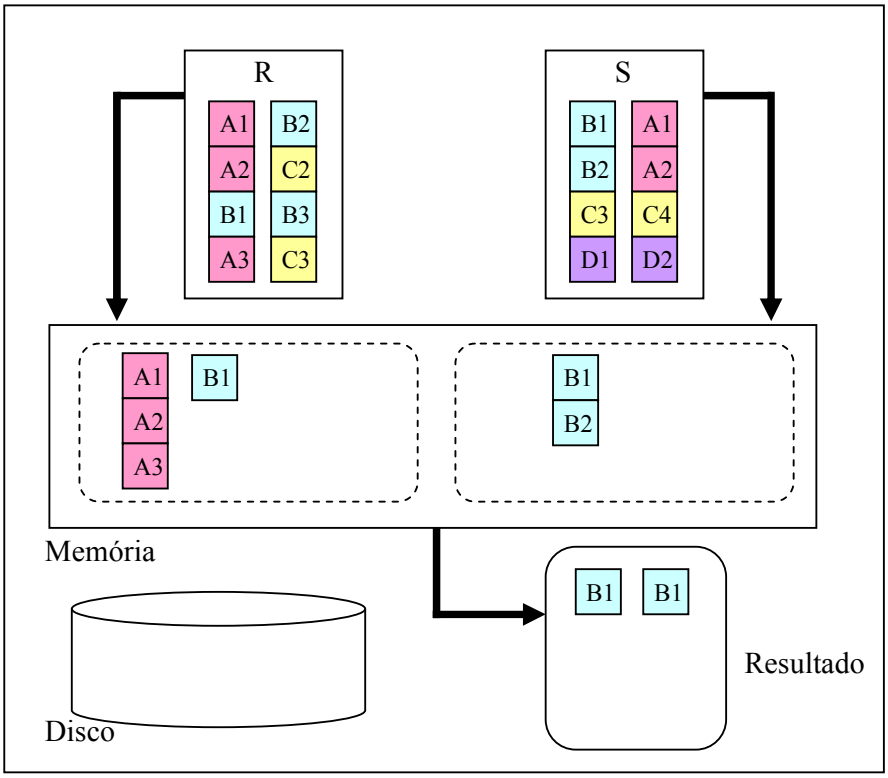


Figura 10: Funcionamento do *XJoin*

Caso as fontes r e s parem de enviar tuplas, o algoritmo passa à segunda fase. Nesta situação, o algoritmo escolhe uma partição do disco, através do seu *timestamp* e envia a partição para a memória, comparando-a com a partição correspondente. Quando pelo menos uma das fontes reiniciar o envio de novas tuplas, o algoritmo retorna para a primeira fase.

As vantagens em se usar o *XJoin* são: (1) execução em background de algum processamento quando as entradas estão paralisadas ou atrasadas; (2) permite que partes das tabelas *hash* sejam movidas para o disco, utilizando dessa forma, uma menor quantidade de memória.

O algoritmo *XJoin* tem três desvantagens: (1) escolher apenas uma partição para enviar ao disco, (2) possibilidade de chegar duas tuplas ao mesmo tempo para serem processadas, e (3) usar *timestamps*.

O primeiro caso é uma desvantagem por que enviando apenas uma partição ao disco no momento do *overflow* de memória, pode acarretar um número maior das tuplas de

apenas uma das relações, dificultando a junção das tuplas em memória e conseqüentemente tornando o processo total da junção de todas as tuplas mais demorado, pois havendo mais tuplas de uma relação que da outra em memória, ocorre à necessidade de buscar as tuplas em disco mais vezes. O processo de buscar as tuplas do disco para a memória é mais demorado do que apenas ler as tuplas em memória.

O segundo caso é uma desvantagem porque quando chegam tuplas de duas relações ao mesmo tempo para serem processadas, o *clock* da máquina precisa especificar inclusive os milissegundos para que não haja colisão de *timestamps*. Neste caso, dependendo da precisão do *clock* da máquina, o *timestamp* colocado em ambas as tuplas será mesmo[19].

O terceiro caso é uma desvantagem porque usando *timestamps* há a necessidade de um espaço extra em cada tupla para armazenar cada um dos três *timestamps*. Os dois primeiros *timestamps* ATS e DTS ocupam um espaço pequeno, no entanto, a lista de *timestamps* para garantia de não duplicidade na segunda fase, pode ocupar bastante espaço. Além de espaço ocupado, há também a complexidade na implementação dos mesmos, pois para controlar os *timestamps*, deve-se executar uma implementação mais detalhada[15].

3.4. Algoritmo *MobiJoin*

O algoritmo *MobiJoin*[1] é um algoritmo de junção que consegue lidar com os atrasos da chegada de dados e ainda produz resultados iniciais rapidamente.

O *MobiJoin* têm as seguintes características para garantir as condições acima: (1) execução do algoritmo com base na técnica de *pipelining*, ou seja, ele produz tuplas para o resultado de forma incremental, à medida em que as tuplas são disponibilizadas; (2) continuidade no processamento do algoritmo quando o recebimento de tuplas das fontes de dados está bloqueado e; (3) corretude dos resultados produzidos pelo algoritmo.

Assim como o *XJoin*, o algoritmo *MobiJoin*[1] também se divide em três fases. A primeira e a segunda fase executam enquanto ainda há tuplas a serem recebidas das fontes de dados. Já a terceira fase faz uma conferência das tuplas após todo o recebimento, similar ao *XJoin*.

3.4.1. Funcionamento do *MobiJoin*

Para executar a operação de junção r $\bowtie$ s, o algoritmo *MobiJoin* inicia pela primeira fase. O objetivo desta fase é executar a junção da maior quantidade de tuplas possível no espaço de memória disponível. Esta fase é iniciada quando as primeiras tuplas das fontes de dados começam a chegar. A primeira fase é executada enquanto houver tuplas sendo entregues de pelo menos uma das relações, e é finalizada quando todas as tuplas de ambas as relações tiverem sido recebidas. Pode ocorrer interrupção na execução da primeira fase do algoritmo quando ambas as relações bloquearem seus envios de tuplas. Nesse momento, a execução do algoritmo suspende temporariamente a primeira fase e passa a executar a segunda fase. O algoritmo retorna para a primeira fase quando as tuplas voltarem a chegar.

À medida que as tuplas vão chegando, elas são alocadas em partições dentro de tabelas *hash* na memória. Cada relação mantém uma tabela *hash* em memória.

As partições armazenadas em memória possuem o mesmo tamanho. Quando o espaço de uma partição for completamente preenchido e houver necessidade de alocar novas tuplas nesta partição, o conteúdo da mesma é enviado ao disco. Após esta transferência, a memória é liberada e a partição pode receber novas tuplas. Pode ocorrer que após a transferência de partições da memória para o disco, algumas tuplas da partição correspondente que chegarem após tal transferência ainda podem chegar à memória. Isso pressupõe que tais tuplas não foram comparadas com as tuplas que foram enviadas ao disco, podendo gerar resultados incorretos ou incompletos da junção.

Para evitar resultados incorretos devido a não comparação de tuplas de partições correspondentes, por estarem na memória em momentos diferentes, foi definido um mecanismo para identificar quais partições de ambas as relações já foram comparadas. Este mecanismo funciona como a seguir: sempre que uma partição for transferida da memória para o disco, é criada uma tabela de controle, em memória, para o par de partições transferido.

A tabela de controle é formada por linhas e colunas. O número de linhas e colunas da tabela de controle é correspondente ao número de tuplas recebidas de cada relação. Na criação da tabela uma das relações é definida como linha e a outra relação como coluna. Cada célula da tabela pode assumir valores 0 ou 1. O valor 0 indica a não comparação das tuplas por ela representada e o valor 1 indica a comparação das tuplas por ela representada. Inicialmente, as células da tabela apresentam o valor 1, uma vez

que, neste momento podemos garantir que todas as tuplas do par de partições foram comparadas.

O cenário da Figura 11 descreve uma memória principal composta pelas tabelas *hash HA* e *HB*. Cada tabela *hash* é formada por partições das relações *A* e *B*. Os nomes das partições em memória são, respectivamente, MA_1, *MA2,* MB_1 *e* MB_2. Dentro de cada partição pode conter até três tuplas. Note que a partição MA_1 está com três tuplas e a partição MB_1 está com duas tuplas. As demais partições estão vazias, inicialmente. As partições correspondentes são comparadas através do atributo de junção pela ligação PAB_1 e PAB_2. O disco está vazio e ainda não existe tabela de controle alguma.

Considere o cenário da Figura 12, onde é apresentado o estado de execução do *MobiJoin* após a transferência de uma partição cheia para disco e a tabela de controle gerada pelo algoritmo.

Note que as células da tabela de controle, da Figura 12, possuem valor 1. Tal fato indica que as tuplas de MA_1 (que foram descarregadas em disco) já foram comparadas com as tuplas de MB_1.

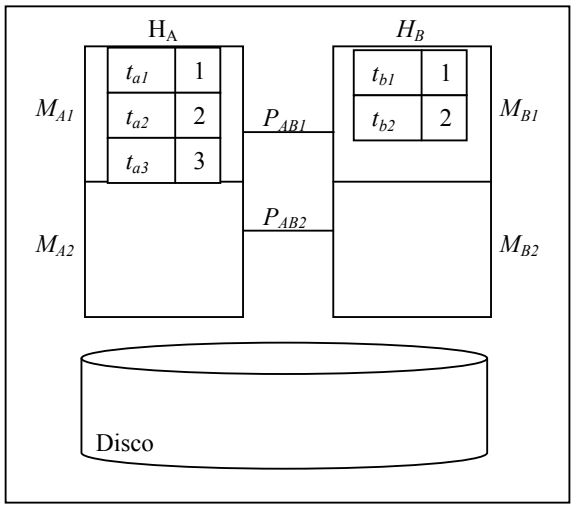


Figura 11: O funcionamento do *MobiJoin*

Após a construção da tabela de controle para um par de partições, esta tabela precisa ser redimensionada sempre que uma nova tupla chegar e necessitar ser alocada em uma das partições do par de partições. Ou seja, uma nova linha ou coluna deve ser adicionada à tabela quando uma nova tupla for alocada em uma das partições do par de partições.

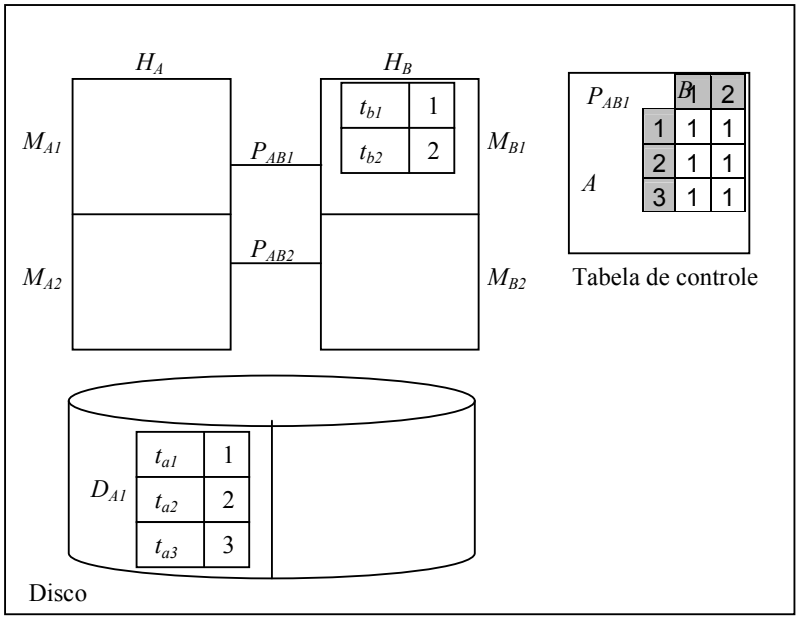


Figura 12: Estado da Execução após Transferência da partição para o disco

Uma vantagem do *MobiJoin* sobre o *XJoin* é a otimização do uso da memória disponível a partir do momento que ocorre uma verificação para analisar se todas as tuplas da relação oposta já foram recebidas. Neste caso, só é necessário armazenar esta tupla na partição em memória se a partição correspondente da relação oposta possuir uma partição em disco. Caso contrário, como a relação oposta já foi completamente recebida, pode-se garantir que todas as tuplas da partição correspondente já chegaram e estão armazenadas em memória. Portanto, as próximas tuplas que estão chegando só precisam ser comparadas com as tuplas da partição em memória da relação oposta e podem ser descartadas após a comparação.

Para evitar tuplas duplicadas no resultado, o algoritmo usa a tabela de controle. O controle é feito através das células desta tabela. Antes de ler as tuplas da partição em disco de uma das relações, o *MobiJoin* lê a tabela de controle associada a partição e obtém as tuplas com valor 0 desta relação. Em seguida o algoritmo lê as tuplas não processadas da partição em disco da mesma relação e compara com as tuplas da partição correspondente em memória.

Após o processamento de todas as tuplas da partição em disco, o algoritmo verifica se alguma das relações já voltou a enviar tuplas. Se o bloqueio de tuplas ainda permanecer, outra partição de disco é processada, caso contrário, a execução desta fase é suspensa e a primeira fase é reiniciada.

A terceira fase inicia quando todas as tuplas de ambas as relações já foram recebidas, da mesma forma que a terceira fase do *XJoin*. Essa fase analisa cada tabela de controle verificando quais tuplas ainda não foram processadas, em seguida as compara com tuplas da partição correspondente em memória da relação oposta.

Como o *MobiJoin* estende o *XJoin* efetuando duas melhorias: substituição dos *timestamps* pela tabela de controle e uma otimização do uso da memória disponível. Conforme [20], o custo do *MobiJoin* pode ser estimado de acordo com as seguintes condições:

- $s \geq 2c$. Neste caso não há *overflow* de memória. Assim, nenhuma tupla é liberada para disco e por conseqüência a terceira fase do algoritmo não ocorre. Neste caso EC = 2c, para leitura de ambas as relações.
- $s/2 < c \leq s$. Neste caso há *overflow* de memória de algumas tuplas, assumindo que a distribuição das tuplas é feita nas tabelas *hash*. O custo é dado por $2c + (c/2 + c/2) + (c/2 + c/2)$, ou seja, $EC = 4c$. Note que o termo $(c/2 + c/2)$ representa o custo de liberação de tuplas par ao disco e leitura de tuplas do disco para execução da terceira fase do algoritmo.
- $s < c$. Nesta situação há *overflow* de $(2c – s)$ tuplas, ou seja, um overflow de (c – s/2) tuplas para cada tabela. Assim $EC = 2c + (2c – s) + (2c – s)$, simplificando $EC = 2(3c - s)$

A desvantagem do *MobiJoin* é que ele escolhe apenas uma partição para enviar ao disco. Isso pode acarretar um desbalanceamento da memória, ou seja, pode haver mais tuplas de uma das relações que de outra, dificultando a coincidência de tuplas. Isso ocorre também com o *XJoin*.

3.5. Algoritmo *Hash-Merge Join*

O algoritmo *Hash-Merge Join*(HMJ)[16] é um algoritmo que tem como objetivo dar continuidade ao processamento das junções de tuplas, mesmo que ambas as fontes de dados de entrada estejam bloqueadas, além de diminuir o tempo de produção das primeiras tuplas do resultado, assim como os algoritmos *XJoin*, *MobiJoin*, *Ripple Join* e *Progressive Merge Join*.

O HMJ estende dois algoritmos: o algoritmo *XJoin* e o algoritmo *Progressive Merge Join* (PMJ). Ambos os algoritmos são baseados no tradicional algoritmo *Hash Join*[17,18].

O algoritmo HMJ estende o *XJoin* quando transfere duas partições correspondentes para o disco, ao invés de apenas uma, como ocorre com o *XJoin*. Dessa forma, o algoritmo HMJ garante o balanceamento mais eficiente da memória, haja vista que enviando um par de partições, haverá espaço liberado na memória de ambas as relações. Logo, se a memória se encontra balanceada, a tendência é que a mesma se mantenha balanceada, partindo do pressuposto que os pares de partições escolhidos sejam os mais indicados para manter a memória balanceada. Memória balanceada é quando existe na memória um número semelhante de pares de tuplas das duas relações de entrada, garantindo que ambas as relações estão ocupando praticamente a mesma quantidade de espaço na memória.

Os pares a serem enviados ao disco, são de responsabilidade da política de liberação de memória. Descreveremos no próximo capítulo diversas políticas de liberação de memória, assim como as características, vantagens e desvantagens de cada uma.

