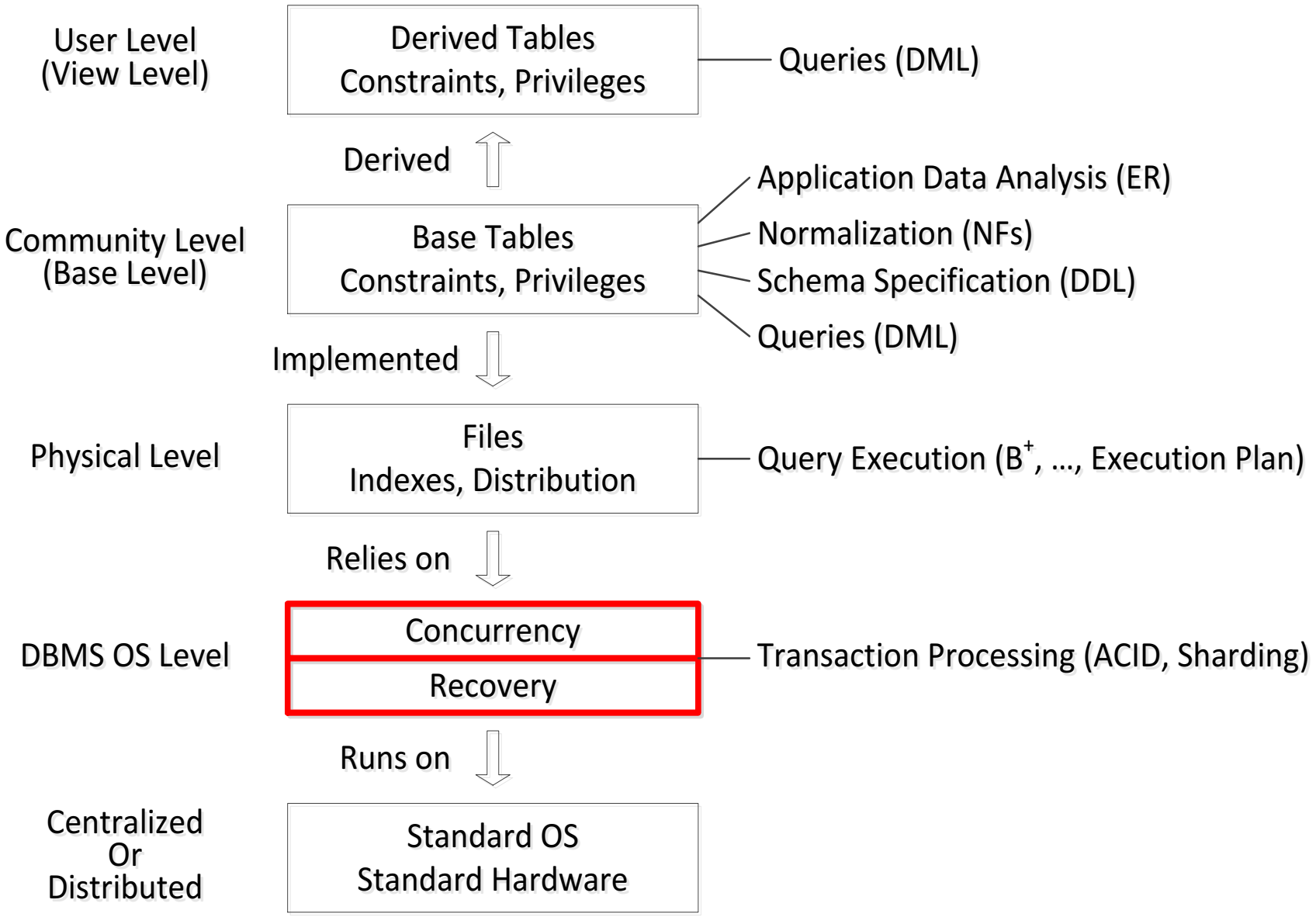
Unit 10
Transaction Processing: Concurrency

© 2016 Zvi M. Kedem

Concurrency in Context



© 2016 Zvi M. Kedem

Transactions

- Transaction is an execution of a user's program
- In the ***cleanest and most important*** model a transaction is supposed to satisfy the ***ACID*** conditions
  - ***Atomic***
  - ***Consistent***
  - ***Isolated***
  - ***Durable***
- Some transactions may not satisfy all of these conditions in order to improve performance, as we will see later

© 2016 Zvi M. Kedem

Recovery and Concurrency Management

- The job of these recovery/concurrency modules of the database operating system is to assure the ***ACID*** properties, and handle other related issues
- Recovery does ***ACD***, but can use help from Concurrency, though strictly Recovery is needed even if there is no Concurrency
- Concurrency does ***I*** while possibly (and in our description, definitely) supporting ***ACD***

© 2016 Zvi M. Kedem

The Concurrency Problem

- Here we focus on ***Isolation*** in the presence of concurrently executing transactions
- Each transaction should run as if there were no other transactions in the system
- Our execution platform: a single centralized system, with concurrent execution of transactions
- Distributed databases more difficult to handle, as we will see briefly later in the class

© 2016 Zvi M. Kedem

A Toy Example

- A database consisting of two items: x, y
- Initially x = 0 and y = 0
- The only criterion for correctness is the single integrity constraint:

$$x = y$$

- Consider two simple transactions, T1 and T2
  - T1: x := x + 1; y := y + 1
  - T2: read and print x; read and print y
- Both transactions in isolation are correct: they preserve the consistency of the database

© 2016 Zvi M. Kedem

An Execution History

- An execution history

T1	T2
x := x + 1	
	read and print x
	read and print y
y := y + 1	

- T2 read x after it was incremented
- T2 read y before it was incremented
- Note that T2 thinks x = 1 and y = 0
- T2 reports that the database is inconsistent and people who see the report will be upset

© 2016 Zvi M. Kedem

A Toy Example

- A database consisting of two items: x, y
- Initially x = 0 and y = 0
- The only criterion for correctness is the single integrity constraint:

$$x = y$$

- Consider two simple transactions, T1 and T2
  - T1: x := x + 1; y := y + 1
  - T2: x := 2x; y := 2y
- Both transactions are correct: they preserve the consistency of the database

© 2016 Zvi M. Kedem

An Execution History

- An execution history

T1	T2
x := x + 1	
	x := 2x
	y := 2y
y := y + 1	

- After the execution:
  - $x_{new} = 2(x_{old} + 1) = 2x_{old} + 2 = 2$
  - $y_{new} = 2y_{old} + 1 = 1$

- Therefore, we now have: $x \neq y$!

- Note, the history was not recoverable, so could not be permitted in any case, but we will not focus on this now
- Ultimately we will have strict histories (later in the unit)

© 2016 Zvi M. Kedem

The Problem

- The problem was: the transactions ***were not Isolated***:
  - T2 read the old value of y and the new value of x
  - T1 read the old value of x and the new value of y
- But sometimes this is not a problem, if the operations performed are commutative
- So assume now, that T1 multiplied x and y by 4 and T2 multiplies x and y by 2

© 2016 Zvi M. Kedem

An Execution History

- An execution history

T1	T2
x := 4x	x := 2x
y := 4y	y := 2y

- After the execution:
  - $x_{new} = 2(4x_{old}) = 8x_{old} = 0$
  - $y_{new} = 4(2y_{old}) = 8y_{old} = 0$

- Therefore, if we had x = y, we now have x = y now also

© 2016 Zvi M. Kedem

Abstraction

- In general, DB OS cannot understand what the transaction does and which operations are commutative
- ***The DB OS can only see patterns of reads and writes***
  - ***Who read/wrote what item and when, and what was the value read/written***
- Abstracting out, we get for our example:

T1	T2
READ x 0	
WRITE x 1	
	READ x 1
	WRITE x 2
	READ y 0
	WRITE y 0
READ y 0	
WRITE y 1	

© 2016 Zvi M. Kedem

Abstraction

- In general, DB OS cannot understand what to do based on knowing the values read/written
- ***The DB OS can understand only patterns of reads and writes***
  - ***Who read/wrote what item and when***
- Abstracting out, we get for our example:

T1	T2
READ x	
WRITE x	
	READ x
	WRITE x
	READ y
	WRITE y
READ y	
WRITE y	

© 2016 Zvi M. Kedem

Abstraction

- ***Since there are possible actions that result in this pattern of accesses that produce incorrect executions, we must prevent such patterns, even though sometimes they may produce correct executions***

- Here it is relatively easy to see what went wrong
- We can say:
  - T1 wrote something and then T2 read it
  - T2 wrote something and then T1 read
- We will want in general to avoid such “circular” dependencies, but they may be more subtle that in this example
- We need a formal and precise statement
- And we also want strict histories to help recovery

© 2016 Zvi M. Kedem

Concurrency And Correctness?

