Transacoes

Propósito de controlo de concorrência

Muitos SGBDs permitem aos utilizadores efetuar operações simultâneas sobre a base de dados. Se estes operações não são controladas, os acessos podem interferir uns com os outros e a base de dados pode ficar inconsistente. Para solucionar isto, o SGBD implementa um protocolo de controlo de concorrência que previne os acessos à base de dados de interferirem uns com os outros.

Propósito de database recovery

Recuperação de base de dados é o processo de restaurar a base de dados para o estado correto após uma falha. Esta falha pode ser causada por crash de hardware ou software, erro na aplicação ou até por tentativas de corrupção de dados. Qualquer que seja a causa, o SGBD deve ser capaz de recuperar da falha e restaurar a base de dados para um estado consistente.

Transacção: Uma acção ou série de acções efetuados por um único utilizador ou aplicação, que lê ou atualiza os conteudos de uma base de dados.

A principal razão de transacoes é a necessidade de uma série de comandos serem feitos e se um destes falhar, todos falharem (sob o risco dos dados ficarem inconsistentes). Também pode ser usado para controlo de concorrência num sistema com vários acessos em simultâneo. Quando uma transação está a ocorrer, aplica locks.

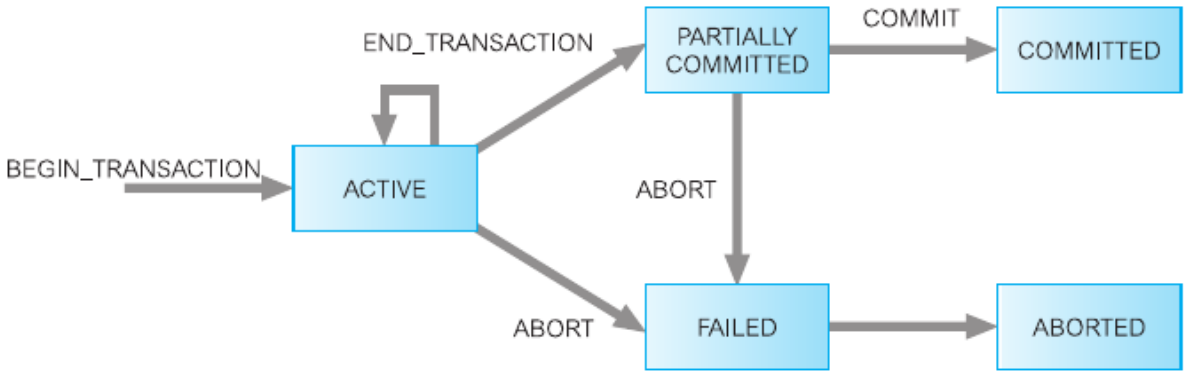
Uma transacção transforma sempre a base de dados de um estado consistente para outro, no entanto isto pode não ser verdade durante a transacção.

Tem dois resultados: ou corre bem e a base de dados chega a um novo estado consistente (commit) ou corre mal(aborted) e volta ao estado anterior ao começar da transacção (rollback).

Uma transação depois de estar commited nao pode ser abortada. Se decidirmos que uma transação foi um erro, temos que fazer uma transação compensatória (compensating transaction) para reverter os seus efeitos.

Uma transação que faz rollback pode voltar a ser executada mais tarde e ter um desfecho diferente (commit com sucesso).

A transação tem vários estados. Começa por estar **ativa**. Se falha, entra no estado **failed**. Se chega ao fim e não tem erros, fica no estado **partially commited** (neste estado, é feito um conjunto de verificações. Se se verificar que a base de dados violou a serialibilidade - ver mais à frente - ou violou alguma constante de integridade, terá de ser abortada e passa ao estado de failed). De failed passa para **aborted**. Se passou as validações em partially commited, passa a **commited**



