ST. XAVIER’S COLLEGE

**Maitighar,Kathmandu**

**(Affiliated to Tribhuvan University)**



**Database Management System**

**Lab Assignment #12**

**Submitted By**

Aashish Raj Shrestha

B.Sc. CSIT

Year II/IV Semester

013BSCIT002

**Submitted To**

Er. Sanjay Kumar Yadav

Lecturer,

Department of Computer Science

St. Xavier’s College

Maitighar, Kathmandu

**Submitted On**

October 30, 2015

**Database Concurrency Control**

1. **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:

1. 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.
2. Even if the entire database can be present in primary memory, there may be multiple processes.
3. **Two Phase Locking**

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

1. 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

1. 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.

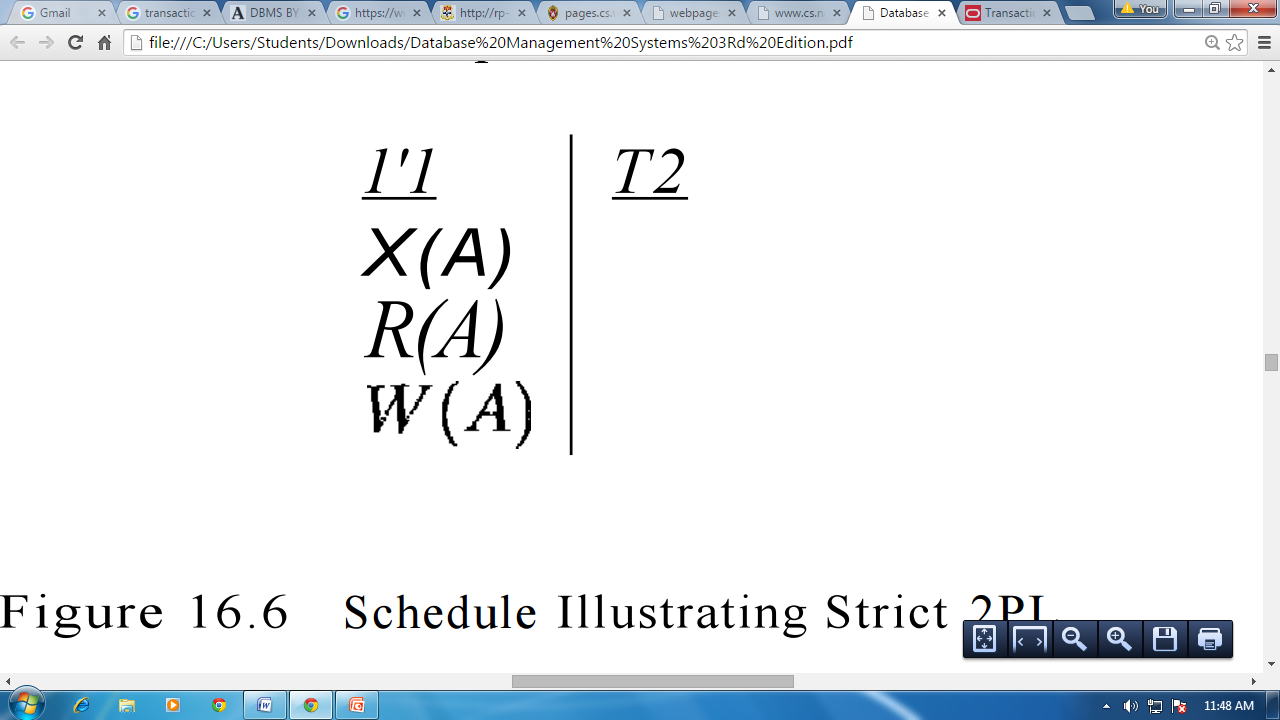


Figure I: Schedule Illustrating Strict 2PL

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.

