Implementing Isolation

Chapter 20

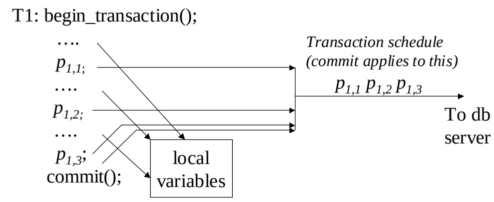
The Issue

- Maintaining database correctness when many transactions are accessing the database concurrently
  - – Assuming each transaction maintains database correctness when executed in isolation

Isolation

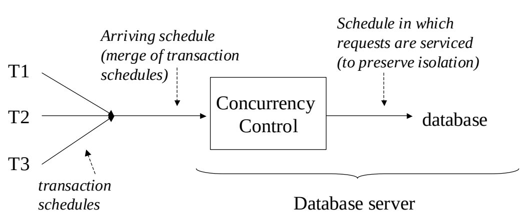
- Serial execution:
  - – Since each transaction is consistent and isolated from all others, schedule is guaranteed to be correct for all applications
  - – Inadequate performance
    - Since system has multiple asynchronous resources and transaction uses only one at a time
- Concurrent execution:
  - – Improved performance (multiprogramming)
  - – Some interleavings produce correct result, others do not
  - – We are interested in concurrent schedules that are *equivalent* to serial schedules. These are referred to as *serializable* schedules.

Transaction Schedule



- Consistent - performs correctly when executed in isolation starting in a consistent database state
  - – Preserves database consistency
  - – Moves database to a new state that corresponds to new real-world state

Schedule



Schedule

- Representation 1:

  T_1: p_1 p_2 p_3 p_4
  T_2: p_1 p_2
  time $\rightarrow$

- Representation 2:

  $p_{1,1}$ $p_{1,2}$ $p_{2,1}$ $p_{1,3}$ $p_{2,2}$ $p_{1,4}$
  time $\rightarrow$

Concurrency Control

- Transforms arriving interleaved schedule into a correct interleaved schedule to be submitted to the DBMS
  - Delays servicing a request (reordering) - causes a transaction to wait
  - Refuses to service a request - causes transaction to abort
- Actions taken by concurrency control have performance costs
  - Goal is to avoid delaying or refusing to service a request

7

Correct Schedules

- Interleaved schedules *equivalent* to serial schedules are the only ones guaranteed to be correct for *all* applications
- Equivalence based on *commutativity* of operations
- **Definition:** Database operations p_1 and p_2 commute if, *for all initial database states*, they
  (1) *return the same results* and
  (2) *leave the database in the same final state*
  when executed in either order.

$p_1 \ p_2 \qquad p_2 \ p_1$

8

Conventional Operations

- Read
  - *r(x, X)* - copy the value of database variable *x* to local variable *X*
- Write
  - *w(x, X)* - copy the value of local variable *X* to database variable *x*
- We use $r_1(x)$ and $w_1(x)$ to mean a read or write of *x* by transaction T_1

9

Commutativity of Read and Write Operations

- p_1 commutes with p_2 if
  - They operate on different data items
    - $w_1(x)$ commutes with $w_2(y)$ and $r_2(y)$
  - Both are reads
    - $r_1(x)$ commutes with $r_2(x)$
- Operations that do not commute *conflict*
    - $w_1(x)$ conflicts with $w_2(x)$
    - $w_1(x)$ conflicts with $r_2(x)$

10

Equivalence of Schedules

- An interchange of adjacent operations *of different transactions* in a schedule creates an equivalent schedule if the operations commute

  S_1: $S_{1,1}\ p_{i,j}\ p_{k,l}\ S_{1,2}$ where $i \neq k$
  S_2: $S_{1,1}\ p_{k,l}\ p_{i,j}\ S_{1,2}$
  - Each transaction computes the same results (since operations return the same values in both schedules) and hence writes the same values to the database.
  - The database is left in the same final state (since the state seen by $S_{1,2}$ is the same in both schedules).

11

Equivalence of Schedules

- Equivalence is transitive: If S_1 can be derived from S_2 by a series of such interchanges, S_1 is equivalent to S_2

12

Example of Equivalence

conflict

S_1: $r_1(x)\ r_2(x)\ w_2(x)\ r_1(y)\ w_1(y)$

S_2: $r_1(x)\ r_2(x)\ r_1(y)\ w_2(x)\ w_1(y)$

S_3: $r_1(x)\ r_1(y)\ r_2(x)\ w_2(x)\ w_1(y)$

S_4: $r_1(x)\ r_1(y)\ r_2(x)\ w_1(y)\ w_2(x)$

S_5: $r_1(x)\ r_1(y)\ w_1(y)\ r_2(x)\ w_2(x)$

conflicting operations ordered in same way

S_1 is equivalent to S_5
S_5 is the serial schedule T_1, T_2
S_1 is *serializable*
S_1 is *not* equivalent to the serial schedule T_2, T_1

13

Example of Equivalence

T_1: *begin transaction*
read (x, X);
X = X+4;
write (x, X);
commit;

T_2: *begin transaction*
read (x,Y);
write (y,Y);
commit;

	<u>*initial state*</u>		<u>*final state*</u>
	$x=1, y=3$	$r_1(x)\ r_2(x)\ w_2(y)\ w_1(x)$	$x=5, y=1$
Interchange commuting operations	$x=1, y=3$	$r_2(x)\ w_2(y)\ r_1(x)\ w_1(x)$ T_2 T_1	$x=5, y=1$
Interchange conflicting operations	$x=1, y=3$	$r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)$ T_1 T_2	$x=5, y=5$

