

Database Management Systems (BCS403) - 2023-24 - Module 4

Dr. Narender M
Department of CS&E
The National Institute of Engineering

Topics

SQL Advanced Queries

- More complex SQL retrieval queries
- Specifying constraints as assertions and action triggers
- Views in SQL

Transaction Processing

- 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

Chapter 1: SQL Advanced Queries

More complex SQL retrieval queries

Comparisons Involving NULL and Three-Valued Logic

- **Meanings of NULL**

1. Unknown value. A person's date of birth is not known, so it is represented by NULL in the database.
2. Unavailable or withheld value. A person has a home phone but does not want it to be listed, so it is withheld and represented as NULL in the database.
3. Not applicable attribute. An attribute LastCollegeDegree would be NULL for a person who has no college degrees.

More complex SQL retrieval queries

-
- Each individual NULL value is considered to be different from every other NULL value in the various database records.
 - When a record with NULL in one of its attributes is involved in a comparison operation, the result is considered to be UNKNOWN (it may be TRUE or it may be FALSE).
 - Hence, SQL uses a three-valued logic with values TRUE, FALSE, and UNKNOWN instead of the standard two-valued (Boolean) logic with values TRUE or FALSE.
 - It is therefore necessary to define the results (or truth values) of three-valued logical expressions when the logical connectives AND, OR, and NOT are used.

More complex SQL retrieval queries

Table 7.1 Logical Connectives in Three-Valued Logic

(a)	AND	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	FALSE	UNKNOWN
	FALSE	FALSE	FALSE	FALSE
	UNKNOWN	UNKNOWN	FALSE	UNKNOWN
(b)	OR	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE	UNKNOWN
	UNKNOWN	TRUE	UNKNOWN	UNKNOWN
(c)	NOT			
	TRUE	FALSE		
	FALSE	TRUE		
	UNKNOWN	UNKNOWN		

More complex SQL retrieval queries

-
- In queries, the general rule is that only those combinations of tuples that evaluate the logical expression in the WHERE clause of the query to TRUE are selected.
 - Tuple combinations that evaluate to FALSE or UNKNOWN are not selected.
 - SQL allows queries that check whether an attribute value is NULL. It uses the comparison operators IS or IS NOT.

Query 18. Retrieve the names of all employees who do not have supervisors.

Q18: **SELECT** Fname, Lname
 FROM EMPLOYEE
 WHERE Super_ssn **IS NULL**;

More complex SQL
retrieval queries

- This is because SQL considers each NULL value as being distinct from every other NULL value, so equality comparison is not appropriate.
- It follows that when a join condition is specified, tuples with NULL values for the join attributes are not included in the result.

More complex SQL retrieval queries

Nested Queries, Tuples, and Set/Multiset Comparisons

- Nested queries are complete select-from-where blocks within another SQL query.
- That other query is called the outer query.
- These nested queries can also appear in the WHERE clause or the FROM clause or the SELECT clause or other SQL clauses as needed.

Q4A: **SELECT** **DISTINCT** Pnumber
 FROM PROJECT
 WHERE Pnumber IN
 (**SELECT** Pnumber
 FROM PROJECT, DEPARTMENT, EMPLOYEE
 WHERE Dnum = Dnumber **AND**
 Mgr_ssn = Ssn **AND** Lname = 'Smith')

 OR
 Pnumber IN
 (**SELECT** Pno
 FROM WORKS_ON, EMPLOYEE
 WHERE Essn = Ssn **AND** Lname = 'Smith');

More complex SQL retrieval queries

- The first nested query selects the project numbers of projects that have an employee with last name 'Smith' involved as manager.
- The second nested query selects the project numbers of projects that have an employee with last name 'Smith' involved as worker.

More complex SQL retrieval queries

If a nested query returns a single attribute and a single tuple, the query result will be a single value.

In such cases, it is permissible to use = instead of IN for the comparison operator.

SQL allows the use of tuples of values in comparisons by placing them within parentheses.

