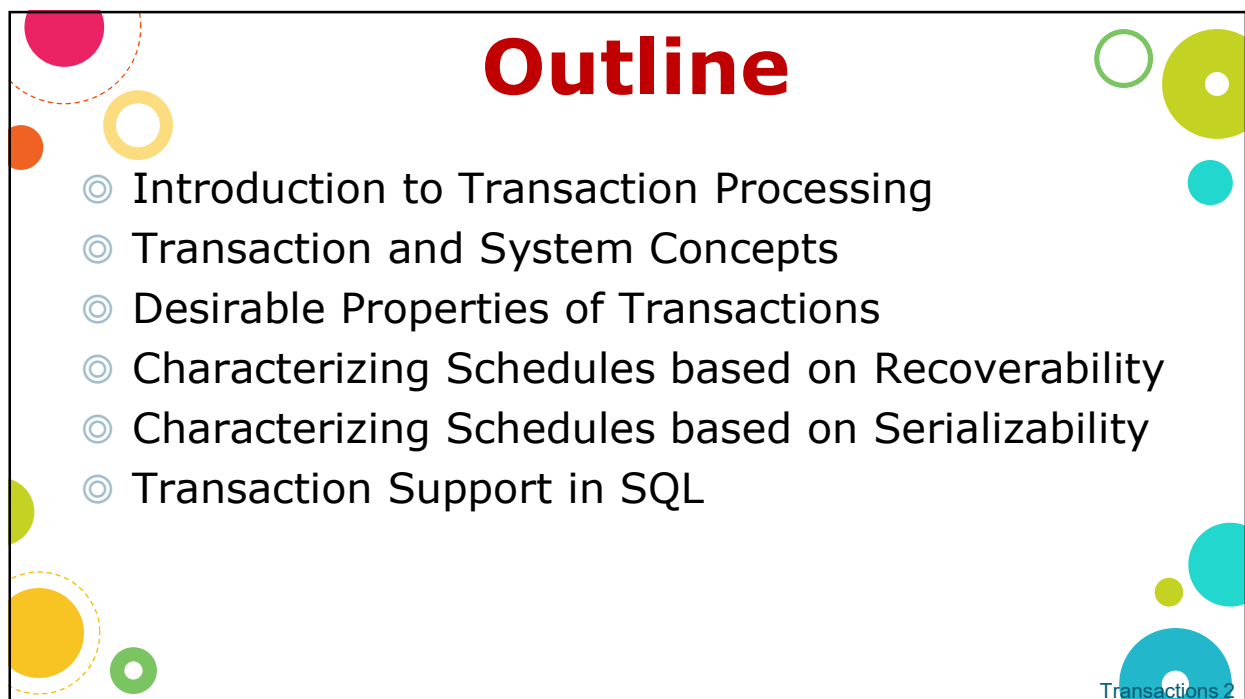


CSIE30600/CSIEB0290
Database Systems

Lecture 12: Transactions

The slide features a decorative background with various colored circles (cyan, green, yellow, orange, pink) and a dashed line. A blue database icon is visible in the bottom right corner.



Outline

- ⊙ Introduction to Transaction Processing
- ⊙ Transaction and System Concepts
- ⊙ Desirable Properties of Transactions
- ⊙ Characterizing Schedules based on Recoverability
- ⊙ Characterizing Schedules based on Serializability
- ⊙ Transaction Support in SQL

Transactions 2

The slide features a decorative background with various colored circles (pink, orange, yellow, green, cyan) and a dashed line.

Concurrent Execution

- ⊙ **Single-User System:**
 - ⊙ At most one user at a time can use the system.
- ⊙ **Multuser System:**
 - ⊙ Many users can access the system concurrently.
- ⊙ **Concurrency**
 - ⊙ **Interleaved processing:**
 - ⊙ Concurrent execution of processes is interleaved in a single CPU
 - ⊙ **Parallel processing:**
 - ⊙ Processes are concurrently executed in multiple CPUs.

Transactions 3

Interleaved/Parallel

- ⊙ Concurrent execution may be **interleaved** or **parallel**.
- ⊙ With proper management, parallel execution is equivalent to interleaving execution.

