Flat Transaction

Intro 1. Everything inside the BEGIN WORK and COMMIT WORK is at the same level. 2.**The transaction will either survive together with everything else (commit) 3.Or it will be rolled back with everything else (abort) – if some errors happen.**

Limitation: Such mechanism is not suitable for long transactions with multiple operations, if some errors happen in the middle of these long running transactions, and according to the rule of flat transactions we have to roll back and start these transaction again, which results in unnecessary waste of time and computational power

Flat transactions do not model many real applications.

E.g. airline booking

BEGIN WORK

S1: book flight from Melbourne to Singapore \n S2: book flight from Singapore to London

S3: book flight from London to Dublin

END WORK

From Dublin if we cannot reach our final destination instead we wish to fly to Paris from Singapore and then reach our final destination. If we roll back we need to re do the booking from Melbourne to Singapore *which is a waste.* *(solution: adding save point after executing S1, then we can save time)*

Nested Transaction

Pros:

1.Unlike flat transaction, everything will get rolled back if error happens. Within a nested Transaction, we can divide a large transaction into several small subtransactions, so that multiple subtransactions can run in parallel if there are no association between them, making the whole transaction much faster. More importantly, if some of subtransactions get aborted or we want to update or change some subtransactions purposefully, which won't affect others, enhancing the modularity.

Cons:

1.Increased Complexity: Nested transactions add complexity to transaction management and error handling. Each sub-transaction must maintain its own state, and coordinating commit or rollback across multiple levels of transactions is challenging, especially when there are failures at different levels. 2.**Higher Resource Consumption**: Managing nested transactions requires more resources, including memory and processing power. Locking, logging, and maintaining multiple transaction states can lead to higher overhead, especially in systems with limited resources.

Q In a nested transaction, a sub-transaction can commit, but the actual commit will not take place until all its ancestor transactions commit. Then why is a nested transaction still useful?

Although such rule of commit is a constrain on subtransaction, but nested transaction is still useful because it allows each sub-transaction to perform isolated work and handle partial failures independently. This modularity improves error handling and recovery since a failure in one sub-transaction doesn't require rolling back the entire main transaction immediately; instead, only the specific sub-transaction can be retried or rolled back. Additionally, nested transactions enable finer-grained concurrency control by allowing certain sub-transactions to proceed in parallel, improving overall efficiency within a complex transaction structure.

Transaction

For banking deposit: A transfer 100 to B: 1.A-100; 2.B+100; 3.log to record A--->B **happen as a whole or no action happen**

Atomicity - All changes to data are performed as if they are a single operation. That is, all the changes are performed, or none of them are.

Consistency - Data is in a 'consistent' state when a transaction starts and when it ends – in other words, any data written to the database must be valid according to all defined rules (e.g., no duplicate student ID, no negative fund transfer, etc.)

Isolation - transaction are executed as if it is the only one in the system.

Durability - the system should tolerate system failures and any committed updates should not be lost.

Unprotected actions - No ACID property; **Protected actions** - These actions are not externalised before they are completely done. These actions are controlled and can be rolled back if required. These have ACID property.

Real actions - These are real physical actions once performed cannot be undone. In many situations, atomicity is not possible with real actions.

A flat transaction with save-points has the following statements. If condition1 is true once and the final commit is successful, what will be printed as the value of the count?

BEGIN WORK

count = 10

SAVE WORK 1 count:10

count = count+10 20

SAVE WORK 2 count:20

count = count+5 25

SAVE WORK3 count:25

count = count+5 30

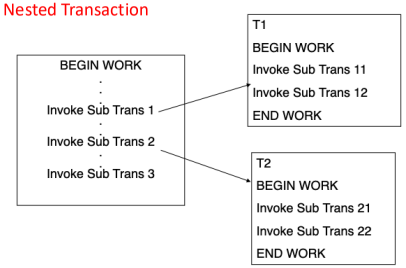
If (condition1) ROLLBACK WORK(3) :count:25

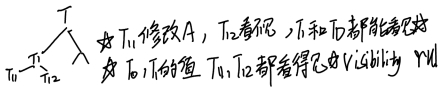
count = count-1 25-1=24

print count

COMMIT WORK

Nested Transaction





Commit rule

- A sub-transaction can either commit or abort, however, commit cannot take place unless the parent itself commits. (Sub的祖先提交了他才能提交)
- Sub-transactions have Atomicity, Consistency, and Isolation properties but not have Durability property unless all its ancestors commit.(祖先不提交的话: sub只满足ACI，不满足D)
- Commit of a sub transaction makes its results available only to its parents

Roll back rules:

If a sub-transaction rolls back all its children are forced to roll back(一损俱损)

Visibility rules

1.Changes made by a sub-transaction are visible to the parent only when the sub-transaction commits. 2.All objects of parent are visible to its children.

Implication of this is that the parent should not modify objects while children are accessing them. This is not a problem as parent is not run in parallel with its children.