Page 10 of 10

- This query will select the Essns of all employees who work the same (project, hours) combination on some project that employee 'John Smith' (whose Ssn = '123456789') works on.

```
SELECT DISTINCT Essn
FROM WORKS_ON
WHERE (Pno, Hours) IN ( SELECT Pno, Hours
                        FROM WORKS_ON
                        WHERE Essn = '123456789' );
```

More complex SQL retrieval queries

-
- In addition to the IN operator, other operators can be used to compare a single value v (typically an attribute name) to a set or multiset V (typically a nested query).
 - The $= ANY$ (or $= SOME$) operator returns TRUE if the value v is equal to some value in the set V and is hence equivalent to IN.
 - The two keywords ANY and SOME have the same effect. Other operators that can be combined with ANY (or SOME) include $>$, $>=$, $<$, $<=$, and $<>$.
 - The keyword ALL can also be combined with each of these operators. For example, the comparison condition $(v > ALL V)$ returns TRUE if the value v is greater than all the values in the set (or multiset) V .

More complex SQL
retrieval queries

- An example is the following query, which returns the names of employees whose salary is greater than the salary of all the employees in department 5:

```
SELECT      Lname, Fname
FROM        EMPLOYEE
WHERE       Salary > ALL      ( SELECT      Salary
                                FROM        EMPLOYEE
                                WHERE       Dno = 5 );
```

Query 16. Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

```
Q16:  SELECT  E.Fname, E.Lname
      FROM    EMPLOYEE AS E
      WHERE   E.Ssn IN ( SELECT  D.Essn
                        FROM    DEPENDENT AS D
                        WHERE   E.Fname = D.Dependent_name
                        AND E.Sex = D.Sex );
```

More complex SQL retrieval queries

- We can have several levels of nested queries and it might result into ambiguity among attribute names if attributes of the same name exist—one in a relation in the FROM clause of the outer query, and another in a relation in the FROM clause of the nested query.
- The rule is that a reference to an unqualified attribute refers to the relation declared in the innermost nested query.

More complex SQL retrieval queries

Correlated Nested Queries

- Whenever a condition in the WHERE clause of a nested query references some attribute of a relation declared in the outer query, the two queries are said to be correlated.
- In Nested Query, the Inner query runs first, and only once. Outer query is executed with the result from Inner query.
- Hence, the Inner query is used in execution of the Outer query.

More complex SQL retrieval queries

- In Nested Query, the Inner query runs first, and only once. Outer query is executed with the result from Inner query.
 - Hence, the Inner query is used in execution of the Outer query.
-
- In Correlated Query, Outer query executes first and for every Outer query row Inner query is executed.
 - Hence, the Inner query uses values from the Outer query.

More complex SQL retrieval queries

```
SELECT name
FROM employees
WHERE department_id = (
    SELECT id
    FROM departments
    WHERE department_name = 'Engineering'
);
```

Example for nested query

More complex SQL retrieval queries

```
SELECT e.name, (  
    SELECT d.department_name  
    FROM departments d  
    WHERE d.id = e.department_id  
) AS department_name  
FROM employees e;
```

Example for correlated nested query

More complex SQL retrieval queries

The EXISTS and UNIQUE Functions in SQL

- EXISTS and UNIQUE are Boolean functions that return TRUE or FALSE; hence, they can be used in a WHERE clause condition.
- The EXISTS function in SQL is used to check whether the result of a nested query is empty (contains no tuples) or not.
- The result of EXISTS is a Boolean value TRUE if the nested query result contains at least one tuple, or FALSE if the nested query result contains no tuples.

```
SELECT      E.Fname, E.Lname
FROM        EMPLOYEE AS E
WHERE       EXISTS ( SELECT      *
                      FROM        DEPENDENT AS D
                      WHERE       E.Ssn = D.Essn AND E.Sex = D.Sex
                      AND E.Fname = D.Dependent_name);
```

More complex SQL retrieval queries

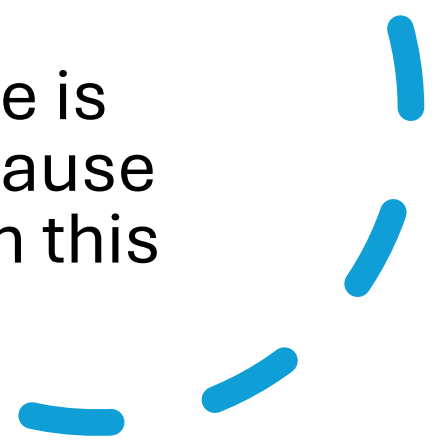
- EXISTS and NOT EXISTS are typically used in conjunction with a correlated nested query.
- For each EMPLOYEE tuple, evaluate the nested query, which retrieves all DEPENDENT tuples with the same Essn, Sex, and Dependent_name as the EMPLOYEE tuple; if at least one tuple EXISTS in the result of the nested query, then select that EMPLOYEE tuple.

