OPERADORES DE JUNÇÃO BASEADOS EM MECANISMOS DE HASH PARA O PROCESSAMENTO DE CONSULTAS EM BANCOS DE DADOS

Ângelo Brayner
brayner@unifor.br
Universidade de Fortaleza. Av. Washington Soares, 1321, Edson Queiroz, CEP 60.811-905

Aretusa M. Almeida Lopes
aretusaal@edu.unifor.br
Universidade de Fortaleza. Av. Washington Soares, 1321, Edson Queiroz, CEP 60.811-905

Resumo

Os algoritmos de junção constituem um elemento chave para o desempenho do processamento de consultas. Com a evolução dos ambientes de execução de consultas tornou-se necessária o desenvolvimento de algoritmos mais eficientes para implementar o operador de junção. Neste trabalho é realizado um estudo sobre a evolução dos algoritmos de junção baseados na técnica de *hashing*. Serão analisadas estratégias convencionais como o *Simple Hash Join*, o *Grace Hash Join* e o *Hybrid Hash Join*, projetadas para arquiteturas de bancos de dados convencionais, até aquelas capazes de oferecer suporte a ambientes com processamentos de consultas mais complexos, como os de computação móvel. Os algoritmos *hash* capazes de atender a algumas das necessidades destes novos ambientes incluem o *Symmetric Hash Join*, o MobiJoin, o *Hash-Merge Join* e o MJoin.

Palavras-chave: *algoritmos hash de junção, processamento adaptativo de consultas, bancos de dados móveis.*

Abstract

Join algorithms constitute a key element for processing queries on databases. In this paper, the evolution of hash-based join algorithms is investigated. Conventional algorithms such as Simple Hash Join, Grace Hash Join and Hybrid Hash Join which were designed for conventional databases architectures are described and analyzed. Furthermore, algorithms such as Symmetric Hash Join, MobiJoin, Hash-Merge Join and MJoin for implementing the join operator in environments with more complex query processing requirements (e.g. mobile computing environment) are presented and analyzed as well.

Keywords: *hash join algorithms, adaptive query processing, mobile databases.*

1. Introdução

Com o desenvolvimento de sistemas de comunicação sem fio e de computadores portáteis, a computação móvel tornou-se realidade no ambiente computacional moderno. A integração destas duas tecnologias possibilita que computadores móveis (*laptops*, *notebooks*, *palms*, etc...) conectem-se, como componentes, a um ambiente distribuído e disponibilizem seus recursos computacionais mesmo que mudem constantemente sua localização, ou seja, apesar de não apresentarem localização fixa em uma rede.

Atualmente, o compartilhamento de informações entre múltiplas fontes de dados heterogêneas, autônomas, distribuídas e móveis tem emergido como um requerimento estratégico que deve ser suportado pela tecnologia de banco de dados. Usuários que carregam equipamentos portáteis estão habilitados para acessar serviços de bancos de dados independentemente da sua localização física ou padrão de movimentação. De acordo com este novo cenário, comunidades de bancos de dados móveis podem ser formadas. Definimos uma Comunidade de Bancos de Dados Móveis (MDBC) como uma coleção dinâmica de bancos de dados móveis, distribuídos e autônomos, na qual cada usuário pode acessar dados através de uma infra-estrutura de comunicação sem fio [0].

O processador de consultas tem por objetivo extrair os dados de um banco de dados da forma mais eficiente possível, caracterizando-se como um dos módulos mais importantes de um Sistema Gerenciador de Banco de Dados (SGBD). Diferentemente do processamento de consultas em sistemas de bancos de dados distribuídos convencionais, processar consultas em bancos de dados móveis significa executar consultas sobre um número variável de bancos de dados. Isto significa que o processador de consultas, ao executar consultas sobre bancos de dados móveis, opera em um ambiente imprevisível e que pode se modificar constantemente.

Portanto, técnicas tradicionais para processamento de consultas distribuídas e operadores precisam ser revistos, pois, o uso de tais técnicas e operadores tem se mostrado ineficiente ao se processar consultas sobre bancos de dados móveis, por várias razões, entre elas, as mais críticas são as seguintes:

- Imprevisibilidade do tempo de resposta dos bancos de dados móveis. Há pouco conhecimento a respeito das taxas de entrega de dados dos bancos de dados envolvidos em uma consulta. Desta forma, mesmo que o otimizador de consulta tenha escolhido o melhor plano em um determinado instante, baixas taxas de entrega dos dados (causadas por desconexões constantes na rede sem fio) no momento da execução da consulta podem tornar o plano ineficiente. Para contornar tal problema é necessária a utilização de operadores que se adaptem ao novo cenário de execução da consulta, para reagir a latências não previstas;
- Utilização de operadores que possuem mais de uma fase de execução e que não apresentam suporte para a técnica de *pipelining*[1]. Por exemplo, o operador *hash join* possui duas fases de execução (construção e comparação). A fase de comparação só pode ser executada, após a execução da fase de construção por completo (ou seja, para todas as tuplas das duas relações da operação de junção). O desempenho deste operador é afetado caso ocorra atraso na entrega dos dados, fazendo com que a primeira fase forneça seu resultado para a fase de comparação a taxas muito baixas de entrega de tuplas;
- Ausência de estatísticas. Considerando o número variado de bancos de dados móveis autônomos que entram na comunidade o sistema tem pouca ou nenhuma estatística sobre os mesmos;

Nesse sentido, houve um direcionamento nas pesquisas com o objetivo de elaborar propostas destinadas a tornar o processamento de consultas em bancos de dados adaptável ao cenário de execução de cada consulta. A idéia é capacitar mecanismos de consultas de SGBDs a adequar o seu funcionamento, em tempo de execução, a novos cenários. Duas estratégias têm sido investigadas no intuito de dotar processadores de consultas com a propriedade de adaptabilidade. Uma estratégia é alterar o Plano de Execução de Consultas em tempo de execução, tornando-o adaptativo. A outra estratégia é o desenvolvimento de operadores adaptativos que ajustam sua execução dinamicamente em resposta à ocorrência de eventos pré-definidos (por exemplo, *overflow* de memória ou atraso na entrada de tuplas).

Algumas abordagens sobre planos de consultas adaptativos incluem a construção de planos alternativos parciais ou completos. Neste contexto estão as propostas PAIR, no qual o otimizador constrói um plano de consulta parcial para cada junção, cujas relações não estejam bloqueadas no momento, ordenando os planos por ordem decrescente de custo e executando-os nesta ordem; IN (*Include Delay*), no qual o otimizador gera um plano de consulta alternativo completo, considerando longos atrasos para relações cujos dados não chegaram ainda, assim a execução da junção entre estas relações terá menor prioridade no plano; e ED (*Estimated Delay*), que inicialmente assume que o atraso dos dados de uma relação será curto, aumentando, sucessivamente, a estimativa de atraso a cada nova geração de plano de consulta, caso os dados continuem sem chegar [0]. Em [0] é proposta uma estratégia para trabalhar com a migração dinâmica de planos, na qual o otimizador seleciona, dinamicamente, o plano de consulta mais eficiente para cada novo cenário apresentado, tendo por base as estatísticas.

