Module 17: Transactions

Database System Concepts, 7th Ed.

Slides, having no footer, have been taken from Prof Mohammad Hammoud, CMU Qatar, lecture notes

Outline

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g., transaction to transfer Rs50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Example of Fund Transfer

- Transaction to transfer Rs50 from account A to account B:
    1. **read**(*A*)
    2. *A* := *A* – 50
    3. **write**(*A*)
    4. **read**(*B*)
    5. *B* := *B* + 50
    6. **write**(*B)*
- **Atomicity requirement**
    - If the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state
        - Failure could be due to software or hardware
    - The system should ensure that updates of a partially executed transaction are not reflected in the database
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the Rs50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Example of Fund Transfer (Cont.)

- **Consistency requirement** in above example:
  - The sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer (Cont.)

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

T1	**T2**
1. **read**(A)	
2. $A := A – 50$	
3. **write**(A)	
	read(A), read(B), print(A+B)
4. **read**(B)	
5. $B := B + 50$	
6. **write**(B	

- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

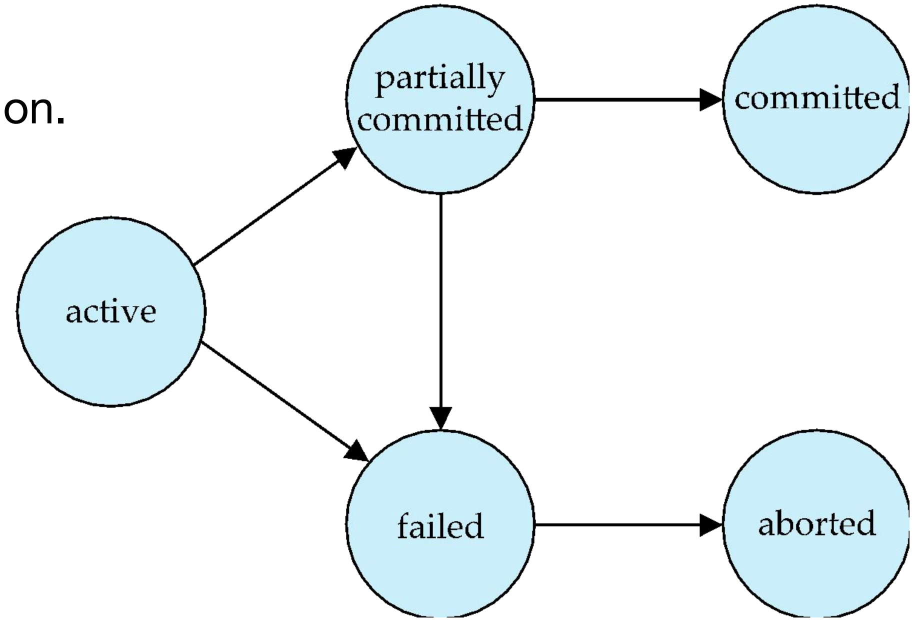
ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** -- after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - Restart the transaction
    - Can be done only if no internal logical error
  - Kill the transaction
- **Committed** – after successful completion.



Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
    - **Increased processor and disk utilization**, leading to better transaction *throughput*
        - E.g., one transaction can be using the CPU while another is reading from or writing to the disk
        - Better system throughput
    - **Reduced average response time** for transactions: short transactions need not wait behind long ones.
        - Better response time
        - Fairer
- **Concurrency control schemes** – mechanisms to achieve isolation
    - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - By default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedule 1

- Let T_1 transfer Rs50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B.*
- A serial schedule in which T_1 is followed by T_2 :

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read (A)
	temp := A * 0.1
	A := A - *temp*
	write (A)
	read (B)
	B := B + *temp*
	write (B)
	commit
read (A)	
A := A – 50	
write (A)	
read (B)	
B := B + 50	
write (B)	
commit	

Schedule 3

- Let T_1 transfer Rs50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B.*
- The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

- In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of ($A + B$).

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus, serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. **Conflict serializability**
  2. **View serializability**

Serializable Schedules

- Two schedules are said to be *equivalent* if for any database state, the effect of executing the 1st schedule is identical to the effect of executing the 2nd schedule

- A serializable schedule is a schedule that is equivalent to a serial schedule

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)
	Commit
Commit	

A *Serializable* Schedule

Equivalent

T1	T2
R(A)	
W(A)	
R(B)	
W(B)	
Commit	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit

A *Serial* Schedule

Not Equivalent

Equivalent to which serial schedule?

T1	T2
	R(A)
	W(A)
R(A)	
	R(B)
	W(B)
W(A)	
R(B)	
W(B)	
	Commit
Commit	

Another *Serializable* Schedule

Examples

- Assume transactions T1 and T2 as follows:

```
T1:  BEGIN   A=A-100,   B=B +100   END
T2:  BEGIN   A=1.06*A,   B=1.06*B   END
```