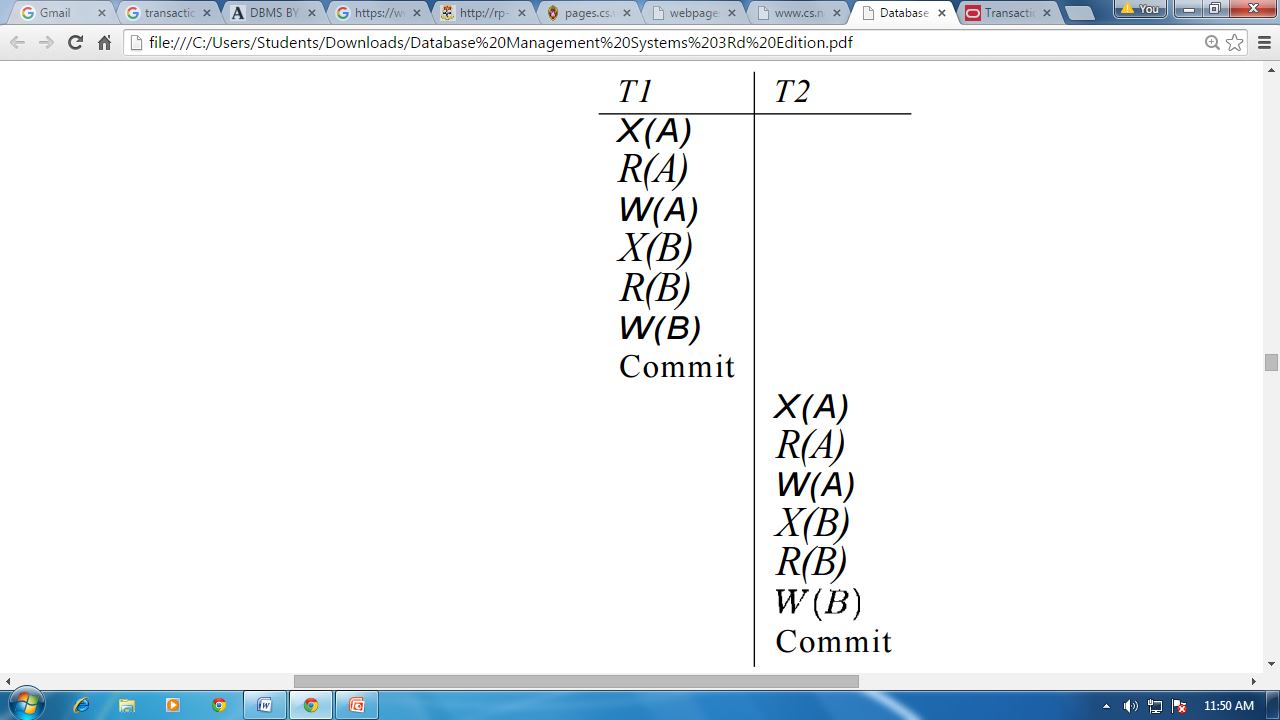


Figure II: Schedule Illustrating Strict 2PL with Serial Execution

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.