Query 6. Retrieve the names of employees who have no dependents.

```
Q6:    SELECT    Fname, Lname
        FROM      EMPLOYEE
        WHERE     NOT EXISTS ( SELECT    *
                                FROM      DEPENDENT
                                WHERE     Ssn = Essn );
```

More complex
SQL retrieval
queries

- The correlated nested query retrieves all DEPENDENT tuples related to a particular EMPLOYEE tuple.
- If none exist, the EMPLOYEE tuple is selected because the WHERE-clause condition will evaluate to TRUE in this case.



Query 7. List the names of managers who have at least one dependent.

```
Q7:  SELECT      Fname, Lname
      FROM      EMPLOYEE
      WHERE      EXISTS ( SELECT      *
                           FROM      DEPENDENT
                           WHERE      Ssn = Essn )
      AND
      EXISTS ( SELECT      *
                FROM      DEPARTMENT
                WHERE      Ssn = Mgr_ssn );
```

More complex SQL retrieval queries

More complex SQL
retrieval queries

Explicit Sets and Renaming in SQL

- It is also possible to use an explicit set of values in the WHERE clause, rather than a nested query.
- Such a set is enclosed in parentheses in SQL.

Query 17. Retrieve the Social Security numbers of all employees who work on project numbers 1, 2, or 3.

```
Q17:      SELECT      DISTINCT Essn
          FROM        WORKS_ON
          WHERE       Pno IN (1, 2, 3);
```


More complex SQL retrieval queries

- It is possible to rename any attribute that appears in the result of a query by adding the qualifier AS followed by the desired new name.
- The AS construct can be used to alias both attribute and relation names in general, and it can be used in appropriate parts of a query.

```
SELECT      E.Lname AS Employee_name, S.Lname AS Supervisor_name
FROM        EMPLOYEE AS E, EMPLOYEE AS S
WHERE       E.Super_ssn = S.Ssn;
```

More complex SQL retrieval queries

Joined Tables in SQL and Outer Joins

- The concept of a joined table (or joined relation) was incorporated into SQL to permit users to specify a table resulting from a join operation in the FROM clause of a query.
- In a NATURAL JOIN on two relations R and S, no join condition is specified; an implicit EQUIJOIN condition for each pair of attributes with the same name from R and S is created.

```
SELECT      Fname, Lname, Address
FROM        (EMPLOYEE JOIN DEPARTMENT ON Dno = Dnumber)
WHERE       Dname = 'Research';
```

```
SELECT      Fname, Lname, Address
FROM        (EMPLOYEE NATURAL JOIN
              (DEPARTMENT AS DEPT (Dname, Dno, Mssn, Msdate)))
WHERE       Dname = 'Research';
```

- If the names of the join attributes are not the same in the base relations, it is possible to rename the attributes so that they match, and then to apply NATURAL JOIN.
- The default type of join in a joined table is called an inner join, where a tuple is included in the result only if a matching tuple exists in the other relation.

More complex SQL retrieval queries

More complex SQL retrieval queries

- If the user requires that all non-matching rows to be included, a different type of join called OUTER JOIN must be used explicitly.
- LEFT OUTER JOIN - every tuple in the left table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the right table.
- RIGHT OUTER JOIN - every tuple in the right table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the left table.
- If the join attributes have the same name, one can also specify the natural join variation of outer joins by using the keyword NATURAL before the operation (for example, NATURAL LEFT OUTER JOIN).

```
SELECT      Pnumber, Dnum, Lname, Address, Bdate
FROM        ((PROJECT JOIN DEPARTMENT ON Dnum = Dnumber)
             JOIN EMPLOYEE ON Mgr_ssn = Ssn)
WHERE       Plocation = 'Stafford';
```

- The keyword CROSS JOIN is used to specify the CARTESIAN PRODUCT operation, although this should be used only with the utmost care because it generates all possible tuple combinations.
- It is also possible to nest join specifications; that is, one of the tables in a join may itself be a joined table.
- This allows the specification of the join of three or more tables as a single joined table, which is called a multiway join.

More complex SQL retrieval queries

More complex SQL retrieval queries

Aggregate Functions in SQL

- Aggregate functions are used to summarize information from multiple tuples into a single-tuple summary.
- Grouping is used to create subgroups of tuples before summarization. Grouping and aggregation are required in many database applications.
- A number of built-in aggregate functions exist: COUNT, SUM, MAX, MIN, and AVG.
- The COUNT function returns the number of tuples or values as specified in a query.
- The functions SUM, MAX, MIN, and AVG can be applied to a set or multiset of numeric values.

