piFP-growth: Um Algoritmo Paralelo para Geração Incremental de Regras de Associação

Tiago Adriano de Knegt López de Prado
Universidade Federal de Minas Gerais
tiago@dcc.ufmg.br

Gustavo Menezes Siqueira
Universidade Federal de Minas Gerais
gms@dcc.ufmg.br

Wagner Meira Júnior
Universidade Federal de Minas Gerais
meira@dcc.ufmg.br

Márcio Luiz Bunte de Carvalho
Universidade Federal de Minas Gerais
mlbc@dcc.ufmg.br

Resumo

Mineração de regras de associação tem sido amplamente utilizada em sistemas para suporte à decisão e para personalização do atendimento a clientes. Muitos destes sistemas são caracterizados por uma adição constante de dados às bases, o que inevitavelmente determina reavaliações constantes dos resultados das minerações. Nestes casos, utilizar algoritmos tradicionais, que mineram as bases inteiras a cada execução pode ser dispendioso e até impraticável.

Neste artigo apresentamos o piFP-growth, um algoritmo que, a partir da árvore de padrões freqüentes armazenada em minerações anteriores e dados adicionados posteriormente, calcula as regras de associação do conjunto resultante em paralelo e de forma escalável. Este algoritmo é uma paralelização do algoritmo iFP-growth e se distingue não apenas pelo seu caráter incremental, mas também pela otimização do uso dos recursos computacionais.

O algoritmo foi implementado em linguagem C, utilizando a biblioteca POSIX Threads, e avaliado para bases de dados reais e sintéticas, demonstrando os ganhos da sua utilização e a manutenção das propriedades do algoritmo não-incremental sequencial.

1. Introdução

Mineração de dados é o processo de aquisição de informações implícitas em uma base de dados, como os agrupamentos, os conjuntos freqüentes e as regras de associação. Estas últimas são freqüentemente utilizadas para determinar correlações implícitas em bases de transações permitindo, por exemplo, entender melhor o comportamento de consumidores que utilizam *sites* de comércio eletrônico.

Formalmente, uma regra de associação é uma regra da forma $X \implies Y$, a qual é usualmente caracterizada por duas métricas, confiança e suporte. Uma regra $X \implies Y$ tem confiança c se $c\%$ dos conjuntos de entrada que contêm X também contêm Y, e suporte s se $s\%$ contêm $X \cup Y$, onde $X \cap Y = \emptyset$, $s > minsup$, $c > minconf$, onde $minsup$ e $minconf$ são limiares providos pelo usuário. O processo de determinação de regras de associação inerentes a uma base de dados pode ser dividido em duas etapas: determinar os conjuntos freqüentes, isto é, os conjuntos de ítens que satisfaçam o critério de suporte, e a partir deles, as regras de associação. Este artigo se concentra no primeiro problema, ou seja, a determinação dos conjuntos freqüentes, que é a etapa computacionalmente mais intensiva. Vários algoritmos têm sido propostos na literatura para determinar conjuntos freqüentes, tais como o *Apriori* [AS94], o *ECLAT* [Zak00] e o *FP-growth* [HPY00]. Esse último algoritmo se diferencia dos anteriores por adotar uma estratégia diferente para representar os conjuntos freqüentes. Enquanto os demais algoritmos estimam os conjuntos potencialmente freqüentes (chamados conjuntos candidatos) para posterior verificação de pertinência na base de dados, o *FP-growth* não demanda tais estimativas, representando os conjuntos freqüentes de forma condensada e sistemática. Considerando que o número de conjuntos candidatos pode crescer exponencialmente com o número de ítens distintos na base de dados, a redução de custo resultante com a utilização do *FP-growth* pode ser significativa.