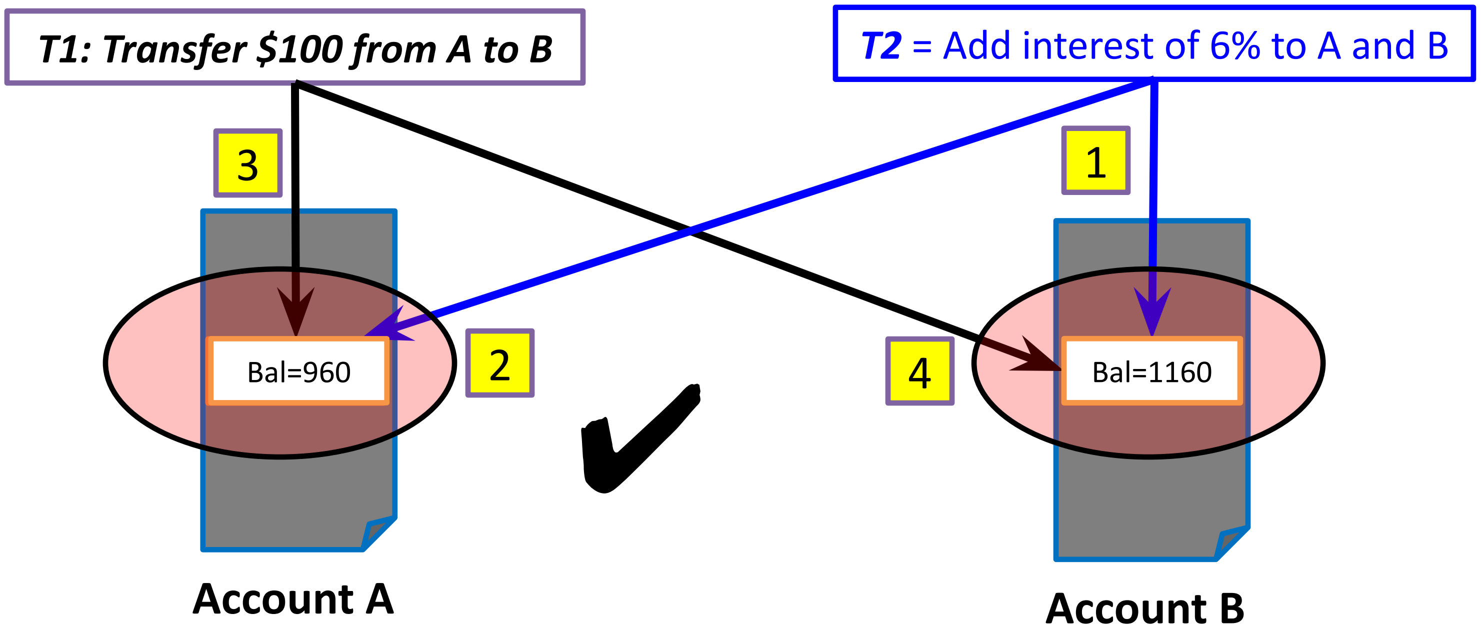
- T1 can be thought of as transferring $100 from A’s account to B’s account

- T2 can be thought of as crediting accounts A and B with a 6% interest payment

Examples: A *Serial* Schedule

- Assume transactions T1 and T2 as follows:

```
T1:  BEGIN   A=A-100,   B=B +100   END
T2:  BEGIN   A=1.06*A,   B=1.06*B   END
```

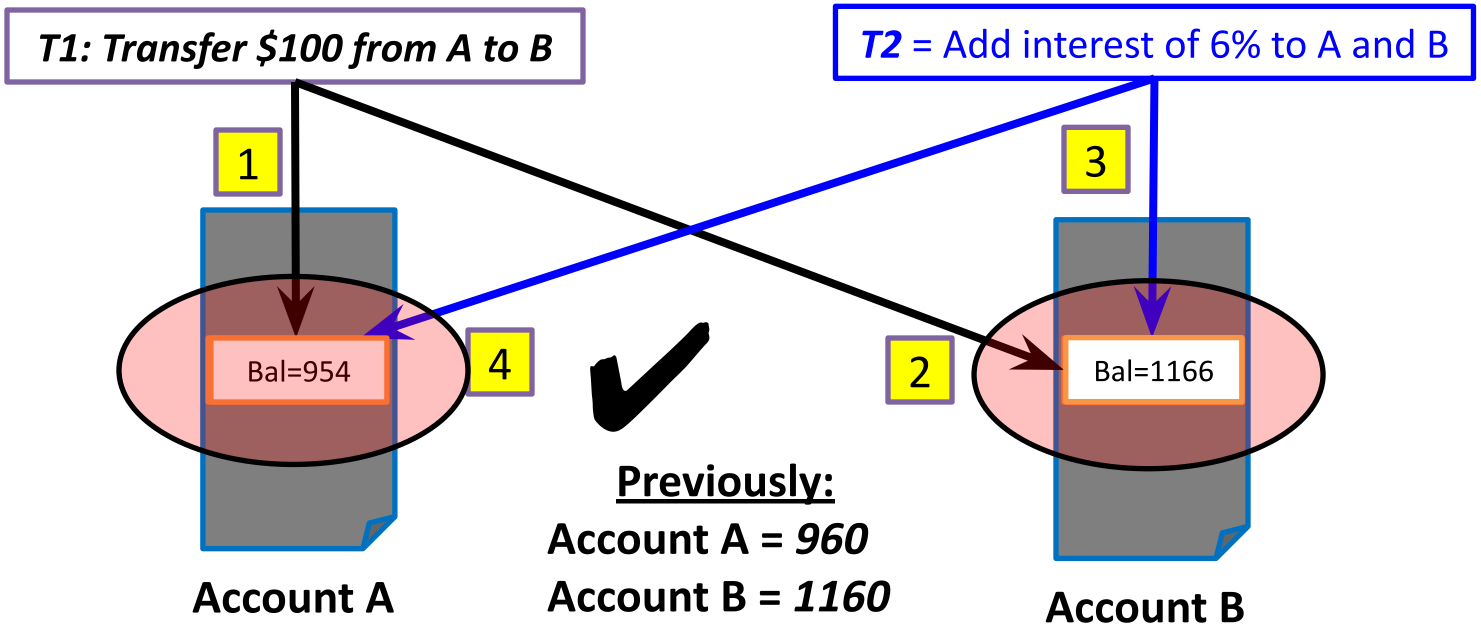


T1: Transfer $100 from A to B
T2 = Add interest of 6% to A and B
3
1
2
4
Bal=960
Bal=1160
✓
Account A
Account B

