FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Introduction to Transaction Processing Concepts and Theory

DATABASES - lectures

Outline

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

Introduction to Transaction Processing

- **Single-User System**:
  - At most one user at a time can use the system.
- **Multiuser 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.

Introduction to Transaction Processing

- A **Transaction**:
  - Logical unit of database processing that includes one or more access operations
    - Access operations
      - read –retrieval,
      - write - insert
      - update, delete.
- A transaction (set of operations) may be:
  - stand-alone specified in a high level language like SQL submitted interactively,
  - embedded within a program.
- **Transaction boundaries**:
  - Begin and End transaction.
- An **application program** may contain several transactions separated by the Begin and End transaction boundaries.

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.

Simple model of a database

- 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.

Simple model of a database

- **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).

- Step 4 updates the database file on disk

Example

- Two simple transactions example:
  - X and Y are database item storing the number of reserved seats for two flights
  - N and M are number of reservations

(a)

T_1

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

(b)

T_2

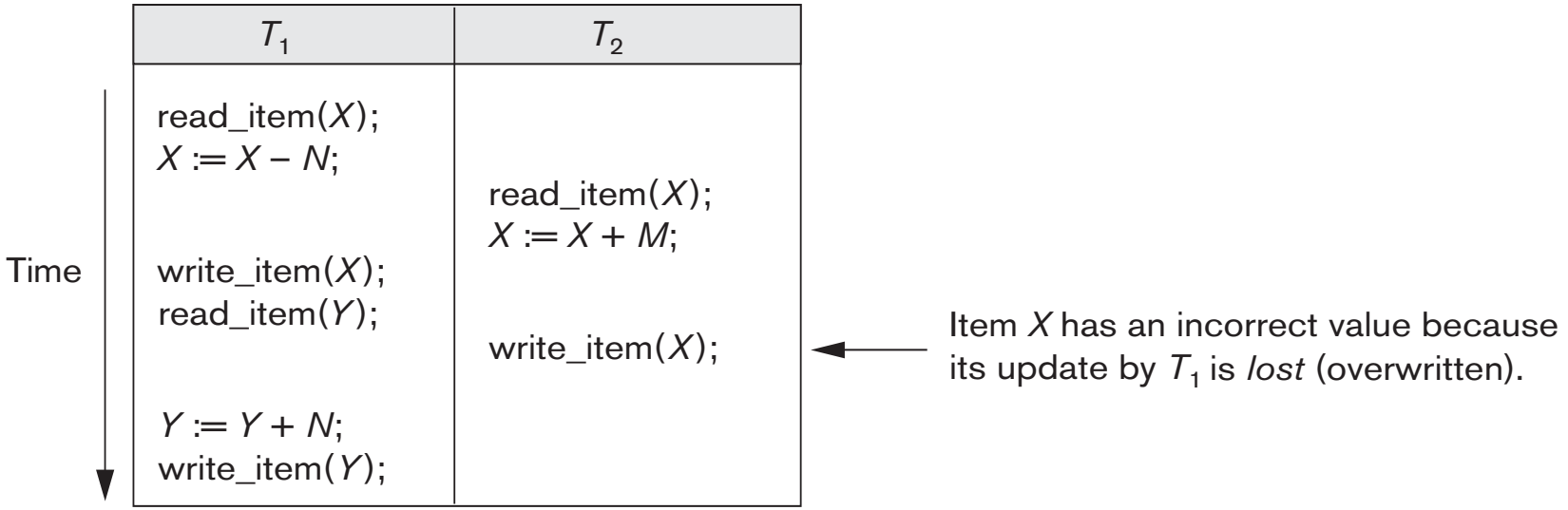
read_item(X);
$X := X + M$;
write_item(X);

What happens when T1 and T2 run concurrently?

