**ST. XAVIER’S COLLEGE**

**(Affiliated to Tribhuvan University)**

**Maitighar, Kathmandu**



**Lab Assignment on Database Management System**

**Submitted by**

Divya Jyoti Pokharel

013BSCCSIT019

**Submitted to**

Er. Sanjay Kumar Yadav

Lecturer, Department of Computer Science

October 30, 2015

**Database Concurrency Control**

**Purpose of Concurrency Control**

Concurrency control is a database management systems (DBMS) concept that is used to address conflicts with the simultaneous accessing or altering of data that can occur with a multi-user system. Concurrency control, when applied to a DBMS, is meant to coordinate simultaneous transactions while preserving data integrity [1]. The Concurrency is about to control the multi-user access of Database.

To illustrate the concept of concurrency control, consider two travelers who go to electronic kiosks at the same time to purchase a train ticket to the same destination on the same train. There's only one seat left in the coach, but without concurrency control, it's possible that both travelers will end up purchasing a ticket for that one seat. However, with concurrency control, the database wouldn't allow this to happen. Both travelers would still be able to access the train seating database, but concurrency control would preserve data accuracy and allow only one traveler to purchase the seat.

This example also illustrates the importance of addressing this issue in a multi-user database. Obviously, one could quickly run into problems with the inaccurate data that can result from several transactions occurring simultaneously and writing over each other. The following section provides strategies for implementing concurrency control.

Simultaneous execution of transactions over a shared database can create several data integrity and consistency problems:

* Lost Updates.
* Uncommitted Data.
* Inconsistent retrievals.

Concurrent access to data is desirable when: The amount of data is sufficiently great that at any given time only fraction of the data can be in primary memory & rest should be swapped from secondary memory as needed.

Even if the entire database can be present in primary memory, there may be multiple processes.

**Two Phase Locking**

The most widely used locking protocol, called Strict Two-Phase Locking, or Strict 2PL, has two rules. The first rule is

If a transaction T wants to read (respectively, modify) an object, it first requests a shared (respectively, exclusive) lock on the object.

Of course, a transaction that has an exclusive lock can also read the object; an additional shared lock is not required. A transaction that requests a lock is suspended until the DBMS is able to grant it the requested lock. The DBMS keeps track of the locks it has granted and ensures that if a transaction holds an exclusive lock on an object, no other transaction holds a shared or exclusive lock on the same object. The second rule in Strict 2PL is

All locks held by a transaction are released when the transaction is completed.

Requests to acquire and release locks can be automatically inserted into transactions by the DBMS; users need not worry about these details.

In effect, the locking protocol allows only 'safe' interleaving of transactions. If two transactions access completely independent parts of the database, they concurrently obtain the locks they need and proceed merrily on their ways. On the other band, if two transactions access the same object, and one wants to modify it, their actions are effectively ordered serially, all actions of one of these transactions (the one that gets the lock on the common object first) are completed before (this lock is released and) the other transaction can proceed.

We denote the action of a transaction T requesting a shared (respectively, exclusive) lock on object 0 as 5T(0) (respectively, XT(O)) and omit the subscript denoting the tn1l1saction when it is clear from the context. As an example, consider the schedule shown in Figure 16.4. This interleaving could result in a state that cannot result from any serial execution of the three transactions. For instance, T1 could change A from 10 to 20, then T2 (which reads the value 20 for A) could change B from 100 to 200, and then T1 would read the value 200 for B. If run serially, either Tl or T2 would execute first, and read the values 10 for A and 100 for B: Clearly, the interleaved execution is not equivalent to either serial execution.

If the Strict 2PL protocol is used, such interleaving is disallowed. Let us see why. Assuming that the transactions proceed as before, T1 would obtain an exclusive lock on A first and then read and write A (Figure I). Then, 1'2 would request a lock on A.