- In general, it may be very difficult to define under what conditions a concurrent system is correct
- So we will say: ***no errors are introduced because of concurrent execution that would not have occurred in a serial execution***
- Because of the difficulty of figuring out what is correct and what is not, concurrency control mechanisms are to some extent "overkill"

  They use mechanisms that may sometimes be ***too strong***, but will always be ***strong enough (unless we weaken them on purpose to speed up processing)***

© 2016 Zvi M. Kedem

Formal Definition Of History In Our Context

- A ***history*** (or **schedule**) is a trace of behavior of a set of transactions, listing the reads and the writes in order of execution
- In our example

T1: READ x
T1: WRITE x

T2: READ x
T2: WRITE x
T2: READ y
T2: WRITE y

T1: READ y
T1: WRITE y

© 2016 Zvi M. Kedem

An Important Restriction For Now

- We will assume that no items are added during a run of a transaction
  - This is so called “phantoms” problem
- This is not realistic but can be easily fixed later
- To state our assumption more formally

  The database consists of a fixed set of items

  Transactions may read and write (modify) them

© 2016 Zvi M. Kedem

Serial Histories

- A history is ***serial*** if it describes a serial execution:
  Transactions follow each other: no concurrency
- Example of a serial execution

```
T1:  READ x
T1:  WRITE x
T1:  READ y
T1:  WRITE y
                    T2:  READ x
                    T2:  WRITE x
                    T2:  READ y
                    T2:  WRITE y
```

- ***A concurrent execution that happens to be serial is a correct concurrent execution***
  - By assumption, each transaction is correct when run by itself

© 2016 Zvi M. Kedem

Serializable Histories

- We are given:
  - A database and its initial state
  - A set of transactions
  - A history H of these transactions on this database in this initial state
- H is ***serializable*** if it is equivalent to some serial history H' of this set of transactions on this database in this initial state
- We need to define “equivalent” formally

© 2016 Zvi M. Kedem

Equivalent Histories

- Let us assume, that the initial state of the database (before current execution starts) was produced by some transaction T_0
- Then some transactions T_1, T_2, …, T_n executed (possibly concurrently fully or partially) and the execution ended producing some final state of the database
- Let us now consider two histories, H and H'
- ***We will say that these histories are equivalent if the behavior is the same in some formal sense in both histories***
  - ***Transactions read and write the same values***
  - ***The final state of the database is the same***
- We will discuss next a more "operational" definition of equivalency, which does not rely on values (which are generally not known) but on temporal order of certain actions

© 2016 Zvi M. Kedem

Operational Definition Of Equivalent Histories

- In both histories, if a transaction T_j read some item x, it read the value that was written by the same T_i (T_i could be T_0)
  - This implies, that T_j read the same values in both histories, and as we assume that the transactions are deterministic, T_j produced the same values when it wrote
- In both histories, each item x is last written by the same transaction
  - This implies that the final state of the database is the same under both histories

© 2016 Zvi M. Kedem

Serializable Histories

- Assume
  - H and H' equivalent
  - H' serial
- H' was correct, because it was a serial execution
- Therefore H was correct, because it was equivalent to H'
- Therefore:
  Each serializable history describes a correct execution!
- How to determine if a history is serializable?
- We will do something weaker
  - Whenever we say that a history is serializable, it will indeed will be serializable
  - But sometimes when it is serializable but we will not be able to recognize this
- So we may be overly cautious but will never accept incorrect executions

© 2016 Zvi M. Kedem

Serializable Histories

- So we really have algorithms that partition histories into two classes
  - Serializable
  - Perhaps not serializable

- We will focus on ***conflict serializability***
- We will partition histories into two classes:
  - Conflict serializable (guaranteed to be serializable)
  - Not conflict serializable (we do not know whether they are serializable or not)

© 2016 Zvi M. Kedem

Conflict Serializable Histories

- The idea is to figure out something along the following lines:

  If a transaction accessed some item, who else could have accessed this item in a way that implies a potential temporal constraint on any equivalent serial schedule
- Tests are local and therefore will be
  - Efficient
  - Non-comprehensive: more temporal constraints will be imposed than needed in general; therefore serializable histories may not be conflict-serializable and we will act as if there were not serializable and not permit them

- We proceed to examine four histories of two transactions each and look only at one operation from each of the two transactions: we ignore everything else
- Here and later we may write “W” for “WRITE” and “R” for “READ”

© 2016 Zvi M. Kedem

Conflicting Operations (No Implication This Is A Bad Thing)

.
T1: W x
.
T2: R x
.

- It is possible that these are the only operations (we do not examine others)
- Based only on the above, it is not possible that the following serial history is equivalent to our history

  T2 (all instructions of this transaction, whatever they are)
  T1 (all instructions of this transaction, whatever they are)
- Because:
  - In the original history T2 read x as produced by T1
  - In the above serial history T2 could not have done this
- The only hope for equivalent serial history: T1 before T2
- But this may not work either because of other actions

© 2016 Zvi M. Kedem

Conflicting Operations (No Implication This Is A Bad Thing)

.
T1: R x
.
T2: W x
.

- It is possible that these are the only operations (we do not examine others)
- Based only on the above, it is not possible that the following serial history is equivalent to our history

  T2 (all instructions of this transaction, whatever they are)
  T1 (all instructions of this transaction, whatever they are)
- Because:
  - In the original history T1 read x not produced by T2
  - In the above serial history T1 had to read x produced by T2
- The only hope for equivalent serial history: T1 before T2
- But this may not work either because of other actions

© 2016 Zvi M. Kedem

Conflicting Operations (No Implication This Is A Bad Thing)

.
T1: W x
.
T2: W x
.

- It is possible that these are the only operations (we do not examine others)
- Based only on the above, it is not possible that the following serial history is equivalent to our history

  T2 (all instructions of this transaction, whatever they are)
  T1 (all instructions of this transaction, whatever they are)
- Because:
  - In the original history x was produced for the future by T2
  - In the above serial history T2 could not have done this
- The only hope for equivalent serial history: T1 before T2
- But this may not work either because of other actions

© 2016 Zvi M. Kedem

Conflicting Operations (No Implication This Is A Bad Thing)

.
T1: R x
.
T2: R x
.

- It is possible that these are the only operations (we do not examine others)
- Based only on the above, it is possible that the following serial history is equivalent to our history

  All of T1 then all of T
- Same for

  All of T2 then all of T1
- Because mutual order of reads does not matter
- But neither may work because of other actions

© 2016 Zvi M. Kedem

Conflicting Operations (No Implication This Is A Bad Thing)

- Why did we consider Read as conflicting with Write?
- After a transaction that read did not do anything, so why does it matter what it read
  - For similar reason that we had while discussing recoverable histories
- Consider the following case:
  - Initially: x = 0 and y = 0
  - T1 is: x := 1
  - T2 is: **if** x = 0 **then** y := 1
  - At the end: x = 1 and y = 0
- Consider the following case with reversed order:
  - Initially x = 0 and y = 0
  - T2 is: **if** x = 0 **then** y := 1
  - T1 is: x := 1
  - At the end: x = 1 and y = 1
- So we need to pay attention to this as the “reading transaction,” could have done something else

© 2016 Zvi M. Kedem

Conflicting Operations (No Implication This Is A Bad Thing)

- We will define when two operations (READ / WRITE) ***conflict*** (does not necessarily mean a bad thing happened)
  - Intuitively: their relative order may matter
- If a history contains

```
. . .
Ti:  OP' x
. . .
Tj:  OP" x
. . .
```

- Ti and Tj conflict if and only if:
  - i ≠ j (two transactions)
  - x (**same** variable/item accessed by both transactions)
  - At least one of the OP' and OP" is a WRITE

© 2016 Zvi M. Kedem

Conflict Graphs

- ***Conflict graph*** is used to decide whether a history
  - Is conflict serializable, or
  - Is not conflict serializable
- The vertices of the conflict graph will be the transactions
- We draw an arc from T' to T" if the two transactions conflict and T' made the relevant access first
- Sometimes we may label the arc by the item that was accessed (just for easier reading of the graph, it is not needed)

