TRANSACTION MANAGEMENT

Transaction

A transaction is a unit of a program execution that accesses and possibly modifies various data objects (tuples, relations).

A transaction is a Logical unit of database processing that includes one or more access operations (read -retrieval, write - insert or update, delete).

A transaction (set of operations) may be stand-alone specified in a high level language like SQL submitted interactively, or may be embedded within a program.

A transaction (collection of actions) makes transformations of system states while preserving the database consistency.

A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.

A transaction is the DBMS's abstract view of a user program: a sequence of reads and writes.

PROPOERTIES OF TRANSACTION

The DBMS need to ensure the following properties of transactions:

1. Atomicity

– Transactions are either done or not done

– They are never left partially executed

An executing transaction completes in its entirety or it is aborted altogether.

–e.g., Transfer_Money (Amount, X, Y) means i) DEBIT (Amount, X);
ii) CREDIT (Amount, Y). Either both take place or none

2. Consistency

– Transactions should leave the database in a consistent state

If each Transaction is consistent, and the DB starts consistent, then the Database ends up consistent.

–If a transaction violates the database's consistency rules, the entire transaction will be rolled back and the database will be restored to a state consistent with those rules.

3. Isolation

– Transactions must behave as if they were executed in isolation.

An executing transaction cannot reveal its (incomplete) results before it commits.

–Consequently, the net effect is identical to executing all transactions, the one after the other in some serial order.

4. Durability

– Effects of completed transactions are resilient against failures

Once a transaction commits, the system must guarantee that the results of its operations will never be lost, in spite of subsequent failures.

SIMPLE MODEL OF A DATABASE

A database is a collection of named data items.

Granularity of data - a field, a record, or a whole disk block (Concepts are independent of granularity).

Basic operations are read and write:

read_item(X): Reads a database item named X into a program variable. To simplify our notation, we assume that the program variable is also named X.

write_item(X): Writes the value of program variable X into the database item named X.

READ AND WRITE OPERATIONS:

Basic unit of data transfer from the disk to the computer main memory is one block. In general, a data item (what is read or written) will be the field of some record in the database, although it may be a larger unit such as a record or even a whole block.

read_item(X) command includes the following steps:

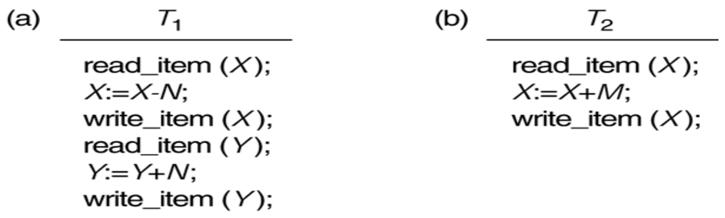
1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).

3. Copy item X from the buffer to the program variable named X.

write_item(X) command includes the following steps:

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the program variable named X into its correct location in the buffer.
4. Store the updated block from the buffer back to disk (either immediately or at some later point in time).

Two sample transactions



```
(a)        T1                    (b)        T2
     read_item (X);                   read_item (X);
     X:=X-N;                          X:=X+M;
     write_item (X);                  write_item (X);
     read_item (Y);
     Y:=Y+N;
     write_item (Y);
```

Transaction Example in MySQL

START TRANSACTION;

SELECT@A:=SUM(salary) FROMtable1 WHEREtype=1;

UPDATEtable2 SETsummary=@A WHEREtype=1;

COMMIT;

Transaction Example in Oracle(same with SQL Server)

•When you connect to the database with sqlplus(Oracle command-line utility that runs SQL and PL/SQL commands interactively or from a script) a transaction begins.

•Once the transaction begins, every SQL DML (Data Manipulation Language) statement you issue subsequently becomes a part of this transaction

TRANSACTION STATES

1. Active state
2. Partially committed state
3. Committed state
4. Failed state
5. Terminated State

State transition diagram illustrating the states for transaction execution:

