CONTROLE DE CONCORRÊNCIA EM ÁRVORES-B*

Roberto M. F. Souza[1]
Osvaldo S. F. Carvalho[2]

RESUMO

Existem na literatura vários algoritmos para o acesso concorrente a árvores-B. Estes algoritmos foram classificados em algoritmos conservadores e algoritmos agressivos. Este artigo descreve novos algoritmos para o acesso concorrente a árvores-B que podem ser classificados como híbridos, pois adotam a postura agressiva durante o caminhamento na árvore e a postura conservadora durante a reestruturação da árvore.

ABSTRACT

There are many algorithms for concurrent access in B-trees in the literature. These algorithms are classified as conservative or agressive. In this paper new algorithms for concurrent access in B-trees are presented, which can be classified as hibrid, because they are agressive during tree traversal and conservative during tree reestruturing.

[1] Bacharel em Ciência da Computação pela UFMG em 1987; Mestrando em Ciência da Computação pela UFMG.
Endereço: Rua Orenoco 23/501 - Belo Horizonte - MG - 30310-060
Telefone: (031) 225-8023
e-mail: vado@dcc.ufmg.br

[2] Bacharel em Física pela UFMG em 1974; Doutor de Estado pela Universidade Pierre et Marie Curie (Paris VI) em 1985.
Endereço: Caixa Postal 702 - Belo Horizonte - MG - 30161
e-mail: vado@dcc.ufmg.br

*Este trabalho foi parcialmente financiado com recursos da Leme Informática Ltda, FAPEMIG (proc. téc. 1113/90) e do CNPq (proc. 502353/91-0(NV))

1. INTRODUÇÃO

Existem na literatura vários algoritmos para o acesso concorrente a árvores-B. Estes algoritmos foram classificados por Kwong em algoritmos conservadores e algoritmos agressivos [Kwong 82]. Este artigo descreve novos algoritmos para o acesso concorrente a árvores-B que podem ser classificados como híbridos, pois adotam a postura agressiva durante a fase de caminhamento na árvore, e a postura conservadora durante a fase de reestruturação da árvore.

O sistema LEMIX[1], constituído de um compilador MUMPS integrado a um ambiente interativo de programação, possui um gerenciador de bases de dados baseado em árvores-B [Wagner 73, Wirth 76, Bayer 77, Comer 79]. Este sistema foi originalmente desenvolvido para ambientes monousuários (MS-DOS) e tempos depois foi portado para ambientes multiusuários (UNIX). Um dos principais problemas encontrados neste porte foi o compartilhamento das bases de dados entre os vários processos MUMPS concorrentes.

Para solucionar o problema do compartilhamento das bases de dados do LEMIX, foram propostos algoritmos eficientes para o acesso concorrente a árvores-B, que são descritos neste artigo. São descritos também neste artigo os algoritmos para o acesso concorrente a árvores-B encontrados na literatura.

O restante do texto é organizado da seguinte forma: a seção 2 descreve a estrutura da árvore-B. A seção 3 descreve os algoritmos para o acesso concorrente a árvores-B encontrados na literatura. A seção 4 descreve os algoritmos propostos. E finalmente, a seção 5 apresenta as conclusões obtidas no decorrer do trabalho.

2. A ÁRVORE-B

A árvore-B é uma estrutura de armazenamento bastante simples e eficiente, que vem sendo largamente utilizada em sistemas de gerenciamento de bases de dados.

As principais características estruturais de uma árvore-B são as seguintes [Wagner 73, Wirth 76, Comer 79]:

a) uma árvore-B é uma árvore **n**-ária de ordem **m**, onde **n = 2m + 1**;
b) todo caminho da raiz até os nodos folha tem comprimento **h**, onde **h** é o nível da raiz. Todos os nodos folha estão no nível zero;

[1]LEMIX é marca registrada da Leme Informática.

c) todo nodo, exceto a raiz e os nodos folha, tem no mínimo **m + 1** filhos, onde cada filho é uma sub-árvore;
d) a raiz ou é um nodo folha ou possui no mínimo **2** filhos;
e) todo nodo não folha, ou nodo de índice, tem no máximo **2m + 1** filhos;
f) um nodo de índice com **j + 1** filhos contém **j** chaves (ou separadores).