Apesar de serem bastante eficientes, estes algoritmos não são adequados para ambientes onde transações são adicionadas à base de dados continuamente e os conjuntos freqüentes devem ser atualizados periodicamente, a exemplo do que acontece em servidores de comércio eletrônico que empregam técnicas de personalização [JMCG02]. Para estes casos, seria desejável o emprego de algoritmos que atuassem de forma incremental, ou seja, algoritmos que ex-

plorassem o conhecimento previamente adquirido de forma a evitar a repetição de processamento sobre as mesmas transações, como é o caso quando utilizamos os algoritmos tradicionais mencionados. Já há algumas propostas de algoritmos incrementais na literatura [TBAR97, VPJdC01, VJdC+02] que permitiram ganhos significativos sobre os seus pares não incrementais, mas todas essas propostas ainda são afetadas pelo problema da geração de conjuntos candidatos. Em um trabalho anterior [SdKLdPJdC02] descrevemos a implementação do *iFP-Growth*, um algoritmo incremental baseado em *FP-Trees*.

Mesmo utilizando técnicas incrementais, o custo computacional ainda é grande. Uma solução é a paralelização de algoritmos de mineração de dados [ZOPL96]. Há uma proposta atraente de algoritmo paralelo que utiliza a *FP-tree*, o *MLFPT* [ZEHL01], que particiona eficientemente essa estrutura entre várias *threads*.

Neste artigo apresentamos um novo algoritmo paralelo e incremental para a determinação de conjuntos freqüentes que é baseado nos algoritmos *iFP-growth* e *MLFPT*. O artigo está dividido em cinco seções, sendo esta a introdução. A Seção 2 apresenta o algoritmo *FP-growth* e a sua estrutura fundamental, a árvore de padrões freqüentes. Uma descrição detalhada do algoritmo incremental *iFP-growth* é apresentada na Seção 3 e o algoritmo paralelo é descrito na Seção 4. Os resultados da utilização do algoritmo proposto usando bases de dados reais são apresentados na Seção 5 e as conclusões e os trabalhos futuros são discutidos na Seção 6.

2. O algoritmo FP-growth

Esta seção descreve brevemente o algoritmo FP-growth, que é o ponto de partida do nosso algoritmo incremental. Inicialmente descrevemos a estrutura fundamental do algoritmo, que é a árvore de padrões freqüentes, seguida do algoritmo propriamente dito.

Uma *FP-tree*, ou árvore de padrões freqüentes (APF), é uma árvore de prefixos estendida que representa de forma compacta um conjunto de transações, armazenando informação suficiente para a determinação dos conjuntos freqüentes. Cada nó da árvore é associado a um item e também armazena uma freqüência de ocorrência do item nas transações representadas pelo nó. Uma propriedade interessante de uma árvore de prefixos é que todos os caminhos a partir da raiz obedecem uma ordenação total pré-estabelecida entre os ítens. Por exemplo, seja uma base de dados composta por quatro ítens distintos: A, B, C e D, ordenados alfabeticamente. A árvore de prefixos associada à base de dados possui caminhos como ABC e ACD, mas caminhos como DAC e CB são inválidos.

Cada transação da base de dados é um caminho na APF a partir da raiz. A profundidade do caminho é igual à freqüência da transação, ou seja, nem todos os caminhos atingem uma folha. Por exemplo, considere a base de dados composta por três transações {A, B, C}, {C, B, D} e {D, B}. Cada uma das transações é ordenada de acordo com o critério pré-estabelecido, no caso ordenação alfabética. Cada item também possui um contador que é utilizado para controlar a freqüência global do item. Definida a ordem entre os ítens, podemos construir a APF associada percorrendo a árvore a partir da raiz, e, para cada item, verificamos a existência de um nó que contenha o item como descendente imediato. Se este for o caso, o contador do nó é incrementado, senão, um novo nó é criado. Este processo é ilustrado para a base de dados exemplo na Figura 1 e descrito a seguir.

