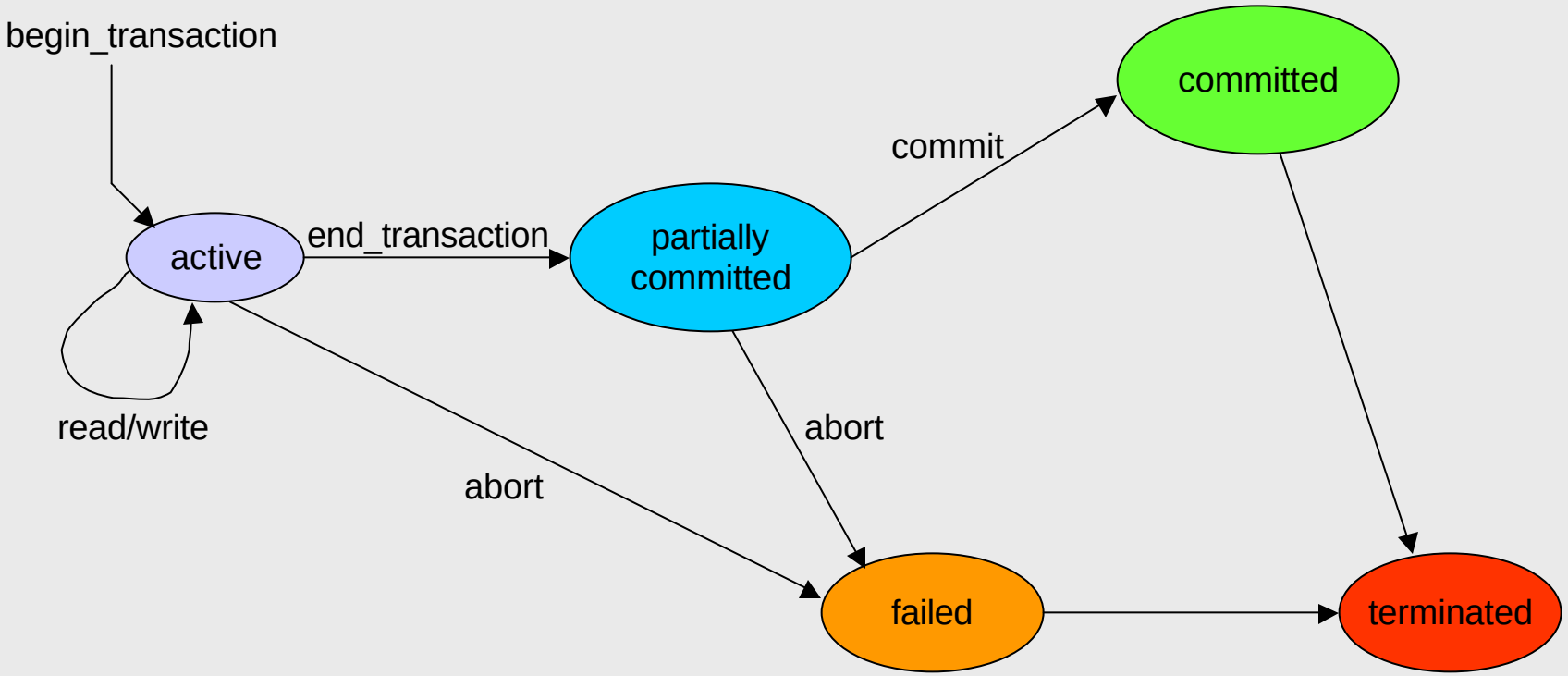
COP 4710: Database Systems
Spring 2004

-Day 23 – March 31, 2004 –
Transaction Processing

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CC1 211, 823-2790
http://www.cs.ucf.edu/courses/cop4710/spr2004

School of Electrical Engineering and Computer Science
University of Central Florida

The States of a Transaction (cont.)



System Log

- The system log keeps track of all transaction operations that affect values of database items.
- The information in the log is used to perform recovery operations from transaction failures.
- Most logs consist of several levels ranging from the log maintained in main memory to archival versions on backup storage devices.
- Upon entering the system, each transaction is given a unique transaction identifier (timestamps are common).

System Log (cont.)