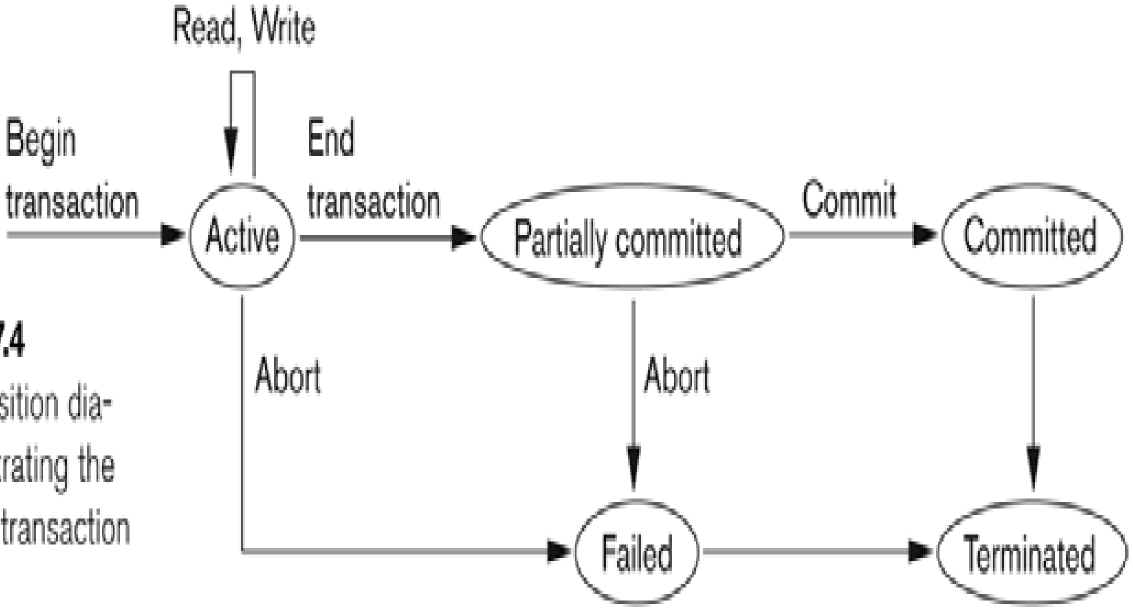
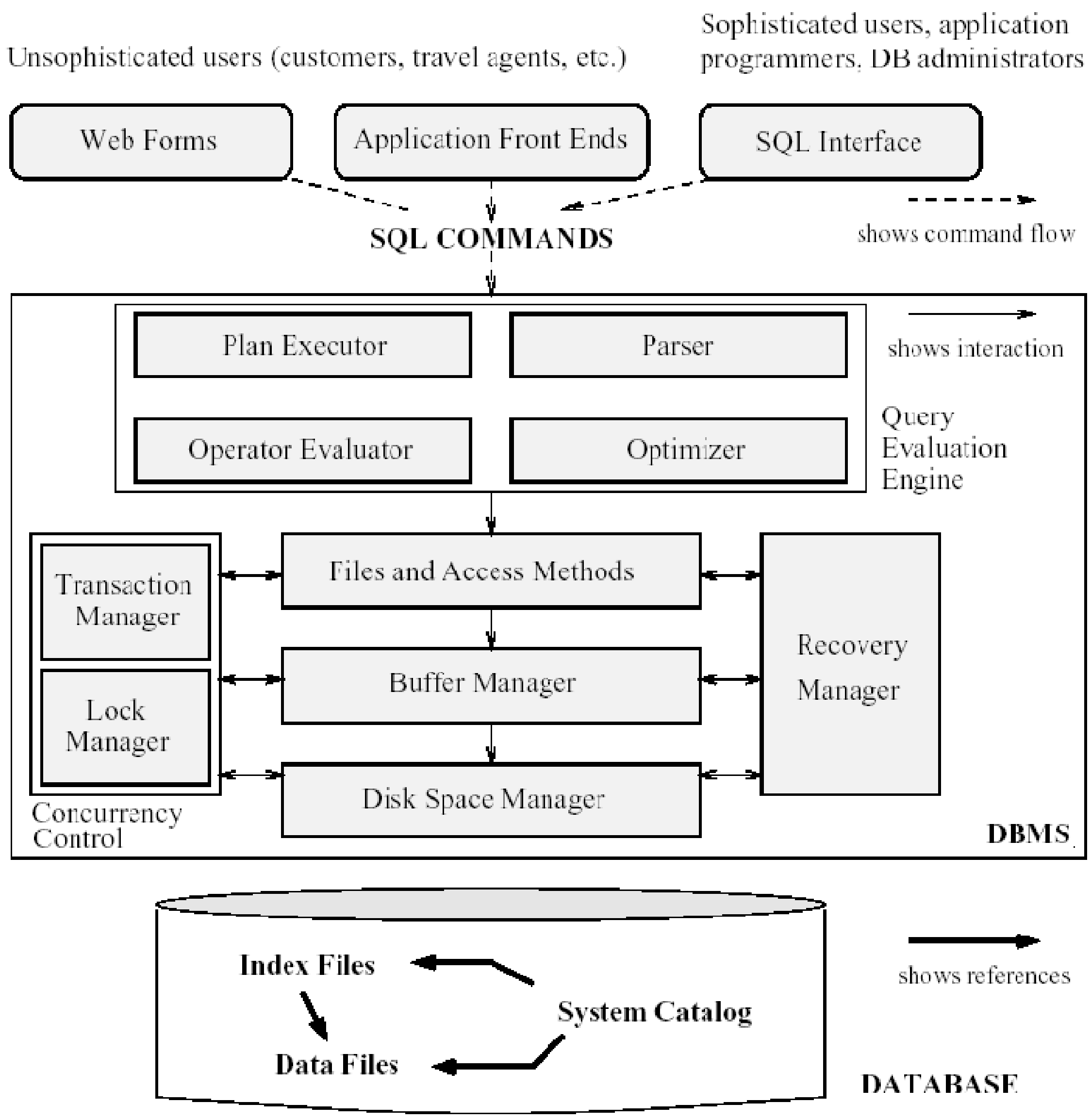


Figure 17.4
State transition diagram illustrating the states for transaction execution.

Transaction Processing System



CONCURRENCY CONTROL

Concurrency in a DBMS

Concurrent execution of user programs is essential for good DBMS performance.
Because disk accesses are frequent, and relatively slow, it is important to keep the CPU humming by working on several user programs concurrently.

Users submit transactions, and can think of each transaction as executing by itself.

Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions. Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins. DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements. Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).

Things get even more complicated if we have several DBMS programs (transactions) executed concurrently.

“Synchronization" of transactions; allowing concurrency (instead of insisting on a strict serial transaction execution, i.e., process complete T1, then T2, then T3 etc.)
- increase the throughput of the system,
- minimize response time for each transaction

Why do we need concurrent executions?

–It is essential for good DBMS performance!
Disk accesses are frequent, and relatively slow.
Overlapping I/O with CPU activity increases throughput and response time.

What is the problem with concurrent transactions?