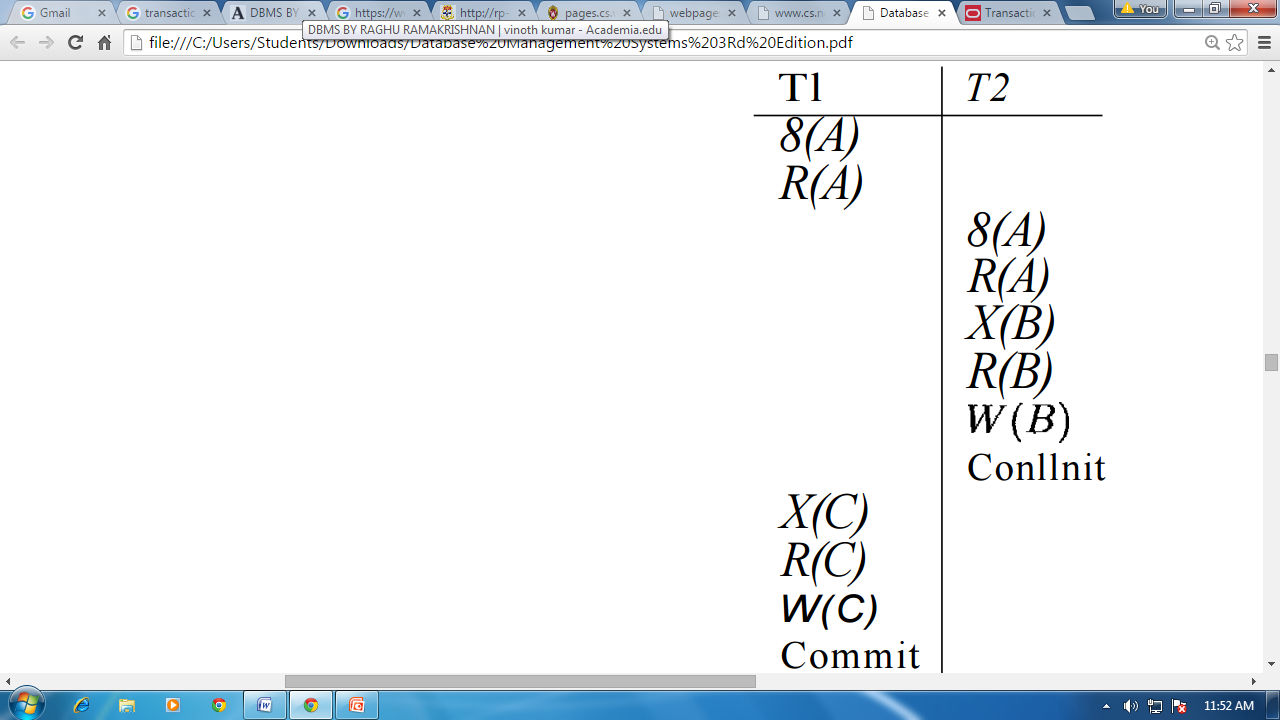


Figure III: Schedule Following Strict 2PL with Interleaved Actions

1. **Limitation of CCMs**

By implementing a few measures such as data sanitization and the limitation of executable queries and analyses. Data sanitization can be defined as the process.

Today’s level of ever evolving computer, database, and internet technology has enabled the collection and mining of data, as well as the utilization of that data on a level that was previously unimaginable. Each individual exists in a numerous databases around the world, from purchases made at the local supermarket to purchases made on-line to confidential medical records to credit information, etc.

While individually the information that resides within the various databases listed above may not reveal much about a person, access to several databases may provide a detailed and possibly invasive amount of personal information. Inherently, there is a fundamental tradeoff between the functionality of a database or a database management system (DBMS) and the level of privacy given to the subjects of the database.

While there are many benefits to advancements in database and DBMS technology, its advent has also created the possibility for significant abuses. Ideally, the design and implementation of a database would be constructed in a manner that will allow users to obtain and analyze information from a database(s) without allowing its users to access subjects’ private information. This problem can be dramatically reduced by implementing a few measures such as data sanitization and the limitation of executable queries and analyses. Data sanitization can be defined as the process of removing sensitive information from a document or other medium, so that it may be distributed to a broader audience.

Sanitization attempts to reduce the personal content present in a database, while at the same time retaining enough functionality to supply the reader/user with the necessary information. The concept of the limitation of analyses is based on providing the user with either a limited amount of preset queries or query variables by which they must operate and/or limiting the number of queries and individual user may execute.

1. **Time stamp based protocols**

The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.

Suppose a transaction Ti issues a **read** (*Q*)

* 1. If TS (*Ti*) ≤ **W**-timestamp (*Q*), then *Ti* needs to read a value of *Q* that was already overwritten.
     + Hence, the **read** operation is rejected, and *Ti* is rolled back.
  2. If TS (*Ti*)≥ **W**-timestamp (*Q*), then the **read** operation is executed, and R-timestamp (*Q*) is set to **max**(R-timestamp (*Q*), TS (*Ti*)).

A timestamp is a unique identifier created by the DBMS to identify a transaction. Typically, timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the *transaction start time.*We will refer to the timestamp of transaction T as TS (T). Concurrency control techniques based on timestamp ordering do not use locks; hence, *deadlocks cannot occur.*

**Generation of Timestamp:**

Timestamps can be generated in several ways. One possibility is to use a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps are numbered 1, 2, 3, . . . in this scheme. A computer counter has a finite maximum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time. Another way to implement timestamps is to use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

**Timestamps**

With each transaction Ti in the system, we associate a unique fixed timestamp, denoted by TS (Ti). This timestamp is assigned by the database system before the Transaction Ti starts execution.