Why Concurrency Control is needed

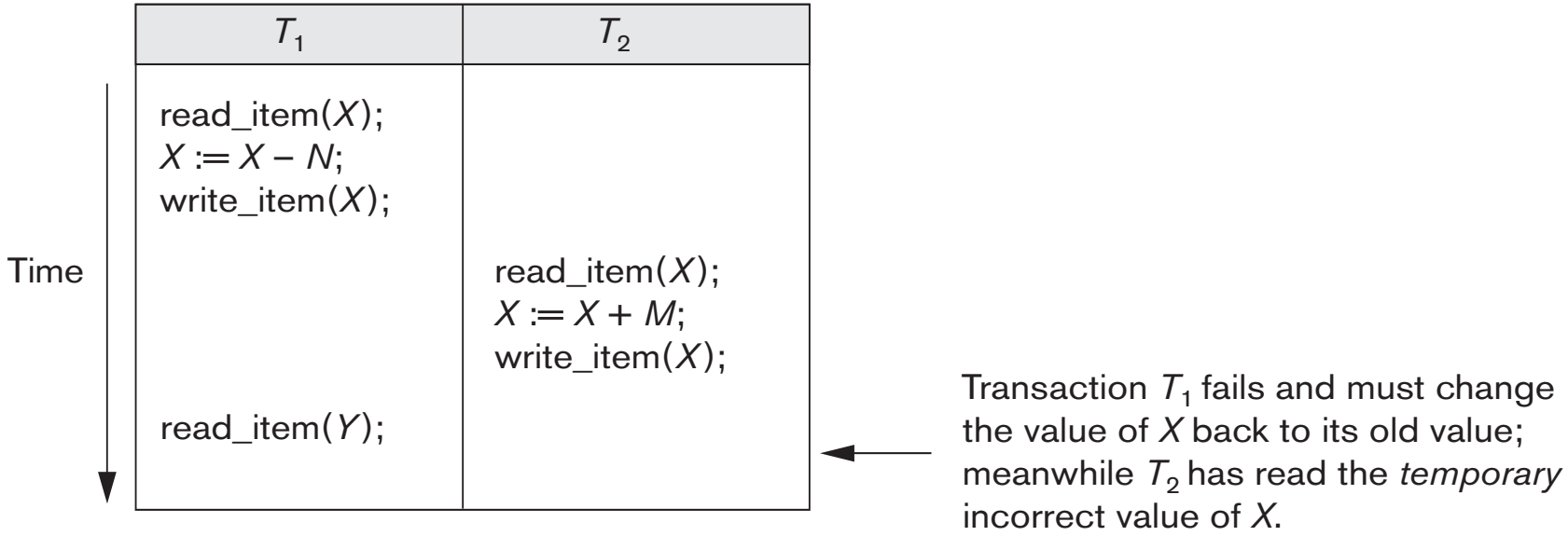
The Lost Update Problem

- 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.



Why Concurrency Control is needed

- **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.



Why Concurrency Control is needed

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.

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*).

Why Concurrency Control is needed

- **The Unrepeatable Read Problem**
  - A transaction T reads the same item twice and the item is changed by another transaction T' between the two reads. Hence, T receives *different values* for its two reads of the same item.
  - Example:
    If during an airline reservation transaction, a customer inquires about seat availability on several flights. When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation, and it may end up reading a different value for the item.

What causes a transaction to fail

1. A computer failure (system crash)
2. A transaction or system error
3. Local errors or exception conditions detected by the transaction
4. Concurrency control enforcement
5. Disk failure
6. Physical problems and catastrophes

The DBMS should store enough information to recover from failure
(most commonly failures from 1 to 4)

Transaction Concepts

- A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all.
- 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.

Transaction Concepts

- 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.

Transaction Concepts

- **Execution states of a transaction**:
  - Active state
  - Partially committed state
  - Committed state
  - Failed state
  - Terminated state