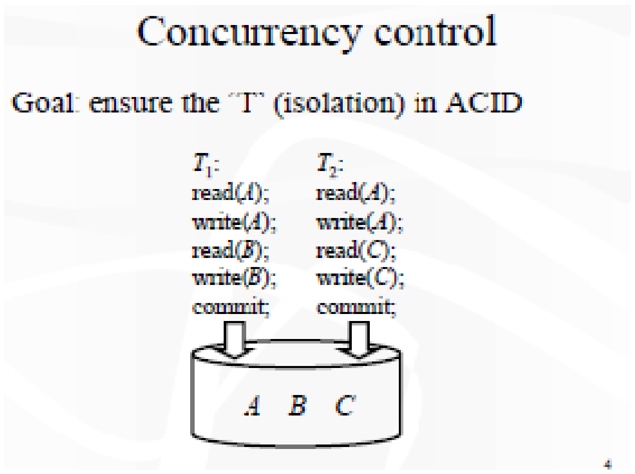
–Interleaving transactions might lead the system to an inconsistent state (like previous example):

Scenario: A Xact prints the monthly bank account statement for a user U (one bank transaction at-a-time).Before finalizing the report another Xact withdraws $X from user U.

Result: Although the report contains an updated final balance, it shows nowhere the bank transaction that caused the decrease (unrepeatable read problem, explained next)

A DBMS guarantees that these problems will not arise.

–Users are given the impression that the transactions are executed sequentially, the one after the other.



Concurrency control

Goal: ensure the "I" (isolation) in ACID

T_1:	T_2:
read(A);	read(A);
write(A);	write(A);
read(B);	read(C);
write(B);	write(C);
commit;	commit;

Why Concurrency Control is needed?

Problems that can occur for certain transaction schedules without appropriate concurrency control mechanisms:

The Lost Update Problem

This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.

The Temporary Update (or Dirty Read) Problem

This occurs when one transaction updates a database item and then the transaction fails for some reason.
The updated item is accessed by another transaction before it is changed back to its original value.

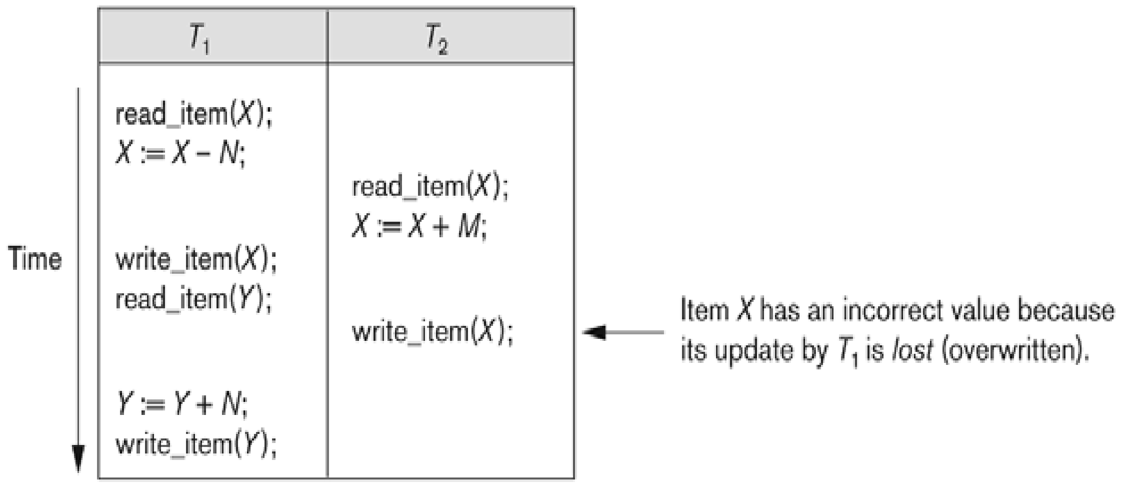
The Incorrect Summary Problem

If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

a) **The Lost Update Problem**

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(a)



T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;

write_item(X);
read_item(Y);	

	write_item(X);
$Y := Y + N$; write_item(Y);	

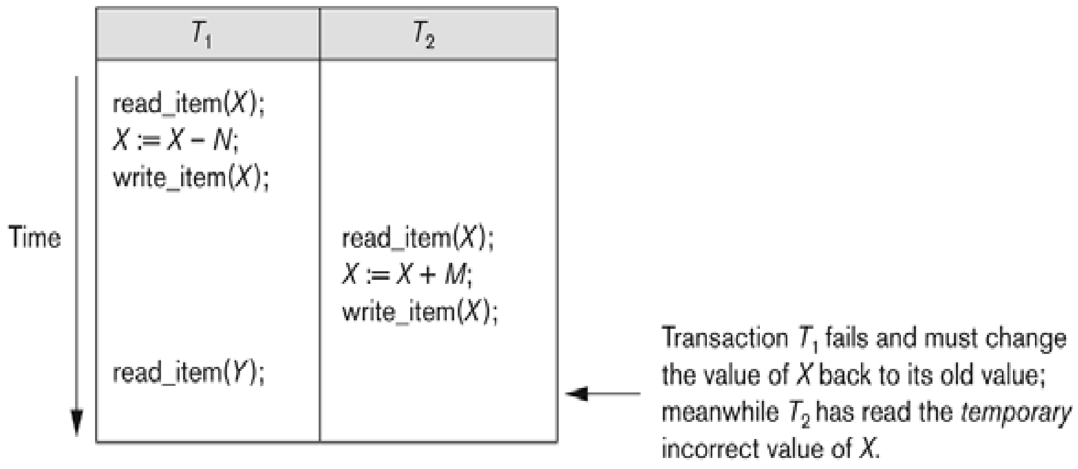
Time
Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

The update performed by T1 gets lost; possible solution: T1 locks/unlocks database object X =) T2 cannot read X while X is modified by T1

b) The temporary update problem

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(b)



T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);	