O algoritmo HMJ estende o PMJ, no momento em que ambos ordenam suas tuplas antes de enviá-las ao disco. No entanto, no momento do *overflow* de memória, ou seja, quando a memória alcança sua capacidade máxima de armazenamento, o HMJ usa uma política de liberação de partições da memória denominada *Adaptive Flushing Policy* (AFP) para escolher qual partição ordenada internamente deve ser enviada ao disco. Esta política se adapta ao espaço livre da memória e ao tamanho das partições das tabelas *hash*. Veremos maiores detalhes da AFP no próximo capítulo. Já o PMJ envia todas as tuplas ordenadas que não tiveram tuplas coincidentes com as tuplas da relação oposta para o disco, pois o PMJ não particiona as tuplas.

Outro fato que o HMJ estende o PMJ ocorre no momento em que ele alterna o controle entre as fases várias vezes até que todas as tuplas tenham sido recebidas. Essa alternância facilita a geração dos resultados mesmo que ocorra a paralisação das tuplas de entrada, o que não ocorre no PMJ, pois o mesmo só passa à segunda fase quando todas as tuplas foram recebidas, deixando de gerar resultados quando ocorre algum bloqueio no envio de tuplas das relações.

3.5.1. O Funcionamento do algoritmo HMJ

Para executar a operação de junção r ⋈ s o HMJ inicia com a fase de *Hashing* onde as tuplas de entrada são recebidas a partir das fontes *A* e *B*. As tuplas que estão chegando são armazenadas em partições, na memória, baseadas nos valores de *hash*. Na fase de *hashing*, ocorre a junção das tuplas, em memória, e em seguida é gerado o resultado. Uma vez que a memória está na sua capacidade máxima de preenchimento, as partições da memória são enviadas ao disco. Se ambas as fontes, estão bloqueadas, por alguma razão, o algoritmo HMJ transfere o controle para a fase de *Merging*. Na fase de *Merging*, é executada a junção das tuplas que foram enviadas previamente ao disco. Assim, mesmo que ambas as fontes estejam bloqueadas o algoritmo produz resultados, executando a junção das tuplas em disco já lidas anteriormente. Se o bloqueio de uma das fontes for resolvido, o HMJ retorna à fase de *Hashing*. Veja que a Figura 13 apresenta um resumo das fases do HMJ.

O algoritmo HMJ alterna a execução entre as duas fases, *Hashing* e *Merging*, até que todos os dados de entrada sejam processados. Então, a fase de *Merging* toma o controle total e definitivo para finalizar a execução do mesmo.

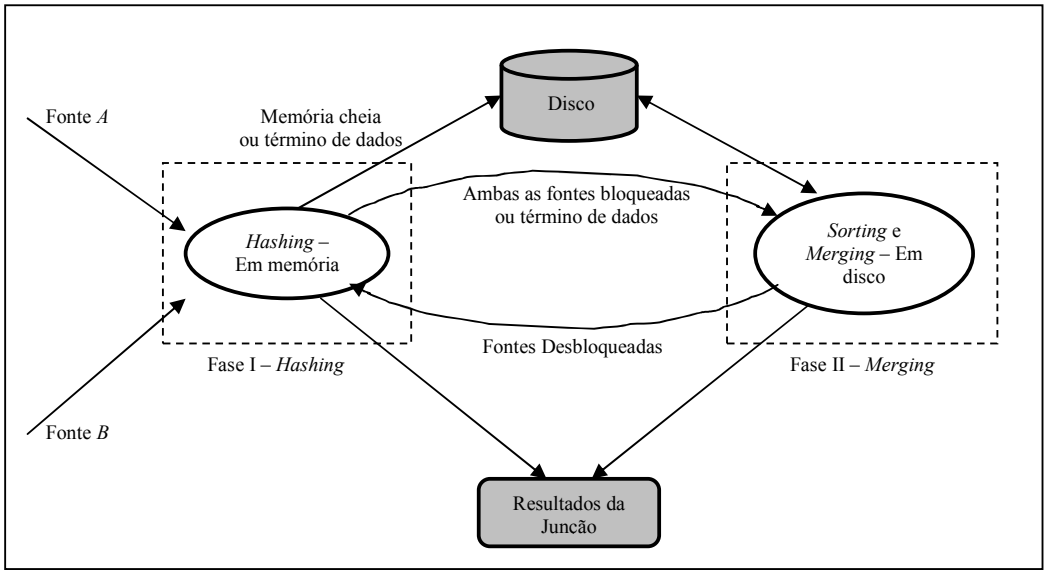


Figura 13: Algoritmo Hash-MergeJoin

A fase de *Hashing* do HMJ é similar à primeira etapa do algoritmo *XJoin*. No entanto, há duas grandes diferenças: (1) O HMJ seleciona duas partições com o mesmo

valor de *hash* para serem enviadas ao disco, uma de cada fonte. Já o algoritmo *XJoin* escolhe apenas uma partição de uma das fontes para enviar ao disco; (2) No HMJ, as partições que serão enviadas ao disco precisam ser ordenadas em memória antes de serem enviadas ao disco.

Conforme mostrado na Figura 14, a fase de *Hashing* mantém duas tabelas *hash* em memória, cada uma com *N* partições, para cada uma das fontes *A* e *B*. Como as partições podem ter tamanhos diferentes, a memória não é dividida igualmente para cada uma das tabelas *hash*.

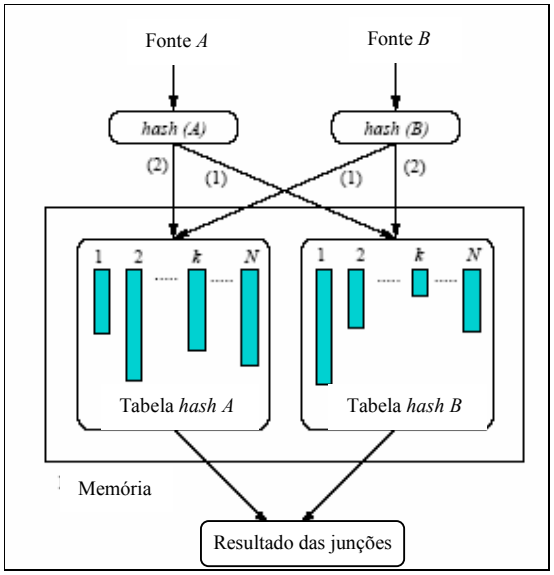


Figura 14: Fase de *Hashing* do HMJ

A fase de *Merging* do HMJ é executada em disco. Tal fase trabalha com as partições que foram previamente jogadas para o disco durante a fase de *Hashing*. A Figura 15 mostra como está o disco no início da fase de *Merging*. Para cada partição com valor de *hash h*, existem m_h páginas de disco para as fontes *A* e *B*. As m_h páginas indicam que esta partição foi escolhida para ser enviada ao disco m_h vezes na fase de *Hashing,* disponibilizando espaço em memória para novas tuplas com o mesmo valor de *hash h*. Note que a partição 1 da Figura 15 tem quatro páginas em disco. Tal partição necessitou liberar espaço em memória quatro vezes para novas tuplas das relações de entrada, enquanto a partição 2 da Figura 15 tem apenas duas páginas, determinando que a partição com valor de *hash* 2, necessitou liberar espaço em memória duas vezes para novas tuplas das relações de entrada.

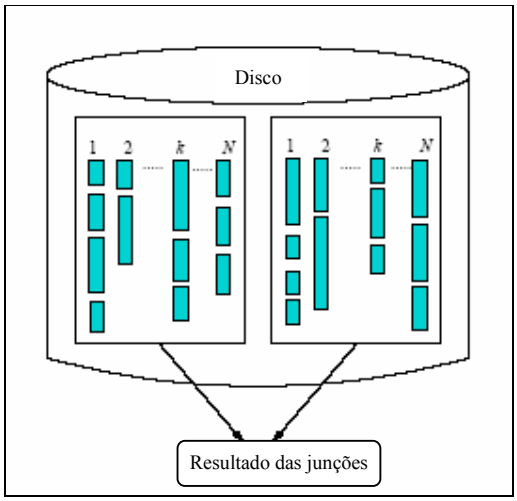


Figura 15: Armazenamento do disco no início da fase de *Merging*

A principal idéia da fase de *Merging* é aplicar uma versão otimizada do algoritmo *Sort-Merge Join* para cada partição individualmente. As otimizações sobre o *Sort-Merge Join* são duas: (1) Os resultados das junções são produzidos durante a fase de *Merging* para evitar o bloqueio nas etapas de *Sorting* e *Merging*. (2) Para evitar a produção de resultados duplicados, as tuplas que resultam de partições com o mesmo valor *hash* não são produzidas, pois estas tuplas já foram enviadas ao resultado na fase de *Hashing*.

A Figura 16 mostra um exemplo da fase de *Merging* do algoritmo HMJ. Uma partição da fonte A contém duas páginas A_{b1} e A_{b2}. Uma partição da fonte *B* contém duas páginas correspondentes B_{b1} e B_{b2}. Na fase de *Hashing* é executada a junção dos pares de (A_{b1}, B_{b1}) e (A_{b2}, B_{b2}) quando as tuplas (4,4) e (6,6) são produzidas. Na fase de *Merging*, só há a necessidade de executar a junção do par de blocos que ainda não foi processado, (A_{b1}, B_{b2}) e (A_{b2}, B_{b1}).

A fase de *Merging* do algoritmo HMJ é similar à fase de *Merging* do algoritmo *Progressive Merge Join*(PMJ), no sentido que ambos os algoritmos aplicam uma otimização no algoritmo tradicional *Sort-Merge Join*. No entanto, há duas diferenças: (1) O HMJ aplica o *Sort-Merge Join N* vezes para as *N* partições, para garantir a totalidade e não redundância dos resultados, enquanto que o PMJ aplica o *Sort-Merge Join* apenas uma vez para todas as partições de cada fonte de dados. (2) O HMJ alterna o controle entre as fases de *Merging* e *Hashing* enquanto que no PMJ a fase de *Merging*

inicia sempre após os dados já terem sido lidos pela fase anterior, além desta ser processada em memória produzindo as tuplas (1,1) e (7,7).

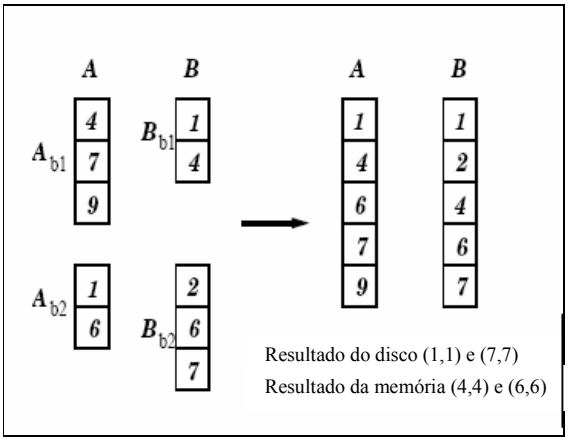


Figura 16: Exemplo da fase de *Merging*

Para estimar o custo da ordenação no HMJ, segundo [20] consideramos que k é o número de partições de cada tabela *hash* e n é a cardinalidade de cada relação envolvida na operação de junção. Considerando que as tuplas de ambas as relações são uniformemente distribuídas entre as partições de ambas as tabelas *hash*, $EC = {}^{n}/_{k} \log {}^{n}/_{k}$, para a ordenação em uma partição. Assim o custo depende da ordenação de todas as tuplas durante a execução do HMJ é $EC = 2n \log {}^{n}/_{k}$.. Observe que o custo depende da cardinalidade de ambas as relações envolvidas na operação de junção, podendo aumentar dramaticamente o tempo de resposta da consulta.

3.6. Comparativo entre os algoritmos de junção

Essa seção apresenta um resumo das características mais importantes dos algoritmos de junção descritos anteriormente. O objetivo desta comparação é apresentar os motivos que nos levaram a escolher o algoritmo *Hash Merge-Join* como base para esse trabalho.

A Tabela 1 e Tabela 2 apresentam nas suas colunas oito critérios importantes dos algoritmos de junção: alternância de execução entre as fases do algoritmo, se as fases do algoritmo ocorrem simultaneamente, se ocorre uma continuidade no processamento do algoritmo, caso haja paralisação de entrega de tuplas, se as tuplas são comparadas mais de uma vez para verificação, rapidez na entrega de tuplas inicial, se as relações precisam

estar em memória para o algoritmo ser executado, qual a política de liberação de tuplas da memória para o disco e a quantidade de tuplas que são enviadas ao disco. Nas linhas estão os nove algoritmos descritos nas seções 2.1 e 2.2.

Tabela 1: Comparativo I entre algoritmos de junção

Algoritmos	Continuidade no processamento	Recarga de tuplas já comparadas	Alterna execução entre as fases
Nested-Loop Join	Não	Não	Não
Merge Join	Não	Não	Não
Hash-Join	Não	Não	Não
Hybrid Hash Join	Não	Não	Não
Double Pipelined Hash Join	Sim	Não	Não
Progressive Merge Join	Não	Não	Não
XJoin	Sim	Sim	Sim
MobiJoin	Sim	Sim	Sim
Hash-Merge Join	Sim	Sim	Sim

Para uma melhor compreensão dos critérios utilizados na comparação dos algoritmos, a Tabela 3 descreve cada um deles em maiores detalhes.

Tabela 2: Comparativo II entre algoritmos de junção

Algoritmos	Produção incremental de resultados	Relações em memória	Liberação de espaço	Política de liberação de espaço em memória
Nested-Loop Join	Não	Sim	Não há	Não há
Merge Join	Não	Sim	Não há	Não há
Hash Join	Não	Sim	Não há	Não há
Hybrid Hash Join	Não	Não	Todas as Partições	Balanceada
Double Pipelining Hash Join	Sim	Não	Todas as Tuplas	Balanceada
Progressive Merge Join	Sim	Não	Todas as Tuplas	Balanceada
XJoin	Sim	Não	Uma partição	Desbalanceada
MobiJoin	Sim	Não	Uma partição	Desbalanceada
Hash-Merge Join	Sim	Não	Par de partições	Balanceada

Analisando as tabelas vemos que apenas os algoritmos *XJoin*, *Hash-Merge Join*, *MobiJoin* e *Double Pipelining Hash Join* continuam o processamento mesmo que ambas as relações parem de entregar tuplas, temporariamente.

Outra análise feita sobre as tabelas é que apenas os algoritmos *XJoin*, *Hash-Merge Join*, *MobiJoin* executam uma verificação de tuplas ao final do processamento, de forma a evitar a redundância de tuplas no resultado e garantir a completude do mesmo. Os demais algoritmos não o fazem por que eles apenas executam a junção quando todas as tuplas já estão em memória.

Alguns algoritmos se alternam entre suas fases da execução, ou seja, alguns algoritmos são divididos em fases e se intercalam entre a primeira e a segunda fase. Os algoritmos que permitem essa ocorrência são: *XJoin*, *Hash-Merge Join*, *MobiJoin*. Isso significa que os demais algoritmos precisam esperar para iniciar uma fase somente após o término da outra.

Uma característica importante na execução dos algoritmos é a envio inicial de tuplas ao resultado rapidamente, mesmo que nem todas as tuplas tenham sido recebidas. Dos algoritmos analisados, apenas os algoritmos *XJoin*, *Hash-Merge Join*, *MobiJoin, Double Pipelining Hash Join* e *Progressive Merge Join* detém tal característica.

Manter as relações em memória para executar o algoritmo é uma restrição crítica, haja vista que há relações muito grandes que não cabem em memória, inviabilizando o uso em uma MDBC. Apenas alguns algoritmos têm essa restrição, como: *Nested-Loop Join*, *Merge-Join e Hash Join e Hybrid Hash Join*.

Tabela 3: Descrição dos critérios utilizados na comparação dos algoritmos

Critério	Descrição
Continuidade no processamento	Verifica se o algoritmo continua o processamento mesmo que as relações paralisem a entrega de tuplas.
Recarga de tuplas já comparadas	Verifica se o algoritmo confere se as tuplas já foram comparadas para não gerar um resultado com redundância.
Duas fases simultaneamente	Verifica se as fases do algoritmo são executadas simultaneamente.
Alterna a execução entre as fases	Verifica se o algoritmo alterna na execução das fases.
Entrega de tuplas inicial	Verifica como ocorre a entrega das primeiras tuplas, ou seja, se o algoritmo espera todas as tuplas chegarem para iniciar o processo de junção e em seguida gerar o resultado ou assim que as primeiras tuplas chegam à memória as junções são iniciadas e à medida que houver tuplas coincidentes as mesmas são enviadas ao resultado.
Relações em memória	Verifica se o algoritmo precisa manter as relações em memória para executar a junção
Liberação de espaço	Verifica como o algoritmo libera espaço em memória no caso de ocorrer *overflow*, ou seja, se a liberação é através de uma partição, um par de partições, todas as partições ou todas as tuplas
Política de liberação de espaço em memória	Verifica se o algoritmo utiliza alguma política de liberação de espaço em memória preocupada em manter a memória balanceada.

