RV Institute of Technology and Management®

Course Name: Database Management System
Course Code: BCS403

Module 4

Advanced Queries Chapter Outline

- 9.1 General Constraints as Assertions
- 9.2 Views in SQL
- 9.3 Database Programming
- 9.4 Embedded SQL
- 9.5 Functions Calls, SQL/CLI
- 9.6 Stored Procedures, SQL/PSM
- 9.7 Summary

Chapter Objectives

- Specification of more general **constraints** via assertions
- SQL facilities for defining views (virtual tables)
- Various techniques for accessing and manipulating a database via programs in general-purpose languages
  - E.g., Java, C++, etc.

Relational Database Schema--Figure 5.5

EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO

DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE

DEPT_LOCATIONS

DNUMBER	DLOCATION

PROJECT

PNAME	PNUMBER	PLOCATION	DNUM

WORKS_ON

ESSN	PNO	HOURS

DEPENDENT

ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP

Using CHECK clause – Restrict attribute and domain values

- For example, suppose that department numbers are restricted to integer numbers between 1 and 20; then, we can change the attribute declaration of Dnumber in the DEPARTMENT table to the following:

  Dnumber INT NOT NULL

  CHECK (Dnumber > 0 AND Dnumber < 21);

- The CHECK clause can also be used in conjunction with the CREATE DOMAIN statement.
- For example, we can write the following statement:

  CREATE DOMAIN D_NUM AS INTEGER

  CHECK (D_NUM > 0 AND D_NUM < 21);

Constraints as Assertions

- General constraints: constraints that do not fit in the basic SQL
- Mechanism: **`CREAT ASSERTION`**
  - Components include:
    - a constraint name,
    - followed by `CHECK`,
    - followed by a condition

RV Institute of Technology and Management®

Assertions: An Example

- "The salary of an employee must not be greater than the salary of the manager of the department that the employee works for''

constraint name, CHECK, condition

```
CREAT ASSERTION SALARY_CONSTRAINT
CHECK (NOT EXISTS (SELECT *
      FROM EMPLOYEE E, EMPLOYEE M,
   DEPARTMENT D
      WHERE E.SALARY > M.SALARY AND
            E.DNO=D.NUMBER AND
      D.MGRSSN=M.SSN))
```

Using General Assertions

- Specify a query that violates the condition; include inside a `NOT EXISTS` clause
- The DBMS is responsible for ensuring that the condition is not violated.
- Query result must be empty
  - if the query result is not empty, the assertion has been violated

SQL Triggers

- Objective: to monitor a database and take initiate action when a condition occurs
- Triggers are expressed in a syntax similar to assertions and include the following:
  - Event
    - Such as an insert, deleted, or update operation
  - Condition
  - Action
    - To be taken when the condition is satisfied

RV Institute of Technology and Management®

SQL Triggers: An Example

- A trigger to compare an employee's salary to his/her supervisor during insert or update operations:

```
CREATE TRIGGER INFORM_SUPERVISOR
BEFORE INSERT OR UPDATE OF
   SALARY, SUPERVISOR_SSN ON EMPLOYEE
   FOR EACH ROW
        WHEN
        (NEW.SALARY> (SELECT SALARY FROM EMPLOYEE
                    WHERE SSN=NEW.SUPERVISOR_SSN))
        INFORM_SUPERVISOR (NEW.SUPERVISOR_SSN,NEW.SSN);
```

Views in SQL

- A view is a “virtual” table that is derived from other tables
- Allows for limited update operations
  - Since the table may not physically be stored
- Allows full query operations
- A convenience for expressing certain operations

Specification of Views

- SQL command: **`CREATE VIEW`**
  - a table (view) name
  - a possible list of attribute names (for example, when arithmetic operations are specified or when we want the names to be different from the attributes in the base relations)
  - a query to specify the table contents

SQL Views: An Example

- Specify a different WORKS_ON table

```
CREATE VIEW WORKS_ON_NEW AS
SELECT FNAME, LNAME, PNAME, HOURS
    FROM EMPLOYEE, PROJECT, WORKS_ON
    WHERE SSN=ESSN AND PNO=PNUMBER
    GROUP BY PNAME;
```