ACID

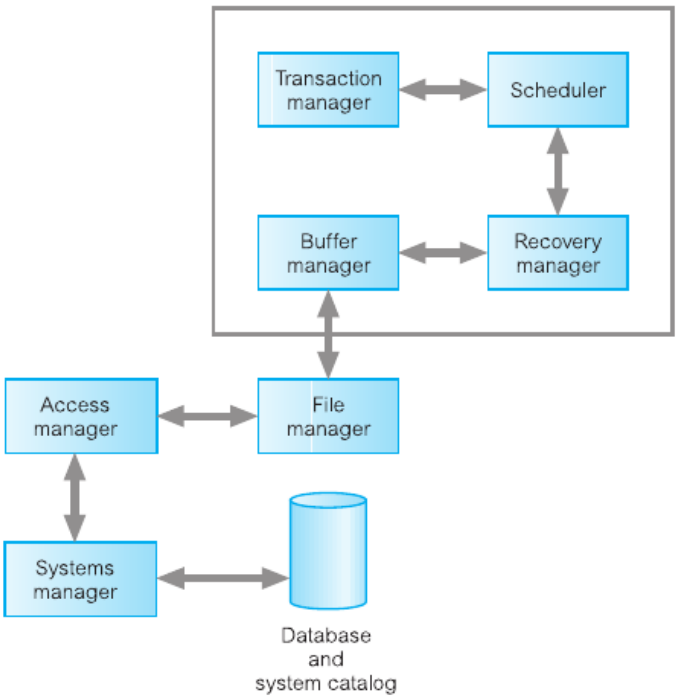
Propriedades das transaçoes:

- **Atomicidade** - a transação é uma unidade indivisivel que ou é efetuada na integra, ou não é efetuada. É da responsabilidade do subsistema de recuperação do SGDB assegurar a atomicidade.
- **Consistência** - Uma transação tem de transformar a base de dados de um estado consistente para outro. É da responsabilidade do SGDB e dos developers assegurar a consistência. O SGDB pode assegurar a consistência forçando que todas as restrições especificadas sejam respeitadas (integridade e restriçoes gerais). No entanto, isto não chega. Se um programador programou uma transação e esta dá um estado inconsistente (por exemplo ao remover dinheiro de uma conta, colocar na errada), a culpa é do programador e o SGDB não tem forma de detetar o erro.
- **Isolamento** - as transaçoes executam independentes umas das outras. Ou seja, os efeitos parciais de uma transação não podem ser visíveis a outras. É da responsabilidade do subsistema de controlo de concorrencia assegurar o isolamento.

- **Durabilidade** - uma vez que uma transição chega a um estado commited, as suas mudanças devem ser guardadas permanentemente na base de dados e nao podem ser perdidas caso alguma instrução posterior falhe. A responsabilidade é do subsistema de recuperação.

Arquitetura de base de dados

O gestor de transacções coordena transacções. Comunica com o scheduler (modulo responsavel por implementar uma estratégia particular de controlo de concorrência).
O scheduler por vezes é referido como gestor de locks se o controlo de concorrência for baseado em locks. O objetivo do scheduler é maximizar a concorrência sem permitir que as transacções que estejam a executar concorrentemente interfiram uma com a outra, o que poderia afetar a integridade e consistência da base de dados.
Se ocorrer uma falha durante a transacção, a base de dados poderia ficar inconsistente. É responsabilidade do gestor de recuperação assegurar que a base de dados é restaurada para o estado em que estava antes do inicio da transacção (estado consistente). Finalmente, o gestor de buffers é responsável pela transferência eficiente de dados entre a memória em disco e a memória principal.



Controlo de concorrência (o processo de gerir operações simultâneas na base de dados sem que interfiram umas com as outras)

Um dos principais objetivos ao desenvolver uma base de dados é permitir a vários utilizadores o acesso a dados partilhados concorrentemente. O acesso é relativamente fácil se todos os utilizadores apenas prentenderem ler dados, dado que não existe forma de interferirem uns com os outros.

Lost Update Problem

Supondo as transações de retirar 10€ da conta X e adicionar 100€ à conta. Vamos supor que a conta tem atualmente 200€. Uma transação e outra começam aproximadamente ao mesmo tempo e correm concorrentemente. Ambas leram o saldo de 200€. A de adicionar coloca lá 300€. Entretanto, a de retirar 10€ atua e guarda 190€ no saldo.

Resolve-se impedindo a segunda transação de ler o valor do saldo antes da primeira o atualizar.

Uncommited dependency problem (ou dirty read problem)

Ao contrário da anterior, esta não é um problema convencional de concorrência

Quando uma transação consegue ver os resultados **intermédios** de outra. Por exemplo, a transação de retirar leu o saldo intermédio da outra transação, que iria ser 300€, antes de esta fazer commit (estava no estado partially commited).

Lê o valor, coloca a 290€ e guarda o saldo. Só que entretanto a outra transação fez rollback e o valor do saldo deveria ser 90, mas é 290€.

Inconsistent Analysis Problem

Até agora temos visto problemas de escrita e leitura. No entanto, há problemas que podem ocorrer só com leituras, se estas forem ao estado intermédio de uma transação.