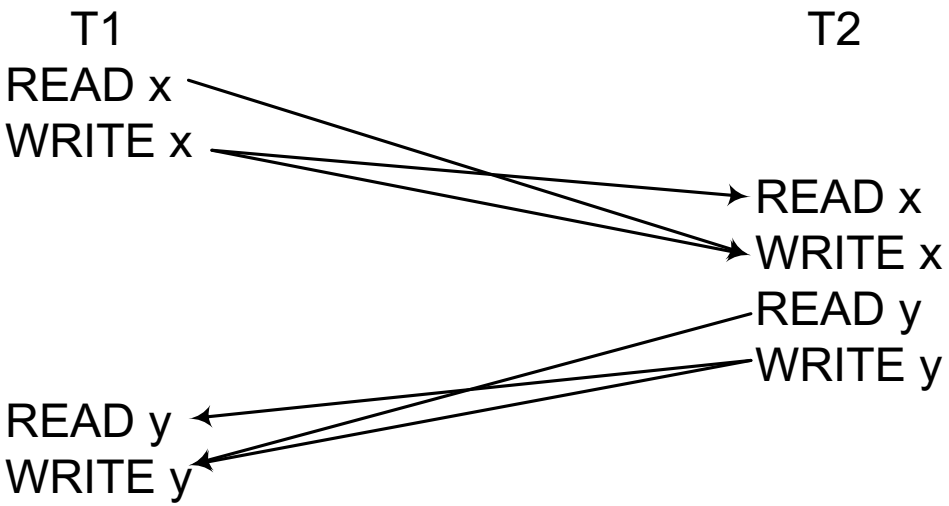
© 2016 Zvi M. Kedem

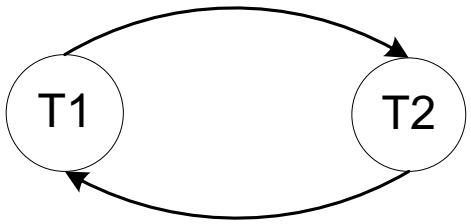
Our Example

- Back to our example:

T1: READ x
T1: WRITE x
T2: READ x
T2: WRITE x
T2: READ y
T2: WRITE y
T1: READ y
T1: WRITE y

- Note: there is a cycle





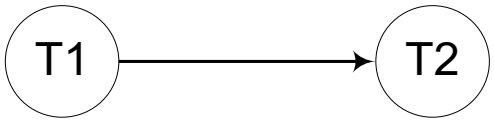
© 2016 Zvi M. Kedem

Serial And Conflict-Serializable Histories

- The conflict graph for a serial history does not have cycles (is acyclic)
- All arcs point from an “older” to a “younger” transaction
- Serial history

T1: READ x
T1: WRITE x
T1: READ y
T1: WRITE y

T2: READ x
T2: WRITE x
T2: READ y
T2: WRITE y



© 2016 Zvi M. Kedem

Serial And Serializable Histories

- Another history:

T1: READ x

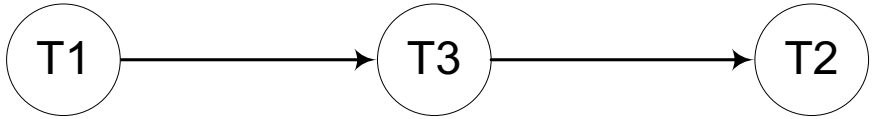
T3: READ z
T3: WRITE z

T2: READ z
T2: WRITE z

T1: WRITE x

T3: READ x
T3: WRITE x

- This conflict graph does not have a cycle, and the history is serializable



- Equivalent serial history:

T1: READ x
T1: WRITE x

T3: READ z
T3: WRITE z
T3: READ x
T3: WRITE x

T2: READ z
T2: WRITE z

© 2016 Zvi M. Kedem

Conflict Graphs And Conflict Serializability

- ***Theorem: If the conflict graph is acyclic (has no cycles), then the history is serializable***
- The proof is simple: The graph can be ***topologically sorted***:
  - An order can be given to transactions, so all the arcs go from early (old) to late (young) transactions
- Topological sorting of an acyclic graph on N vertices

  Create N levels.

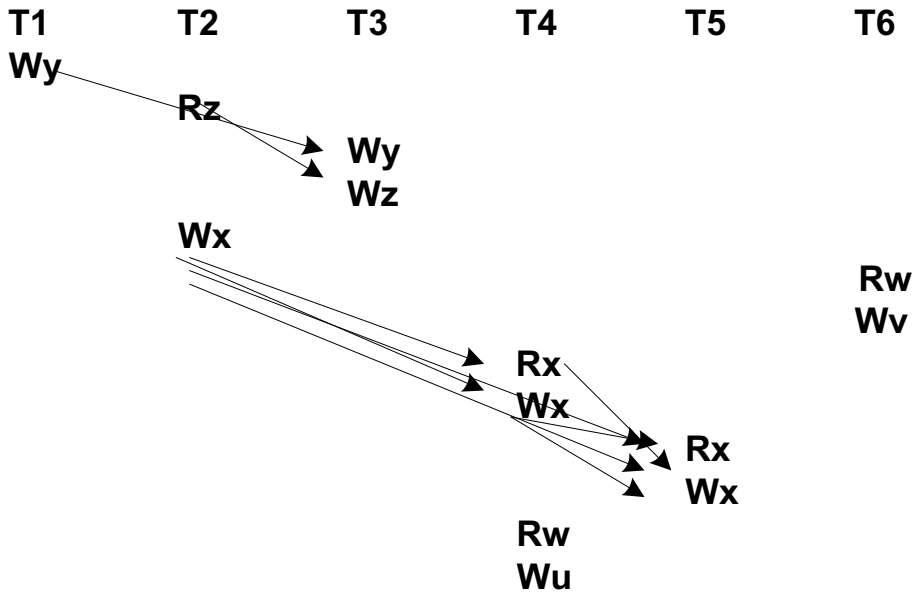
  Starting from the top level, for each level do the following:

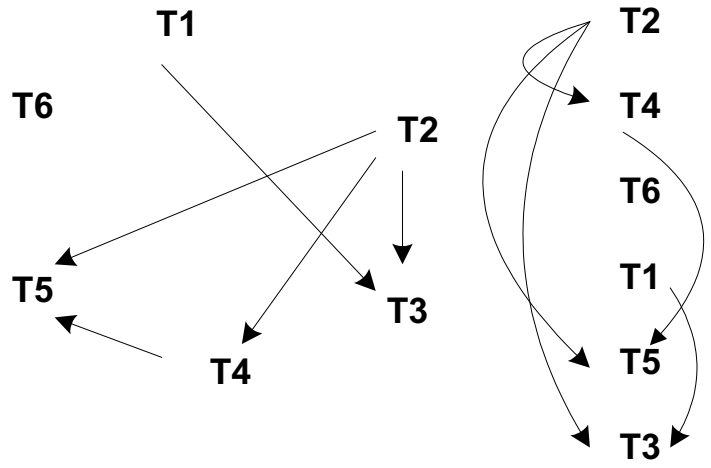
  - Pick a vertex that has no incoming edges (there is always such a vertex as the graph is acyclic)
  - Place it on the level
  - Remove it, and the edges outgoing from it from the graph

  Redraw the graph, keeping the vertices on the levels

© 2016 Zvi M. Kedem

Example Of A Serializable History

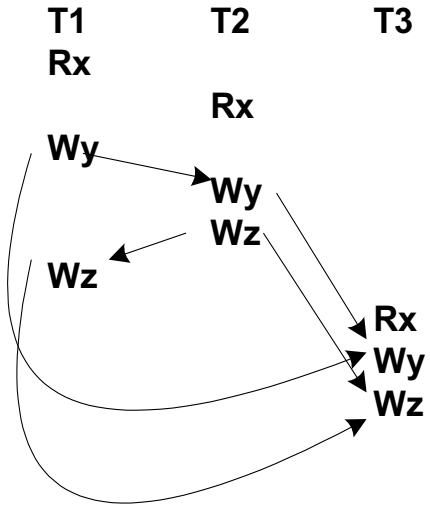




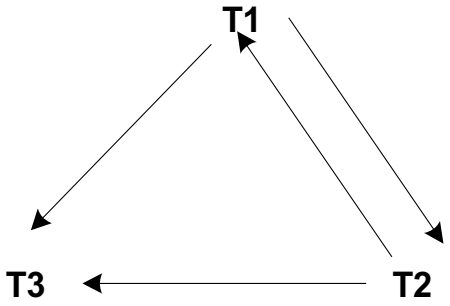
© 2016 Zvi M. Kedem

Conflict Graph Is "Too Pessimistic"

- There are serializable histories (i.e., equivalent to serial) that have cyclic conflict graph.
- Following is an example
  - R x is not needed for the example but was added to make the example "more realistic"
- Each transaction "behaves the same" in both histories.
- The final database is the same in both histories.