Query 19. Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

Q19: **SELECT** **SUM** (Salary), **MAX** (Salary), **MIN** (Salary), **AVG** (Salary)
 FROM EMPLOYEE;

Q19A: **SELECT** **SUM** (Salary) **AS** Total_Sal, **MAX** (Salary) **AS** Highest_Sal,
 MIN (Salary) **AS** Lowest_Sal, **AVG** (Salary) **AS** Average_Sal
 FROM EMPLOYEE;

- These functions can be used in the SELECT clause or in a HAVING clause.
- The functions MAX and MIN can also be used with attributes that have nonnumeric domains if the domain values have a total ordering among one another.

More complex SQL retrieval queries

More complex SQL retrieval queries

Query 20. Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the average salary in this department.

```
Q20:      SELECT      SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)  
          FROM        (EMPLOYEE JOIN DEPARTMENT ON Dno = Dnumber)  
          WHERE       Dname = 'Research';
```

Queries 21 and 22. Retrieve the total number of employees in the company (Q21) and the number of employees in the 'Research' department (Q22).

```
Q21:      SELECT      COUNT (*)  
          FROM        EMPLOYEE;
```

```
Q22:      SELECT      COUNT (*)  
          FROM        EMPLOYEE, DEPARTMENT  
          WHERE       DNO = DNUMBER AND DNAME = 'Research';
```


Query 23. Count the number of distinct salary values in the database.

Q23: **SELECT** **COUNT (DISTINCT Salary)**
 FROM **EMPLOYEE;**

- If we write COUNT(SALARY) instead of COUNT(DISTINCT SALARY), then duplicate values will not be eliminated. However, any tuples with NULL for SALARY will not be counted.
- NULL values are discarded when aggregate functions are applied to a particular column (attribute); the only exception is for COUNT(*) because tuples instead of values are counted.

More complex SQL retrieval queries

More complex SQL retrieval queries

- When an aggregate function is applied to a collection of values, NULLs are removed from the collection before the calculation.
- If the collection becomes empty because all values are NULL, the aggregate function will return NULL.
- We can specify a correlated nested query with an aggregate function, and then use the nested query in the WHERE clause of an outer query.

SELECT	Lname, Fname	
FROM	EMPLOYEE	
WHERE	(SELECT	COUNT (*)
	FROM	DEPENDENT
	WHERE	Ssn = Essn) >= 2;

- For example, to retrieve the names of all employees who have two or more dependents.

More complex SQL retrieval queries

More complex SQL retrieval queries

—

Grouping: The GROUP BY and HAVING Clauses

- In many cases we want to apply the aggregate functions to subgroups of tuples in a relation, where the subgroups are based on some attribute values.
- In these cases, we need to partition the relation into nonoverlapping subsets (or groups) of tuples.
- Each group (partition) will consist of the tuples that have the same value of some attribute(s), called the grouping attribute(s).

Query 24. For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
Q24:      SELECT      Dno, COUNT (*), AVG (Salary)
           FROM        EMPLOYEE
           GROUP BY    Dno;
```

- The GROUP BY clause specifies the grouping attributes, which should also appear in the SELECT clause, so that the value resulting from applying each aggregate function to a group of tuples appears along with the value of the grouping attribute(s).

More complex SQL retrieval queries

Query 25. For each project, retrieve the project number, the project name, and the number of employees who work on that project.

```
Q25:      SELECT      Pnumber, Pname, COUNT (*)  
           FROM        PROJECT, WORKS_ON  
           WHERE       Pnumber = Pno  
           GROUP BY   Pnumber, Pname;
```

- If NULLs exist in the grouping attribute, then a separate group is created for all tuples with a NULL value in the grouping attribute.
- We can use a join condition in conjunction with GROUP BY.

More complex SQL retrieval queries

More complex SQL retrieval queries

- Sometimes we want to retrieve the values of these functions only for groups that satisfy certain conditions.
- SQL provides a HAVING clause, which can appear in conjunction with a GROUP BY clause, for this purpose.
- HAVING provides a condition on the summary information regarding the group of tuples associated with each value of the grouping attributes.
- Only the groups that satisfy the condition are retrieved in the result of the query.

Query 26. For each project *on which more than two employees work*, retrieve the project number, the project name, and the number of employees who work on the project.