Vamos supor que a transação T1 está a correr concorrentemente com a transação T2. T1 está a atualizar o saldo da conta X e Y, T2 está a recolher o somatório de todos os saldos do banco. Ambas lêem os valores da conta X que tem 100€. T2 soma e mete o balanço total a 100€. T1 subtrai 10€ à conta X. T2 continua a fazer o somatório. Entretanto T1 atualizou a conta Z com os 10€ que retirou da conta X. T2 quando for a ler, soma os 10€. O somatório de T2 está mal apesar de nao ter havido problemas de leitura e escrita, mas sim porque leu valores intermédios de T1. Os 10€ da transação foram contados 2x (em X e em Z).

Nos exemplos anteriores, a base de dados ficou num estado inconsistente (nos primeiros 2) e apresentou o resultado errado (no último) por não respeitar as propriedades ACID (mais concretamente, a Isolamento)

Serializabilidade e Recuperabilidade

A primeira soluçao é apenas permitir uma transação de cada vez, no entanto queremos maximizar a concorrência.

Transaçoẽs que não têm nada a haver uma com a outra (acedem a diferentes partes da base de dados) podem correr paralelamente. Assim, precisamos de um scheduler.

Schedule é uma sequencia de operações de um conjunto de transações de tal modo que a ordem de operações para cada transação é preservada. Ou seja, para cada sequência neste conjunto, as suas operações devem estar sempre sob a mesma ordem.

Pode ser sequencial (nesse conjunto, primeiro correm as operaçoes da transação 1, depois a 2, depois a 3) ou não-sequencial (as operações estão intercaladas pelas transações)

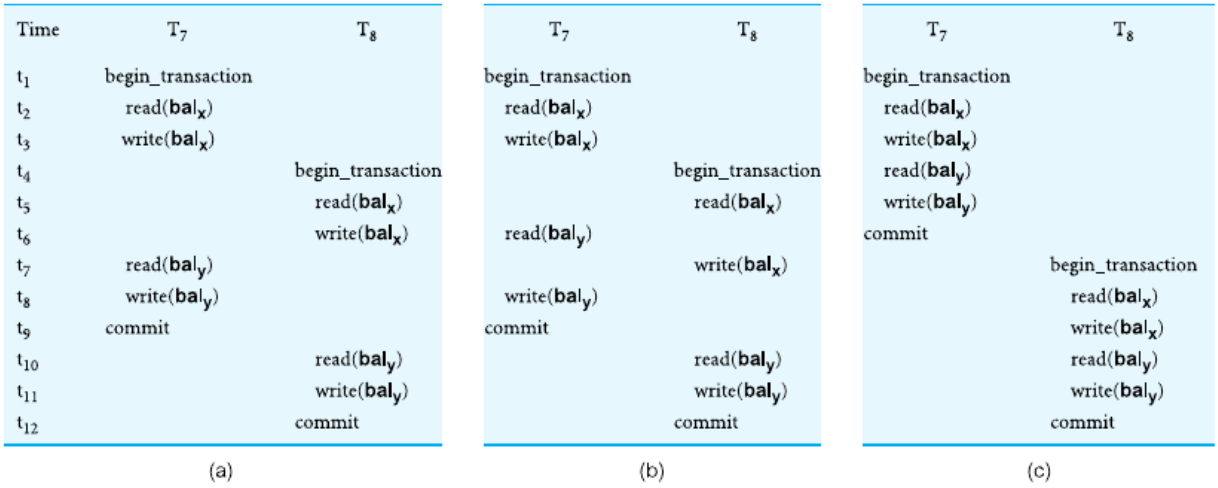
Serializabilidade é um meio de identificar as execuções de transações que garantem a consistência. Ou seja, garantir que um schedule não-sequencial produz os mesmos resultados que um sequencial (este é garantido que independentemente da ordem das transações, a base de dados fica num estado consistente). Se isto acontecer, o schedule diz-se serializavel (ou seja, que dá para andar a intercalar operações de diferentes transacções).

Serial schedule: Um schedule onde as operações de cada transacção são executadas consecutivamente sem operações intercaladas de outras transacções.

Non-serial schedule: Um schedule onde as operações de um conjunto de transacções são intercaladas.