                If a transaction Ti has been assigned timestamp TS (Ti) and a new transaction Tj enters the system, then TS (Ti) <TS (Tj). Thus if TS (Ti) <TS (Tj), then they must ensure that the produced schedule is equivalent to a serial schedule in which transaction Ti appears before Transaction Tj.

To implement this scheme we associate with each data item Q, two timestamp values:

1. **W-timestamp (Q)**: denotes the largest timestamp of any transaction that executed write (Q) successfully.

2. **R-timestamp (Q)**: denotes the largest timestamp of any transaction that executed read (Q) successfully.

**The timestamp ordering protocol**

1. Suppose that transaction Tiissues read (Q)

* If TS (Ti) < W-timestamp (Q), then Ti needs to read value of Q that was already overwritten. Hence read operation is rejected and Ti is rolled back.
* if TS(Ti) < W –timestamp(Q), the read operation is executed and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(Ti)

 2. Suppose that transaction Ti issues write (Q)

* If TS(Ti) < R- timestamp(Q), then the value of Q that Ti­ is producing was needed previously and the system assumed that, that value would never be produced, Hence  the system rejects the write operation and rolls Ti ­back.
* If TS(Ti)<W-timestamp(Q), then Ti is attempting to write an obsolete value of Q, Hence the system rejects this write operation and rolls Ti back.
* Otherwise, the system executes the write operation and sets W-timestamp to TS(Ti)

**Few Points**

1. Timestamp Ordering protocol ensures conflict serializability.
2. The protocol ensures freedom from the deadlock, since no transaction ever waits. However, there is a possibility of Starvation of long transaction if a sequence of conflicting start transactions caused repeated restarting of the long transaction.
3. Protocol generates the schedules that are not recoverable.

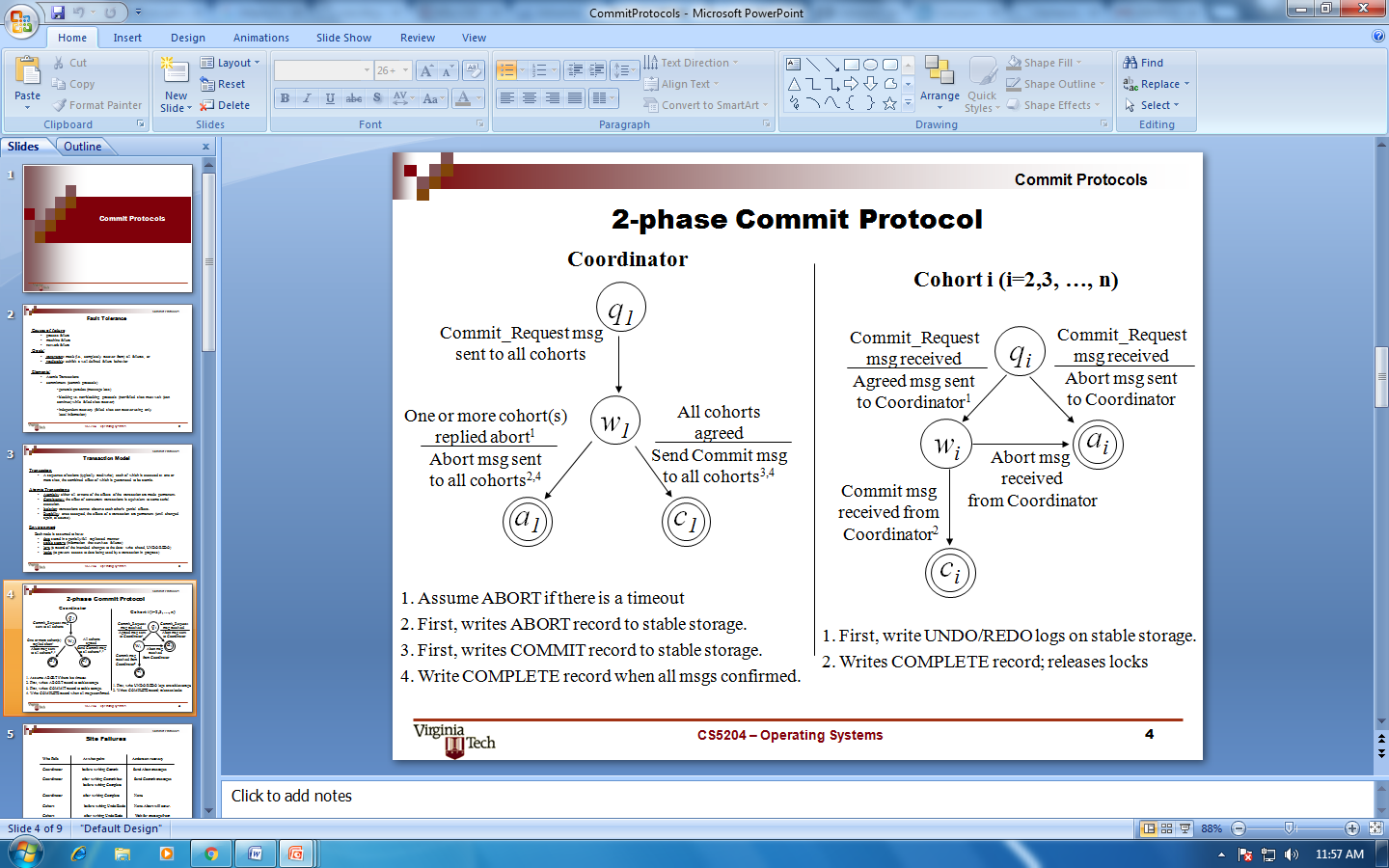
**Thomas Write Rule (modification to Timestamp ordering protocol)**

