COP 4710: Database Systems
Fall 2009

CHAPTERS 16 & 17 – Transaction Processing

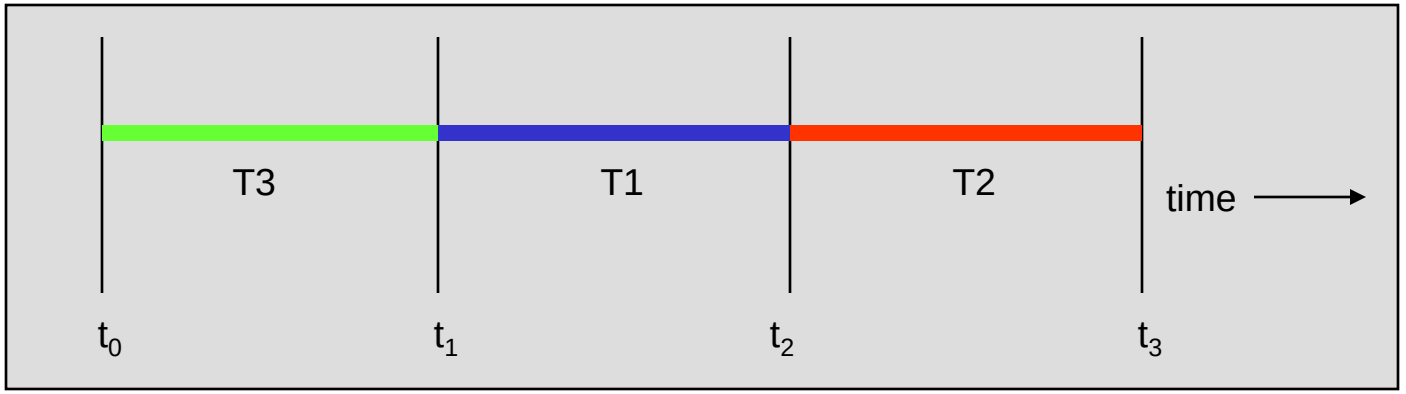
Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cop4710/fall2009

School of Electrical Engineering and Computer Science
University of Central Florida

Introduction to Transaction Processing

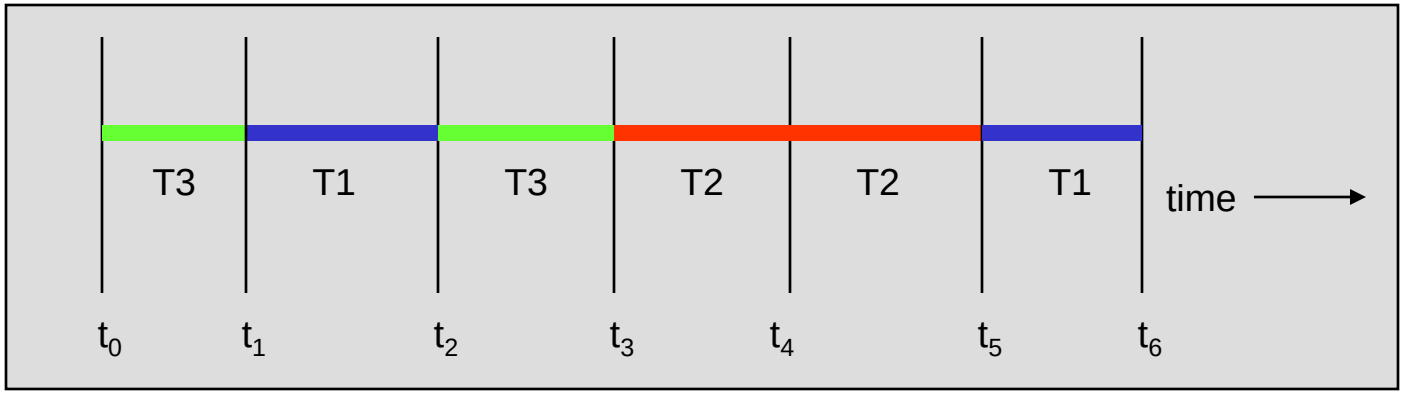
- The execution of any "program" that either accesses (queries) or changes the database contents is called a transaction.
- Serial transactions – two or more transactions are processed in serial fashion with one transaction starting and completing before the next transaction begins execution. At no time, is more than one transaction processing or making progress.
- Interleaved transactions – two or more transactions are processed concurrently with only one transaction at a time actually making progress. This most often occurs on a single multi-programmed CPU.
- Simultaneous transactions – two or more transactions are processed concurrently with any number progressing at one time. This is a multiple CPU situation.

Introduction to Transaction Processing (cont.)



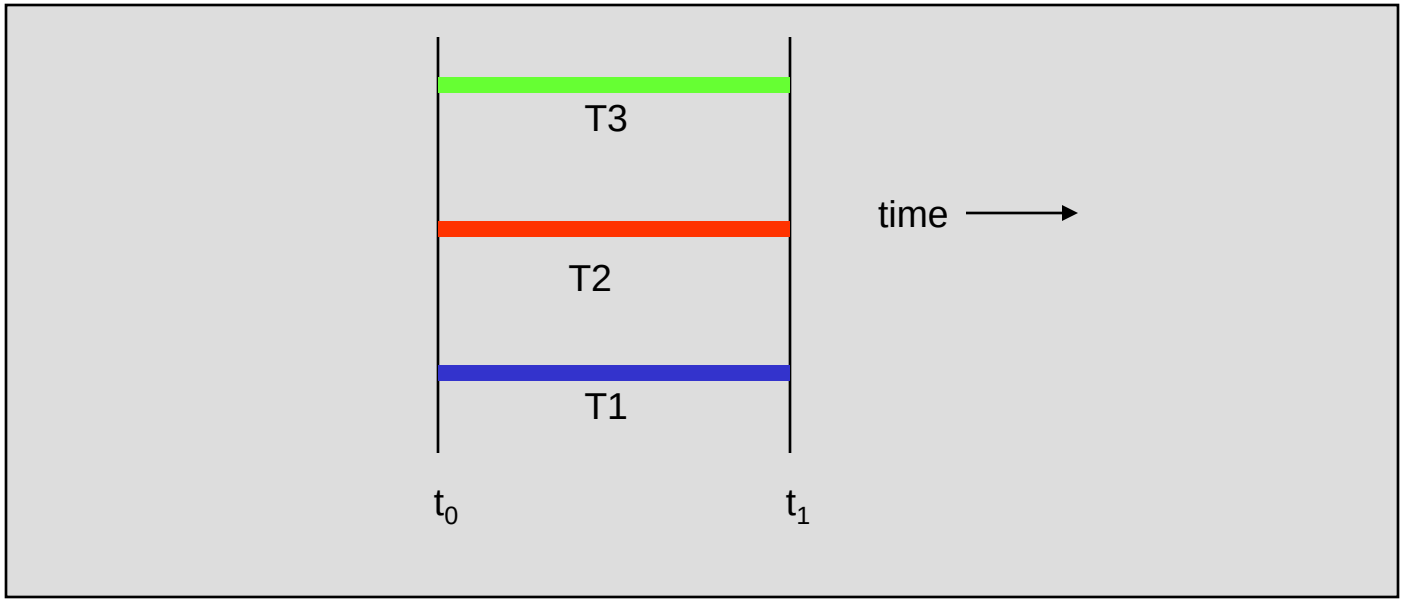
Serial transactions (unknown number of CPUs)

Introduction to Transaction Processing (cont.)



Interleaved transactions (single CPU)

Introduction to Transaction Processing (cont.)