14

Serializable Schedules

- S is serializable if it is equivalent to a serial schedule
- Transactions are totally isolated in a serializable schedule
- A schedule is correct for *any* application if it is a serializable schedule of consistent transactions
- The schedule :
  $r_1(x)\ r_2(y)\ w_2(x)\ w_1(y)$
  is *not* serializable

15

Isolation Levels

- Serializability provides a *conservative* definition of correctness
  - For a particular application there might be many acceptable *non*-serializable schedules
  - Requiring serializability might degrade performance
- DBMSs offer a variety of isolation levels:
  - SERIALIZABLE is the most stringent
  - Lower levels of isolation give better performance
    - *Might* allow incorrect schedules
    - *Might* be adequate for some applications

16

Serializable

- **Theorem** - Schedule S_1 can be derived from S_2 by a sequence of commutative interchanges if and only if conflicting operations in S_1 and S_2 are ordered in the same way

  Only If: Commutative interchanges do not reorder conflicting operations

  If: A sequence of commutative interchanges can be determined that takes S_1 to S_2 since conflicting operations do not have to be reordered (see text)

17

Conflict Equivalence

- **Definition**- Two schedules, S_1 and S_2, of the same set of operations are *conflict equivalent* if conflicting operations are ordered in the same way in both
  - Or (using theorem) if one can be obtained from the other by a series of commutative interchanges

18

Conflict Equivalence

- **Result**- A schedule is serializable if it is conflict equivalent to a serial schedule

$$\underbrace{r_1(x)\ w_2(x)}_{conflict}\ \underbrace{w_1(y)\ r_2(y)}_{conflict} \equiv r_1(x)\ w_1(y)\ w_2(x)\ r_2(y)$$

- If in S transactions T_1 and T_2 have several pairs of conflicting operations ($p_{1,1}$ conflicts with $p_{2,1}$ and $p_{1,2}$ conflicts with $p_{2,2}$) then $p_{1,1}$ must precede $p_{2,1}$ and $p_{1,2}$ must precede $p_{2,2}$ (or vice versa) in order for S to be serializable.

19

View Equivalence

- Two schedules of the same set of operations are *view equivalent* if:
  - Corresponding read operations in each return the same values (hence computations are the same)
  - Both schedules yield the same final database state
- Conflict equivalence implies view equivalence.
- View equivalence *does not* imply conflict equivalence.

20

View Equivalence

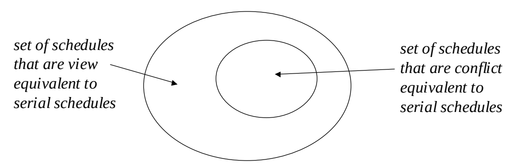
T_1: $w(y)$ $w(x)$
T_2: $r(y)$ $w(x)$
T_3: $w(x)$

T_1:		$w(y)$	$w(x)$		
T_2:	$r(y)$			$w(x)$	
T_3:					$w(x)$

- Schedule *is not* conflict equivalent to a serial schedule
- Schedule has same effect as serial schedule $T_2\ T_1\ T_3$. It *is* view equivalent to a serial schedule and hence serializable

21

Conflict vs View Equivalence



- A concurrency control based on view equivalence should provide better performance than one based on conflict equivalence since less reordering is done but …
- It is difficult to implement a view equivalence concurrency control

22

Conflict Equivalence and Serializability

- Serializability is a conservative notion of correctness and conflict equivalence provides a conservative technique for determining serializability
- However, a concurrency control that guarantees conflict equivalence to serial schedules ensures correctness and is easily implemented

23

Serialization Graph of a Schedule, S

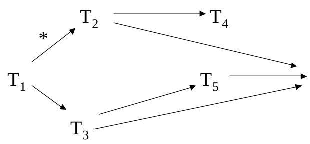
- Nodes represent transactions
- There is a directed edge from node T_i to node T_j if T_i has an operation $p_{i,k}$ that conflicts with an operation $p_{j,r}$ of T_j and $p_{i,k}$ precedes $p_{j,r}$ in S
- **Theorem** - A schedule is conflict serializable if and only if its serialization graph has no cycles

24

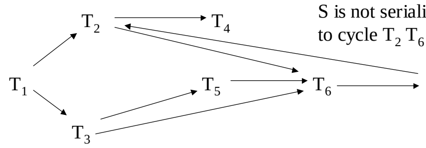
Example

Conflict (*)

S: … $p_{1,i}$, …, $p_{2,j}$, …



S is serializable in order $T_1\ T_2\ T_3\ T_4\ T_5\ T_6\ T_7$



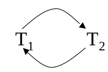
S is not serializable due to cycle $T_2\ T_6\ T_7\ T_2$

25

Intuition: Serializability and Nonserializability

- Consider the nonserializable schedule

  $r_1(x)\ w_2(x)\ r_2(y)\ w_1(y)$



- Two ways to think about it:
  - Because of the conflicts, the operations of T_1 and T_2 cannot be interchanged to make an equivalent serial schedule
  - Because T_1 read *x* before T_2 wrote it, T_1 must precede T_2 in any ordering, and because T_1 wrote *y* after T_2 read it, T_1 must follow T_2 in any ordering --- clearly an impossibility

26

Recoverability: Schedules with Aborted Transactions

T_1 : *r (x) w(y) commit*
T_2: *w(x)* *abort*

- T_2 has aborted but has had an indirect effect on the database – schedule is *un*recoverable
- **Problem**: T_1 read uncommitted data - *dirty read*
- **Solution:** A concurrency control is *recoverable* if it does not allow T_1 to commit until all other transactions that wrote values T_1 read have committed

T_1 : *r (x) w(y)* request commit *abort*
T_2: *w(x)* *abort*

27

Cascaded Abort

- Recoverable schedules solve abort problem but allow *cascaded abort*: abort of one transaction forces abort of another

T_1: *r (y) w(z)* *abort*
T_2: *r (x) w(y)* *abort*
T_3: *w(x)* *abort*

- Better solution: prohibit dirty reads

28

Dirty Write

- *Dirty write*: A transaction writes a data item written by an active transaction
- Dirty write complicates rollback:

no rollback necessary

T_1: *w(x)* *abort*
T_2 : *w(x)* *abort*

what value of x should be restored?