Os operadores adaptativos são comumente implementados por algoritmos que consideram diferentes etapas para cada evento pré-definido a ser tratado. A maior parte destes algoritmos está concentrada na implementação de operadores de junção.

Considerando a eficiência dos algoritmos baseados em *hash* e visando atender aos requisitos, que não poderiam ser tratados eficientemente pelos algoritmos de junção convencionais, novos operadores de junção, baseados em *hash*, foram propostos. Algumas dessas novas propostas foram construídas como extensões dos algoritmos *Simple Hash Join, Grace*

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

Hash Join e *Hybrid Hash Join* [0]. Os novos algoritmos buscavam suportar o processamento de consultas sobre dados distribuídos, atender às necessidades de adaptabilidade e oferecer bons desempenhos. Estas características vêm sendo tratadas pela classe de algoritmos adaptativos como o *Symmetric Hash Join* [0], o MobiJoin [0], o XJoin [0,0], o *Hash-Merge Join* [0] e o MJoin [0].

Neste artigo, é realizada uma análise de alguns dos operadores de junção baseados em técnicas de *hash*. São abordadas as características, pontos fortes e pontos fracos, tanto dos algoritmos para processamento convencional de consultas quantos das propostas que estenderam estes algoritmos, incorporando a eles características de adaptabilidade.

Este trabalho está estruturado conforme descrito a seguir. A seção 2 apresenta a evolução dos ambientes e dos módulos processadores de consulta. A seção 3 analisa os mais importantes algoritmos de junção baseados em hash que incorporam a propriedade de adaptação ao cenário de execução de cada consultas. Na seção 4 é apresentada a conclusão do artigo.

2. Evolução dos ambientes de processamento de consultas

A importância do processador de consultas para os SGBDs fez deste módulo um dos principais focos de estudo voltados à necessidade de aumentar o desempenho no fornecimento de resultados. Nesta seção, é feita uma discussão sobre o processamento de consultas em bancos de dados. Para tanto, será descrito o processamento convencional de consultas. Após isto, será caracterizado o que vem a ser o processamento adaptativo de consultas.

2.1 Processamento de consultas

O processador de consultas tem por objetivo extrair os dados de um banco de dados da forma mais eficiente possível, caracterizando-se como um dos módulos mais importantes de um SGBD. Basicamente, o processamento de uma consulta consiste das seguintes fases [0]:

1. *Parsing* (Análise): nesta fase, a consulta recebida deve estar escrita em uma linguagem de alto nível como o SQL (*Sequel Query Language*). O *parser* verifica se as relações e atributos utilizados são válidos para o esquema do banco de dados considerado e, ainda, se a consulta foi escrita conforme as regras gramaticais da linguagem adotada.
2. Tradução: em seguida é construída uma árvore de análise (*parse-tree*) para a consulta. Esta árvore é comumente traduzida para uma expressão equivalente da álgebra relacional, que é uma forma de representação interna utilizada em bancos de dados relacionais.
3. Otimização: a fase seguinte consiste no processo de seleção de uma estratégia de execução que minimize o custo de processamento da consulta.

4. Avaliação: nesta fase a consulta é executada com base na estratégia escolhida pelo otimizador.

A etapa de avaliação já possui um roteiro da execução da consulta, porém, a escolha do algoritmo que efetivamente irá buscar e combinar os dados é determinante para o desempenho da consulta como um todo.

Por ser o operador de junção um dos pontos mais críticos no processamento de uma consulta, inúmeros algoritmos foram propostos para realizar esta tarefa, grande parte deles baseou-se na utilização de laços aninhados, estratégias de *merge* e de *hash*. Entre os que utilizaram laços aninhados estão: *nested loop join*, *block nested loop join* e o *index nested loop* [18]. O *Merge-Join* [18] é o exemplo mais clássico de utilização de algoritmos *merge* para o processamento de junções. Já a classe de algoritmos que adotaram técnicas de *hash* é extensa e compreende, entre outros, o *Simple Hash Join*, o *Grace Hash Join* e o *Hybrid Hash Join* [0]. Embora nem todos estes algoritmos tenham se consolidado como alternativas viáveis em muitos bancos de dados comerciais, sua relevância como base para outras propostas, particularmente para os algoritmos adaptativos, é incontestável.

2.2 Processamento de consultas adaptativo

Um processador de consultas adaptativo deve ser capaz de alterar seu comportamento em resposta a mudanças no ambiente de execução da consulta. A adaptabilidade permite ao processador de consultas ajustar-se, de modo a reduzir as perdas de desempenho causadas por problemas como: alterações nas taxas de fornecimento de dados, mudanças no ambiente de execução da consulta ou desconhecimento de estatísticas. Além disso, esta classe de processadores de consultas consegue melhorar seu tempo global de resposta intercalando estágios de otimização e execução [1]. Os aspectos mais importantes a considerar nestes sistemas são [4]:

1. Freqüência da adaptabilidade: freqüência com que o sistema pode receber informações e adaptar seu comportamento ao novo cenário de execução da consulta.
2. Efeitos da adaptabilidade: efeitos provocados no comportamento do sistema diante de um determinado evento que alterou a configuração do ambiente.
3. Extensão da adaptabilidade: consiste na repetição, durante a execução da consulta, do processo de adaptação do comportamento do sistema em função das mudanças no ambiente.

Para muitas aplicações, longas esperas até se obter o resultado final de uma consulta é inviável. A alternativa encontrada pelos algoritmos adaptativos para amenizar este problema foi buscar a disponibilização de resultados de forma simultânea ao processamento da consulta. Assim, seria possível ao usuário tomar decisões baseadas nos dados até então liberados, antes mesmo de receber o resultado completo da consulta. A adaptabilidade requer um processamento complexo que pode envolver a alteração dinâmica de planos de execução e a utilização de algoritmos de difícil implementação. Embora estas duas etapas não sejam dependentes entre si, é mais natural que um processador de consultas adaptativo seja capaz de reuni-las.