Using a Virtual Table

- We can specify SQL queries on a newly create table (view):

```
SELECT FNAME, LNAME
    FROM WORKS_ON_NEW
    WHERE PNAME='Seena';
```

- When no longer needed, a view can be dropped:

```
DROP WORKS_ON_NEW;
```

Efficient View Implementation

- Query modification:
  - Present the view query in terms of a query on the underlying base tables
- Disadvantage:
  - Inefficient for views defined via complex queries
    - Especially if additional queries are to be applied to the view within a short time period

RV Institute of Technology and Management®

Efficient View Implementation

- View materialization:
  - Involves physically creating and keeping a temporary table
- Assumption:
  - Other queries on the view will follow
- Concerns:
  - Maintaining correspondence between the base table and the view when the base table is updated
- Strategy:
  - Incremental update

Update Views

- Update on a single view without aggregate operations:
  - Update may map to an update on the underlying base table
- Views involving joins:
  - An update *may* map to an update on the underlying base relations
    - Not always possible

Un-updatable Views

- Views defined using groups and aggregate functions are not updateable
- Views defined on multiple tables using joins are generally not updateable
- **`WITH CHECK OPTION`**: must be added to the definition of a view if the view is to be updated
  - To allow check for updatability and to plan for an execution strategy

Schema Change Statements in SQL

- Adding or dropping tables, attributes, constraints, and other schema elements.
- **DROP SCHEMA COMPANY CASCADE;**
- **DROP TABLE DEPENDENT CASCADE;**
- **ALTER TABLE COMPANY.EMPLOYEE ADD COLUMN Job VARCHAR(12);**
- **ALTER TABLE COMPANY.EMPLOYEE DROP COLUMN Address CASCADE;**
- **ALTER TABLE COMPANY.DEPARTMENT ALTER COLUMN Mgr_ssn DROP DEFAULT;**
- **ALTER TABLE COMPANY.DEPARTMENT ALTER COLUMN Mgr_ssn SET DEFAULT '333445555';**
- **ALTER TABLE COMPANY.EMPLOYEE DROP CONSTRAINT EMPSUPERFK CASCADE;**

RV Institute of Technology and Management®

CHAPTER 20

Introduction to Transaction Processing Concepts and Theory

RV Institute of Technology and Management®

Introduction

- Transaction
  - Describes local unit of database processing
- Transaction processing systems
  - Systems with large databases and hundreds of concurrent users
  - Require high availability and fast response time

20.1 Introduction to Transaction Processing

- Single-user DBMS
  - At most one user at a time can use the system
  - Example: home computer
- Multiuser DBMS
  - Many users can access the system (database) concurrently
  - Example: airline reservations system

RV Institute of Technology and Management®

Introduction to Transaction Processing (cont'd.)

- Multiprogramming
  - Allows operating system to execute multiple processes concurrently
  - Executes commands from one process, then suspends that process and executes commands from another process, etc.

Introduction to Transaction Processing (cont'd.)

- Interleaved processing
- Parallel processing
  - Processes C and D in figure below

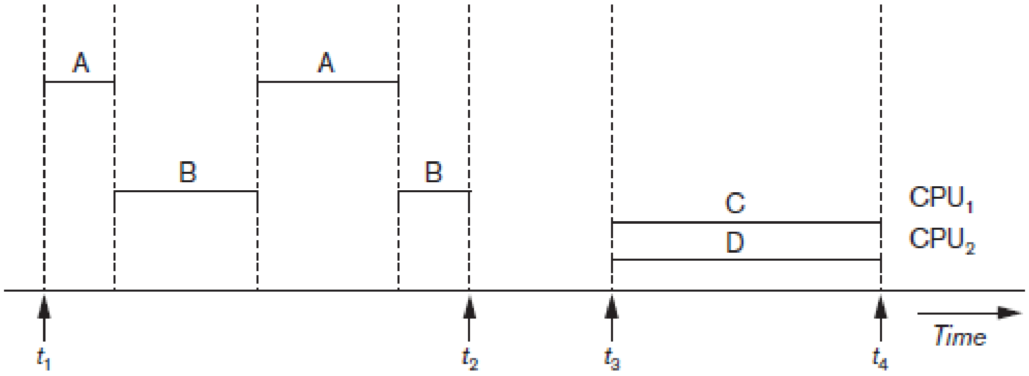


Figure 20.1 Interleaved processing versus parallel processing of concurrent transactions