Concurrency Problem: multiple transactions working on the same shared variable, running concurrently

For correct execution, we need to impose exclusive access to the shared variable counter by Task1 and Task2.

Shared counter = 100;

Task1/Trans/Process/Thread	Task2/Trans/Process/Thread
counter = counter +10;	counter = counter +30;

Task1 and Task2 run concurrently. What are the possible values of counter after the end of Task1 and Task2?

Note: == means equals.

a) counter == 110	b) counter == 130	c) counter == 140;
Sequence of actions	Sequence of actions	Sequence of actions
T1: Reads counter == 100	T1: Reads counter == 100	T1: Reads counter == 100
T2: Reads counter == 100	T2: Reads counter == 100	T1: Writes counter == 100+10
T2: Writes counter == 100+30	T1: Writes counter == 100+10	T2: Reads counter == 110
T1: Writes counter == 100+10	T2: Writes counter == 100+30	T2: Writes counter == 110+30

Concurrency Problem

– Spin locks (需要 hard ware support; using atomic lock/unlock instructions) – most commonly used

Pros: All modern processors do support some form of spin locks;
Executed using atomic machine instructions such as test and set or compare and swap; Need hardware support
Use busy waiting; Does not depend on number of processes; Very efficient for low lock contentions – all DB systems use them

Implementation of Atomic operation: compare and swap in spin lock for exclusive access

code may have lost values

```
temp = counter +1; //unsafe to increment a shared counter
counter = temp;  //this assignment may suffer a lost update
```

Implementation of Atomic operations: test and set

```
T1
/*acquire lock*/
while (!testAndSet( &lock );
    /*Xlock granted*/
//exclusive access for T1;
counter  = counter+1;
/* release lock*/
lock = 1;
```

```
T2
/*acquire lock*/
while (!testAndSet( &lock );
    /*Xlock granted*/
//exclusive access for T2;
counter  = counter+1;
/* release lock*/
lock = 1;
```

Why we prefer CAS, rather than TAS? ANS: 1.Pros for CAS: it ask for exclusive lock only when 'write' operation is required.
So if in the scenario where multiple transactions only execute reading operation simultaneously (browsing products on online shopping platform) but not do any writing operation on it, using TAS will prevent any other transaction from reading the resources as well. 2.For reading and writing if we use TAS, which means one transaction has to start and completely finish its operations on the shared variable, then only the second transaction will be able to start (busy waiting). 3. However, if both transactions are just reading that value, reading does not conflict what happening for other transaction if both of them just reading and not making any changes to the share variable, CAS is allow multiple transaction to run concurrently for that reading part.

TP monitor

The main function of a TP monitor is to **integrate other system components and manage resources**.
1.TP monitors manage the transfer of data between clients and servers
2.Breaks down applications or code into transactions and ensures that all databases are updated properly
3.It also takes appropriate actions if any error occurs
Example: If A has just bought last product on a online shopping platform, but next second B is about to checkout the same item, such scenario needed to be managed (rolled back the B's operation) by TP monitor, to make DB's change is consistent.

TP monitor service

Heterogeneity: If the application needs access to different DB systems, local ACID properties of individual DB systems is not sufficient. Local TP monitor needs to interact with other TP monitors to ensure the **overall ACID** property. (Two-phase commit in distributed DB)
Control: If the application communicates with other remote processes, the local TP monitor should maintain the communication status among the processes to be able to recover from a crash.

Terminal management: Since many terminals run client software, the TP monitor should provide appropriate ACID property between the client and the server processes.

Presentation service: this is similar to terminal management in the sense it has to deal with different presentation (user interface) software -- e.g. X-windows

Context management: E.g. maintaining the sessions etc.

Start/Restart: There is no difference between start and restart in TP based system.

Concurrency Problem solution

Different ways for concurrency control

– Dekker's algorithm (using code) Pros:Dekker's algorithm needs almost no hardware support although it needs atomic reads and writes to main memory. That is exclusive access of one time cycle of memory access time!
Cons: The code is very complicated to implement if more than two transactions/process are involved; Harder to understand the algorithm for more than two process takes lot of storage space; Uses busy waiting: if resource has been locked by T1, then T2 cannot do anything but just keeps waiting in the loop for the resource to be released, hampers the overall performance of transaction; Efficient if the lock contention (that is frequency of access to the locks) is low.