Examples: Another *Serial* Schedule

- Assume transactions T1 and T2 as follows:

```
T1:  BEGIN   A=A-100,   B=B +100   END
T2:  BEGIN   A=1.06*A,   B=1.06*B   END
```



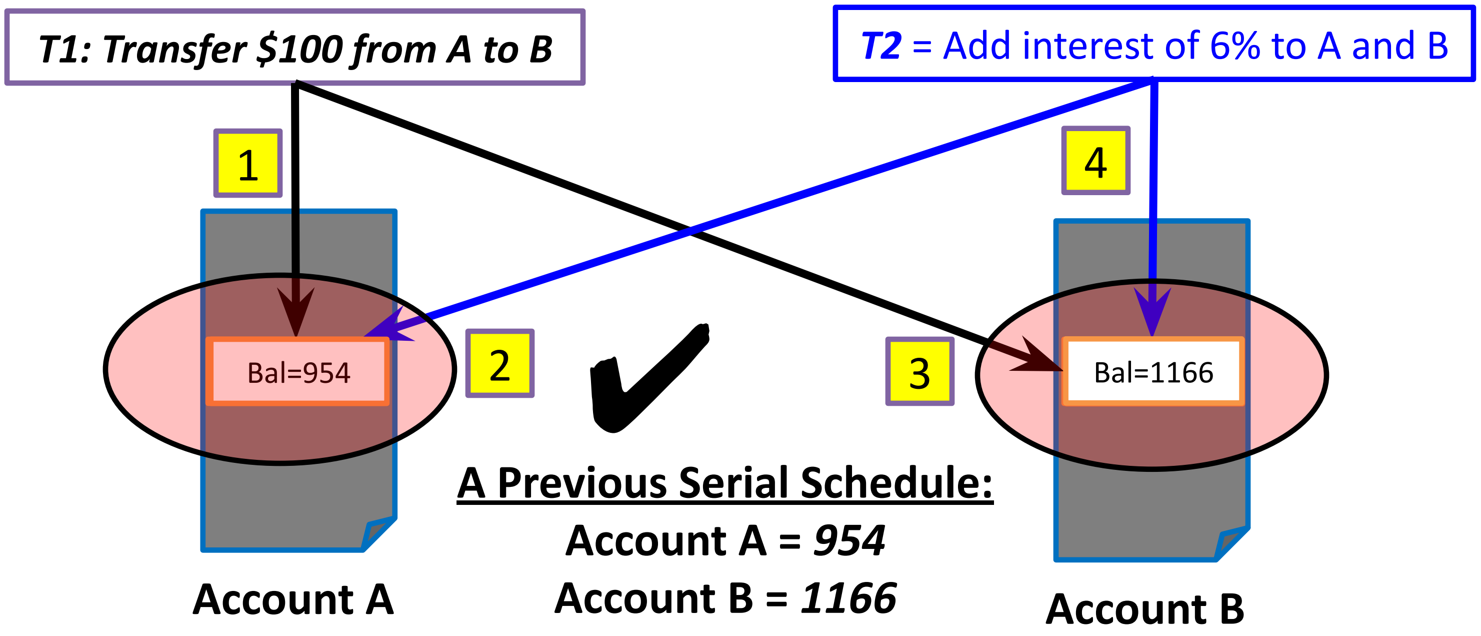
T1: Transfer $100 from A to B
T2 = Add interest of 6% to A and B
1
2
3
4
Bal=954
Bal=1166
Account A
Account B
✔

Previously:
Account A = *960*
Account B = *1160*

Examples: A *Serializable* Schedule

- Assume transactions T1 and T2 as follows:

```
T1:  BEGIN   A=A-100,   B=B +100   END
T2:  BEGIN   A=1.06*A,   B=1.06*B   END
```



T1: Transfer $100 from A to B
T2 = Add interest of 6% to A and B
1
2
3
4
Bal=954
Bal=1166
✔
A Previous Serial Schedule:
Account A = *954*
Account B = *1166*
Account A
Account B

Anomalies Due to Concurrency

- Interleaving actions of different transactions can leave the database in an inconsistent state

- Two actions on the same data object are said to ***conflict*** if at least one of them is a write

- There are 3 anomalies that can arise upon interleaving actions of different transactions (say, T1 and T2):
  - Write-Read (WR) Conflict: T2 reads a data object previously written by T1
  - Read-Write (RW) Conflict: T2 writes a data object previously read by T1
  - Write-Write (WW) Conflict: T2 writes a data object previously written by T1

Reading Uncommitted Data: WR Conflicts

- WR conflicts arise when transaction T2 reads a data object A that has been modified by another transaction T1, *which has not yet committed*
  - Such a read is called a dirty read