Transactions

- Transaction: an executing program
  - Forms logical unit of database processing
- Begin and end transaction statements
  - Specify transaction boundaries
- Read-only transaction
- Read-write transaction

Database Items

- Database represented as collection of named data items
- Size of a data item called its **granularity**
- Data item
  - Record
  - Disk block
  - Attribute value of a record
- Transaction processing concepts independent of item granularity

Read and Write Operations

- read_item(X)
  - Reads a database item named X into a program variable named X
  - Process includes finding the address of the disk block, and copying **to and from a memory buffer**
- write_item(X)
  - Writes the value of program variable X into the database item named X
  - Process includes **finding the address** of the disk block, copying to and from a memory buffer, and storing the updated disk block back to disk

Read and Write Operations (cont’d.)

- Read set of a transaction
  - Set of all items read
- Write set of a transaction
  - Set of all items written

(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);

Figure 20.2 Two sample transactions (a) Transaction *T*1 (b) Transaction *T*2

Figure 20.2(a) shows a transaction *T1* that *transfers N reservations from one flight whose number of reserved seats is stored* in the database item named *X to another flight whose number of reserved seats is* stored in the database item named *Y*

Figure 20.2(b) shows a simpler transaction *T2* that just *reserves M seats on the first flight (X) referenced in transaction T1.*

DBMS Buffers

- DBMS will maintain several main memory data buffers in the database cache
- When buffers are occupied, a **buffer replacement policy** is used to choose which buffer will be replaced
  - Example policy: least recently used

RV Institute of Technology and Management®

Concurrency Control

- Transactions submitted by various users may execute concurrently
  - Access and update the same database items
  - Some form of concurrency control is needed
- The lost update problem
  - Occurs when two transactions that access the same database items have operations interleaved
  - Results in incorrect value of some database items

The Lost Update Problem

(a)

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);	

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

Figure 20.3 Some problems that occur when concurrent execution is uncontrolled (a) The lost update problem

The Temporary Update(Dirty read) Problem

(b)

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);	

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.

Figure 20.3 (cont'd.) Some problems that occur when concurrent execution is uncontrolled (b) The temporary update problem

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*).

Figure 20.3 (cont'd.) Some problems that occur when concurrent execution is uncontrolled (c) The incorrect summary problem

The 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

Why Recovery is Needed

- Committed transaction
  - Effect recorded permanently in the database
- Aborted transaction
  - Does not affect the database
- Types of transaction failures
  - Computer failure (system crash)
  - Transaction or system error
  - Local errors or exception conditions detected by the transaction

Why Recovery is Needed (cont'd.)

- Types of transaction failures (cont'd.)
  - Concurrency control enforcement
  - Disk failure
  - Physical problems or catastrophes
- System must keep sufficient information to recover quickly from the failure
  - Disk failure or other catastrophes have long recovery times

20.2 Transaction and System Concepts

- System must keep track of when each transaction starts, terminates, commits, and/or aborts
  - BEGIN_TRANSACTION (b)
  - READ or WRITE (r or w)
  - END_TRANSACTION (e)
  - COMMIT_TRANSACTION (c)
  - ROLLBACK (or ABORT) (a)

Transaction and System Concepts (cont'd.)