As operações mais comumente utilizadas para o acesso a árvores-B são as operações de *inserção, remoção* e *pesquisa*, que são descritas em [Wirth 76].

Existem muitas variações de árvores-B, dentre as quais a mais comum é a árvore-B*[2] [Wirth 76, Bayer 77, McCreight 77, Comer 79]. Uma árvore-B* é uma árvore-B onde todos os registros são armazenados nos nodos folha. As principais vantagens da árvore-B* em relação às árvores-B tradicionais são a simplicidade de implementação das operações de inserção e remoção, e a facilidade de ser feita uma recuperação seqüencial dos registros em ordem lexicográfica.

O sistema LEMIX utiliza a estrutura de árvores-B* com algumas modificações. A principal modificação diz respeito ao armazenamento de registros de tamanho variável. Esta importante modificação altera a definição da estrutura da árvore, bem como os algoritmos para a manipulação da mesma. Devido a esta modificação a taxa de ocupação de um nodo passa a ser medida em relação ao número de bytes ocupados, e não mais em relação ao número de registros ocupados.

Outra modificação implementada na árvore-B* do LEMIX diz respeito ao algoritmo de inserção, que utiliza o tratamento de *overflow*. Esta técnica consiste em distribuir o conteúdo de um nodo entre seus irmãos para evitar a criação de mais um nodo na árvore.

Deste ponto em diante o termo árvore-B* modificada será utilizado para denotar uma árvore-B* tal qual a implementada no sistema LEMIX. Sem perda de generalidade, o restante do texto irá tratar apenas dos algoritmos de inserção e pesquisa em árvores-B* e árvores-B* modificadas.

3. ALGORITMOS TRADICIONAIS DE CONCORRÊNCIA

Um determinado processo ao fazer acesso a uma árvore-B* o faz por meio dos algoritmos de pesquisa e inserção. Estes algoritmos no contexto deste trabalho são denominados transações [Eswaram 76, Gray 78, Bayer 80, Kedem 83, Papadimitriou 86, Bernstein 87], que são divididas em duas categorias: transações

[2] Alguns autores [Comer 79] se referem a estas árvores como árvores-B+.

de leitura (pesquisa) e transações de escrita (inserção). Por definição, uma transação ao ser executada sobre uma árvore-B* íntegra a mantém íntegra após o seu término. Contudo, durante e somente durante a execução de uma transação a árvore-B* pode ficar momentaneamente em um estado não íntegro, o que pode provocar problemas quando duas transações forem executadas em paralelo.

Quando vários processos estão fazendo acesso a uma árvore-B* em um mesmo instante, é necessária a utilização de mecanismos que garantam a integridade da mesma. Normalmente estes mecanismos são implementados através de protocolos de *lock*[3]. O objetivo destes protocolos é tornar o sistema *serializável* [Eswaram 76, Gray 78, Bayer 80, Kedem 83, Papadimitriou 86, Bernstein 87]. Um sistema é dito serializável, se para qualquer conjunto de transações executadas paralelamente corresponder uma execução seqüencial em qualquer ordem destas transações.

Antes de prosseguir, algumas definições relacionadas ao acesso a árvores-B* são necessárias:

- um caminho é um conjunto ordenado de nodos com a seguinte disposição: $N_1, N_2, \ldots N_h$, onde N_1 é a raiz da árvore, N_h é um nodo folha e h é a altura da árvore-B*;

- fase de pesquisa é a fase da transação onde são lidos todos os nodos (em ordem, isto é, da raiz para a folha) pertencentes a um determinado caminho com o objetivo de se atingir um determinado nodo folha para posterior pesquisa ou inserção de um item;

- fase de reestruturação é a fase posterior à fase de pesquisa de uma transação de escrita. Nesta fase o caminho percorrido durante a fase de pesquisa é utilizado na ordem inversa (iniciando da folha), quando são feitas as alterações necessárias para a reestruturação da árvore-B* após a inserção de um novo item;

- escopo é o conjunto dos nodos pertencentes ao caminho que são alterados na fase de reestruturação. Se o nodo N_i pertence ao escopo da transação então o nodo N_{i+1} também pertence ao escopo;

---

[3]Nos algoritmos aqui apresentados os objetos onde serão feitos os *locks* são os nodos das árvores-B*.