History



Conflict graph (not acyclic)

T1

T2

T3

Equivalent Serial history

© 2016 Zvi M. Kedem

A Common Practice: Rigorous Two-Phase Locking Protocol

- Commercial systems generally allow only histories that are conflict-serializable, i.e., allow histories that have acyclic conflict graphs
- They generally do not examine the conflict graph as it is being created
- Instead, ***DB OS forces the transactions to follow a protocol, which will assure that the resulting graph would be acyclic (if examined)***
- Therefore there is no need to examine the graph
- The major protocol in use: rigorous two-phase locking
- The protocol uses locking

© 2016 Zvi M. Kedem

Locks

- Two types of lock can be set on each item:
  - ***X-lock*** (e***X***clusive lock)
  - ***S-lock*** (***S***hared lock)
- There are privileges associated with locks:
  - To write an item, a transaction must hold an X-lock on it
  - To read an item, a transaction must hold an S-lock or an X-lock on it

© 2016 Zvi M. Kedem

Locks

- Two types of lock can be set on each item:
  - ***X-lock*** (e***X***clusive lock)
  - ***S-lock*** (***S***hared lock)
  - ***N*** means: ***N***o lock
- Compatibility of locks for an item:
  - Any number of transactions can hold S-locks on the item;
  - If any transaction holds an X-lock on the item, then no transaction may hold any lock on the item

Type	**N**	**S**	**X**
N	Yes	Yes	Yes
S	Yes	Yes	No
X	Yes	No	No

© 2016 Zvi M. Kedem

Locks

- If a transaction wants to get a lock (we will see later how/when this is done)
  - DB OS may give the lock to it if possible (obeys compatibility as above),
  - Otherwise the transaction may need to wait for the lock, or the DB OS may abort it
- When the transaction no long needs a lock (we will know later when this time arrives) the lock may be released by the DB OS
- There problems that need to be addressed (well known from OS, so we do not focus on them for now):
  - Deadlocks
  - Starvation

© 2016 Zvi M. Kedem

Acquiring And Releasing Locks

- T is one of the following states with respect to x
  - **N x** (not locked: does not have a lock on x)
  - **S x** (has a shared lock on x)
  - **X x** (has an exclusive lock on x)
- Operations to acquire locks (lock requests)
  - T: N $\rightarrow$ S x (T acquires an S-lock on x), also written T S x
  - T: N $\rightarrow$ X x (T acquires an X-lock on x), also written T X x
  - T: S $\rightarrow$ X x (T upgrades from an S-lock to an X-lock on x)
- Operations to release locks (unlock requests)
  - T: S $\rightarrow$ N x (T releases an S-lock on x), also written T N x
  - T: X $\rightarrow$ N x (T releases an X-lock on x), also written T N x
  - T: X $\rightarrow$ S x (T downgrades from an X-lock to an S-lock on x)

© 2016 Zvi M. Kedem

Locking Is Not Enough

- In our example, we can bracket each operation by acquiring and releasing a lock and still get a non-serializable history (we put it just for the first operation, we need to do for all, but no space to do it here…)

T1: S x
T1: READ x
T1: N x
T1: WRITE x

T2: READ x
T2: WRITE x
T2: READ y
T2: WRITE y

T1: READ y
T1: WRITE y

© 2016 Zvi M. Kedem

Two-Phase Locking Constraint On Timing

- ***Two phase locking (2PL)*** satisfies the following constraint
- During its execution, each transaction is divided into two phases:
  - During the ***first phase***, it issues lock requests: N $\rightarrow$ S x, N $\rightarrow$ X x, S $\rightarrow$ X x; this phase is also called the ***growing phase***
  - During the ***second phase***, it releases the locks: X $\rightarrow$ N x, S $\rightarrow$ N x, X $\rightarrow$ S x; this phase is also called the ***shrinking phase***
- For each transaction Ti, we can define a time point, ***Li***, its ***lock point***: the boundary between the first and the second phase
- For convenience, this will be the point when Ti requires its last lock

© 2016 Zvi M. Kedem

Two-Phase Locking
Constraint On Timing

- This transaction followed the two-phase locking protocol
  1. N → S a
  2. N → X b
  3. S → X a
  4. N → S c
  5. N → X d
  6. X → S d
  7. S → N c
  8. X → N a
  9. S → N d
  10. X → N b
- The lock point was 5
- The growing phase was 1–5
- The shrinking phase was 6–10

© 2016 Zvi M. Kedem

Two-Phase Locking Constraint On Timing

- This transaction ***did not follow*** the two-phase locking protocol
  1. N → S a
  2. N → X b
  3. S → X a
  4. N → S c
  5. X → S a
  6. N → X d
  7. S → N c
  8. S → N a
  9. X → N d
  10. X → N b
- There was a “growing” action (6) after a “shrinking” action (5)
- Therefore, the transaction did not follow the two-phase locking protocol

© 2016 Zvi M. Kedem

Example Of Two Phase Locking

	T1	T2
1.	Starts	
2.	S a	
3.		Starts
4.	R a	
5.		S b
6.		X c
7.		R b
8.	S c (waits)	
9.		W c
10.		X d
11.		N c
12.		N b
13.	S c	
14.	R c	
15.	N c	
16.	N a	
17.	Commits	
18.		W d
19.		N d
20.		Commits

© 2016 Zvi M. Kedem

Example Of Two Phase Locking

- Some observations follow
- The execution was concurrent and not serial
- L1 = 13
- L2 = 10
- Transactions do not have to unlock items in the same ordered they locked them
- A transaction can continue executing and accessing items it still has locked even after it has unlocked some items
- This history is not recoverable: more about this later

© 2016 Zvi M. Kedem

Our Original Non-Serializable History

	T1	T2
1.	R x	
2.	W x	
3.		R x
4.		W x
5.		R y
6.		W y
7.	R y	
8.	W y	

- This could not have happened under 2PL, because
  - T2 had to have an X-lock on y before (6).
  - T1 had to have an X-lock on y before (8)
  - Since T1 cannot unlock y and then lock it again (2PL), it could have locked it only after (6)
  - But T1 had to both X-lock x before (2) and unlock it before (4), so that T2 could lock it
  - But then T1 unlocked x and then locked y, a contradiction

© 2016 Zvi M. Kedem

But Using Two Phase Locking

1.	T1	T2
2.	X x	
3.	R x	
4.	W x	
5.		X x (waits)
	T1 completes	
		T2 continues and completes

- So we got a serializable execution, which happens to be serial in this case

© 2016 Zvi M. Kedem

Another Example

	T1	T2
1.		
2.	X x	
3.	R x	
4.	W x	
5.		X y
6.		W y
7.		X z
8.	X z (waits)	
9.		N y
10.		W z
11.		N z
12.	X z	
13.	W z	
14.	N z	
15.	N x	

- So we got a serializable execution, which was concurrent in this case

© 2016 Zvi M. Kedem

2PL Guarantees Serializability

- ***Theorem: if all the transactions in the system follow the two-phase locking protocol, then the conflict graph is acyclic (and therefore the history is serializable)***

- ***Lemma: If T1 → T2 in the conflict graph, then L1 < L2 in time (L1 was earlier than L2)***
- Proof:
  - On some x, for conflicting operations (at least one of them WRITE)

    T1 OP1 x at time t1
    T2 OP2 x at time t2

    t1 < t2

© 2016 Zvi M. Kedem

2PL Guarantees Serializability

- Therefore:
  - T1 held a lock on x at time t1
  - T2 held a lock on x at time t2
  - the two locks could not co-exist in time as at least one of them was an X-lock (to allow the WRITE)
- So T2 could lock x only after T1 unlocked it and therefore for some instances t' and t'', such that $t1 < t' < t'' \leq t2$:
  - T1 unlocked x at t'
  - T2 locked x at t''
- So by the definitions of L1 and L2
  - $L1 < t' < t'' \leq L2$
- This finishes the proof of the lemma