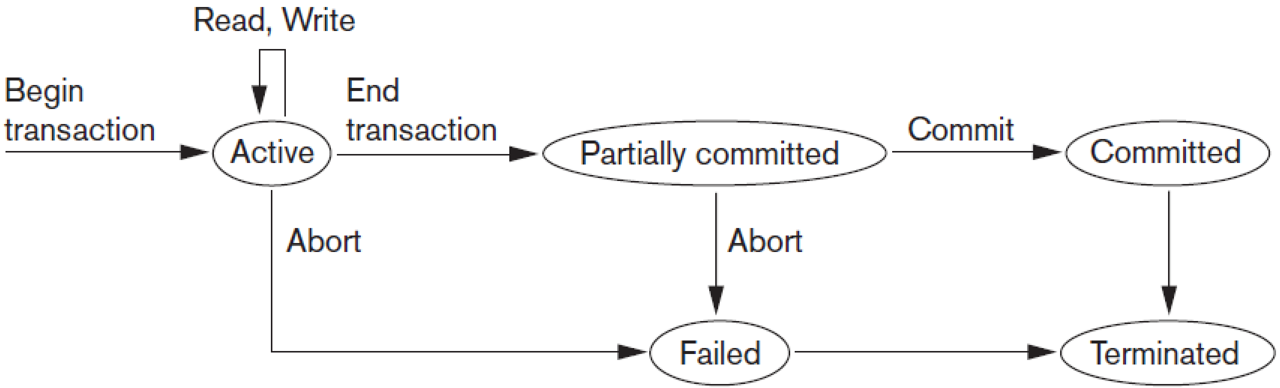


Figure 20.4 State transition diagram illustrating the states for transaction execution

The System Log

- System log keeps track of transaction operations
- Sequential, append-only file
- Not affected by failure (except disk or catastrophic failure)
- Log buffer
  - Main memory buffer
  - When full, appended to end of log file on disk
- Log file is backed up periodically
- Undo and redo operations based on log possible

RV Institute of Technology and Management®

The System Log - example

1. **[start_transaction**, ***T]****. Indicates that transaction T has started execution.*
2. **[write_item**, ***T, X, old_value, new_value]****. Indicates that transaction T has* changed the value of database item *X from old_value to new_value.*
3. **[read_item**, ***T, X]****. Indicates that transaction T has read the value of database* item *X.*
4. **[commit**, ***T]****. Indicates that transaction T has completed successfully, and affirms* that its effect can be committed (recorded permanently) to the database.
5. **[abort**, ***T]****. Indicates that transaction T has been aborted.*

Commit Point of a Transaction

- Occurs when all operations that access the database have completed successfully
  - And effect of operations recorded in the log
- Transaction writes a commit record into the log
  - If system failure occurs, can search for transactions with recorded start_transaction but no commit record
- Force-writing the log buffer to disk
  - Writing log buffer to disk before transaction reaches commit point

DBMS-Specific Buffer Replacement Policies

- Page replacement policy
  - Selects particular buffers to be replaced when all are full
- Domain separation (DS) method
  - Each domain handles one type of disk pages
    - Index pages
    - Data file pages
    - Log file pages
  - Number of available buffers for each domain is predetermined

RV Institute of Technology and Management®

DBMS-Specific Buffer Replacement Policies (cont'd.)

- Hot set method
  - Useful in queries that scan a set of pages repeatedly (nested loops)
  - Does not replace the set in the buffers until processing is completed
- The DBMIN method
  - Predetermines the pattern of page references for each algorithm for a particular type of database operation
    - Calculates locality set using query locality set model (QLSM)- buffers to file instance

20.3 Desirable Properties of Transactions

- ACID properties
  - Atomicity
    - Transaction performed in its entirety or not at all
  - Consistency preservation
    - Takes database from one consistent state to another
  - Isolation
    - Not interfered with by other transactions
  - Durability or permanency
    - Changes must persist in the database

Desirable Properties of Transactions (cont'd.)

- 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 - transaction sees the data items that it reads based on the committed values of the items in the *database snapshot (or database state) when the transaction starts.*

20.4 Characterizing Schedules Based on Recoverability

- Schedule or history
  - Order of execution of operations from all transactions
  - Operations from different transactions can be interleaved in the schedule
- Total ordering of operations in a schedule
  - For any two operations in the schedule, one must occur before the other
  - *Sa: r1(X); r2(X); w1(X); r1(Y); w2(X); w1(Y);*
  - *Sb: r1(X); w1(X); r2(X); w2(X); r1(Y); a1;*

RV Institute of Technology and Management®

Characterizing Schedules Based on Recoverability (cont'd.)