- nodos seguros são nodos que não estão totalmente ocupados, isto é, ainda comportam pelo menos mais um item. Se o nodo N_i é um nodo seguro então todos os nodos $N_1, N_2, \ldots N_{i-1}$ não pertencem ao escopo da transação, pois não serão alterados durante a fase de reestruturação.

Quando duas ou mais transações estiverem fazendo acesso concorrente a uma mesma árvore-B*, conflitos poderão ocorrer se algum nodo pertencente ao caminho de uma transação pertencer ao escopo de uma transação de escrita, ou se duas ou mais transações de escrita possuírem algum nodo em comum em seus escopos. Para que os conflitos não ocorram, é necessário um mecanismo para serializar o acesso aos nodos pertencentes ao escopo de uma transação. Estes mecanismos, ou protocolos de *lock*, serão apresentados a seguir na forma de algoritmos concorrentes.

Na literatura existem vários algoritmos para o acesso concorrente a árvores-B* e outras estruturas correlatas [Kung 80, Lehman 81, Kwong 82, Manber 84, Bernstein 87, Mohan 89]. Estes algoritmos foram divididos em duas categorias, segundo [Kwong 82]: algoritmos conservadores e algoritmos agressivos.

Os algoritmos são classificados como conservadores quando uma transação serializa o acesso a todos os nodos pertencentes ao seu caminho e/ou escopo, não permitindo que transações de escrita concorrentes e conflitantes acessem nodos que ainda serão utilizados durante as fases de pesquisa e/ou reestruturação da mesma. Os algoritmos conservadores evitam que os caminhos e/ou escopos das transações sejam alterados por transações conflitantes.

Os algoritmos são classificados como agressivos quando uma transação permite que transações de escrita concorrentes e conflitantes alterem seus caminhos e/ou escopos durante a execução da mesma. Os algoritmos agressivos fornecem mecanismos para a detecção e correção dos caminhos e/ou escopos alterados por outras transações.

Os algoritmos conservadores são mais simples e não exigem modificações na definição das árvores-B, porém possuem um grau de paralelismo menor que os algoritmos agressivos, que exigem modificações na estrutura das árvores-B e são mais complexos.

Os algoritmos conservadores e agressivos são descritos a seguir. Estes algoritmos utilizam dois tipos de *lock*: *r-lock* e *w-lock*. Um *r-lock* representa um *lock* de leitura e um *w-lock* representa um *lock* de escrita, isto é, quando uma transação deseja fazer acesso a um objeto de forma compartilhada ela utiliza um *r-lock*, e

quando deseja fazer acesso a um objeto de forma exclusiva ela utiliza um *w-lock*. Um *r-lock* é compatível com outro *r-lock*, mas é incompatível com um *w-lock*. Um *w-lock* é incompatível com *r-lock* e *w-lock*.

Apenas as operações de pesquisa e inserção são tratadas nos algoritmos para o acesso concorrente a árvores-B neste texto.

3.1. ALGORITMOS CONSERVADORES

Vários algoritmos conservadores para o acesso concorrente a árvores-B foram propostos, dentre os quais pode ser citado [Kwong 82].

Como foi dito anteriormente, conflitos podem ocorrer quando algum nodo pertencente ao caminho de uma transação pertencer ao escopo de uma transação de escrita, ou quando duas ou mais transações de escrita possuírem algum nodo em comum em seus escopos.

Para que os conflitos não ocorram, é necessário um mecanismo para serializar o acesso aos nodos pertencentes ao escopo de uma transação.

A idéia básica é executar *w-locks* em todos os nodos pertencentes ao escopo de uma transação de escrita. Contudo, uma transação não tem conhecimento de seu escopo a priori. Somente ao final da fase de pesquisa é que a transação tem o seu escopo determinado.

O problema pode ser resolvido com o seguinte protocolo: durante a fase de pesquisa a transação de escrita executa *w-locks* nos nodos lidos, e à medida que for encontrando nodos seguros os *locks* anteriores vão sendo liberados. Ao final da fase de pesquisa apenas os nodos pertencentes ao escopo da transação estarão com *w-locks*.

As transações de leitura utilizam apenas *r-locks*, através do protocolo denominado *lock coupling* [Kwong 82], que funciona da seguinte maneira: a transação faz um *r-lock* no nodo N_i, para poder lê-lo. Após a leitura e conseqüente determinação do nodo N_{i+1}, a transação libera o *lock* em N_i. Este processo se repete até atingir um nodo folha.