- Assume T1 and T2 such that:
  - T1 transfers $100 from A's account to B's account
  - T2 credits accounts A and B with a 6% interest payment

```
T1:  BEGIN   A=A-100,   B=B +100   END
T2:  BEGIN   A=1.06*A,   B=1.06*B   END
```

Reading Uncommitted Data: WR Conflicts

- Suppose that T1 and T2 actions are *interleaved* as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B
  - T1 credits $100 to account B

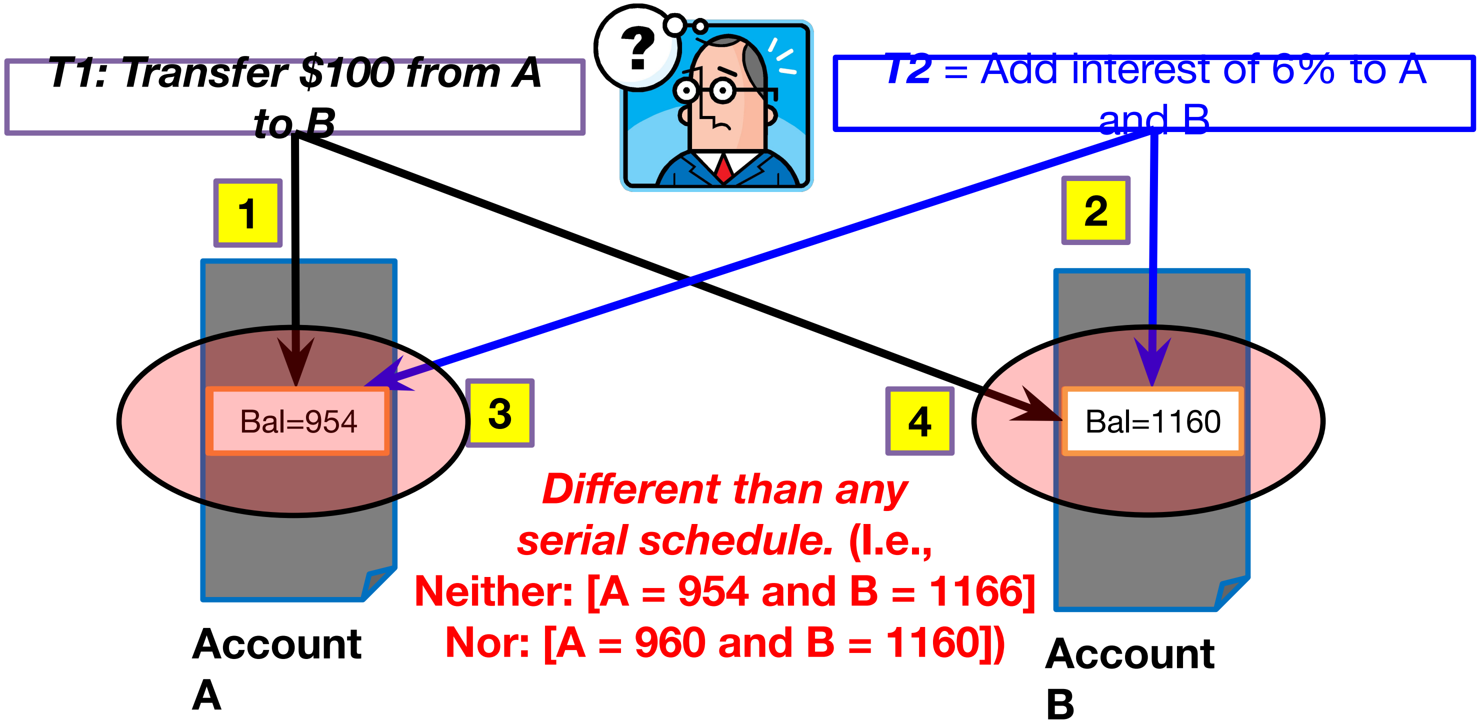
T1: Transfer $100 from A to B

T2 = Add interest of 6% to A and B

Bal=1000
Account A
Bal=1000
Account B

Reading Uncommitted Data: WR Conflicts

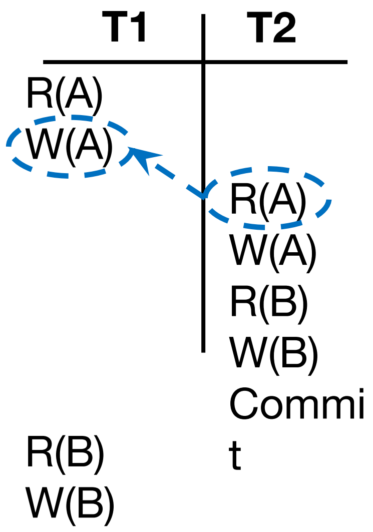
- Suppose that T1 and T2 actions are *interleaved* as follows:
  - T1 deducts $100 from account A **1**
  - T2 adds 6% interest to accounts A and B **2 and 3**
  - T1 credits $100 to account B **4**


T1: Transfer $100 from A to B
T2 = Add interest of 6% to A and B
?
1
2
3
4
Bal=954
Bal=1160
Account A
Account B

Different than any serial schedule. **(I.e., Neither: [A = 954 and B = 1166] Nor: [A = 960 and B = 1160])**

Reading Uncommitted Data: WR Conflicts

- T1 and T2 can be represented by the following schedule:

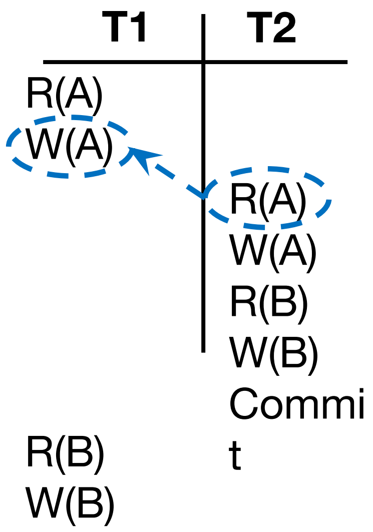


The value of A written by T1 is read by T2 before T1 has completed all its changes!