Simultaneous transactions (3 CPUs shown)

Introduction to Transaction Processing (cont.)

- When viewed at the transaction level, any transaction has the potential to access the database in two ways:
  - read(item): reads the value of some database item.
  - write(item): write the value of an item into the database.
- These are not atomic operations.
- To read an item the following must occur:
  - find the address of the disk block that contains the item.
  - copy the disk block into buffer (if not already present).
  - copy the item from the buffer into the "program".

Introduction to Transaction Processing (cont.)

- To write an item the following must occur:
  - find the address of the disk block that contains the item.
  - copy the disk block into buffer (if not already present).
  - copy the item from the buffer into the "program".
  - store the updated block from the buffer back onto the disk (at some point in time, usually not immediately).
- When to write back is typically up to the recovery system of the database and may involve OS control.
- Too early of a write back may cause unnecessary data transfers.
- Too late of a write back may cause unnecessary blocking.

Concurrency Control

- Given a consistent (correct?) state of the database as input an individually correct transaction will produce a correct state of the database as output, if that transaction is executed in isolation.
- The goal of concurrency control is to allow multiple transactions to be processing simultaneously within a certain time period with all of the concurrent transactions producing a correct state of the database at then end of their concurrent execution.

Concurrency Control – Why Its Needed

- There are many different types of conflicts that can occur between concurrently executing processes if concurrency control is not enforced.

Lost Update Problem (A Write-Write Conflict) (overwriting uncommitted data)

- Suppose two distinct transactions T1 and T2 are processing in the concurrent order shown below accessing a common value *n*.

time	action	comment
t0	T1 performs read(n)	suppose T1 reads value of n = 5
t1	T2 performs read(n)	T2 will read a value of n = 5
t2	T1 performs write(n-1)	T1 will write value of n = 4
t3	T2 performs write(n-1)	T2 will also write value of n = 4!

- Problem: The update performed by T1 at time t2 is "lost" since the update written by T2 at time t3 overwrites the previous value.

Handling the Lost Update Problem

- There are several different ways in which the lost update problem can be handled.

1. Prevent T2 from reading the value of n at time t1 on the grounds that T1 has already read the value of n and may therefore update the value.

2. Prevent T1 from writing the value of n-1 at time t2 on the grounds that T2 has also read the same value of n and would therefore be executing on an obsolete value of n, since T2 cannot re-read n.

3. Prevent T2 from writing the value of n-1 at time t3 on the grounds that T1 has already updated the value of n and since T1 preceded T2, then T2 is using an obsolete value of n.

- The first two of these techniques can be implemented using locking protocols, while the third technique can be implemented with time-stamping. We'll see both of these techniques later.

The Dirty Read Problem

Dirty Read Problem (A Write-Read Conflict)(reading uncommitted data)

- Suppose two distinct transactions T1 and T2 are processing in the concurrent order shown below accessing a common value *n*.

time	action	comment
t0	T1 performs read(n)	suppose T1 reads value of n = 5
t1	T1 performs write(n-1)	T1 writes a value of n = 4
t2	T2 performs read(n)	T2 will read value of n = 4
t3	T1 aborts	T2 is executing with a "bad" value of n

- Problem: T2 is operating with a value that was written by a transaction that aborted prior to the completion of T2. When T1 aborts all of its updates must be undone, which means that T2 is executing with a bad value of n and therefore cannot leave the database in a consistent state.

Solution: T2 must also be aborted.

The Unrepeatable Read Problem

Unrepeatable Read Problem (A Read-Write Conflict)

- Suppose two distinct transactions T1 and T2 are processing in the concurrent order shown below accessing a common value *n*.

time	action	comment
t0	T1 performs read(n)	suppose T1 reads value of n = 5
t1	T1 performs read(n)	T1 reads a value of n = 5
t2	T2 performs write(n-1)	T2 will write value of n = 4
t3	T1 performs read(n)	T1 reads a different value of n this time

- Problem: When T1 performs its second read of n, the value is not the same as its first read of n. T1 cannot repeat its read.

Solution: This problem is typically handled with locking which is rather inflexible, but can also be solved with time-stamping.

The Transaction Recovery System

- Whenever a transaction is submitted to the DBMS for execution, the DBMS is responsible for making sure that either:

1. All operations of the transaction are completed successfully and their effect is permanently recorded in the database, or

2. The transaction has no effect whatsoever on the the database or any other transaction.

- If a transaction fails after executing some of its operations, problems will occur with consistency in the database. Therefore, if a transaction fails after its is initiated but prior to its commitment, all of the effects of that transaction must be undone from the database.

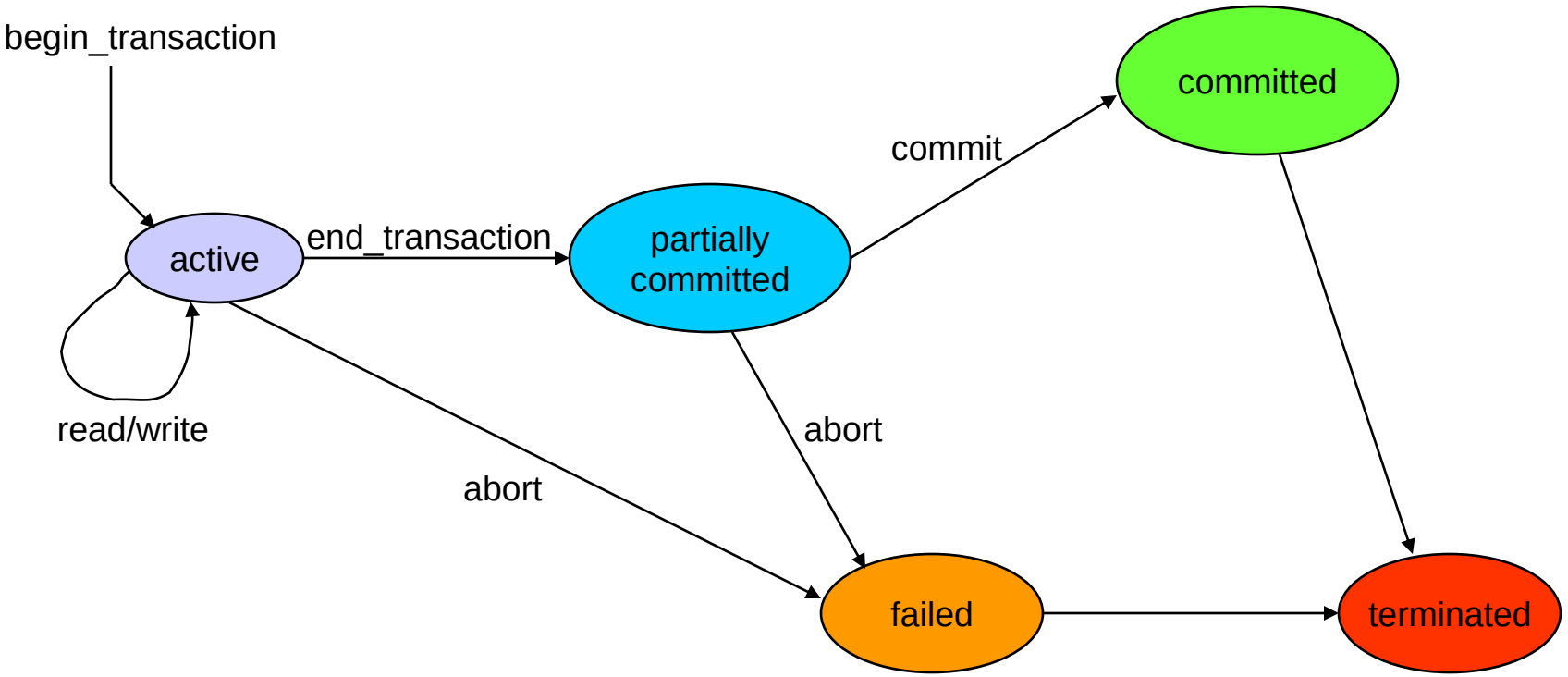
The Transaction Recovery System (cont.)