O principal problema dos algoritmos conservadores diz respeito ao grau de paralelismo. Podem ocorrem casos onde apenas uma transação esteja acessando a

árvore. Isto ocorre quando todos os nodos pertencentes ao caminho da transação também pertencerem ao seu escopo.

3.2. ALGORITMOS AGRESSIVOS

Vários algoritmos agressivos para o acesso concorrente a árvores-B foram propostos, dentre os quais pode ser citado [Lehman 81].

Um meio de aumentar o grau de paralelismo das transações é obter um *lock* apenas no nodo que estiver sendo acessado naquele instante (tanto para consulta como para alteração), isto é, toda transação detém no máximo um *lock* por vez. No entanto, este método pode levar a inconsistências, pois o caminho de uma transação pode ser alterado por outra transação de escrita. Para resolver este problema as transações utilizam mecanismos para poder recuperar os caminhos alterados, ao contrário das transações conservadoras discutidas na seção anterior, que evitam que seus caminhos sejam alterados.

Para recuperar seus caminhos, as transações necessitam de alterações na estrutura da árvore-B*. Cada nodo da árvore-B* possui um campo a mais denominado *link*. O campo *link* de um nodo aponta para o seu irmão à direita em um mesmo nível. A função deste novo campo é o de recuperar o caminho alterado por outra transação. O algoritmo de inserção é responsável pela manutenção do campo *link*.

Como as transações detêm no máximo um *lock*, o caminho da mesma pode ser alterado a qualquer instante. O caminho de uma transação é alterado somente quando um determinado nodo N_i pertencente ao escopo de uma transação de escrita tiver sido alterado de forma que alguns de seus ítens tenham sido movidos para um novo nodo N'_i, irmão à direita de N_i.

O protocolo utilizado nos algoritmos agressivos é o seguinte: uma transação na fase de pesquisa obtém um *lock* no nodo N_i para poder lê-lo e determinar o nodo N_{i+1}. Esta determinação é feita utilizando-se uma chave de pesquisa. Para fazer o *lock* no nodo N_{i+1} a transação deve antes liberar o *lock* em N_i. Após ler o nodo N_{i+1} a transação verifica se este nodo realmente pertence ao seu caminho, comparando o último item do nodo N_{i+1} com a chave de pesquisa. Caso a transação constate a alteração no seu caminho[4], ela deve seguir o campo *link*, isto é, deve ler o nodo N'_{i+1} irmão à direita de N_{i+1} e repetir a comparação utilizando a chave de pesquisa. Este

[4]Apenas o algoritmo de inserção é responsável pelas alterações no caminho de outra transação. Lembre-se de que o algoritmo de remoção não está sendo tratado neste texto.

processo se repete até a transação atingir um nodo N''_{i+1} tal que seu caminho esteja recuperado. Feito isto, a transação pode continuar normalmente, com seu novo caminho.

A situação descrita acima ocorre quando um nodo pertencente ao caminho de uma transação é alterado antes da transação lê-lo. Porém, um nodo pertencente ao caminho da transação pode ser alterado após a leitura do mesmo. Neste caso, este nodo só será novamente utilizado pela transação se esta transação for de escrita e o nodo pertencer ao escopo da mesma. A transação de escrita deve então proceder da mesma forma utilizando o campo *link* durante a fase de reestruturação para poder recuperar o seu caminho.

4. O ALGORITMO PROPOSTO

Na seção anterior foram descritos os algoritmos com postura conservadora e agressiva para o acesso concorrente a árvores-B* encontrados na literatura. Nesta seção é apresentada a motivação que levou ao projeto de novos algoritmos para o acesso concorrente a árvores-B*, que são descritos a seguir.

Os algoritmos conservadores se baseiam no conceito de nodos seguros para poder predeterminar o escopo de uma transação, isto é, onde serão feitos os *locks*. No LEMIX, o conceito de nodo seguro é muito vago, pois a priori é impossível determinar se um nodo comporta ou não mais um item, devido ao fato dos ítens serem de tamanho variável. Por este motivo, os algoritmos com postura conservadora foram descartados da implementação do LEMIX.