Um processador de consultas adaptativo deve ser capaz de considerar as alterações do ambiente durante a execução da consulta. Alterações quanto ao volume de dados a processar, criação de índices, falhas ou atrasos na recepção de dados deveriam ser consideradas durante o processamento, de forma a ser possível ajustar os planos de execução, dinamicamente, tornando-os mais adequados às novas configurações do ambiente. Neste contexto, algumas abordagens incluem as estratégias [0,0]:

1. PAIR (*Total work-based optimizer*): nesta proposta o otimizador constrói um plano de execução para cada operador de junção cujas relações não estejam com o fornecimento de dados bloqueado[2]. O otimizador analisa, sucessivamente, cada um dos planos, calcula o custo correspondente, escolhe o plano de menor custo e o executa. Quando todas as relações estiverem desbloqueadas, o PAIR constrói uma árvore de consultas única, de modo a compor um plano de consulta completo.
2. IN (*Include Delayed*): neste caso o otimizador considera que menores tempos de resposta levam a menores custos. Assim, o otimizador gera, a cada interação, um plano de consulta alternativo completo, porém, para as relações que estiverem bloqueadas é assumido que o atraso dos dados será grande e, portanto, estas relações deverão ter menor precedência de execução dentro do plano de consulta elaborado.
3. ED (*Estimated delay*): Assim como a estratégia IN, a ED trabalha com o otimizador baseado em tempos de resposta. Assume que os atrasos de uma fonte não serão longos, porém, a cada nova interação, a qual leva à geração de um plano completo, atribui piores custos para as relações que continuarem bloqueadas.
4. Migração dinâmica de planos: propõe a troca, em tempo de execução, de um plano de consulta por outro que seja semanticamente equivalente, mas melhor que o anterior, por ser mais adequado à nova configuração do ambiente. A avaliação de custo é feita considerando que as estatísticas são conhecidas.

Estas propostas podem ser particularmente aplicáveis a ambientes que tratam os dados como streams, os quais são fornecidos continuamente, levando à imprevisibilidade tanto quanto ao volume de dados a ser recebido quanto à freqüência de entrega destes. Apesar de muitas propostas já terem surgido, a formulação dinâmica de planos de consulta adaptáveis ainda está distante de alcançar a maturidade das estratégias convencionais [0].

Os operadores adaptativos de consultas têm como foco principal o processamento de operadores de junção. Entre esses algoritmos adaptativos estão o *Symmetric Hash Join*, o MobiJoin [0], XJoin [0,0], o *Hash-Merge Join* [0] e o MJoin [0]. Estes algoritmos associam estratégias de *hash* a técnicas de *pipeline*[3]. A estratégia do *pipeline* é extensivamente utilizada pelos algoritmos adaptativos, pois permitem paralelisar etapas da execução da consulta. O paralelismo dos operadores pode ser classificado em duas estratégias: *inter-operator*, quando ocorre a execução paralela de diferentes operadores considerando uma única consulta e; *intra-operator*, quando um mesmo operador é aplicado a partições diferentes, de um mesmo conjunto de dados, submetidas a diferentes processadores.

[2] O bloqueio se refere ao fato de não estar sendo fornecido, correntemente, dados de uma relação.

[3] Permite que os dados sejam processados por um operador e passados diretamente ao operador seguinte, evitando o custo da materialização do dado e permitindo a paralisação de etapas da execução da consulta.

Embora os algoritmos adaptativos não sejam implementados em grande parte dos SGBDs comerciais, o seu estudo ganhou impulso devido ao surgimento de aplicações que trabalham com *data streams* [0,0,**Erro! A origem da referência não foi encontrada.**,0,0,0] e bancos de dados distribuídos.

2.3 Processamento de consultas adaptativo e distribuído

Aplicações desenvolvidas para ambientes distribuídos, como é o caso das redes sem fio, têm se tornado cada vez mais comuns. Bancos de dados capazes de suportar esta arquitetura precisam contar com otimizadores de consulta que levem em consideração, durante a escolha do melhor plano, não mais apenas a busca pela redução de acessos a disco, mas também, a imprevisibilidade na freqüência de recepção dos dados, a falta de informações quanto às estatísticas das fontes remotas e a recepção de um volume desconhecido de dados. Estas características inerentes a este ambiente implicam em complexidades extras a serem tratadas pelo módulo processador de consultas.

Considerando distribuição de dados, uma nova categoria de aplicações vem sendo particularmente estudada, são as que trabalham com *data streams*. Os *data streams* caracterizam-se como uma seqüência de itens que chega continuamente. Sendo assim, não é possível armazenar o *data stream* inteiro localmente, o que requer que as consultas também sejam executadas continuamente e que os resultados sejam retornados incrementalmente, a medida em que vão sendo produzidos [**Erro! A origem da referência não foi encontrada.**].

Aplicações sobre *data streams* estão particularmente presentes em redes sem fio e redes WANs como a internet. Estas aplicações modelam os dados como *data streams* transientes[4] e não mais como dados persistentes, conforme mostrado na Tabela 1. Arquiteturas convencionais de bancos de dados não são capazes de suportar consultas contínuas; assim, já é reconhecido que aproximação e adaptabilidade são ingredientes chaves para a execução de consultas sobre *data streams* [0].

A dificuldade envolvida no processamento de consultas sobre *data streams* pode ser verificada na realização de uma operação de junção simples. O operador de junção deve consumir a entrada inteira antes de fornecer o resultado final [0], assim, de acordo com o algoritmo considerado, cada tupla de uma relação pode precisar ser comparada com todas as demais da outra relação. Fica difícil atender a esta propriedade quando se consideram conjuntos de dados potencialmente infinitos, como ocorre com os *data streams*, e quando se têm memórias de tamanho limitado. Algumas alternativas encontradas para o processamento de consultas sobre *data streams* envolvem a utilização de *sliding windows*[5], limitantes do volume de dados, ou a adoção de intervalos de confiança, associados aos resultados parciais fornecidos [0]; sendo retornados, em ambos os casos, resultados apenas próximos ao real.

Uma das técnicas para a produção de respostas aproximadas para uma consulta sobre *data streams* é a realização da consulta não sobre todo o histórico de dados, mas utilizando *sliding windows* [0], intervalos que especificam um conjunto limitado de dados que será considerado no processamento da consulta. Algoritmos que usam *sliding windows* podem trabalhar com três possíveis abordagens: baseada em tempo, neste caso são considerados apenas os dados recepcionados em um determinado intervalo de tempo; baseada no volume de dados suportado pela consulta; e baseada em pontos de referência, assume condições de seleção sobre os dados para reduzir a quantidade de informações a ser considerada [0]. Ainda considerando filtros sobre *data streams*, vem sendo fortemente explorada a técnica de *punctuation*, que são restrições estáticas ou dinâmicas assumidas para a eliminação de dados. Outras estratégias, ainda, utilizam-se da combinação destas abordagens, como é o caso do PWJoin [0], que associa *sliding windows* baseadas em tempo com técnicas de *punctuation*.

Os *data streams* também podem ser tratados por algoritmos como o *Ripple join*. Este algoritmo fornece intervalos de confiança associados a resultados parciais. Estes intervalos de confiança são obtidos através de análises estatísticas sobre os dados, os quais determinam a margem de certeza quanto a corretude dos resultados parciais fornecidos até o momento. *Ripple Joins* são adaptativos, sendo capazes de ajustar seu comportamento de acordo com as propriedades estáticas dos dados. Com base nestas propriedades, este algoritmo é capaz, não apenas de estimar a seletividade e o custo de processamento, mas também estimar a qualidade do resultado correntemente mostrado ao usuário, permitindo que a consulta seja interrompida tão logo a resposta seja suficientemente precisa [0].