```
Q26:    SELECT    Pnumber, Pname, COUNT (*)
        FROM      PROJECT, WORKS_ON
        WHERE     Pnumber = Pno
        GROUP BY  Pnumber, Pname
        HAVING    COUNT (*) > 2;
```

More complex SQL retrieval queries

- Notice that although selection conditions in the WHERE clause limit the tuples to which functions are applied, the HAVING clause serves to choose whole groups.

More complex SQL retrieval queries

Other SQL Constructs: WITH and CASE

- The WITH clause allows a user to define a table that will only be used in a particular query.
- This construct was introduced as a convenience in SQL:99 and may not be available in all SQL based DBMSs.
- A temporary table BIG_DEPTS is defined which result holds the Dno's of departments with more than five employees and then it is used in the subsequent query.
- Once this query is executed, the temporary table BIGDEPTS is discarded.

```
WITH          BIGDEPTS (Dno) AS
              ( SELECT      Dno
                FROM        EMPLOYEE
                GROUP BY    Dno
                HAVING      COUNT (*) > 5)

SELECT
FROM
WHERE
GROUP BY     Dno, COUNT (*)
EMPLOYEE
Salary>40000 AND Dno IN BIGDEPTS
Dno;
```

More complex SQL retrieval queries

- SQL also has a CASE construct, which can be used when a value can be different based on certain conditions.
- This can be used in any part of an SQL query where a value is expected, including when querying, inserting or updating tuples.
- The salary raise value is determined through the CASE construct based on the department number for which each employee works.

```
UPDATE EMPLOYEE
SET Salary =
CASE
    WHEN Dno = 5 THEN Salary + 2000
    WHEN Dno = 4 THEN Salary + 1500
    WHEN Dno = 1 THEN Salary + 3000
    ELSE Salary + 0 ;
```

More complex SQL retrieval queries

Recursive Queries in SQL

- An example of a recursive relationship between tuples of the same type is the relationship between an employee and a supervisor.
- An example of a recursive operation is to retrieve all supervisees of a supervisory employee *e* at all levels.

EMPLOYEE

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
-------	-------	-------	------------	-------	---------	-----	--------	-----------	-----

More complex SQL retrieval queries

```
WITH RECURSIVE SUP_EMP (SupSsn, EmpSsn) AS
( SELECT
  FROM
  UNION
  SELECT
  FROM
  WHERE
  SELECT*
  FROM
  SUP_EMP;
```

- We are defining a view SUP_EMP that will hold the result of the recursive query. The view is initially empty.
- It is first loaded with the first level (supervisor, supervisee) Ssn combinations via the first part, which is called the base query.
- This will be combined via UNION with each successive level of supervisees through the second part, where the view contents are joined again with the base values to get the second level combinations, which are UNIONed with the first level.
- This is repeated with successive levels until a fixed point is reached, where no more tuples are added to the view.

Specifying Constraints as Assertions and Actions as Triggers

Specifying General Constraints as Assertions in SQL

- CREATE ASSERTION can be used to specify additional types of constraints that are outside the scope of the built-in relational model constraints.
- Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query.
- The salary of an employee must not be greater than the salary of the manager of the department that the employee works for.

Specifying Constraints as Assertions and Actions as Triggers

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS ( SELECT *
                     FROM   EMPLOYEE E, EMPLOYEE M,
                     DEPARTMENT D
                     WHERE  E.Salary > M.Salary
                     AND    E.Dno = D.Dnumber
                     AND    D.Mgr_ssn = M.Ssn ) );
```

- The constraint name SALARY_CONSTRAINT is followed by the keyword CHECK, which is followed by a condition in parentheses that must hold true on every database state for the assertion to be satisfied.
- The constraint name can be used later to disable the constraint or to modify or drop it. The DBMS is responsible for ensuring that the condition is not violated.

