# Lock-I

Hi! In this chapter I am going to discuss with you about Locks in DBMS.

## Locking

Locking is a common technique by which a database may synchronize execution of concurrent transactions. Using locking a transaction can obtain exclusive or shareable access rights (called *locks*) to an object. If the access provided by the lock is shareable, the lock is often called a *shared lock* (sometime called a *read* lock). On the other hand if the access is exclusive, the lock is called an *exclusive lock* (sometime called a *write* lock). If a number of transactions need to read an object and none of them wishes to write that object, a shared lock would be the most appropriate. Of course, if any of the transactions were to write the object, the transaction must acquire an exclusive lock on the object to avoid the concurrency anomalies that we have discussed earlier. A number of locking protocols are possible. Each protocol consists of

1. a set of locking types
2. a set of rules indicating what locks can be granted concurrently
3. a set of rules that transactions must follow when acquiring or releasing locks

We will use the simplest approach that uses shared and exclusive locks, a transaction would set a read lock on the data item that it reads and exclusive lock on the data item that it needs to update. As the names indicate, a transaction may obtain a shared lock on a data item even if another transaction is holding a shared lock on the same data item at the same time. Of course, a transaction cannot obtain a shared or exclusive lock on a data item if another transaction is holding an exclusive lock on the data item. The shared lock may be upgraded to an exclusive lock (assuming no other transaction is holding a shared lock on the same data item at that time) for items that the transaction wishes to write. This technique is sometime also called *blocking* because if another transaction requests access to an exclusively locked item, the lock request is denied and the requesting transaction is blocked.

A transaction can hold a lock on one or more items of information. The data item must be unlocked before the transaction is completed. The locks are granted by a database system software called the *lock manager* which maintains information on all locks that are active and controls access to the locks. As noted earlier, several modes of locks may be available. For example, in shared mode, a transaction can read the locked data item but cannot update it. In exclusive lock mode, a transaction has exclusive access to the locked data item and no other transaction is allowed access to it until the lock is released. A transaction is allowed to update the data item only as long as an exclusive lock is held on it.

If transactions running concurrently are allowed to acquire and release locks as data item is read and updated, there is a danger that incorrect results may be produced. Consider the following example:

| Transaction 1 | Time | Transaction 2 |
|---|---|---|
| XL(A) | 1 | - |
| Read (A) | 2 | - |
| A:= A - 50 | 3 | - |
| - | 4 | SL(B) |
| Write(A) | 5 | - |
| - | 6 | Read(B) |
| UL(A) | 7 | - |
| - | 8 | UL(B) |
| - | 9 | SL(A) |
| XL(B) | 10 | - |
| - | 11 | Read (A) |
| Read(B) | 12 | - |
| - | 13 | UL(A) |
| B:= B + 50 | 14 | - |
| - | 15 | Display (A+B) |
| Write (B) | 16 | - |
| UL(B) | 17 | - |

Table 8 - Unserializable Schedule using Locks

XL (called exclusively lock) is a request for an exclusive lock and UL is releasing the lock (sometimes called unlock). SL is a request for a shared lock. In the example above, the result displayed by transaction 2 is incorrect because it was able to read *B* before the update but read *A* only after the update. The above schedule is therefore not serializable. As discussed earlier, the problem with above schedule is inconsistent retrieval.

The problem arises because the above schedule involves two RW-conflicts and the first one that involves *A* involves Transaction 1 logically appearing before Transaction 2 while the second conflict involving *B* involves Transaction 2 logically appearing before Transaction 1.

To overcome this problem a *two-phase* locking ( *2PL*) scheme is used in which a transaction cannot request a new lock after releasing a lock. Two phase locking involves the following two phases:

1. **Growing Phase** ( *Locking* Phase) -- During this phase locks may be acquired but not released.
2. **Shrinking Phase** ( *Unlocking* Phase) - During this phase locks may be released but not acquired.

In summary, the main feature of locking is that conflicts are checked at the *beginning* of the execution and resolved by *waiting*.