- In the system log, several different types of entries occur depending on the action of the transaction:
  - [start, T]: begin transaction T.
  - [write, T, X, old, new]: transaction T performs a write on object X, both old and new values of X are recorded in the log entry.
  - [read, T, X]: transaction T performs a read on object X.
  - [commit, T]: transaction T has successfully completed and indicates that its changes can be made permanent.
  - [abort, T]: transaction T has aborted.
- Some types of recovery protocols do not require read operations be logged.

Commit Point

- A transaction T reaches its commit point when all of its operations that access the database have successfully completed and the effect of all of these operations have been recorded in the log.

- Beyond the commit point, a transaction is said to be committed and its effect on the database is assumed to be permanent. It is at this point that [commit, T] is entered into the system log.

- If a failure occurs, a search backward through the log (in terms of time) is made for all transactions that have written a [start, T] into the log but have not yet written [commit, T] into the log. This set of transactions must be rolled back.

ACID Properties of Transactions

- Atomcity – a transaction is an atomic unit of processing; it is either performed in its entirety or not at all.
- Consistency – a correct execution of the transaction must take the database from one consistent state to another.
- Isolation – a transaction should not make its updates visible to other transactions until it is committed. Strict enforcement of this property solves the dirty read problem and prevents cascading rollbacks from occurring.
- Durability – once a transaction changes the database and those changes are committed, the changes must never be lost because of a failure.

Schedules and Recoverability

- When transactions are executing concurrently in an interleaved fashion, the order of execution of the operations from the various transactions forms what is known as a transaction schedule (sometimes called a history).

A schedule S of *n* transactions T_1, T_2, T_3, ..., T_n is an ordering of the operations of the transactions where for each transaction $T_i \in S$, each operation in T_i occurs in the same order in both T_i and S.

Schedules and Recoverability (cont.)

- The notation used for depicting schedules is:
  - $r_i(x)$ means that transaction *i* performs a read of object x.
  - $w_i(x)$ means that transaction *i* performs a write of object x.
  - c_i means that transaction *i* commits.
  - a_i means that transaction *i* aborts.
- An example schedule: $S_A = (r_1(x), r_2(x), w_1(x), w_2(x), c_1, c_2)$
- This example schedule represents the lost update problem.
- Another example:

  $S_B = (r_1(x), r_1(y), w_1(y), r_2(x), w_1(x), w_2(y), c_2, c_1)$

Conflict in a Schedule

- Two operations in a schedule are said to conflict if they belong to different transactions, access the same item, and one of the operations is a write operation.
- Consider the following schedule:

$S_A = (r_1(x), r_2(x), w_1(x), c_1, c_2)$

$r_2(x)$ and $w_1(x)$ conflict

$r_1(x)$ and $r_2(x)$ do not conflict.

Recoverability

- For some schedules it is easy to recover from transaction failures, while for others it can be quite difficult and involved.

- Recoverability from failures depends in large part on the scheduling protocols used. A protocol which never rolls back a transaction once it is committed is said to be a recoverable schedule.

- Within a schedule a transaction T is said to have read from a transaction T* if in the schedule some item X is first written by T* and subsequently read by T.

Recoverability (cont.)

- A schedule S is a recoverable schedule if no transaction T in S commits until all transactions T* that have written an item which T reads have committed.
  - For each pair of transactions T_x and T_y, if T_y reads an item previously written by T_x, then T_x must commit before T_y.

Example: $S_A = (r_1(x), r_2(x), w_1(x), r_1(y), w_2(x), c_2, w_1(y), c_1)$

This is a recoverable schedule since, T_2 does not read any item written by T_1 and T_1 does not read any item written by T_2.

Example: $S_B = (r_1(x), w_1(x), r_2(x), r_1(y), w_2(x), c_2, a_1)$

This is not a recoverable schedule since T_2 reads value of x written by T_1 and T_2 commits before T_1 aborts. Since T_1 aborts, the value of x written by T_2 must be invalid so T_2 which has committed must be rolled back rendering schedule S_B not recoverable.

Cascading Rollback

- Cascading rollback occurs when an uncommitted transaction must be rolled back due to its read of an item written by a transaction that has failed.

Example: $S_A = (r_1(x), w_1(x), r_2(x), r_1(y), r_3(x), w_2(x), w_1(y), a_1)$

In S_A, T_3 must be rolled back since T_3 read value of x produced by T_1 and T_1 subsequently failed. T_2 must also be rolled back since T_2 read value of x produced by T_1 and T_1 subsequently failed.