- Types of failures for a transaction:
  - System crash – some type of hardware or system failure occurs.
  - Transaction error – integer overflow, division by zero, operator intervention.
  - Local errors or exception conditions – required data is not available.
  - Concurrency control enforcement – serializability is violated, deadlock detection victim selection, etc.
  - Disk errors – error correction/detection.
  - Physical problems – fire, power failure, operator error, etc.

The States of a Transaction

- A transaction can be in one of several different states:
  - begin_transaction: marks the beginning of the transaction.
  - read/write: specifies the various db operations performed by the transaction.
  - end_transaction: specifies that all read/write operations have ended and the transaction is ready to terminate. Note: this does not actually end the transactions time in the system – now it heads to the concurrency control system for verification.
  - commit: marks the successful end of the transaction – its effects are now permanent (committed) in the database and cannot be undone.
  - abort (rollback): marks the unsuccessful end of the transaction. All changes and effects in the database must be undone and/or other transactions must be aborted. No changes are committed for the transaction.

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 no 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

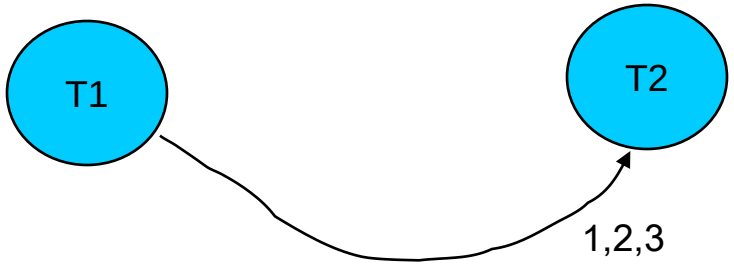
//output: no – if S is not conflict serializable, a serial schedule S* equivalent to S otherwise.

Conflict_Serializable(S)

1. for each transaction $T_X \in S$, create a node (in the graph) labeled T_X.
2. (RAW: READ AFTER WRITE) 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. (WAR: WRITE AFTER READ) 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. (WAW: WRITE AFTER WRITE) 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 $S_C = (r_1(a), w_1(a), r_2(a)\ w_2(a), r_1(b), w_1(b))$



1: $r_1(a)$ precedes $w_2(a)$ (WAR)

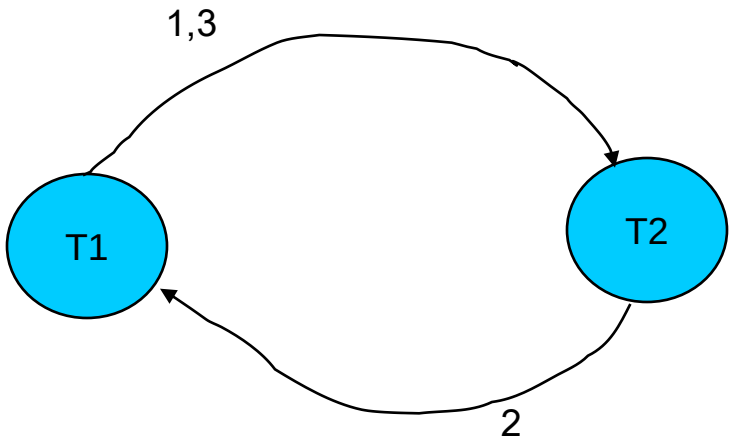
2: $w_1(a)$ precedes $r_2(a)$ (RAW)

3: $w_1(a)$ precedes $w_2(a)$ (WAW)

Graph contains no cycle, so S_C is conflict serializable

Conflict Serializability – Example #1

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



1: $r_1(a)$ precedes $w_2(a)$ (WAR)

2: $r_2(a)$ precedes $w_1(a)$ (WAR)

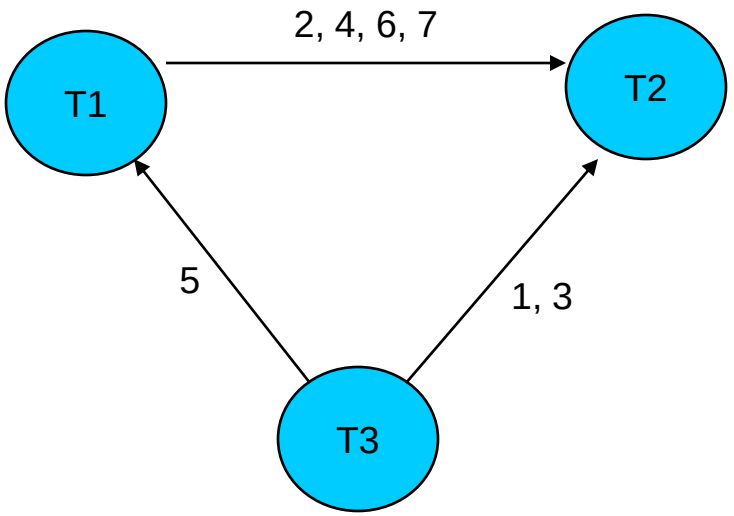
3: $w_1(a)$ precedes $w_2(a)$ (WAW)

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 $r_2(y)$ (RAW)
2	$w_1(x)$ precedes $r_2(x)$ (RAW)
3	$w_3(z)$ precedes $r_2(z)$ (RAW)
4	$w_1(y)$ precedes $r_2(y)$ (RAW)
5	$r_3(y)$ precedes $w_1(y)$ (WAR)
6	$r_1(x)$ precedes $w_2(x)$ (WAR)
7	$r_1(y)$ precedes $w_2(y)$ (WAR)

There are seven other conflicts that can be found in this schedule, but none of them will introduce a cycle. Find the missing seven.

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

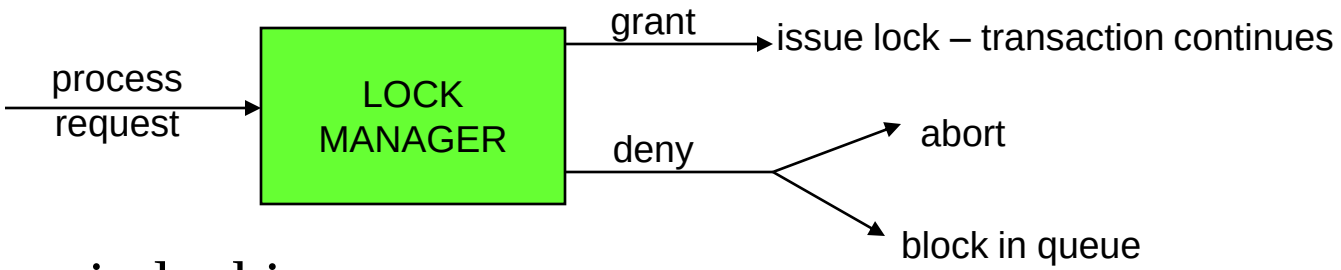
Concurrency Control Techniques

- 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 × 10 × 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.

Types of Locks

- There are different types of locks that locking protocols may utilize.
- The most restrictive systems use only exclusive-locks (X-lock also called a binary lock).
- An exclusive lock permits the transaction which holds the lock exclusive access to the object of the lock.

