Controle de Concorrência

Fernando Menezes Matos, Luiz Celso Gomes Jr.

11 de outubro de 2005

Trabalho desenvolvido como parte da avaliação do curso MO410/051, UNICAMP. É um resumo livre do Capítulo 17 do livro Ramarkrishnan, R. and Gehrke, J. (2003) "Data Management Systems", McGraw-Hill, 3rd edition.

1 2PL, seriabilidade e recuperabilidade

Técnicas de *locking* podem ser utilizadas para garantir seriabilidade e recuperabilidade nos escalonamentos das transações do banco de dados. Um escalonamento conflito-serializável é um escalonamento equivalente a alguma execução serial das transações.

Os conflitos entre as transações de um dado escalonamento S podem ser identificados através da construção de um **grafo de precedência**, também chamado de **grafo de seriabilidade**, onde: *(i)* cada nó corresponde a uma transação de S; *(ii)* há um arco de *Ti* para *Tj* se uma ação de *Ti* precede e conflita com alguma das ações de *Tj*. Duas ações são conflitantes quando operam sobre o mesmo objeto e pelo menos uma delas é uma ação de escrita. Um escalonamento S é conflito serializável se e somente se seu grafo de precedência é acíclico.

Os principais protocolos para garantir seriabilidade são o ***Strict Two-Phase Lock*** **(Strict 2PL)** e o ***Two-Phase Lock*** **(2PL)**. Ambos garantem que o gráfico de precedência do escalonamento é acíclico. Dependendo da estratégia para garantir a seriabilidade do escalonamento a recuperabilidade do sistema pode ser comprometida.

Uma outra alternativa, menos restritiva que Strict 2PL e 2PL, para garantir a seriabilidade de um escalonamento é a chamada **Seriabilidade de Visão**.

Esta estratégia possibilita uma maior concorrência entre as transações mas o custo da sua avaliação é proibitivo, vetando sua utilização na prática.

2 Introdução ao gerenciamento de *lock*

O **gerenciador de *lock*** é a parte do SGBM responsável pelo gerenciamento dos *locks* concedidos às transações. Para cumprir sua função ele mantém uma **tabela de *locks*** onde armazena todos os *locks* associados aos objetos de dados. Uma **entrada da tabela de *lock*** contém os seguintes dados: o número de transações atualmente com o *lock* de um objeto (mais de um se o *lock* é do tipo compartilhado), a natureza do *lock* (compartilhado ou exclusivo) e um apontador para a lista de requisições de *lock*.

De posse das informações contidas na tabela de *lock*, o gerenciador é capaz de manipular as requisições de *lock* das transações, concedendo *locks* sobre objetos que não representem ameaça de conflito ou colocando as requisições na fila até que possam ser atendidas.

A implementação dos comandos de *lock* e *unlock* deve garantir que eles sejam operações atômicas. Para tal o acesso à tabela de *lock* deve utilizar algum mecanismo de sincronização (e.g. semáforos).

Em alguns casos, uma transação pode precisar obter um *lock* exclusivo sobre um objeto no qual ela já possui um *lock* compartilhado. Por exemplo, um comando SQL update sobre uma tabela pode obter um *lock* compartilhado sobre cada linha da mesma para verificar condição da cláusula where e, em seguida, obter *locks* exclusivos sobre as linhas que satisfazem a condição. Isto é chamado de **upgrade de *lock*** e é uma característica importante e desejável nos SGDBs.

3 Tratando *deadlocks*

Situações de *deadlock* tendem a ser raras e envolver poucas transações. Na prática, porém, os sistemas de bancos de dados verificam periodicamente a incidência de *deadlocks*. Para tal, o gerenciador de *locks* utiliza uma estrutura chamada **grafo waits-for** onde os nós correspondem a transações ativas e há um arco da Ti a Tj se e somente se Ti está esperando que Tj libere um *lock*.