Devido à grande quantidade de tuplas das relações, alguns algoritmos utilizam políticas de liberação de tuplas da memória para envio das mesmas ao disco quando

ocorre *overflow*. A política escolhida pode determinar o bom ou mau desempenho do algoritmo, pois a forma como é processada a seleção das tuplas e a quantidade de tuplas que são enviadas ao disco no momento do *overflow* pode dificultar o processamento posteriormente. Algumas dificuldades no processamento são:

- Desbalanceamento da memória
- Grande número de tuplas redundantes para conferência
- Aumento de entrada/saída de tuplas

Os algoritmos que enviam pares de partições para o disco, quando ocorre *overflow* são apenas dois dos analisados acima: *Hybrid Hash Join* e *Hash-Merge Join*. Além disso, apenas o algoritmo *Hash-Merge Join* utiliza uma política adaptativa de liberação de tuplas, objetivando a manutenção da memória balanceada e ele envia apenas um par de partições, enquanto o *Hybrid Hash Join* envia todas as partições.

Considerando o cenário de uma MDBC, podemos constatar que o HMJ é o algoritmo mais indicado pelos motivos abaixo relacionados:

1. O algoritmo não aguarda todas as tuplas das relações chegarem à memória para iniciar o processo de junção.
2. À medida que as tuplas são recebidas pelas fontes de dados e é efetuado o processo de junção, o algoritmo envia as tuplas para o resultado.
3. Utiliza uma política adaptativa ao processo de liberação de partições de memória, procurando sempre deixar a memória balanceada com o envio de pares de partições ao disco.
4. Executa o processamento da junção mesmo quando as relações paralisam a entrega de tuplas.
5. Garante a não redundância de tuplas no resultado, através da terceira fase de seu algoritmo.

4. Políticas de liberação de espaço em memória do *Hash-Merge Join*

Liberar partições da memória para o disco é uma técnica importante em vários algoritmos, principalmente para o algoritmo HMJ. Nessa seção serão abordadas algumas políticas de liberação de espaço em memória que podem ser utilizadas no HMJ. De acordo com [16], uma política eficiente de liberação de espaço em memória deve atender a três necessidades:

- **Dar suporte à fase de *Hashing***. A política deve sempre manter espaço suficiente na memória de forma que as novas tuplas que cheguem à memória possam produzir resultados. Caso este espaço fique reduzido, quando novas tuplas chegarem à memória o algoritmo paralisa o recebimento de tuplas e executará a liberação de espaço para em seguida voltar a receber as novas tuplas.
- **Dar suporte à fase de *Merging***. A política deve evitar liberar pequenas partições que diminuam o desempenho da fase de *Merging*. Com a liberação de pequenas partições para o disco, a fase de *Merging* terá um processamento maior devido à quantidade enorme de pequenas partições, pois nesta fase ocorre a comparação entre as partições. Ou seja, se houver muitas partições com poucas tuplas, esta fase levará um tempo maior do que se houver poucas partições com poucas tuplas, pois cada par de partições deverá ser enviada do disco para a memória para ser comparada com a partição oposta.
- **Manter a memória balanceada**. A política deve tentar manter a memória balanceada ente as fontes A e B. Para melhor compreender a importância do balanceamento da memória, assumimos o caso onde 90% da memória está alocada para a fonte A e apenas 10% está alocada para a fonte B. Uma nova tupla de A terá pouca chance de achar tuplas correspondentes em B, considerando o fato de as tabelas terem tamanhos semelhantes. Por conseqüência, o desempenho da fase de *Hashing* é reduzido e o desempenho da fase de *Merging* fica muito ruim.

Com o intuito de facilitar o entendimento das políticas de liberação de memória, será utilizado o cenário, extraído de [16] apresentado na Figura 17. Nessa figura, temos uma memória com 100MBytes dividida em partições de acordo com a função *hash*, cinco partições para cada fonte de dados, ou seja, a fonte A utiliza 59Mbytes com suas partições e a fonte B utiliza 41Mbytes com suas partições. Cada partição contém diferentes quantidades de tuplas. Assim, a partição 1 da fonte A tem quatro tuplas, a partição 2 tem onze tuplas, a partição 3 tem treze tuplas, a partição 4 tem seis tuplas e a partição 5 tem vinte e cinco tuplas. Uma política de liberação precisa escolher duas partições para serem enviadas ao disco, uma de cada fonte, com o mesmo valor da função *hash*. Para acelerar o processo de seleção das partições que irão para o disco, é mantida em memória uma tabela resumo que contêm o número de tuplas de cada par de partições de ambas as fontes, além de haver a quantidade total de tuplas de cada fonte, localizado na ultima linha da tabela resumo e a quantidade total de tuplas por partição, localizado em cada linha da tabela resumo.

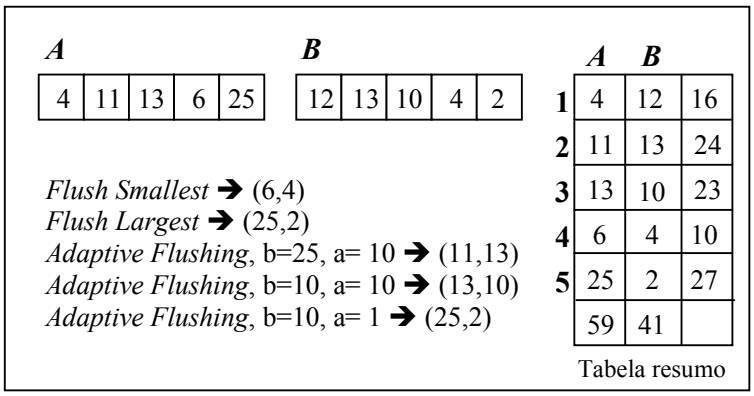


Figura 17: Exemplo de Políticas de Liberação

4.1. *Flush All Policy* (FAP)

Essa política une todas as partições de memória em apenas uma partição. Em seguida, toda a memória é liberada para o disco. A política *Flush All Policy* é utilizada no algoritmo *Progressive Merge Join*[8,9]. Os dois objetivos principais da política *Flush All Policy* são: (1) Liberação de todo o espaço de memória com menos uso de entrada/saída onde às páginas em disco ficam completamente cheias. (2) As partições *hash* são organizadas em disco em grandes blocos, havendo uma

maior eficiência na fase de *Merging*. No entanto, a política FAP resulta em alguns problemas: (1) Após a liberação de todo o espaço em memória, a mesma se encontra vazia acarretando uma demora na produção de novas junções para o resultado, na fase de *Hashing*. (2) A combinação de todas as tuplas em apenas uma partição resulta na junção desnecessária de tuplas em disco. Por exemplo: uma tupla com valor de *hash* h_1 irá ser unida com as tuplas com valor de *hash* h_2.

4.2. *Flush Smallest Policy* (FSP)

Essa política seleciona um par de partições com menor tamanho total de tuplas, para serem enviados ao disco. Por exemplo, na Figura 17 a política FSP escolhe o quarto par de partições (6,4), pois ele contém o menor valor total (10) entre todos os pares de partições em memória. A política FSP é baseada na fase de *Hashing*, tendo como principal idéia manter a memória quase cheia. Dessa forma, as novas tuplas que chegarem terá grandes chances de serem combinadas com outras tuplas já em memória. No entanto, o desempenho fica prejudicado na fase de *Merging* desde que a maioria dos blocos em disco tem tamanho pequeno. Além disso, a fase de *Hashing* resulta em excessivos processos de entradas/saídas devido à grande quantidade de pequenas partições que são liberadas da memória continuamente.

4.3. *Flush Largest Policy* (FLP)

Essa política seleciona um par de partições com maior tamanho total de tuplas, para ser enviado ao disco. Por exemplo, na Figura 17 a política FLP escolhe o quinto par de partições(25,2), pois ele contém o maior valor total igual a 27 entre todos os pares de partições em memória. A política FSP é baseada na fase de *Merging*, tendo como principal idéia sempre ter grandes blocos em disco. Grandes blocos resultam em uma fase de *Merging* eficiente. Ao mesmo tempo, a política FLP não deixa que a memória fique totalmente liberada como ocorre na FAP. No entanto, a FLP tem alguns problemas: (1) selecionar o par de partições com a maior soma pode resultar em liberar pequenas partições. Por exemplo, na Figura 17, um partição de tamanho 2 da fonte B é liberada. (2) Se a memória não está balanceada entre as fontes, ou seja, a fonte A detém 80% da memória e a fonte B detém 20%,

selecionar o par de partições com maior tamanho pode resultar no aumento do desbalanceamento.

Baseado no funcionamento de cada uma das políticas acima podemos fazer uma análise e constatar que cada uma delas infringe pelo menos uma das necessidades básicas de se ter uma política de liberação eficiente. A FAP não dá suporte na fase de *Hashing*. A FSP não dá suporte na fase de *Merging* e a FLP não mantém a memória balanceada e não dá suporte eficiente na fase de *Merging*.

4.4. *Adaptive Flushing Policy* (AFP)

A principal idéia da política *Adaptive Flushing Policy* é a habilidade com que ela se adapta às mudanças que ocorrem no comportamento das fontes. Por exemplo, se a fonte A fica bloqueada, então a memória pode ter mais tuplas da fonte B que da fonte A. Assim, a política *Adaptive Flushing Policy* ajuda a balancear a memória para garantir uma proximidade no número de tuplas de cada fonte.

Nessa política são utilizados dois parâmetros, *a* e *b*, definidos pelo usuário, para selecionar qual par de tuplas deve ser enviado ao disco, tendo como principal objetivo deixar a memória balanceada. O parâmetro *b* é utilizado para adaptar a política ao balanceamento da memória. Esse parâmetro é usado como porcentagem, ou seja, a diferença entre as cardinalidades das relações deve ser menor que o percentual utilizado da memória. Se |A| e |B| são as cardinalidades das tuplas em A e B, respectivamente, para todas as tuplas em memória, então a memória está balanceada apenas se $|A|-|B| < b\%$. O parâmetro *a* representa a quantidade mínima de tuplas necessária dentro de uma partição para que a mesma seja enviada ao disco. Esse parâmetro é utilizado para evitar que partições com poucas tuplas sejam enviadas ao disco, ou seja, os pares de partições devem ter quantidades de tuplas acima do parâmetro *a*.

O funcionamento da política *Adaptive Flushing Policy* ocorre da seguinte forma: Inicialmente, a política pode selecionar dentre todas as possíveis partições, um par de partições.

Se a memória está balanceada, ou seja, $|A|-|B| < b\%$, então a política busca os pares de partições que satisfazem a condição de quantidade mínima de tuplas em cada partição. Se não houve um par de partições que satisfaça essa condição a

seleção se estende a todos os pares de partições, desde que os pares de partições selecionados para irem para o disco não afetem o balanceamento da memória. Após um *overflow* de memória, o principal objetivo da política é obter espaço na memória para poder receber novas tuplas. Dessa forma, se a memória está balanceada e já foram selecionados os pares de partições que satisfazem a condição de quantidade mínima de tuplas, o ideal é retirar deste primeiro sub-conjunto de partições previamente selecionado, o par de partições que ocupe o maior espaço na memória, ou seja, aquele par de partições que tenha a maior quantidade de tuplas dentre as partições.

Se a memória não está balanceada, ou seja, $|A|-|B| > b\%$, a política busca inicialmente os pares de partições que reduzam o desbalanceamento. Por exemplo, se a memória tem mais tuplas de A que de B, então o par selecionado deverá ter mais tuplas de A que de B. Em seguida, a política busca dentro do sub-conjunto de partições previamente selecionado, os pares de partições que satisfazem a condição de quantidade mínima de tuplas em cada partição. Se não houve um par de partições que satisfaça essa condição a seleção se estende a todos os pares de partições deste sub-conjunto. Após um *overflow* de memória, o principal objetivo da política é obter espaço na memória para poder receber novas tuplas. Dessa forma, se a memória está balanceada e já foram selecionados os pares de partições que satisfazem a condição de quantidade mínima de tuplas, o ideal é retirar deste primeiro sub-conjunto de partições previamente selecionado, o par de partições que ocupe o maior espaço na memória, ou seja, aquele par de partições que tenha a maior quantidade de tuplas dentre as partições.

Para facilitar a compreensão vamos analisar o cenário da Figura 17. Vamos considerar também que os parâmetros são: $b = 25\%$ e $a = 10$. Logo, se o total da memória é 100%, então 25% da memória tem como resultado 25. Se as cardinalidades de A e B são respectivamente 59 e 41, então a diferença entre as cardinalidades é 18, ou seja, 59 - 41 = 18. Essa diferença é menor que o parâmetro b, ou seja, 18 < 25. Assim, a memória é considerada balanceada. Então a política busca os pares de partições que satisfazem a condição de quantidade mínima de tuplas, selecionando dois pares de partições, o segundo par (11,13) e o terceiro par (13,10), onde todas as partições têm quantidade de tuplas acima do parâmetro a, ou seja, maior que 10%. Como ambos os pares não afetam o balanceamento da memória, podemos escolher o par de partições com maior tamanho total (11,13). Se

alterarmos o parâmetro para b = 10% e mantivermos o parâmetro a = 10, então a memória é considerada desbalanceada. Nesse caso a política busca inicialmente os pares de partições que reduzam o desbalanceamento. Considerando que no cenário da Figura 17 há mais tuplas de A que tuplas de B, os pares de partições selecionados deverão ter a forma $|A_k| \geq |B_k|$. Assim, os pares de partições escolhidos são o terceiro par (13,10) e o quinto par (25,2). Tendo selecionado os pares que reduzem o desbalanceamento da memória a política seleciona dentre os dois pares dos escolhidos, o par que satisfaça a condição de quantidade mínima de tuplas em cada partição. Nesse caso, apenas o terceiro par (13,10) tem quantidade de tuplas maior que a, ou seja, maior que 10.

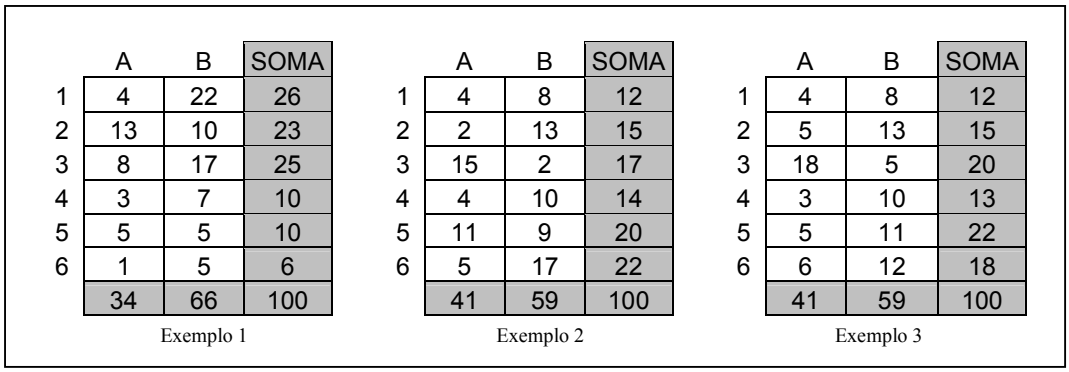
Note que a principal idéia do parâmetro a é não selecionar partições com quantidades pequenas de tuplas para serem enviadas ao disco. Assim, se considerarmos a = 1 e mantivermos b = 10%, a política irá selecionar o par de partições (25,2).

5. Uma Política Eficiente para Liberação de Memória

Após uma análise das políticas de liberação de tuplas da memória usadas pelo algoritmo *Hash Merge Join*, notamos que a mais adequada às necessidades básicas de uma política de liberação eficiente é a *Adaptive Flushing Policy*(AFP). No entanto, alguns problemas, ainda persistem nessa política, como os citados abaixo:

- Permanência de memória desbalanceada;
- Não haver par de partições que satisfaçam os parâmetros quando a memória está desbalanceada;
- Intervenção humana quando o algoritmo não encontra um par de partições que satisfaça os parâmetros para ser enviado ao disco.

Para ilustrar os problemas que ocorrem com a política AFP, iremos utilizar os exemplos da Figura 18. Para contextualizar o exemplo, temos uma memória de 100MB dividida em partições *hash*, cinco partições para cada relação, relação A e relação B. Uma política de liberação precisa escolher duas partições que devem ser enviadas para o disco, uma partição de cada relação, com o mesmo valor *hash* no atributo de junção. Para acelerar o processo de seleção das partições que irão para o disco é mantida em memória uma tabela resumo que contêm a quantidade de tuplas de cada par de partições de ambas as relações, além da quantidade total de tuplas de cada relação.