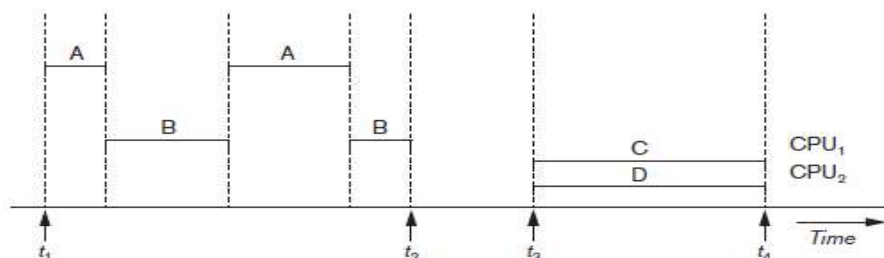


Figure 20.1 Interleaved processing versus parallel processing of concurrent transactions

Transactions 4

Transactions

- ⊙ A **Transaction**: 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.
- ⊙ **Transaction boundaries**:
 - ⊙ Begin and End transaction.
- ⊙ An **application program** may contain several transactions separated by the Begin/End transaction.
- ⊙ **Read-only** transactions, **Read-write** transactions

Transactions 5

Transaction Processing System (TPS)

- ⊙ Systems with large databases may have hundreds or thousands of **concurrent** users/transactions.
- ⊙ A **transaction processing system** is part of the DBMS that manages the processing of concurrent transactions.
- ⊙ Goals:
 - ⊙ **Correctness**
 - ⊙ **Effectiveness**

Transactions 6

Simple Model of a DB

- ⊙ A **database** is a collection of **named data items**
- ⊙ **Granularity** of data – an attribute, a record, or a whole disk block.
- ⊙ Transaction processing 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.

Transactions 7

Read/Write Operations

- ⊙ Basic unit of data transfer (disk ↔ 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)** includes the following steps:
 - ⊙ Find the **address** of the disk block that contains item X.
 - ⊙ Copy that disk block into a **buffer** in main memory (if that disk block is not already in some main memory buffer).
 - ⊙ Copy item X from the buffer to the program **variable** named X.

Transactions 8

Read/Write Operations

- ⦿ **write_item(X)** includes the following steps:
 - ⦿ Find the **address** of the disk **block** that contains item X.
 - ⦿ Copy that disk **block** into a **buffer** in main memory (if that disk block is not already in some memory buffer).
 - ⦿ Copy item **X** from the program **variable** named X into its correct **location** in the **buffer**.
 - ⦿ Store the updated **block** from the **buffer** back to **disk** (either immediately or at some later point in time).
- ⦿ **Read set** of a transaction: set of all items read
- ⦿ **Write set** of a transaction: set of items written

Transactions 9

Sample Transactions

- ⦿ Two sample transactions:
 - ⦿ (a) Transaction T1
 - ⦿ (b) Transaction T2

(a)

T_1
<pre>read_item(X); X := X - N; write_item(X); read_item(Y); Y := Y + N; write_item(Y);</pre>

(b)

T_2
<pre>read_item(X); X := X + M; write_item(X);</pre>

Transactions 10

Why Concurrency Control?

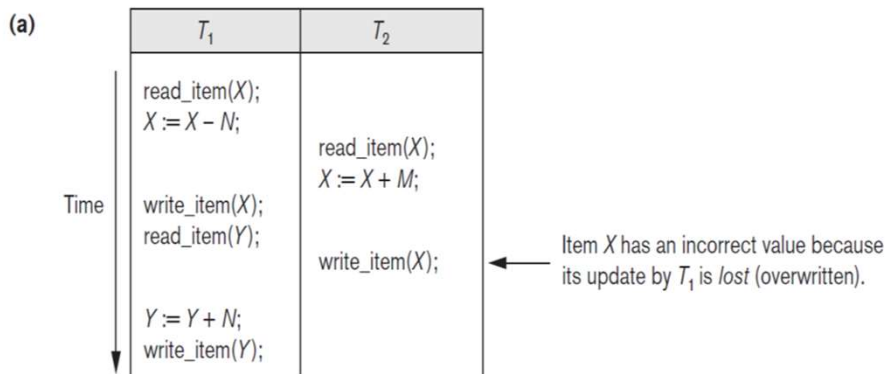
- ⊙ **The Lost Update Problem**
 - ⊙ This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.
- ⊙ **The Temporary Update (or Dirty Read) Problem**
 - ⊙ This occurs when one transaction updates a database item and then the transaction fails for some reason (see Section 17.1.4).
 - ⊙ 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.

Transactions 11

Lost Update

Figure 20.3

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



Transactions 12

Temporary Update

(b)

	T_1	T_2
Time ↓	read_item(X); $X := X - N$; write_item(X);	read_item(X); $X := X + M$; write_item(X);
	read_item(Y);	

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 .

Transactions 13

Incorrect Summary

(c)

	T_1	T_3
Time ↓	read_item(X); $X := X - N$; write_item(X);	$sum := 0$; read_item(A); $sum := sum + A$; ⋮
	read_item(Y); $Y := Y + N$; write_item(Y);	read_item(X); $sum := sum + X$; read_item(Y); $sum := sum + Y$;

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

Transactions 14

Unrepeatable Read Problem

- ⦿ Transaction T reads the same item **twice**
- ⦿ Value is changed by another transaction T' **between** the two reads
- ⦿ T receives **different** values for the two reads of the **same** item
- ⦿ This leads to **inconsistency** of data in T.