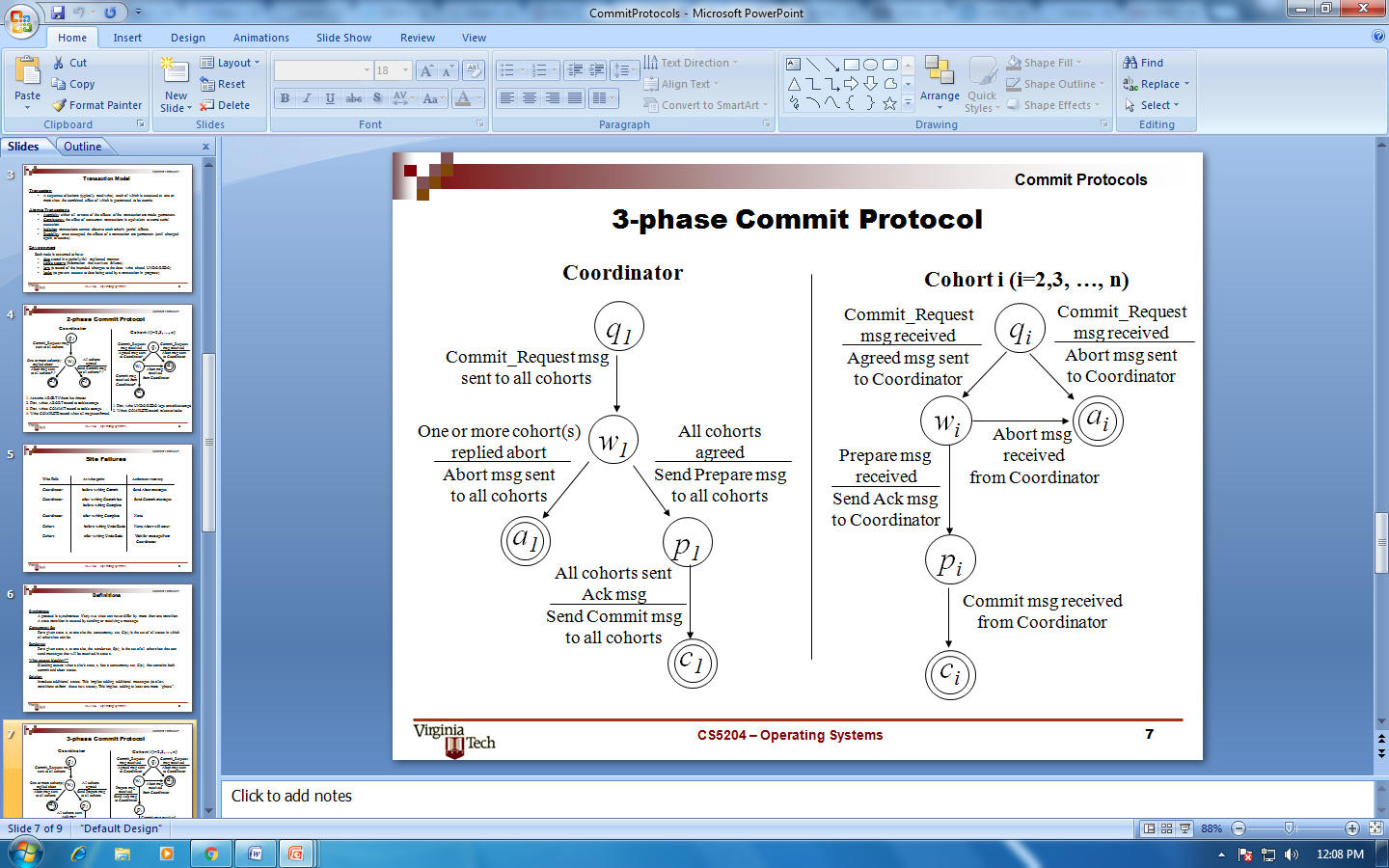
Ensures greater potential concurrency