No LEMIX o algoritmo de inserção implementado utiliza o tratamento de *overflow*, o que significa que durante a fase de reestruturação de uma transação de escrita ítens de um determinado nodo podem ser movidos para seu irmão da esquerda ou da direita. Este fato implicaria na utilização de dois *links* por nodo para poder recuperar o caminho alterado, se fossem utilizados os algoritmos agressivos. Neste caso, a transação ao ler um nodo deveria comparar a chave de pesquisa com o primeiro e último ítens para poder determinar alguma alteração no seu caminho, e seguir um dos *links* se fosse necessário. Para evitar este *overhead*, os algoritmos com postura agressiva foram também descartados da implementação do LEMIX.

Os fatos descritos acima motivaram a criação de novos algoritmos para o acesso concorrente a árvores-B*, que são descritos a seguir.

As transações de escrita, que foram anteriormente divididas em fase de pesquisa e fase de reestruturação, são agora divididas em três fases, a saber:

- fase de pesquisa;
- fase de alteração simples;
- fase de reestruturação.

A fase de pesquisa é idêntica à fase de pesquisa das transações de escrita já descrita anteriormente.

A fase de alteração simples ocorre quando a transação de escrita insere um novo item em um nodo folha sem propagar a alteração para os níveis superiores. A ocorrência desta fase implica na não ocorrência da fase de reestruturação. A fase de reestruturação só ocorre quando o nodo folha não comporta um novo item, quando, então é executado o tratamento de *overflow* ou a criação de um novo nodo. A execução da fase de reestruturação implica em alterações estruturais na árvore, enquanto a execução da fase de alteração simples implica na não alteração na estrutura da árvore.

Conflitos podem ocorrer quando duas ou mais transações de escrita se encontram na fase de reestruturação, ou quando uma transação se encontra na fase de pesquisa e outra na fase de reestruturação. Duas transações em fase de pesquisa ou em fase de alteração simples não são conflitantes, pois nestas fases não há alteração estrutural na árvore, alterações estas que são responsáveis pelos conflitos.

Os algoritmos propostos devem então evitar estes tipos de conflitos. No primeiro caso, isto é, duas transações em fase de reestruturação, os algoritmos propostos evitam o conflito não permitindo que duas ou mais transações reestruturem a árvore ao mesmo instante, ou seja, não se encontrem ao mesmo instante na fase de reestruturação. Este protocolo pode parecer radical, mas sabe-se que as alterações nos níveis de índice (nodos internos) são bem menos freqüentes se comparadas a pesquisas e alterações simples nos nodos folha [Bernstein 87].

Para evitar conflitos do segundo tipo, isto é, uma transação na fase de pesquisa e outra na fase de reestruturação, os algoritmos propostos detectam alterações no caminho e/ou escopo das transações durante a fase de pesquisa. Se uma transação detecta alguma alteração em seu caminho e/ou escopo, ela termina e recomeça novamente do início. Este protocolo também pode parecer radical à primeira vista, mas utilizando a mesma justificativa [Bernstein 87], alterações

estruturais nos níveis de índice são bem menos freqüentes se comparadas a pesquisas e alterações simples no nível de folhas.

Uma transação na fase de pesquisa, ao caminhar pela árvore utilizando um determinado caminho, deve ler todos os nodos pertencentes a este caminho até atingir um nodo folha. Esta leitura é feita nodo a nodo, isto é, dado o caminho $N_1, N_2, \ldots N_h$, a transação deve ler o nodo N_i para depois ler o nodo N_{i+1}. Durante a fase de pesquisa outra transação pode alterar o caminho desta transação modificando um ou mais nodos deste caminho. Suponha que a outra transação altere este caminho modificando o nodo N_i. Duas situações podem ocorrer, dependendo de a transação que se encontra na fase de pesquisa já ter lido ou não o nodo N_i. Se a transação ainda não leu o nodo N_i, ao lê-lo a mesma deve detectar imediatamente a modificação e recomeçar novamente a fase de pesquisa. Esta primeira situação é denominada *alteração no caminho*. A segunda situação ocorre quando a alteração no nodo N_i tiver sido feita após a leitura do mesmo pela transação que se encontra na fase de pesquisa. Neste caso, a transação só irá detectar a alteração no nodo N_i se a transação for de escrita e se o nodo N_i pertencer ao escopo da mesma. Esta situação é denominada *alteração no escopo*.