Example: $S_B = (r_1(x), w_1(x), r_2(x), w_2(x), r_3(x), w_1(y), a_1)$

In S_B, T_2 must be rolled back since T_2 read value of x produced by T_1 and T_1 subsequently failed. T_3 must also be rolled back since T_3 read value of x produced by T_2 and T_2 subsequently failed. T_3 is rolled back, not because of the failure of T_1 but because of the failure of T_2.

Cascading Rollback (cont.)

- Cascading rollback can be avoided in a schedule if every transaction in the schedule only reads items that were written by committed transactions.
- A strict schedule is a schedule in which not transaction can read or write an item *x* until the last transaction that wrote *x* has committed (or aborted).
  - Example: $S_A = (r_1(x), w_1(x), c_1, r_2(x), c_2)$

Serializability

- Given two transactions T_1 and T_2, if no interleaving of the transactions is allowed (they are executed in isolation), then there are only two ways of ordering the operations of the two transactions.

  Either: (1) T_1 executes followed by T_2

  or (2) T_2 executes followed by T_1

- Interleaving of the operations of the transactions allows for many possible orders in which the operations can be performed.

Serializability (cont.)

- Serializability theory determines which schedules are correct and which are not and develops techniques which allow for only correct schedules to be executed.
- Interleaved execution, regardless of what order is selected, must have the same effect of some serial ordering of the transactions in a schedule.
- A serial schedule is one in which every transaction T that participates in the schedule, all of the operations of T are executed consecutively in the schedule, otherwise the schedule is non-serial.

Serializability (cont.)

- A concurrent (or interleaved) schedule of *n* transactions is serializable if it is equivalent (produces the same result) to some serial schedule of the same *n* transactions.
- A schedule of *n* transactions will have *n*! serial schedules and many more non-serial schedules.
- Example: Transactions T1, T2, and T3 have the following serial schedules: (T1, T2, T3), (T1, T3, T2), (T2, T1, T3), (T2, T3, T1), (T3, T1, T2), and (T3, T2, T1).
- There are two disjoint sets of non-serializable schedules:
  - Serializable: those non-serial schedules which are equivalent to one or more of the serial schedules.
  - Non-serializable: those non-serial schedules which are not equivalent to any serial schedule.

Serializability (cont.)

- There are two main types of serializable schedules:
  - Conflict serializable: In general this is an $O(n^3)$ problem where *n* represents the number of vertices in a graph representing distinct transactions.
  - View serializable: This is an NP-C problem, meaning that the only known algorithms to solve it are exponential in the number of transactions in the schedule.
- We'll look only a conflict serializable schedules.
- Recall that two operations in a schedule conflict if (1) they belong to different transactions, (2) they access the same database item, and (3) one of the operations is a write.

Conflict Serializability

- If the two conflicting operations are applied in different orders in two different schedules, the effect of the schedules can be different on either the transaction or the database, and thus, the two schedules are not conflict equivalent.
  - Example: $S_A = (r_1(x), w_2(x))$

    $S_B = (w_2(x), r_1(x))$

    The value of x read in S_A may be different than in S_B.
  - Example: $S_A = (w_1(x), w_2(x), r_3(x))$

    $S_B = (w_2(x), w_1(x), r_3(x))$

    The value of x read by T_3 may be different in S_A than in S_B

Conflict Serializability (cont.)

- To generate a conflict serializable schedule equivalent to some serial schedule using the notion of conflict equivalence involves the reordering of non-conflicting operations of the schedule until an equivalent serial schedule is produced.

- The technique is this: build a precedence graph based upon the concurrent schedule. Use a cycle detection algorithm on the graph. If a cycle exists, S is not conflict serializable. If no cycle exists, a topological sort of the graph will yield an equivalent serial schedule.