© 2016 Zvi M. Kedem

2PL Guarantees Serializability

- Assume, by contradiction, that a history obtained following 2PL contains a cycle T1 $\rightarrow$ T2, T2 $\rightarrow$ T3, …, Tn $\rightarrow$ T1
- By the lemma:
  - L1 < L2
  - L2 < L3
  - …
  - Ln < L1
- Therefore: L1 < L2 < L3 < … < Ln < L1
- And we reach a contradiction: L1 < L1.
- Therefore there could not have been a cycle and the history was serializable

© 2016 Zvi M. Kedem

Standard 2PL Is Not Sufficient

- It allows non-recoverable histories, such as

T1	T2
X x	
W x	
N x	
	S x
	R x
	X y
	W y
	N x
	N y
	Commit
Abort	

- T2 has to abort, but cannot because it has committed
- So we will modify the protocol so that ***it only produces strict histories*** (better than recoverable) and this is exactly what recovery needed (as we discussed and assumed in the recovery unit)

© 2016 Zvi M. Kedem

Strict 2PL

- All the conditions of 2PL
- All exclusive locks are released after commit or abort
- Therefore:
  - Every transaction reads only values produced by transactions that have already committed
  - Every transaction, if it writes an item, all the transactions that previously wrote that item have already committed or aborted
- These were exactly the conditions for a strict history

© 2016 Zvi M. Kedem

Rigorous 2PL

- In practice, the programmer does not issue the various locking and unlocking instructions
- In practice, whenever a transaction attempts to access a variable for the first time in some mode (Read or Write), the DB OS tries to give it the appropriate lock (Shared or Exclusive)
- The transaction may have to wait to get the lock (and may have to be aborted in case there are deadlocks), but all this is transparent to the programmer who wrote the transaction
- All locks released only after a commit or an abort
- So: concurrency control is transparent to the programmer

- And of course histories are strict (and therefore also no cascading aborts and they are recoverable)

© 2016 Zvi M. Kedem

DB OS Enforcing Rigorous Histories

- When transaction issues a Read on x
  - If it has any lock on x, let it proceed
  - Otherwise, if no other transaction has an X-lock on x, give it an S-lock
  - Otherwise, keep it waiting until an S-lock can be given
  - May have/want to abort it
- When a transaction issues a Write on x
  - If it has an X-lock on x, let it proceed
  - Otherwise, if no other transaction has any lock on x, give it an X-lock, by either giving it directly or by upgrading/converting an existing S-lock it has on x to give it an X-lock
  - Otherwise, keep it waiting until an X-lock can be given
  - May have/want to abort it
- Release all locks only after commit or abort (of course, abort requires undoing, which recovery should take care of, though we did not discuss details there)

© 2016 Zvi M. Kedem

Phantoms
(Discussed More Extensively Later)

- ***We assumed in our discussion that no new items are added to the database***
- If new items are added, ***phantoms*** may appear
- For example if we to multiply every account for SSN between 123456789 and 200000000 by 2 (converting into a different currency, perhaps) the following may happen:
  - We X-lock all accounts
  - We start multiplying accounts by 2
  - In the ***middle of processing*** another account is added
  - We do not know about it, so we do not multiply it by 2
- Such accounts that appear in the middle and not processed are ***phantoms***
- To handle phantoms, that is preventing their appearance, ***range locks*** can be introduced
- So in our example accounts between 123456789 and 200000000 cannot be added during processing

© 2016 Zvi M. Kedem

SQL Standard and Oracle Implementation

© 2016 Zvi M. Kedem

Transaction And Queries

- ***A transaction is a sequence of queries***
- A typical query is a SELECT statement

- We encountered this before, without explicitly talking about this

- Some constrains needed to be satisfied after each query: they could not be deferred
- Some constrains needed to be satisfied only after the last query: they could be deferred

© 2016 Zvi M. Kedem

SQL Access Modes And Isolation Levels

- The standard is somewhat controversial and ***not consistently applied in commercial systems***
- User can set ***access mode*** and ***isolation level*** for a transaction
- Access mode is one of the following
  - READ ONLY (implies the transaction will only read; if it tries to write it must be aborted)
  - READ WRITE (implies the transaction may read and write)
- Isolation level is one of the following, in ***decreasing order of correctness***
  - SERIALIZABLE
  - REPEATABLE READ
  - READ COMMITTED
  - READ UNCOMMITTED

© 2016 Zvi M. Kedem

Serializable Isolation Level (Reference Implementation Using Locks)

- Two-phase locking with X-Locks and S-Locks held until after commit
- Guarantees Serializability (in our original sense)

- **Range Locks are required to handle phantoms**
- That is, a transaction can lock, say, all records with ID in the range from 100 to 199
- Guarantees no phantoms
- First we see why something needs to be done to handle phantoms

© 2016 Zvi M. Kedem

Handling Phantoms

- Imagine that we want to give each employee in the range $1 raise, and do not consider phantoms
- We lock sequentially all the ***existing*** items in the range, one by one, and give each employee a raise
- Say we just have 3 employees in the range
- We lock 105, give raise, lock 135, give raise, lock 189, give raise, unlock all
- After 135 got a raise, a new item with ID 127 is inserted. There is no conflict with anything but our transaction does not know about it as it already moved beyond 135
- Item 127 was a ***phantom***
- But if the range 100…199 is locked, no items can be added
- In this way phantoms are prevented

© 2016 Zvi M. Kedem

Read Repeatable Isolation Level (Reference Implementation Using Locks)

- Two-phase locking with X-Locks and S-Locks held until after commit
- No Range Locks are assumed

- Guarantees Serializability (in our original sense)
- Phantoms possible

© 2016 Zvi M. Kedem

Read Committed Isolation Level (Reference Implementation Using Locks)

- Two-phase locking for writing, with X-Locks held until after commit

- For reading a committed value is given to the transaction (from the log, likely)
- Different committed values of the same item can be given to the transaction at different points of the execution of the transaction
- But even if the ***same committed value*** is given, the execution ***may not preserve consistency*** of the database even if the transactions when run serially preserve it
  - Example next

© 2016 Zvi M. Kedem

Read Committed Isolation Level May Not Preserve Consistency

- Consistency requirement: $a + b \geq 1$
- T1: **if** $a = 1$ **then** $b := 0$
- T2: **if** $b = 1$ **then** $a := 0$
- Old committed values: $a = b = 1$
- History

T1	T2
Read *a*	
	Read *b*
X-lock *b*	
b := 0	
Commit	
Unlock *b*	
	X-lock *a*
	a := 0
	Commit
	Unlock *a*

- New committed values: $a = b = 0$

© 2016 Zvi M. Kedem

Not Being Serializable May Be OK

- Alice has account *a* and Bob has account *b*
- They both can look at both accounts but only modify their own accounts
- Each of them, when seeing that the other's account is below $100 adds $100 to his/her own account
- Let us look at a scenario in which T1 is run by Alice and T2 is run by Bob

© 2016 Zvi M. Kedem

Not Being Serializable May Be OK

- Initial state *a* = *b* = $50 and these are committed values
- History

T1	T2
Read *a*	
	Read *b*
X-lock *b*	
b := $150	
Commit	
Unlock *b*	
	X-lock *a*
	a := $150
	Commit
	Unlock *a*

© 2016 Zvi M. Kedem

Not Being Serializable May Be OK

- This could not have happened in any serial execution
- In a serial execution we will end up in one of the two situations
  - *a* = $50 and *b* = $150
  - *a* = $150 and *b* = $50
- However, they may not care that both of them replenished their accounts thinking that the other's account was too low
- So we could allow such a non-serializable execution

- ***Non-serializable histories may be OK if they perform in a semantically acceptable way, but this needs to handled in a case-by-case basis as the general theory does not understand these considerations***

© 2016 Zvi M. Kedem

Read Uncommitted Isolation Level (Reference Implementation Using Locks)

- Two-phase locking for writing with X-Locks held until after commit

- The transaction can read an item at any time

  Dirty reads (temporary values during an execution of some transaction) may be read

© 2016 Zvi M. Kedem

Implication Of Isolation Levels (Summary)

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
READ UNCOMMITED	Y	Y	Y
READ COMMITTED	N	Y	Y
READ REPEATABLE	N	N	Y
SERIALIZABLE	N	N	N