	A	B	SOMA
1	4	22	26
2	13	10	23
3	8	17	25
4	3	7	10
5	5	5	10
6	1	5	6
	34	66	100

Exemplo 1

	A	B	SOMA
1	4	8	12
2	2	13	15
3	15	2	17
4	4	10	14
5	11	9	20
6	5	17	22
	41	59	100

Exemplo 2

	A	B	SOMA
1	4	8	12
2	5	13	15
3	18	5	20
4	3	10	13
5	5	11	22
6	6	12	18
	41	59	100

Exemplo 3

Figura 18: Tabelas Resumo

Na Figura 18 apresentamos três exemplos de tabelas resumo. Nos três exemplos há seis partições de cada relação A e B. Cada partição contém a sua quantidade de tuplas, ou seja, a quantidade de tuplas que tem o mesmo valor de *hash* no atributo de

junção. As partições correspondentes entre as relações estão lado a lado. No exemplo 1, a quantidade total de tuplas da relação A é 34 e a quantidade de tuplas da relação B é 66. No exemplo 2, a quantidade total de tuplas da relação A é 41 e a quantidade de tuplas da relação B é 59. No exemplo 3, a quantidade total de tuplas de A e de B é a mesma que o exemplo 2. No entanto, a quantidade de tuplas em cada par de partições é diferente.

Para exemplificar os problemas supracitados temos as situações abaixo:

- Caso nenhum par de partições satisfaça os parâmetros *a* e *b*, onde o parâmetro *a* determina o tamanho mínimo das partições, e o parâmetro *b* determina o balanceamento de memória, o algoritmo deverá ser novamente parametrizado para garantir a escolha de algum par de partições e liberar espaço em memória. Aproveitando o cenário da Figura 18, exemplos 1 e 2, e determinando que os parâmetros *a* e *b* são iguais a 10, podemos ver que temos inicialmente uma memória desbalanceada, pois a diferença da quantidade total de tuplas entre as duas relações resulta em 32, no exemplo 1 e 18 no exemplo 2. Em ambos os exemplos, nenhuma partição será escolhida, pois não há um par de partições que tenha a quantidade de tuplas acima do estabelecido pelo parâmetro *a*, ou o parâmetro *a* será desconsiderado, podendo enviar partições ao disco com poucas tuplas.
- Pode ocorrer ainda que o par de partições escolhido para ser enviado ao disco não deixe a memória balanceada. Aproveitando o cenário da Figura 18, exemplo 3, e determinando que os parâmetros *a* e *b* são respectivamente 5 e 10, podemos ver que temos inicialmente uma memória desbalanceada, pois a diferença da quantidade total de tuplas entre as duas relações resulta em 18. Neste caso, é preciso escolher um par de partições para liberar espaço em memória e ainda balancear a mesma. No entanto, o par selecionado deve ser o último par de partições, o par (6,12), pois esse par de partições é o único que as duas partições tem a quantidade de tuplas acima do estabelecido pelo parâmetro *a*. Apesar disso, ao liberar esse par de partições para o disco a memória permanece desbalanceada, pois a diferença da quantidade das relações ainda está acima do estabelecido pelo parâmetro *b*, conforme pode ser visto na Figura 19.

Note que a Figura 19 apresenta apenas cinco pares de partições, pois já ouve a aplicação do algoritmo AFP sobre a tabela resumo, liberando o último par de partições, (6,12). Veja que mesmo havendo espaço em memória a tabela mostra que a memória permanece desbalanceada, pois a diferença entre a quantidade de tuplas ainda é maior que o parâmetro estabelecido de balanceamento. Segundo o cenário descrito acima, o parâmetro de balanceamento da memória é b = 10 e a diferença entre as cardinalidades é igual a 12.

	A	B	SOMA
1	4	8	12
2	2	13	15
3	15	5	20
4	4	10	14
5	11	11	22
	35	47	82

Figura 19: Tabela resumo resultante de liberação de memória do par (6,12)

5.1. *A política* MFP

A política *Mobile Flushing Policy* - MFP propõe duas mudanças na política AFP: (1) uma atualização na tabela resumo mantida em memória com o objetivo de garantir que o par de partições que será enviado ao disco deixará a memória balanceada e, (2) não mais utilizar o parâmetro para definição da quantidade mínima de tuplas, com o objetivo de diminuir a parametrização do algoritmo.

Na política MFP, para saber qual par de partições é o mais apropriado para ser enviado ao disco, propomos uma coluna a mais na tabela resumo contendo a diferença entre as quantidades de tuplas de cada par de partições, proporcionando sempre um balanceamento da memória sem os problemas citados na política AFP. Essa diferença determina a quantidade de tuplas em que cada par de partições correspondentes se distancia umas das outras com relação ao tamanho ocupado, em bytes, pela partição na memória.

Na política proposta vamos nomear o parâmetro de balanceamento da memória como *p_balance*. Esta constante é definida na configuração do algoritmo HMJ.

Na Figura 20 é apresentamos a proposta da nova tabela resumo. Note que nessa nova tabela resumo há quatro colunas. Na primeira coluna estão as partições da

relação A. Na segunda coluna, as partições da relação B. Na terceira coluna está o somatório da quantidade das tuplas de cada par de partições. A quarta coluna é a nova coluna. Nela está à diferença entre a quantidade de tuplas de cada par de partições. Assim como a tabela resumo antiga, ao final da mesma há a quantidade total ocupada pelas tuplas de cada relação.

	A	B	SOMA	DIF
1	4	8	12	4
2	2	13	15	11
3	15	5	20	10
4	4	10	14	6
5	11	11	22	0
6	5	12	17	7
	41	59	100	18

Figura 20: Tabela resumo proposta

Para explicar melhor a nova tabela resumo, vamos considerar o cenário da Figura 21, exemplos 1 e 2. Nesse cenário, vamos considerar que *p_balance* = 10%.

	A	B	SOMA	DIF
1	6	1	7	5
2	10	2	12	8
3	30	2	32	28
4	4	10	14	6
5	2	13	15	11
6	13	7	20	6
	65	35	100	30

Exemplo 1

	A	B	SOMA	DIF
1	11	14	25	3
2	9	7	16	2
3	1	2	3	1
4	17	19	36	2
5	4	6	10	2
6	7	3	10	4
	49	51	100	2

Exemplo 2

Figura 21: Exemplos de novas tabelas resumo

Usaremos como padrão *o p_balance* com 10% por dois motivos: (1) acima deste valor a memória estará balanceada mais vezes, no entanto, a quantidade de tuplas das partições correspondentes tende a se tornar mais distante, sugerindo um desbalanceamento entre as partições das relações, ou seja, alguma relação tende a ter muito mais tuplas que a outra. Isso gera uma incoerência, pois a memória não pode estar balanceada se existe muito mais tuplas de uma relação que da outra; (2) abaixo deste valor a memória estará desbalanceada mais vezes, ocasionando poucos momentos de memória balanceada, dificultando o processamento, pois quando a

memória está desbalanceada, o algoritmo deve escolher um par de partições que satisfaça as condições descritas previamente e ainda deixe a memória balanceada.

A primeira análise que deverá ser feita é se a memória está ou não balanceada. Com esse intuito, deve ser verificado se a diferença entre a quantidade total de tuplas das partições de ambas às relações de entrada estão acima do valor do parâmetro de balanceamento *p_balance*.

Levando em consideração o cenário da Figura 21, exemplo 1, a relação A tem cardinalidade igual a 65 e a relação B tem cardinalidade igual a 35, logo a diferença entre as cardinalidades está acima de 10%, isto é, $|65 - 35| > 10$. Com esse resultado, notamos que a memória está desbalanceada. A partir daí, a política deve procurar um par de partições que a tenha a maior diferença entre suas quantidades de tuplas. No exemplo 1 da Figura 21, o terceiro par de partições, (30,2) é o que tem a maior diferença entre as quantidades de tuplas. Liberando este par de partições para o disco, o estado da memória é alterado de desbalanceada para balanceada.

Levando em consideração o cenário da Figura 21, exemplo 2, a relação A tem cardinalidade igual a 51 e a relação B tem cardinalidade igual a 49, logo a diferença entre as cardinalidades está abaixo de 10, isto é, $|51 - 49| < 10$. Com esse resultado, notamos que a memória está balanceada. A partir daí, a política deve procurar um par de partições que a tenha a menor diferença entre suas quantidades de tuplas. No exemplo 2 da Figura 21, o terceiro par de partições, (1,2) é o que tem a menor diferença entre as quantidades de tuplas. Liberando este par de partições para o disco, a memória permanece balanceada.

Quando a memória está desbalanceada procuramos na tabela resumo as partições com maior valor na coluna da diferença porque quando liberamos esse par de partições para o disco, além de liberar espaço em memória para novas partições serem recebidas, ainda há a tentativa de balancear a memória, ou pelo menos, deixá-la menos desbalanceada. Considerando o exemplo 1 da Figura 22, e considerando o valor de *p_balance* = 10%, vemos que a memória está desbalanceada, pois a diferença entre as cardinalidades das relações está acima do parâmetro de balanceamento. Assim a política deverá escolher o quarto par de partições para ser enviado ao disco.

Por outro lado, quando a memória está balanceada procuramos na tabela resumo as partições com menor valor na coluna da diferença porque quando liberamos esse par de partições para o disco, além de liberar espaço em memória para novas tuplas

serem recebidas mantemos a memória balanceada. Considerando o exemplo 2 da Figura 22, e considerando o valor de *p_balance* = 10%, vemos que a memória está balanceada, pois a diferença entre as cardinalidades das relações está abaixo do parâmetro de balanceamento. Assim a política deverá escolher o terceiro par de partições para ser enviado ao disco.

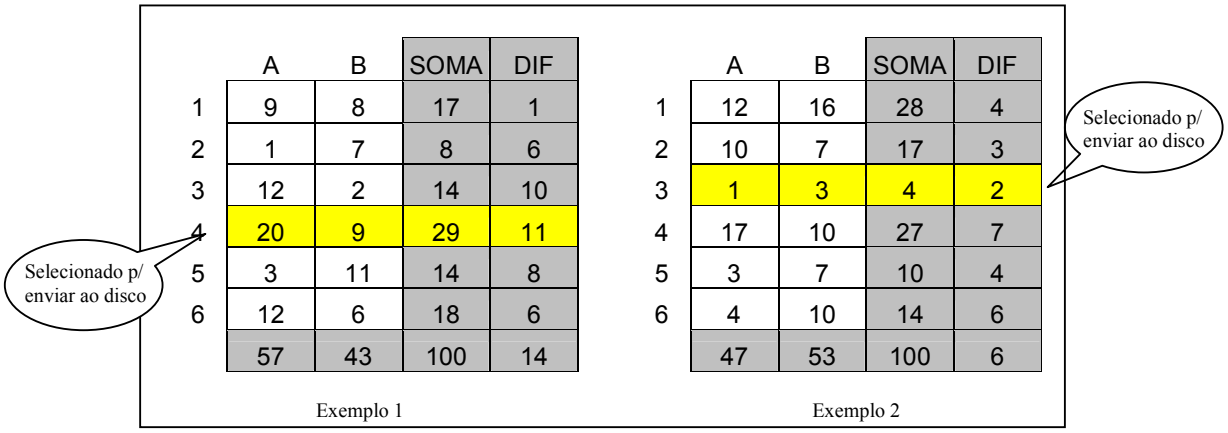


Figura 22: Exemplos de Tabelas Resumo

A coluna SOMA ainda é mantida na nova estrutura da tabela resumo porque quando a memória está balanceada a política deve escolher os pares de partições que têm a menor diferença entre suas quantidades de tuplas. Nesse momento pode haver a ocorrência de vários pares de partições com a mesma diferença pequena. Sendo assim, escolhe-se o par de partições com maior somatório, com a maior quantidade de tuplas, através da coluna SOMA, pois esse par escolhido será o par de partições que liberará mais espaço em memória, aumentando, dessa forma, o tempo para ocorrer o próximo *overflow*.

Para melhorar a compreensão da utilização da coluna SOMA, nesta nova proposta da tabela resumo, vamos considerar o cenário da Figura 23. Considerando que o parâmetro de balanceamento, *p_balance* é igual a 10%, notamos que a memória está balanceada, partindo do pressuposto que a diferença absoluta entre as cardinalidades das relações é 9. Sabendo disso, a política seleciona um par de partições que tenha a menor diferença entre as quantidades de tuplas das relações, isto é, quando a coluna DIF é a menor possível. Conforme o exemplo da Figura 23, há três pares de partições com o mesmo valor da coluna DIF. Os pares são: o quarto par (19,17), o quinto par (5,3) e o sexto par (4,2). Todos eles têm um valor igual a 2

na coluna DIF. Assim, para a política enviar ao disco apenas um desses pares, a mesma deve selecionar aquele que têm o maior valor da coluna SOMA, ou seja, o par que tenha o maior número de tuplas. No exemplo da Figura 23, o par de partições selecionado é o quarto par (19,17), pois ele liberará mais espaço em memória e ainda manterá a memória balanceada.

Vale salientar que no momento que a memória está desbalanceada os pares de partições com a mesma quantidade de tuplas são desconsiderados, pois eles têm a menor diferença possível entre as quantidades de tuplas, ou seja, a coluna DIF será igual a zero. Por esse motivo, são procurados apenas os pares de partições com as quantidades de tuplas mais distantes. Como esses pares de partições não têm quantidades próximas, são os responsáveis pelo desbalanceamento da memória. Por outro lado, se o algoritmo escolher o par de partições com a diferença igual a 0 (zero), a memória permanecerá desbalanceada.

	A	B	SOMA	DIF
1	11	7	18	4
2	10	15	25	5
3	5	2	7	3
4	19	17	36	2
5	5	3	8	2
6	4	2	6	2
	54	46	100	8

Figura 23: Exemplo de partições com valores de diferença iguais

Para facilitar o entendimento do que foi citado no parágrafo anterior, podemos considerar o cenário da Figura 24. Considerando que o parâmetro de balanceamento, *p_balance* é igual a 10 e que a memória está desbalanceada, partindo do pressuposto que a diferença absoluta entre as cardinalidades das relações é 18, a política deve procurar um par de partições que tenha o maior valor na coluna DIF, ou seja, o par de partições que têm as quantidades de suas tuplas menos próximas. No cenário da Figura 24, o par selecionado será o terceiro par de partições, (15,2).

Note que os pares de partições com as mesmas quantidades de tuplas de ambas as relações terão na coluna DIF um valor igual à zero. Esse exemplo pode ser representado pelo quinto par de partições da Figura 24, (11,11). Esse par de partições jamais poderá ser selecionado para ser enviado ao disco, pois ele não afetará o estado da memória.

Por outro lado, quando a memória está balanceada, os pares de partições com quantidades iguais são os mais propícios a serem enviados ao disco, pois esse par detém a menor diferença entre as partições, DIF = 0. Essa situação ocorre devido ao fato do algoritmo tentar manter a memória sempre balanceada.

	A	B	SOMA	DIF
1	4	8	12	4
2	2	13	15	11
3	15	2	17	13
4	4	10	14	6
5	11	11	22	0
6	5	15	20	10
	41	59	100	18

Figura 24: Exemplo de memória desbalanceada e pares de partições iguais

A situação acima descrita pode ser ilustrada na Figura 25. Considerando que o parâmetro de balanceamento é igual a 10 e que a memória está balanceada, partindo do pressuposto que a diferença absoluta entre as cardinalidades das relações é 0, a política deve procurar um par de partições que tenha o menor valor na coluna DIF. Sabendo que os pares de partições com as mesmas quantidades terão na coluna DIF um valor igual a zero, eles são os mais propícios a serem escolhidos. Esse exemplo pode ser representado pelo primeiro par de partições da Figura 25, (7,7), que será o par enviado ao disco.

	A	B	SOMA	DIF
1	7	7	14	0
2	8	9	17	1
3	6	4	10	2
4	10	5	15	5
5	4	7	11	3
6	15	18	33	3
	50	50	100	0

Figura 25: Exemplo de memória balanceada e pares de tuplas iguais

Como resultado da execução do algoritmo podemos ainda ter a memória balanceada e com espaço para novas tuplas como a Figura 26.

No entanto, quando a memória está balanceada e houver mais de um par de partições com o valor mínimo igual na coluna diferença, é preciso usar a coluna da soma para retirar o par de partições que ocupa mais espaço em memória, ou seja, o

par de partições com o maior valor da coluna SOMA. Este par de partições representa o par que tem o maior número de tuplas em memória.