Não importa que serial schedule é escolhido, visto que execução sequencial nunca deixa a base de dados num estado inconsistente (isto se o programador não fizer merda obviamente), embora diferentes resultados possam ser obtidos.
O objetivo da serialibilizidade é encontrar nonserial schedules que permitem transacções executar concorrentemente sem interferirem umas com as outras, e então deixar a base de dados num estado que seria alcançado com execução sequencial. Se um conjunto de transacções executa concorrentemente, dizemos que o nonserial schedule é correto se produz os mesmos resultados que um serial schedule. Este schedule é então serializável.
Para prevenir inconsistência resultante de transacções interferirem umas com as outras, é

essencial garantir serializabilidade das transacções concorrentes. Na serializabilidade, a ordem das operações de read e write é importante:
- Se duas transacções apenas lêm dados de um item, elas não entram em conflito e a ordem não é importante.
- Se duas transacções lêm ou escrevem em itens de dados separados, elas não entram em conflito e a ordem não é importante.
- Se uma transacção escreve num dado item e outra lê ou escreve nesse mesmo item, a ordem de execução é importante.



Time	T_7	T_8
t_1	begin_transaction	
t_2	read(bal_x)	
t_3	write(bal_x)	
t_4		begin_transaction
t_5		read(bal_x)
t_6		write(bal_x)
t_7	read(bal_y)	
t_8	write(bal_y)	
t_9	commit	
t_{10}		read(bal_y)
t_{11}		write(bal_y)
t_{12}		commit

(a)

T_7	T_8
begin_transaction	
read(bal_x)	
write(bal_x)	
	begin_transaction
	read(bal_x)
read(bal_y)	
	write(bal_x)
write(bal_y)	
commit	
	read(bal_y)
	write(bal_y)
	commit

(b)

T_7	T_8
begin_transaction	
read(bal_x)	
write(bal_x)	
read(bal_y)	
write(bal_y)	
commit	
	begin_transaction
	read(bal_x)
	write(bal_x)
	read(bal_y)
	write(bal_y)
	commit

(c)

O schedule S3 é um serial schedule e, visto que S1 e S2 são equivalentes a S3, S1 e S2 são schedules serializáveis. Este tipo de serialibilização é conhecida como serialibilização conflituosa.

Serializabilidade de conflitos (conflict serializability) é um schedule serializável de modo a que resolva conflitos e que possa ser serializado de outra forma que não o sequencial (por exemplo, T1 lê e escreve em X, depois T2 lê e escreve em X, depois T1 lê e escreve em Y, depois T2 lê e escreve em Y)

Para detetar isto, tem de se construir um grafo de serialização (igual ao grafo de dependencias de ArqC)

Non-conflict serializable é quando duas transações não podem ser serializadas por causa de conflitos, ficando igual ao sequencial. Por consequencia o seu grafo de precedências tem um ciclo.

Como construir o grafo:

Um grafo de precedencia serve para testar se existe conflito de serializabilidade.
- Criar um nodo para cada transacção.
- Criar um arco direcionado T1 -> T2 se T2 ler um valor escrito por T1.
- Criar um arco direcionado T1 -> T2 se T2 escrever um valor num dado item após ter sido lido por T1.
- Criar um arco direcionado T1 -> T2 se T2 escrever um valor num dado item após ter sido escrito por T1.

Se T1->T2 aparecer num grafo de precendências, então T1 tem de ser executado antes de T2. Se o grafo contiver um ciclo, então o schedule não é serizalizavel conflituosamente (quer dizer que não da para intercalar operaçoes de estas duas transacções.)

View serializability

Os valores de escrita e leitura em cada transaçao sao iguais entre schedules.

Existem outros tipos de serializabilidade que oferecem definiçoes de equivalencia de schedule menos rigidas que as oferecidas por conflict serializability. Uma dessas definiçoes é view serializability. Dois schedules S1 e S2 que consistem das mesmas operações de n transacções T1,T2,...Tn são vistos equivalentes se as próximas 3 condiçoes se verificarem:
- Para cada item de dados x, se a transacção T1 ler o valor inicial de x no schedule S1, então a transacção Ti deve ler também o valor inicial de x em S2.
- Para cada operação de leitura num item de dados x pela transação Ti no schedule S1, se o valor lido de x foi escrito pela transacção Tj, então a transacção Ti também deve ler o valor de x produzido pela transacção Tj no schedule S2.
- Para cada item de dados x, se a ultima operação de escrita em x foi efetuada pela transacção Ti no schedule S1, a mesma transacção deve efetuar a escrita final no item x no schedule S2.

Um schedule é view serializable se é equivalente em termos de view a um serial schedule. Todos os conflict serializable schedules são view serializable, embora o contrário não seja verdade.