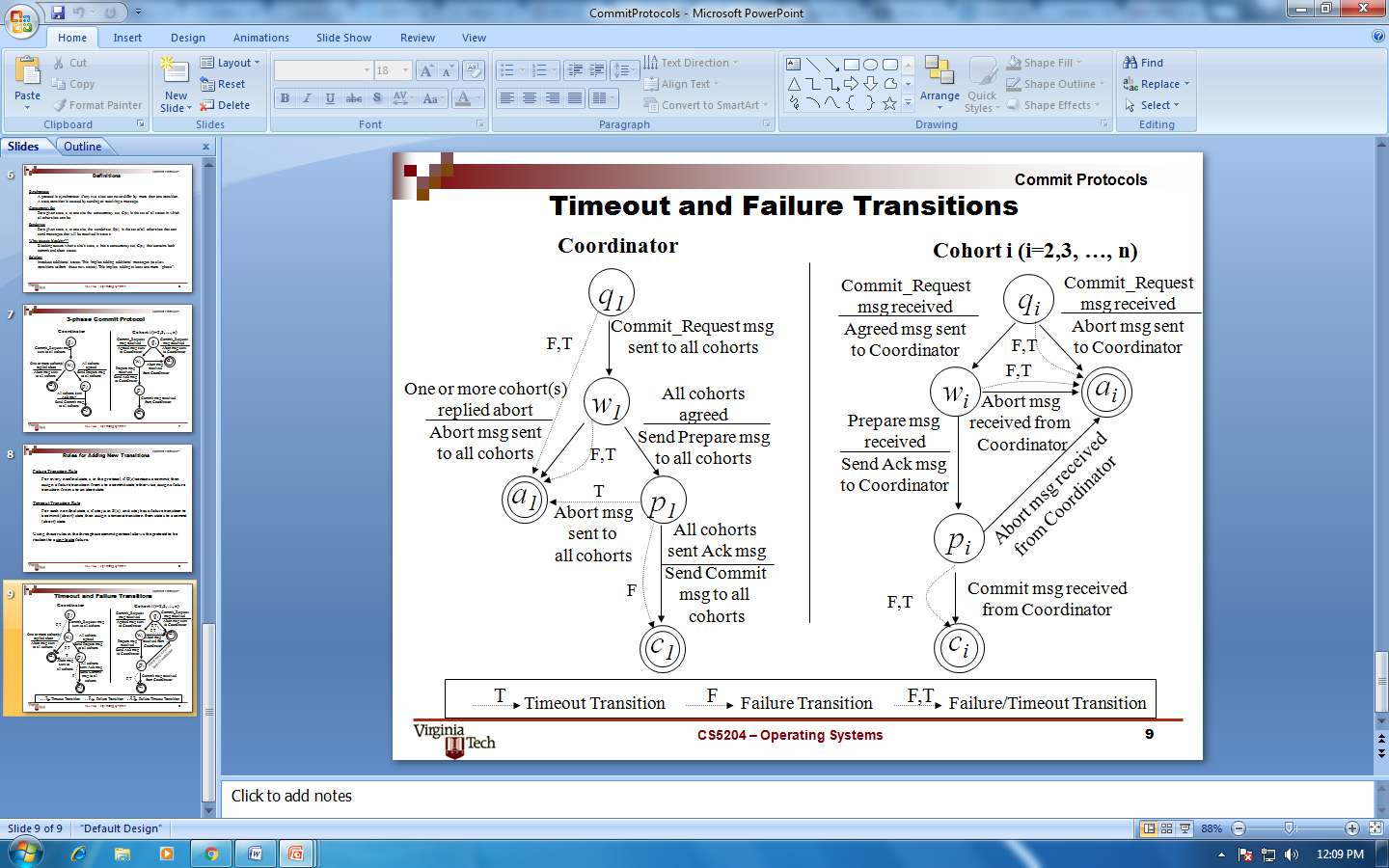
Suppose that Transaction Ti issues write (Q)