29

Strict Schedules

- *Strict schedule*: Dirty writes and dirty reads are prohibited
- Strict and serializable are two different properties
  - Strict, non-serializable schedule:

    $r_1(x)\ w_2(x)\ r_2(y)\ w_1(y)\ c_1\ c_2$
  - Serializable, non-strict schedule:

    $w_2(x)\ r_1(x)\ w_2(y)\ r_1(y)\ c_1\ c_2$

30

Concurrency Control

- Concurrency control cannot see entire schedule:
  - It sees one request at a time and must decide whether to allow it to be serviced
- Strategy: Do not service a request if:
  - It violates strictness or serializability, or
  - There is a possibility that a subsequent arrival might cause a violation of serializability

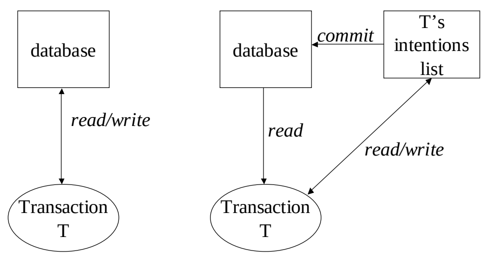
31

Models of Concurrency Controls

- **Immediate Update** – (*the model we have discussed*)
  - A write updates a database item
  - A read copies value from a database item
  - Commit makes updates durable
  - Abort undoes updates
- **Deferred Update –** *(we will discuss this later)*
  - A write stores new value in the transaction's intentions list (does *not* update the database)
  - A read copies value from the database or the transaction's intentions list
  - Commit uses intentions list to durably update database
  - Abort discards intentions list

32

Immediate vs. Deferred Update



Immediate Update

Deferred Update

33

Models of Concurrency Controls

- **Pessimistic** –
  - A transaction requests permission for each database (read/write) operation
  - Concurrency control can:
    - *Grant* the operation (submit it for execution)
    - *Delay* it until a subsequent event occurs (commit or abort of another transaction), or
    - *Abort* the transaction
  - Decisions are made *conservatively* so that a commit request can *always* be granted
    - Takes precautions even if conflicts do not occur

34

Models of Concurrency Controls

- **Optimistic -**
  - Request for database operations (read/write) are *always granted*
  - Request to commit *might be denied*
    - Transaction is aborted if it performed a non-serializable operation
  - Assumes that conflicts are not likely

35

Immediate-Update Pessimistic Control

- The most commonly used control
- Consider first a simple case
  - Suppose such a control allowed a transaction, T_1 , to perform some operation and then, while T_1 was still active ,it allowed another transaction, T_2 , to perform a conflicting operation
  - The schedule would not be strict and so this situation cannot be allowed
    - But consider a bit further what might happen …

36

Immediate-Update Pessimistic Control

- If T_1 executes $op_1(x)$ and then T_2 executes a conflicting operation, $op_2(x)$, T_2 must follow T_1 in any equivalent serial schedule.
- **Problem**: If T_1 and T_2 later make conflicting accesses to *y*, control cannot allow ordering $op'_2(y), op'_1(y)$
  - control has to use transitive closure of transaction ordering to prevent loop in serialization graph (too complicated)
- **Worse problem**:

$$\underbrace{w_1(x)\ r_2(x)\ w_2(y)\ commit_2}_{\text{looks good}}\ \ \underset{\substack{\uparrow\\ \text{disaster}}}{request\_r_1(y)}$$

37

Immediate-Update Pessimistic Control

- **Rule:**
  - *Do not grant* a request that imposes an ordering among *active* transactions (*delay* the requesting transaction)
  - *Grant* a request that does not conflict with previously granted requests of *active* transactions
- Rule can be used as each request arrives
- If a transaction's request is delayed, it is forced to wait (but the transaction is still considered active)
  - Delayed requests are reconsidered when a transaction completes (aborts or commits) since it becomes inactive

38

Immediate-Update Pessimistic Control

- **Result**: Each schedule, S, is equivalent to a serial schedule in which transactions are ordered in the order in which they commit in S (and possibly other serial schedules as well)
  - **Reason**: When a transaction commits, none of its operations conflict with those of other active transactions. Therefore it can be ordered before all active transactions.
  - **Example**: The following (non-serializable) schedule is not permitted because T_1 was active at the time $w_2(x)$ (which conflicts with $r_1(x)$) was requested

$$r_1(x)\ \ w_2(x)\ \ r_2(y)\ \ w_1(y)$$

39

Immediate-Update Pessimistic Control

$$S:\ \underbrace{op_1\ op_2 \ldots op_n}_{\text{no conflicting operations}}\ \underset{\substack{\uparrow\\ \text{first commit}}}{c_1}$$

$$S':\ \underset{\substack{\uparrow\\ \text{all operations of } T_1}}{T_1}\ \ \underbrace{op'_1\ op'_2 \ldots op'_n}_{\text{remaining operations of } S}$$

- S and S′ are conflict equivalent
  - The argument can be repeated at subsequent commits

40

Immediate-Update Pessimistic Control

- Commit order is useful since transactions might perform external actions visible to users
  - After a deposit transaction commits, you expect a subsequent transaction to see the new account balance