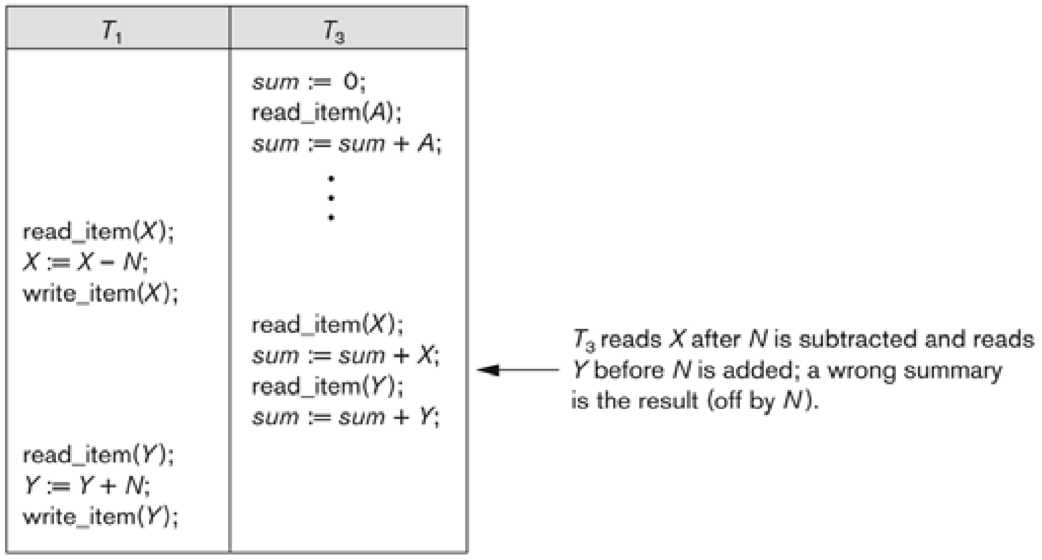
Time
Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

T1 modifies db object, and then the transactionT1 fails for some reason. Meanwhile the modified db object, however, has been accessed by another transaction T2. Thus T2 has read data that never existed.

c) The incorrect summary problem

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(c)



T_1	T_3

	sum := 0;
	read_item(*A*);
	sum := *sum* + *A*;
	⋮

read_item(*X*);
X := *X* − *N*;
write_item(*X*);	

	read_item(*X*);
	sum := *sum* + *X*;
	read_item(*Y*);
	sum := *sum* + *Y*;

read_item(*Y*);
Y := *Y* + *N*;
write_item(*Y*);	

T_3 reads *X* after *N* is subtracted and reads *Y* before *N* is added; a wrong summary is the result (off by *N*).

In this schedule, the total computed by T1 is wrong.

=) T1 must lock/unlock several db objects.

The following are the Problems that arise when interleaving Transactions (and they are already discussed above but the terminology is different)**:**

Problem 1: Reading Uncommitted Data (WR Conflicts)

Reading the value of an uncommitted object might yield an inconsistency
–Dirty Reads or Write-then-Read (WR) Conflicts.

Problem 2: Unrepeatable Reads (RW Conflicts)

Reading the same object twice might yield an inconsistency
–Read-then-Write (RW) Conflicts (ή Write-After-Read)

Problem 3: Overwriting Uncommitted Data (WW Conflicts)

Overwriting an uncommitted object might yield an inconsistency
–Lost Update or Write-After-Write (WW) Conflicts.

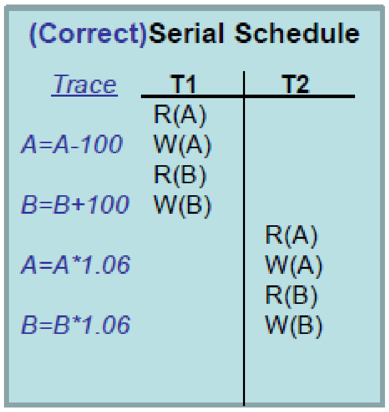
Remark: There is no notion of RR-Conflicts no object is changed

1. Reading Uncommitted Data (WR Conflicts)

To illustrate the WR-conflict consider the following problem:

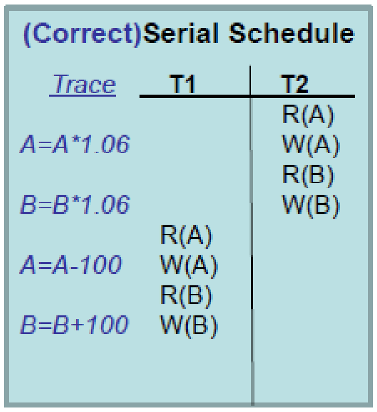
T1: Transfer $100 from Account A to Account B

T2: Add the annual interest of 6% to both A and B.



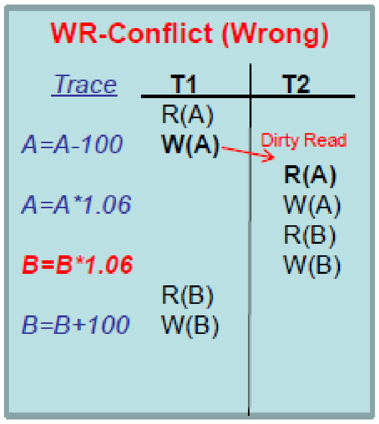
(Correct)Serial Schedule

Trace	T1	T2
	R(A)	
A=A-100	W(A)	
	R(B)	
B=B+100	W(B)	
		R(A)
*A=A*1.06*		W(A)
		R(B)
*B=B*1.06*		W(B)



(Correct)Serial Schedule

Trace	T1	T2
		R(A)
*A=A*1.06*		W(A)
		R(B)
*B=B*1.06*		W(B)
	R(A)	
A=A-100	W(A)	
	R(B)	
B=B+100	W(B)	