- Two **conflicting operations** in a schedule
  - Operations belong to different transactions
  - Operations access the same item X
  - At least one of the operations is a write_item(X)
- Two operations ***conflict*** if changing their order results in a different outcome
- Read-write conflict - *r1(X);w2(X) to w2(X); r1(X)*
- Write-write conflict - *w1(X);w2(X) to w2(X); w1(X)*
- *Complete Schedule of n transactions: same operations, same relative order, Conflicting -one must occur before other*

Characterizing Schedules Based on Recoverability (cont'd.)

- Recoverable schedules
  - Recovery is possible
- Nonrecoverable schedules should not be permitted by the DBMS –

Sc: r1(X); w1(X); r2(X); r1(Y); w2(X); c2; a1;

- No committed transaction ever needs to be rolled back
- Cascading rollback may occur in some recoverable schedules
  - Uncommitted transaction may need to be rolled back

Characterizing Schedules Based on Recoverability (cont'd.)

- Cascadeless schedule - if every transaction reads from committed transactions
  - Avoids cascading rollback
- Strict schedule
  - Transactions can neither read nor write an item X until the last transaction that wrote X has committed or aborted
  - Simpler recovery process
    - Restore the before image

20.5 Characterizing Schedules Based on Serializability

- Serializable schedules
  - Always considered to be correct when concurrent transactions are executing
  - Places simultaneous transactions in series
    - Transaction T_1 before T_2, or vice versa

RV Insti
and Ma

(a)

Time ↓ 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);

Schedule A

(b)

Time ↓ 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);	

Schedule B

(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

Figure 20.5 Examples of serial and nonserial schedules involving transactions *T*1 and *T*2 (a) Serial schedule A: *T*1 followed by *T*2 (b) Serial schedule B: *T*2 followed by *T*1 (c) Two nonserial schedules C and D with interleaving of operations

Characterizing Schedules Based on Serializability (cont'd.)

- 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 of *n* transactions
  - Equivalent to some serial schedule of same *n* transactions

Characterizing Schedules Based on Serializability (cont'd.)

- Result equivalent schedules
  - Produce the same final state of the database
    - May be accidental
  - Cannot be used alone to define equivalence of schedules

S_1

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

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

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

Characterizing Schedules Based on Serializability (cont'd.)

- Conflict equivalence
  - Relative order of any two conflicting operations is the same in both schedules
- Serializable schedules
  - Schedule S is serializable if it is conflict equivalent to some serial schedule S'.

Characterizing Schedules Based on Serializability (cont'd.)

- Testing for serializability of a schedule

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.

Algorithm 20.1 Testing conflict serializability of a schedule S

Characterizing Schedules Based on Serializability (cont'd.)