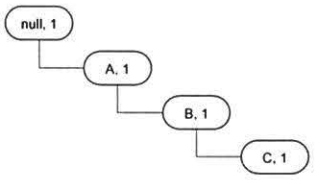
Inicialmente, uma árvore possui apenas um nó raiz, que não possui item associado (*null*) e contador 0. Para inserir a transação $\{A, B, C\}$, incrementamos os contadores de A, B e C ($A = 1, B = 1, C = 1$) e ordenamos a transação ($\{A, B, C\}$) de acordo com o critério. A seguir, incrementamos o contador da raiz e criamos o caminho ABC, resultando na árvore da Figura 1(a). Para a segunda transação, incrementamos as freqüências de C, B e D ($C = 2, B = 2, D = 1$) e a ordenamos ($\{B, C, D\}$). A seguir, incrementamos o contador da raiz e criamos o caminho BCD, resultando na árvore da Figura 1(b). Para a terceira transação, incrementamos os contadores de D e B ($B = 3, D = 2$) e ordenamos a transação ($\{B, D\}$). A seguir, incrementamos o contador da raiz e o contador do nó B e criamos o nó D, resultando na árvore da Figura 1(c).

É interessante notar que é possivel reconstruir a base de dados original a partir da respectiva APF, utilizando sucessivas buscas em profundidade na árvore, onde cada busca decrementa o valor do contador dos nós no caminho. Uma outra propriedade interessante é que a freqüência dos nós de um dado caminho é monotonicamente decrescente, ou seja, a freqüência de um nó filho é sempre menor ou igual à freqüência do seu ancestral imediato.

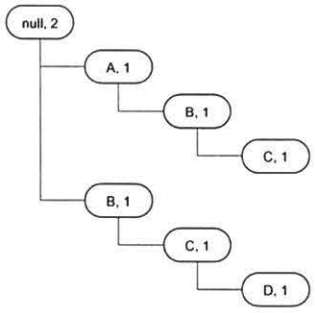
No caso da APF do *FP-growth*, o critério de ordenação não é alfabético, mas popularidade decrescente, ou seja, o primeiro item é o mais popular na base de dados e o último item na ordenação é o menos popular. Este critério de ordenação permite uma representação compacta do conjunto de transações, uma vez que os ítens mais freqüentes se localizam próximos à raiz, e a árvore é bastante esparsa.

O algoritmo *FP-growth* pode ser então dividido em duas fases. Na primeira fase, a freqüência de cada item na base de dados é determinada. Esta função de popularidade é utilizada na segunda fase, quando a APF correspondente é construída, como ilustrado. Uma vez construída a APF, as regras podem ser geradas usando um mecanismo denominado árvores condicionais, que não é afetado pelo caráter incremental das bases e não será discutido neste artigo. A dificuldade para a implementação de um algoritmo incremental é que a ordenação total dos ítens com relação à sua

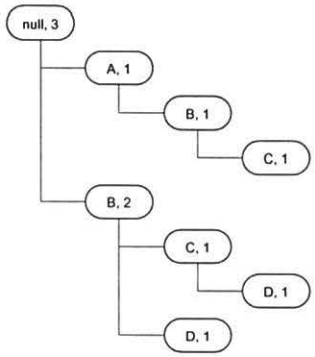
freqüência pode mudar ao longo da execução, reduzindo a eficiência da APF. Esta dificuldade pode ser visualizada no exemplo apresentado, onde as freqüências de A, B, C e D são, respectivamente, 1, 3, 2 e 2. Neste caso, o critério de ordenação deveria ser B, C e D e A, o que exige mudanças na APF, como discutido na próxima seção.



(a) Adição de $\{A, B, C\}$



(b) Adição de $\{C, B, D\}$



(c) Adição de $\{D, B\}$

Figura 1. Exemplo de construção de árvore de padrões freqüentes

3. O algoritmo iFP-growth

Nesta seção apresentamos o nosso algoritmo incremental, *iFP-growth*. Como mencionado, este algoritmo estende o *FP-growth*, resolvendo o problema de mudanças no critério de ordenação ao longo do tempo.

Na prática, o *iFP-growth* se inicia com a construção da APF, que é realizada de forma idêntica ao *FP-growth* tradicional. Uma vez construída a árvore, o algoritmo avalia se o critério de ordenação é válido, ou seja, se os ítens estão ordenados de acordo com a sua freqüência global decrescente. Caso o critério não seja mais válido, é necessário determinar quais ramos da árvore devem ser alterados, de forma a satisfazer o novo critério de ordenação. Essa verificação é realizada através de uma busca em profundidade com uma pequena modificação que permite verificar se um nó e seu ancestral satisfazem o critério de ordenação. Caso isso não aconteça, é necessário promover alterações na APF de forma a restaurar as suas propriedades.