41

Deadlock

- **Problem**: Controls that cause transactions to wait can cause deadlocks

$$w_1(x)\ w_2(y)\ \ \substack{\text{request}\\ r_1(y)}\ \ \substack{\text{request}\\ r_2(x)}$$

- **Solution**: Abort one transaction in the cycle
  - Use wait-for graph to detect cycle when a request is delayed or
  - Assume a deadlock when a transaction waits longer than some time-out period

42

Locking Implementation of an Immediate-Update Pessimistic Control

- A transaction can read a database item if it holds a read (shared) lock on the item
- It can read *or* update the item if it holds a write (exclusive) lock
- If the transaction does not already hold the required lock, a lock request is automatically made as part of the (read or write) request

43

Locking

- Request for read lock on an item is granted if no transaction currently holds write lock on the item
  - Cannot read an item written by an active transaction
- Request for write lock granted if no transaction holds any lock on item
  - Cannot write an item read/written by an active transaction
- Transaction is delayed if request cannot be granted

Requested mode	Granted mode *read*	Granted mode *write*
read		x
write	x	x

44

Locking

- All locks held by a transaction are released when the transaction completes (commits or aborts)
  - Delayed requests are re-examined at this time

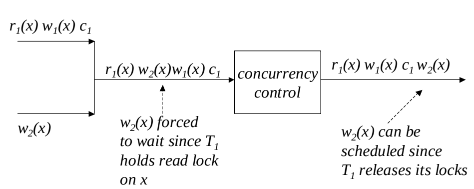
45

Locking

- **Result**: A lock is not granted if the requested access conflicts with a prior access of an active transaction; instead the transaction waits. This enforces the rule:
  - Do not grant a request that imposes an ordering among active transactions (delay the requesting transaction)
- Resulting schedules are serializable and strict

46

Locking



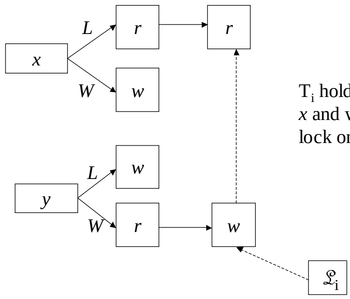
47

Locking Implementation

- Associate a *lock set, L(x),* and a *wait set, W(x),* with each active database item, *x*
  - *L(x)* contains an entry for each granted lock on *x*
  - *W(x)* contains an entry for each pending request on *x*
  - When an entry is removed from *L(x)* (due to transaction termination), promote (non-conflicting) entries from *W(x)* using some scheduling policy (*e.g.*, FCFS)
- Associate a lock list, $\mathcal{L}_i$, with each transaction, T_i.
  - $\mathcal{L}_i$ links T_i's elements in all lock and wait sets
  - Used to release locks on termination

48

Locking Implementation



T_i holds an *r* lock on *x* and waits for a *w* lock on *y*

Manual Locking

- Better performance possible if transactions are allowed to release locks before commit
  - Ex: release lock on item when finished accessing the item

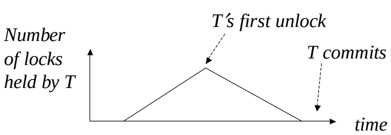
T_1: *l(x) r(x) l(y) r(y) u(x) w(y) u(y)*
T_2: *l(x) l(z) w(x) w(z) u(x) u(z)*

- However, early lock release can lead to non-serializable schedules

T_1: *l(x) r(x) u(x)* *l(y) r(y) u(y)*
T_2: *l(x) l(y) w(x) w(y)* $\underbrace{u(x)\ u(y)}_{commit}$

Two-Phase Locking

- Transaction does not release a lock until it has all the locks it will ever require.
- Transaction has a locking phase followed by an unlocking phase



- Guarantees serializability when locking is done manually

Two-Phase Locking

- **Theorem**: A concurrency control that uses two phase locking produces only serializable schedules.
  - *Proof (sketch):* Consider two transactions T_1 and T_2 in schedule S produced by a two-phase locking control and assume T_1's first unlock, t_1, precedes T_2's first unlock, t_2.
    - If they do not access common data items, then all operations commute.
    - Suppose they do. All of T_1's accesses to common items precede all of T_2's. If this were not so, T_2's first unlock must precede a lock request of T_1. Since both transactions are two-phase, this implies that T_2's first unlock precedes T_1's first unlock, contradicting the assumption. Hence, all conflicts between T_1 and T_2 are in the same direction.
  - It follows that the serialization graph is cycle-free since if there is a cycle T_1, T_2, …T_n then it must be the case that $t_1 < t_2 < \ldots < t_n < t_1$

Two-Phase Locking

- A schedule produced by a two-phase locking control is:
  - Equivalent to a serial schedule in which transactions are ordered by the time of their first unlock operation
  - Not necessarily recoverable (dirty reads and writes are possible)

T1: *l(x) r(x) l(y) w(y) u(y)* *abort*
T2: *l(y) r(y) l(z) w(z) u(z) u(y) commit*

Two-Phase Locking

- A two-phase locking control that holds write locks until commit produces strict, serializable schedules
- A strict two-phase locking control holds *all* locks until commit and produces strict serializable schedules
  - This is automatic locking
  - Equivalent to a serial schedule in which transactions are ordered by their commit time
- "Strict" is used in two different ways: a control that releases read locks early guarantees *strictness*, but is not *strict* two-phase locking control