- We had READ REPEATABLE in our formal development because we did not handle phantoms, just to simplify the development and the presentation

© 2016 Zvi M. Kedem

Oracle
Rigorous Two-Phase Locking

- This is the basic concurrency control mechanism

- As we discussed:
  - Locks are issued “automatically” based on what the transaction requests: read or write
  - All locks are held until after commit

© 2016 Zvi M. Kedem

Oracle
Multiversion Concurrency Control

- Oracle uses a ***multiversion*** concurrency control
- It maintains versions of committed items
- Actually not too difficult to do, but needs to be implemented efficiently

- Recall that committed values are stored in the ***log*** used for recovery as needed
- ***So if some transaction has an item locked for writing, and maybe already have written it but has not committed, the system can pull out a committed value from the log and give it to another transaction***

© 2016 Zvi M. Kedem

Oracle
Isolation Levels

- There are various ways of assigning consistency requirements to individual queries/statements and transactions
- If a transaction "gets into trouble" (isolation level is violated) some message is given by the system, and there are various options, e.g.,
  - Rollback the transaction partially (by the application code, if the programmer knows what to do)
  - Rollback the transaction completely (abort it)
- Specifying isolation levels (without full details):

ORACLE terminology/commands	SQL-standard equivalent
Set transaction isolation level read committed	Read committed
Set transaction isolation level serializable	Serializable
Set transaction isolation level read only	Read repeatable and no writes

© 2016 Zvi M. Kedem

Oracle
Read Committed

- ***This is the default isolation level for a transaction in Oracle***
- This is the same as SQL standard

- Implies, e.g.,

  If the same query is executed twice within a single transaction, the two executions of the query may get different values for the same item

© 2016 Zvi M. Kedem

Oracle Serializable

- This is at least as strong as SQL standard

- Oracle implementation

  For every item it will see the value that was committed when the transaction started or the value that it itself produced

© 2016 Zvi M. Kedem

Oracle
Correct Execution That is Not Permitted

- Consider T1 and T2 with SERIALIZABLE isolation level and a history of their execution

T1	T2
Start	
	Start
	X-Lock *a*
	Write *a*
	Commit
	Unlock *a*
S-Lock *a*	
Read *a*	
Commit	
Unlock *a*	

- T1 will be aborted when it attempts to S-Lock item *a*

© 2016 Zvi M. Kedem

Oracle
Read Only

- This is not a part of SQL standard

- Oracle implementation

  A transaction can only read

  For every item it will see the value that was committed when the transaction started

© 2016 Zvi M. Kedem

Oracle
Reading Version Committed at Start

- Consider T1 with READ ONLY isolation level and T2 with SERIALIZABLE isolation level and a history of their execution

T1	T2
Start	
	Start
	X-Lock a
	Write a
	Commit
	Unlock a
S-Lock a	
Read a	
Commit	
Unlock a	

- T1 will get the value of a that had been the most recently committed before T1 started and not the one produced by T2

© 2016 Zvi M. Kedem

Oracle
Read-Only Transaction

- ***Oracle handling of this makes perfect sense***
- It is enough to give a read-only transaction a consistent (preferably recent) snapshot of the database
- Such a snapshot can be created by looking at the log (used for recovery) and taking into account what was produced by committed transactions at a certain point in time.
- This snapshot reflected a correct state of the database
- We will see a simple example, but in detail, next

- := denotes write into the variable
- = denotes read the variable

© 2016 Zvi M. Kedem

T3 Read Only
Equivalent Histories; Second is Serial

T1	T2	T3	T4
Start			
a := 11			
b := 12			
Commit			
	Start		
	a := 21		
		Start	
	Commit		
		a = 11	
			Start
			b := 42
		b = 12	
			Commit
		Commit	

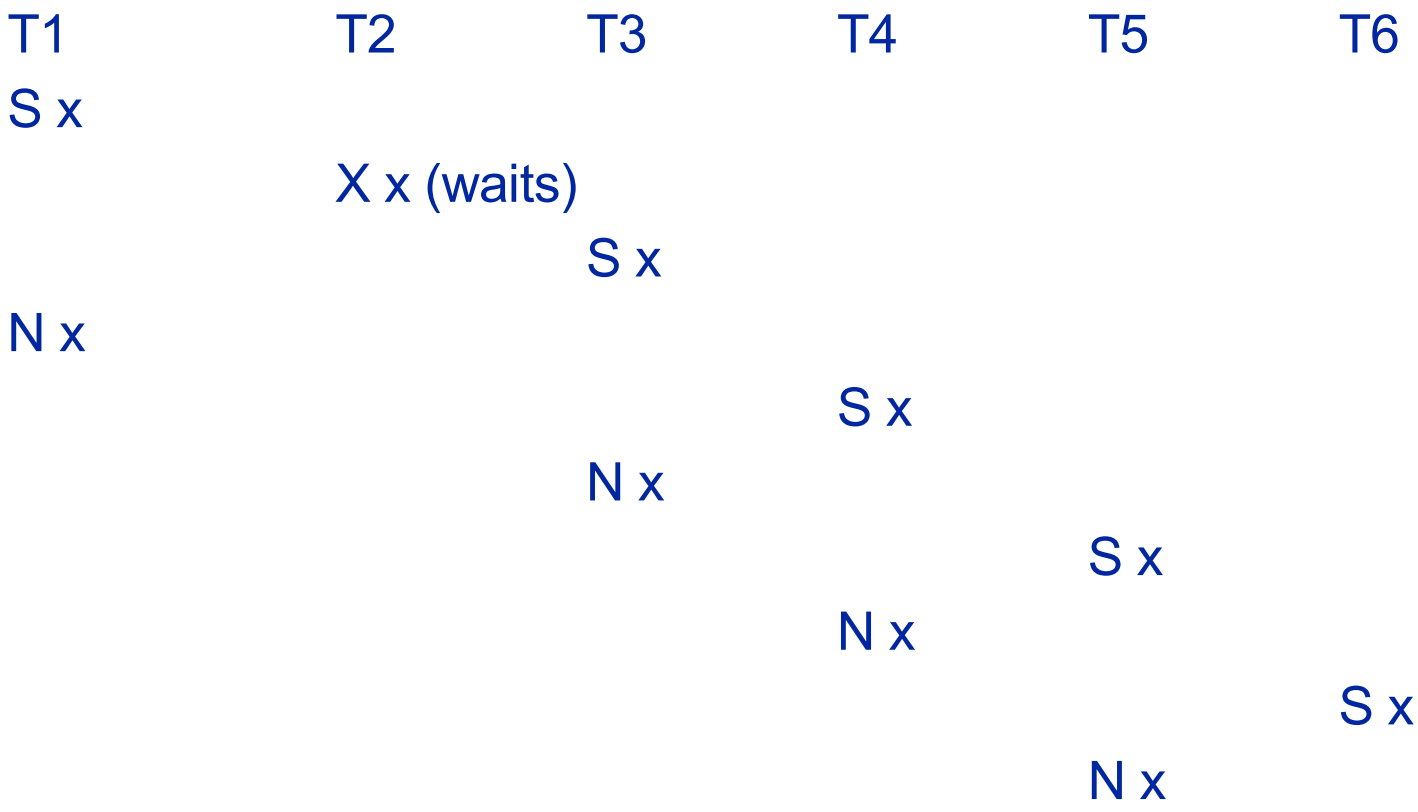
T1	T2	T3	T4
Start			
a := 11			
b := 12			
Commit			
		Start	
		a = 11	
		b = 12	
		Commit	
	Start		
	a := 21		
	Commit		
			Start
			b := 42
			Commit

© 2016 Zvi M. Kedem

Advanced Material

© 2016 Zvi M. Kedem

Locking Is Prone To Starvation



- This can continue indefinitely: T7, …
- Unless something is done, T2 will never gets the lock it wants
- Obvious solution, stop granting S-locks and when the last S-lock is released, give X-lock to T2

© 2016 Zvi M. Kedem

Two-Phase Locking Is Prone To Deadlocks

- Two transactions

  T1: x := x + 1; y := y + 1

  T2: y := 2y; x := 2x

	T1	T2
1.	T1	T2
2.	X x	
3.	R x	
4.	W x	
5.		X y
6.		R y
7.		W y
8.		X x (waits)
9.	X y (waits)	