Essas alterações são realizadas através de três operações: permutação, fatoração e união.

A execução da operação de permutação troca as posições de dois nós, ou seja, o nó que era pai passa a ser filho e viceversa. Esta operação é a base da restauração, mas possui condições estritas de aplicação. Mais especificamente, dois nós só podem ser permutados se os contadores de ambos forem iguais, de tal forma que a propriedade de freqüência decrescente não seja violada por tal troca. As duas outras operações se destinam a viabilizar uma permutação ou corrigir violações resultantes da mesma, como descrito a seguir.

A operação de fatoração, como o próprio nome diz, divide as informações associadas a um nó (i.e., contador e descendentes) entre dois nós de forma consistente, ou seja, os novos nós não podem ter contadores menores que os seus descendentes. A operação de fatoração pode ser aplicada a qualquer nó da árvore, sendo utilizada no nosso algoritmo para permitir a execução da permutação.

A operação de união, por outro lado, agrega as informações representadas por dois nós em um único nó, que vai ter como contador a soma dos contadores dos nós agregados. A união é normalmente aplicada para restaurar a propriedade de que um nó não pode ter dois filhos associados ao mesmo item. Essa operação é aplicada após uma permutação que viole a referida propriedade.

Um aspecto fundamental dessas operações é que a sua aplicação não deve resultar em perda de informação, ou seja, transações que antes eram representadas na APF, não o são após execução da operação. A operação de permutação mantém a informação contida na APF pois apenas troca dois nós de posição, sem que seus contadores ou identificadores sejam alterados. Como os algoritmos de mineração de dados não são sensíveis à ordenação dos ítens nas transações, podemos concluir que a permutação não provoca perda de informação. Uma outra evidência da capacidade de manutenção de informação é que a operação inversa de uma permutação é a própria, ou seja, podemos realizar um

número indeterminado de permutações que as transações representadas na APF não se alteram. O mesmo raciocínio se aplica às demais operações, que não causam qualquer alteração nos contadores associados a um item e aos ramos que deles descendem.

Com base nestas operações, podemos definir o procedimento de "ascensão", que é invocado pelo *iFP-growth* sempre que a propriedade da APF é violada, ou seja, um item B menos freqüente no contexto da base de dados se torna mais freqüente que o seu pai que contem item A. Desta forma, uma ascensão "promove" um nó para a posição de seu pai, de forma a refletir uma inversão de popularidade entre os ítens associados aos nós. Há dois critérios que determinam a natureza da ascensão do nó contendo B sobre o nó contendo A:

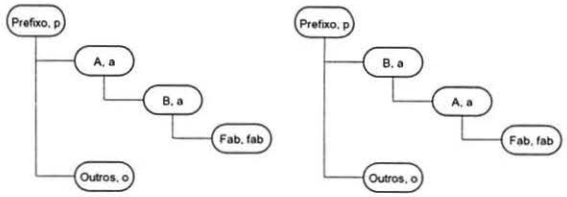
Relação entre os contadores: Se os contadores são idênticos, então os nós podem ser simplesmente permutados, pois eles ocorrem nas mesmas transações. Se a (o contador de A) é maior que b (o contador de B) em um dado caminho, então há caminhos que contém A mas não contém B. Neste caso, a execução de uma fatoração antes separa os descendentes de A que não contêm B, viabilizando a permutação.

Possibilidade de junção: Este critério verifica se a ascensão de B vai violar a propriedade da árvore de prefixos de que para cada nó há no máximo N filhos, onde N é o número de ítens distintos. Assim, se houver um nó associado a B que seja "irmão" de A, então a ascensão deve promover a união dos dois nós.

Como cada um dos critérios pode ser satisfeito ou não, temos quatro tipos de procedimentos de ascensão possíveis. Em todos os casos, a ascensão é executada em $O(\log n)$. A seguir descrevemos a execução de cada tipo de operação, invocada pela inversão de popularidade entre os nós associados aos ítens A e B, como mencionado.