	A	B	SOMA	DIF
1	8	9	17	1
2	6	4	10	2
3	10	5	15	5
4	4	7	11	3
5	15	18	33	3
	43	43	86	0

Figura 26: Resultado da aplicação do Algoritmo MFP

A situação descrita anteriormente pode ser visualizada no cenário da Figura 27, considerando que o parâmetro de balanceamento tenha o valor igual a 10. Como a memória está balanceada, partindo do pressuposto que a diferença absoluta entre as cardinalidades das relações é 8, a política procurar um par de partições que tenha o menor valor na coluna DIF. Os pares de partições selecionados, a partir dessa condição são: o quinto par, (4,7) e o sexto par, (7,4). Como há dois pares de partições, apenas um deles deve ser enviado ao disco. Assim, tendo como objetivo liberação de espaço em memória para inserir novas tuplas, o par de partições que tem a maior quantidade de tuplas em memória, ou seja, o par de partições que tenha o maior valor na coluna SOMA, dentre os pares previamente selecionados é representado pelo sexto par de partições da Figura 27, (9,6).

	A	B	SOMA	DIF
1	10	15	25	5
2	13	9	22	4
3	8	4	12	4
4	10	5	15	5
5	4	7	11	3
6	9	6	15	3
	54	46	100	8

Figura 27: Várias partições com a menor diferença

Outro caso que ainda pode ocorrer quando as partições são enviadas à memória aleatoriamente é quando vários pares de partições têm a mesma diferença entre suas quantidades de tuplas e o mesmo valor máximo na coluna de somatório, tendo como premissa que a memória está balanceada. Neste caso, todos os pares de partições que se encontram nessa situação devem ser enviados para o disco e o espaço em

memória liberado. Imaginando a situação da Figura 28 e considerando que o parâmetro de balanceamento tenha um valor 10, a memória balanceada, pois a diferença absoluta entre as cardinalidades das relações é 8. Com essa informação, a política procura um par de partições que tenha o menor valor na coluna DIF. Os pares de partições escolhidos são o quinto par, (4,7) e o sexto par, (7,4). Como há dois pares de partições selecionados, apenas um deles deve ser enviado ao disco. Como o objetivo é liberação de espaço em memória para inserir novas tuplas, o par de partições que tem a maior quantidade de tuplas em memória, ou seja, o par de partições que tenha o maior valor na coluna SOMA deve ser o escolhido, dentre os pares previamente selecionados. No entanto, a situação da Figura 28 mostra que ambos os pares de partições selecionados a partir da condição de menor valor na coluna DIF têm também o mesmo valor na coluna SOMA. Nesse caso, ambos os pares devem ser enviados ao disco.

	A	B	SOMA	DIF
1	11	7	18	4
2	10	15	25	5
3	10	6	16	4
4	12	7	19	5
5	4	7	11	3
6	7	4	11	3
	54	46	100	8

Figura 28: Exemplo de várias tuplas com o mesmo somatório máximo

5.2. O algoritmo

O algoritmo que implementa a política proposta MFP é executado sempre que ocorrer um *overflow* de memória durante a execução do algoritmo HMJ. Ele é estruturado em duas grandes condições, conforme pode ser ilustrado na Figura 29: (1) quando a memória está balanceada e (2) quando a memória está desbalanceada. As condições podem ser visualizadas nas linhas 1 e 8, respectivamente, da Figura 29. Dentro dessas duas condições existe em cada uma delas uma outra condição, saber se o total de partições da relação A é maior ou menor que o total de partições da relação B. Estas sub-condições podem ser vistas nas linhas 2 e 5 para a condição de memória balanceada e nas linhas 9 e 14 para a condição de memória

desbalanceada, respectivamente, na Figura 29. Essa última condição tem como objetivo indicar o par de partições que deve ser enviado ao disco, conforme o total de partições, ou seja, se o total de partições da relação A é 60 e da relação B é 40, então os pares de partições que devem ser indicados para sair da memória são aqueles que o tamanho das partições da relação A é maior que o tamanho das partições da relação B. Essa condição garante que a memória ficará balanceada após o envio de partições para o disco ou ainda garantindo que a memória se manterá balanceada. Após ser verificado esta condição, o número de partições possíveis a serem enviadas ao disco, já foi reduzido, e então ocorre uma nova pesquisa.

O algoritmo em seguida, busca o par de partições que pode ser enviado ao disco de acordo com a coluna da diferença de partições. Quando a memória está desbalanceada, a procura é pela maior diferença e quando a memória está balanceada a procura é pela menor diferença. Essa condição da diferença pode ser visto nas linhas 4 e 7 quando a memória está balanceada e nas linhas 11 e 16 quando a memória está desbalanceada, respectivamente, na Figura 29.

```
1 -  Se   |A - B| > 10                  // Memória desbalanceada
2 -       Se   A >= B                  // Verifica se o tamanho das tuplas de A é maior que o tamanho das tuplas de B
3 -            Procura um par de tuplas com Ak > Bk
4 -                 Procurar dentre as tuplas achadas acima a que tiver a maior diferença
5 -       senão
6 -            Procura um par de tuplas com Ak < Bk
7 -                 Procurar dentre as tuplas achadas acima a que tiver a maior diferença
8 -  senão                             // Memória balanceada
9 -       Se   A >= B
10 -           Procura um par de tuplas com Ak >= Bk
11 -                Procura dentre as tuplas achadas acima a que tiver a menor diferença
12 -                Se  vários pares têm a mesma diferença
13 -                    Procura um par com maior somatório
14 -       senão
15 -           Procura um par de tuplas com Ak < Bk
16 -                Procura dentre as tuplas achadas acima a que tiver a menor diferença
17 -                Se  vários pares têm a mesma diferença
18 -                    Procura um par com maior somatório
19 -           Se há vários pares com o mesmo somatório
20 -                Envia todos esses pares para o disco
```

Figura 29: Algoritmo *Mobile Flushing Policy*

Há ainda uma pequena condição quando a memória está balanceada: se forem encontrados vários pares de partições com o mesmo valor da diferença. Neste caso, o algoritmo usa a coluna do somatório da tabela resumo, para retirar o par de partições com maior somatório e liberar mais espaço em memória. Essa última condição pode ser vista nas linhas 12 e 17, respectivamente, na Figura 29.

Caso ainda haja mais de um par de partições com o mesmo valor de somatório, o algoritmo envia para o disco todos os pares de tuplas. Essa última operação do algoritmo pode ser vista nas linhas 19 e 20 da Figura 29.

5.3. Discussão comparativa

A política *Mobile Flushing Policy*(MFP) propõe uma nova tabela resumo com base na tabela resumo mantida em memória usada na política *Adaptive Flushing Policy*(AFP). O algoritmo proposto cria uma nova coluna com a diferença entre as partições de memória, de forma a desconsiderar o parâmetro de tamanho mínimo de partições. No entanto, a nova proposta ainda mantém o parâmetro de balanceamento da memória. Na política proposta denominamos este parâmetro de *p_balance*.

Os objetivos da nova coluna na tabela resumo são:

- Estender a política AFP garantindo que o par de partições selecionado para ser enviado ao disco não deixará a memória desbalanceada. A garantia do algoritmo de não deixar a memória desbalanceada é devido à escolha certa das partições enviadas ao disco, conforme as condições descritas no algoritmo da Figura 29, da seção anterior.
- Garantir que haverá pelo menos um par de partições que será enviado ao disco. Após todas as condições descritas na Figura 29, da seção anterior, vemos que em último caso são enviados vários pares de partições ao disco, mas sempre algum par de partições é enviado.

Nas sub-seções 5.3.1 e 5.3.2 faremos uma comparação das duas políticas nas situações em que a memória está balanceada e não balanceada e veremos que a nova política proposta seleciona melhor qual par de partições deve ser enviado ao disco, mantendo sempre como premissa a eficiente liberação de espaço em memória.

5.3.1. Memória Desbalanceada

Para uma melhor compreensão, usaremos como exemplo a Figura 30. Quando estivermos usando o algoritmo AFP, vamos considerar os parâmetros $a = 5$ e $b = 10\%$. Quando estivermos usando o algoritmo MFP, vamos considerar o parâmetro

de balanceamento, *p_balance* = 10. Vamos considerar também que temos uma memória com tamanho de 100Mb. Essa memória se divide em partições das relações A e B, onde A tem cardinalidade 41 e B tem cardinalidade 59.

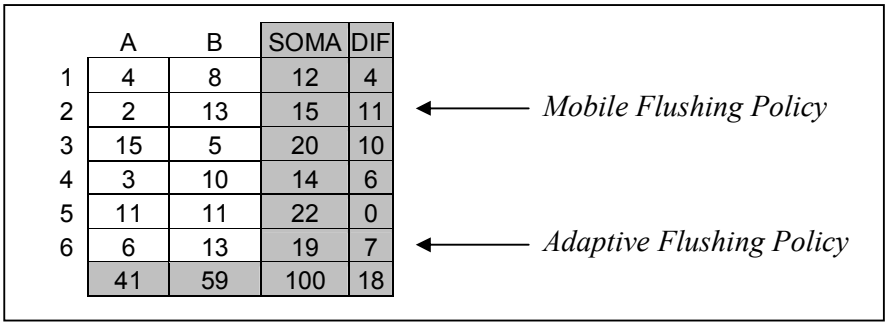


Figura 30: Comparativo entre as políticas com memória desbalanceada

Usando o algoritmo AFP, a memória é considerada desbalanceada, pois o módulo da diferença entre as cardinalidades está acima do parâmetro estipulado, ou seja, |41 – 59| > 10%. Para selecionar o par de partições que deve ser enviado ao disco, analisa-se duas condições: Na primeira condição, deve-se achar um par de partições com as quantidades de tuplas superior ao parâmetro *a*. Satisfazendo essa condição, temos como resultado os pares (11,11) e (6,13). A segunda condição que deve ser satisfeita é a da quantidade de tuplas nas partições na relação B estarem acima da quantidade de tuplas das partições na relação A. Satisfazendo essa condição, temos como resultado, partindo do sub-conjunto de pares resultantes da condição anterior. O par (6,13) será o escolhido para ser enviado ao disco.

	A	B	SOMA
1	4	8	12
2	2	13	15
3	15	5	17
4	3	10	14
5	11	11	22
	35	47	82

Figura 31: Tabela resumo após o algoritmo AFP, com memória desbalanceada

Como conseqüência, do envio do sexto par para o disco, a memória permanece desbalanceada, como pode ser visto na Figura 31, pois o módulo da diferença entre as cardinalidades ainda está acima do parâmetro estipulado, ou seja, |35 – 47| > 10%.

Usando o algoritmo MFP, a memória também é considerada desbalanceada, pois o módulo da diferença entre as cardinalidades está acima do parâmetro estipulado, ou seja, |41 – 59| > 10%. Para selecionar o par de partições que deve ser enviado ao disco, analisam-se duas condições: Na primeira condição, deve-se achar um par de partições com o maior valor na coluna da Diferença, ou seja, com a maior diferença entre as tuplas do par de partições correspondentes. Satisfazendo essa condição, temos como resultado apenas o par (2,13). A segunda condição que deve ser satisfeita é a da quantidade de tuplas nas partições na relação B estarem acima da quantidade de tuplas das partições na relação A. Como resultado, temos que o par (2,13), previamente selecionado atende a condição. Logo o par de partições escolhido para ser enviado ao disco é o segundo par, (2,13), pois este par tem a quantidade de partições de B maior que a quantidade de partições de A e esse par tem a maior diferença. Como conseqüência, a memória fica balanceada, como pode ser visto na Figura 32.

	A	B	SOMA	DIF
1	4	8	12	4
2	15	5	20	10
3	3	10	13	7
4	11	11	22	0
5	6	13	17	7
	39	47	86	8

Figura 32: Tabela resumo após o algoritmo MFP, com memória desbalanceada

5.3.2. Memória Balanceada

Para uma melhor compreensão, usaremos como exemplo a Figura 33. Quando estivermos usando o algoritmo AFP, vamos considerar os parâmetros a = 5 e b = 10%. Quando estivermos usando o algoritmo MFP, vamos considerar o parâmetro de balanceamento, *p_balance* = 10%. Vamos considerar também que temos uma memória com tamanho de 100Mb. Essa memória se divide em partições das relações A e B, onde A tem cardinalidade 46 e B tem cardinalidade 54.

Usando o algoritmo AFP, a memória é considerada balanceada, pois o módulo da diferença entre as cardinalidades está abaixo do parâmetro estipulado, ou seja, |46 – 54| < 10%. Para selecionar o par de partições que deve ser enviado ao disco,

analisa-se duas condições: Na primeira condição, deve-se achar um par de partições com as quantidades de tuplas superior ao parâmetro *a*. Satisfazendo essa condição, temos como resultado os pares (13,8), (11,15), (3,7) e (12,9). A segunda condição que deve ser satisfeita é a da quantidade de tuplas nas partições na relação B estarem acima da quantidade de tuplas das partições na relação A. Satisfazendo essa condição, temos como resultado, partindo do sub-conjunto de pares resultantes da condição anterior que os pares de partições escolhidos para serem enviados ao disco são os pares (11,15) e (3,7).

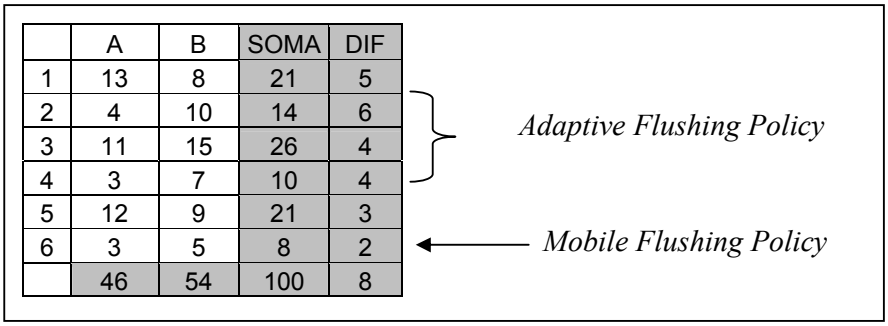


Figura 33: Comparativo entre as políticas com memória balanceada

Como conseqüência, apesar da memória permanecer balanceada, o tempo para envio dos dois pares de partições ao disco prejudica o desempenho da política, pois a mesma precisa enviar dois pares de partições invés de apenas um par. A tabela resultado está apresentada na Figura 34.

	A	B	SOMA	DIF
1	13	8	21	5
2	4	10	14	6
3	12	9	21	3
4	3	5	8	2
	32	32	64	8

Figura 34: Tabela resumo após o algoritmo AFP, com memória balanceada

Usando o algoritmo MFP, a memória também é considerada balanceada, pois módulo da diferença entre as cardinalidades está acima do parâmetro estipulado, ou seja, $|46 - 54| < 10\%$. Para selecionar o par de partições que deve ser enviado ao disco, analisa-se duas condições: Na primeira condição, deve-se achar um par de partições com o menor valor na coluna da Diferença, ou seja, com a menor diferença entre as tuplas do par de partições correspondentes. Satisfazendo essa condição, o

par (3,5) atende essa condição. A segunda condição que deve ser satisfeita é a da quantidade de tuplas nas partições na relação B estarem acima da quantidade de tuplas das partições na relação A. Satisfazendo essa condição, temos o par (3,5). Logo o par de partições escolhido para ser enviado ao disco é o segundo par, (3,5), pois este par tem a quantidade de partições de B maior que a quantidade de partições de A e esse par tem a menor diferença. Como conseqüência, a memória permanece balanceada, como pode ser visto na Figura 35.

Comparando as políticas MFP e AFP, na tabela resumo da política MFP há uma coluna a mais, ocupando mais espaço em memória. No entanto, a política MFP seleciona com mais eficiência que a política AFP, os pares de partições que devem ir para o disco em caso de *overflow* de memória, além de garantir que sempre haverá um par de partições dentro das especificações do algoritmo. Outra vantagem da política MFP sobre a política AFP é que a primeira usa apenas um parâmetro: o parâmetro de balanceamento da memória, denominado agora de *p_balance*, enquanto o AFP usa dois parâmetros.

	A	B	SOMA	DIF
1	13	8	21	5
2	4	10	14	6
3	11	15	26	4
4	3	7	10	4
5	12	9	21	3
	43	49	92	6

Figura 35: Tabela resumo após o algoritmo MFP, com memória balanceada

A Tabela 4 apresenta as principais características de cada política mais detalhadamente.

Tabela 4: Comparativo entre as políticas AFP e MFP

Características	AFP	MFP
Número de parâmetros	2	1
Número de colunas na tabela resumo	3	4
Memória balanceada em todos os casos	Não	Sim
Seleciona um par de partições para ser enviado ao disco	Não	Não
Ocorre alteração de parâmetro para selecionar pelo menos um par de partições	Sim	Não
Seleciona o melhor par de partições para liberar mais espaço em memória	Não	Sim

A primeira característica da Tabela 4 descreve a quantidade de parâmetros utilizada nas políticas. A segunda característica descreve o número de colunas na tabela resumo. Note que o MFP trocou um parâmetro da política AFP por uma coluna, diminuindo com isso, a intervenção humana na configuração dos parâmetros.