Using the two-phase locking scheme in the above example, we obtain the following schedule by modifying the earlier schedule so that all locks are obtained before any lock is released:

| Transaction 1 | Time | Transaction 2 |
|---|---|---|
| XL(A) | 1 | - |
| Read (A) | 2 | - |
| A:= A - 50 | 3 | - |
| - | 4 | SL(B) |
| Write(A) | 5 | - |
| - | 6 | Read(B) |
| XL(B) | 7 | - |
| UL(A) | 8 | - |
| - | 9 | SL(A) |
| - | 10 | UL(B) |
| - | 11 | Read (A) |
| Read(B) | 12 | - |
| - | 13 | UL(A) |
| B:= B + 50 | 14 | - |
| - | 15 | Display (A+B) |
| Write (B) | 16 | - |
| UL(B) | 17 | - |

Table 9 - A Schedule using Two-Phase Locking

Unfortunately, although the above schedule would not lead to incorrect results, it has another difficulty: both the transactions are blocked waiting for each other; a classical deadlock situation has occurred! Deadlocks arise because of circular wait conditions involving two or more transactions as above. A system that does not allow deadlocks to occur is called *deadlock-free*. In two-phase locking, we need a deadlock detection mechanism and a scheme for resolving the deadlock once a deadlock has occurred. We will look at such techniques in Section 5.

The attraction of the two-phase algorithm derives from a theorem which proves that the two-phase locking algorithm always leads to serializable schedules that are equivalent to serial schedules in the order in which each transaction acquires its last lock. This is a sufficient condition for serializability although it is not necessary.

To show that conditions of serializability are met if 2PL is used we proceed as follows. We know that a test of serializability requires building a directed graph in which each of the transactions is represented by one node and an edge between $T_i$ and $T_j$ exists if any of the following conflict operations appear in the schedule:

1. $T_i$ executes WRITE( *X*) before $T_j$ executes READ( *X*), or
2. $T_i$ executes READ( *X*) before $T_j$ executes WRITE( *X*)
3. $T_i$ executes WRITE( *X*) before $T_j$ executes WRITE( *X*).

A schedule will not be serializable if a cycle is found in the graph. We assume a simple concurrent execution of two transactions and assume that the graph has a cycle. The cycle is possible only if there is a edge between $T_i$ and $T_j$ because one of the above conditions was met followed by the schedule meeting one of the following conditions:

1. $T_j$ executes WRITE( *Y*) before $T_i$ executes READ( *Y*), or
2. $T_j$ executes READ( *Y*) before $T_i$ executes WRITE( *Y*)
3. $T_j$ executes WRITE( *Y*) before $T_i$ executes WRITE( *Y*).

These two sets of conditions provide a total of nine different combinations of conditions which could lead to a cycle. It can be shown that none of these combinations are possible. For example, assume the following two conditions have been satisfied by a schedule:

1. $T_i$ executes WRITE( *X*) before $T_j$ executes READ( *X*),
2. $T_j$ executes WRITE( *Y*) before $T_i$ executes WRITE( *Y*).

For these conditions to have met, $T_i$ must have had an XL(X) which it would release allowing $T_j$ to acquire SL(X). Also, for the second condition to have met, $T_j$ must have had an XL(Y) which it would release allowing $T_i$ to acquire XL(Y). This is just not possible using 2PL. Similarly, it can be shown that none of the nine combinations of conditions are possible if 2PL is used.

Several versions of two-phase locking have been suggested. Two commonly used versions are called the *static* two-phase locking and the *dynamic* two-phase locking. The static technique basically involves locking all items of information needed by a transaction before the first step of the transaction and unlocking them all at the end of the transaction. The dynamic scheme locks items of information needed by a transaction only immediately before using them. All locks are released at the end of the transaction. [which is better??]