WR-Conflict (Wrong)

Trace	T1	T2
	R(A)	
A=A-100	**W(A)**	Dirty Read
		R(A)
*A=A*1.06*		W(A)
		R(B)
B=B*1.06		W(B)
	R(B)	
B=B+100	W(B)	

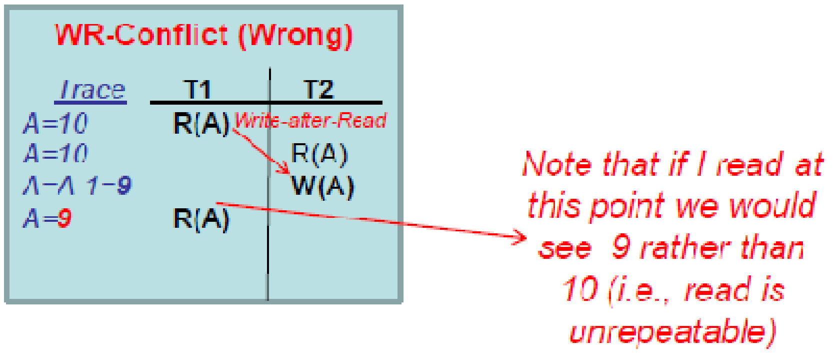
Problem caused by the WR-Conflict? Account B was credited with the interest on a smaller amount (i.e., 100$ less), thus the result is not equivalent to the serial schedule.

2. Unrepeatable Reads (RW Conflicts)

To illustrate the RW-conflict, consider the following problem:

T1: Print Value of A

T2: Decrease Global counter A by 1.


WR-Conflict (Wrong)

Trace	T1	T2
A=10	R(A)	Write-after-Read
A=10		R(A)
A=A-1=9		W(A)
A=9	R(A)	

Note that if I read at this point we would see 9 rather than 10 (i.e., read is unrepeatable)

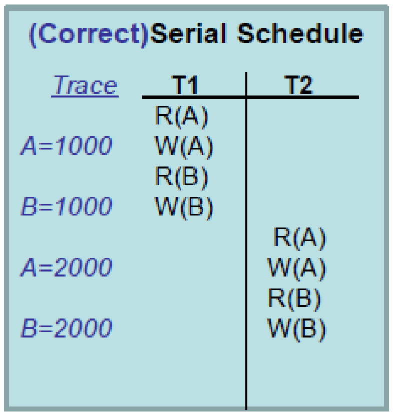
Problem caused by the RW-Conflict? Although the "A" counter is read twice within T1 (without any intermediate change) it has two different values (unrepeatable read)! what happens if T2 aborts? T1 has shown an incorrect result.

3. Overwriting Uncommitted Data (WW Conflicts)

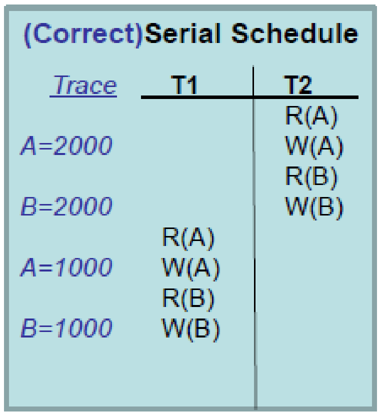
To illustrate the WW-conflict consider the following problem:

Salary of employees A and B must be kept equal

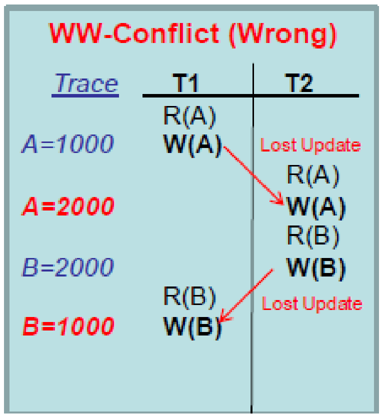
T1: Set Salary to 1000; T2: Set Salary equal to 2000


(Correct)Serial Schedule

Trace	T1	T2
	R(A)	
A=1000	W(A)	
	R(B)	
B=1000	W(B)	
		R(A)
A=2000		W(A)
		R(B)
B=2000		W(B)


(Correct)Serial Schedule

Trace	T1	T2
		R(A)
A=2000		W(A)
		R(B)
B=2000		W(B)
	R(A)	
A=1000	W(A)	
	R(B)	
B=1000	W(B)	


WW-Conflict (Wrong)

Trace	T1	T2
	R(A)	
A=1000	W(A)	Lost Update
		R(A)
A=2000		W(A)
		R(B)
B=2000		W(B)
	R(B)	Lost Update
B=1000	W(B)	

Problem caused by the WW-Conflict?

Employee "A" gets a salary of 2000 while employee "B" gets a salary of 1000, thus result is not equivalent to the serial schedule!

Summary of Conflicts

1. WR Conflict (dirty read): A transaction T2 could read a database object A that has been modified by another transaction T1, which has not yet committed.

2. RW Conflict (unrepeatable read): A transaction T2could change the value of an object A that has been read by a transaction T1, while T1is still in progress.

3. WW Conflict (lost update): A transaction T2 could overwrite the value of an object A, which has already been modified by a transaction T1, while T1is still in progress.

Why **recovery is needed:** (What causes a Transaction to fail)

1. **A computer failure (system crash):**

A hardware or software error occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer's internal memory may be lost.

2. **A transaction or system error:**

Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.

3. **Local errors or exception conditions detected by the transaction:**

Certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled. A programmed abort in the transaction causes it to fail.

4. **Concurrency control enforcement:**

The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock.