- We got a deadlock
- ***In fact this deadlock prevented a non-serializable history***
- "Deadlocks are not a bug, but a feature"

© 2016 Zvi M. Kedem

Detecting And Avoiding Deadlocks

- Deadlocks are characterized by a cyclic “wait for” graph
- Ours was very simple, T1 waited for T2 and T2 waited for T1
- To detect if there is a deadlock, draw a “wait for” graph
  - Nodes: Transactions
  - Arc from T1 to T2 iff T1 waits for T2
- If there are cycles, some transaction need to be aborted
- There are protocols that avoid deadlock by aggressive abortion of transactions, sometimes not necessary
- They abort enough transactions, so that no cycles could ever appear in the “wait for” graph, so not need to draw it during execution
  - But they may abort transactions unnecessarily

© 2016 Zvi M. Kedem

Kill-Wait Protocol: Locking + More

- Each transaction, when entering the system is timestamped with the current time: timestamp of T is denoted by TS(T)
- If transaction Ti wants to lock x, which another transaction Tj holds in a conflicting mode (at least one of the two locks is an X-lock),
  - If TS(T1) < TS(T2), then abort T2 and give the lock to T1 (the older transaction kills the younger transaction)
  - If TS(T1) > TS(T2), then TS(T1) waits
- If a transaction unlocks a lock, the oldest from among the waiting transactions (all younger than the unlocking transaction) gets it
- In the “wait for” graph all the arcs are from a younger transaction to an older transaction, and therefore there cannot be a cycle

© 2016 Zvi M. Kedem

Wait-Die Protocol: Locking + More

- Each transaction, when entering the system is timestamped with the current time: timestamp of T is denoted by TS(T)
- If transaction Ti wants to lock x, which another transaction Tj holds in a conflicting mode (at least one of the two locks is an X-lock),
  - If TS(T1) < TS(T2), then T1 waits
  - If TS(T1) > TS(T2), TS(T1) abort T1 (T1 dies)
- If a transaction unlocks a lock, the youngest from among the waiting transactions (all older than the unlocking transaction) gets it
- In the “wait for” graph all the arcs are from an older transaction to a younger transaction, and therefore there cannot be a cycle

© 2016 Zvi M. Kedem

Timestamp-Based Protocol For Concurrency

- Each transaction is issued a timestamp when accepted by DB OS
- The first transaction gets the timestamp 1
- Every subsequent transaction gets the timestamp that is the previously largest assigned timestamp + 1
- So will can refer to transactions as “older” and “younger” based on their timestamps and also use timestamp value for transaction identification
- The system maintains for each item x two timestamps:
  - ***RT(x)*** is the youngest transaction (largest timestamp) that read it
  - ***WT(x)*** is the youngest transaction (largest timestamp) that wrote it

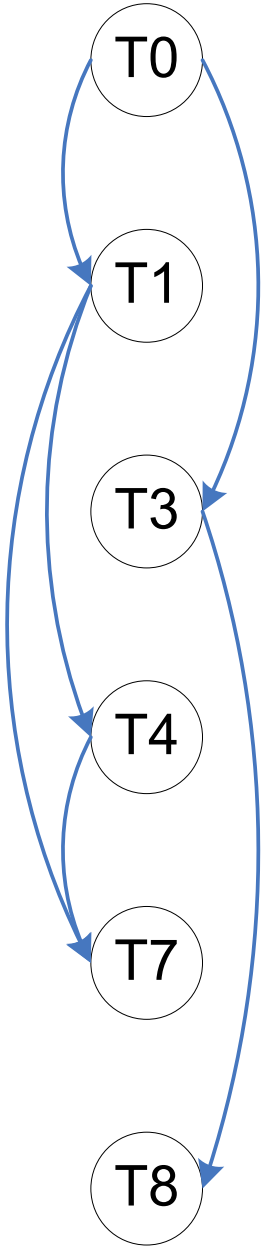
© 2016 Zvi M. Kedem

Timestamp-Based Protocol

- Assume that T1, T2, … arrive in this order and that the time stamp of Ti is i
- For simplicity assume that the database was created by transaction T0
- ***We want to get schedules equivalent to the serial order T0, T1, T2, …, or some subsequence of this***, as some transactions can abort, so will not appear in the schedule
- ***If we do this, our schedule will be serializable***
- Similarly to what we did during topological sort, we could say that Ti executed instantaneously at virtual time Ti