A terceira característica descreve qual dos algoritmos deixa a memória sempre balanceada. Note que a política AFP nem sempre deixa a memória balanceada, como foi mostrado nas Figura 30, Figura 31 e Figura 32.

A quarta e a quinta características descrevem que se a política usada for a AFP e se não houver par de partições que satisfaça aos parâmetros pré-definidos, deve-se alterar a configuração dos mesmos para enviar um par de partições, ou enviar todas as partições para o disco, acarretando uma paralisação muito grande no processamento de novas tuplas. Em contrapartida, usando a política MPF, essa situação não ocorre, ou seja, não há necessidade de alteração de parâmetros, pois pelo menos um par de partições será escolhido e enviado ao disco.

A última característica descreve que apenas a política MFP seleciona o melhor par de partições para ser enviado ao disco, enquanto a política AFP seleciona qualquer par de partições que esteja dentro dos limites dos parâmetros. O melhor par de partições é aquele que deixa a memória balanceada e ainda libera o maior espaço possível para que a ocorrência de um novo *overflow* seja mais tardia.

6. Resultados Experimentais

6.1. Introdução

Nos capítulos anteriores foram apresentados os principais algoritmos de junção e algumas políticas de liberação de espaço de memória utilizadas em ambientes móveis. Esses algoritmos têm características comuns que permitem ser utilizados no contexto de consultas móveis tais como: adaptação das fontes de dados a diferentes processamentos, alteração do plano de execução em tempo de execução, e fornecimento de resultados parciais. Dentre os algoritmos de junção apresentados, apenas o *Hash-Merge Join* (HMJ) utiliza uma política de adaptativa de liberação de tuplas da memória. Essas políticas de liberação referem-se a regras adotadas para liberação do espaço de memória para permitir a produção de junções de tuplas provenientes de fontes de dados distintas, evitando-se problemas de *overflow*.

O desenvolvimento da ferramenta MFP partiu da necessidade de validar e avaliar a eficiência do uso da política de liberação de tuplas da memória MFP, apresentada no Capítulo 4, como política a ser utilizada no algoritmo de junção HMJ em substituição a política AFP. A ferramenta permite avaliar tanto a estrutura da tabela resumo proposta como a redução do número de parâmetros utilizada para controle de liberação de tuplas da memória.

A política foi implementada em Java, escolhida por possuir recursos poderosos que facilitaram o trabalho de desenvolvimento do protótipo, como a fácil manipulação de *threads* (linhas de execução), a presença de um mecanismo embutido de sincronização destas linhas de execução e a implementação de uma nova função geradora de *hash codes* chamada *GeraCódigoHash*.

Este capítulo possui a seguinte estrutura: a seção 2 descreve o funcionamento do algoritmo. Na seção 3 apresentamos a arquitetura representativa do algoritmo. As classes e estruturas de dados utilizadas neste trabalho, são abordados em detalhes na seção 4, bem como a implementação de todas as estruturas de dados necessárias para a execução do algoritmo. Os tipos de simulação executados, a configuração do ambiente na qual as simulações foram executadas e os resultados das simulações são descritos na seção 5.

6.2. O algoritmo

O algoritmo *Mobile Flushing Policy* (MFP), foi desenvolvido utilizando-se a tecnologia Java e tem como objetivo fornecer uma implementação do algoritmo de junção *Hash-Merge* Join (HMJ) adaptada para o uso da política de liberação MFP a ser utilizado em ambientes móveis. O algoritmo MFP foi implementado visando abstrair-se do tipo de ambiente móvel, bem como o Sistema Gerenciador de Banco de Dados utilizado para esse ambiente.

A implementação do MFP apresenta uma forma simplificada de representação das tuplas em memória como também de seu armazenamento em disco. Para isso considerou-se apenas 3 informações para definição da tupla (código da chave primária, nome do atributo e valor do atributo). A tupla quando enviada para disco é serializada e então armazenada na forma binária ao invés de texto. Dessa forma, permite-se que toda tupla seja facilmente recuperada do disco para então ser realizado o processo de junção.

6.3. A arquitetura

A estrutura básica da arquitetura que representa o algoritmo MFP é apresentada na Figura 36. Essa arquitetura é composta por três níveis: Apresentação, Processamento e Classes e estruturas. Cada nível é composto por um ou mais componentes. Na próxima seção será descrito cada nível da arquitetura, bem como os possíveis componentes. Dentre os componentes da arquitetura podem se destacar os componentes referentes à implementação do algoritmo HMJ, ao processo de junção de tuplas, como também os componentes referentes à política de liberação de tuplas da memória e componentes relacionados ao armazenamento dos dados em disco.

No nível da Apresentação encontra-se a ferramenta MFP que apresenta uma interface gráfica ao usuário para permitir iniciar o processamento de consultas e visualizar os dados provenientes das junções realizadas pelo HMJ. Como resultado a interface apresenta os dados através de uma visão relacional.

No nível do Processamento a arquitetura representa a lógica do algoritmo HMJ e da política MFP, é composta por um conjunto de classes desenvolvidas para atender desde a inserção dos dados em memória até seu envio para disco.

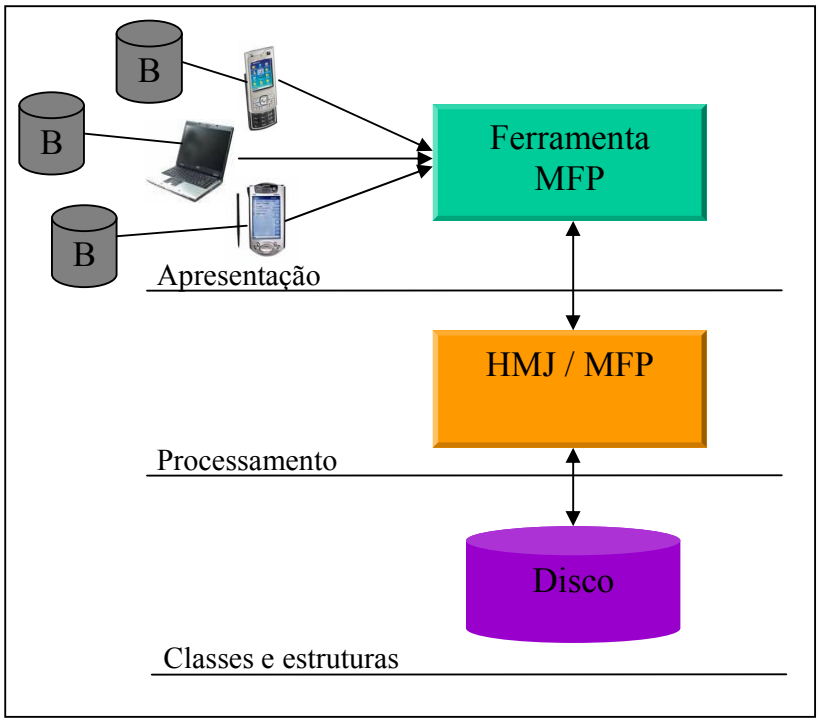


Figura 36: Visão Geral da Arquitetura MFP

Por fim, no nível das Classes e estruturas há um conjunto de classes utilizadas para obter dados de uma fonte de dados (SGBD) bem como o armazenamento e recuperação de dados utilizados pela HMJ em um sistema de arquivos binários.

Assim, o algoritmo MFP apresenta funcionalidades que permitem oferecem um melhor gerenciamento dos dados armazenados em memória e disco em um ambiente móvel para realização de junções provenientes de um processamento de consulta.

6.3.1. Apresentação

Na camada de apresentação tem-se a ferramenta MFP que disponibiliza uma *interface* gráfica ao usuário para permitir iniciar o processamento de consultas e visualizar os dados provenientes das junções realizadas pelo HMJ, conforme pode ser visto na Figura 37.

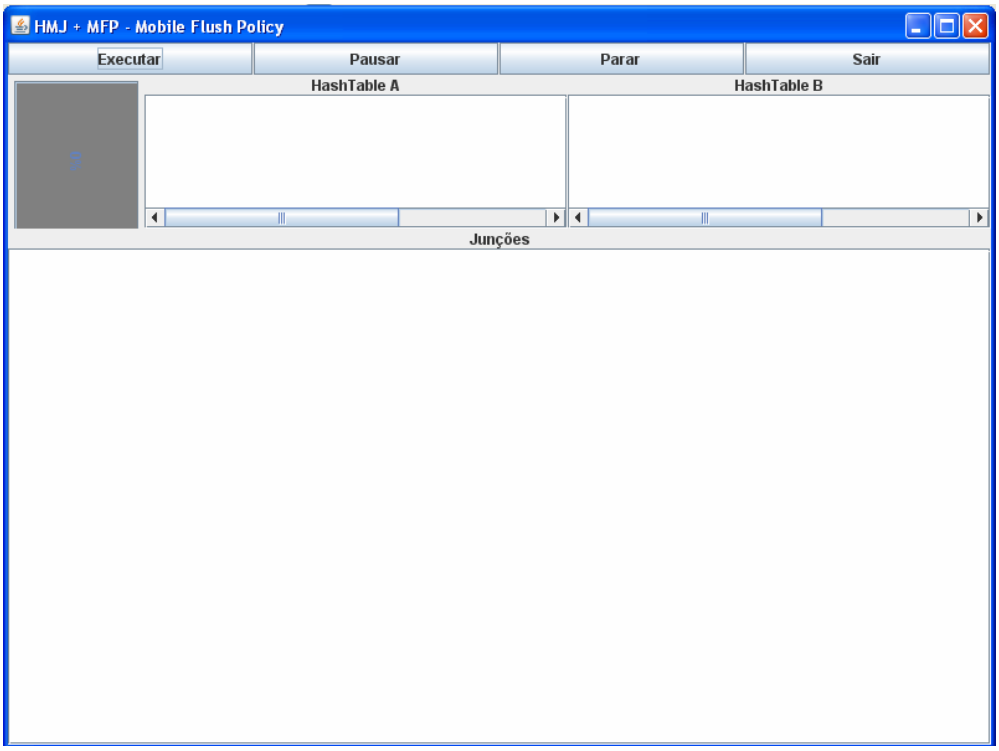


Figura 37: Interface Gráfica

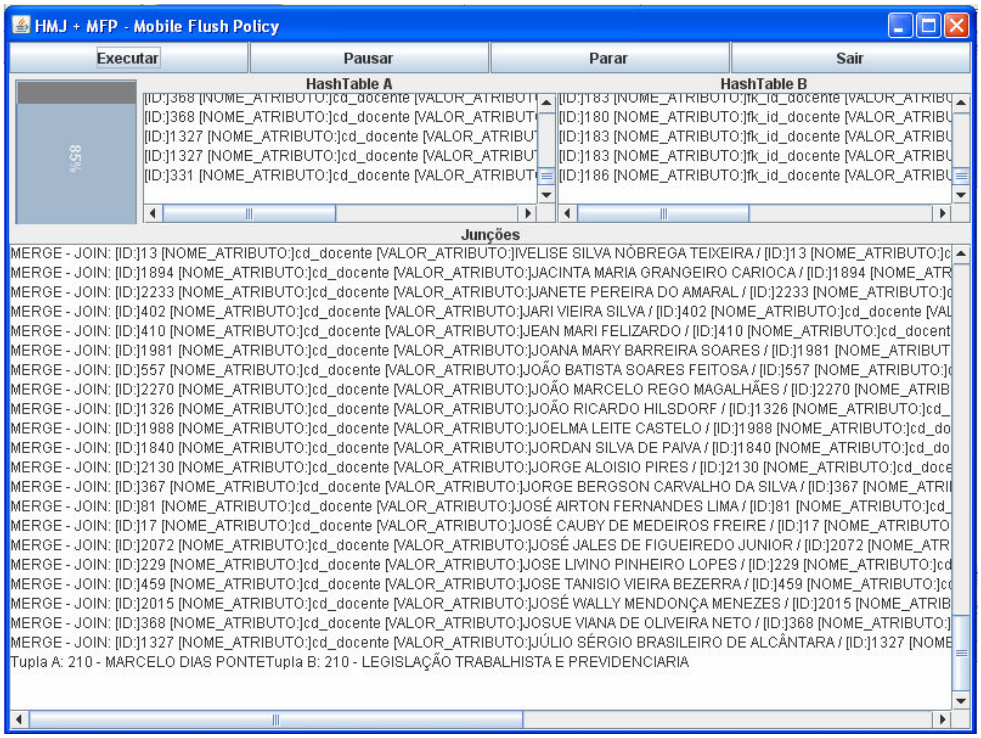


Figura 38: Interface Gráfica após a execução das junções

Conforme pode ser visto na Figura 38, após a execução do algoritmo HMJ, a interface gráfica apresenta as tuplas de cada tabela *hash*, apresenta a quantidade de uso da memória no canto superior esquerdo e na maior parte da tela o resultado das junções. Note também que a interface mostra em que fase ocorreu a junção, se na fase de *Hash* ou na fase de *Merge*.

Existe um *log* que nesta versão da política de liberação MFP só funciona em tempo de projeto, através da ferramenta de desenvolvimento Eclipse, no entanto está previsto como trabalhos futuros. A parte inicial do *log* da execução do algoritmo HMJ pode ser visto na Figura 39.

```
Tam.Tabela Hash: 3
Tam.Tabela Hash: 3
Consumo Memória: 0
Consumo Memória: 68
Consumo Memória: 68
 *** Tabela Resumo ***
TabResumo[0] = 44, 0
TabResumo[1] = 0, 24
TabResumo[2] = 0, 0
 *** ************* ***
*** BALANCEADA - Maior Soma Menor Diferença ***
Maior Soma:[0]:44
POLITICA DE LIBERAÇÃO - 0
ID=1806, ATRIBUTO=cd_docente,VALOR=ABIMAEL CLEMENTINO FERREIRA DE
CARVALHO NETO
Consumo Memória: 71
 *** Tabela Resumo ***
TabResumo[0] = 0, 0
TabResumo[1] = 26, 45
TabResumo[2] = 0, 0
 *** ************* ***
*** BALANCEADA - Maior Soma Menor Diferença ***
Maior Soma:[1]:71
POLITICA DE LIBERAÇÃO - 1
ID=2053, ATRIBUTO=cd_docente,VALOR=ADERSON DOS SANTOS SAMPAIO
ID=1, ATRIBUTO=fk_id_docente,VALOR=MONOGRAFIA JURIDICA I
Consumo Memória: 89
 *** Tabela Resumo ***
TabResumo[0] = 26, 0
TabResumo[1] = 0, 0
TabResumo[2] = 0, 39
 *** ************* ***
*** BALANCEADA - Maior Soma Menor Diferença ***
Maior Soma:[2]:39
POLITICA DE LIBERAÇÃO - 2
ID=1, ATRIBUTO=fk_id_docente,VALOR=MONOGRAFIA JURIDICA I
ID=1, ATRIBUTO=fk_id_docente,VALOR=MONOGRAFIA JURIDICA I
ID=8, ATRIBUTO=fk_id_docente,VALOR=EVOLUÇÃO DO PENSAMENTO DA
ADMINISTRAÇÃO
Consumo Memória: 50
 *** Tabela Resumo ***
TabResumo[0] = 26, 0
TabResumo[1] = 0, 0
TabResumo[2] = 0, 0
```

```
 *** ************* ***
*** DESBALANCEADA - Maior Diferença ***
Maior Diferença:[0]:26
POLITICA DE LIBERAÇÃO - 0
ID=27, ATRIBUTO=cd_docente,VALOR=ADHEMAR NUNES FREIRE FILHO
Consumo Memória: 79
 *** Tabela Resumo ***
TabResumo[0] = 23, 0
TabResumo[1] = 0, 0
TabResumo[2] = 0, 32
 *** ************* ***
```

Figura 39: Log de execução do algoritmo HMJ

6.3.2. Processamento

No nível do Processamento, a ferramenta MFP possui um conjunto de classes necessárias à implementação do algoritmo HMJ e da política MFP de liberação de tuplas da memória, conforme a arquitetura MFP. Nessa camada, as classes de negócio encontram-se organizadas em pacotes. Dentre os pacotes ressaltam-se o pacote modelo, o *hmj* e o pacote *Política*.

No diagrama de classes ilustrado na Figura 40 encontram-se as classes fundamentais para implementação do HMJ, como a classe *HashPhase* e *MergePhase*. A classe MFP representa a implementação da política de liberação MFP que é associada a classe *TabelaResumo*. A unidade de representação dos dados é a classe Tupla, utilizada pelas demais classes para realização do processo de junção do algoritmo HMJ.