5. **Disk failure:**

Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

6. **Physical problems and catastrophes:**

This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Recovery manager keeps track of the following operations:

begin_transaction: This marks the beginning of transaction execution.

read or write: These specify read or write operations on the database items that are executed as part of a transaction.

end_transaction: This specifies that read and write transaction operations have ended and marks the end limit of transaction execution.

At this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database or whether the transaction has to be aborted because it violates concurrency control or for some other reason.

commit_transaction: This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.

rollback (or abort): This signals that the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.

Recovery techniques use the following operators:

undo: Similar to rollback except that it applies to a single operation rather than to a whole transaction.

redo: This specifies that certain *transaction* *operations must be redone to ensure that all the* operations of a committed transaction have been applied successfully to the database.

The System Log

Log or Journal: The log keeps track of all transaction operations that affect the values of database items.
This information may be needed to permit recovery from transaction failures.
The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure.
In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures.
T in the following discussion refers to a unique **transaction-id** that is generated automatically by the system and is used to identify each transaction:

The following actions are recorded in the log:
_ Ti writes an object: the old value and the new value.
Log record must go to disk before the changed page!
_ Ti commits/aborts: a log record indicating this action.

Log records are chained together by Xact id, so it's easy to undo a specific Xact.

Log is often duplexed and archived on stable storage.

All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.

Types of log record:

[start_transaction,T]: Records that transaction T has started execution.

[write_item,T,X,old_value,new_value]: Records that transaction T has changed the value of database item X from old_value to new_value.

[read_item,T,X]: Records that transaction T has read the value of database item X.

[commit,T]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.

[abort,T]: Records that transaction T has been aborted.

Protocols for recovery that *avoid cascading rollbacks do not require that read operations be written to the system log, whereas other protocols*require these entries for recovery.

Strict protocols require simpler write entries that do not include new_value

Recovery using log records:

If the system crashes, we can recover to a consistent database state by examining the log.

1. Because the log contains a record of every write operation that changes the value of some database item, it is possible to **undo the effect of these write operations of a transaction T** by tracing backward through the log and resetting all items changed by a write operation of T to their old_values.
2. We can also **redo the effect of the write operations of a** transaction T by tracing forward through the log and setting all items changed by a write operation of T (that did not get done permanently) to their new_values.

Commit Point of a Transaction:

Definition a Commit Point:

A transaction T reaches its **commit point when all its** operations that access the database have been executed successfully *and the effect of all the transaction operations on* the database has been recorded in the log.

Beyond the commit point, the transaction is said to be committed, and its effect is assumed to be permanently recorded in the database.

The transaction then writes an entry [commit,T] into the log.

Roll Back of transactions:

Needed for transactions that have a [start_transaction,T] entry into the log but no commit entry [commit,T] into the log.

Redoing transactions:

Transactions that have written their commit entry in the log must also have recorded all their write operations in the log; otherwise they would not be committed, so their effect on the database can be redone from the log entries. (Notice that the log file must be kept on disk. At the time of a system crash, only the log entries that have been written back to disk are considered in the recovery process because the contents of main memory may be lost.)

Force writing a log:

Before a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called force-writing the log file before committing a transaction.

Characterizing Schedules based on Recoverability

Transaction schedule or history:

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

A **schedule (or history) S of n transactions T1, T2, …,** Tn:

It is an ordering of the operations of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of T1 in S must appear in the same order in which they occur in T1.

Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in S.

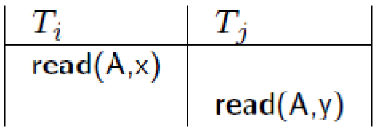
Serializability:

DBMS must control concurrent execution of transactions to ensure read consistency, i.e., to avoid dirty reads etc.

A (possibly concurrent) schedule S is serializable if it is equivalent to a serial schedule S0, i.e., S has the same result database state as S0.

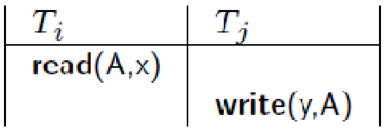
How to ensure serializability of concurrent transactions?

Conflicts between operations of two transactions:



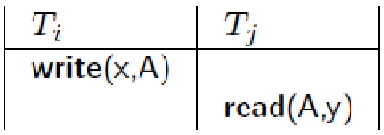
T_i	T_j
read(A,x)	
	read(A,y)

(order does not matter)



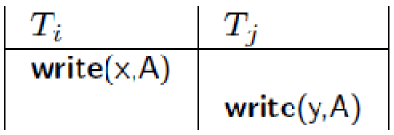
T_i	T_j
read(A,x)	
	write(y,A)

(order matters)



T_i	T_j
write(x,A)	
	read(A,y)

(order matters)



T_i	T_j
write(x,A)	
	write(y,A)

(order matters)

A schedule S is serializable with regard to the above conflicts iff S can be transformed into a serial schedule S' by a series of swaps of non-conflicting operations.

Checks for serializability are based on precedence graph that describes dependencies among concurrent transactions; if the graph has no cycle, and then the transactions are serializable.
- they can be executed concurrently without affecting each other's transaction result.

Atomicity of Transactions

A transaction might commit after completing all its actions, or it could abort (or be aborted by the DBMS) after executing some actions.

A very important property guaranteed by the DBMS for all transactions is that they are atomic. That is, a user can think of a Xact as always executing all its actions in one step, or not executing any actions at all.

