CS 344 Database Management Systems

Lecture: 1st November 2011

Submitted by:

Prateek Gupta(08012313)

Navdeep Singh(08012308)

In the earlier lectures we have been discussing the problems faced while we are doing transactions. It was stated in previous lectures that when we promise transactions, we need to ensure what are ubiquitously called ACID properties.

A – Atomicity

C – Consistency

I – Isolation

D – Durability

While we have already discussed atomicity, we will now discuss the others. This lecture opens with a discussion on the problem of consistency during transactions. The problem of consistency occurs when a particular change in a data entry does not reflect in the main memory.

Consider a scenario where the transaction is partially done. Then, what do we do? One possible solution could be to restart the transaction whenever such cases occur. Essentially, the logic of “everything or nothing” works here.

_____Begin of transactions

.

.

. say ‘n’

.

.

_____End of transactions

Problem of Isolation

The property of Isolation has a simple demand that one transaction should not affect other transactions.

Now, when we promise transactions, which one of the ACID properties can be ignored/omitted initially?

Here, we can initially choose to ignore Isolation as we can simply guarantee Isolation by allowing Single Transaction at a time.

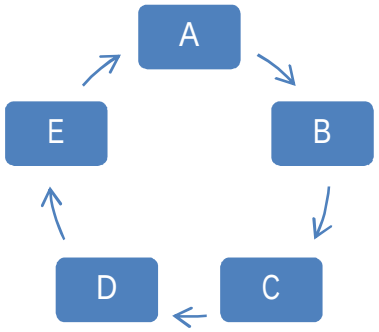
But, such a system will not be a favored one. The user would very much want his system to allow multiple transactions. But, multiple transactions will have their own problems which are essentially the same problems one faces while dealing with multiple processes in Operating Systems which are:-

Deadlocks

Deadlock is a state where each process is waiting for some other process in order to acquire the resource. Commercial Operating Systems do not do much about it. They just assume that the number of deadlocks would be less and so they just overlook them and the user has to restart the machine whenever deadlocks occur. In fact, DBMS also doesn’t do anything inherently. But in DBMS, the

implementation is design-based. Some people might not be in favor of just "overlooking" the deadlocks. Then what is the way out?

1. Deadlock Identification: The first step is to identify the deadlock. Typically, finding a cycle in the Resource-Process graph does the trick in OS. Similar approach can be used in DBMS as well.



In the figure, there is a deadlock as A is waiting on B which is waiting on C and so on. We can easily see that there is a cycle in the graph and hence deduce that there is a deadlock. But what if we are not able to figure out the cycle every time or say the overhead for checking for a cycle is too much, then what do we do? We need to have some sort of Thumb rules:

- Some processes are waiting for a long time
- Some resources are not free for a long time

Note: The deadlock "check" can't be done very frequently since there is a significant overhead in doing so.

2. Breaking the Deadlock: We have identified the cycle. Now, what do we do? We need to break the cycle by selecting a victim process to kill. Now that choice can be made in several ways:
    - We can see the time already spent by that particular process. Killing a process which has spent only 2 minutes instead of a process that has spent 5 hours seems quite rational.
    - We can see how many number of resources each process requires and then kill a process which requires maximum number of resources

because that will reduce the probability of occurrence of deadlocks in future.

3. Starvation: Now we have made a choice to kill a particular process in order to break the deadlock. But we need to make sure that our choice doesn't lead to starvation (a particular process never gets the resource). We again have different ways of avoiding starvation which are:
    - We can increase the priority of the processes which are selected as the victims. This will make sure that at some point of time, this particular process gets the resource.
    - We can use Time-stamping

Note: The DBMS equivalent of "resources" is the data itself.

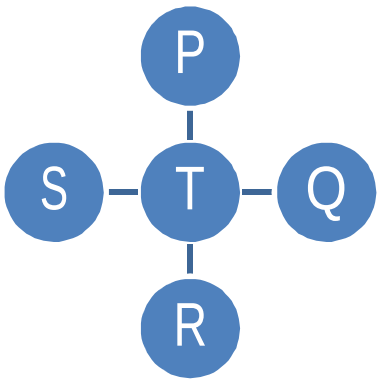
Concurrency control in Transactions

We have seen above, how to identify and break the deadlocks. But, what if we decide to "prevent" deadlocks? We will see that in a while.

For transaction recovery, we need some log information. Now, that log information can be "general" or "detailed" as per requirement.

Suppose multiple transactions are sharing the resources, what are the simple recovery schemes?

Cascaded rollback: When one transaction fails, rollback other transactions sharing the resources.



Like suppose, the figure represents a process relation graph. Then suppose there is problem with R, immediately rollback T, since it shares a resource with R.

And for reducing the cascading, we need detailed log information. Here, we have taken care of the Durability property but what about others, say, Consistency? For consistency, we just need to make sure that the transaction begins and ends with a consistent state.