> If transaction T_X holds an X-lock on object A then no distinct transaction T_Y can obtain an X-lock on object A until transaction T_X releases the X-lock on object A. T_Y is blocked awaiting the X-lock on object A.

- The process of locking and un-locking objects must be indivisible operations within a critical section. There can be no interleaving of issuing and releasing locks.

X-Lock Protocol

Before any transaction T_X can read or write an object A, it must first acquire an X-lock on object A. If the request is granted T_X will proceed with execution. If the request is denied, T_X will be placed into a queue of transactions awaiting the X-lock on object A, until the lock can be granted. After T_X finishes with object A, it must release the X-lock.

- When the lock manager grants a transaction's request for a particular lock, the transaction is said to "hold the lock" on the object.
- Under the X-lock protocol a transaction must obtain, for every object required by the transaction, an X-lock on the object. This applies to both reading and writing operations.

Serializability Under X-Lock Protocol

Algorithm TestSerializiabiltyXLock

//input: a concurrent schedule S under X-lock protocol

//output: if S is serializable, then a serially equivalent schedule S′ is produced, otherwise, no.

TestSerializabilityXLock(S)

1. let S = (a_1, a_2, ..., a_n) where "action" a_i is either (T_X: Xlock A) or (T_X: Unlock A)
2. construct a precedence graph of *n* nodes where *n* is the number of distinct transactions in S.
3. proceed through S as follows:
   - if a_r = (T_X: Unlock A) then look for the next action a_s of the form (T_Y: Xlock A). If one exists, draw an edge in the graph from T_X to T_Y. The meaning of this edge is that in any serially equivalent schedule T_X must precede T_Y.
4. if the graph constructed in step 3 contains a cycle, then S is not equivalent to any serial schedule (i.e., S is not serializable). If no cycle exists, then any topological sort of the graph will yield a serial schedule equivalent to S.

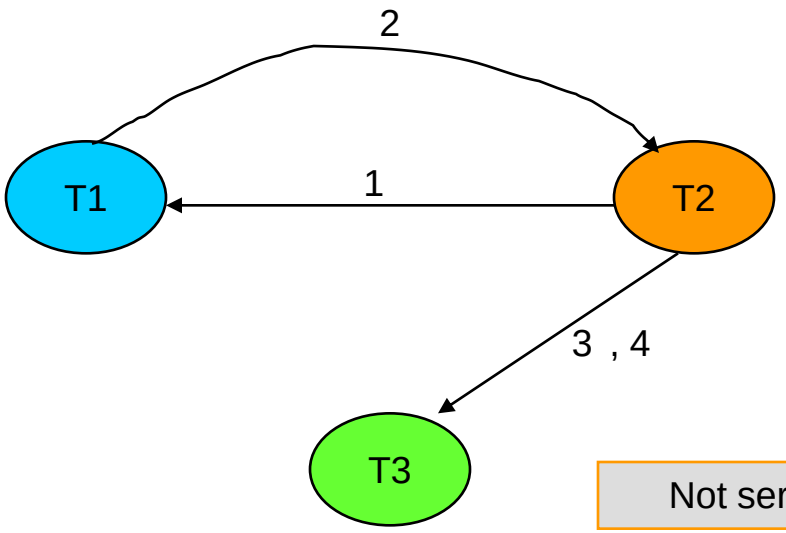
Example - X-Lock Protocol and Serializability

Let S = [(T1: Xlock A), (T2: Xlock B), (T2: Xlock C), (T2: Unlock B),

(T1: Xlock B), (T1: Unlock A), (T2: Xlock A), (T2:Unlock C),

(T2: Unlock A), (T3: Xlock A), (T3: Xlock C), (T1: Unlock B),

(T3: Unlock C), (T3: Unlock A)]



Edge #1: (T2: Unlock B)...(T1:Xlock B)

Edge #2: (T1: Unlock A)...(T2: Xlock A)

Edge #3: (T2: Unlock C)...(T3: Xlock C)

Edge #4: (T2: Unlock A)...(T3: Xlock A)

Not serializable, cycle exists

Problems with X-Lock Protocol

- The X-lock protocol is too restrictive.
- Several transactions that need only to read an object must all wait in turn to gain an X-lock on the object, which unnecessarily delays each of the transactions.
- One solution is to issue different types of locks, called shared-locks (S-locks or read-locks) and write-locks (X-locks).
- The lock manager can grant any number of shared locks to concurrent transactions that need only to read an object, so multiple reading is possible. Exclusive locks are issued to transactions needing to write an object.
- If an X-lock has been issued on an object to transaction T_X, then no other distinct transaction T_Y can be granted either an S-lock or an X-lock until T_X releases the X-lock. If any transaction T_X holds an S-lock on an object, then no other distinct transaction T_Y can be granted an X-lock on the object until all S-locks have been released.

Serializability Under X/S-Lock Protocol

Algorithm TestSerializiabiltyX/SLock

//input: a concurrent schedule S under X/S-lock protocol

//output: if S is serializable, then a serially equivalent schedule S′ is produced, otherwise, no.

TestSerializabilityXLock(S)

1. let S = (a_1, a_2, ..., a_n) where "action" a_i is one of (T_X: Slock A), (T_X: Xlock A) or (T_X: Unlock A).
2. construct a precedence graph of *n* nodes where *n* is the number of distinct transactions in S.
3. proceed through S as follows:
   - if a_x = (T_X: Slock A) and a_y is the next action (if it exists) of the form (T_Y: Xlock A) then draw an edge from T_X to T_Y.
   - if a_x = (T_X: Xlock A) and there exists an action a_z = (T_Z: Xlock A) then draw an edge in the graph from T_X to T_Z. Also, for each action a_y of the form (T_Y: Slock A) where a_y occurs after a_x (T_X: Unlock A) but before a_Z (T_Z: Xlock A) draw an edge from T_X to T_Y. If a_z does not exist, then T_Y is any transaction to perform (T_Y: Slock A) after (T_X: Unlock A).
4. if the graph constructed in step 3 contains a cycle, then S is not equivalent to any serial schedule (i.e., S is not serializable). If no cycle exists, then any topological sort of the graph will yield a serial schedule equivalent to S.

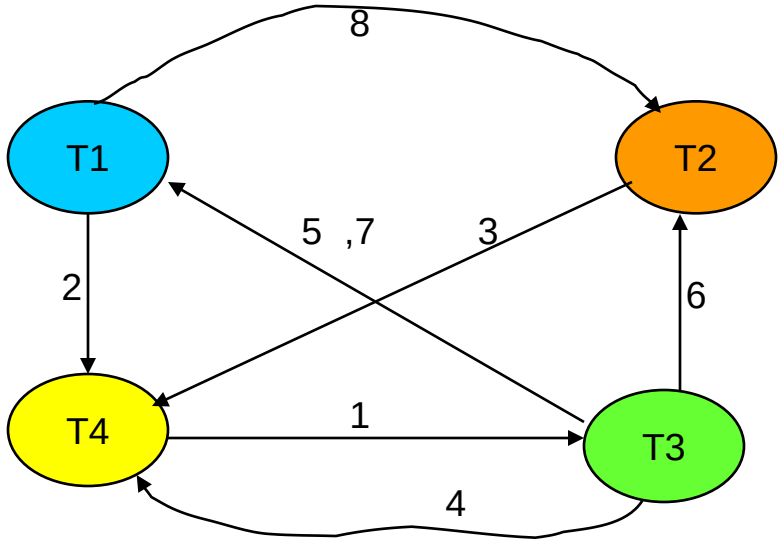
Example – X/S-Lock Protocol and Serializability