[4] Data streams transientes são constituídos por seqüência de dados transitórios.

[5] Sliding windows permitem definir um filtro sobre um conjunto de dados de forma a compor um conjunto finito destes dados.

Os planos de consulta formulados para *data streams* também apresentam diferenças claras em relação a planos de consultas convencionais. Enquanto estes últimos trabalham com planos construídos como uma árvore, que é executada de maneira *top-down*, orientada ao consumidor; os planos sobre *data streams* adotam um modelo *botton-up*, orientado pelo produtor, pois os dados são entregues continuamente, independentemente de eles estarem sendo requisitados para a produção de resultados de consultas [0].

Considerando o processamento distribuído de consultas e objetivando minimizar os problemas advindos deste ambiente de execução, foi sugerido em [0] que a arquitetura de um processador dinâmico de consultas deveria ser dividido em três camadas: 1. *Dynamic query optimizer*, responsável por implementar dinamicamente a estratégia de reotimização; 2. *Dynamic query scheduller*, capaz de receber um plano como entrada e produzir um escalonamento de planos, incluindo as prioridades de execução das consultas e; 3. *Query fragment evaluator*, responsável pela execução da consulta, seguindo as prioridades estabelecidas pelo *Dynamic query scheduler* para a execução dos fragmentos da consulta. O processamento consistiria em enviar sub-consultas para empacotadores das fontes de dados, e o resultado final seria fruto da integração dos resultados das sub-consultas. Porém, devido ao conhecimento limitado das fontes remotas, a execução integrada destes planos pode resultar em desempenhos pobres. Estes problemas podem ser amenizados com a adoção de estratégias de adaptabilidade, capazes de realizar re-otimizações em cada uma das três camadas da arquitetura proposta, de acordo com cada nova configuração assumida durante a execução.

Resumidamente, é apresentada na Tab. (1) [0] uma comparação entre algumas das características que diferenciam um sistema de banco de dados convencional daqueles que trabalham com *data streams.*

Tabela 1: Análise comparativa entre sistemas de banco de dados e sistemas de data streams.

Características	**Sistemas de bancos de dados**	**Sistemas de *data streams***
Recursos	Razoável disponibilidade	Limitados
Modelo de dados	Relações persistentes	Relações transitórias
Relação	Conjunto de tuplas	Seqüência de tuplas
Atualização	Modificação	Acrescentar dados ao fluxo
Consulta	Transitória	Persistente
Resposta das consultas	Exata	Aproximada
Avaliação das consultas	Arbitrária	De uma única vez
Plano de consultas	Fixo	Adaptativo
Processamento de Consulta	Suporta análises sofisticadas	Razoável complexidade

3. Evolução dos algoritmos *hash* de junção

Os algoritmos baseados em *hash* para o processamento de junções têm-se mostrado eficientes em diferentes arquiteturas de banco de dados. A associação deste algoritmo com outros como *nested loops*, *pipelines* e, mais recentemente, *sliding windows*, têm sido fundamentais na elaboração de muitas das soluções propostas no contexto do processamento de consultas.

3.1 Algoritmos convencionais de junção baseados em hash

Nas arquiteturas convencionais de bancos de dados o principal objetivo, como requisito de desempenho, é a redução do número de acessos a disco. Embora muitas propostas baseadas em *nested loops* e em *merges* tenham sido elaboradas, elas não ofereciam desempenhos tão bons quanto aqueles conseguidos pelos algoritmos baseados em *hash*, pelo menos quando se tratam de consultas cujas cláusulas de seleção sejam igualdades.

O *Simple Hash Join* foi uma das primeiras propostas adotando algoritmos *hash* para o processamento de junções e, juntamente com o *Grace Hash Join*, foi base para a elaboração do *Hybrid Hash Join*, largamente adotado nos bancos de dados comerciais.

3.1.1 Simple Hash Join

Este algoritmo está entre as versões mais simples de algoritmos de junção baseados em *hash*. Caracteriza-se pela utilização de funções *hash* para classificar as tuplas de acordo com os valores destas funções. A qualidade do *Simple Hash Join* está em

reduzir o número de comparações entre as tuplas. Ele mostra bons resultados quando é possível manter, pelo menos, uma das relações em memória. A eficiência do algoritmo também está diretamente relacionada à qualidade da função *hash*, a qual deve ter como propriedades básicas a randomicidade e a uniformidade na distribuição dos dados.

Na primeira etapa do *Simple Hash Join* é utilizada uma função *hash* para particionar as tuplas das duas relações envolvidas na junção, compondo uma tabela *hash* para cada uma das relações [0]. Cada entrada na tabela *hash* corresponde a um valor de função *hash* compartilhado por um conjunto de tuplas. Na etapa seguinte é utilizada uma junção de laço aninhado indexada entre as duas relações, de forma que a menor relação é eleita para ser a de construção e a outra será a de teste. Um índice *hash* deve ser criado para cada uma das partições da relação de construção, como resultado da aplicação de uma nova função *hash*, diferente da primeira, aplicada às duas relações. O algoritmo examina, então, todos as tuplas da relação de teste, utilizando os índices associados às partições da relação de construção para buscar valores coincidentes.

Um aspecto importante a considerar na segunda etapa é que, pelo menos, uma partição da relação de construção e o seu índice *hash* correspondente devem ser completamente acomodados em memória, caso contrário as duas relações devem ser submetidas a novas funções *hash*, até que este requisito seja atendido. Esta estratégia de particionar recursivamente as relações é chamada de *resolution*, por dividir a relação em partições cada vez menores, até chegar a tamanhos aceitáveis. Vale salientar que a mesma preocupação não existe para a relação de teste.processador de consultas adaptativo deve ser capaz de alterar seu comportamento em resposta a

3.1.2 Grace Hash Join

O *Grace Hash Join* trabalha em duas etapas. De modo semelhante ao que ocorre no *Simple Hash Join*, na primeira etapa as duas relações são inteiramente particionadas usando a mesma função *hash* aplicada sobre os atributos de junção. Na etapa seguinte as partições correspondentes às duas relações são combinadas e o resultado da junção é gerado. A principal diferença em relação ao *Simple Hash Join* é que o *Grace Hash Join* assume que haverá pouca disponibilidade de memória. Caso isso se confirme, seu desempenho será melhor que o do *Simple Hash Join* [0].

Como estratégia para garantir que partições da relação de construção caibam em memória, juntamente com os índices *hash* correspondentes, este algoritmo procura dividir a relação, inicialmente, em um grande número de pequenas partições. Quando é verificado que o espaço em memória é muito maior que o tamanho das partições, a estratégia *avoidance* é utilizada para combinar as partições recursivamente, até que o tamanho ideal seja encontrado. Assim, se o espaço disponível em memória for grande, a tendência será de perda de desempenho, devido às inúmeras combinações que terão de ser feitas.