RDB Design I 15

Committed/Aborted Transactions

- ⦿ Transactions may be
 - ⦿ **Committed**: all operation performed and effect recorded permanently in the DB.
 - ⦿ **Aborted**: does **not** affect the DB
- ⦿ Transactions may succeed or fail (next slide), TPS must ensure that the effects of **ALL** committed transactions are recorded permanently while **NONE** of the aborted transactions affect the DB in anyway. (known as **All-or-None**)

RDB Design I 16

Why Recovery ?

(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.

Transactions 17

Why Recovery ?

(What causes a Transaction to fail)

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 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.

Transactions 18

Why Recovery ?

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.
- System must keep sufficient info to **recover** from failures to **ensure All-or-none**.

Transactions 19

Transaction Concepts

- A **transaction** is an **atomic unit** of work that is either completed in its **entirety** or **not at all**. (**all or none**)
 - For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.
- Transaction states:**
 - Active state
 - Partially committed state
 - Committed state
 - Failed state
 - Terminated State

Transactions 20

Transaction Execution States

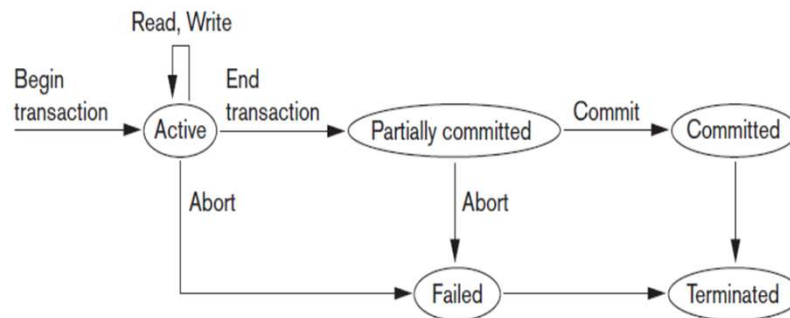


Figure 20.4
State transition diagram illustrating the states for transaction execution.

Transactions 21

Transaction Concepts (2)

- ⦿ **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.
 - ⦿ **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.

Transactions 22

Transaction Concepts (3)

- ⦿ Recovery manager keeps track of the following operations (cont):
 - ⦿ **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.

Transactions 23

Transaction Concepts (4)

- ⦿ 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.

Transactions 24

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.
- ⦿ **Undo** and **redo** operations are possible based on log.

Transactions 25

Log Records

- ⦿ 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:
- ⦿ 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.

Transactions 26

Log Records (cont)

- ⦿ [**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.

Transactions 27

Log Records (cont)

- ⦿ 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` (see Section 21.4).
- ⦿ (more on this later)

Transactions 28

Recovery Using Log Records

- ⦿ If the system crashes, we can recover to a consistent database state by examining the log and using one of the techniques described in Chapter 19.
 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.

Transactions 29

Commit Point

⦿ 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.
- ⦿ Transaction then writes an entry [**commit,T**] into the log.
- ⦿ Force-writing log to disk before transaction reaches commit point.

Transactions 30

Transaction Roll Back

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.
- The effect of all operations that have already performed must be undone.
- Roll back so that transaction T does not have any effect on the DB.

Transactions 31

Transaction Redo

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.)

Transactions 32

Force Write

⦿ 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.

Transactions 33

ACID Properties

- ⦿ **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- ⦿ **Consistency preservation**: A correct execution of the transaction must take the database from one consistent state to another.
- ⦿ **Isolation**: A transaction should not make its updates visible to other transactions until it is committed; this property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks of transactions unnecessary (see Chapter 21).
- ⦿ **Durability or permanency**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Transactions 34