Let S = [(T3: Xlock A), (T4: Slock B), (T3: Unlock A), (T1: Slock A),

(T4: Unlock B), (T3: Xlock B), (T2: Slock A), (T3:Unlock B),

(T1: Xlock B), (T2: Unlock A), (T1: Unlock A), (T4: Xlock A),

(T1: Unlock B), (T2: Xlock B), (T4: Unlock A), (T2: Unlock B)]

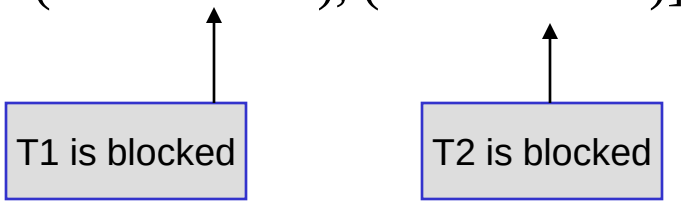


Edge #1: (T4: Slock B)...(T3: Xlock B)
Edge #2: (T1: Slock A)...(T4: Xlock A)
Edge #3: (T2: Slock A)...(T4: Xlock A)
Edge #4: (T3: Xlock A)...(T4: Xlock A)
Edge #5: (T3: Unlock A)...(T1: Slock A)
Edge #6: (T3: Unlock A)...(T2: Slock A)
Edge #7: (T3: Xlock B)...(T1:Xlock B)
Edge #8: (T1: Xlock B)...(T2: Xlock B)

Not serializable, cycle exists

Problems Locking Protocols

- The X-lock protocol can lead to deadlock.
  - For example consider the schedule S = [(T1:Xlock A), (T2: Xlock B), (T1: Xlock B), (T2: Xlock A)]



- While there are many different techniques that can be used to avoid deadlock, most are not suitable to the database environment.

Deadlock Avoidance - Problems Locking Protocols (cont.)

- Impose a total ordering on the objects.
  - Problem is the set of lockable objects is very large and changes dynamically.
  - Many database transactions determine the lockable object based on content and not name.
  - The locking scope of a transaction is typically determined dynamically.
- Two-phase locking protocols.
  - All locks are granted at the beginning of a transaction's processing or no locks are granted. Transactions which cannot acquire all of the locks they need are suspended without being granted any locks.
  - Leads to low data utilization, low-levels of concurrency and livelock.
  - Livelock occurs when a transaction that needs several "popular" items is consistently blocked by transactions which need only one of the popular items.

Concurrency Control: Locking in B^+ Trees

- An often used and straightforward approach to concurrency control for B^+ trees and ISAM indices is to ignore the index structure and treat each page as a data object utilizing some variant of two-phase locking.

- Unfortunately, this simplistic locking strategy leads to very high lock contention in the higher levels of the tree, since each search begins at the route and proceeds along some path to a leaf node.

- Fortunately, there are several much better locking approaches available that exploit the hierarchical nature of the tree index that will ensure serializability and reduce the locking overhead.

Concurrency Control: Locking in B+ Trees (cont.)

- Two observations are important to understand how B+ locking strategies can be developed:

  1. The higher levels of the tree only direct searches. All of the "real" data is in the leaf level.

  2. For insertions, a node must be locked (exclusively) only if a split can propagate up to it from the modified leaf node.

- Searches should obtain shared locks on modes, starting at the root and proceeding along a path to the desired leaf.

- The first observation suggests that a lock on a node can be released as soon as a lock on a child node is obtained, because searches never go back up the tree.

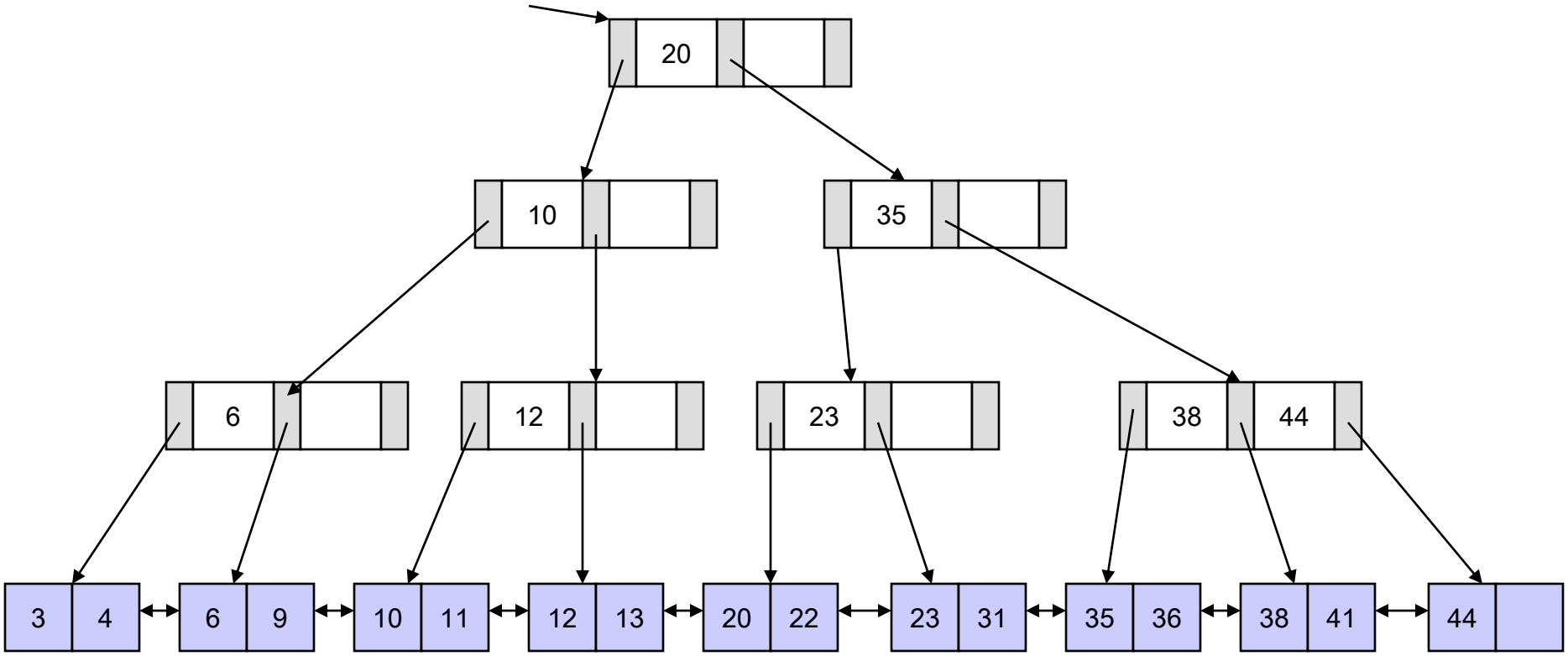
Concurrency Control: Locking in B^+ Trees (cont.)

- A conservative (pessimistic) locking strategy for inserts would be to obtain exclusive locks on all the nodes as we go down from the root to the leaf node that will be modified, because splits can propagate all the way from the leaf to the root in the worst case.
- However, once the child of a node is locked, the lock on that node would only be required to be maintained in the event that a split could propagate back to it.
- Specifically, if the child of this node (on the path to the modified leaf) is not full when it is locked, any split that propagates up to the child can be resolved at the child and will not propagate further up the tree to the current node.
- Therefore, when the child node is locked, the lock on the parent node can be released if the child node is not full.

Concurrency Control: Locking in B^+ Trees (cont.)

