Lecture Notes for class held on 2nd Nov, 2011.

Submitted by:

Paranjay Srivastava **Ankit Aggarwal**

08012311 **08012305**

There are four properties which should be guaranteed by a DBMS when performing transactions.

1. **Atomicity**: Users should be able to regard the execution of each transaction as atomic, either all actions are carried out or none are.
2. **Consistency:** Each transaction, run by itself with no concurrent execution of other transactions, must preserve the consistency of the database. The DBMS assumes that it holds for each transaction. Ensuring this property of a transaction is the responsibility of the user.
3. **Isolation:** Users should be able to understand a transaction without considering the effect of other concurrently executing transactions.
4. **Durability:** Once the DBMS informs the user that a transaction has been successfully completed; its effects should persist even if the system crashes before all its changes are reflected on disk.

Isolation can be trivially achieved by running each transaction separately on the DBMS. But for better performance and utilization of resources, Transactions are done in parallel in a DBMS. Typically there is a limit to the number of transactions which can be run in parallel. If there are too many transactions running, it will lead to something similar to "Thrashing" in Operating Systems. Therefore a choice needs to be made as to how many transactions are allowed to run in parallel.

A transaction can be viewed as a series of reads and writes. Typically, concurrent reads do not cause any problem; problems arise when concurrent writes happen. Lock-based protocols are used to solve the problems of concurrent writes. They are generally very easy to understand and implement. We shall now look at various lock based protocols.

Suppose we have two transactions running in parallel which need to perform the following operations as mentioned in the table below.

T1	**T2**
R_A	R_A
W_A	R_B
	W_A

So T1 goes and requests the DBMS for read access to A. Then it requests for write access to A. Once the DBMS has provided the necessary permissions, T1 writes to A. While T1 is writing if T2 requests the DBMS for access to A, it is asked to wait. Once T1 successfully commits the DBMS there are two approaches, which it can take to intimate T2. One approach is that T2 constantly polls the DBMS and checking whether the resource is available or not, second approach is that the DBMS goes and wakes up T2 and grants it the available permissions. The second approach is preferable because polling typically has a lot of overhead.

We now discuss the allocation of read and write locks in parallel transactions.

	Read	Write
Read	Yes	No
Write	No	No

The above table indicates that we allow two read locks in parallel, but once a write lock is acquired, another write/read lock is not given to another transaction.

We have multiple transactions running on the system. When a transaction requests read/write access, we see if it can be given using the above matrix, if not, then the transaction is asked to wait! This leads to the possibility of another problem, that of deadlock.

A scenario in which deadlock occurs.

A	B
W_A	W_B
W_B	W_A

In order to prove that a deadlock exists, we can use a resource-wait graph as in operating systems.

To deal with the problem of deadlock we have several approaches.

1. Lazy approach: assume deadlocks don't exist.
2. 2 phase locking
3. Strict 2 phase locking
4. Rigorous 2 phase locking.

2 phase locking

The life of a transaction is divided into two phases, in the first phase it acquires locks and in the second phase it releases the locks.

Figure for simple two phase locking

A variant of 2PL, strict two-phase locking, makes the transaction release its read locks before releasing the write locks.

Another variant of 2PL, rigorous two-phase locking, makes the transaction release all its locks at the end once all its computation is done.

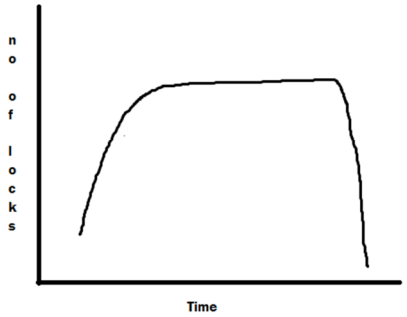
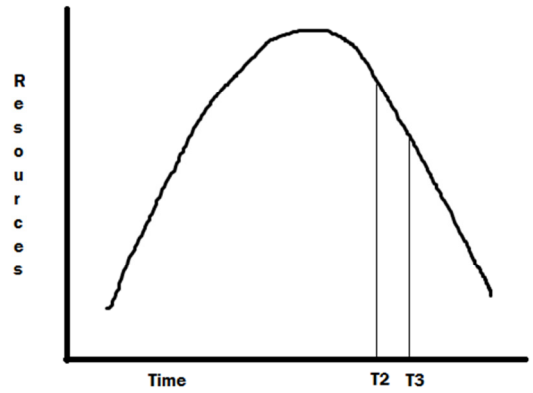


Figure for rigorous two phase locking

Simple two phase locking can result in deadlocks but makes better use of resources.
In rigorous two-phase locking resource utilization is less as the resources are held till the end of the transaction even if they are not required. In all the versions of two phase locking, there is not guarantee that deadlocks won't occur.
In simple two phase locking, the probability of deadlocks occurrence is less as it releases resources in between.
We still define and use rigorous two phase locking because it prevents cascading rollbacks.



Resource-Time graph of a Transaction (T1) following simple two-phase locking.

Consider T1 which acquires read and write locks for resources, makes some modifications and then writes them back and releases the read locks. These written values are read by other transactions T2 and T3. Now if T1 aborts, T2 and T3 also need to restart because the values which they read are invalid. This is called cascading rollbacks and it is prevented in rigorous two-phase locking as there is no release of locks in between.

So the choice now is between 2-phase locking or rigorous two phase locking. One provides a better utilization of resources along with a lower probability of deadlocks and the other avoids cascading rollbacks.
There is a tradeoff between probability of failure of a transaction and occurrence of deadlocks. Hence there is no clear winner!

Now going back to the relational modeling we typically have cells, rows, columns and tables. We can lock a cell, a row, a column or the whole table.
Locking a table leads to worst utilization of resources. But the number of locks used is less. Also if one transaction accesses only one table then the probability of deadlock is zero. Locking a cell provides most efficient use of resources.

Another method of concurrency control is to make copies of the records and make it available to the transaction. So if 100 transactions request for A, we create 100 copies of A and give each transaction a copy. So there is no concept of waiting and hence no deadlock can occur. The problem with this scheme is, to choose which transaction should be allowed to write when more than one transaction has made a modification to A.
One strategy (first committer wins) says that whoever commits first is allowed to make the write, and when another transaction comes, it is sent back with the fresh value to re do its work.