3.1.2 Hybrid Hash Join

O *Simple Hash Join* funciona bem com memórias grandes, enquanto que o *Grace Hash Join* oferece bons resultados com memórias menores. Como proposta para flexibilizar os requisitos de memória, o *Hybrid Hash Join* (HHJ) [0] adotou uma eficiente política de gerenciamento de dados entre disco e memória. Este algoritmo propõe uma otimização em situações em que o tamanho da memória é relativamente grande, mas não o suficiente para abrigar a relação de construção inteira [0]. Basicamente, o HHJ é realizado em duas etapas, a primeira ocorre em memória, se houver espaço suficiente, caso contrário ocorrerá em disco [0].

Assim como outros algoritmos *hash*, o HHJ prevê uma fase que consiste no particionamento das duas relações, através da aplicação de uma função *hash* sobre os atributos de junção. Primeiramente uma relação inteira deve ser particionada, impondo um bloqueio que impede o adiantamento das operações da junção. Caso o limite de memória ou da partição seja alcançado, a última partição que teve tuplas alocadas é descarregada em disco. Porém, é mantido um bloco vazio em memória para esta entrada na tabela *hash*. A segunda relação é então particionada com a mesma função *hash* e, caso um novo limite de memória seja alcançado, a partição a ser eleita para ser descarregada em disco será, se possível, alguma que tenha um bloco vazio na relação oposta. Esta estratégia melhora o desempenho da etapa seguinte, visto que mantêm pares de partições correspondentes ou só em memória ou só em disco [0].

Na segunda etapa os resultados começarão a ser produzidos. Primeiramente são combinadas as partições que estão em memória e depois serão comparadas aquelas que estão em disco. Durante o processo de comparação, uma nova função *hash* é aplicada às relações para compor os índices associados às partições da relação de construção, sendo que a memória deve ser capaz de guardar, pelo menos, uma partição e seu índice. É realizada uma junção de laço aninhado indexado, fazendo com que cada valor de tupla da partição de teste seja procurada, via índice *hash*, na partição correspondente da relação oposta. Caso valores coincidentes sejam encontrados eles serão materializados para compor o resultado final da junção.

3.2 Algoritmos adaptativos de junção baseados em hash

Os algoritmos convencionais destinados ao processamento de junção baseados em *hash* não são capazes de suportar os requisitos de arquiteturas como as de um banco de dados móvel. Assim, uma nova classe de algoritmos, denominados adaptativos, foi proposta. Estes algoritmos deveriam ter a capacidade de oferecer bons resultados no processamento de consultas, mesmo em situações de imprevisibilidade, inerentes aos ambientes móveis.

3.2.1 Symmetric Hash Join

O *Symmetric Hash Join* (SHJ) [0] foi um dos primeiros algoritmos a introduzir a idéia de adaptabilidade para operadores de junção. Este algoritmo administra as operações de *hash* e combina as tuplas utilizando apenas memória principal. Também trabalha fortemente com o conceito de *pipeline*[6], que permite um alto grau de paralelismo. Assim, os resultados podem ser fornecidos simultaneamente à realização da junção, aumentando a usabilidade das aplicações. Embora garanta ótimos resultados devido à eficiência de trabalhar inteiramente em memória, esta é também a maior limitação deste algoritmo.

Durante o processamento da junção, é construída uma tabela *hash* para cada uma das relações. Quando uma nova tupla chega, uma função *hash* é aplicada sobre os seus atributos de junção. Em seguida esta tupla é comparada com todas as tuplas da partição correspondente da relação oposta. Se alguma coincidência de valores for encontrada, a tupla é devolvida como resultado [0]. Finalmente, a tupla é adicionada à partição da tabela *hash* que tem o mesmo valor de função *hash*. A tupla deve ser guardada para que possa ser usada em comparações com as novas tuplas que irão chegar.

Um outro ponto a ser considerado é que não há necessidade de se eleger uma relação para ser a de construção e uma outra para ser a de teste, pois o SHJ é simétrico. Esta propriedade permite que cada relação assuma, ao mesmo tempo, o papel de construção e de teste. Assim, se uma das fontes interromper momentaneamente o fornecimento dos dados, o processamento da consulta continua, embora a tendência seja a redução dos resultados parciais gerados.

3.2.2 MobiJoin

Assim como o SHJ, o MobiJoin é um operador simétrico de junção para processamento de consultas, tendo sido projetado para atender os requisitos dos bancos de dados móveis. Este algoritmo estende o XJoin, sugerindo uma nova estratégia para evitar a duplicação dos resultados da junção. Basicamente, este operador busca garantir as seguintes propriedades: 1. produção incremental de resultados, simultaneamente ao processamento da junção; 2. continuidade de processamento mesmo quando ambas as fontes cessarem o fornecimento dos dados; e 3. reação adequada em situações de limitação de memória durante à execução do operador [5]. O MobiJoin comporta-se como um operador de consultas adaptativo, capaz de tirar proveito da imprevisibilidade na entrega dos dados.

O algoritmo MobiJoin trata a junção de registros, provenientes de fontes distribuídas, aplicando técnicas de gerenciamento de memória que permitem uma eficiente alocação de registros entre memória principal e secundária. Além disso, o MobiJoin controla processos executados em *background*, como forma de garantir bons tempos de resposta, tanto para um rápido retorno dos resultados iniciais quanto do *throughput* geral da junção.

O MobiJoin é organizado em três etapas: na primeira, a junção é realizada usando apenas dados residentes em memória; na segunda, a junção é feita combinando uma partição de uma fonte que está em memória com a partição em disco correspondente da outra fonte; e na terceira, a junção é realizada entre a partição de uma fonte em disco e as partições correspondentes da relação oposta, residentes em disco e em memória.

No primeiro estágio, cada uma das tuplas que chega é submetida a uma função *hash* que irá identificar a partição da tabela *hash* que a tupla deverá ser alocada. Caso a tupla chegue e haja espaço suficiente em memória, ela é comparada a todas as tuplas da partição correspondente, em memória, da relação oposta, sendo que os valores coincidentes poderão ser retornados imediatamente como resultado. Se o limite da memória for alcançado, partições são eleitas e descarregadas em disco, de acordo com a necessidade de espaço. Quando todos as tuplas forem recebidas, a terceira etapa é iniciada. Porém, caso as entradas cessem o fornecimento de dados apenas por um determinado tempo, um bloqueio provocado por *timeout* é acionado e a segunda etapa é iniciada.

[6] Pipeline: estratégia que evita que os dados tenham que ser materializados antes de serem liberados para o próximo operador.

Na segunda etapa são realizadas junções entre partições correspondentes das duas relações, sendo que a porção residente em disco de uma partição é combinada com a porção residente em memória da outra. Após o exame de cada porção de partição em disco, é verificado se alguma das entradas de dados produziu novos registros, se isto se confirmar a primeira etapa é retomada, caso contrário, a segunda etapa continua examinando novas porções residentes em disco.