Levels of Isolation

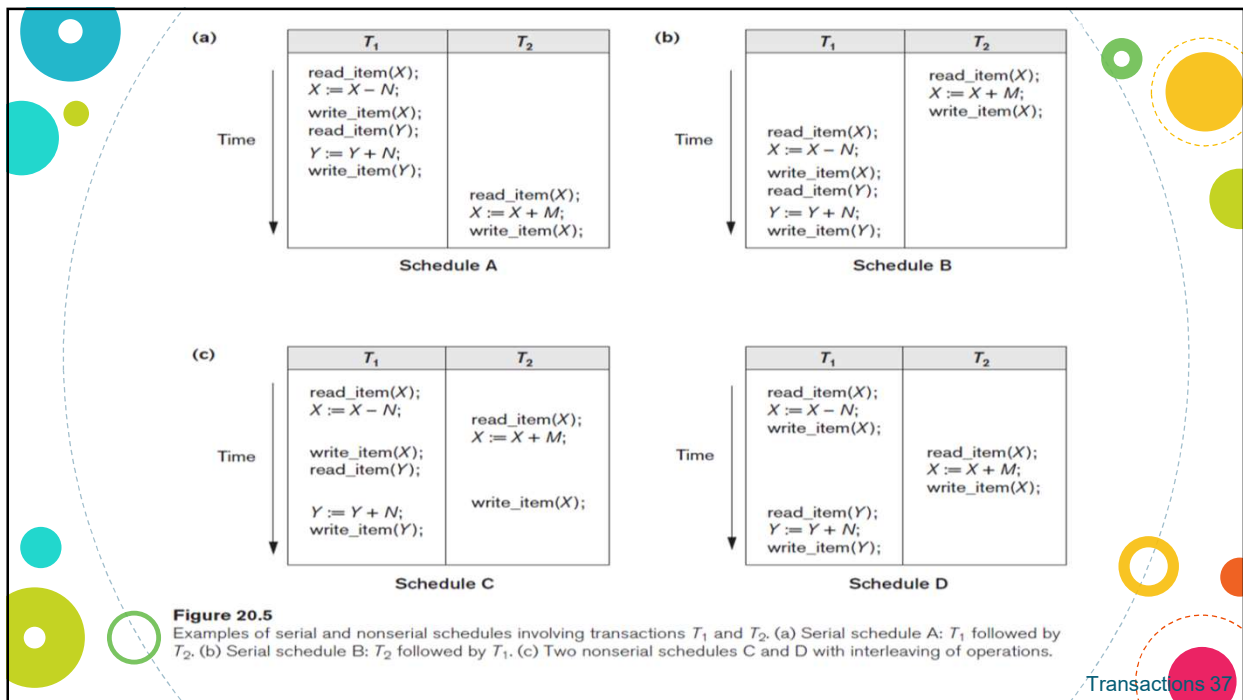
- ⊙ **Level 0 isolation** does not overwrite the dirty reads of higher-level transactions
- ⊙ **Level 1 isolation** has no lost updates
- ⊙ **Level 2 isolation** has no lost updates and no dirty reads
- ⊙ **Level 3 (true) isolation** has repeatable reads
 - ⊙ In addition to level 2 properties
- ⊙ **Snapshot isolation**: data read within a transaction will never reflect changes made by other concurrent transactions.

RDB Design I 35

Schedules

- ⊙ **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 T_1, T_2, \dots, T_n :
 - ⊙ It is an ordering of the operations of the transactions subject to the constraint that, for each transaction T_i that participates in S , the operations of T_i in S must **appear in the same order** in which they occur in T_i .
 - ⊙ Note, however, that operations from other transactions T_j can be interleaved with the operations of T_i in S .

Transactions 36



Transactions 37

Characterizing Schedules

Schedules classified on recoverability:

⊙ **Recoverable schedule:**

- ⊙ One where no committed 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.
- ⊙ **Nonrecoverable schedules** should NOT be permitted.
- ⊙ **Cascading rollback** may still occur in some recoverable schedules

⊙ **Cascadeless schedule:**

- ⊙ One that avoids cascading rollback.
- ⊙ Every transaction reads only the items that are written by committed transactions.

Transactions 38

Characterizing Schedules (2)