Ciclos no grafo waits-for, indicando *deadlocks*, são periodicamente verificados. Um *deadlock* é resolvido abortando-se uma transação que está num ciclo

e liberando todos os seus *locks*, o que permite que outras transações obtenham os *locks* que estavam esperando. A escolha da transação a ser abortada pode se basear em diversos critérios: aquela com menos *locks*, aquela que realizou menos trabalho, aquela que está mais longe de terminar e assim por diante. Seja qual for a estratégia, deve-se garantir que nenhuma transação sofrerá starvation.

Uma alternativa simples para evitar o custo computacional de se manter o grafo waits-for é identificar *deadlocks* através de um mecanismo de timeout: se uma transação está esperando tempo demais por um *lock*, assumimos (de forma pessimista) que ela está num ciclo de *deadlock* e a abortamos.

Na prática casos de *deadlocks* não são freqüentes e estratégias de tratamento baseadas em detecção funcionam bem. Porém, se há um alto nível de contenção de *locks* e, conseqüentemente, uma maior propensão a *deadlocks*, estratégias baseadas na prevenção de *deadlocks* podem ser utilizadas.

Uma forma de se prevenir *deadlocks* é atribuir prioridades às transações (por exemplo, baseadas no *timestamp*) e não permitir que transações de menor prioridade esperem por *locks* de transações de maior prioridade. Nesta linha existem duas políticas possíveis, considerando que a transação *Ti* requisita um *lock* de posso de *Tj*:

- **Wait-die**: Se *Ti* tem maior prioridade ela pode esperar; caso contrário ela é abortada.
- **Wound-wait:** Se *Ti* tem maior prioridade, *Tj* é abortada; caso contrário *Ti* espera.

4 Técnicas especializadas de *locking*

Até agora o banco de dados foi tratado como uma coleção *fixa* de objetos *independentes*. A discussão nesta seção relaxa estas premissas e discute os problemas que devem ser considerados quando nosso banco de dados pode crescer ou diminuir através da inclusão ou exclusão de objetos e também as considerações que podem ser feitas a partir dos relacionamentos entre os objetos do banco (por exemplo, uma base de dados possui tabelas que são armazenadas em páginas que contêm tuplas).

4.1 Bancos de dados dinâmicos e o *phantom problem*

A partir do momento em que se considera que dados são dinâmicos, transações que fazem alguma operação sobre um determinado conjunto de dados podem ser afetadas por transações que incluem ou excluem algum dado sobre este conjunto. Isto pode levar à situações que comprometem a seriabilidade dos escalonamentos ainda que protocolos como o Strict 2-PL sejam usados. Este é o chamado *phantom problem*.

Para evitar que tais situações aconteçam, sempre que uma transação for manipular um conjunto de dados o banco deve garantir que nenhuma outra transação poderá alterar este conjunto. Uma alternativa é utilizar a técnica de ***lock* de índic**e, que consiste em bloquear as entradas do índice que correspondem ao conjunto de dados que se deseja manipular. Esta técnica só pode ser utilizada quando há um índice sobre as colunas que designam o conjunto. Caso contrário todas as páginas devem ser bloqueadas.

Outra alternativa é o *lock* de predicados, onde o sistema armazena os predicados utilizados nas consultas e não permite que consultas conflitantes sejam executadas. Esta alternativa é computacionalmente cara e de difícil implementação e, em geral, não é utilizada.

4.2 Controle de concorrência em árvores B+

As árvores B+ e suas variantes são as estruturas de dados mais utilizadas para indexação em bancos de dados, o que demanda protocolos eficientes de *lock* na utilização dos índices. Uma estratégia simplista de *lock* seria a de se percorrer a árvore a partir da raiz obtendo *locks* até se alcançar a folha desejada. O problema desta abordagem é que ela provoca um gargalo nos níveis mais altos da árvore impedindo que outras transações utilizem a estrutura.

Duas observações importantes devem ser consideradas para um melhor proposta:

1. os níveis mais altos da árvore apenas direcionam as procuras até as folhas;
2. para inclusões na árvore um nó no caminho de procura só precisa ser bloqueado quando splits em cascata podem se propagar até ele (raciocínio similar para exclusões).