A terceira e última etapa é executada apenas quando todas as tuplas tiverem sido recebidas. Neste estágio, cada uma das partições em disco deverá ter suas tuplas comparadas as tuplas das porções residentes em disco e em memória da partição oposta correspondente. A terceira etapa garante que o resultado completo será produzido, porém, todas as tuplas devem ser avaliadas, embora apenas aquelas ainda não comparadas gerem resultados.

O algoritmo também deve garantir que não haja duplicações no resultado final da consulta, a exemplo do que também ocorre no XJoin, a segunda e terceira etapas do algoritmo podem vir a comparar o mesmo par de tuplas inúmeras vezes. Como forma de evitar duplicações o XJoin adota um complexo controle de marcadores de tempo.

O MobiJoin distingue-se pela simplicidade da estratégia adotada para evitar as duplicações. O algoritmo prevê a associação de um identificador único para cada tupla que chega, considerando a partição em que esta é alocada. Para identificar tuplas já comparadas, sempre que uma porção de partição for transferida de memória para disco, uma tabela de controle é gerada indicando quais tuplas desta partição foram comparadas com quais tuplas da partição oposta. Uma relação é representada na tabela como linha e a outra como coluna, a ocorrência de 1 ou 0 nas células da tabela indica se o par já foi comparado ou não. A vantagem desta abordagem é que a tabela gerada trabalha com valores binários, o que requer pouca memória.

3.2.3 Hash-Merge Join

O *Hash-Merge Join* (HMJ) é um algoritmo adaptativo para o processamento de junções sobre dados de fontes remotas capaz de lidar com situações de imprevisibilidade na entrega dos dados. Trabalha com dois objetivos básicos: busca reduzir os tempos para produção dos resultados iniciais e permite o fornecimento de resultados mesmo em situações de atrasos das fontes remotas. O HMJ estende os algoritmos XJoin e PMJ (*Progressive Merge Join*) [0], além de propor uma nova política para o gerenciamento de memória denominada *adaptative flushing* [0].

O problema encontrado no XJoin e no MobiJoin é que, apesar do rápido fornecimento inicial dos dados, a complexidade envolvida nas operações de E-S é alta; o controle das tuplas visando evitar duplicações no resultado gera overheads e as comparações, inúmeras vezes, do mesmo par de tuplas prejudica o desempenho final consulta. Já o PMJ divide a memória em apenas duas únicas partições, uma para cada fonte, não produzindo nenhum resultado até que todas as tuplas sejam recepcionadas ou que o limite da memória seja alcançado.

Como alternativa à resolução das limitações destes dois algoritmos, o HMJ propõe agregar a eficiência no fornecimento dos resultados inicias presente no XJoin, aos bons resultados no desempenho total da consulta demonstrado pelo PMJ.

O HMJ é executado em duas fases. A primeira fase trabalha com uma estratégia de *hash* e a segunda com uma estratégia de *merge*. A fase de *hash* é semelhante à primeira fase dos algoritmos XJoin e MobiJoin, na qual os dados são combinados em memória produzindo resultados imediatamente. A fase de *merge* é ativada quando as duas fontes remotas interromperam o envio dos dados.

Durante a primeira fase do HMJ, caso haja espaço suficiente em memória, uma função *hash* é aplicada sobre os atributos de junção da tupla recepcionada para identificar a partição a que ela pertence. A tupla é então comparada a todas as tuplas da partição correspondente, em memória, da relação oposta. Caso os valores comparados sejam coincidentes, a tupla irá compor o resultado. Em seguida, a tupla é adicionada em memória à partição da tabela *hash* a que pertence. Se as duas fontes remotas cessarem o envio dos dados a fase de *merge* é ativada.

A diferença entre a primeira fase do HMJ e a primeira fase do XJoin e do MobiJoin está nas ações realizadas quando os limites da memória são alcançados. No caso do HMJ a política de alocação de memória proposta, *adaptative flushing*, elege duas partições, uma de cada fonte, que tenham o mesmo valor de função hash, para serem descarregados em disco. Porém, antes das partições irem para disco, elas são ordenadas e ganham rótulos de mesmo valor, que serão utilizados na segunda fase para identificar um par de partições que já foi comparado em memória. Como o algoritmo empregado para ordenar as tuplas é o *Sort-Merge*, é possível produzir resultados enquanto a ordenação é realizada.

Na segunda etapa, fase de *merge*, o fornecimento das tuplas deve ter cessado, momentânea ou definitivamente. O processamento segue com a combinação das tuplas que estão em disco. Deverá haver inúmeros pares de partições em disco, sendo que cada uma das partições de uma fonte será combinada com todas as partições em disco da outra relação, exceto a que tenha o mesmo valor de rótulo, pois estas já devem ter sido combinadas na primeira fase.

O principal objetivo da política *adaptative flushing* é descarregar as tuplas para disco adaptativamente. Se uma fonte for bloqueada e a outra não, haverá uma tendência de aumento do volume de dados de uma fonte em relação à outra. A política de gerenciamento de memória deve ser capaz de administrar situações como estas, de forma a manter o balanceamento entre dos dados das duas relações. Outro fato importante a observar é que a fase de *hash* é beneficiada por um grande número de pequenas partições, enquanto que a fase de *merge* beneficia-se de um pequeno número de grandes partições.

No caso da fase de hash, partições pequenas vão reduzir o número de comparações a que cada tupla terá de ser submetida em memória, além disso, partições menores serão descarregadas para disco, deixando sempre a memória próxima a seus limites, maximizando o número de tuplas de resultado produzidas. Por outro lado, um pequeno número de grandes partições em disco resulta em menos operações de *merge* e em páginas de dados mais bem aproveitadas, beneficiando a fase de *merge*. Assim, a política *adaptative flushing* compromete-se com os seguintes requisitos: 1.Oferecer suporte à fase de *hash*, podendo produzir resultados a medida que novas tuplas vão chegando; 2. Oferecer suporte à fase de *merge*, evitando descarregar partições muito pequenas em disco e; 3. Manter a memória balanceada em relação ao volume de dados em memória das duas fontes.

3.2.4 Mjoin

O MJoin [0] é um algoritmo de junção que estende o XJoin pela capacidade de explorar metadados[7], visando identificar dados que podem ser eliminados devido a suas baixas expectativas de uso. Além de economizar recursos computacionais, este algoritmo refina a lógica de execução para atingir diferentes objetivos de otimização.