To use the locking mechanism, the nature and size of the individual data item that may be locked must be decided upon. A lockable data item could be as large as a relation or as small as an attribute value of a tuple. The lockable item could also be of a size in between, for example, a page of a relation or a single tuple. Larger the size of the items that are locked, the smaller the concurrency that is possible since when a transaction wants to access one or more tuples of a relation, it is allowed to lock the whole relation and some other transaction wanting to access some other tuples of the relation is denied access. On the other hand, coarse granularity reduces the overhead costs of locking since the table that maintains information on the locks is smaller. Fine granularity of the locks involving individual tuple locking or locking of even smaller items allows greater concurrency at the cost of higher overheads. Therefore there is a tradeoff between level of concurrency and overhead costs. Perhaps a system can allow variable granularity so that in some situations, for example when computing the projection of a relation, the whole relation may be locked while in other situations only a small part of the relation needs locking.

## Timestamping Control

The two-phase locking technique relies on locking to ensure that each interleaved schedule executed is serializable. Now we discuss a technique that does not use locks and works quite well when level of contention between transactions running concurrently is not high. It is called Timestamping.

Timestamp techniques are based on assigning a unique *timestamp* (a number indicating the time of the start of the transaction) for each transaction at the start of the transaction and insisting that the schedule executed is always serializable to the serial schedule in the chronological order of the timestamps. This is in contrast to two-phase locking where any schedule that is equivalent to some serial schedule is acceptable.

Since the scheduler will only accept schedules that are serializable to the serial schedule in the chronological order of the timestamps, the scheduler must insist that in case of conflicts, the junior transaction must process information only after the older transaction has written them. The transaction with the smaller timestamp being the older transaction. For the scheduler to be able to carry this control, the data items also have read and write timestamps. The read timestamp of a data item X is the timestamp of the youngest transaction that has read it and the write timestamp is the timestamp of the youngest transaction that has written it. Let timestamp of a transaction be TS(T).

Consider a transaction $T_1$ with timestamp = 1100. Suppose the smallest unit of concurrency control is a relation and the transaction wishes to read or write some tuples from a relation named *R*. Let the read timestamp of *R* be *RT(R)* and write timestamp be *WT(R)*. The following situations may then arise:

1. $T_1$ wishes to read *R*. The read by $T_1$ will be allowed only if $WT(R) < TS(T_1)$ or $WT(R) < 1100$ that is the last write was by an older transaction. This condition ensures that data item read by a transaction has not been written by a younger transaction. If the write timestamp of *R* is larger than 1100 (that is, a younger transaction has written it), then the older transaction is rolled back and restarted with a new timestamp.
2. $T_1$ wishes to read *R*. The read by $T_1$ will be allowed if it satisfies the above condition even if $RT(R) > TS(T_1)$, that is the last read was by a younger transaction. Therefore if data item has been read by a younger transaction that is quite acceptable.

3. $T_1$ wishes to write some tuples of a relation *R*. The write will be allowed only if read timestamp $RT(R) < TS(T_1)$ or $RT(R) < 1100$, that is the last read of the transaction was by an older transaction and therefore no younger transaction has read the relation.
4. $T_1$ wishes to write some tuples of a relation *R*. The write need not be carried out if the above condition is met and if the last write was by a younger transaction, that is, $WT(R) > TS(T_1)$. The reason this write is not needed is that in this case the younger transaction has not read *R* since if it had, the older transaction $T_1$ would have been aborted.

If the data item has been read by a younger transaction, the older transaction is rolled back and restarted with a new timestamp. It is not necessary to check the write (??) since if a younger transaction has written the relation, the younger transaction would have read the data item and the older transaction has old data item and may be ignored. It is possible to ignore the write if the second condition is violated since the transaction is attempting to write obsolete data item. [expand!]

Let us now consider what happens when an attempt is made to implement the following schedule:

???

We assume transaction $T_1$ to be older and therefore $TS(T_1) < TS(T_2)$.