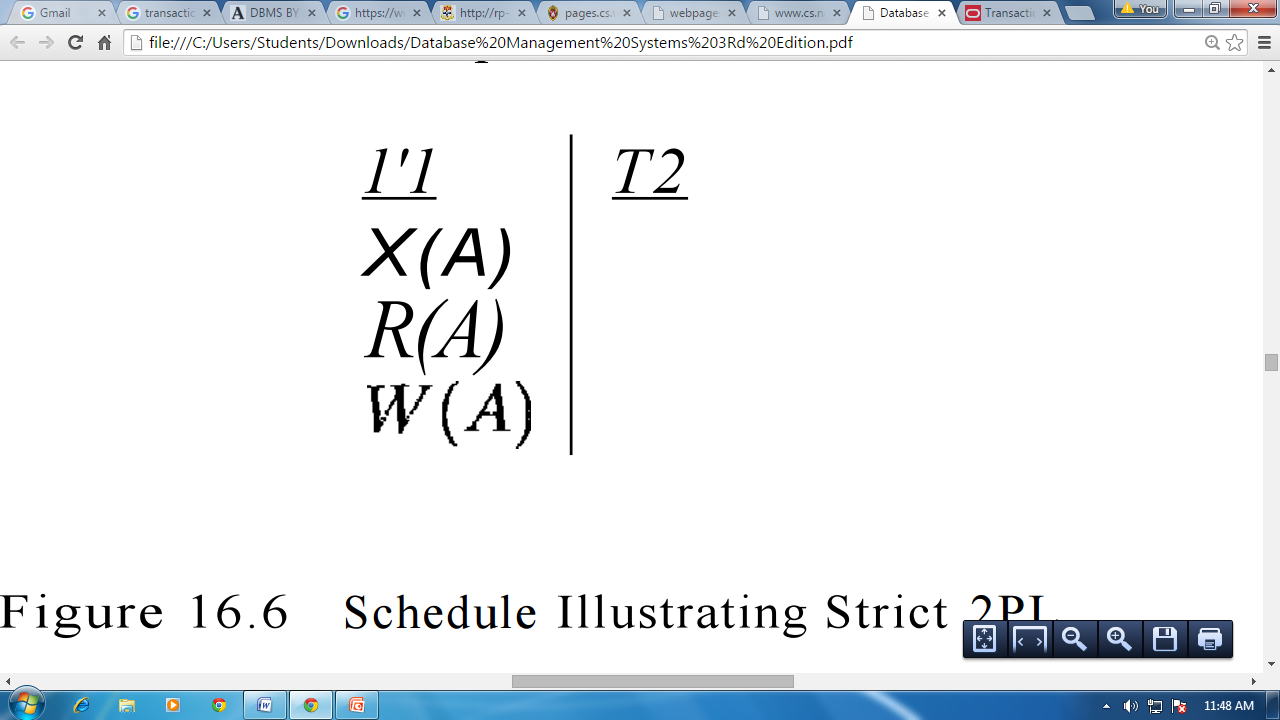


Figure I: Schedule Illustrating Strict 2PL

However, this request cannot be granted until 1'1 releases its exclusive lock on A, and the DBMS therefore suspends 1'2. 1'1 now proceeds to obtain an exclusive lock on B, reads and writes B, then finally commits, at which time its locks are released. T2's lock request is now granted, and it proceeds. In this example the locking protocol results in a serial execution of the two transactions, shown in Figure II.