Os algoritmos propostos devem então detectar alterações no caminho das transações de leitura e detectar alterações no caminho e no escopo das transações de escrita. Para que isto seja possível, algumas modificações foram feitas na definição da árvore-B* modificada do LEMIX:

- criação de um campo denominado *bit_modificação* nos nodos da árvore. Este campo normalmente está desligado (valor 0). Quando uma transação de escrita altera um determinado nodo para poder reestruturar a árvore, o campo *bit_modificação* deste nodo é ligado (recebe o valor 1). A transação de escrita ao terminar desliga todos os campos *bit_modificação* que porventura tenham sido ligados pela mesma[5].

- criação de um campo denominado *time_stamp* nos nodos da árvore. O valor deste campo é incrementado sempre que um nodo for alterado por alguma transação de escrita.

A função do campo *bit_modificação* é possibilitar que uma transação detecte alguma alteração em seu caminho durante a fase de pesquisa. Ao ler um nodo, a

[5] O LEMIX utiliza um *buffer-pool* compartilhado onde são armazenados todos os nodos lidos durante a fase de pesquisa. Estes nodos ficam fixos em memória até o termino da transação. Esta característica facilita o *desligamento* de todos os campos *bit_modificação* dos nodos alterados, pois estes com certeza se encontram em memória, não sendo necessário nenhuma leitura extra.

transação deve verificar se o *bit_modificação* no nodo lido está ligado. Se estiver, significa que uma alteração no caminho foi detectada.

A função do campo *time_stamp* é possibilitar que uma transação de escrita detecte alguma alteração em seu escopo. A verificação é feita comparando-se os *time_stamps* anotados durante a fase de pesquisa com os *time_stamps* dos nodos lidos antes de entrar na fase de reestruturação. Se algum *time_stamp* tiver sido incrementado significa que a transação detectou uma alteração em seu escopo[6].

Os algoritmos propostos utilizam três tipos de *lock*: *r-lock* e *w-lock* já descritos, e um terceiro tipo, *p-lock*. Um *p-lock* é compatível com *r-locks* e incompatível com *w-lock* e outro *p-lock*. Um *p-lock* é utilizado quando a transação de escrita deseja fazer acesso a um nodo que no futuro será modificado. Enquanto o nodo estiver com *p-lock* outras transações com *r-lock* no nodo também terão acesso ao mesmo. Quando a transação de escrita resolver alterar o nodo o *p-lock* deve ser convertido para *w-lock*.

Os algoritmos utilizam o protocolo de *lock coupling* durante a fase de pesquisa, que também é utilizado nos algoritmos conservadores descritos anteriormente. As transações de escrita utilizam um *lock* especial na árvore para serializar a fase de reestruturação. Os algoritmos são apresentados a seguir.

Transações de leitura (pesquisa):

1. faça um *r-lock* no nodo N_1, onde N_1 é a raiz da árvore
2. repita os passos 2.1 a 2.5 variando i de 1 até h, onde h é a altura da árvore
   - 2.1. leia o nodo N_i
   - 2.2. se o *bit_modificação* de N_i estiver ligado então faça um *unlock* em N_i e recomece do passo 1
   - 2.3. pesquise em N_i pela sub-árvore onde se encontra a chave de pesquisa. Seja N_{i+1} a raiz desta sub-árvore.
   - 2.4. faça um *r-lock* em N_{i+1}
   - 2.5. faça um unlock em N_i
3. pesquise em N_h pelo registro onde se encontra a chave de pesquisa
4. faça um *unlock* em N_h

---

[6] O algoritmo proposto supõe que todos os nodos pertencentes ao caminho da transação pertencem ao escopo da mesma para efeito de comparação dos campos *time_stamp*.