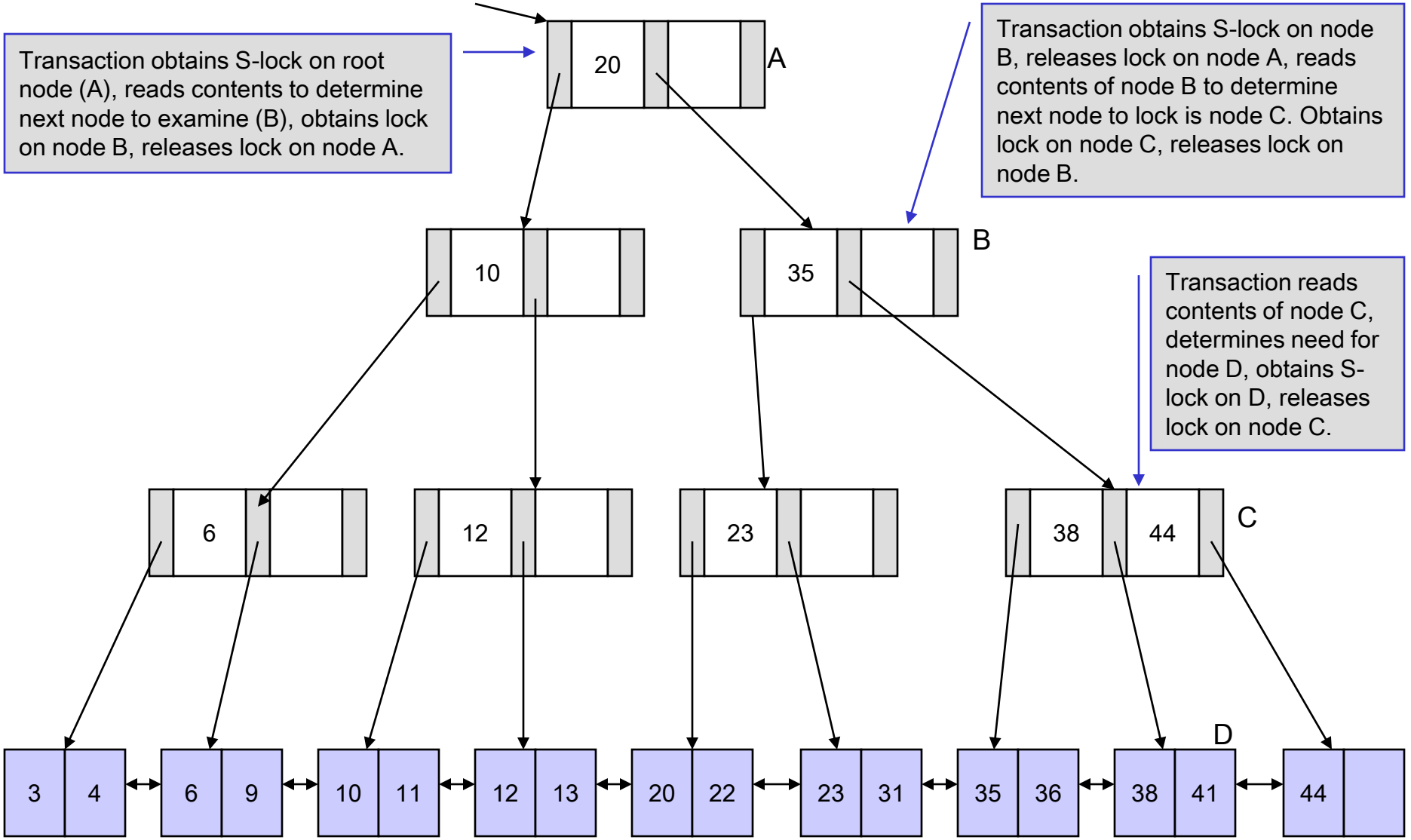
- The locks held by an inserting transaction force any other transaction following the same path to wait at the earliest point (the node closest to the root) that might be affected by the insert.
- This technique of locking a child node and (if possible) releasing the lock on its parent is called lock-coupling.
- The examples on the next few pages illustrate concurrency control in B^+ trees.

Concurrency Control: Locking in B+ Trees (cont.)



Initial B+ tree

Search For Key Value 38



Concurrency Control: Locking in B+ Trees (cont.)

- Notice in the preceding example that the transaction always maintains a lock on one node in the path, to force new transactions that want to read or modify nodes on the same path to wait until the current transaction is done.

- If some other transaction (other than the one doing the search for key value 38) wants to delete the record containing key value 38, it must also traverse the same path from root to node D and is forced to wait until the current transaction has completed.

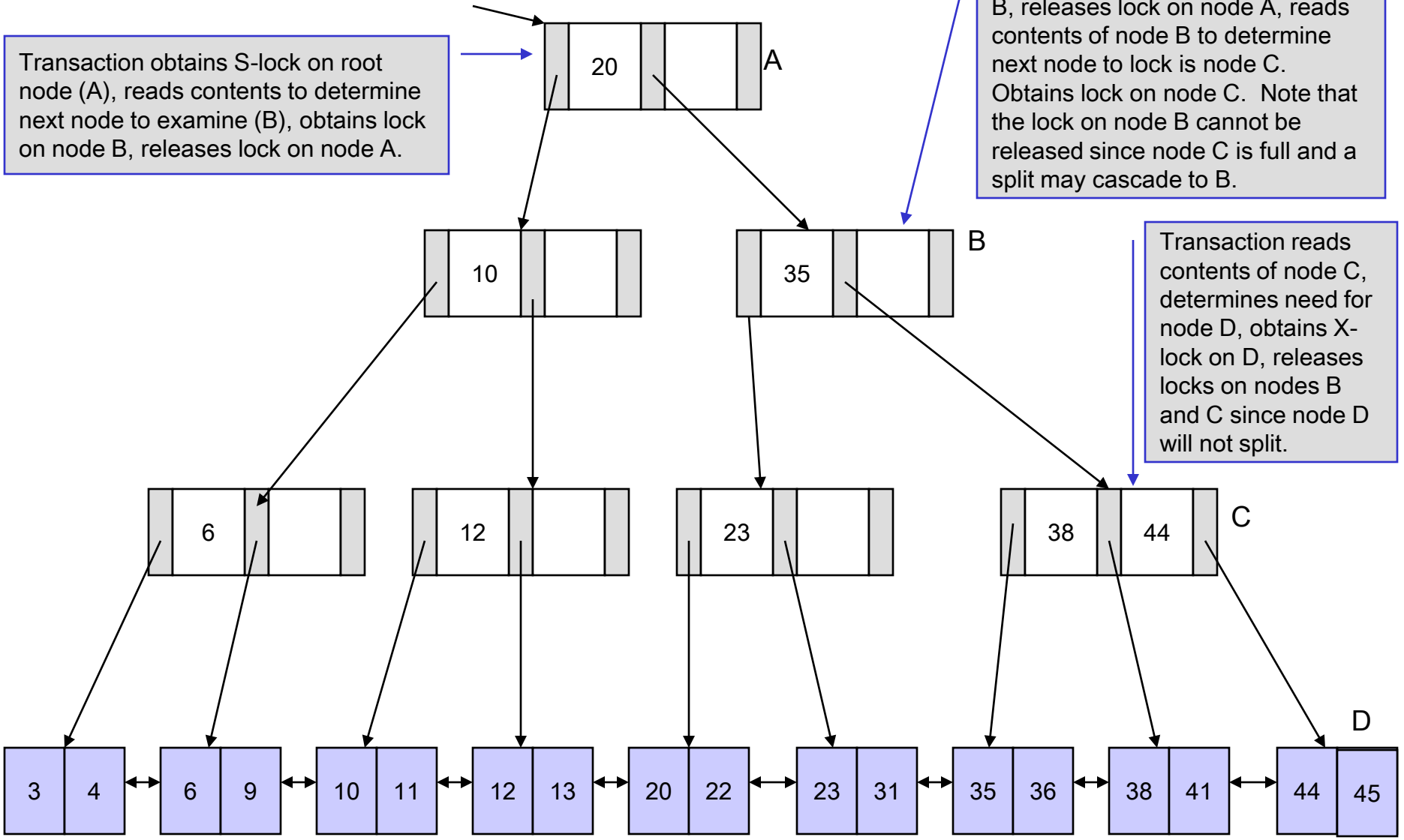
  – Notice that this also implies that if some earlier transaction preceded our transaction searching for key value 38, that our searching transaction would have been similarly delayed as the earlier transaction would hold the lock on some node in this path.