Lock Granularity

- Data item: variable, record, row, table, file
- When an item is accessed, the DBMS locks an entity that *contains* the item. The size of that entity determines the *granularity* of the lock
  - Coarse granularity (large entities locked)
    - **Advantage**: If transactions tend to access multiple items in the same entity, fewer lock requests need to be processed and less lock storage space required
    - **Disadvantage:** Concurrency is reduced since some items are unnecessarily locked
  - Fine granularity (small entities locked)
    - Advantages and disadvantages are reversed

55

Lock Granularity

- Table locking (*coarse*)
  - Lock entire table when a row is accessed.
- Row (tuple) locking (*fine*)
  - Lock only the row that is accessed.
- Page locking (compromise)
  - When a row is accessed, lock the containing page

56

Objects and Semantic Commutativity

- Read/write operations have little associated semantics and hence little associated commutativity.
  - Among operations on the same item, only reads commute.
- Abstract operations (for example operations on objects) have more semantics, allowing
  - More commutativity to be recognized
  - More concurrency to be achieved

57

Abstract Operations and Commutativity

- A concurrency control that deals with operations at an abstract level can recognize more commutativity and achieve more concurrency
- **Example**: operations *deposit(acct,n)*, *withdraw(acct,n)* on an account object (where *n* is the dollar amount)

	Granted Mode	
Requested Mode	*deposit( )*	*withdraw( )*
deposit( )		X
withdraw( )	X	X

58

A Concurrency Control Based on Abstract Operations

- Concurrency control grants *deposit* and *withdraw* locks based on this table
- If one transaction has a *deposit* lock on an account object, another transaction can also obtain a *deposit* lock on the object
- Would not be possible if control viewed *deposit* as a *read* followed by a *write* and attempted to get *read* and *write* locks

59

A Concurrency Control Based on Abstract Operations

- Since T_1 and T_2 can both hold a *deposit* lock on the same *account* object their deposit operations do not delay each other
  - As a result, the schedule can contain:
    … $deposit_1(acct,n)$ … $deposit_2(acct,m)$ … $commit_1$
    or
    … $deposit_2(acct,m)$ … $deposit_1(acct,n)$ … $commit_2$
  - But the two deposit operations must be isolated from each other. Assuming *bal* is the account balance, the schedule
    $r_1(bal)\ r_2(bal)\ w_1(bal)\ w_2(bal)$
    cannot be allowed

60

Partial vs. Total Operations

- *deposit( ), withdraw( )* are *total operations*: they are defined in all database states.
- *withdraw( )* has two possible outcomes: *OK, NO*
- **Partial operations** are operations that are not defined in all database states
- *withdraw( )* can be decomposed into two partial operations, which cover all database states:
    - *withdrawOK( )*
    - *withdrawNO( )*

61

Partial Operations

- Example: account object
    - *deposit( )*: defined in all initial states (total)
    - *withdrawOK(acct,x)*: defined in all states in which $bal \geq x$ (partial)
    - *withdrawNO(acct,x)*: defined in all states in which $bal < x$ (partial)
- When a transaction submits *withdraw( ),* control checks balance and converts to either *withdrawOK( )* or *withdrawNO( )* and acquires appropriate lock

62

Partial Operations

- Partial operations allow even more semantics to be introduced
- Insight: while *deposit( )* does not commute with *withdraw( )*, it does (backward) commute with *withdrawOK( )*

withdrawOK(a,n) deposit(a,m) $\rightarrow$ *deposit(a,m) withdrawOK(a.n)*

63

Backward Commutativity

- *p backward commutes* through *q* iff in all states in which the sequence *q, p* is defined, the sequence *p, q* is defined and
    - *p* and *q* return the same information in both and
    - The database is left in the same final state
- Example:
    - *deposit(a,m)* backward commutes through *withdrawOK(a,n)*
        - In all database states in which *withdrawOK(a,n), deposit(a,m)* is defined, *deposit(a,m), withdrawOK(a,n)* is also defined.
    - *withdrawOK(a,n)* does not backward commute through *deposit(a,m)*
    - Backward commute is not symmetric

64

A Concurrency Control Based on Partial Abstract Operations

Requested Mode	Granted Mode: *deposit( )*	*withdrawOK( )*	*withdrawNO( )*
deposit( )			X
withdrawOK( )	X		
withdrawNO( )		X	

- Control grants *deposit*, *withdrawOK*, and *withdrawNO* locks
    - Conflict relation is
        - not symmetric
        - based on backward commutativity

65

A Concurrency Control Based on Partial Abstract Operations

- **Advantage**: Increased concurrency and hence increased transaction throughput
- **Disadvantage**: Concurrency control has to access the database to determine the return value (hence the operation requested) before consulting table
- Hence (with an immediate update system) if T writes *x* and later aborts, physical restoration can be used.

66

Atomicity and Abstract Operations

- A write operation (the only conventional operation that modifies items) conflicts with *all* other operations on the same data
- **Physical restoration** (restore original value) does not work with abstract operations since two operations that modify a data item might commute
  - How do you handle the schedule: $\ldots p_1(x)\ q_2(x)\ abort_1 \ldots$ if both operations modify x?
- **Logical restoration** (with compensating operations) must be used
  - *e.g., increment(x)* compensates for *decrement(x)*

67

A Closer Look at Compensation

- We have discussed compensation before, but now we want to use it in combination with locking to guarantee serializability and atomicity
- We must define compensation more carefully

68

Requirements for an Operation to Have a Compensating Operation

- For an operation to have a compensating operation, it must be one-to-one
  - For each input there is a unique output
  - The parameters of the compensating operation are the same as the parameters of the operation being compensated
    - *increment(x)* compensate *decrement(x)*

69

Logical Restoration (Compensation)