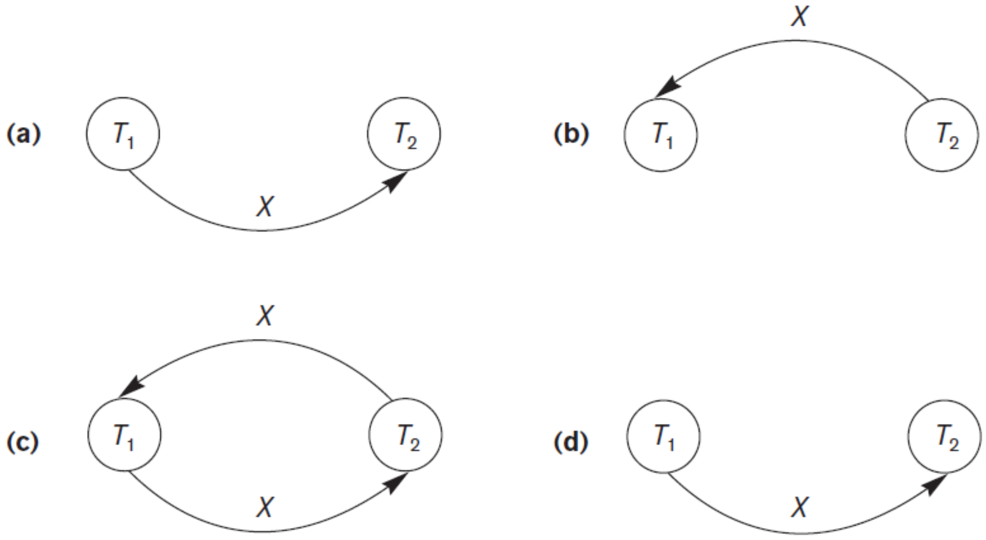


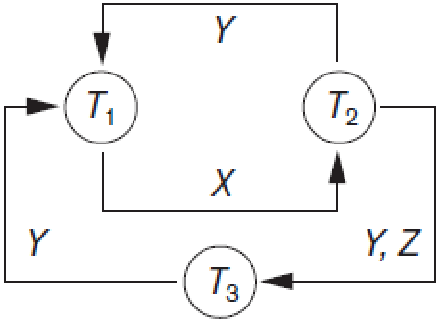
Figure 20.7 Constructing the precedence graphs for schedules A to D from Figure 20.5 to test for conflict serializability (a) Precedence graph for serial schedule A (b) Precedence graph for serial schedule B (c) Precedence graph for schedule C (not serializable) (d) Precedence graph for schedule D (serializable, equivalent to schedule A)

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 $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$

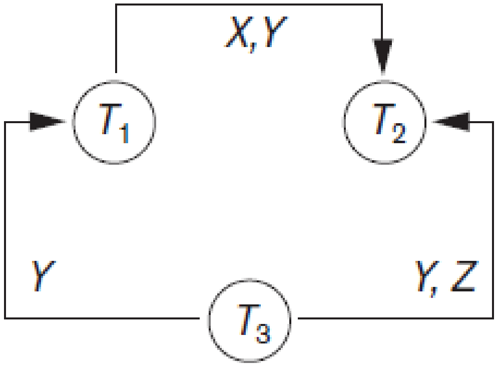
Cycle $X(T_1 \rightarrow T_2), YZ\ (T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

RV Institute of Technology and Management®

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);	

Time

Schedule F



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

How Serializability is Used for Concurrency Control

- Being serializable is different from being serial
- Serializable schedule gives benefit of concurrent execution
  - Without giving up any correctness
- Difficult to test for serializability in practice
  - 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

RV Institute of Technology and Management®

View Equivalence and View Serializability

- View equivalence of two schedules
  - 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
  - Read operations said to see the same view in both schedules
- View serializable schedule
  - View equivalent to a serial schedule

View Equivalence and View Serializability (cont'd.)

- Conflict serializability similar to view serializability if constrained write assumption (no blind writes) applies
- Unconstrained write assumption
  - Value written by an operation can be independent of its old value
- Debit-credit transactions
  - Less-stringent conditions than conflict serializability or view serializability

20.6 Transaction Support in SQL

- No explicit Begin_Transaction statement
- Every transaction must have an explicit end statement
  - COMMIT
  - ROLLBACK
- Access mode is READ ONLY or READ WRITE
- Diagnostic area size option
  - Integer value indicating number of conditions held simultaneously in the diagnostic area

Transaction Support in SQL (cont'd.)

- Isolation level option
  - Dirty read
  - Nonrepeatable read
  - Phantoms – T1 querying, T2 inserts new row, T1 doesn't have the new row

	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

Table 20.1 Possible violations based on isolation levels as defined in SQL

Transaction Support in SQL (cont’d.)

- Snapshot isolation
  - Used in some commercial DBMSs
  - Transaction sees data items that it reads based on the committed values of the items in the database snapshot when transaction starts
  - Ensures phantom record problem will not occur

20.7 Summary

- Single and multiuser database transactions
- Uncontrolled execution of concurrent transactions
- System log
- Failure recovery
- Committed transaction
- Schedule (history) defines execution sequence
  - Schedule recoverability
  - Schedule equivalence
- Serializability of schedules