Algorithm Conflict_Serializable

Algorithm Conflict_Serializable

//input: a concurrent schedule S

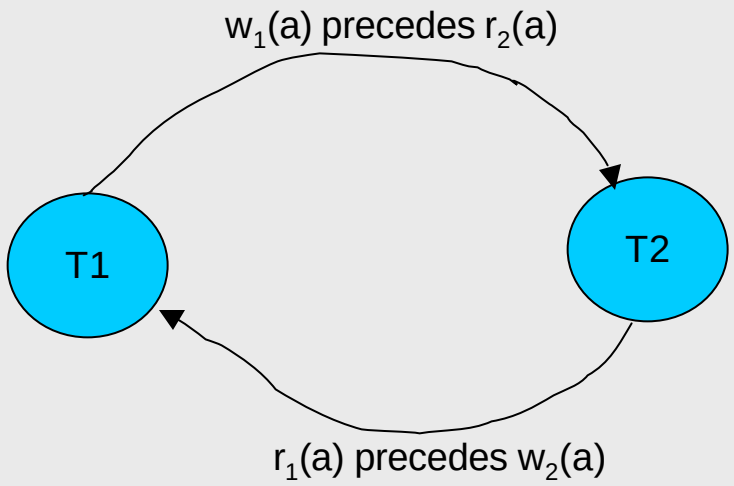
//output: no – if S is not conflict serializable, a serial schedule S* equivalent to S otherwise.

Conflict_Serializable(S)

1. for each transaction $T_X \in$ S, create a node (in the graph) labeled T_X.
2. for each case in S where T_Y executes read(a) after T_X executes write(a) create the edge $T_X \rightarrow T_Y$. The meaning of this edge is that T_X must precede T_Y in any serially equivalent schedule.
3. for each case in S where T_Y executes write(a) after T_X executes read(a) create the edge $T_X \rightarrow T_Y$. The meaning of this edge is that T_X must precede T_Y in any serially equivalent schedule.
4. for each case in S where T_Y executes write(a) after T_X executes write(a) create the edge $T_X \rightarrow T_Y$. The meaning of this edge is that T_X must precede T_Y in any serially equivalent schedule.
5. if the graph contains a cycle then return no, otherwise topologically sort the graph and return a serial schedule S* which is equivalent to the concurrent schedule S.

Conflict Serializability – Example #1

Let $S_C = (r_1(a), w_1(a), r_2(a), w_2(a), r_1(b), w_1(b), r_2(b), w_2(b))$

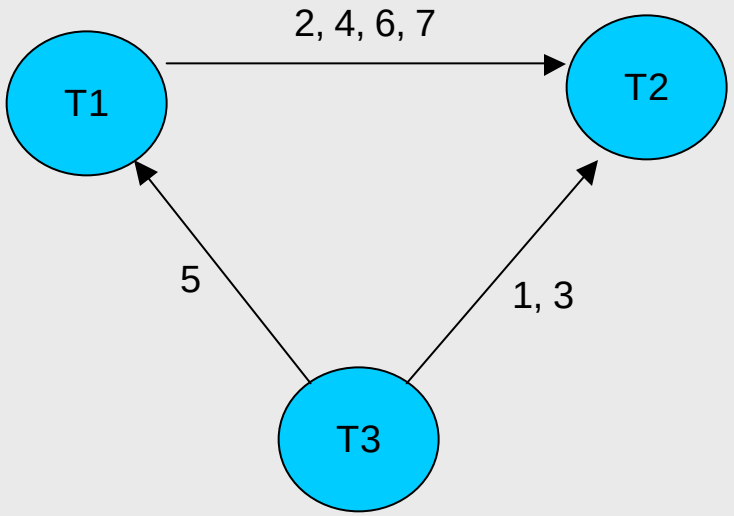


Graph contains a cycle, so S_C is not conflict serializable

Conflict Serializability – Example #2

Let $S_C = (r_3(y), r_3(z), r_1(x), w_1(x), w_3(y), w_3(z), r_2(z), r_1(y), w_1(y),$

$r_2(y), w_2(y), r_2(y), w_2(y))$



edge	reason
1	$w_3(y)$ precedes $r_2(y)$
2	$w_1(x)$ precedes $r_2(x)$
3	$w_3(z)$ precedes $r_2(z)$
4	$w_1(y)$ precedes $r_2(y)$
5	$r_3(y)$ precedes $w_1(y)$
6	$r_1(x)$ precedes $w_2(x)$
7	$r_1(y)$ precedes $w_2(y)$

Graph contains no cycles, so a serially equivalent schedule would be T3, T1, T2.