Insert Key Value 45

Transaction obtains S-lock on root node (A), reads contents to determine next node to examine (B), obtains lock on node B, releases lock on node A.

Transaction obtains S-lock on node B, releases lock on node A, reads contents of node B to determine next node to lock is node C. Obtains lock on node C. Note that the lock on node B cannot be released since node C is full and a split may cascade to B.

Transaction reads contents of node C, determines need for node D, obtains X-lock on D, releases locks on nodes B and C since node D will not split.

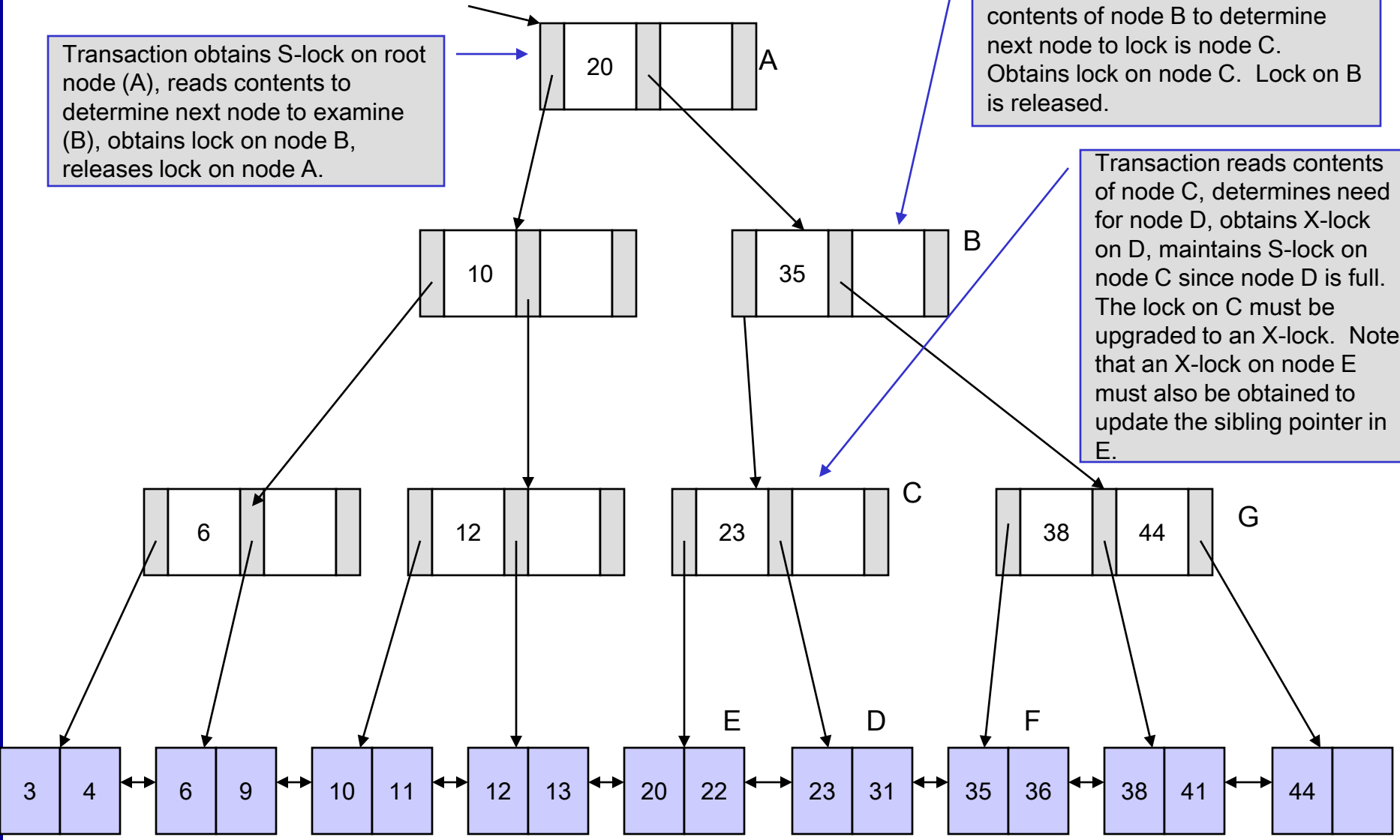


Insert Key Value 25

Transaction obtains S-lock on root node (A), reads contents to determine next node to examine (B), obtains lock on node B, releases lock on node A.

Transaction obtains S-lock on node B, releases lock on node A, reads contents of node B to determine next node to lock is node C. Obtains lock on node C. Lock on B is released.

Transaction reads contents of node C, determines need for node D, obtains X-lock on D, maintains S-lock on node C since node D is full. The lock on C must be upgraded to an X-lock. Note that an X-lock on node E must also be obtained to update the sibling pointer in E.



Concurrency Control: Locking in B^+ Trees (cont.)

- Notice in the preceding example if another transaction holds an S-lock on node C and also wants to access node D, then a deadlock situation will occur because the inserting transaction holds an X-lock on node D.
  - Inserting transaction holds an X-lock on node D, and is requesting an upgrade to an X-lock on node C. The upgrade request cannot be granted because the other transaction holds an S-lock on node C, further, the other transaction's request to access node D cannot be granted since the inserting transaction already holds an X-lock on node D.
- The previous example also illustrates an interesting point about sibling pointers: when node D splits, the new node must be added to the left of node D, otherwise the node whose sibling pointer needs to be changed would be node F, which has a different parent.
  - To modify a sibling pointer on F, we would have to lock its parent, node G (and possibly ancestors of G, in order to lock G).

Deadlock Avoidance - Problems Locking Protocols (cont.)

- There is also a timestamp based protocol (under locking – don't confuse this with timestamp based concurrency controls we'll see later) to prevent deadlock under locking protocols.

- A timestamp is a unique identifier assigned to each transaction based upon the time a transaction begins.

  – if $ts(T_X) < ts(T_Y)$ then T_X is the older transaction and T_Y is the younger transaction.

  – In resolving deadlock issues, the system uses the value of the timestamp to determine if a transaction should wait or rollback. Locking is still used to control concurrency.

  – Under rollback a transaction retains its original timestamp.

Deadlock Resolution – Wait or Die

- Assume that T_X requests an object whose lock is held by T_Y.
- This is a non-preemptive strategy where if $ts(T_X) < ts(T_Y)$ (T_X is older than T_Y) then T_X is allowed to wait on T_Y, otherwise T_X dies (is rolled back). T_Y continues to hold the lock and T_X subsequently restarts with its original timestamp.
  - if request is made by older transaction – it waits on the younger transaction.
  - if request is made by younger transaction – it dies.
- Example: let ts(T1) = 5, ts(T2) = 10, ts(T3) = 15

  Suppose T2 requests object held by T1. T2 is younger than T1, T2 dies.