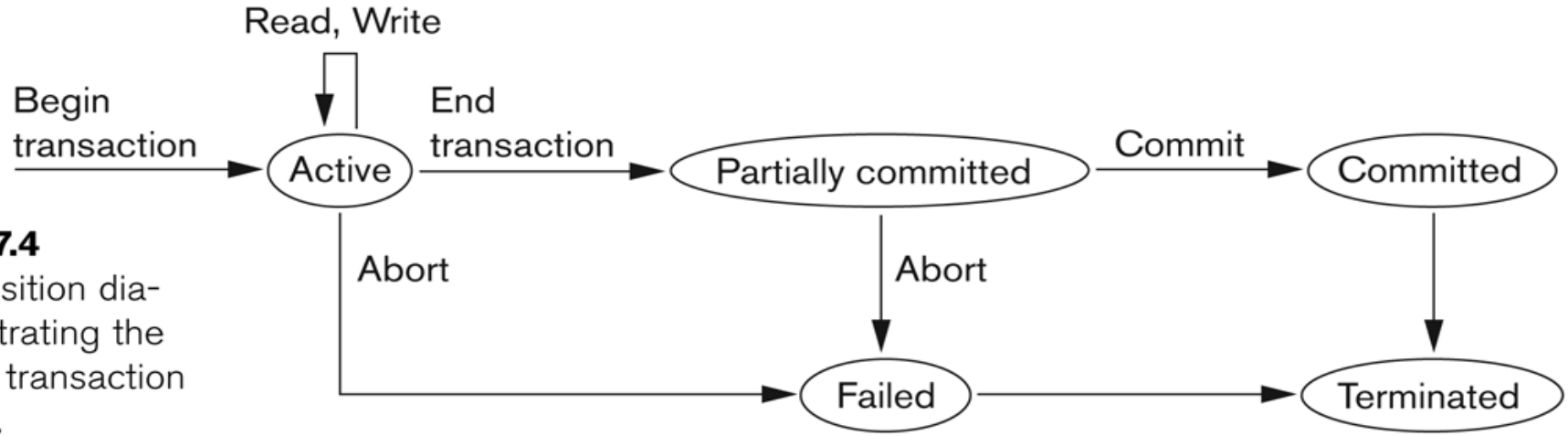


Figure 17.4
State transition diagram illustrating the states for transaction execution.

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.

The System Log

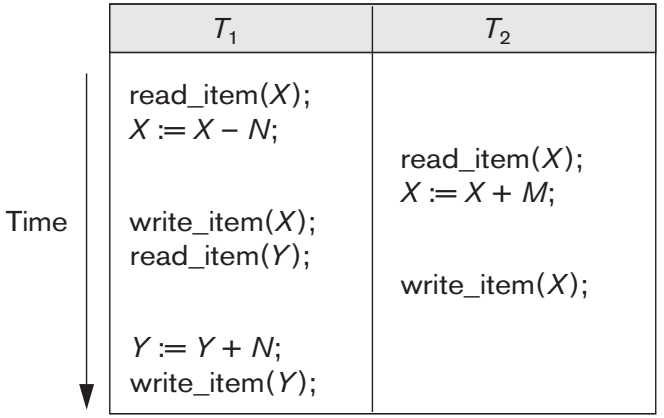
- T in the following discussion refers to a unique transaction-id that is generated automatically by the system and is used to identify each transaction:
- Types of log record:
  - [**start_transaction, T**]
    Records that transaction T has started execution.
  - [**write_item, T, X, old_value, new_value**]
    Records that transaction T has changed the value of database item X from old_value to new_value.
  - [**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.

Desirable Properties of Transactions

- **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.
- **Durability or permanency.** Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Transaction Schedule (History)

- A **schedule** (or **history**) S of n transactions T_1, T_2, …, 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.



S_a: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$

Transaction Schedule (History)

- Two operations in a schedule are said to **conflict** if they satisfy all three of the following conditions:
  - they belong to *different transactions*;
  - they access the *same item X*;
  - *at least one* of the operations is a write_item(X).

- Example:
  - In schedule S_a, operations $r_1(X)$ and $w_2(X)$ conflict, as do the operations $r_2(X)$ and $w_1(X)$, and the operations $w_1(X)$ and $w_2(X)$. However, the operations $r_1(X)$ and $r_2(X)$ do not conflict, since they are both read operations; the operations $w_2(X)$ and w_1 (Y) do not conflict because they operate on distinct data items *X* and *Y*; The operations $r_1(X)$ and $w_1(X)$ do not conflict because they belong to the same transaction.

$$S_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y)$$

Characterizing Schedules based 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.