Why is this a problem?

Reading Uncommitted Data: WR Conflicts

- T1 and T2 can be represented by the following schedule:



T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
R(B)	
W(B)	

The value of A written by T1 is read by T2 before T1 has completed all its changes!

Why is this a problem?

- T1 may write some value into A that makes the database inconsistent
- As long as T1 overwrites this value with a ‘correct’ value of A before committing, no harm is done if T1 and T2 are run in some serial order (this is because T2 would then not see the <u>temporary</u> inconsistency)

Reading Uncommitted Data: WR Conflicts

- T1 and T2 can be represented by the following schedule:

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
R(B)	
W(B)	
Commit	

The value of A written by T1 is read by T2 before T1 has completed all its changes!

Why is this a problem?

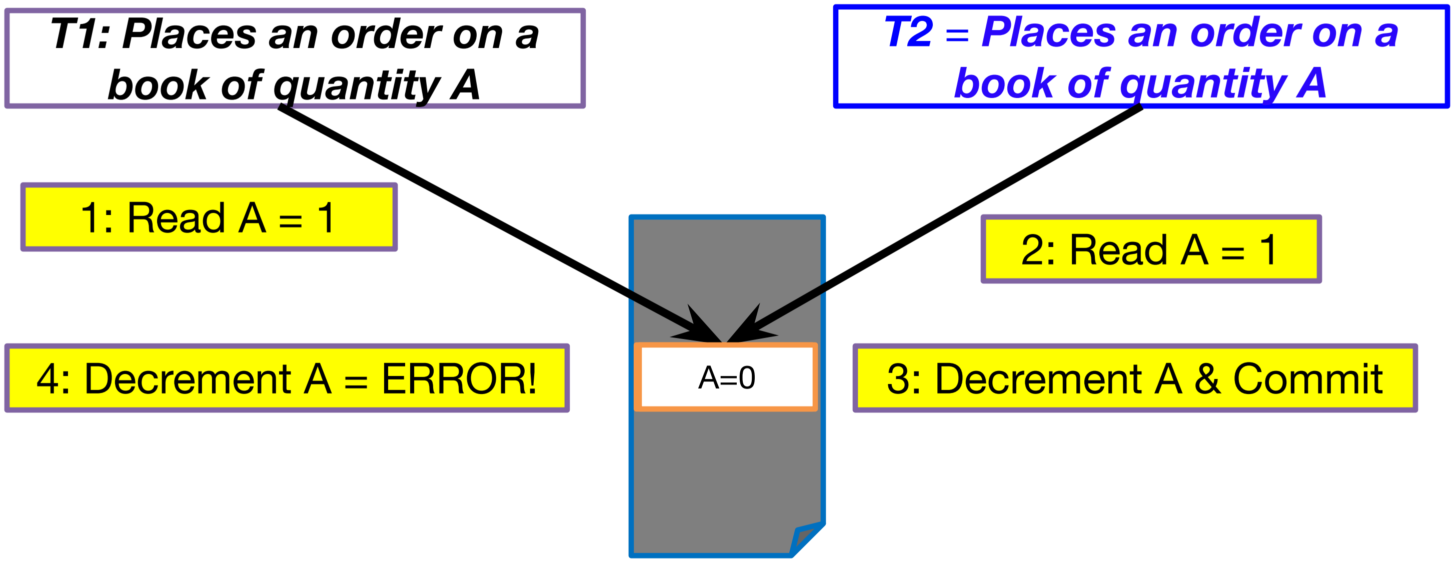
Note that although a transaction must leave a database in a consistent state *after* it completes, it is not required to keep the database consistent while it is still in progress!

Unrepeatable Reads: RW Conflicts

- RW conflicts arise when transaction T2 writes a data object A that has been read by another transaction T1, *while T1 is still in progress*
- If T1 tries to read A again, it will get a different result!
  - Such a read is called an unrepeatable read
- Assume A is the number of available copies for a book
  - A transaction that places an order on the book reads A, checks that A > 0 and decrements A
  - Assume two transactions, T1 and T2

Unrepeatable Reads: RW Conflicts

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 reads A
  - T2 reads A, decrements A and commit
  - T1 tries to decrement A



T1: Places an order on a book of quantity A
T2 = Places an order on a book of quantity A
1: Read A = 1
2: Read A = 1
A=0
4: Decrement A = ERROR!
3: Decrement A & Commit

This situation will never arise in a serial execution of T1 and T2; T2 would read A and see 0 and therefore not proceed with placing an order!