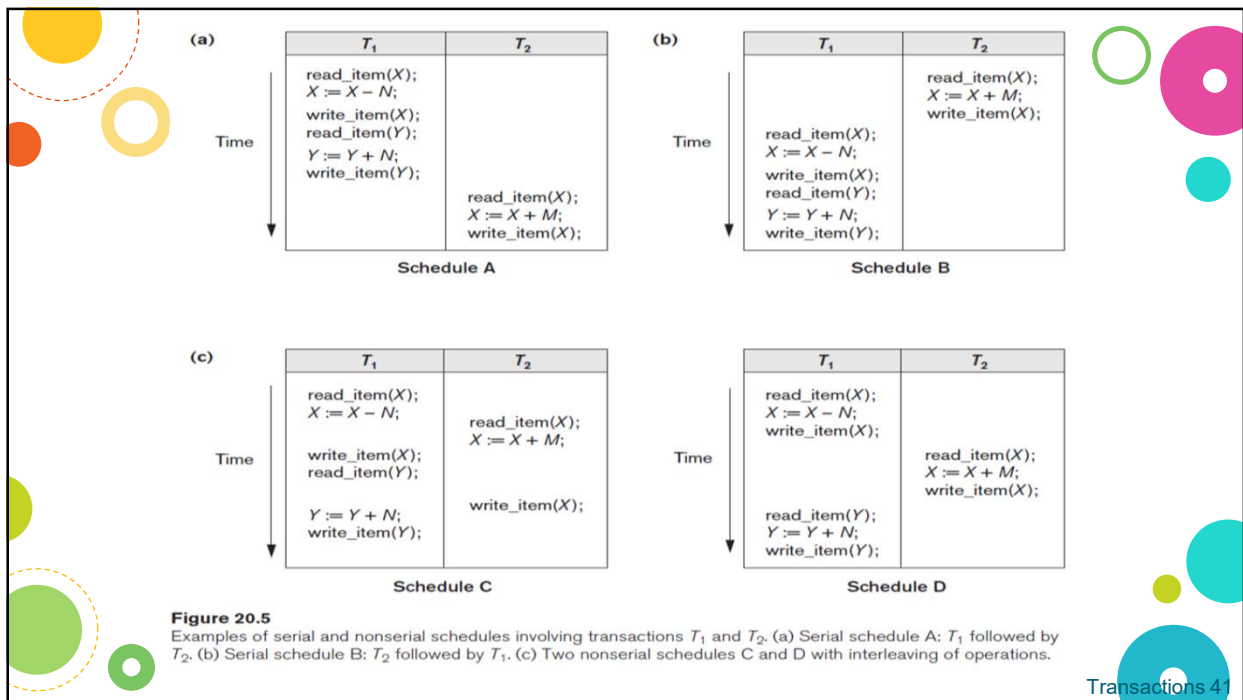
- ⦿ **Schedules requiring cascaded rollback:**
 - ⦿ A schedule in which uncommitted transactions that read an item from a failed transaction must be **rolled back**.
- ⦿ **Strict Schedules:**
 - ⦿ A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.
 - ⦿ Simpler recovery process: restore the before image

Transactions 39

Serializability

- ⦿ **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.
- ⦿ Any serial schedule is considered a **correct** schedule since all transactions are committed.

Transactions 40



Serializability

- ⊙ Problem with serial schedules
 - ⊙ Limit concurrency by prohibiting interleaving of operations
 - ⊙ Unacceptable in practice
 - ⊙ Solution: determine which schedules are equivalent to a serial schedule and allow those to occur
- ⊙ **Serializable schedule:**
 - ⊙ A schedule S is **serializable** if it is **equivalent** to some serial schedule of the same n transactions.

Transactions 42

Result Equivalent

- ⊙ **Result equivalent:**
 - ⊙ Two schedules are called result equivalent if they produce the same **final state** of the database.
- ⊙ Two different schedules may accidentally produce the same final state.
- ⊙ Not a good definition of equivalence of schedules.

Figure 20.6 Two schedules that are result equivalent for the initial value of $X = 100$ but are not result equivalent in general

S_1
read_item(X); $X := X + 10$; write_item(X);

S_2
read_item(X); $X := X * 1.1$; write_item(X);

Transactions 43

Conflict Serializability

- ⊙ Two operations are **conflict** if
 - ⊙ They belong to different transactions.
 - ⊙ They access the same database item.
 - ⊙ At least one of them is **write**.
- ⊙ **Read-write conflict, Write-write conflict**
- ⊙ **Conflict equivalent:**
 - ⊙ Two schedules are said to be conflict equivalent if the **relative 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' .

Transactions 44

Testing Conflict Serializability

Algorithm 20.1:

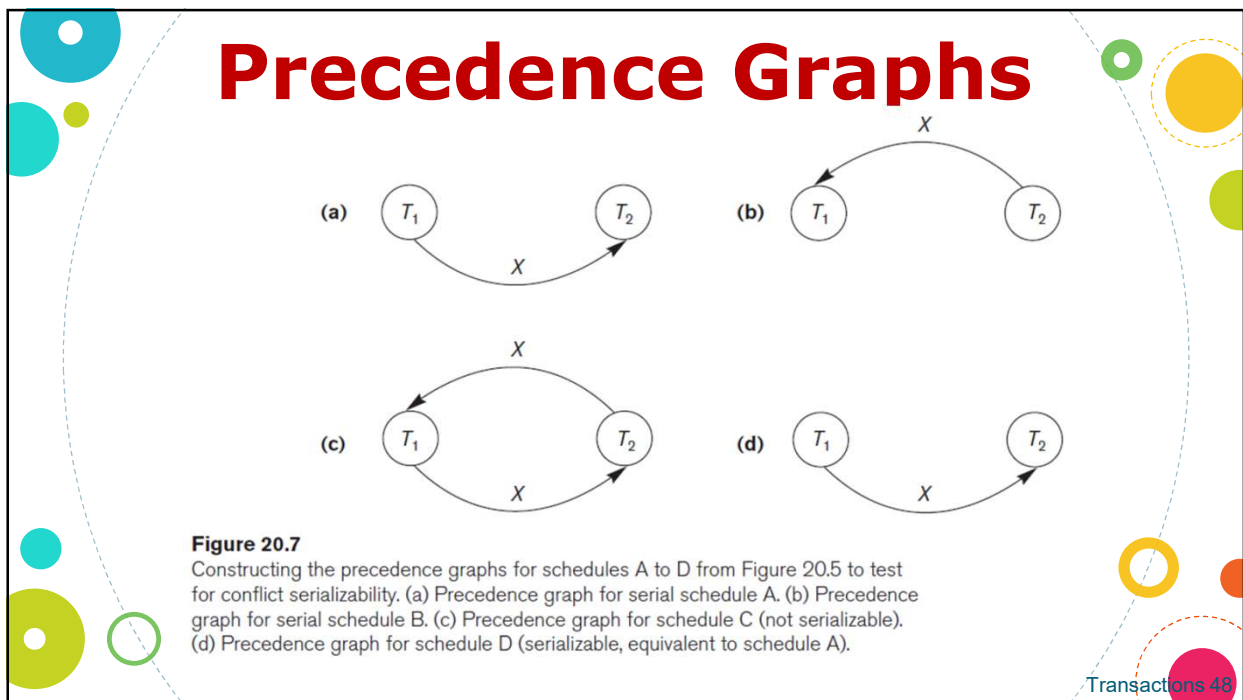
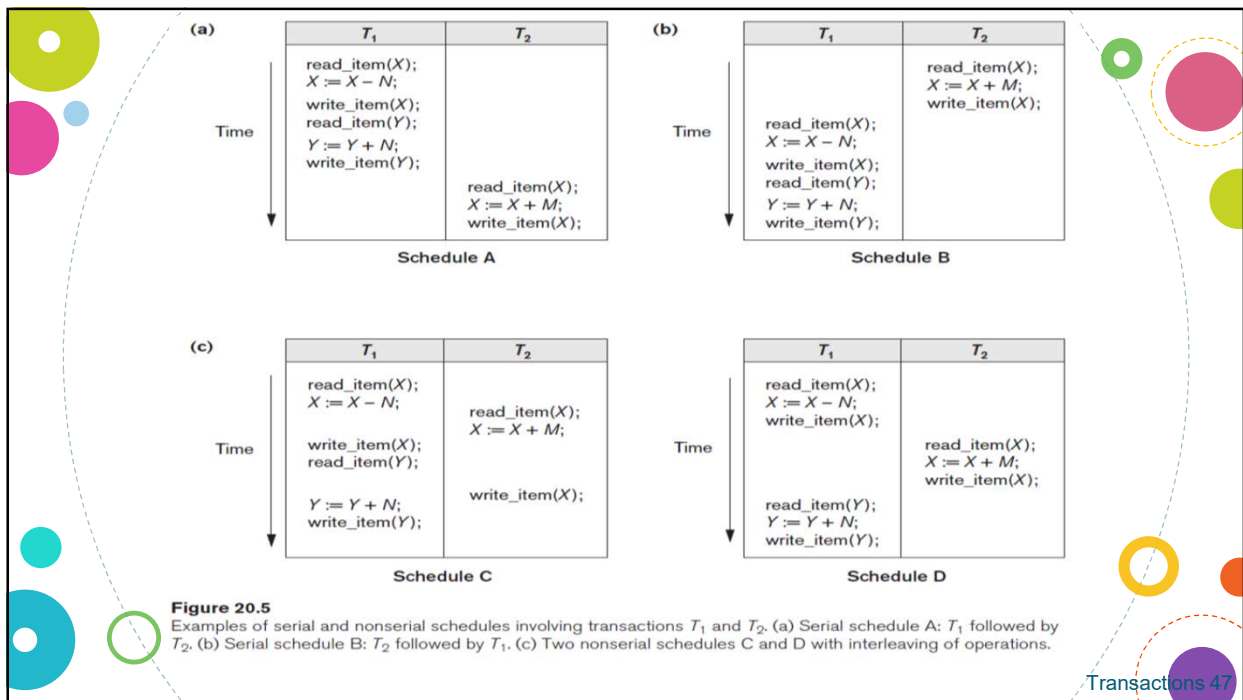
- Looks at only `read_Item(X)` and `write_Item(X)` operations
- Constructs a **precedence graph (serialization graph)** - a graph with directed edges
- An edge is created from T_i to T_j if one of the operations in T_i appears before a conflicting operation in T_j
- The schedule is serializable if and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

Transactions 45

Algorithm 20.1

1. For each transaction T_i participating in schedule S , create a node labeled T_i in the precedence graph.
2. For each case in S where T_j executes a `read_item(X)` after T_i executes a `write_item(X)`, create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
3. For each case in S where T_j executes a `write_item(X)` after T_i executes a `read_item(X)`, create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
4. For each case in S where T_j executes a `write_item(X)` after T_i executes a `write_item(X)`, create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
5. The schedule S is serializable if and only if the precedence graph has no cycles.