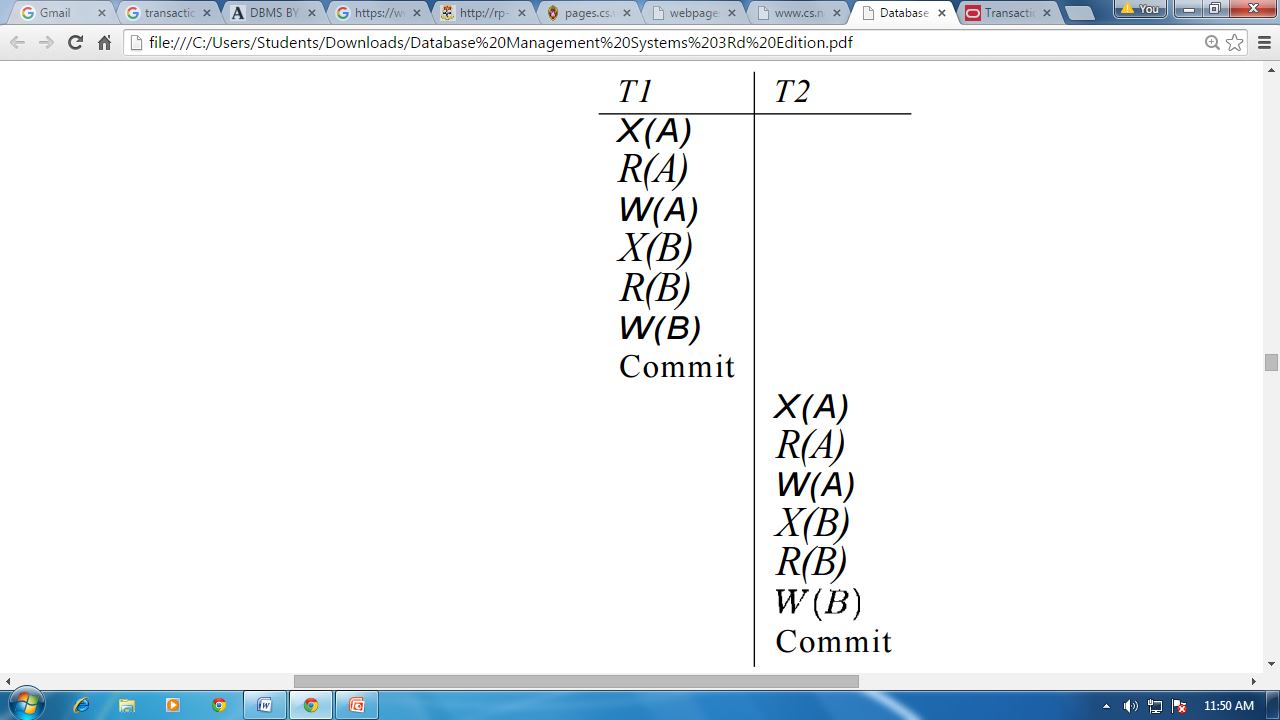


Figure II: Schedule Illustrating Strict 2PL with Serial Execution

In general, however, the actions of different transactions could be interleaved. As an example, consider the interleaving of two transactions shown in Figure III, which is permitted by the Strict 2PL protocol. It can be shown that the Strict 2PL algorithm allows only serializable schedules. None of the anomalies discussed in Section 16.3.:3 can arise if the DBMS implements strict 2PL.

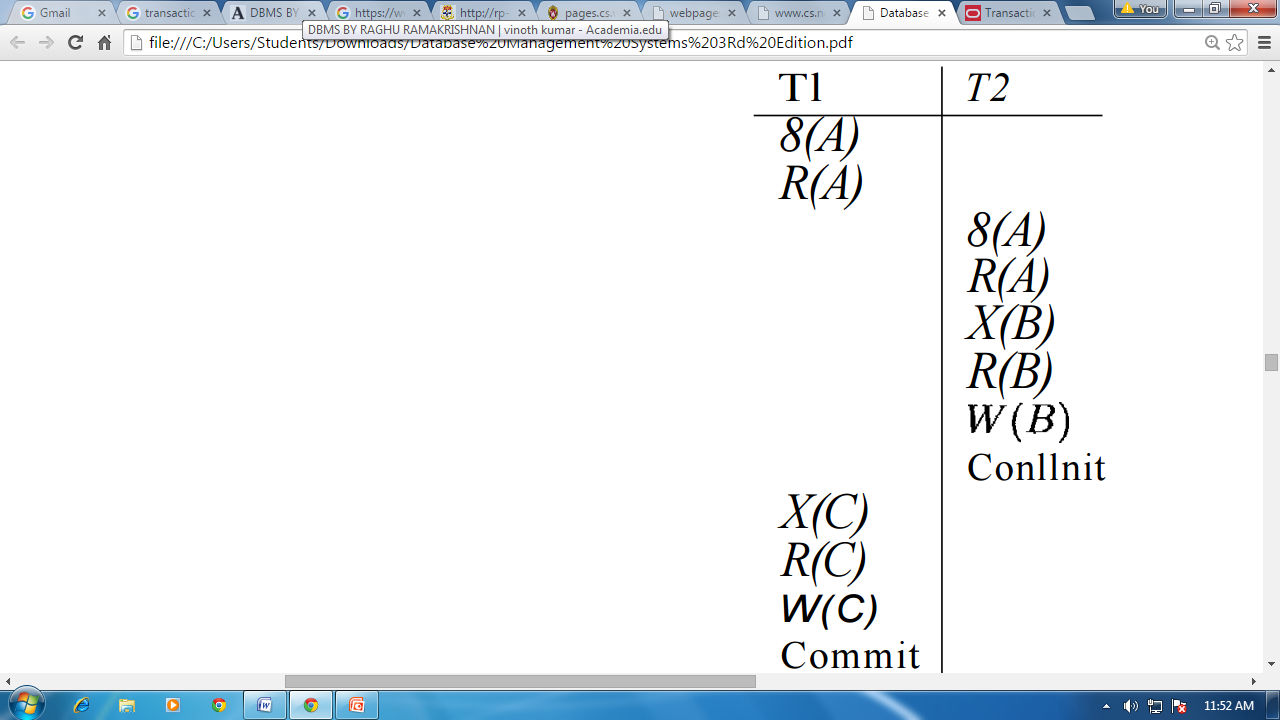


Figure III: Schedule Following Strict 2PL with Interleaved Actions

Limitation of CCMs

**Time stamp based protocols**

The most commonly used concurrency protocol is the timestamp based protocol. This protocol uses either system time or logical counter as a timestamp.

Lock-based protocols manage the order between the conflicting pairs among transactions at the time of execution, whereas timestamp-based protocols start working as soon as a transaction is created.