Read (A) by transaction $T_1$ is permitted since A has not been written by a younger transaction. Write (A) by transaction $T_1$ is also permitted since A has not been read or written by a younger transaction. Read (B) by transaction $T_2$ is permitted since *B* has not been written by a younger transaction. Read (B) by transaction $T_1$ however is not allowed since *B* has been read by (younger) transaction $T_2$ and therefore transaction $T_1$ is rolled back. [needs to be read again and fixed]

Refer to page 383 of Korth and Silberchatz for a schedule that is possible under the timestamping control but not under 2PL

**DEADLOCKS**

A deadlock may be defined as a situation in which each transaction in a set of two or more concurrently executing transactions is blocked circularly waiting for another transaction in the set, and therefore none of the transactions will become unblocked unless there is external intervention.

A deadlock is clearly undesirable but often deadlocks are unavoidable. We therefore must deal with the deadlocks when they occur. There are several aspects of dealing with deadlocks: one should prevent them if possible (deadlock *prevention*), detect them when they occur (deadlock *detection*) and resolve them when a deadlock has been detected (deadlock *resolution*). Deadlocks may involve two, three or more transactions. To resolve deadlocks, one must keep track of what transactions are waiting and for which transaction they are waiting for.

Once a deadlock is identified, one of the transactions in the deadlock must be selected and rolled back (such transaction is called a *victim*) thereby releasing all the locks that that transaction held breaking the deadlock.

Since locking is the most commonly used (and often the most efficient) technique for concurrency control, we must deal with deadlocks. The deadlocks may either be prevented or identified when they happen and then resolved. We discuss both these techniques.

**Deadlock Prevention**

Deadlocks occur when some transactions wait for other transactions to release a resource and the wait is circular (in that $T_1$waits for $T_2$which is waiting for $T_4$which in turn is waiting for $T_1$or in the simplest case $T_1$waits for $T_2$and $T_2$for $T_1$). Deadlocks can be prevented if circular waits are eliminated. This can be done either by defining an order on who may wait for whom or by eliminating all waiting. We first discuss two techniques that define an ordering on transactions. [Further info .. Ullman page 373]

**Wait-die Algorithm**

When a conflict occurs between $T_1$and $T_2$($T_1$ being the older transaction), if $T_1$possesses the lock then $T_2$(the younger one) is not allowed to wait. It must rollback and restart. If however $T_2$possessed the lock at conflict, the senior transaction is allowed to wait. The technique therefore avoids cyclic waits and generally avoids *starvations* (a transaction's inability to secure resources that it needs). A senior transaction may however find that it has to wait for every resource that it needs, and it may take a while to complete.

**Wound-die Algorithm**

To overcome the possibility of senior transaction having to wait for every item of data that it needs, the wound-die scheme allows a senior transaction to immediately acquire a data item that it needs and is being controlled by a younger transaction. The younger transaction is then restarted.

**Immediate Restart**

In this scheme, no waiting is allowed. If a transaction requests a lock on a data item that is being held by another transaction (younger or older), the requesting transaction is restarted immediately. This scheme can lead to starvation if a transaction requires several popular data items since every time the transaction restarts and seeks a new item, it finds some data item to be busy resulting in it being rolled back and restarted.

**Deadlock Detection**

As noted earlier, when locking is used, one needs to build and analyse a waits-for graph every time a transaction is blocked by a lock. This is called *continuous detection*. One may prefer a *periodic detection* scheme although results of experiments seem to indicate that the continuous scheme is more efficient.

Once a deadlock has been detected, it must be resolved. The most common resolution technique requires that one of the transactions in the waits-for graph be selected as a *victim* and be rolled back and restarted. The victim may be selected as

1. randomly
2. the youngest transaction since this is likely to have done the least amount of work.
3. the one that has written the least data back to the database since all the data written must be undone in the rollback.
4. the one that is likely to affect the least number of other transactions. That is, the transaction whose rollback will lead to least other rollbacks (in cascade rollback).
5. the one with the fewest locks.