Transactions 46



Another Example

Figure 20.8

Another example of serializability testing. (a) The read and write operations of three transactions T_1 , T_2 , and T_3 . (b) Schedule E. (c) Schedule F.

(a)

Transaction T_1	Transaction T_2	Transaction T_3
read_item(X);	read_item(Z);	read_item(Y);
write_item(X);	read_item(Y);	read_item(Z);
read_item(Y);	write_item(Y);	write_item(Y);
write_item(Y);	read_item(X);	write_item(Z);
	write_item(X);	

Another Example (2)

(b)

	Transaction T_1	Transaction T_2	Transaction T_3
Time ↓	read_item(X); write_item(X);	read_item(Z); read_item(Y); write_item(Y);	read_item(Y); read_item(Z);
	read_item(Y); write_item(Y);	read_item(X); write_item(X);	write_item(Y); write_item(Z);

Schedule E

Another Example (3)

(c)

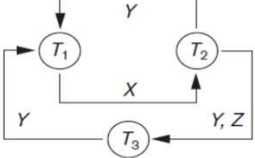
	Transaction T_1	Transaction T_2	Transaction T_3
Time ↓	read_item(X); write_item(X); read_item(Y); write_item(Y);	 read_item(Z); read_item(Y); write_item(Y); read_item(X); write_item(X);	read_item(Y); read_item(Z); write_item(Y); write_item(Z);

Schedule F

Transactions 51

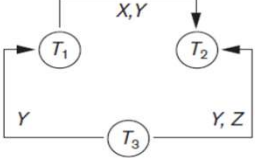
Figure 20.8 (continued)
 Another example of serializability testing. (d) Precedence graph for schedule E. (e) Precedence graph for schedule F. (f) Precedence graph with two equivalent serial schedules.

(d)



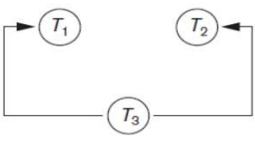
Equivalent serial schedules
None
Reason
 Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$
 Cycle $X(T_1 \rightarrow T_2), YZ(T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

(e)



Equivalent serial schedules
 $T_3 \rightarrow T_1 \rightarrow T_2$

(f)



Equivalent serial schedules
 $T_3 \rightarrow T_1 \rightarrow T_2$
 $T_3 \rightarrow T_2 \rightarrow T_1$

Transactions 52

Serializability

- ⊙ 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.

Transactions 53

How Serializability is Used

- ⊙ Serializable schedule gives **benefit of concurrent execution** without giving up any correctness
- ⊙ Difficult to test for serializability in practice
 - ⊙ Interleaving of operations occurs in an operating system through some scheduler
 - ⊙ Difficult to determine beforehand how the operations in a schedule will be interleaved.
 - ⊙ Factors such as system load, time of transaction submission, and process priority affect ordering of operations
- ⊙ DBMS enforces protocols
 - ⊙ Set of rules to ensure serializability

Transactions 54

Ensuring Serializability

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 (2PL)**

Transactions 55

View Serializability

- ⊙ **View equivalence**:
 - ⊙ A less restrictive definition of equivalence of schedules(next slide)
- ⊙ **View serializability**:
 - ⊙ Definition of serializability based on view equivalence.
 - ⊙ A schedule is **view serializable** if it is *view equivalent* to a serial schedule.

Transactions 56

View Equivalent Conditions

- ⊙ Two schedules S and S' are said to be **view equivalent** if the following 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 $R_i(X)$ of T_i in S , if the value of X read by the operation has been written by an operation $W_j(X)$ of T_j (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 $R_i(X)$ of T_i in S' .
 3. If the operation $W_k(Y)$ of T_k is the **last** operation to write item Y in S , then $W_k(Y)$ of T_k must also be the last operation to write item Y in S' .

Transactions 57

View Equivalence

- ⊙ 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.

Transactions 58

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., $\text{new } X = f(\text{old } X)$
- ⊙ 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.

Transactions 59

View and Conflict Equivalence

- ⊙ Relationship between view and conflict equivalence (cont):
 - ⊙ 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 T2 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.