As características do MJoin permitem a este algoritmo suportar as propriedades complexas dos *data streams*. Visando atender a essas propriedades o MJoin tem por objetivos: 1. Ser capaz de trabalhar com conjuntos de dados potencialmente infinitos; 2. Suportar a imprevibilidade das taxas de recepção de dados; 3. Adaptar-se a cenários de execução dinâmicos, nos quais os metadados dos *data streams* não estão disponíveis para o otimizador de consultas antes da execução; 4. Atender a múltiplos objetivos de otimização, inclusive execução contínua de consultas e geração incremental de resultados; 5. Permitir que mais de um operador possa ser executado simultaneamente; 6. Oferecer um *overhead* mínimo de memória e, ainda; 7. Oferecer taxas de saída estáveis no envio de metadados dinâmicos.

A maior parte dos algoritmos de junção busca a otimização através da diminuição de acessos a disco. Porém, o cenário de execução de *data streams* apresenta complicadores que influem no desempenho geral da junção, particularmente, as limitações de recursos e as características físicas da rede de comunicação. O MJoin otimiza a execução lógica da consulta através da exploração tanto de metadados estatísticos quanto dinâmicos. Utilizando informações dos metadados é possível reduzir o volume de dados materializados durante o processamento da junção, através da eliminação dos dados menos necessários. O MJoin explora metadados sobre o esquema, avaliando restrições de integridade, e metadados especificados através de predidados definidos sobre os data streams .

O algoritmo MJoin é dividido em três etapas, com funcionalidades semelhantes àquelas encontradas no XJoin e no MobiJoin, para administrar a alocação dos dados entre disco e memória. Porém, o MJoin diferencia-se pela sua capacidade de avaliar a semântica dos dados, através da adoção de duas diferentes abordagens para a eliminação das tuplas: exclusão imediata dos dados desnecessários, logo que estes sejam identificados e; eliminação retardada, na qual as tuplas são excluídas em um momento oportuno. Assim, predicados conhecidos poderiam ser utilizados para eliminação imediata dos dados e, durante a segunda etapa do algoritmo, quando o fornecimento de tuplas deve estar suspenso, poderia ser realizada a eliminação retardada dos dados.

3.3 Análise comparativa dos algoritmos de junções baseados em hash

Cada um dos algoritmos abordados dá maior ênfase ao fator que considera mais crítico ao desempenho, considerando as características do cenário de sua execução. Porém, dar prioridade a um fator geralmente implica em conseqüências negativas em algum outro ponto do algoritmo. Como forma de dar subsídios à elaboração de uma nova proposta é apresentada na Tabela 2 : Análise dos algoritmos *hash* abordadosuma análise do diferencial, pontos fortes e fracos de cada um dos algoritmos abordados.

Tabela 1 mostra um quadro resumo da evolução das estratégias *hash* de junção, considerando os algoritmos abordados no item anterior. O quadro procura tornar mais clara a relação entre os algoritmos citados, mostrando que o *Hybrid Hash Join*

[7] Metadados: Definem como o dado é armazenado, incluindo a definição de seu esquema, restrições de integridade e informações de índices.

foi proposto estendendo as características do *Simple Hash Join* e do *Grace Hash Join*; O Xjoin estendeu o *Hybrid Hash Join* e o *Symmetric hash Join e* foi base para as propostas: MobiJoin, MJoin e *Hash-Merge Join*; este último também incorporou algumas propriedades do algoritmo *Progressive Hash Join*.

Tabela 2 : Análise dos algoritmos *hash* abordados.

	Diferencial	Pontos fortes	Pontos fracos
Simple hash join	Um dos pioneiros no uso de algoritmos *hash* para execução de operadores de junção.	Melhores desempenhos com grandes memórias.	Ruim para ambientes com pouca memória, devido à necessidade de aplicações recursivas de funções hash.
			Nenhum resultado está disponível até que todos os dados sejam agrupados em partições.
Grace hash join	Utilização da estratégia *avoidance* de particionamento de dados.	Melhores desempenhos com pequenas memórias.	Ruim para memórias grandes, devido à necessidade de combinar recursivamente as partições.
			Nenhum resultado está disponível até que todos os dados sejam agrupados em partições.
Hybrid hash join	Combina o *hashing* em memória com o *overflow resolution*.	Flexível quanto ao limite da memória, embora ofereça melhores desempenhos quando a memória é pouco menor que a relação de construção.	O espaço requerido em memória para ser usado como *buffer* de cada partição descarregada em disco representa um *overhead*.
			Memórias menores resultarão em mais partições em disco, levando a perdas de desempenho.
			Nenhum resultado está disponível até que todos os dados sejam agrupados em partições.
Symmetric hash Join	Foi um dos pioneiros na classe de algoritmos adaptativos. Utiliza pipeline e execução simétrica da junção.	A junção é feita em memória, permitindo bons desempenhos tanto no fornecimento parcial quanto total do resultado.	As duas relações devem poder ser completamente abrigadas em memória.
			Não prevê situações de overflow de memória.
		Não requer que todos os dados sejam recepcionados para começar a fornecer resultados.	Havendo atrasos no fornecimento dos dados, o processamento da junção é suspenso.
MobiJoin	O mecanismo utilizado para evitar a duplicação dos dados requer menos memória que o XJoin, caracterizando-o como uma alternativa mais viável a ambientes com limitação de recursos.	Rápido retorno inicial dos dados.	Comparação de um mesmo par de tuplas inúmeras vezes, sendo que na terceira etapa, todas as tuplas são reprocessadas, embora poucos resultados sejam produzidos, o que influi negativamente no tempo total da junção.
		A junção continua sendo processada, mesmo quando os dados atrasarem.	
		Requer menos memória para controlar a não duplicação de tuplas.	
Hash-Merge Join	Combina o rápido retorno inicial dos dados do XJoin com o rápido retorno do resultado final oferecido pelo (PMJ) *Progressive Merge Join*.	Rápido retorno inicial dos dados.	É difícil dimensionar o tamanho ideal de partição adequado tanto à fase de *hash* quanto à de *merge*.
		Estratégia *adaptive flushing* promove melhor balanceamento entre os dados das duas fontes, mantidos em memória.	Exige um passo extra, necessário à ordenação dos dados antes de eles irem para disco.
Mjoin	Primeira estratégia de junção a explorar tanto metadados estatísticos quanto dinâmicos para otimizar a execução lógica de um algoritmo de junção.	Rápido retorno inicial dos dados.	Implica em passos extras, necessários à avaliação dos metadados e à eliminação de dados desnecessários.
		A análise de metadados permite a eliminação de dados descessários. Assim, mais tuplas podem ser combinadas em disco, mais resultados podem ser produzidos rapidamente e menos acessos a disco são necessários.	Quanto mais metadados forem considerados, mais avaliações sobre os dados devem ser feitas, implicando em custos extras.
		A eliminação de tuplas desnecessárias pode ser feita mesmo quando o fornecimento de dados é interrompido.	