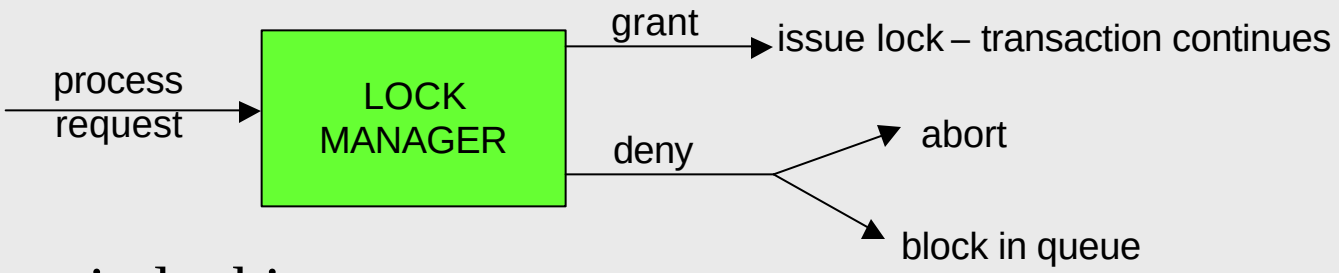
Concurrency Control Techniques

- There are several different techniques that can be employed to handle concurrent transactions.
- The basic techniques fall into one of four categories:

1. Locking protocols
2. Timestamping protocols
3. Multiversion protocols – deal with multiple versions of the same data
4. Optimistic protocols – validation and certification techniques

Locking Protocols

- Transactions "request" locks and "release" locks on database objects through a system component called a lock manager.



- Main issues in locking are:
  - What type of locks are to be maintained.
  - Lock granularity: runs from very coarse to very fine.
  - Locking protocol
  - Deadlock, livelock, starvation
  - Other issues such as serializability

Locking Protocols (cont.)

- Locking protocols are quite varied in their degree of complexity and sophistication, ranging from very simple yet highly restrictive protocols, to quite complex protocols which nearly rival time-stamping protocols in their flexibility for allowing concurrent execution.
- In order to give you a flavor of how locking protocols work, we'll focus on only the most simple locking protocols.
- While the basic techniques of all locking protocols are the same, in general, the more complex the locking protocol the higher the degree of concurrent execution that will be permitted under the protocol.

Locking Granularity

- When devising a locking protocol, one of the first things that must be considered is the level of locking that will be supported by the protocol.
- Simple protocols will support only a single level of locking while more sophisticated protocols can support several different levels of locking.
- The *locking level* (also called the *locking granularity*), defines the type of database object on which a lock can be obtained.
- The coarsest level of locking is at the database level, a transaction basically locks the entire database while it is executing. Serializability is ensured because with the entire database locked, only one transaction can be executing at a time, which ensures a serial schedule of the transactions.

Locking Granularity (cont.)

- Moving toward a finer locking level, typically the next level of locking that is available is at the relation (table) level. In this case, a lock is obtained on each relation that is required by a transaction to complete its task.
  - If we have two transactions which need different relations to accomplish their tasks, then they can execute concurrently by obtaining locks on their respective relations without interfering with one another. Thus, the finer grain lock has the potential to enhance the level of concurrency in the system.
- The next level of locking is usually at the tuple level. In this case several transactions can be executing on the same relation simultaneously, provided that they do not need the same tuples to perform their tasks.
- At the extreme fine end of the locking granularity would be locks at the attribute level. This would allow multiple transactions to be simultaneously executing in the same relation in the same tuple, as long as they didn't need the same attribute from the same tuple at the same time. At this level of locking the highest degree of concurrency will be achieved.

Locking Granularity (cont.)

- There is, unfortunately a trade-off between enhancing the level of concurrency in the system and the ability to manage the locks.
  - At the coarse end of the scale we need to manage only a single lock, which is easy to do, but this also gives us the least degree of concurrency.
  - At the extremely fine end of the scale we would need to manage an extremely large number of locks in order to achieve the highest degree of concurrency in the system.
- Unfortunately, with VLDB (Very Large Data Bases) the number of locks that would need to be managed at the attribute level poses too complex of a problem to handle efficiently and locking at this level almost never occurs.

Locking Granularity (cont.)

- For example, consider a fairly small database consisting of 10 relations each with 10 attributes and suppose that each relation has 1000 tuples. This database would require the management of $10 \times 10 \times 1000 = 100{,}000$ locks. A large database with 50 relations each having 25 attributes and assuming that each relation contained on the order of a 100,000 tuples; the number of locks that need to be managed grows to 1.25×10^8 (125 million locks).

- A VLDB with hundreds of relations and hundreds of attributes and potentially millions of tuples can easily require billions of locks to be maintained if the locking level is at the attribute level.

- Due to the potentially overwhelming number of locks that would need to be maintained at this level, a compromise to the tuple level of locking is often utilized.