Transactions 60

Other Types of Equivalence

- ⦿ Under special **semantic constraints**, schedules that are otherwise not conflict serializable may work correctly.
 - ⦿ Using commutative operations of addition and subtraction (which can be done in any order) certain non-serializable transactions may work correctly

Transactions 61

Other Types of Equivalence

- ⦿ Example: bank credit /debit transactions on a given item are **separable** and **commutative**.
 - ⦿ Consider the following schedule S for the two transactions:
 - ⦿ Sh : r1(X); w1(X); r2(Y); w2(Y); r1(Y); w1(Y); r2(X); w2(X);
 - ⦿ Using conflict serializability, it is **not serializable**.
 - ⦿ However, if it came from a (read, update, write) sequence as follows:
 - ⦿ r1(X); X := X - 10; w1(X); r2(Y); Y := Y - 20; w2(Y); r1(Y); Y := Y + 10; w1(Y); r2(X); X := X + 20; w2(X);
 - ⦿ Sequence explanation: debit, debit, credit, credit.
 - ⦿ It is a *correct schedule for the given semantics*

Transactions 62

Transactions in SQL

- ⊙ A **single** SQL statement is always considered to be **atomic**.
 - ⊙ Either the statement completes execution without error or it fails and leaves the database unchanged.
- ⊙ With older SQL, there is no explicit Begin Transaction statement.
 - ⊙ Transaction initiation is done **implicitly** when particular SQL statements are encountered.
- ⊙ Every transaction must have an explicit end statement, which is either a **COMMIT** or **ROLLBACK**.

Transactions 63

Transaction in SQL2

Characteristics specified by a **SET TRANSACTION** statement in SQL2:

- ⊙ **Access mode**:
 - ⊙ READ ONLY or READ WRITE.
 - The default is READ WRITE unless the isolation level of READ UNCOMMITTED is specified, in which case READ ONLY is assumed.
- ⊙ **Diagnostic size n**, specifies an integer value n, indicating the **number of conditions** that can be held simultaneously in the diagnostic area.

Transactions 64

Transaction in SQL2

Characteristics specified by a SET TRANSACTION statement in SQL2 (contd.):

- ◎ **Isolation level** <isolation>, where <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ or SERIALIZABLE. The default is SERIALIZABLE.
 - ◎ With SERIALIZABLE: the interleaved execution of transactions will adhere to our notion of serializability.
 - ◎ However, if any transaction executes at a lower level, then serializability may be violated.

Transactions 65

Transaction in SQL2

Potential problems with lower isolation levels:

- ◎ **Dirty Read:**
 - ◎ Reading a value that was written by a transaction which failed.
- ◎ **Nonrepeatable Read:**
 - ◎ Allowing another transaction to write a new value between multiple reads of one transaction.
 - ◎ A transaction T1 may read a given value from a table. If another transaction T2 later updates that value and T1 reads that value again, T1 will see a **different value**.
 - Consider that T1 reads the employee salary for Smith. Next, T2 updates the salary for Smith. If T1 reads Smith's salary again, then it will see a different value for Smith's salary.

Transactions 66

Transactions in SQL2

⊙ Potential problem with lower isolation levels (contd.):

⊙ **Phantoms:**

- ⊙ New rows being read using the same read with a condition.
 - A transaction T1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE clause.
 - Now suppose that a transaction T2 inserts a new row that also satisfies the WHERE clause condition of T1, into the table used by T1.
 - If T1 is repeated, then T1 will see a row that previously did not exist, called a **phantom**.

Transactions 67

Sample SQL Transaction

```
EXEC SQL whenever sqlerror go to UNDO;
EXEC SQL SET TRANSACTION
    READ WRITE
    DIAGNOSTICS SIZE 5
    ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT
    INTO EMPLOYEE (FNAME, LNAME, SSN, DNO, SALARY)
    VALUES ('Robert','Smith','991004321',2,35000);
EXEC SQL UPDATE EMPLOYEE
    SET SALARY = SALARY * 1.1
    WHERE DNO = 2;
EXEC SQL COMMIT;
    GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ...
```

Transactions 68

Possible Violation of Serializability

Isolation level	Type of Violation		
	Dirty read	nonrepeatable read	phantom
READ UNCOMMITTED	yes	yes	yes
READ COMMITTED	no	yes	yes
REPEATABLE READ	no	no	yes
SERIALIZABLE	no	no	no

Transactions 69

- ## Transactions in SQL
- ⦿ Newer versions of SQL support transactions:
 - ⦿ BEGIN TRANSACTION
 - ⦿ SET TRANSACTION
 - ⦿ COMMIT
 - ⦿ ROLLBACK
 - ⦿ SAVEPOINT
 - ⦿ RELEASE SAVEPOINT
 - ⦿ Check the standard doc for more details.
- Transactions 70

Summary

- ⦿ Transaction and System Concepts
- ⦿ Desirable Properties of Transactions
- ⦿ Characterizing Schedules based on Recoverability
- ⦿ Characterizing Schedules based on Serializability
- ⦿ Transaction Support in SQL