- Consider schedule: $p_1(x)\ q_2(x)\ abort_1$
- $q_2(x)$ must (backward) commute through $p_1(x)$, since the concurrency control scheduled the operation
- This is equivalent to $q_2(x)\ p_1(x)\ abort_1$
- Then $abort_1$ can be implemented with a compensating operation: $q_2(x)\ p_1(x)\ p_1^{-1}(x)$
  - This is equivalent to $q_2(x)$
- Thus $p_1(x)\ q_2(x)\ p_1^{-1}(x)$ is equivalent to $q_2(x)$

70

Logical Restoration (Compensation)

- Example:

$p_1(x) = decrement(x)$
$p_1^{-1}(x) = increment(x)$

compensating operation

$$decrement_1(x)\ increment_2(x)\ increment_1(x) \equiv increment_2(x)$$

71

Undo Operations

- Not all operations have compensating operations
  - For example, *reset(x)*, which sets *x* to *0*, is not one-to-one and has no compensating operation
  - It does have an undo operation, *set(x, X)*, which sets the value of *x* to what it was right before *reset(x)* was executed.

72

The Previous Approach Does Not Work

$reset_1(x)\ reset_2(x)\ set_1(x, X_1)$

- Since the two *reset*s commute, we can rewrite the schedule as

  $reset_2(x)\ reset_1(x)\ set_1(x, X_1)$

- But this schedule does not undo the result of $\text{reset}_1(\text{x})$, because the value when $reset_1(x)$ starts is different in the second schedule

What to Do with Undo Operations

- One approach is to require that the operation get an exclusive lock, so that no other operation can come between an operation and its undo operation

Another Approach

- Suppose p^{undo} commutes with *q*. Then

  $p\ q\ p^{undo} \equiv p\ p^{undo}\ q$

- Now *p* has the same initial value in both schedules, and thus the undo operation works correctly.

Another Approach

- Theorem
  - Serializability and recoverability is guaranteed if the condition under which an operation *q* does not conflict with a previously granted operation *p* is
    - *q* backward commutes through *p*, and
    - Either *p* has a compensating operation, or when a *p* lock is held, p^{undo} backward commutes through *q*

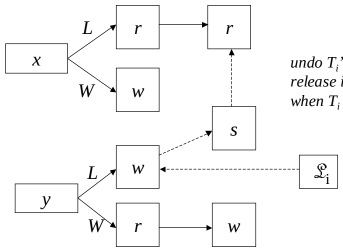
Still Another Approach

- Sometimes we can decompose an operation that does not have a compensating operation into two partial operations, each of which does have a compensating operation
  - *withdraw(x)* does not have a compensating operation
    - Depending on the initial value of the account, it might perform the withdrawal and decrement that value by x or it might just return no
    - It has an undo operation, *conditionalDeposit(x,y)*
  - The two partial operations, *withdrawOK(x)* and *withdrawNO(x)* are one-to-one and hence do have compensating operations.

Locking Implementation of Savepoints

- When T_i creates a savepoint, *s*, insert a marker for *s* in T_i's lock list, $\mathfrak{L}_i$, that separates lock entries acquired before creation from those acquired after creation
- When T_i rolls back to *s*, release all locks preceding marker for *s* in $\mathfrak{L}_i$ (in addition to undoing all updates made since savepoint creation)

Locking Implementation



undo T_i's update to y and release its write lock when T_i rolls back to s

79

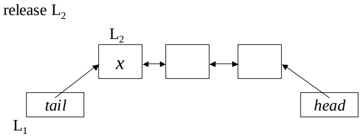
Locking Implementation of...

- *Chaining*: nothing new
- *Recoverable queue*: Since queue is implemented by a separate server (different from DBMS), the locking discipline need not be two-phase; discipline can be designed to suit the semantics of (the abstract operations) *enqueue* and *dequeue*
  - Lock on head (tail) pointer released when dequeue (enqueue) operations complete
    - Hence not strict or isolated
  - Lock on entry that is enqueued or dequeued held to commit time

80

Recoverable Queue

begin transaction
….
enqueue(x) ← acquire L_1, L_2
← release L_1
….
commit ← release L_2



81

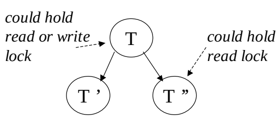
Locking Implementation of Nested Transactions

- Nested transactions satisfy:
  - Nested transactions are isolated with respect to one another
  - A parent does not execute concurrently with its children
  - A child (and its descendants) is isolated from its siblings (and their descendants)

82

Locking Implementation of Nested Transactions

- A request to read *x* by subtransaction T′ of nested transaction T is granted if:
  - No other nested transaction holds a write lock on *x*
  - All other subtransactions of T holding write locks on *x* are ancestors of T′ (hence are not executing)



83

Intuition

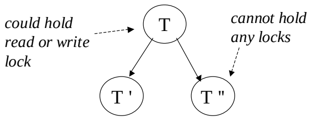
- A request to read *x* by subtransaction T' of nested transaction T is granted even though an ancestor of T' holds a write lock on *x*

without nesting		with nesting
T: begin transaction		*T: begin transaction*
…		…
w(x)		*w(x)*
		T': begin transaction
…		…
r(x)	*r(x) does not conflict with w(x)*	*r(x)*
…		…
		commit
commit		*commit*

84

Locking Implementation of Nested Transactions

- A request to write *x* by subtransaction T ' of nested transaction T is granted if:
  - No other nested transaction holds a read or write lock on *x*
  - All other subtransactions of T holding read or write locks on *x* are ancestors of T ' (and hence are not executing)



85

Locking Implementation of Nested Transactions

- All locks obtained by T' are held until it completes
  - If it aborts, all locks are discarded
  - If it commits, any locks it holds that are not held by its parent are inherited by its parent
- When top-level transaction (and hence entire nested transaction) commits, all locks are discarded