3.1. Troca simples

A troca simples é aplicada quando o nó a ser ascendido tem o contador de mesmo valor que seu pai e não há possibilidade de junção. Neste caso, apenas uma permutação é realizada.

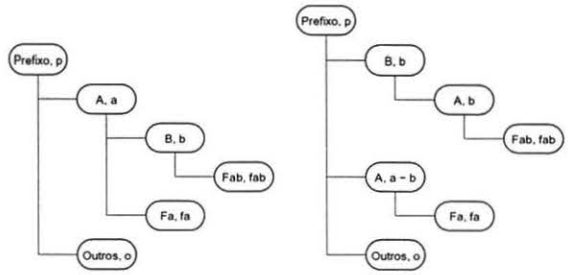


(a) original (b) atual

Figura 2. Troca simples

3.2. Divisão

A divisão é aplicada quando o nó a ser descendido (A) tem contador (a) maior do que o do nó ascendente (B). Neste caso, a operação de fatoração é executada, criando um novo nó, que vai representar as transações nas quais A aparece mas B não, cujo contador é igual a $a-b$ e que recebe os nós filhos de A, exceto B. A seguir, a permutação é executada, tornando B pai de um nó associado a A com contador igual a b.

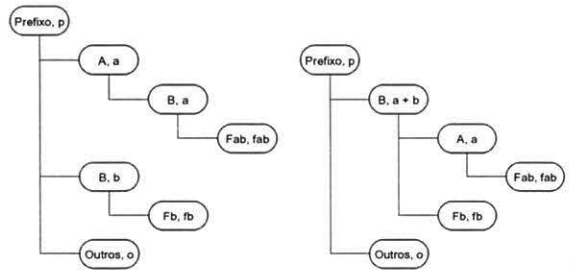


(a) original (b) atual

Figura 3. Divisão

3.3. Junção

A junção é aplicada quando os contadores de A e B são iguais, mas já há um nó no nível de A associado a B. Neste caso, inicialmente é realizada a permutação seguida de união entre os nós associados a B, conforme ilustrado nas figuras a seguir.

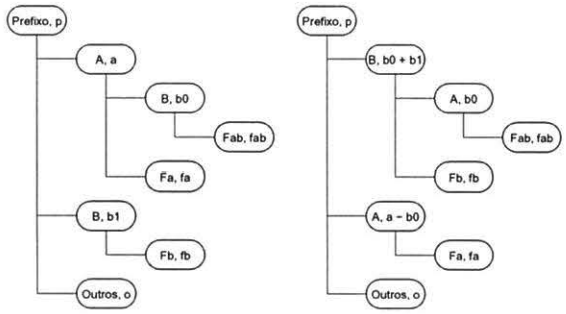


(a) original (b) atual

Figura 4. Junção

3.4. Divisão com junção

A divisão com junção é aplicada quando os contadores de A e B são diferentes (ou seja, $a > b0$), e há um nó associado a B no nível de A. Neste caso, é aplicada uma fatoração sobre o nó A, seguida da permutação e da união de nós associados a B, como ilustrado na figura a seguir.



(a) original (b) atual

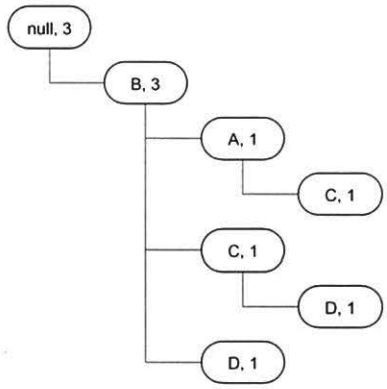
Figura 5. Divisão com junção

3.5. Exemplo de restauração

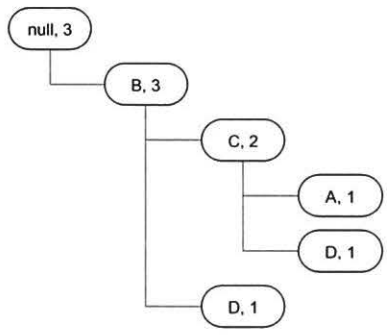
Para demonstrar o funcionamento do algoritmo *iFP-growth*, vamos apresentar a sua aplicação para restaurar a árvore construída na Seção 2.