Dekker's algorithm

```
int c1, c2, turn = 1; /* global variable*/
```

```
T1
{ some code T1}
/* T1 wants exclusive access to the resource
   and we assume initially c1 = = 0*/
c1 = 1; turn = 2;
repeat until { c2 == 0 or turn == 1}
/* Start of exclusive access to the
   shared resource (successfully
   changed variables) */
use the resource
counter = counter+1;
/* release the resource */
c1 = 0;
{some other code of T1}
```

```
T2
{ some code T2}
/* T2 wants exclusive access to the resource
   and we assume initially c2 = = 0 */
c2 = 1; turn = 1;
repeat until { c1 == 0 or turn == 2}
/* Start of exclusive access to the
   shared resource */
use the resource
counter = counter+1;
/* release the resource */
c2 = 0 ;
{some other code of T2}
```

– OS supported primitives (through interruption call) - expensive, independent of number of processes, machine independent

Convoy avoiding semaphore

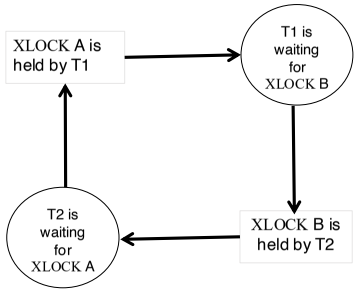
The Semaphores implementation may result a long list of waiting processes (convoy), if one or couple of them are long-running processors then the one that make request later has to wait for longer time in the queue.

Solution: once one transaction is done, the lock on the shared variable gets released, every process in the waiting list is weaken up, then they all re-execute the routine for acquiring semaphore (recall the resources again), which leads to the order of waiting transaction not be the same as the queue before. (i.e., the order get reshuffled), reducing the probability of long waiting queue occurs, and all current transaction can have another chance to ask disk for resources.

Deadlocks

- In a deadlock, each process in the deadlock is waiting for another member to release the resources it wants.

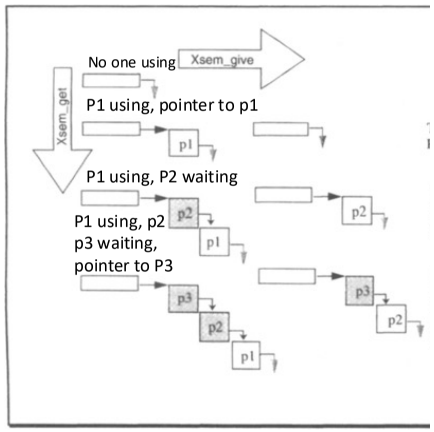
T1	T2
Begin	Begin
XLOCK(A)	XLOCK(B)
Write to A	Write to B
XLOCK(B)	XLOCK(A)
Write B	Write A
Unlock (A)	Unlock (A)
Unlock (B)	Unlock (B)
end	end



Spin lock – Compare and Swap: improve overall # of operations that are being executed in the system=> run more operations concurrently.

```
boolean cs(int *cell, int *old, int *new)
{/* the following is executed atomically*/
if (*cell == *old)      { *cell = *new; return TRUE;}
else { *old = *cell; return FALSE;}
}
```

Semaphores



Semaphores

After usage, the owner process wakes up the oldest process in the queue (first in, first out scheduler)

Pointer is a mechanism maintaining the queue of processors in the waiting.

Deadlock - Solutions

Have enough resources so that no waiting occurs – not practical;Do not allow a process to wait, simply rollback after a certain time. This can create live locks which are worse than deadlocks;Linearly order the resources and request of resources should follow this order, i.e., a transaction after requesting i^{th} resource can request j^{th} resource if $j > i$. This type of allocation guarantees no cyclic dependencies among the transactions.(在全局保持这个顺序)

1.Pre-declare all necessary resources and allocate in a single request. 2. Periodically check the resource dependency graph for cycles. If a cycle exists - rollback (i.e., terminate) one or more transaction to eliminate cycles (deadlocks). The chosen transactions should be cheap (e.g., they have not consumed too many resources).3. Allow waiting for a maximum time on a lock then force Rollback. Many successful systems (IBM, Tandem) have chosen this approach. QMany distributed database systems maintain only local dependency graphs and use time outs for global deadlocks.

Pros: 1.simple and easy to implement: we don't have to monitor state of transaction instantly for detecting deadlock cycles, which may cause high overhead. 2. Scalability:For large system with many concurrent transactions, deadlock detection algorithm is costly and complex. Timeout based strategy provides a scalable solution, allowing system to handle a large # of trans without tracking complex dependency.(adapt to many system.)

Cons:1. Potential for Unnecessary Rollbacks: This approach can cause transactions to roll back even if they are not involved in a deadlock, simply because they exceeded the wait time. This can lead to wasted work and reduced efficiency, especially in cases of high contention. 2.Difficult to Set Optimal Timeout: If the timeout is set too short, transactions may frequently roll back unnecessarily. If it's too long, actual deadlocks may take too long to resolve, leading to resource contention and delayed processing. 3. Increased System Overhead Due to Rollbacks: Frequent rollbacks can increase the system's workload, as it has to repeatedly restart transactions that could have otherwise completed, impacting overall throughput and response times.
Overall, while simple and resource-efficient, the timeout-based strategy can lead to inefficiency and performance degradation in high-concurrency environments or when transactions are long-running.