Baseado nestas observações uma melhor estratégia para bloqueio nas árvores pode ser baseada em:

- Procura: começa na raiz e desce; repetidamente obtém um *lock* compartilhado no filho e desbloqueia o pai.
- Inclusão/Exclusão: começa na raiz e desce obtendo *locks* exclusivos. Uma vez que o filho está bloqueado, verifica se ele é seguro. Se o filho é seguro, libera todos os *locks* dos ancestrais.

Um nó é dito seguro quando as mudanças na estrutura causadas pela inclusão ou exclusão não podem afetá-lo (por exemplo, numa inclusão, quando o nó não está cheio).

Algoritmos melhores podem ser propostos a partir desta idéia, por exemplo o algoritmo de Bayer-Schkolnick.

4.3 *Locking* de múltipla granularidade

A técnica de *locking* de múltipla granularidade permite bloqueios eficientes de objetos que contêm outros objetos (e.g. uma banco de dados contém um conjunto de arquivos, cada arquivo contém um conjunto de páginas e cada página contém um conjunto de registros). A idéia é explorar a natureza hierárquica do relacionamento 'contém' como numa árvore. Um *lock* em um nó bloqueia ele próprio e todos os seus descendentes.

Os protocolos de *locking* de múltipla granularidade complementam os *locks* compartilhados (S) e exclusivos (X) com dois novos tipos de *locks*: **intenção de compartilhado** (IS), que conflitam com apenas com os *locks* X; e **intenção de exclusivo** (IX), que conflitam com *locks* S e X.

Numa situação típica onde uma transação precisa ler um arquivo inteiro e modificar alguns poucos registros, ela pode obter *locks* S e IX no arquivo e subseqüentemente obter *locks* X para os registro que serão modificados. Isto evita o bloqueio no modo exclusivo de todo o arquivo permitindo maior concorrência no escalonamento das transações.

5 Controle de concorrência sem *lock*

5.1 Controle de concorrência otimista

Protocolos de controle de concorrência baseados em *lock* utilizam uma abordagem pessimista para a resolução de conflitos entre transações. Eles assumem que na maioria das transações ocorrerá conflitos, por isso eles bloqueiam ou

abortam estas transações. Contudo em sistemas onde não há grande acesso de escrita a objetos, esta abordagem acaba sendo prejudicial.

A abordagem em controles de concorrência otimistas é a de que na maioria das transações não haverá conflitos com outras transações. As transações executam em três fases:

- Leitura: Nesta fase a transação lê o valor da banco de dados e guarda uma cópia em uma área local, chamada *workspace*;
- Validação: Nesta fase o SGBD verifica se a transação pode conflitar com alguma outra transação que está executando concorrentemente. Caso exista este risco, a transação é abortada, seu *workspace* é esvaziado e ela é reiniciada; e
- Escrita: Caso a transação passe pela fase de validação, as alterações feitas no objeto em seu *workspace* são persistidas no banco de dados.

Importante notar que está abordagem somente acarretará em um melhor desempenho se hoverem poucos conflitos e a validação puder ser feita de maneira eficiente. Caso contrário, o custo de reiniciar as transações continuamente será muito alto.

A cada transação Ti é associado um *timestamp* $TS(Ti)$ no início de sua fase de Validação. Para cada par de transação Ti e Tj, tal que $TS(Ti) < TS(Tj)$, pelo menos uma das seguintes condições deve ser satisfeita:

- a) Ti termina, todas as três fases, antes de Tj iniciar;
- b) Ti termina antes de de Tj iniciar sua fase de Escrita, e Ti não modifica nenhum objeto lido por Tj;
- c) Ti termina sua fase de Leitura antes que Tj termine sua fase de Leitura, e Ti não modifica nenhum objeto que é lido ou modificado por Tj.

Caso alguma dessa condições sejam satisfeitas, é possível garantir que para toda transação Ti que realizou commit, tal que $TS(Ti) < TS(Tj)$, as alterações relizadas por Tj não serão visíveis por Ti.