Overwriting Uncommitted Data: WW Conflicts

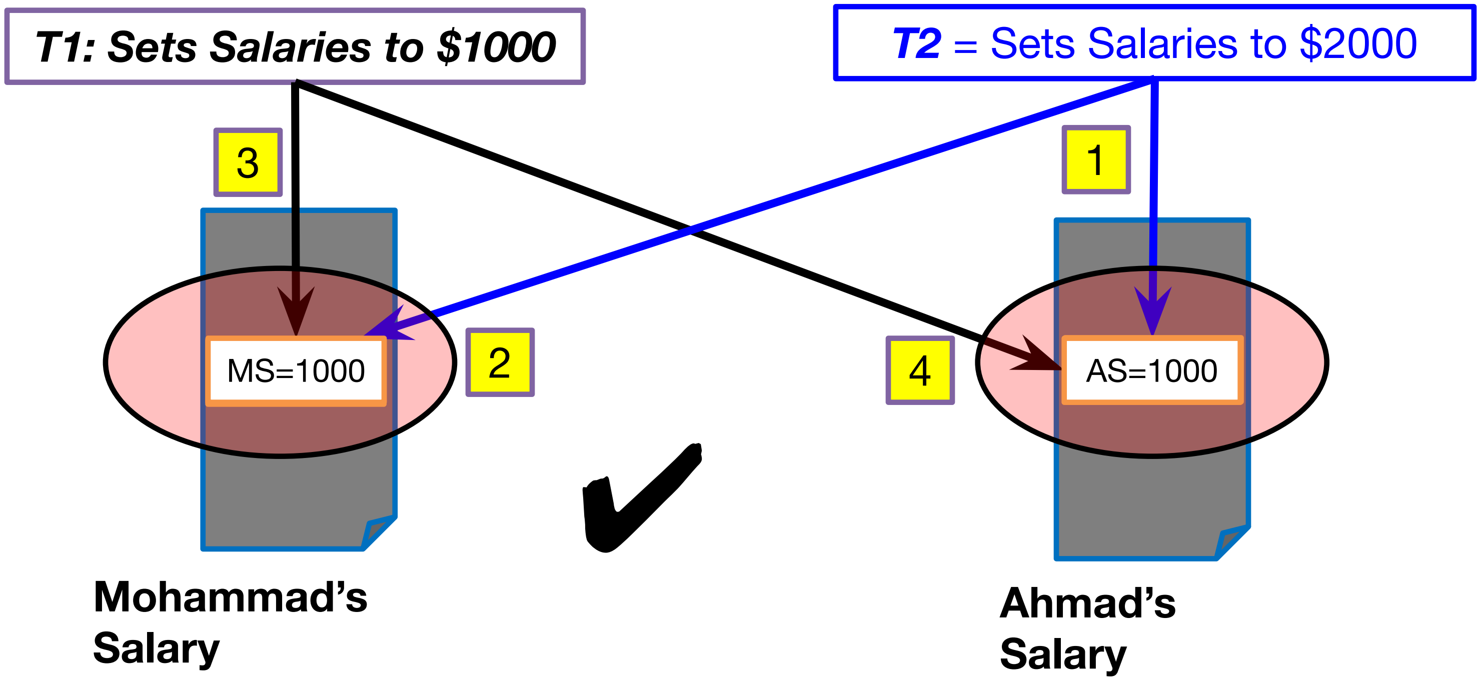
- WW conflicts arise when transaction T2 writes a data object A that has been written by another transaction T1, *while T1 is still in progress*

- Suppose that Mohammad and Ahmad are two employees and their salaries <u>*must be kept equal*</u>

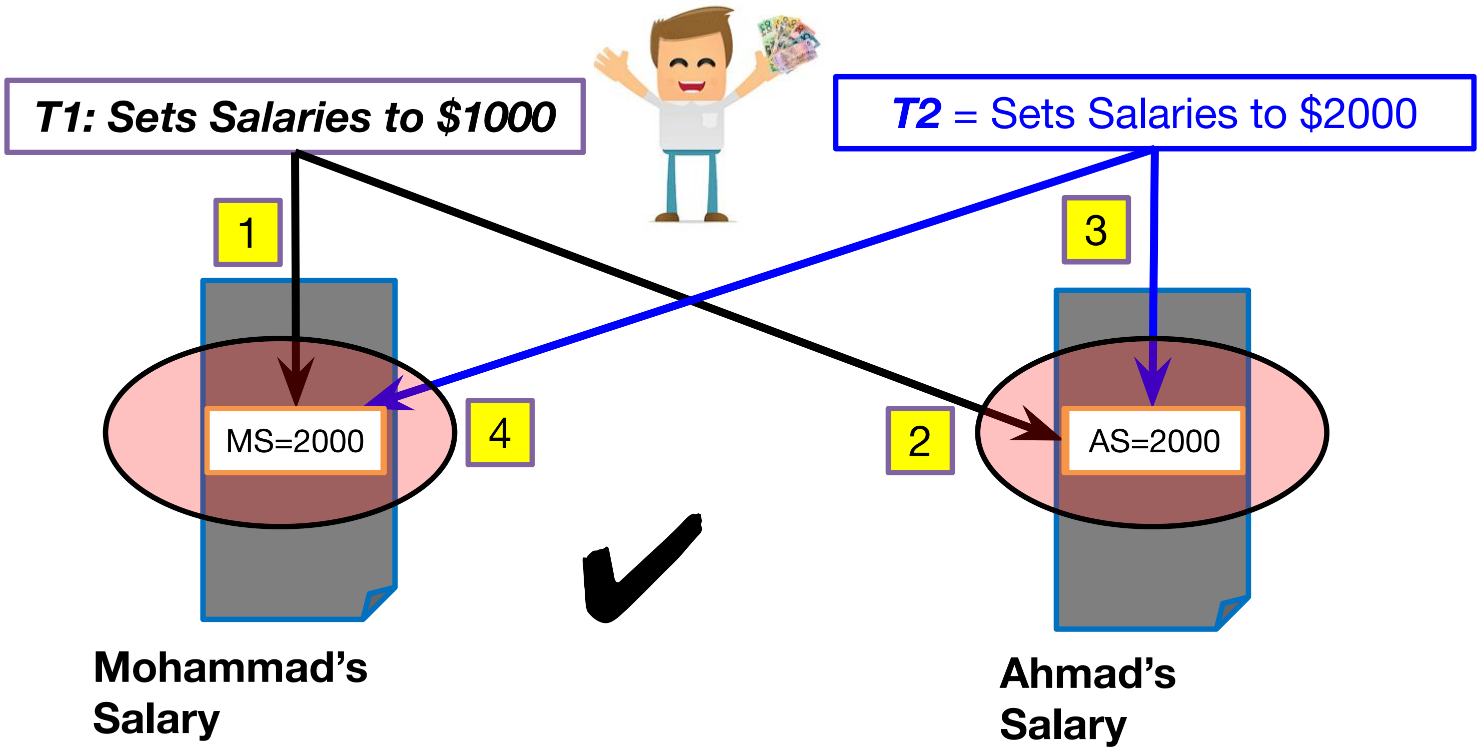
- Assume T1 sets Mohammad’s and Ahmad’s salaries to $1000