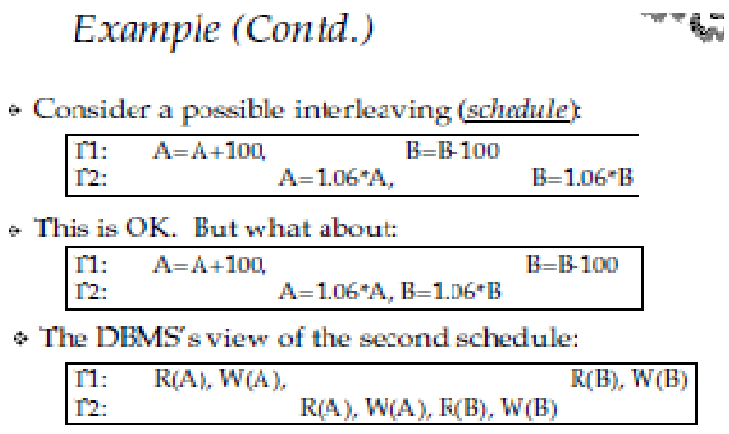
_ DBMS logs all actions so that it can undo the actions of aborted transactions.

Example

Consider two transactions (Xacts):
T1: BEGIN A=A+100, B=B-100 END
T2: BEGIN A=1.06*A, B=1.06*B END

Intuitively, the first transaction is transferring $100 from B's account to A's account. The second is crediting both accounts with a 6% interest payment.

There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect must be equivalent to these two transactions running serially in some order.



Example (Contd.)

- Consider a possible interleaving (*schedule*):

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

- This is OK. But what about:

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

- The DBMS's view of the second schedule:

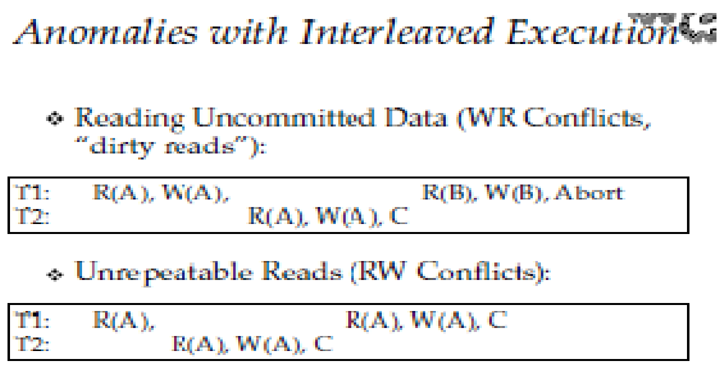
```
T1:   R(A), W(A),                          R(B), W(B)
T2:                R(A), W(A), R(B), W(B)
```

Scheduling Transactions

Serial schedule: Schedule that does not interleave the actions of different transactions.

Equivalent schedules: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.

Serializable schedule: A schedule that is equivalent to some serial execution of the transactions.
(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)



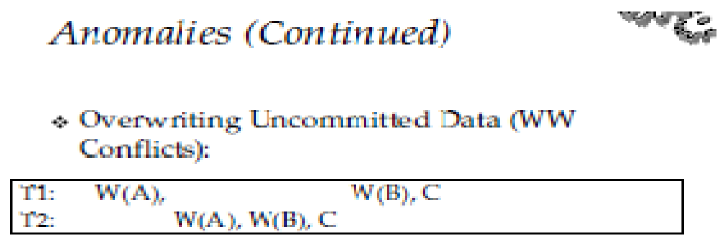
Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, "dirty reads"):

T1:	R(A), W(A),		R(B), W(B), Abort
T2:		R(A), W(A), C	

- Unrepeatable Reads (RW Conflicts):

T1:	R(A),		R(A), W(A), C
T2:		R(A), W(A), C	



Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

T1:	W(A),		W(B), C
T2:		W(A), W(B), C	

Schedules classified on recoverability:

Recoverable schedule:

One where no transaction needs to be rolled back.

A schedule S is recoverable if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.

Cascadeless schedule:

One where every transaction reads only the items that are written by committed transactions.

Schedules requiring cascaded rollback:

A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.

Strict Schedules:

A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.

Serial schedule:

A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule.

Otherwise, the schedule is called nonserial schedule.

Serializable schedule:

A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions

Result equivalent:

Two schedules are called result equivalent if they produce the same final state of the database.

Conflict equivalent:

Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.

Conflict serializable:

A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S'.

Being serializable is not the same as being serial.

Being serializable implies that the schedule is a correct schedule.

It will leave the database in a consistent state.

The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution.

Serializability is hard to check.

Interleaving of operations occurs in an operating system through some scheduler

Difficult to determine beforehand how the operations in a schedule will be interleaved

Practical approach:

Come up with methods (protocols) to ensure serializability.

It's not possible to determine when a schedule begins and when it ends.
Hence, we reduce the problem of checking the whole schedule to checking only a committed project of the schedule (i.e. operations from only the committed transactions.)

Current approach used in most DBMSs:
Use of locks with two phase locking

View equivalence:
A less restrictive definition of equivalence of schedules

View serializability:
Definition of serializability based on view equivalence.
A schedule is view serializable if it is view equivalent to a serial schedule.

Two schedules are said to be view equivalent if the following three conditions hold:
1. The same set of transactions participates in S and S', and S and S' include the same operations of those transactions.
2. For any operation Ri(X) of Ti in S, if the value of X read by the operation has been written by an operation Wj(X) of Tj (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation Ri(X) of Ti in S'.
3. If the operation Wk(Y) of Tk is the last operation to write item Y in S, then Wk(Y) of Tk must also be the last operation to write item Y in S'.

The premise behind view equivalence:
As long as each read operation of a transaction reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same results.
"The view": the read operations are said to see the same view in both schedules

Relationship between view and conflict equivalence:
The two are same under constrained write assumption which assumes that if T writes X, it is constrained by the value of X it read; i.e., new X = f(old X)

Conflict serializability is stricter than view serializability. With unconstrained write (or blind write), a schedule that is view serializable is not necessarily conflict serializable.
Any conflict serializable schedule is also view serializable, but not vice versa.