Every transaction has a timestamp associated with it, and the ordering is determined by the age of the transaction. A transaction created at 0002 clock time would be older than all other transactions that come after it. For example, any transaction 'y' entering the system at 0004 is two seconds younger and the priority would be given to the older one.

In addition, every data item is given the latest read and write-timestamp. This lets the system know when the last ‘read and write’ operation was performed on the data item.

**Commit Protocols**

* Commit protocols are used to ensure atomicity across sites
  + a transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
  + not acceptable to have a transaction committed at one site and aborted at another
* The two-phase commit (2 PC) protocol is widely used
* The three-phase commit (3 PC) protocol is more complicated and more expensive, but avoids some drawbacks of two-phase commit protocol.

**Index Locking**

* Index locking protocol states:
* Every relation must have at least one index. Access to a relation must be made only through one of the indices on the relation.
* A transaction Ti that performs a lookup must lock all the index buckets that it accesses, in S-mode.
* A transaction Ti may not insert a tuple ti into a relation r without updating all indices to r.
* It must perform a lookup on every index to find all index buckets that could have possibly contained a pointer to tuple ti, had it existed already, and obtain locks in X-mode on all these index buckets. Ti must also obtain locks in X-mode on all index buckets that it modifies.
* The rules of the two-phase locking protocol must be observed.

Lock Granularity

**Time Stamp Ordering Multi-version Concurrency Control**

Basic time stamping is a concurrency control mechanism that eliminates deadlock. This method doesn’t use locks to control concurrency, so it is impossible for deadlock to occur. According to this method a unique timestamp is assigned to each transaction, usually showing when it was started. This effectively allows an age to be assigned to transactions and an order to be assigned. Data items have both a read-timestamp and a write-timestamp. These timestamps are updated each time the data item is read or updated respectively.

Problems arise in this system when a transaction tries to read a data item which has been written by a younger transaction. This is called a late read. This means that the data item has changed since the initial transaction start time and the solution is to roll back the timestamp and acquire a new one. Another problem occurs when a transaction tries to write a data item which has been read by a younger transaction. This is called a late write. This means that the data item has been read by another transaction since the start time of the transaction that is altering it. The solution for this problem is the same as for the late read problem. The timestamp must be rolled back and a new one acquired [2].

Adhering to the rules of the basic time stamping process allows the transactions to be serialized and a chronological schedule of transactions can then be created. Time stamping may not be practical in the case of larger databases with high levels of transactions. A large amount of storage space would have to be dedicated to storing the timestamps in these cases [3].

Basic Timestamp Ordering

1. Transaction T issues a write\_item(X) operation:

If read\_TS(X) > TS(T) or if write\_TS(X) > TS(T), then an younger transaction has already read the data item so abort and roll-back T and reject the operation.

If the condition in part (a) does not exist, then execute write\_item(X) of T and set write\_TS(X) to TS(T).

2. Transaction T issues a read\_item(X) operation:

If write\_TS(X) > TS(T), then an younger transaction has already written to the data item so abort and roll-back T and reject the operation.

If write\_TS(X) ≤ TS(T), then execute read\_item(X) of T and set read\_TS(X) to the larger of TS(T) and the current read\_TS(X).

Strict Timestamp Ordering

1. Transaction T issues a write\_item(X) operation:

If TS(T) > read\_TS(X), then delay T until the transaction T’ that wrote or read X has terminated (committed or aborted).

2. Transaction T issues a read\_item(X) operation:

If TS(T) > write\_TS(X), then delay T until the transaction T’ that wrote or read X has terminated (committed or aborted).

Thomas’s Write Rule

If read\_TS(X) > TS(T) then abort and roll-back T and reject the operation.

If write\_TS(X) > TS(T), then just ignore the write operation and continue execution. This is because the most recent writes counts in case of two consecutive writes.

If the conditions given in 1 and 2 above do not occur, then execute write\_item(X) of T and set write\_TS(X) to TS(T).

**Deadlock Handling Detection and Resolution**

When dealing with locks two problems can arise, the first of which being deadlock. Deadlock refers to a particular situation where two or more processes are each waiting for another to release a resource, or more than two processes are waiting for resources in a circular chain. Deadlock is a common problem in multiprocessing where many processes share a specific type of mutually exclusive resource. Some computers, usually those intended for the time-sharing and/or real-time markets, are often equipped with a hardware lock, or hard lock, which guarantees exclusive access to processes, forcing serialization. Deadlocks are particularly disconcerting because there is no general solution to avoid them.

A fitting analogy of the deadlock problem could be a situation like when you go to unlock your car door and your passenger pulls the handle at the exact same time, leaving the door still locked. If you have ever been in a situation where the passenger is impatient and keeps trying to open the door, it can be very frustrating. Basically you can get stuck in an endless cycle, and since both actions cannot be satisfied, deadlock occurs.