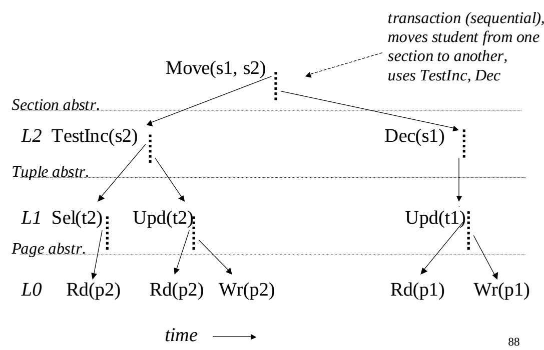
86

Locking Implementation of Multilevel Transactions

- Generalization of strict two-phase locking concurrency control
  - Uses semantics of operations at each level to determine commutativity
  - Uses different concurrency control at each level

87

Example - Switch Sections



88

Multilevel Transactions

- **Example**:
  - *Move(s1,s2)* produces *TestInc(s2), Dec(s1)*
  - *$Move_1$(s1,s2), $Move_2$(s1, s3)* might produce
    $TestInc_1$(s2), $TestInc_2$(s3), Dec_2(s1), Dec_1(s1)
  - Since two *Dec* operations on the same object commute (they do not impose an ordering among transactions), this schedule is equivalent to
    $TestInc_1$(s2), Dec_1(s1), $TestInc_2$(s3), Dec_2(s1)
    and hence could be allowed by a multilevel control, but ...

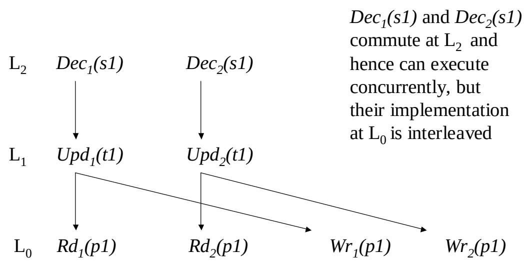
89

Multilevel Control

- **Problem**: A control assumes that the execution of operations it schedules is isolated: If op_1 and op_2 do not conflict, they can be executed concurrently and the result will be either op_1, op_2 or op_2, op_1
  - Not true in a multilevel control where an operation is implemented as a program at the next lower level that might invoke multiple operations at the level below. Hence, concurrent operations at one level might not be totally ordered at the next

90

Multilevel Transactions



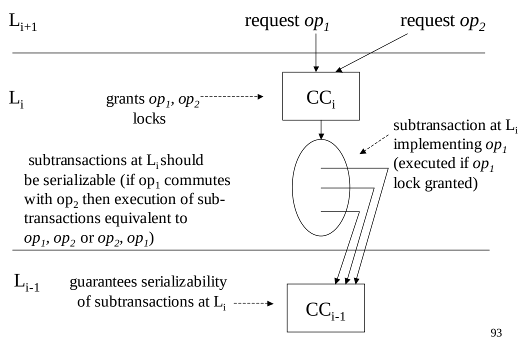
91

Guaranteeing Operation Isolation

- **Solution**: Use a concurrency control at each level
  - L_i receives a request from L_{i+1} to execute *op*
  - Concurrency control at L_i, CC_i, schedules *op* to be executed; it assumes execution is isolated
  - *op* is implemented as a program, *P*, in L_i
  - *P* is executed as a subtransaction so that it is serializable with respect to other operations scheduled by CC_i
  - Serializability guaranteed by CC_{i-1}

92

Guaranteeing Operation Isolation



93

A Multilevel Concurrency Control for the Example

- The control at L_2 uses *TestInc* and *Dec* locks
- The control at L_1 uses *Sel* and *Upd* locks
- The control at L_0 uses *Rd* and *Wr* locks

94

Timestamp-Ordered Concurrency Control

- Each transaction given a (unique) timestamp (current clock value) when initiated
- Uses the immediate update model
- Guarantees equivalent serial order based on timestamps (initiation order)
  - Control is *static* (as opposed to *dynamic*, in which the equivalent serial order is determined as the schedule progresses)

95

Timestamp-Ordered Concurrency Control

- Associated with each database item, *x*, are two timestamps:
  - *wt(x)*, the largest timestamp of any transaction that has written *x*,
  - *rt(x)*, the largest timestamp of any transaction that has read *x*,
  - and an indication of whether or not the last write to that item is from a committed transaction

96

Timestamp-Ordered Concurrency Control

- If T requests to read *x*:
  - **R1**: if $TS(T) < wt(x)$, then T is too old; abort T
  - **R2**: if $TS(T) > wt(x)$, then
    - if the value of *x* is committed, grant T's read and if $TS(T) > rt(x)$ assign $TS(T)$ to $rt(x)$
    - if the value of *x* is not committed, T waits (to avoid a dirty read)

Timestamp-Ordered Concurrency Control

- If T requests to write *x* :
  - **W1**: If $TS(T) < rt(x)$, then T is too old; abort T
  - **W2**: If $rt(x) < TS(T) < wt(x)$, then no transaction that read *x* should have read the value T is attempting to write and no transaction will read that value (See R1)
    - If *x* is committed, grant the request but do not do the write
      - This is called the Thomas Write Rule
    - If *x* is not committed, T waits to see if newer value will commit. If it does, discard T's write, else perform it
  - **W3**: If $wt(x), rt(x) < TS(T)$, then if *x* is committed, grant the request and assign $TS(T)$ to $wt(x)$, else T waits