Relationship between view and conflict equivalence

Consider the following schedule of three transactions
T1: r1(X), w1(X); T2: w2(X); and T3: w3(X):
Schedule Sa: r1(X); w2(X); w1(X); w3(X); c1; c2; c3;
In Sa, the operations w2(X) and w3(X) are blind writes, since T1 and T3 do not read the value of X.
Sa is view serializable, since it is view equivalent to the serial schedule T1, T2, T3.
However, Sa is not conflict serializable, since it is not conflict equivalent to any serial schedule

Testing for conflict serializability: Algorithm 17.1:

1. Looks at only read_Item (X) and write_Item (X) operations
2. Constructs a precedence graph (serialization graph) - a graph with directed edges
3. An edge is created from Ti to Tj if one of the operations in Ti appears before a conflicting operation in Tj
4. The schedule is serializable if and only if the precedence graph has no cycles.

Lock-Based Concurrency Control

Strict Two-phase Locking (Strict 2PL) Protocol:
_ Each Xact must obtain a S (shared) lock on object before reading, and an X (exclusive) lock on object before writing.
_ All locks held by a transaction are released when the transaction completes
_ If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
Strict 2PL allows only serializable schedules.

Aborting a Transaction

If a transaction Ti is aborted, all its actions have to be undone. Not only that, if Tj reads an object last written by Ti, Tj must be aborted as well

Most systems avoid such cascading aborts by releasing a transaction's locks only at commit time.

_ If Ti writes an object, Tj can read this only after Ti commits.

In order to undo the actions of an aborted transaction, the DBMS maintains a log in which every write is recorded. This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up.

Recovering From a Crash

There are 3 phases in the Aries recovery algorithm:

_ Analysis: Scan the log forward (from the most recent checkpoint) to identify all Xacts that were active, and all dirty pages in the buffer pool at the time of the crash.

_ Redo: Redoes all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out and written to disk.

_ Undo: The writes of all Xacts that were active at the crash are undone (by restoring the before value of the update, which is in the log record for the update), working backwards in the log. (Some care must be taken to handle the case of a crash occurring during the recovery process!)

Conflict Serializable Schedules

Two schedules are conflict equivalent if:

_ Involve the same actions of the same transactions

_ Every pair of conflicting actions is ordered the same way

Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

Example

A schedule that is not conflict serializable:

```
T1:    R(A), W(A),                              R(B), W(B)
T2:                    R(A), W(A), R(B), W(B)
```

$$\begin{tikzcd} T1 \arrow[r, bend left, "A"] & T2 \arrow[l, bend left, "B"] \end{tikzcd}$$

Dependency graph

The cycle in the graph reveals the problem.

The output of T1 depends on T2, and viceversa.

Dependency Graph

Dependency graph: One node per Xact; edge from Ti to Tj if Tj reads/writes an object last written by Ti.

Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic

Review: Strict 2PL

Strict Two-phase Locking (Strict 2PL) Protocol:

_ Each Xact must obtain a S (shared) lock on object before reading, and an X (exclusive) lock on object before writing.

_ All locks held by a transaction are released when the transaction completes

_ If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.

Strict 2PL allows only schedules whose precedence graph is acyclic

Two-Phase Locking (2PL)

Two-Phase Locking Protocol

_ Each Xact must obtain a S (shared) lock on object before reading, and an X (exclusive) lock on object before writing.

_ A transaction can not request additional locks once it releases any locks.

_ If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.

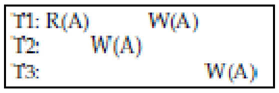
View Serializability

Schedules S1 and S2 are view equivalent if:

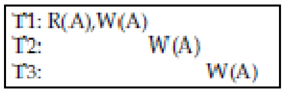
_ If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2

_ If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2

_ If Ti writes final value of A in S1, then Ti also writes final value of A in S2



T1:	R(A)		W(A)	
T2:		W(A)		
T3:				W(A)



T1:	R(A),W(A)		
T2:		W(A)	
T3:			W(A)

Lock Management

Lock and unlock requests are handled by the lock manager

Lock table entry:

_ Number of transactions currently holding a lock

_ Type of lock held (shared or exclusive)

_ Pointer to queue of lock requests

Locking and unlocking have to be atomic operations

Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock

Deadlocks

Deadlock: Cycle of transactions waiting for locks to be released by each other.

Two ways of dealing with deadlocks:

_ Deadlock prevention

_ Deadlock detection

Deadlock Prevention

Assign priorities based on timestamps.

Assume Ti wants a lock that Tj holds. Two policies are possible:

_ Wait-Die: It Ti has higher priority, Ti waits for Tj; otherwise Ti aborts

_ Wound-wait: If Ti has higher priority, Tj aborts; otherwise Ti waits

If a transaction re-starts, make sure it has its original timestamp

Deadlock Detection

Create a waits-for graph:

_ Nodes are transactions

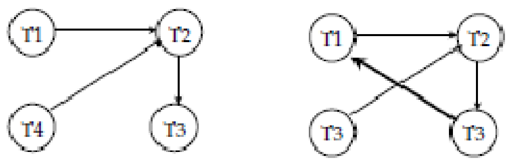
_ There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock

Periodically check for cycles in the waits-for Graph

Deadlock Detection

Example:

T1:	S(A), R(A),		S(B)			
T2:		X(B),W(B)			X(C)	
T3:				S(C), R(C)		X(A)
T4:					X(B)	



Multiple-Granularity Locks

Hard to decide what granularity to lock (tuples vs. pages vs. tables).