A classe *HashPhase* tem como objetivo implementar as funcionalidades da fase *Hash* do algoritmo HMJ. Nessa classe determina-se o consumo de memória, a capacidade máxima da memória, o número total de elementos na memória, como também insere tuplas na memória. Em caso de *overflow* de memória, esta classe invoca a política de liberação de memória.

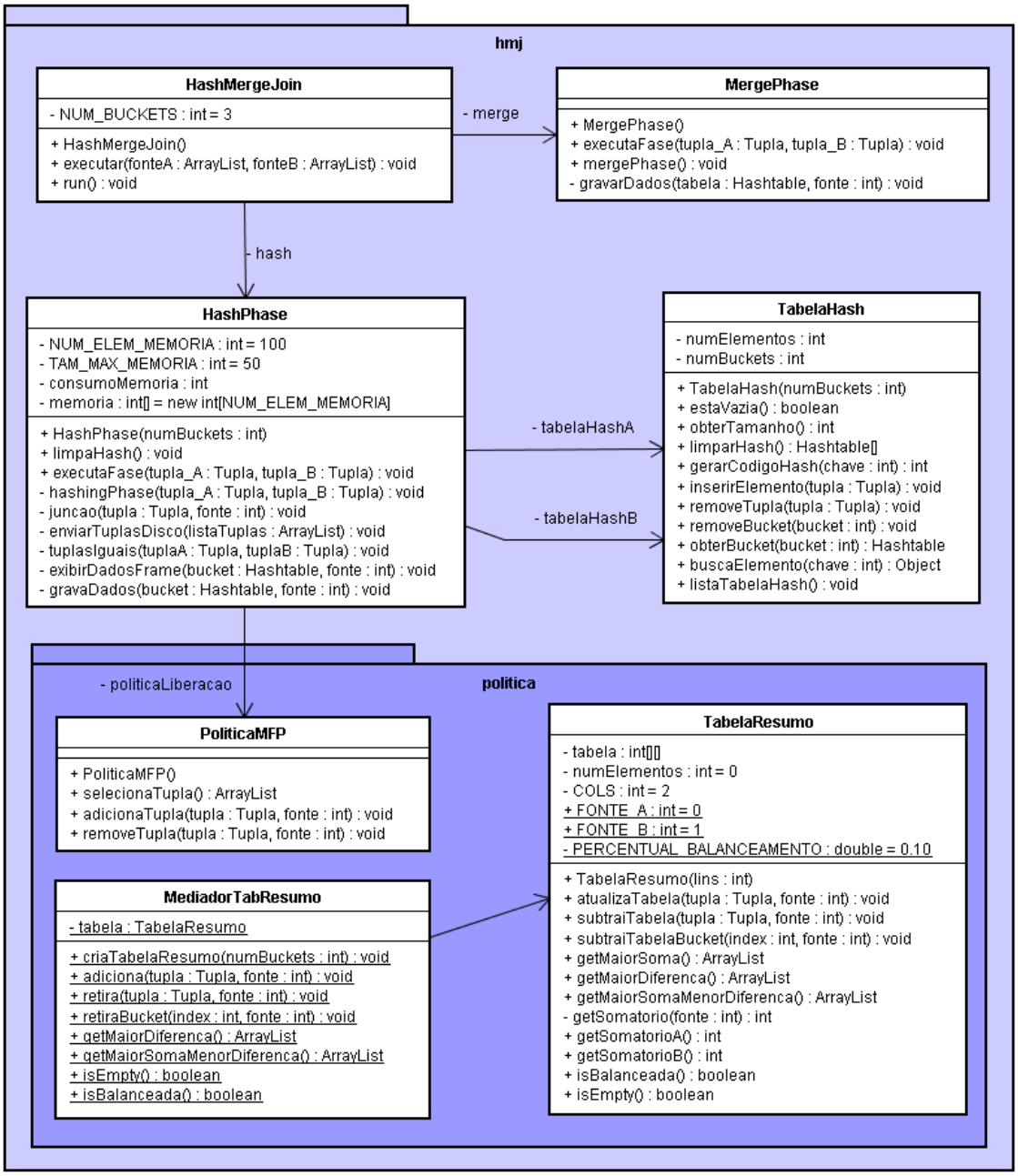


Figura 40: Diagrama de classes do pacote HMJ

A classe *MergePhase* implementa as funcionalidades da fase da *Merge* do algoritmo HMJ. Essa classe determina onde as tuplas serão armazendas, como também sua recuperação e ordenação antes de realizar um novo armazenamento em disco. Esta classe invoca a classe *JfileStream*: classe responsável em gerenciar o processo de criação, armazenamento e recuperação das tuplas em disco.

A classe PoliticaMFP refere-se a definição da política de liberação do *Mobile Flushing Policy* (MFP), que realiza busca em memória da tupla ideal, conforme as regras apresentadas no capítulo 4, para liberação de otimizada de memória e permitir a continuidade do processamento de junções de tuplas. Essa classe está associada a classe *TabelaResumo* que possui o objetivo de identificar quais tuplas atendem as regras da política MFP. A *TabelaResumo* é composta pelos atributos *códigoHash*., tamanho da fonte A, tamanho da fonte B, e as operações que representam o somatório dessas duas fontes, bem como sua diferença.

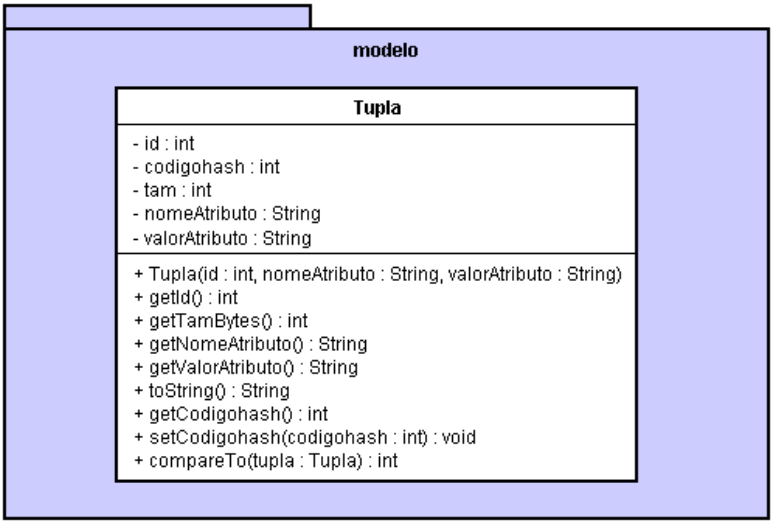


Figura 41: Diagrama de Pacote modelo e classe Tupla

O pacote política é um sub-pacote do pacote hmj, conforme apresentado na Figura 40. Nesse pacote encontram-se as classes que implementam a política MFP de liberação de memória. A classe PolíticaMFP refere-se a implementação da política de liberação de memória. Esta classe realiza a verificação do nível de balanceamento da tabela hash onde as tuplas estão armazenadas em memória como também retorna o bucket a ser liberado da memória e enviada para disco. Ela comunica-se com a classe TabelaResumo através da classe MediadorTabResumo, que é composta de métodos estáticos para facilitar a comunicação da classe PoliticaMFP com a classe TabelaResumo e demais classes que atualizam a classe TabelaHash e que necessitam atualizar a tabela resumo da política MFP. A classe TabelaResumo representa a tabela resumo da política MFP necessária para identificar o nível de balanceamento das tabelas hash das respectivas fontes de dados, bem como implementar as regras de seleção de

buckets a serem removidos de memória, liberando espaço em memória conforme regras da política MFP. Esta classe utiliza uma matriz de inteiros para representar a tabela resumo, onde as colunas representam a quantidade de memória consumida por cada tabela hash de dados (fontes A e B), e as linhas cada *bucket* da tabela hash.

O pacote modelo, conforme apresentado na Figura 41 possui somente a classe Tupla. Essa classe é utilizada como unidade de dados representando cada registro da tabela consultada para realização da junção.

O pacote política é um sub-pacote do pacote *hmj*, conforme apresentado na Figura 40. Nesse pacote encontram-se as classes que implementam a política MFP de liberação de memória. A classe *PolíticaMFP* refere-se a implementação da política de liberação de memória. Esta classe realiza a verificação do nível de balanceamento da tabela *hash* onde as tuplas estão armazenadas em memória como também retorna a partição a ser liberada da memória e enviada para disco. Ela comunica-se com a classe *TabelaResumo* através da classe *MediadorTabResumo*, que é composta de métodos estáticos para facilitar a comunicação da classe *PoliticaMFP* com a classe TabelaResumo e demais classes que atualizam a classe *TabelaHash* e que necessitam atualizar a tabela resumo da política MFP. A classe *TabelaResumo* representa a tabela resumo da política MFP necessária para identificar o nível de balanceamento das tabelas *hash* das respectivas fontes de dados, bem como implementar as regras de seleção de partições a serem removidas de memória, liberando espaço na mesma conforme as regras da política MFP. Esta classe utiliza uma matriz de inteiros para representar a tabela resumo, onde as colunas representam a quantidade de memória consumida por cada tabela *hash* de dados (fontes A e B), e as linhas cada partição da tabela *hash*.

O pacote modelo, conforme apresentado na Figura 41 possui somente a classe Tupla. Essa classe é utilizada como unidade de dados representando cada registro da tabela consultada para realização da junção.

Um pseudo-código do algoritmo MFP, com as principais classes é apresentado nos Apêndices A e B.

6.3.3. Classes e estruturas

Na camada de acesso a dados, encontram-se um conjunto de classes que permitem a conexão a um SGBD e a recuperação de dados deste. Isso se tornou uma necessidade

diante do fato de testar e avaliar o desempenho a aplicação da política MFP no algoritmo de HMJ.

O pacote util, conforme apresentado na Figura 42 é composto pela classe *JFileManager* responsável pela escrita e leitura de tuplas em disco. Encontra-se neste pacote o sub-pacote BD (Banco de Dados) composta pelas classes *TuplaDAO* e *FabricaConexao* que implementam as regras de recuperação de dados das respectivas fontes de dados.

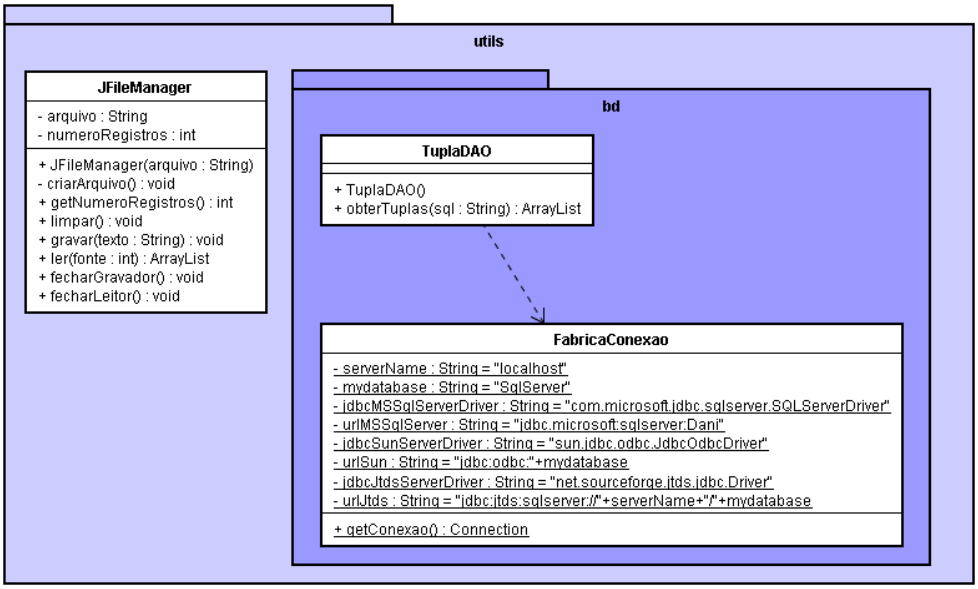


Figura 42: Diagrama de classes do pacote UTIL

A classe *TuplaDAO* implementa o padrão de projeto DAO e possui as operação para chamada de uma conexão ao banco de dados, bem como um conjunto de instrução para realização da solicitação de execução de uma consulta a uma fonte de dados e o envio desses dados em forma de uma lista de tuplas para as classes que iniciam o processo de junção de tuplas pelo algoritmo HMJ.

A classe FabricaConexão é responsável em realizar a conexão a uma fonte de dados remota, e fornecer a conexão a classe TuplaDAO para obtenção das tuplas. Esta classe é composta por membros estáticos que representam uma string de conexão a fonte de dados, em particular implementado para o SGBDR (Sistema Gerenciador de Banco de Dados Relacional) Microsoft SqlServer 2000, e a operação de obtenção da conexão a fonte de dados.

6.4. Simulações

As simulações foram executadas em uma estação Turion 64 com 1 GB de memória RAM, 80GB de HD e com o sistema operacional Windows XP. As duas tabelas A e B utilizadas nas simulações da execução da política MFP, foram armazenadas no sistema gerenciador de banco de dados Microsoft SQL Server 8.0, instalado em um servidor com a seguinte configuração: Turion 64 com 1 GB de memória RAM com 1 processador. As tabelas utilizadas como fonte de dados para a execução do operador possuíam 6,40 MB e 7,35 MB de tamanho, respectivamente. Tais tabelas armazenavam tuplas de tamanhos diferentes, com o objetivo de simular o mundo real. Cada tupla da tabela de professores contém dois atributos: Código e nome do professor. Cada tupla da tabela de disciplinas contém 4 atributos: código, nome, créditos e turno. Os parâmetros para as políticas de liberação usados foram respectivamente: quantidade mínima de tuplas em cada partição igual a 10 e o parâmetro de balanceamento de memória igual a 1.

O primeiro tipo de simulação consistiu na execução do *Hash-Merge Join* utilizando a política AFP em comparação com a política AFP para analisar a liberação de partições para disco contra a quantidade de E/S necessárias. Este tipo de simulação tem como objetivo mostrar que em ambientes com limitação de memória a execução da liberação de partições para o disco.

Note que na política MFP as partições são liberadas em menos tempo, ou seja, com menos necessidade de E/S de disco. Isso torna o custo do desempenho da política MFP menor, conforme a Figura 43.

O segundo tipo de simulação consistiu no comparativo das duas políticas com relação ao desempenho das mesmas durante a execução do Hash-MergeJoin.É interessante executar esta simulação para mostrar que o algoritmo Hash-Merge Join executa a junção mais rapidamente com a política MFP que com a política AFP. Conforme a Figura 44 o desempenho da política AFP é ligeiramente melhor que o da política MFP. Isso indica que o algoritmo HMJ praticamente não altera seu desempenho mesmo alterando a política de liberação da AFP para a MFP.

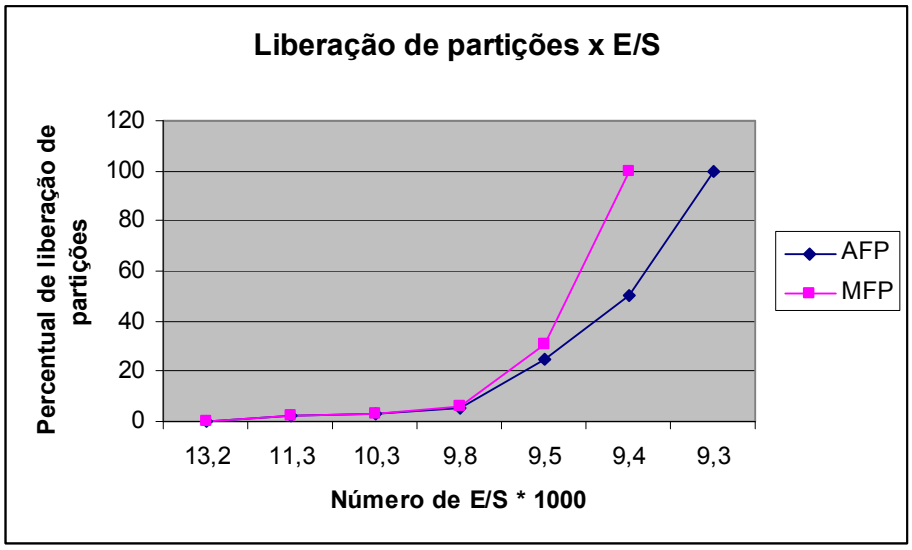


Figura 43: Comparativo de impacto sobre E/S

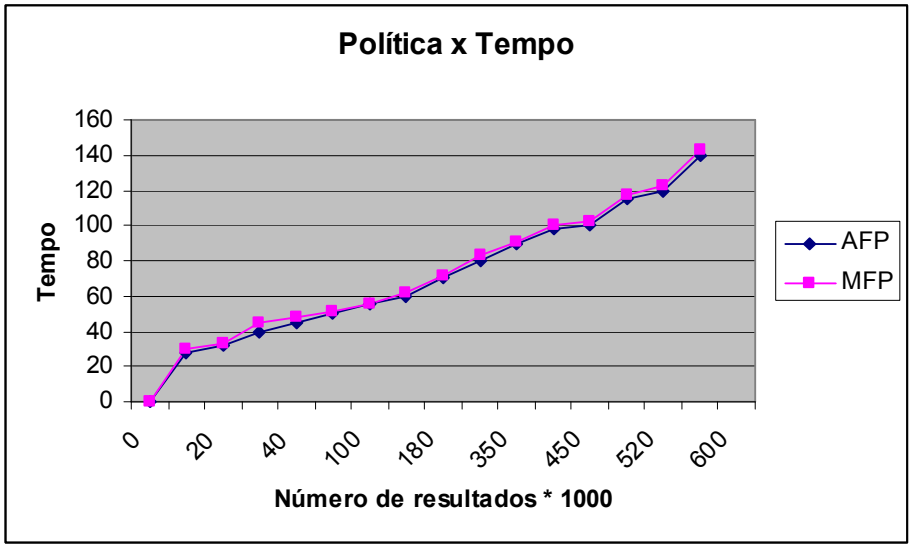


Figura 44: Comparativo das políticas x tempo

6.5. Estimativa de custo do algoritmo

A estimativa de custo do algoritmo MFP é de $EC = O(2n) + O(p) + O(q)$, onde n é a quantidade de linhas da tabela resumo, p e q são as cardinalidades das fontes de dados. Isso ocorre por que as tabelas precisam ser lidas e suas tuplas comparadas apenas uma vez. Já a tabela resumo precisa ser lida duas vezes, uma para a leitura da coluna de somatório e outra para a coluna de diferença.