S_a': $r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); c_2; w_1(Y); c_1$

S_c: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_2; a_1$

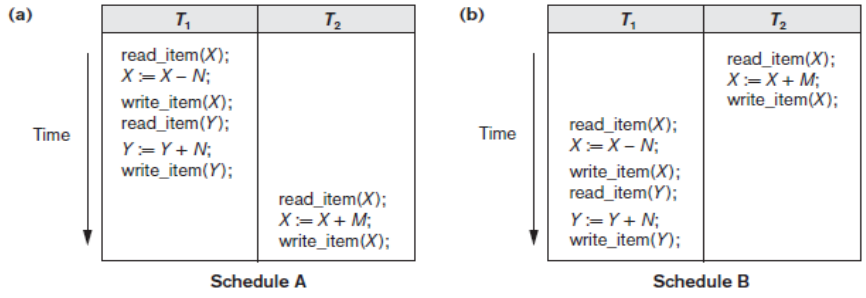
S_d: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2$

Characterizing Schedules based on Recoverability

- **Schedules requiring cascaded rollback**:
  - A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.
- **Cascadeless schedule**:
  - One where every transaction reads only the items that are written by committed transactions.
- **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.

Characterizing Schedules based on Serializability

- Types of schedules that are always considered to be correct when concurrent transactions are executing
- **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.
- **Example:**
  Flight reservations manipulation



What would happen if all schedules are serial?

What if interleaving of operations is allowed?

How many different schedules can there be?

Characterizing Schedules based on Serializability

- **Serializable schedule**:
  - A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.

(c)

Time

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);	

Schedule C

Time

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

Schedule D

We are interested in the nonserial schedules that are serializable!

Characterizing Schedules based on Serializability

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

Figure 21.6
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);

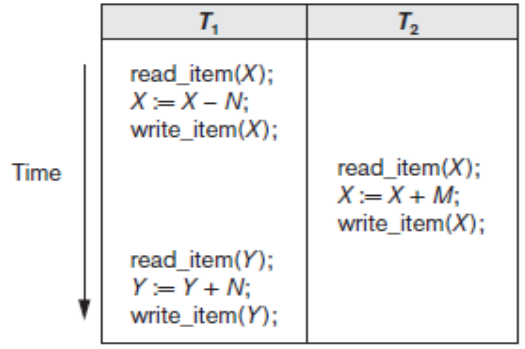
S_2
read_item(X); $X := X * 1.1$; write_item (X);

What happens when the initial value of X changes?

Characterizing Schedules based on Serializability

- 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'.



Schedule D

Characterizing Schedules based on 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.
- 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.

Characterizing Schedules based on 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

Characterizing Schedules based on Serializability

Testing for conflict serializability: Algorithm 17.1:

- 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 Ti to Tj if one of the operations in Ti appears before a conflicting operation in Tj
- The schedule is serializable if and only if the precedence graph has no cycles.

Characterizing Schedules based on Serializability

(a)

T_1	T_2

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

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

Time

Schedule A

(b)

T_1	T_2

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

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

Time

Schedule B

(c)

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

Schedule C

(d)

T_1	T_2

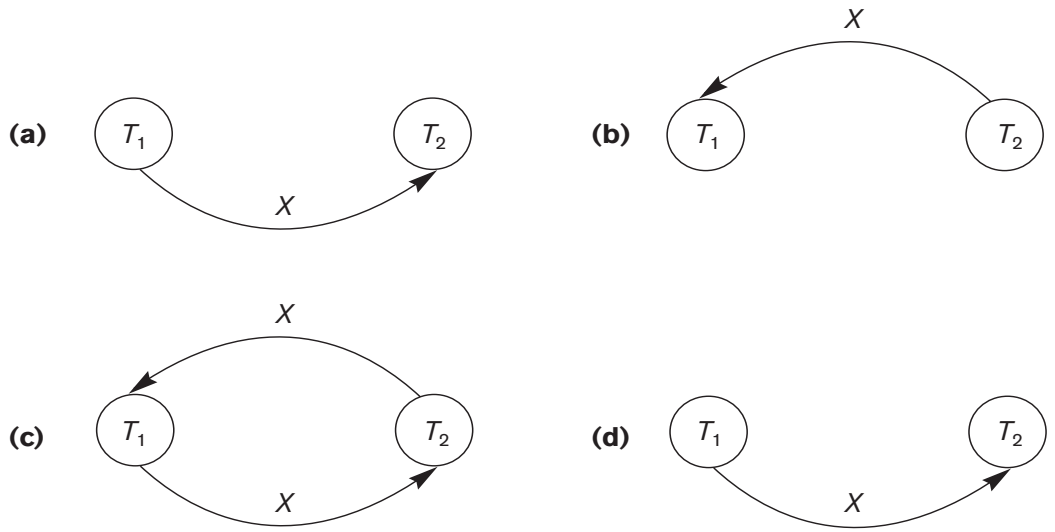
read_item(X);
$X := X - N$;
write_item(X);	