É fácil observar que a primeira condição permite que Tj enxerque as alterações feitas por Ti, uma vez que elas foram executadas em uma ordem serial. A segunda condição permite que Tj leia objetos enquanto Ti ainda não terminou sua execução, contudo não há conflito pois os objetos lidos por Tj e modificados

por *Ti* não são os mesmos. A terceira condição permite que *Ti* e *Tj* modifiquem objetos ao mesmo tempo, contudo nos conjuntos de objetos modificados pelas transações não pode haver intersecção.

Para garantir que estas condições sejam suficientes para verificar se uma transação pode ser executada, é necessário garantir que enquanto uma transação esteja na sua fase de Validação nenhuma outra transação consigo realizar o *commit*, caso contrário a validação da primeira transação poderia deixar de "perceber" um possível conflito com a transação que realizou o *commit*. Um mecanismo de sincronização, como região crítica, poderia ser utilizado para garantir que no máximo uma transação estaria na fase de Validação/Escrita por vez. É necessário também o gerenciamento de listas de objetos lidos e escritos por cada transação, para serem verificadas durante a fase de Validação.

Apesar das vantagens do controle de concorrência otimista, ele também possui desvantagens, tais como: o custo de manter as listas de objetos lidos e escritos por cada transação; a checagem por conflitos; cópia das alterações feitas no objeto no workspace para o banco de dados; e o custo de reiniciar as transações que porventura possam conflitar.

5.2 Resolução de conflito aperfeiçoado

O objetivo da fase de Validação é garantir que *Ti* não modifique nenhum objeto lido por *Tj*, sendo que *TS(Ti)* < *TS(Tj)*, ou se *Ti* modificar objetos lidos por *Tj*, então *Ti* deve executar logicamente antes de *Tj*. O problema é que não é possível dizer quando um objeto modificado por *Ti* foi lido por *Tj*, já que temos apenas a lista de objetos modificados por *Ti* e a lista de objetos lidos por *Tj*.

Uma maneira de aliviar este problema é utilizar um mecanismo similar ao *lock*. A idéia básica é fazer com que cada transação, na sua fase de Leitura, informe ao SGBD quais objetos estão sendo lidos. Quando uma transação *Ti* realiza o *commit*, o SGBD verifica se algum objeto modificado por *Ti* está sendo lido por *Tj*, a qual ainda não foi validada. Caso ocorra, a validação de *Tj* provavelmente vai falhar.

Há duas políticas que podemos assumir durante a validação:

- *Die policy*: Quando o SGBD descobre que uma transação irá conflitar com outra, ele permite que a transação continue até que ela chegue na sua fase de Validação, onde ela será abortada.
- *Kill policy*: Quando o SGBD descobre que uma transação irá conflitar

com outra, ela aborta a transação imediatamente.

5.3 Controle de Concorrência baseado em *Timestamp*

Além do uso de *timestamp* como descrito na sessão anterior, é possível utilizar *timestamp* de outra maneira: À cada transação é associado um *timestamp* no início de sua execução. Com isso é possível garantir que se uma ação *ai* da transação *Ti* conflita com uma ação *aj* da transação *Tj*, então *ai* ocorre antes de *aj* se $TS(Ti) < TS(Tj)$. Caso esta regra seje violada, a transação é aborta e reiniciada.

Para implementar o mecanismo de controle de concorrência descrito anteriormente são associados a cada objeto O da base de dados dois *timestamps*: um *timestamp* de leitura $RTS(O)$ e um *timestamp* de escrita $WTS(O)$. Caso uma transação T deseje ler um objeto O e $TS(T) > WTS(O)$, a leitura é realizada normalmente e $RTS(O)$ é atualizado para o máximo entre $TS(T)$ e $RTS(O)$. Caso $TS(T) < WTS(O)$, a regra de *timestamp* descrita anteriormente é violada e a transação é abortada e reiniciada com um *timestamp* maior. Para ficar mais claro porque a transação é abortada, basta imaginar que se $TS(T) < WTS(O)$, então é porque a transação T está tentando ler um objeto O que foi modificado por outra transação Ti que iniciou após T, desrespeitando assim a regra de serialibilidade imposta pelo *timestamp*.