Recuperabilidade

A serializabilidade assume que nenhuma das transaçoes falha. Se uma destas falhar é necessário voltar atrás. Se um schedule está de tal forma organizado que uma transação, ao fazer rollback, obriga a outra a fazer também (quando há dependência de valores entre as

transações, ou seja, o isolamento não foi garantido) é necessário fazer rollback às duas. Só que a se a segunda já terminou, a constraint de durabilidade foi violada. Isto é um non-recoverable schedule e não pode ser permitido.

Locking

Shared Lock: se uma transaçao tem um shared lock de um item, pode lê-lo mas não atualizá-lo.
Exclusive lock: se uma transaçao tem um lock exclusivo, pode lê-lo e atualizá-lo.

Como funciona no sistema:
A transação pede o lock dizendo o tipo deste. Se não está locked, esta transação fica com o lock. Se já está locked, o DBMS vê se os locks são compatíveis, podendo conferir ou não. Uma transação fica com o lock até o libertar explicitamente ou implicitamente através de um rollback ou abort ou commit.

Se uma transação atualiza os dados dependendo de condições, o sistema primeiro dá-lhe o lock de leitura. Se for necessário escrever, faz upgrade a esse lock para ser um lock exclusivo.

Two-phase locking (2PL)

Uma transação segue o protocolo 2PL se todas as operações de lock precedem a primeira operação de unlock.

De acordo com as regras deste protocolo, as transações têm duas fases: fase de crescimento (adquirem todos os locks que precisam) e fase de encolhimento (libertam os locks e não podem adquirir novos). Normalmente, uma transaçao adquire alguns locks, processa dados, adquire mais alguns e nunca liberta até ter chegado a uma fase em que não precisa de adquirir novos locks. Assim as transações têm estas regras:

- **devem adquirir locks antes de operar num item. podem ser de escrita ou leitura, dependendo do tipo necessário (fase de crescimento/growing phase)**
- **a partir do momento em que libertam os locks, nao podem voltar a adquirir (fase de encolhimento/shrinking phase)**
- **se for necessário fazer upgrade a locks, só podem durante a growing phase. Pode levar à espera que uma outra transação liberte o lock**

A shrinking phase deve ser cuidadosamente planeada, podendo dar origem a **Cascade rollback**. Isto é, uma transação começar durante outra com dados dependentes e a primeira fazer rollback, levando a um rollback em cadeia. Ou seja, muito trabalho deitado fora. Para evitar isto com 2PL, é necessário só libertar os locks antes de terminar de fazer commit/rollback. Assim, se uma transação depender de dados da outra já sabe o valor final dos dados e não precisará de fazer rollback.

Concorrência com índices

No caso de índice de árvore, se for um shared lock, à medida que chega a um nodo, faz lock e liberta o anterior.

Se for um lock de inserçao na árvore, como ela pode mudar toda a estrutura para trás, é necessário recolher os locks para trás.

Deadlocks

Só dá para resolver com o rollback de uma transação. Alguns métodos:

Timeouts

Uma transação que pede um lock só espera por um determinado período de tempo, fazendo rollback ao fim do timeout.

Mesmo que não haja deadlock, o DBMS assume que sim. Bastante simples, prático e usado

Prevençao

Ou usamos uma variante do 2PL em que todos os locks sao obtidos no início e não há processamento enquanto se obtém locks (2PL conservativo). O tempo que os locks ficam guardados à espera de serem usados é algo fora do controlo e pode afetar o desempenho. Pode ser bom ou mau. Por outro lado, o overhead de tentar recolher um lock e ter de espera é elevado, porque nesse caso tem de libertar todos os locks e voltar a tentar recolher.

As alternativas sao timestamps. Ha dois algoritmos: mais velho morre ou mais novo morre. Ambos dao timestamps ás transaçoes. No primeiro, se o mais velho tem de esperar por uma transaçao mais nova, espera. Se tem de esperar por uma transaçao ainda mais velha, morre. Quando voltar a correr, usa o mesmo timestamp e eventualmente vai ser a transaçao mais antiga. A outra faz o contrário. Se o mais velho tiver de esperar pelo mais novo, o mais novo morre e reinicia com o mesmo timestamp.

Deteçao

Atraves de um grafo de locks entre as transaçoes. Se tiver um ciclo, pode haver um deadlock. Este algoritmo vai correr a cada X tempo. Tem de ser bem calculado. Se for muito grande ha muito tempo de espera entre transaçoes. Se for muito pequeno causa demasiado overhead. O que se pode fazer é que a medida que nao deteta, aumenta o tempo, a medida que deteta, diminui.