  Suppose T1 requests object held by T2. T1 is older than T2, T1 waits.

Deadlock Resolution – Wound or Wait

- Assume that T_X requests an object whose lock is held by T_Y.
- This is a preemptive strategy where if $ts(T_X) < ts(T_Y)$ (T_X is older than T_Y) then T_Y is aborted (T_X wounds T_Y). T_X preempts the lock and continues. Otherwise, T_X waits on T_Y.
  - if request is made by the younger transaction – it waits on the older transaction.
  - if request is made by older transaction – it preempts the lock and the younger transaction dies.
- Example: let ts(T1) = 5, ts(T2) = 10, ts(T3) = 15

  Suppose T2 requests object held by T1. T2 is younger than T1, T2 waits.

  Suppose T1 requests object held by T2. T1 is older than T2, T1 gets lock and T2 dies.

Timestamp Deadlock Resolution

- Both wait or die and wound or wait protocols avoid starvation. At any point in time there is a transaction with the smallest timestamp (i.e., oldest transaction) and it will not be rolled back in either scheme.

Operational Differences

- In wait or die, the older transaction waits for the younger one to release its locks, thus, the older a transaction gets, the more it will wait. In wound or wait, the older transaction never waits.
- In wait or die protocol if transaction T1 dies and is rolled back it will in probably be re-issued and generate the same set of requests as before. It is possible for T1 to die several times before it will be granted the lock it is requesting as the older transaction is still using the lock. Whereas, in wound or wait, it would restart once and then be blocked. Typically, the wound or wait protocol will result in fewer roll backs than does the wait or die protocol.

Deadlock Avoidance vs. Detection and Resolution

- If the deadlock prevention or avoidance mechanism is not 100% effective, then it is possible for a set of transactions to become deadlocked.
- Handling this problem can be achieved in one of two basic manners: optimistically or pessimistically.
- Optimistic approaches tend to wait for deadlock to occur before doing anything about it, while pessimistic approaches tend to make sure that deadlock cannot occur.
- Optimistic approaches use detection and resolution schemes while pessimistic approaches use avoidance mechanisms.

Deadlock Detection and Resolution

- Deadlock detection and resolution involves two phases: detection of deadlock and its resolution.
- Deadlock detection is commonly done with wait-for graphs (a form of a precedence graph). Each node in the graph represents a transaction in the system. An edge from transaction T_X to transaction T_Y indicates that T_X is waiting on an object currently held by T_Y. A deadlock is detected if the graph contains a cycle.
- The resolution phase or the recovery from the deadlock, essentially amounts to selecting a victim of the deadlock to be rolled back, thus breaking the deadlock.

Deadlock Detection and Resolution (cont.)

- Selection of a victim to resolve the deadlock can be based upon many different things:
  - how long has the transactions been processing?
  - how much longer does the transaction require to complete?
  - how much data has been read/written?
  - how many data items are still needed?
  - how many transactions will need to be rolled back?
- Once a victim has been selected you can decide how far back to roll it. It is not always necessary for a complete restart.
- Deadlock detection and resolution requires some mechanism to prevent starvation from occurring. Typically this is done by limiting the number of times a single transaction can be identified as the "victim".

Timestamping Concurrency Control

- No locking is used with timestamp concurrency control. Do not confuse this topic with the timestamped method for avoiding deadlock under locking.
- As before, each transaction is issued a unique timestamp indicating the time it arrived in the system.
- The size of the timestamp varies from system to system, but must be sufficiently large to cover transactions processing over long periods of time.
- Assignment of the timestamp is typically handled by the long-term scheduler as transactions are removed from some sort of input queue.

Timestamping Concurrency Control (cont.)

- In addition to the transaction's timestamp, each object in the database has associated with it two timestamps:
  - read timestamp – denoted rts(object), and it represents the highest timestamp of any transaction which has successfully read this object.
  - write timestamp – denoted wts(object), and it represents the highest timestamp of any transaction to successfully write this object.
- As with locking the granularity of an "object" in the database becomes a concern here, since the overhead of the timestamps can be considerable if the granularity is too fine.

Timestamp Ordering Protocol

READ – transaction T_X performs read(object)

if ts(T_X) < wts(object)

then rollback T_X // implies that the value of the object has been written by a // transaction T_Y which is younger than T_X

else // ts(T_X) >= wts(object)

execute read(object)

set rts(object_ = max{ rts(object), ts(T_X)}

WRITE – transaction T_X performs write(object)

if ts(T_X) < rts(object)

then rollback T_X //implies that the value of the object being produced by T_X was //read by a transaction T_Y which is younger than T_X and T_Y //assumed the value of the object was valid.

else if ts(T_X) < wts(object)

then ignore write(object) //implies that T_X is attempting to write an "old" //value which has been updated by a younger //transaction.

else

execute write(object

set wts(object) = max{wts(object), ts(T_X)}

Explanation of the Ignore Write Rule

- In the timestamp ordering protocol, when the timestamp of the transaction attempting to write an object is less than the write timestamp of the object of concern, the write is simply ignored.
- This is known as Thomas's write rule.
- Suppose that we have two transactions T1 and T2 where T1 is the older transaction. T1 attempts to write object X. If ts(T1) < wts(X) then if T2 was the last transaction to write X, wts(X) = ts(T2) and between the time T2 wrote X and T1 attempted to write X, no other transaction Tn read X or otherwise rts(X) > ts(T1) and T1 would have aborted when attempting to write X. Thus T1 and T2 have read the same value of X and since T2 is younger, the value that would have been written by T1 would simply have been overwritten by T2, so T1's write can be ignored.

Example - Timestamp Ordering Protocol

	Transactions				Objects		
time	T1	T2	T3	Action	A	B	C
initial	ts = 200	ts = 150	ts = 175		rts = 0 wts = 0	rts = 0 wts = 0	rts = 0 wts = 0
1	read B			ts(T1) >= wts(B), OK		rts = 200	
2		read A		ts(T2) >= wts(A), OK	rts = 150		
3			read C	ts(T3) >= wts(C), OK			rts = 175
4	write B			ts(T1) >= wts(B), OK		wts = 200	
5	write A			ts(T1) >= wts(A), OK	wts = 200		
6		write C		ts(T2) < rts(C), ABORT T2			
7			write A	ts(T3) < wts(A), IGNORE			
final					rts = 150 wts = 200	rts = 200 wts = 200	rts = 175 wts = 0

Multiversion Concurrency Control

- Multiversion concurrency control falls into the optimistic method of concurrency control and also utilizes transaction timestamps to ensure serializability.
- The basic goal of multiversion concurrency control is to never block a transaction from reading a database object.
- This is done by maintaining several versions of each database object (for objects in play), each with a write timestamp, and each transaction requesting to read the object will read the most recent version of the object whose timestamp precedes that transaction's timestamp.

Multiversion Concurrency Control (cont.)

- If a transaction T_i wants to write an object, concurrency control must ensure that the object has not already be read by some other transaction T_j such that $ts(T_i) < ts(T_j)$.
- If transaction T_i is allowed to write the object, that change should be seen by T_j for serializability, but obviously T_j, which read the object at some time in the past would not see the effect of the write performed by T_i.
- To check this condition, every object also has an associated read timestamp, and whenever a transaction reads an object, the read timestamp is set to the maximum of its current value and the timestamp of the transaction performing the read.
- If T_i wants to write object O and $ts(T_i) < rts(O)$, then T_i is aborted and restarted with a new, larger timestamp. Otherwise, T_i creates a new version of O and sets the read and write timestamps of the new version to $ts(T_i)$.