Transações de escrita (inserção):

Fase de pesquisa:

1. faça um *r-lock* em N_1, onde N_1 é a raiz da árvore
2. repita os passos 2.1 a 2.4 variando i de 1 até h, onde h é a altura da árvore. Suponha h > 1
   - 2.1. leia o nodo N_i
   - 2.2. se o *bit_modificação* de N_i estiver ligado então faça um *unlock* em N_i e recomece do passo 1
   - 2.3. anote o *time_stamp* de N_i
   - 2.4. se i < h então
     - 2.4.1. pesquise em N_i pela sub-árvore onde será inserido o novo registro. Seja N_{i+1} a raiz desta sub-árvore
     - 2.4.2. se N_{i+1} for um nodo folha então faça um *p-lock* em N_{i+1}, senão faça um *r-lock* em N_{i+1}
     - 2.4.3. faça um *unlock* em N_i

Fase de alteração simples:

3. se o nodo N_h comporta o novo registro então
   - 3.1. converta o *p-lock* de N_h para *w-lock*
   - 3.2. insira o novo registro no nodo N_h
   - 3.3. incremente o *time_stamp* de N_h
   - 3.4. faça um *unlock* em N_h e termine a transação

Fase de reestruturação:

4. faça um *unlock* em N_h
5. faça um *w-lock* na árvore
6. verifique os *time_stamps* anotados. Se algum foi alterado então faça um *unlock* na árvore e recomece do passo 1
7. repita os passos 7.1 a 7.6 variando i de h até 1
   - 7.1. faça um *p-lock* no nodo irmão à esquerda de N_i. Seja B'_i este nodo
   - 7.2. faça um *p-lock* em N_i
   - 7.3. se consegue redistribuir os registros entre os nodos N_i e B'_i então
     - 7.3.1. converta os *p-locks* de N_i e B'_i para *w-locks*
     - 7.3.2. redistribua os registros entre os dois nodos
     - 7.3.3. ligue os *bits_modificação* dos nodos N_i e B'_i
     - 7.3.4. incremente os *time_stamps* dos nodos N_i e B'_i
     - 7.3.5. faça *unlocks* nos nodos N_i e B'_i
   - 7.4. se não redistribuiu os registros entre os nodos N_i e B'_i então
     - 7.4.1. faça um *unlock* em B'_i
     - 7.4.2. faça um *p-lock* no nodo irmão à direita de N_i. Seja B''_i este nodo

7.4.3. se consegue redistribuir os registros entre os nodos N_i e B^*_i então
7.4.3.1. repita os passos 7.3.1 a 7.3.5 para os nodos N_i e B^*_i
7.4.4. se não redistribuiu os registros entre os nodos N_i e B^*_i então
7.4.4.1. faça um *unlock* em B^*_i
7.4.4.2. converta o *p-lock* de N_i para *w-lock*
7.4.4.3. crie um novo nodo na árvore, irmão à direita de N_i
7.4.4.4. ligue o *bit_modificação* de N_i
7.4.4.5. incremente o *time_stamp* de N_i
7.4.4.6. faça um *unlock* em N_i
7.5. se i = 1 então crie uma nova raiz e abandone o *loop*
7.6. se i > 1 então
7.6.1. faça um *p-lock* em N_{i-1}
7.6.2. se consegue inserir o item derivado da reestruturação do nível inferior em N_{i-1} então
7.6.2.1. converta o *p-lock* de N_{i-1} para *w-lock*
7.6.2.2. insira o novo item em N_{i-1}
7.6.2.3. incremente o *time_stamp* de N_{i-1}
7.6.2.4. faça um *unlock* em N_{i-1}
7.6.2.5. abandone o *loop*
7.6.3. se não inseriu o item derivado da reestruturação do nível inferior em N_{i-1} então
7.6.3.1. faça um *unlock* em N_{i-1}
8. desligue todos os *bits_modificação* ligados pela transação
9. faça um *unlock* na árvore

Os algoritmos propostos são livres de *deadlock*, pois todos os *locks* são obtidos seguindo-se uma seqüência. Na fase de pesquisa os locks são obtidos seguindo-se a ordenação do caminho, isto é, da raiz para um determinado nodo folha. Na fase de reestruturação o *lock* na árvore é obtido após a transação ter liberado todos os seus *locks*. Nesta mesma fase, durante a propagação das alterações, a transação só obtém algum *lock* em um determinado nível após ter liberado todos os *locks* do nível inferior.