A restauração é baseada em procedimentos de ascenção para restaurar a propriedade de freqüência decrescente, possivelmente violada após a adição conjuntos à APF. Todos nós que representam cada um dos ítens que tornaram-se mais populares que algum outro desde a última restauração são visitados, em ordem decrescente de popularidade do item. Executamos operações de ascenção em conformidade com os critérios de diferença dos contadores e possibilidade de junção para cada nó que viola a propriedade, até que ela seja localmente restaurada. Ao final do procedimento, a propriedade é globalmente restaurada.

Como mencionado, as freqüências de A, B, C e D são, respectivamente, 1, 3, 2 e 2, determinando B, C, D e A como critério de ordenação. A partir da raiz, o algoritmo visita B, onde não há violação e depois A, que possui B como filho, o que viola a propriedade da APF. Executamos uma junção e a propriedade é localmente restaurada (Figura 6(a)). A seguir, o algoritmo visita o ramo BC e depois o ramo BAC, onde há uma violação entre A e C. O algoritmo executa novamente uma junção e a propriedade é localmente restaurada (Figura 6(b)). Finalmente, visitamos o ramo BCD que satisfaz o critério de ordenação. Ao final do procedimento, a propriedade de freqüência decrescente é restaurada.



(a) adição de $\{a, b, c\}$



(b) adição de $\{c, b, d\}$

Figura 6. Exemplo de restauração de APF

4. O algoritmo piFP-growth

Nesta seção vamos apresentar a nossa estratégia de paralelização do algoritmo iFP-growth, a qual é baseada em paralelismo de dados. Três tipos de dados são distribuídos entre as *threads* que executam o algoritmo paralelo. O primeiro tipo de dados distribuído são as próprias transações que servem como entrada para a mineração. Essas transações são distribuídas entre as *threads* usando uma estratégia *round-robin*, garantindo um bom balanceamento de carga de trabalho. Já o segundo tipo de dados afetado pela paralelização é a APF, que deixa de ser única, sendo responsabilidade de cada *thread* criar a sua APF, a qual é construída a partir das transações assinaladas àquela *thread*. O terceiro tipo de dados são os ítens que compõem as transações, os quais são divididos igualmente entre as *threads* para fins de contagem e atualização de freqüência global, como descrito a seguir.

Em termos de implementação, as APF são organizadas como estruturas que contêm uma árvore n-ária, representando as transações, e por duas tabelas *hash*, que relacionam os ítens lidos dos arquivos de entrada a identificadores internos ao programa e vice-versa.

Desta forma, cada *thread* executa, para fins de mineração incremental de dados, a seguinte seqüência de passos:

1. Lê a transação a ser processada.
2. Verifica se algum item está ocorrendo pela primeira vez em uma transação. Se este for o caso, inicializa sincronamente as estruturas globais de controle de freqüência dos ítens.
3. Atualiza a APF de acordo com a transação.
4. Sincroniza esperando todas as *threads* terminarem a atualização das APF.
5. Conta a freqüência de ocorrência da sua partição de ítens em todas as APF.
6. Aguarda que todos os processadores realizem a contagem de freqüência de sua partição.
7. Atualiza a freqüência global de cada item em sua partição.
8. Reorganiza a sua APF de acordo com a freqüência global dos ítens.

Note que, para fins de maior eficiência, em geral cada *thread* lê várias transações, uma vez que o restante dos passos independe do número de transações a serem processadas.