- Assume T2 sets Mohammad’s and Ahmad’s salaries to $2000

Overwriting Uncommitted Data: WW Conflicts

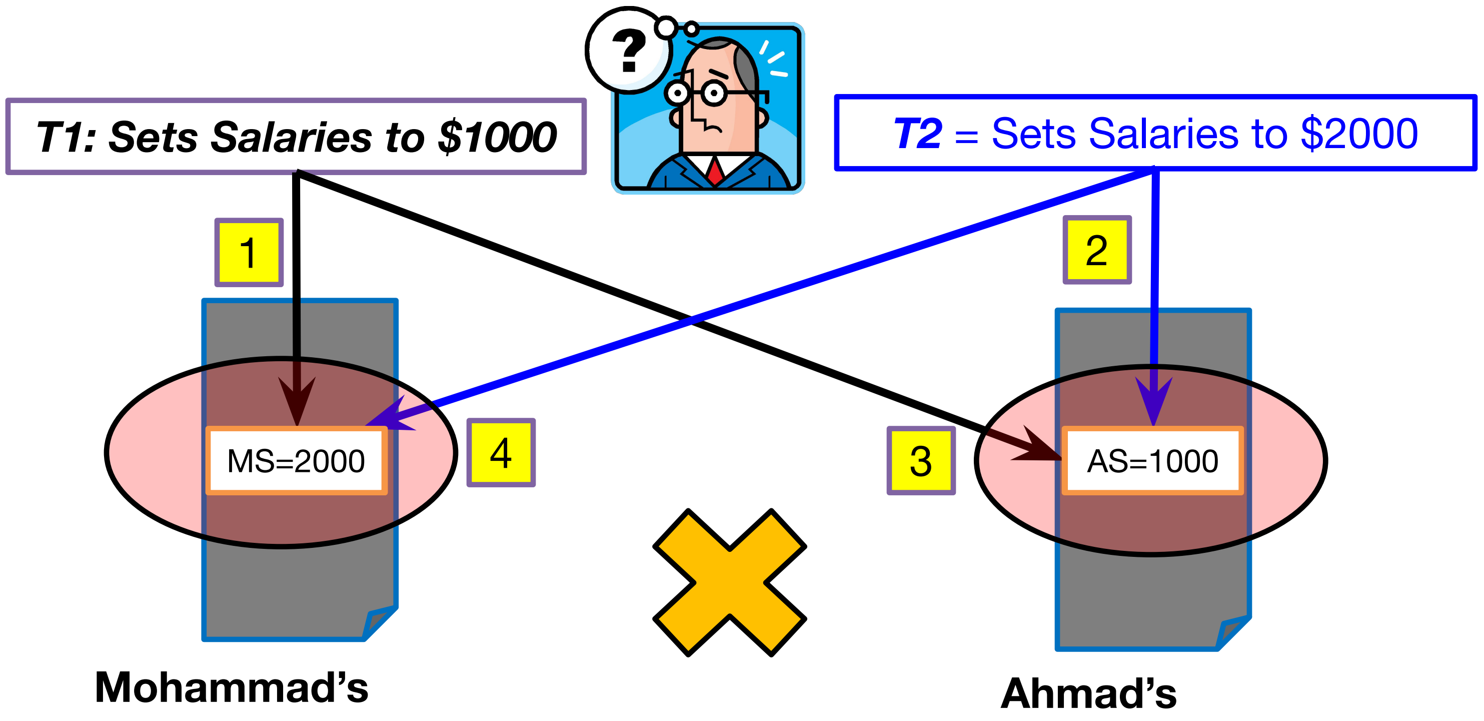


Overwriting Uncommitted Data: WW Conflicts



Either serial schedule is acceptable from a *consistency standpoint* (although Mohammad and Ahmad may prefer higher salaries!)