5. CONCLUSÕES

Neste texto foram apresentados os algoritmos para o acesso concorrente a árvores-B encontrados na literatura. Estes algoritmos foram classificados por [Kwong 82] como algoritmos conservadores e algoritmos agressivos. Foi visto que nos algoritmos conservadores as transações evitam que seus caminhos e/ou escopos

sejam alterados por outras transações concorrentes durante a sua execução. Foi visto também que nos algoritmos agressivos as transações permitem alterações nos seus caminhos e/ou escopos, porém fornecendo mecanismos para a detecção e correção destas alterações. Os algoritmos conservadores são mais simples, porém os agressivos possuem um maior grau de paralelismo, isto é, são mais eficientes.

Os algoritmos propostos podem ser classificados como algoritmos híbridos. Durante a fase de pesquisa, as transações adotam uma postura agressiva, permitindo alterações em seus caminhos, ao custo de terem de detectar estas alterações e recomeçar novamente a fase de pesquisa. Durante a fase de reestruturação, as transações de escrita adotam uma postura conservadora, não permitindo que outra transação se encontre também nesta fase. Esta postura conservadora garante que o escopo da transação não seja alterado por outra transação concorrente durante a sua execução.

Os algoritmos propostos podem ser considerados simples se comparados aos algoritmos agressivos, por não implementarem o mecanismo de recuperação de caminho e/ou escopo alterados. Podem também ser considerados eficientes quanto ao grau de paralelismo se for levado em consideração o fato de que as alterações estruturais na árvore são mínimas se comparadas ao número de pesquisas e alterações simples nos nodos folha.

Ainda não foram feitas medidas de desempenho, pois os algoritmos se encontram em fase final de implementação, mas tem-se espectativas de que os tempos obtidos serão satisfatórios se forem levadas em consideração as características relacionadas ao acesso às árvores-B^*, já discutidas anteriormente.

6. REFERÊNCIAS

[ANSI 90] ANSI/MDC X11.1, "Programming Languages - MUMPS", 1990.

[Bayer 77] Bayer R., & Unterauer, K., "Prefix B-Trees", ACM Trans. on Database Systems, vol. 2, no. 1, Mar 1977.

[Bayer 80] Bayer R., Heller, H. & Reiser, A., "Parallelism and Recovery in Database Systems", ACM Trans. on Database Systems, vol. 5, no. 2, Jun 1980.

[Bernstein 87] Bernstein, P., Hadzilacos, V. & Goodman, N., "Concurrency Control and Recovery in Database Systems", Addison Wesley, 1987.

[Comer 79] Comer, D., "The Ubiquitous B-Tree", ACM Computing Surveys, vol. 11, no. 2, Jun 1979.

[Eswaram 76] Eswaram, K. et al., "The Notions of Consistency and Predicate Locks in a Database System", CACM, vol. 19, no. 11, Nov 1976.

[Gray 78] Gray, J., "Notes on Data Base Operating Systems", IBM Research Report, Fev 1978.

[Kedem 83] Kedem, Z. & Silberschatz, A., "Locking Protocols: From Exclusive to Shared Locks", Journal of the ACM, vol. 30, no. 4, Out 1983.

[Kung 80] Kung, H. & Lehman, P., "Concurrent Manipulation of Binary Search Trees", ACM Trans. on Database Systems, vol. 5, no. 5, Set 1980.

[Kwong 82] Kwong, Y., & Wood, D., "A New Method for Concurrency in B-Trees", IEEE Trans. on Software Engineering, vol. SE-8, no. 3, Mar 1982.

[Lehman 81] Lehman, P. & Yao, S., "Efficient Locking for Concurrent Operations on B-Trees", ACM Trans. on Database Systems, vol. 6, no. 4, Dez 1981.

[Manber 84] Manber, U. & Ladner, R., "Concurrency Control In a Dynamic Search Structure", ACM Trans. on Database Systems, vol. 9, no. 3, Set 1984.

[McCreight 77] McCreight, E., "Pagination of B^*-Trees with Variable-Length Records", CACM, vol. 20, no. 9, Set 1977.

[Mohan 89] Mohan, C., "ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes", IBM Research Report, Set 1989.

[Papadimitriou 86] Papadimitriou, C., "The Theory of Database Concurrency Control", Computer Science Press, 1986.

[Wagner 73] Wagner, R., "Indexing Design Considerations", IBM Systems Journal no. 4, 1973.

[Wirth 76] Wirth, N., "Algorithms + Data Structures = Programs", Prentice-Hall, 1976.