4. Conclusão

Neste artigo, investigou-se a evolução dos algoritmos de junção, baseados em hash, em função de novos ambientes para o processamento de consultas sobre bancos de dados. Foram apresentadas as características que foram introduzidas nos processadores de consultas para que processem com eficiência consultas em ambientes como o de redes de sensores sem fio. Neste contexto, foram mostradas propriedades dos processadores de consultas convencionais, adaptativos e distribuídos. Acompanhando esta evolução, foram apresentados os principais algoritmos de junção baseados em técnicas de *hash.*

Portanto, algoritmos para implementar o operador algébrico de junção adequados para arquiteturas convencionais de banco de dados como o *Simple Hash Join*, o *Grace Hash Join e o Hybrid Hash Join* foram descritos e analisados. Estes algoritmos foram básicos para inúmeras propostas dentro da classe de algoritmos adaptativos como o *Symmetric Hash Join*, o MobiJoin, o *Hash-Merge Join* e o MJoin. As características, pontos fortes e pontos fracos destes algoritmos foram analisados como forma de oferecer subsídios à elaboração de novas propostas, capazes de atender, especialmente, às necessidades de aplicações que trabalham com *data streams*, como é o caso daquelas desenvolvidas para redes de sensores.

Referências

BABU, S.; BIZARRO, P. *Adaptive query processing in the looking glass.* Stanford: Stanford University, 2005.

BABU, S. et al. *Models and issues in data stream systems.* Madison: ACM PODS, 2002.

BOUGANIM, L. et al. A dynamic query processing architecture for data integration systems. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2000. 1 CD-ROM.

BRAYNER, A.; AGUIAR, J. Sharing mobile databases in dynamically configurable environments. In: CONFERENCE ON ADVANCED INFORMATION SYSTEMS ENGINEERING (CAISE03), 15., 2003,Velden. *Proceedings* ... Velden, 2003. 1 CD-ROM.

BRAYNER, A.; PAULINO, T. *Pré-cálculo de junções para o processamento de consultas em ambientes com recursos computacionais limitados.* Fortaleza: Universidade Federal do Ceará, 2004.

BRAYNER, A.; VASCONCELOS, E. MobiJoin: um operador de junção para bancos de dados móveis. In: WORKSHOP DE COMUNICAÇÃO SEM FIO E COMPUTAÇÃO MÓVEL, 6., 2004, Fortaleza. *Anais*... Fortaleza, 2004. p. 153-160.

DAS, A.; GEHRKE, J.; RIEDEWALD, M. *Approximate join processing over data streams.* San Diego: ACM SIGMOD, 2003.

DITTRICH, J. et al. Progressive merge join: a generic and non-blocking sortbased join algorithm. In: THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 2002, Hong Kong. *Proceedings*... Hong Kong, 2002. p. 299-310

GOLAB, L. *Querying sliding windows over on-line data streams.* Waterloo: University of Waterloo, 2003.

GOLAB, L.; ÖZSU, M. T. Issues in data stream management. *ACM SIGMOD,* Waterloo, v 32, n. 2, p. 364-375, Jun. 2003.

GRAEFE, G. Dynamic query evaluation plans: some course corrections? *Bulletin of the technical Committee on Data Engineering IEEE Computer Society*, 2002. 1 CD-ROM.

HAAS, P. J.; HELLERSTEIN, J. M. Ripple joins for online aggregation. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 1999, Philadephia. *Proceedings*... Philadephia: ACM SIGMOD, 1999. p. 287-298.

KOUDAS, N.; SRIVASTAVA, D. *Data streams query processing.* New York: AT&T Labs-Research, 2003.

MOKBEL, M. F.; LU, M.; AREF, W. G. Hash-merge join: a non-blocking join algorithm for producing fast and early join results. In: INTERNATIONAL CONFERENCE OF DATA ENGINEERING, 2004, Boston. *Proceedings*... Boston, 2004. p. 251-263.

PATEL, J. M.; CAREY, M. J.; VERNON, M. K. Accurate modeling of the hybrid hash join algorithm. In: ACM SIGMETRICS - CONFERENCE ON MEASUREMENT AND MODELING OF COMPUTER SYSTEMS, 1994, Nashville. *Proceedings*... Nashville, 1994. p. 56-66, 1994.

RUNDENSTEINER, E.; DING, L. *Evaluating window joins over punctuated streams.* Washington: In: CONFERENCE ON INFORMATION AND KNOWLEDGE MANAGEMENT, 2004, Washington. *Proceedings*... Washington, 2004.

RUNDENSTEINER, E.; HEINEMAN, G. T.; DING, L. Mjoin: a metadata-aware stream join operator. INTERNATIONAL WORKSHOP ON DISTRIBUTED EVENT-BASED SYSTEMS, 2., 2003, San Diego. *Proceedings...* San Diego: ACM SIGMOD, 2003. p. 1-8

SCHNEIDER, D. A. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 1989, Portland. *Proceedings...* Portland: ACM SIGMOD, 1989. p. 110-121.

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. *Data base system concepts.* 3. ed. New Jersey: McGraw-Hill, 1997.

URHAN, T.; FRANKLIN, M. Xjoin: a reactively scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, Washington, p. 27-33, 2000.

URHAN, T.; FRANKLIN, M. *Xjoin:* getting fast answers from slow and burst networks. Maryland: University of Maryland, 1999. (Technical Report CS-TR-3994).

URHAN, T.; FRANKLIN, M.; AMSALEG, L. Cost-based query scrambling for initial delays. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 1998, Seattle. *Proceedings...* Seattle: ACM SIGMOD, 1998. p. 130-141.

WILSCHUT, A. N.; APERS, P. M.G. Dataflow query execution in a parallel main-memory environment In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED INFORMATION SYSTEMS, 1 / INTERNATIONAL WORKSHOP ON PARALLEL AND DISTRIBUTED INFORMATION SYSTEMS, 1., 1991, New York. *Proceedings...* New York ,1991. p. 68-77.

ZHU, Y.; RUNDENSTEINER, E.; HEINMAN, G. Dynamic plan migration for continuous queries over data streams. In: EXTENDING DATABASE TECHNOLOGY CONFERENCE, 2004, Paris. *Proceedings...* Paris: ACM SIGMOD, 2004. p. 587-604.

SOBRE OS AUTORES

Ângelo Brayner
Doutor em Ciência da Computação pela Universidade de Kaiserslautern, Alemanha, em 1999. Consultor ad-hoc CNPq e FUNCAP, atua também como avaliador de cursos de graduação pelo INEP/MEC. É professor titular da Universidade de Fortaleza nos cursos de graduação em Informática e Mestrado em Informática Aplicada. Já orientou dez dissertações de mestrado (nos programas da UNIFOR e UFC) e orienta atualmente alunos de doutorado pelos programas da PUC-Rio e UFRN.

Aretusa Maria Almeida Lopes
Bacharel em Informática pela Universidade de Fortaleza em 1999. Bolsista ITI-1A ProTem/CNPq de 1997 a 1998.Especialista em tecnologias de redes e Internet pela Universidade de Fortaleza em 2003. Aluna do Mestrado em Informática Aplicada pela Universidade de Fortaleza desde 2004.