Analisando a Figura 45 vemos que na tabela resumo três linhas detém o mesmo valor da coluna DIFERENÇA. Neste caso é necessário fazer uma re-leitura da tabela resumo para analisar a coluna SOMA e então descobrir qual dos três pares deverá ser enviado ao disco. Neste caso, o par que será enviado será o par com maior valor na coluna SOMA. O par enviado será o par (13,10).

	A	B	SOMA	DIF
1	10	15	25	5
2	13	10	23	3
3	8	4	12	4
4	10	5	15	5
5	4	7	11	3
6	9	6	15	3
	54	46	100	8

Figura 45: Tabela resumo exemplo

7. Conclusão e Trabalhos Futuros

Esse trabalho contextualiza a necessidade de haver no processamento de consultas em bancos de dados móveis um algoritmo que execute junções com as características inerentes a bancos de dados móveis, descritas abaixo:

- Minimizar o tempo de resposta;
- Adaptação das fontes de dados a diferentes processamentos;
- Contínua otimização de consultas;
- Continuidade no processamento de tuplas;
- Garantia de não redundância e exatidão no resultado final;
- Fornecimento de resultados parciais;
- Balanceamento de memória.

A proposta da dissertação é uma política de liberação de tuplas da memória *Mobile Flushing Policy*(MFP), usada no algoritmo de junção *Hash-Merge Join*. Esta política substitui a política *Adaptive Flushing Policy* (AFP). Ambas as políticas tem o mesmo objetivo: enviar tuplas da memória para o disco quando ocorrer *overflow* de memória. No entanto, a nova política otimiza a política AFP usada no algoritmo *Hash-Merge Join*, evitando algumas falhas, do HMJ, descritas abaixo:

- Caso nenhum par de partições satisfaça os parâmetros a e b, o algoritmo deverá ser novamente parametrizado para garantir a escolha de algum par de partições e liberar espaço em memória. Por exemplo, na Figura 18, os exemplos 1 e 2, ilustram que se os valores dos parâmetros forem a = b = 10, nenhuma partição será escolhida.
- Pode ocorrer de o par de partições escolhido para ser enviado ao disco não deixe a memória balanceada. Por exemplo, na Figura 18, o exemplo 3 mostra que o par selecionado deve ser (5,13). No entanto, a memória ainda permanece desbalanceada, caso os valores dos parâmetros forem a = b = 10.

A política MFP tem como objetivo a liberação de tuplas da memória de forma mais eficiente que a AFP, ou seja, a MFP mantém a memória sempre balanceada quando houver a necessidade de enviar algumas partições para o disco e dar suporte às fases de *Hashing* e *Merging* do algoritmo *Hash-Merge Join*, resolvendo todos os problemas encontrados na política AFP.

Para a política funcionar conforme seu objetivo, ela mantém em memória uma tabela resumo contendo o tamanho de cada partição, o somatório dos tamanhos e a diferença dos tamanhos de cada par de partições correspondentes, além do total de partições de cada relação. As informações da tabela resumo indicam qual par de partições é o mais indicado para ser enviado ao disco de forma a manter a memória balanceada e não afetar o desempenho de nenhuma das fases do algoritmo *Hash-Merge Join*.

Para saber se a memória está ou não balanceada, a política MFP utiliza, da mesma forma que a política AFP uma constante. Se a diferença entre o total de partições da relação A e o total de partições da relação B estiver acima do valor dessa constante, a memória é considerada desbalanceada, caso contrário, a memória é considerada balanceada.

Para saber qual o par de partições é o mais indicado para ser enviado ao disco a política MFP usa a coluna da diferença, onde essa coluna determina a distancia entre a quantidade de espaço ocupado na memória pelas partições correspondentes na tabela resumo em substituição ao parâmetro a da política AFP.

Para garantir um bom desempenho da fase de *Hashing*, a política MFP deve manter um espaço suficiente na memória de forma que as novas partições que chegarem possam produzir resultados.

Para garantir um bom desempenho da fase de *Merging*, a política deve evitar liberar pequenas partições, evitando assim, um aumento no processamento da grande quantidade de pequenas partições.

A realização deste trabalho de dissertação envolveu uma investigação bibliográfica sobre processamento adaptativo, e uma análise das diversas políticas de liberação de dados da memória para o disco. No entanto, comparamos apenas uma das políticas de liberação de dados com a nova política proposta - *Adaptive Flushing Policy* (AFP), pois segundo [3] está é a mais vantajosa diante das demais listadas anteriormente.

Analisando a política MFP com relação ao tempo de processamento, observamos que há uma leve melhora no desempenho com relação à política AFP.

A política MFP com relação ao uso de Entrada/Saída, necessita de muito menos acesso ao disco que a política AFP.

Para demonstrar a viabilidade da política MFP, foi desenvolvido um protótipo que permite a execução do algoritmo de junção *Hash-Merge Join* com um banco de dados simulado.

Durante a implementação do algoritmo e simulações da execução do mesmo, foi identificada uma crítica: a parametrização da política AFP.

Diante disso, como trabalhos futuros, pretendemos:

- Estender o MFP para evitar a parametrização, com o intuito de evitar a intervenção humana;
- Adaptar o MFP para execução de junções em redes de sensores sem fio.
- Apresentar o log de execução do algoritmo HMJ na fase de execução do mesmo.

8. Referências

1. Brayner, A., Vasconcelos, E. "*MobiJoin: Um operador de junção para Bancos de Dados Móveis*". VI Workshop de Comunicação Sem Fio e Computação Móvel – 2004.
2. Aguiar M. Filho, J "AMDC – An Approach for Sharing Mobile Databases". Dissertação de Mestrado 2003.
3. Mokbel, M. F., Lu, M., Aref, W.G. "*Hash-Merge Join: A non-blocking Join Algorithm for Producing Fast and Early Join Results*". ICDE, March 2004.
4. Graef, G. "*Query Evaluation Techniques for Large Databases"*. ACM Computing Surveys, 25(2): 73-170, 1993
5. Jignesh M. Patel, Michael J. Carey, Mary K. Vernon. "*Accurate Modeling of The Hybrid Hash Join Algorithm*". SIGMETRICS 94- 5/94 Santa Clara, CA. USA 1994 ACM.
6. Ives Z. G., Florescu D., Friedman M., Levy A. Y., Weld D. S. "*An Adaptive Query Execution System for Data Integration*". *In Proceedings of the ACM International Conference on Management of Data*, SIGMOD, pages 299–310, Philadelphia, PA, June 1999.
7. Haas P. J., Hellerstein J. M.. "*Ripple Joins for Online Aggregation*". *In Proceedings of the ACM International Conference on Management of Data*, SIGMOD, pages 287–298, Philadelphia, PA, June 1999.
8. Dittrich J.-P., Seeger B., Taylor D.S, Widmayer P. "*Progressive Merge Join: A Generic and Non-Blocking Sort-based Join Algorithm*". *In Proceedings of the International Conference on Very Large Data Bases*, VLDB, pages 299310, Hong Kong, Aug.2002.
9. Dittrich J.-P., Seeger B., Taylor D.S, Widmayer P. "*On Producing Join Results Early*". *In Proceedings of the ACM Symposium on Principles of Database Systems*, PODS, pages 134–142, San Diego, CA, June 2003.
10. H. Lu, K. L. Tan. "*On Sort-Merge Algorithm for Band Jons*". TKDE 7(3): 508-510. 1995.
11. M.D. Soo, R. T. Snodgrass e C.S. Jensen. "*Efficiente Evaluation of the Valid Time Natural Join*". ICDE, pages 282-292, 1994.

12. L. Arge, O. Procopiue, S. Ramaswamy, T. Suel e J.S. Vitter "*Scalable Sweeping-based spatial join*". In VLDB, pages 570-581, 1998.
13. Roger L. Wainwright, "*Quicksort Algorithms With An Early Exit For Sorted Subfiles*" The University of Tulsa. ACM 1987
14. Urhan, T., Franklin, M. "XJoin: Getting Fast answers From Slow and Burst Networks". University of Maryland. Techinical Report. CS-TR-3994, UMIACS-TR-13. 1999.
15. Garcia-Molina H., Ullman J. D., Widom J. "*Database System Implementation*" Prentice-Hall Inc. 2000
16. Mokbel, M. F., Lu, M., Aref, W.G. "*Hash-Merge Join: A non-blocking Join Algotithm for Producing Fast and Early Join Results*". ICDE, March 2004.
17. Wilschut A. N., Apers P. M. G. "*Pipelining in Query Execution*". *In Databases, Parallel, Architectures, and their applications*, Miami, FL, 1990.
18. Wilschut A. N., Apers P. M. G "*Dataflow Query Execution in a Parallel Main-Memory Environment*". *In Proceedings of the First International Conference on Parallel and Distributed Information Systems*, PDIS, pages 68–77, Miami, Florida, Dec. 1991.
19. Werbert, E., Brayner, A. *"AJAX: An Adaptive Join Algorithm for Extreme Restrictions"*. In: Anais do XXII Simpósio Brasileiro de Banco de Dados (SBBD'2007).
20. Campos, E. V. "*Mobijoin: um operador de junção para bancos de dados móveis*". Dissertação de Mestrado. – Universidade de Fortaleza – 2006.

APENDICE A – Código fonte da classe *PolíticaMFP*

```
package hmj.politica;

import modelo.Tupla;
import java.util.*;

public class PoliticaMFP{

	public PoliticaMFP(){
	}

	public ArrayList<Integer> selecionaTupla(){
		//Referência a tupla a ser retirada da memória.
		ArrayList<Integer> linhas = null;

		if (! MediadorTabResumo.isEmpty()){
			//Se taxa de desbalanceamento >= 10% então memória
			// está desbalanceada
			if (! MediadorTabResumo.isBalanceada()){
				linhas = MediadorTabResumo.getMaiorDiferenca();
			} else {
				//Trata Memória Balanceada,
				// Retira um par de tuplas
				linhas =
				MediadorTabResumo.getMaiorSomaMenorDiferenca();
			}
		}
		return linhas;
	}

	public void adicionaTupla(Tupla tupla, int fonte){
		MediadorTabResumo.adiciona(tupla, fonte);
	}

	public void removeTupla(Tupla tupla, int fonte){
		MediadorTabResumo.retira(tupla, fonte);
	}
}
```

APENDICE B – Código fonte da classe *TabelaResumo*

```
package hmj.politica;

import java.util.*;
import modelo.*;
public class TabelaResumo {

	private int[][] tabela;
	private int numElementos = 0;

	private final int COLS = 2;

	public final static int FONTE_A = 0;
	public final static int FONTE_B = 1;

	private static double PERCENTUAL_BALANCEAMENTO = 0.10;

	public TabelaResumo(int lins){
		tabela = new int[lins][COLS];
	}

	public void atualizaTabela(Tupla tupla, int fonte){
		tabela[tupla.getCodigohash()][fonte] += tupla.getTamBytes();
		numElementos++;
	}
	public void subtraiTabela(Tupla tupla, int fonte){
		tabela[tupla.getCodigohash()][fonte] -= tupla.getTamBytes();
		numElementos--;
	}

	public void subtraiTabelaBucket(int index, int fonte){
		tabela[index][fonte] = 0;
		numElementos--;
	}

	public ArrayList<Integer> getMaiorSoma(){
		ArrayList<Integer> linha = null;

		TreeMap listaOrdenada = new TreeMap();

		for (int i = 0; i<tabela.length; i++){
			int soma = tabela[i][FONTE_A] + tabela[i][FONTE_B];
			if (soma > 0)
				listaOrdenada.put(new Integer(soma),
					new Integer(i));
		}
```

```
            if (listaOrdenada.size() > 0)
            {
                  linha = new ArrayList<Integer>();
                  int maiorSoma =(Integer)listaOrdenada.lastKey();

                  linha.add(new Integer(maiorSoma));

                  for(int i=listaOrdenada.size()-2;i>0; i--){
                     if (((Integer)listaOrdenada.get(new
                     Integer(i))).equals(new Integer(maiorSoma)))
                              linha.add((Integer)listaOrdenada.get(
                                                      new Integer(i)));
                  }
            }
            return linha;
      }

      public ArrayList<Integer> getMaiorDiferenca(){
            ArrayList<Integer> linha = null;

            TreeMap listaOrdenada = new TreeMap();

            for (int i = 0; i<tabela.length; i++){
            int dif = Math.abs(tabela[i][FONTE_A] - tabela[i][FONTE_B]);
            if (dif > 0)
                  listaOrdenada.put(new Integer(dif), new Integer(i));
            }

            if (listaOrdenada.size() > 0)
            {
                  linha = new ArrayList<Integer>();
                  int maiorDif =(Integer)listaOrdenada.lastKey();

                  linha.add(new Integer(maiorDif));

                  for(int i=listaOrdenada.size()-2;i>0; i--){
                     if (((Integer)listaOrdenada.get(new
                     Integer(i))).equals(new Integer(maiorDif)))
                              linha.add((Integer)listaOrdenada.get(new
                                                      Integer(i)));
                  }
            }
            return linha;
      }
      public ArrayList<Integer> getMaiorSomaMenorDiferenca(){
            ArrayList<Integer> linha = null;
            TreeMap listaOrdenada = new TreeMap();

            for (int i = 0; i<tabela.length; i++){
                  int dif = Math.abs(tabela[i][0] - tabela[i][1]);
                  int soma = tabela[i][0] + tabela[i][1];
                  listaOrdenada.put(new Integer(soma - dif),
                                    new Integer(i));
            }
```

```
            if (listaOrdenada.size() > 0) {
               linha = new ArrayList<Integer>();
               int maiorSomaMenorDif =(Integer)listaOrdenada.lastKey();

               linha.add((Integer)listaOrdenada.get(maiorSomaMenorDif));
               SortedMap mapa = listaOrdenada.headMap(
                                    new Integer(maiorSomaMenorDif));

               if (mapa.containsKey(new Integer(maiorSomaMenorDif))){
                linha.add(
                  (Integer)listaOrdenada.get(maiorSomaMenorDif));
               }
            }
            return linha;
      }
      private int getSomatorio(int fonte){

            int soma = 0, valor = 0;
            for (int i=0; i< tabela.length; i++){
                  switch(fonte){
                  case FONTE_A: valor = tabela[i][FONTE_A]; break;
                  case FONTE_B: valor = tabela[i][FONTE_B]; break;
                  }
                  soma += valor;
            }
            return soma;
      }
      public int getSomatorioA(){
            return getSomatorio(FONTE_A);
      }
      public int getSomatorioB(){
            return getSomatorio(FONTE_B);
      }

      public boolean isBalanceada(){
            double balanceamento = 0;
            if ((getSomatorioA() + getSomatorioB()) > 0) {
                  balanceamento = Math.abs(getSomatorioA() -
                  getSomatorioB()) / (getSomatorioA() +
                  getSomatorioB());
            }

            //MediadorDesempenho.setPercBalanceamento(balanceamento);
            return balanceamento < PERCENTUAL_BALANCEAMENTO;
      }
      public boolean isEmpty()
      {
            return (numElementos == 0);
      }
}
```

Livros Grátis

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

Milhares de Livros para Download:

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

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