One experimental study has suggested that the victim with the fewest locks provides the best performance. Also it has been suggested that if deadlock prevention with immediate restart is to be used, the database performance suffers.

## EVALUATION OF CONTROL MECHANISMS

A number of researchers have evaluated the various concurrency control mechanisms that have been discussed here. It is generally accepted that a technique that minimizes the number of restarts (like the two-phase locking technique) is superior when the number of conflicts is high. The timestamping and the optimistic algorithms usually performed much worse under high conflict levels since they wasted the most resources through restarts. Deadlock prevention techniques like those based on immediate restart tend to make the database slow down somewhat. One therefore must have deadlock detection technique in place. At low loads, it doesn't really matter much which technique is used although some studies show that the optimistic techniques then are more efficient. It is of course possible to use more than one technique in a system. For example, two-phase locking may be combined with optimistic technique to produce hybrid techniques that work well in many situations.

Review Question

1. What is two phase locking
2. Define
    - Page locking
    - Cluster locking
    - Class or table locking
    - Object or instance locking

**Source**

1. Agarwal, R. and DeWitt, D. (1985), "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation", ACM TODS, Vol. 10, pp. 529-564.
2. Agarwal, R., Carey, M. J. and McWoy, L. (1988?), "The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems", IEEE Trans. Software Engg., pp. ??
3. Agarwal, R., Carey, M. J. and Livny, M. (1987), "Concurrency Control Performance Modeling: Alternatives and Implications", ACM TODS, Vol. 12, pp. 609-654.

4. Bernstein, P. A. and Goodman, N. (1980), "Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems", Proc VLDB, Oct. 1980, pp. 285-300.
5. Bernstein, P. A. and Goodman N. (1981), "Concurrency Control in Distributed Database Systems", ACM Computing Surveys, June 1981, pp. 185-222.
6. Carey, M. and Muhanna, W. (1986), "The Performance of Multiversion Concurrency Control Algorithms", In ACM Trans. Comp. Syst. Vol 4, pp. 338-378.
7. Carey, M. and Stonebraker, M. (1984), "The Performance of Concurrency Control Algorithms for Database Management Systems", In VLDB 1984, pp. 107-118.
8. Coffman, E., Elphich, M., Shoshani, A. (1971), "System Deadlocks", ACM Computing Surveys, 2, 3, pp. 67-78.
9. Eswaran, K. P., Gray, J. N., Lorie, R. A. and Traiger, I. L. (1976), "The Notions of Consistency and Predicate Locks in a Database System", Comm. ACM, 10, 11, pp. 624-633.
10. Gray, J. (1978), "Notes on Data Base Operating Systems", in *Operating Systems: An Advanced Course*, Eds. R. Bayer, R. M. Graham and G. Seegmuller, Springer-Verlag.
11. Gray, J. (1981), "The Transaction Concept: Virtues and Limitations", Proc VLDB, Sept 1981.
12. Gray, J. N., Lorie, R. A. and Putzolu, G. R.(1975), "Granularity of Locks in a Large Shared Data Base", Proc VLDB, Sept 1975.
13. Gray, J. N., Lorie, R. A., Putzolu, G. R. and Traiger, I. L. (1976), "Granularity of Locks and Degrees of Consistency in a Shared Data Base", in Proc. IFIP TC-2 Working Conference on Modelling in Data Base Management Systems, Ed. G. M. Nijssen, North-Holland.
14. Kung, H. T. and Papadimitriou, C. H. (1979), "An Optimality Theory of Concurrency Control in Databases", Proc ACM SIGMOD, pp. ???
15. Kung, H. T. and Robinson, J. T. (1981), "On Optimistic Methods for Concurrency Control", ACM TODS, Vol. 6, pp.??

16. Ries, D. R. and Stonebraker, M. R.(1977), "Effects of Locking Granularity in a Database Management System", ACM TODS, Vol 2, pp.??
17. Papadimitriou, C. H. (1986), "The Theory of Database Concurrency Control", Computer Science Press.