A implementação corrente do algoritmo piFP-growth armazena duas tabelas, a primeira para atribuir identificadores globais para cada cadeia de caracteres que representa um item nos arquivos de entrada e a segunda que relaciona esses identificadores à freqüência e à cadeia de caracteres correspondente. As *threads* do algoritmo paralelo compartilham essas tabelas para que os ítens sejam representados consistentemente, requisito necessário para a determinação dos conjuntos freqüentes. As tabelas globais devem ser acessadas de forma síncrona durante a leitura dos arquivos porque novos ítens podem ser escritos no instante em que outros são lidos ou escritos, causando uma condição de corrida. Uma vez que tenham sido armazenados na tabela global, os identificadores globais não são alterados, o que permite a criação de tabelas locais que funcionam como *caches* para diminuir o impacto da sincronização.

Cada *thread* lê uma transação de um arquivo de entrada e identifica seus conjuntos de ítens, procurando as entradas referentes às cadeias de caracteres primeiro na tabela local e depois na tabela global, criando novas entradas se necessário. O conjunto é ordenado segundo a freqüência global descendente e segundo o identificador de forma crescente, as freqüências locais são atualizadas e o conjunto é adicionado à APF local.

Quando for necessário restaurar as APFs, antes de determinar os conjuntos freqüentes ou simplesmente para procurar reduzir a ocupação da memória, as adições em curso devem ser finalizadas e somente então a freqüência global deve ser atualizada. Cada *thread* soma as freqüências locais de uma partição do conjunto de ítens e atualiza as freqüências globais concorrentemente. Ao final da última soma, as APFs locais são restauradas independentemente.

A determinação dos conjuntos freqüentes é feita de forma idêntica a [ZEHL01] e por isso não será tratada neste artigo.

5. Avaliação experimental

Nesta seção avaliamos uma implementação em C, utilizando POSIX *threads*, do algoritmo *piFP-growth*. Esta implementação foi avaliada para duas bases de dados. A primeira base de dados é um *log* de acesso de um servidor HTTP de um provedor de conteúdo, com 432601 transações com tamanho médio de 1,8. A segunda base é a coleção CFC [SWWT91], de documentos publicados de 1974 a 1979 discutindo aspectos da fibrose cística, que contem 1239 transações com tamanho médio de 12,3.

Para avaliar os ganhos do *piFP-growth*, medimos para todas as bases de dados o número de nós utilizados, o número de operações executadas durante a reestruturação da árvore e o tempo de execução do algoritmo *iFP-growth* e para o algoritmo paralelizado com duas, três e quatro *threads* simultâneas.

Os testes foram realizados em uma partição de quatro processadores e 4GB de memória de uma máquina Sun Starfire ENT10000, cujas principais características são:

N	t(s)	Operações				Nós
		TS	J	D	DJ	
1	5375.61	30821	2172	2823	4692	47276
2	2645.30	32634	2184	2880	4783	51824
3	1737.07	33589	2173	2944	4781	54410
4	1334.32	34142	2181	2972	4803	56194

Tabela 1. Base: provedor de conteúdo

N	t(s)	Operações				Nós
		TS	J	D	DJ	
1	130.45	9385	116	555	297	8958
2	66.03	9557	110	575	252	9208
3	42.98	9707	98	582	243	9394
4	32.10	9742	99	631	214	9481

Tabela 2. Base: CFC

- Processadores ULTRASPARC 250MHz
- 1MB cache por processador
- 8GB mémoria RAM, em módulos de 512MB
- SMP escalável permitindo reconfiguração dinâmica, com partição flexível do sistema
- Largura de banda de I/O de 6,4 GB/s (pico)
- System Boards com 4 processadores e 1GB de memória RAM

Os resultados estão sumarizados nas tabelas 1 e 2. Na primeira coluna (N) está indicado quantas *threads* foram utilizadas. O tempo total de execução (t), em segundos, é indicado pelo segunda coluna. Os quatro valores seguintes são o número de operações de troca simples (TS), junção (J), divisão (D) e divisão com junção (DJ), como definidas na seção 2. A última coluna informa o número de nós utilizados por todas as APFs para representar os ítens das transações da base de dados.

Pode-se observar um crescimento quase linear do paralelismo do algoritmo em função do número de *threads*, devido ao pequeno número de acessos concorrentes à tabela global de identificadores.