* If TS(Ti) < R- timestamp(Q), then the value of Q that Ti­ is producing was needed previously and the system assumed that, that value would never be produced, Hence  the system rejects the write operation and rolls Ti ­back.
* If TS(Ti)<W-timestamp(Q), then Ti is attempting to write an obsolete value of Q, Hence this write operation is ignored.(different from Timestamp based protocol where it is rolled back)
* Otherwise, the system executes the write operation and sets W-timestamp to TS (Ti).

**Commit Protocols**







1. **Index Locking**

Index locking:

* + Every relation must have at least one index.
  + A transaction can access tuples only after finding them through one or more indices on the relation
  + A transaction *Ti* that performs a lookup must lock all the index leaf nodes that it accesses, in S-mode
    - Even if the leaf node does not contain any tuple satisfying the index lookup (e.g. for a range query, no tuple in a leaf is in the range)
  + A transaction *Ti* that inserts, updates or deletes a tuple *ti* in a relation *r*
    - must update all indices to *r*
    - must obtain exclusive locks on all index leaf nodes affected by the insert/update/delete
  + The rules of the two-phase locking protocol must be observed

Guarantees that phantom phenomenon won’t occur

1. **Lock Granularity**

* Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
* Can be represented graphically as a tree (but don't confuse with tree-locking protocol)
* When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode.
* Granularity of locking (level in tree where locking is done):
  + **fine granularity** (lower in tree): high concurrency, high locking overhead
  + **coarse granularity** (higher in tree): low locking overhead, low concurrency

**Multiple Granularities Locking Scheme**

* Transaction *Ti* can lock a node *Q*, using the following rules:
  1. The lock compatibility matrix must be observed.
  2. The root of the tree must be locked first, and may be locked in any mode.
  3. A node *Q* can be locked by *Ti* in S or IS mode only if the parent of *Q* is currently locked by *Ti* in either IX or IS mode.
  4. A node *Q* can be locked by *Ti* in X, SIX, or IX mode only if the parent of *Q* is currently locked by *Ti* in either IX or SIX mode.
  5. *Ti* can lock a node only if it has not previously unlocked any node (that is, *Ti* is two-phase).
  6. *Ti* can unlock a node *Q* only if none of the children of *Q* are currently locked by *Ti.*
* Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.
* **Lock granularity escalation**: in case there are too many locks at a particular level, switch to higher granularity S or X lock

1. **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).

1. **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.

System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies:

* + Require that each transaction locks all its data items before it begins execution (pre-declaration).
  + Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

**References:**

[1] Coronel, Carlos, Peter Rob. *Database Systems*, sixth ed. Thomson Course Technology, 2004.

[2] Ambler, Scott. *Introduction to Concurrency Control*, 2006 <http://www.agiledata.org/essays/concurrencyControl.html>

[2] Ambler, Scott. *Introduction to Concurrency Control*, 2006 <http://www.alkissdesigners.kbo.co.ke>

[3] Ricardo, Catherine. *Databases Illuminated*, second ed. p386-387 Jones & Bartlett Learning, 2012.

[4] Kumar, V. *Transaction Management Concurrency Control Mechanisms*, 2012 <http://sce.umkc.edu/~kumarv/cs470/transaction/T-management.pdf>