Example

- Assume $TS(T_1) < TS(T_2)$, at t_0 *x* and *y* are committed, and *x*'s and *y*'s read and write timestamps are less than $TS(T_1)$

T_1 :		*r(y)*				*w(x)*	*commit*
T_2:			*w(y)*	*w(x)*	*commit*		
	t_0	t_1	t_2	t_3		t_4	

t_1: (R2) $TS(T_1) > wt(y)$; assign $TS(T_1)$ to $rt(y)$
t_2: (W3) $TS(T2) > rt(y), wt(y)$; assign $TS(T_2)$ to $wt(y)$
t_3: (W3) $TS(T2) > rt(x), wt(x)$; assign $TS(T_2)$ to $wt(x)$
t_4: (W2) $rt(x) < TS(T1) < wt(x)$; grant request, but do not do the write

Timestamp-Ordered Concurrency Control

- Control accepts schedules that are *not conflict equivalent* to any serial schedule and would not be accepted by a two-phase locking control
  - Previous example equivalent to T_1, T_2
- But additional space required in database for storing timestamps and time for managing timestamps
  - Reading a data item now implies writing back a new value of its timestamp

Optimistic Algorithms

- Do task under simplifying (optimistic) assumption
  - **Example**: Operations rarely conflict
- Check afterwards if assumption was true.
  - **Example**: Did a conflict occur?
- Redo task if assumption was false
  - **Example**: If a conflict has occurred rollback, else commit
- Performance benefit if assumption is generally true and check can be done efficiently

Optimistic Concurrency Control

- *Under the optimistic assumption that conflicts do not occur*, read and write requests are always granted (no locking, no overhead!)
- *Since conflicts might occur*:
  - Database might be corrupted if writes were immediate, hence a deferred-update model is used
  - Transaction has to be "validated" when it completes
    - If a conflict has occurred abort (but no rollback is necessary) and redo transaction
- Approach contrasts with pessimistic control which assumes conflicts are likely, takes preventative measures (locking), and does no validation

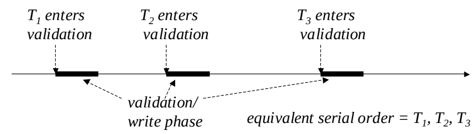
Optimistic Concurrency Control

- Transaction has three phases:
  - Begin transaction
    - *Read Phase* - transaction executes: reads from database, writes to intentions list (deferred-update, no changes to database)
  - Request commit
    - *Validation Phase* - check whether conflicts occurred during read phase; if yes abort (discard intentions list)
  - Commit
    - *Write Phase* - write intentions list to database (deferred update) if validation successful
  - For simplicity, we assume here that validation and write phases form a single critical section (only one transaction is in its validation/write phase at a time)

103

Optimistic Concurrency Control

- Guarantees an equivalent serial schedule in which the order of transactions is the order in which they enter validation (dynamic)
- For simplicity, we will assume that validation and write phases form a single critical section (only one transaction is in its validation/write phase at a time)



104

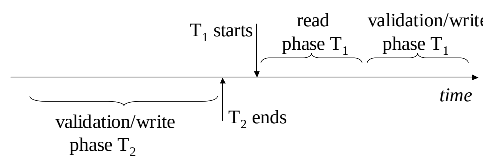
Validation

- When T_1 enters validation, a check is made to see if T_1 conflicted with any transaction, T_2, that entered validation at an earlier time
- Check uses two sets constructed during read phase:
  - $R(T_1)$: identity of all database items T_1 read
  - $W(T_1)$: identity of all database items T_1 wrote

105

Validation

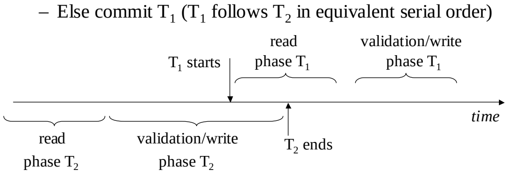
- *Case 1*: T_1's read phase started after T_2 finished its validation/write phase
  - T_1 follows T_2 in all conflicts, hence commit T_1 (T_1 follows T_2 in equivalent serial order)



106

Validation

- *Case 2*: T_1's read phase overlaps T_2's validation/write phase
  - If $WS(T_2) \cap RS(T_1) \neq \Phi$, then abort T_1
    - A read of T_1 *might* have preceded a write of T_2 – a *possible* violation of equivalent serial order
  - Else commit T_1 (T_1 follows T_2 in equivalent serial order)



107

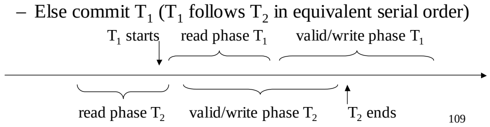
Validation

- *Case 3*: T_1's validation/write phase overlaps T_2's validation/write phase
  - Cannot happen since we have assumed that validation/write phases do not overlap
- Hence, all possible overlaps of T_1 and T_2 have been considered

108

Validation

- A more practical optimistic control allows case 3 and avoids the bottleneck implied by only allowing only one transaction at a time in the validation/write phase.
- *Case 3*: T_1's validation/write phase overlaps T_2's validation/write phase
  - If $WS(T_2) \cap (WS(T_1) \cup RS(T_1)) \neq \Phi$, then abort T_1
    - A read *or* write of T_1 *might* have preceded a write of T_2 – a violation of equivalent serial order
  - Else commit T_1 (T_1 follows T_2 in equivalent serial order)



Optimistic Concurrency Control

- No locking (and hence no waiting) means deadlocks are not possible
- Rollback is a problem if optimistic assumption is not valid: work of entire transaction is lost
  - With two-phase locking, rollback occurs only with deadlock
  - With timestamp-ordered control, rollback is detected before transaction completes