O aumento do número total de nós se explica pela divisão das transações entre as APFs, resultando em uma replicação de nós que representam o mesmo item em vários processadores. Essa observação é confirmada pelos dados de ambas as tabelas. O experimento com a base de dados CFC mostra que o número de operações de junção, tanto simples quanto com divisão, diminui com o aumento do número de processadores utilizados. No outro experimento, o número dessas operações não variou muito absolutamente, mas em proporção com o aumento do número de nós, a diminuição é substancial. As operações de troca simples crescem linearmente com o número total de nós nos dois experimentos. O número de operações de divisão varia pouco com o número de *threads* no primeiro conjunto de dados, e no segundo, o acréscimo de operações de divisão por aumento do uso de processadores é próximo ao decréscimo do número de operações de divisão com junção.

6. Conclusão

Neste artigo apresentamos a proposta e implementação de um algoritmo paralelo para a geração incremental de regras de associação, o *piFP-growth*. Este algoritmo se baseia numa estrutura otimizada para armazenar as várias transações sendo mineradas, a árvore de padrões freqüentes, e trata o problema da desatualização associada à consideração de novas transações.

O algoritmo foi implementado em C, utilizando POSIX Threads, e apresentou bons resultados, permitindo uma redução significativa do custo de mineração e uma otimização do uso de recursos de armazenamento. Com a duplicação de uma parte pequena das estruturas de dados conseguimos reduzir os pontos de sincronização, alcançando um crescimento quase linear do paralelismo em função do número de processadores.

Como trabalhos futuros pretendemos avaliar o problema correlato de determinação de conjuntos frequentes fechados que pode ser resolvido de maneira similar. Planejamos criar uma estrutura de dados para representar os conjuntos freqüentes para que, além da APF, também eles possam ser atualizados de forma incremental.

Referências

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, Proc. 20th Int. Conf. Very Large Data Bases, VLDB, pages 487–499. Morgan Kaufmann, 12–15 1994.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. pages 1–12, 2000.

[JMCG02] W. Meira Jr., C. Murta, S. Campos, and D. Guedes. Comércio Eletrônico: Projeto e Desenvolvimento de Sistemas. edições Campus-SBC, Janeiro 2002.

[SdKLdPJdC02] Gustavo Menezes Siqueira, Tiago Adriano de Knegt López de Prado, Wag-

ner Meira Jr., and Márcio Luiz Bunte de Carvalho. ifp-growth: Um algoritmo incremental para determinar regras de associação. In Anais do 17º Simpósio Brasileiro de Banco de Dados, 2002. Aceito para publicação.

[SWWT91] W. M. Shaw, J. B. Wood, R. E. Wood, and H. R. Tibbo. The cystic fibrosis database: Content and research opportunities. In Library and Information Science Research, volume 13), pages 347–366, 1991.

[TBAR97] Shiby Thomas, Sreenath Bodagala, Khaled Alsabti, and Sanjay Ranka. An efficient algorithm for the incremental updation of association rules in large databases. In Knowledge Discovery and Data Mining, pages 263–266, 1997.

[VJdC+02] A. Veloso, W. Meira Jr., M. B. de Carvalho, B. Pôssas, S. Parthasarathy, and M. Zaki. Mining frequent itemsets in evolving databases. In Proc. of the 2nd SIAM Int'l Conf. on Data Mining, Arlington, USA, April 2002.

[VPJdC01] A. Veloso, B. Pôssas, W. Meira Jr., and M. B. de Carvalho. Knowledge management in association rule mining. In Proc. of the 1st IEEE Int'l Workshop on Integrating Data Mining and Knowledge Management, San Jose, USA, november 2001.

[Zak00] Mohammed Javeed Zaki. Scalable algorithms for association mining. Knowledge and Data Engineering, 12(2):372–390, 2000.

[ZEHL01] Osmar R. Zaiane, Mohammad El-Hajj, and Paul Lu. Fast parallel association rule mining without candidacy generation. In ICDM, pages 665–668, 2001.

[ZOPL96] Mohammed Javeed Zaki, Mitsunori Ogihara, Srinivasan Parthasarathy, and Wei Li. Parallel data mining for association rules on shared-memory multiprocessors. Technical Report TR618, 1996.