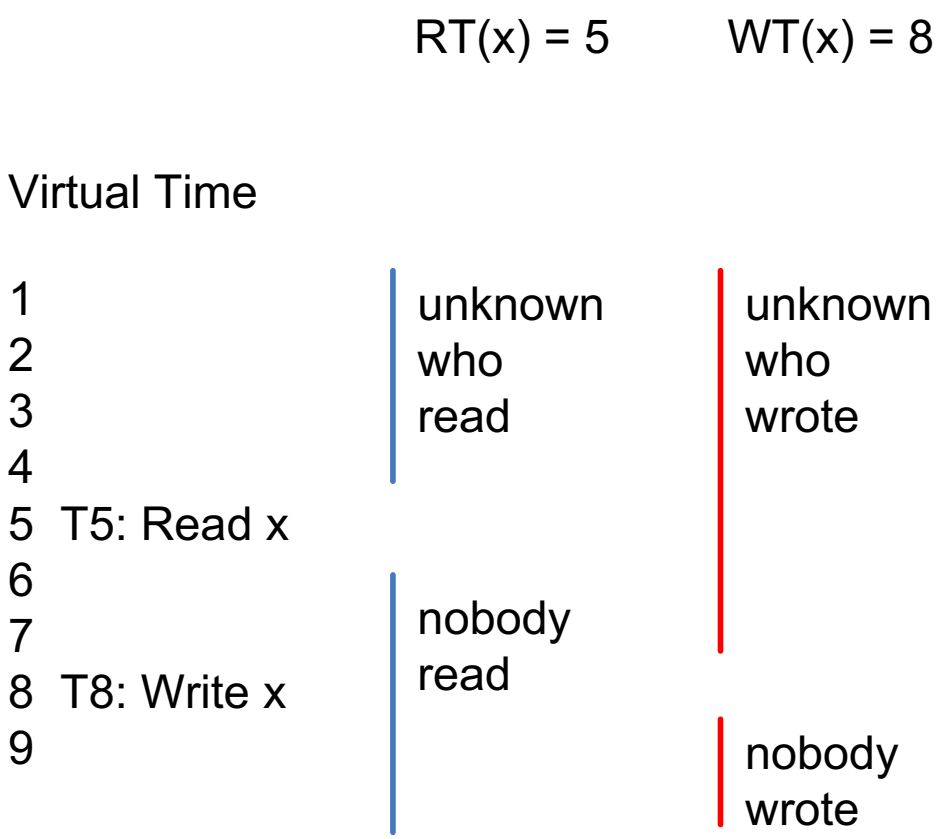
© 2016 Zvi M. Kedem

Equivalent Serial Schedule



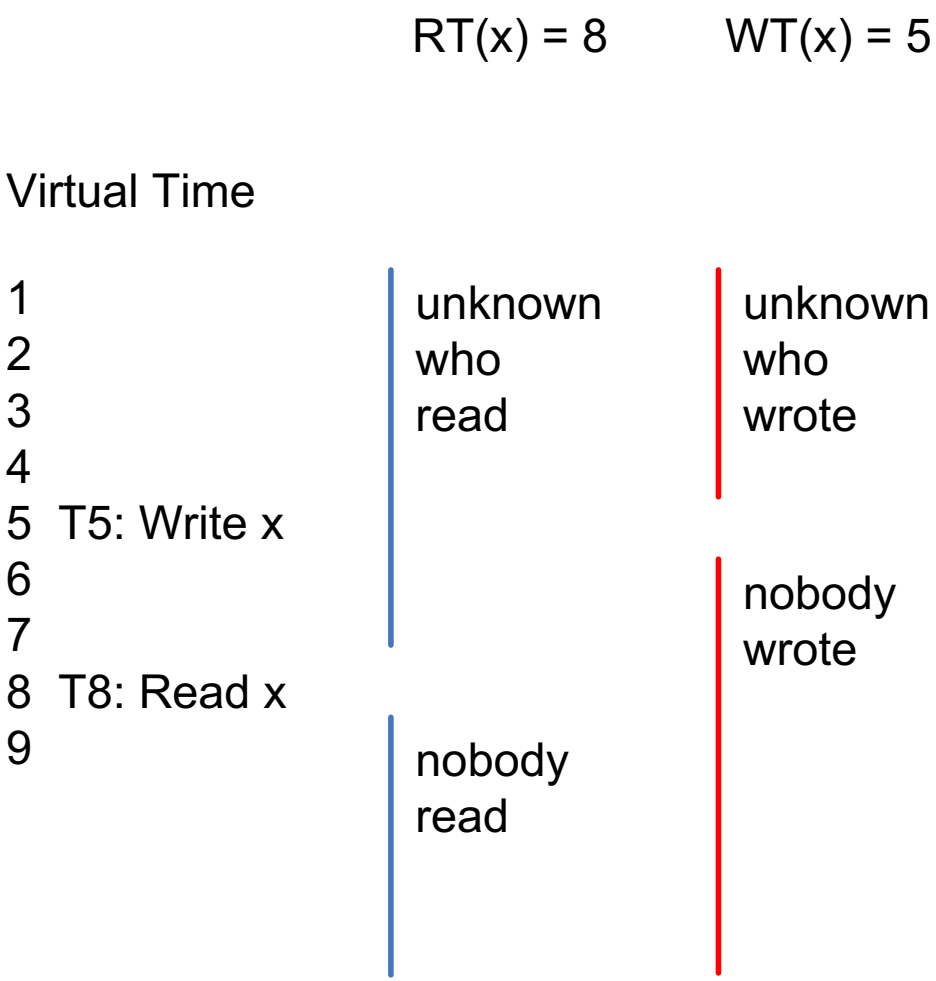
© 2016 Zvi M. Kedem

Scenario



© 2016 Zvi M. Kedem

Scenario



© 2016 Zvi M. Kedem

Timestamp-Based Protocol: Reading

- RT(x) = 5; this is the youngest transaction that read it
- WT(x) = 8; this is the youngest transaction that wrote it
- If a transaction Ti with a timestamp of $i \leq 7$, say T6, wants to read x
  - The value it wanted no longer exists (it had to be written by T0 (i.e., initial state of the DB), or by Ti with $i \leq 6$)
  - T6 cannot read x and has to be aborted
- If a transaction with a timestamp of $i \geq 8$, say T9, wants to read it,
  - T9 reads x, and RT(x) := 9

© 2016 Zvi M. Kedem

Timestamp-Based Protocol: Reading

- RT(x) = 8; this is the youngest transaction that read it
- WT(x) = 5; this is the youngest transaction that wrote it
- If a transaction Ti with a timestamp of $i \leq 4$, say T3, wants to read x
  - T3 cannot read x and has to be aborted (as too new a value of x exists
- If a transaction Ti with a timestamp i, $5 \leq i \leq 8$, say T6, wants to read x
  - T6 reads x
- If a transaction Ti with a timestamp of $i \geq 9$, say T9, wants to read it
  - T9 reads x and RT(x) := 9

© 2016 Zvi M. Kedem

Timestamp-Based Protocol: Writing

- RT(x) = 5; this is the youngest transaction that read it
- WT(x) = 8; this is the youngest transaction that wrote it
- If a transaction Ti with a timestamp of i < 5 wants to write, say T3,
  - T3 has to be aborted
  - Because there was a read of a value of x by transaction T5, and maybe this was a value produced actually by T2. If we allow T3 to write, this would have meant that T5 read a value that was produced by a transaction that was too old
- If a transaction Ti with a timestamp of i > 8 wants to write
  - T9 writes and WT(x) = 9
- If a transaction Ti with a timestamp of i , $6 \leq i \leq 7$, say T7, wants to write
  - We just throw out the write and let T7 proceed
  - This was a blind write, nobody read it and it is obsolete (and nobody will be allowed to read it as described above; we will not go back to re-examine this case and check this out)

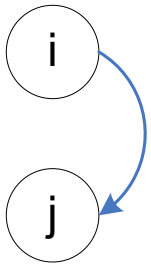
© 2016 Zvi M. Kedem

Timestamp-Based Protocol: Writing

- RT(x) = 8; this is the youngest transaction that read it
- WT(x) = 5; this is the youngest transaction that wrote it
- If a transaction Ti with a timestamp of i < 8, say T6, wants to write
  - T6 has to be aborted
  - Because there was a read of a value of x by transaction T8, and if we allow T6 to write x, this would mean that T8 read a value that was too old
- If a transaction Ti with a timestamp of i > 8, say T9, wants to write
- T9 writes and WT(x) = 9

© 2016 Zvi M. Kedem

Conflict Serializability And Deadlock Freedom



- In the conflict graph all the arcs will be from an older transaction to a younger transaction
- Therefore the history will be conflict-serializable
- And as transactions never wait, there will be no deadlocks
- But the history may not even be recoverable
- We can make it strict, or even rigorous, by having transactions wait until the relevant transactions commit
- There still will not be any deadlocks, because younger transactions wait for older transactions to commit, but not the other way around

© 2016 Zvi M. Kedem

Granularity Of Locks

- The problem of phantoms can be avoided, by say, locking the file that has all the accounts, and therefore no account can be added during the processing
- Sometimes we may want to lock all the accounts (logically, so no new accounts can be added)
- Sometimes we may want to lock an account, to add money to it, for instance
  - And of course, it is not efficient to lock all the accounts in order to modify one account only
- So “lockable” objects are no longer disjoint items
- This can be handled using somewhat more complex types of locks (called ***intention*** locks)
- You can read about Intention Locks and the protocol using them at http://en.wikipedia.org/wiki/Multiple_granularity_locking

© 2016 Zvi M. Kedem

Granularity Of Locks (Simplified Granularity-Based Locking)

- See https://docs.oracle.com/cd/E17952_01/refman-5.1-en/innodb-lock-modes.html
- There are 4 types of locks and their compatibility matrix is

Type	N	S	X	IS	IX
N	Yes	Yes	Yes	Yes	Yes
S	Yes	Yes	No	Yes	No
X	Yes	No	No	No	No
IS	Yes	Yes	No	Yes	Yes
IX	Yes	No	No	Yes	Yes

- IS stands for "Intent ***S***hared"
- IX stands for "Intent e***X***clusive"

- As usual, the matrix states what two transactions can hold simultaneously, a transaction can hold any set of locks on an item (as long as not conflicting with other transactions)

© 2016 Zvi M. Kedem

Granularity Of Locks (Simplified Granularity-Based Locking)

- Assume one table R and five rows 1, 2, 3, 4, 5.
- The items that can be locked are R, 1, 2, 3, 4, 5.
- To have write access to all of R, a transaction needs an X-lock on R
- To have a read access on all of R, a transaction needs an X-lock or an S-lock on R
- To have write access to a row of R, a transaction needs an X-lock on that row
- To have a read access on a row of R, a transaction needs an X-lock or an S-lock on that row
- To set an X-lock on a row of R, a transaction needs an IX-lock on R
- To set an S-lock on a row of R, a transaction needs an IX-lock or an IS-lock on R
- Locking is top to bottom; unlocking is bottom to top

© 2016 Zvi M. Kedem

Granularity Of Locks (Simplified Granularity-Based Locking)

- The goal is to prevent situations such as
  - T1 holds X-lock on R and T2 holds an S-lock on 1
  - T1 holds X-lock on R and T2 holds an X-lock on 1
  - T1 holds S-lock on R and T2 holds an X-lock on 1
- The goal is to permit situations such as
  - T1 holds S-lock on R and T2 holds an S-lock on 1
  - T1 holds S-lock on 1 and T2 holds an X-lock on 2
  - T1 holds X-lock on 1 and T2 holds an X-lock on 2

© 2016 Zvi M. Kedem

Key Ideas

- The concurrency problem
- Ensuring Isolation
- The need for abstraction to sequence of Reads and Writes during concurrent execution
- Histories/Schedules
- Equivalent histories
- Serializability: equivalence with a serial history
- Conflicts
- Conflict serializable histories
- Conflict graphs
- Locks
- Two phase locking (2PL) to assure serializability
- Strict 2PL to assure recoverability

© 2016 Zvi M. Kedem

Key Ideas

- Rigorous 2PL to make correct locking transparent to the user
- Phantoms
- SQL access modes and isolation levels
- Oracle implementation
- Starvation
- Deadlocks
- Time-stamp based protocols
- Granularity of locking

© 2016 Zvi M. Kedem