Overwriting Uncommitted Data: WW Conflicts



The problem is that we have a ***lost update***. In particular, T2 overwrote Mohammad's Salary as set by T1 (this will never happen with a serializable schedule!)

Neither T1 nor T2 reads a salary value before writing it- such a write is called a ***blind write!***

Conflicting Instructions

- Instructions I_i and I_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.
  1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict.
  2. I_i = **read**(Q), I_j = **write**(Q). They conflict.
  3. I_i = **write**(Q), I_j = **read**(Q). They conflict
  4. I_i = **write**(Q), I_j = **write**(Q). They conflict
- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them.
- If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

- If a schedule *S* can be transformed into a schedule *S'* by a series of swaps of non-conflicting instructions, we say that *S* and *S'* are **conflict equivalent**.
- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

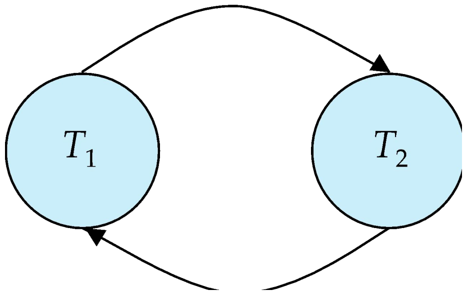
T_3	T_4
read (Q)	
	write (Q)
write (Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

Testing for Serializability

- Consider some schedule of a set of transactions $T_1, T_2, ..., T_n$
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- Example of a precedence graph of the following schedule

T_1	T_2
read (A) $A := A - 50$	
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B)
write (A) read (B) $B := B + 50$ write (B) commit	
	$B := B + temp$ write (B) commit



This schedule is not conflict serializable since it has a cycle.

A schedule is conflict serializable if and only if its precedence graph is acyclic.

S

T1	T2
R(A)	
	W(A)
W(A)	
	W(A)
	Commit
R(A)	
Commit	

(Not conflict serializable)

S

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	Commit
R(A)	
Abort	

Dirty Read (W-R Conflict)

S

T1	T2
R(A)	
	R(A)
	W(A)
R(A)	

Unrepeatable Read (R-W Conflict)

S

T1	T2
W(A)	
	W(A)
	W(B)
	Commit
W(B)	
Commit	

Overwrite (W-W conflict)

Is this schedule conflict serializable? If yes, what would be its equivalent serial schedule?

S: R1(A), R3(B), R3(A), R2(B), R2(C), W3(B), W2(C), R1(C), W1(A), W1(C)

Is this schedule conflict serializable? If yes, what would be its equivalent serial schedule?

S: R1(A), W2(A), W1(A), W3(A), R1(A), W1(A), W2(A), W3(A)

Other schedule equivalence

- View equivalence is based purely on **reads** and **writes** alone.
- View Serializability Is not used in practice due to its high degree of computational complexity (NP-complete problem).

Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i appears before the commit operation of T_j.
- The following schedule is not recoverable

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

- If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state (so *read(A)* in T_9 is called a *dirty read*). Hence, database must ensure that schedules are recoverable.
- When there is NO *dirty read,* the schedule is recoverable*.*

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

  If T_{10} fails, T_{11} and T_{12} must also be rolled back.
- Can lead to the undoing of a significant amount of work
- When there is NO *dirty read*, it avoids cascading rollback.

Cascadeless Schedules

- **Cascadeless schedules**
  - For each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every Cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- **Goal** – to develop concurrency control protocols that will assure serializability.

Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .
- Concurrency control protocols (generally) do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids non-serializable schedules.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g., a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g., database statistics computed for query optimization can be approximate
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read.
  - Repeated reads of same record must return same value.
  - No other transaction is allowed to update the data between these two reads.
  - However, a transaction may not be serializable w.r.t. other transactions – it may find some records inserted by a transaction but not find other data inserted by the same transaction.
- **Read committed** — only committed records can be read.
  - Successive reads of record may return different (but committed) values.
  - That is, between two teads of a data item by the transaction, another transaction may have updated the data item and committed.
- **Read uncommitted** — even uncommitted records may be read.

All these above isolation levels disallow 'lost update' (that is, dirty writes).

Implementation of Isolation Levels

- Locking
  - Lock on whole database vs lock on items
  - How long to hold lock?
  - Shared vs exclusive locks
- Timestamps
  - Transaction timestamp assigned e.g. when a transaction begins
  - Data items store two timestamps
    - Read timestamp
    - Write timestamp
  - Timestamps are used to detect out of order accesses
- Multiple versions of each data item
  - Allow transactions to read from a "snapshot" of the database

Transactions as SQL Statements

- E.g., Transaction 1:
  select *ID, name* **from** *instructor* **where** *salary* > 90000
- E.g., Transaction 2:
  insert into *instructor* **values** ('11111', 'James', 'Marketing', 100000)
- The result of the query (T1) will be different depending on whether T2 comes before or after T1.
- In concurrent execution of these T1 and T2, conflict occurs
  - T1 starts, finds tuples salary > 90000 using index and locks them, and then T2 executes.
  - Instance of the **phantom phenomenon**
- Let's consider T3 below, with Wu's salary = 90000
  update *instructor*
  set *salary* = *salary* * 1.1
  where *name* = 'Wu'
- Key idea: Detect "**predicate**" conflicts, and use some form of "**predicate locking**"

End of Chapter 17