Para o caso de uma transação T desejar escrever em um objeto O:

- Se $TS(T) < RTS(O)$, então T está tentando modificar um objeto que já foi lido por alguma outra transação que iniciou depois de T. Logo, T é abortada e reiniciada;
- Se $TS(T) < WTS(O)$, T poderia ser abortada pelo mesmo motivo anterior, contudo é possível ignorar a escrita anterior e continuar normalmente. Esta regra é chamada de regra de escrita de Thomas;
- Caso contrário, T modifica O e $WTS(O)$ recebe o valor de $TS(T)$.

5.4 A regra de escrita de Thomas

A explicação da regra de escrita de Thomas é a seguinte: Caso ocorra $TS(T) < WTS(O)$, então é como se T modificasse O logo depois da modificação corrente, ou seja, depois da modificação que gerou $WTS(O)$, O não foi lido por nenhuma outra transação e T tenta modificá-la. Sendo assim a modificação feita por

T1	T2
R(A)	
	W(A)
	Commit
W(A)	
Commit	

Tabela 1: Escalonamento serializável que não é coflito serializável

T1	T2
W(A)	
	R(A)
	W(B)
	Commit

Tabela 2: Escalonamento não recuperável

T não acarretará conflitos, já que O não foi lido por nenhuma outra transação entre as duas escritas. Devido ao uso da regra de escrita de Thomas, o protocolo *timestamp* permite escalonamentos que não são conflito-serializáveis, tais como na Tabela **??**

Isto acontece porque a regra de escrita de Thomas diz que a escrita realizada por *T2* nunca é vista por nenhuma outra transação, sendo assim o escalonamento da Tabela **??** permitido.

5.5 Recuperabilidade

Um dos problemas do protocolo baseado em *timestamp* é que ele permite escalonamentos que não são recuperáveis, como o da Tabela **??**. Se *TS(T1)* = 1 e *TS(T2)* = 2, então este escalonamento é permitido. Para resolver este problema é necessário uma alteração no protocolo de *timestamp*, fazendo com que as ações de escrita sejam enfileiradas até que a transação corrente realize o *commit*. Desta forma, quando *T1* deseja escrever em A, $WTS(A)$ é atualizado para refletir esta ação. Quando *T2* quer ler A, o seu *timestamp* é comparado com $WTS(A)$, e pelo protocolo a leitura é permitida, contudo *T1* não realizou *commit* ainda. Então *T2* fica bloqueada até que *T1* finalize. Esta forma de bloqueio é similar se *T1* obtivesse um *lock* exclusivo em A, contudo com esta modificação, o protocolo de *timestamp* ainda permite escalonamento que o 2PL não permite.

5.6 Controle de concorrência multi-versão

Outra maneira de usar *timestamp* é como no protocolo multi-versão. O objetivo é fazer com que uma transação nunca tenha que esperar para ler um objeto. A idéia é fazer com que cada vez que um objeto seja modificado, uma nova versão deste objeto seja criada com o seu *timestamp* de escrita. Desta forma quando uma transação *Ti* deseja ler um objeto, ela lê a versão mais recente cujo *timestamp* do objeto seja menor que *TS(Ti)*.

Se a transação *Ti* deseja modificar um objeto, então é necessário garantir que este objeto não foi lido por nenhuma outra transação *Tj*, tal que *TS(Ti)* $<$ *TS(Tj)*, ou seja, *Ti* está tentando modificar um objeto que foi lido por *Tj*, fazendo com que *Tj* não consiga enxergar esta modificação.

Para realizar esta verificação, um *timestamp* de leitura também é associado ao objeto *O*, sendo que este *timestamp* é definido com o maior valor entre o seu próprio *timestamp* e o *timestamp* da transação que está tentando ler o objeto. Se *Ti* deseja modificar *O* e *TS(Ti)* $<$ *RTS(O)*, *Ti* é abortada e reiniciada com um *timestamp* maior. Caso contrário, *Ti* cria uma nova versão de *O* e muda os seus *timestamps* de leitura e escrita para o valor de *TS(Ti)*.