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

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

Time

Schedule D



Precedence Graphs

Characterizing Schedules based on Serializability

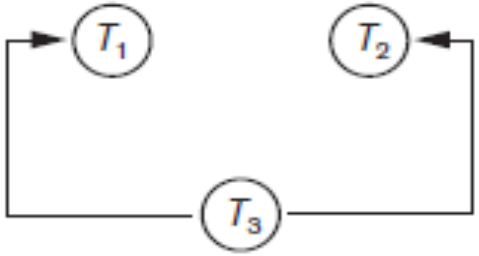
Figure 17.8
Another example of serializability testing. (a) The read and write operations of three transactions T_1, T_2, and T_3. (b) Schedule E. (c) Schedule *F*.

(a)

Transaction T_1
read_item(X);
write_item(X);
read_item(Y);
write_item(Y);

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

Transaction T_3
read_item(Y);
read_item(Z);
write_item(Y);
write_item(Z);



$T_3 \rightarrow T_1 \rightarrow T_2$

$T_3 \rightarrow T_2 \rightarrow T_1$

Characterizing Schedules based on Serializability

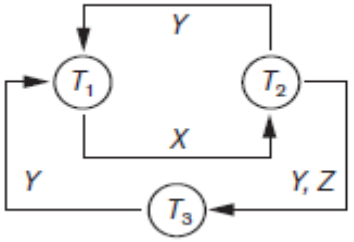
Figure 17.8
Another example of serializability testing. (a) The read and write operations of three transactions T_1, T_2, and T_3. (b) Schedule E. (c) Schedule *F*.

(b)

Time ↓

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

Schedule E



Equivalent serial schedules

None

Reason

Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$
Cycle $X(T_1 \rightarrow T_2), YZ\ (T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

Characterizing Schedules based on Serializability

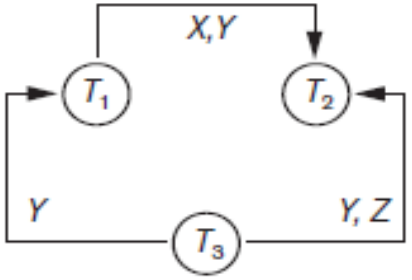
Figure 17.8
Another example of serializability testing. (a) The read and write operations of three transactions T_1, T_2, and T_3. (b) Schedule E. (c) Schedule *F*.

(c)

Time

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

Schedule F



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

Other Types of Equivalence of Schedules

- **View** equivalence and view serializability :
  - A less restrictive definition of equivalence of schedules
  - 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'.

Other Types of Equivalence of Schedules

- 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

Transaction Support in SQL2

- 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 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.

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 UNCOMITTED 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.
  - These conditions supply feedback information (errors or exceptions) to the user or program on the *n* most recently executed SQL statement

Characteristics specified by a SET TRANSACTION statement in SQL2

- **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.

Potential problems with lower isolation levels

- **Dirty Read**:
  - A transaction T1 may read the update of a transaction T2, which has not yet committed.
  - If T2 fails and is aborted, then T1 would have read a value that does not exist and is incorrect.

- **Nonrepeatable Read**:
  - 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.
    - Example: 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.

Potential problems with lower isolation levels

- **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.

Transaction Support in SQL2

- Possible violation of serializabilty:

Table 21.1 Possible Violations Based on Isolation Levels as Defined in SQL

	Type of Violation		
Isolation Level	**Dirty Read**	**Nonrepeatable Read**	**Phantom**
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

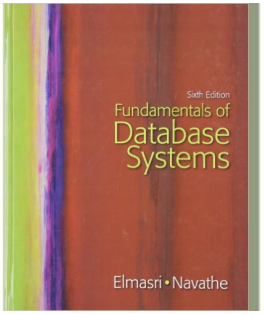
Example

```
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:  ... ;
```

Short summary

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

Bibliography



- Chapter 21

- Chapter 18
- Chapter 19