Specifying Constraints as Assertions and Actions as Triggers

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS ( SELECT *
                     FROM   EMPLOYEE E, EMPLOYEE M,
                          DEPARTMENT D
                     WHERE  E.Salary>M.Salary
                          AND   E.Dno = D.Dnumber
                          AND   D.Mgr_ssn = M.Ssn ) );
```

- Any WHERE clause condition can be used, but many constraints can be specified using the EXISTS and NOT EXISTS style of SQL conditions.
- Whenever some tuples in the database cause the condition of an ASSERTION statement to evaluate to FALSE, the constraint is violated.
- The constraint is satisfied by a database state if no combination of tuples in that database state violates the constraint.

Specifying Constraints as Assertions and Actions as Triggers

- The basic technique for writing such assertions is to specify a query that selects any tuples that violate the desired condition.
- By including this query inside a NOT EXISTS clause, the assertion will specify that the result of this query must be empty so that the condition will always be TRUE.
- Note that the CHECK clause and constraint condition can also be used to specify constraints on individual attributes and domains and on individual tuples.
- The schema designer should use CHECK on attributes, domains, and tuples only when he or she is sure that the constraint can only be violated by insertion or updating of tuples.

Specifying Constraints as Assertions and Actions as Triggers

Introduction to Triggers in SQL

- In many cases it is convenient to specify the type of action to be taken when certain events occur and when certain conditions are satisfied.
- For example, it may be useful to specify a condition that, if violated, causes some user to be informed of the violation.
- A manager may want to be informed if an employee's travel expenses exceed a certain limit by receiving a message whenever this occurs.
- The action that the DBMS must take in this case is to send an appropriate message to that user. The condition is thus used to monitor the database.

Specifying Constraints as Assertions and Actions as Triggers

- The CREATE TRIGGER statement is used to implement such actions in SQL.
- Syntax:
 create trigger [trigger_name]
 [before | after]
 {insert | update [of columns] |
 delete}
 on [table_name]
 [for each row]
 [trigger_body]

Specifying Constraints as Assertions and Actions as Triggers

- Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor in the COMPANY database.
- Several events can trigger this rule: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor.
- Suppose that the action to take would be to call an external stored procedure SALARY_VIOLATION, which will notify the supervisor.

Specifying Constraints as Assertions and Actions as Triggers

```
CREATE TRIGGER SALARY_VIOLATION
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 );
```

Specifying Constraints as Assertions and Actions as Triggers

- A typical trigger which is regarded as an ECA (Event, Condition, Action) rule has three components:
- **The event(s):** These are usually database update operations that are explicitly applied to the database.
- In this example the events are: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor.

Specifying Constraints as Assertions and Actions as Triggers

- The **condition** that determines whether the rule action should be executed: Once the triggering event has occurred, an optional condition may be evaluated.
- If no condition is specified, the action will be executed once the event occurs.
- If a condition is specified, it is first evaluated, and only if it evaluates to true will the rule action be executed.

Specifying Constraints as Assertions and Actions as Triggers

- The **action** to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed.
- Another Example:
 - Create a row level trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old & new Salary.
 - CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)

Specifying Constraints as Assertions and Actions as Triggers

```
CREATE TABLE CUSTOMERS (  
    ID INT PRIMARY KEY AUTO_INCREMENT,  
    NAME VARCHAR (255),  
    AGE INT,  
    ADDRESS VARCHAR (255),  
    SALARY DECIMAL (10, 2)  
);
```



Specifying Constraints as Assertions and Actions as Triggers

```
-- INSERT TRIGGER  
DELIMITER //  
CREATE TRIGGER  
after_insert_salary_difference  
AFTER INSERT ON CUSTOMERS  
FOR EACH ROW  
BEGIN  
    SET @my_sal_diff = CONCAT ('salary  
inserted is ', NEW.SALARY);  
END;//  
DELIMITER ;
```

Specifying Constraints as Assertions and Actions as Triggers

```
INSERT INTO CUSTOMERS (NAME, AGE,  
ADDRESS, SALARY) VALUES ('ABC', 35, '123  
Main St', 50000.00);
```

```
SELECT @my_sal_diff AS SAL_DIFF;
```



Specifying Constraints as Assertions and Actions as Triggers

```
-- UPDATE TRIGGER
DELIMITER //
CREATE TRIGGER after_update_salary_difference
AFTER UPDATE ON CUSTOMERS
FOR EACH ROW
BEGIN
    DECLARE old_salary DECIMAL (10, 2);
    DECLARE new_salary DECIMAL (10, 2);
    SET old_salary = OLD.SALARY;
    SET new_salary = NEW.SALARY;
    SET @my_sal_diff = CONCAT ('salary
difference after update is ', NEW.SALARY -
OLD.SALARY);
END;//
DELIMITER ;
```

Specifying Constraints as Assertions and Actions as Triggers

```
UPDATE CUSTOMERS  
SET SALARY = 55000.00  
WHERE ID = 1;
```

```
SELECT @my_sal_diff AS SAL_DIFF;
```



Views in SQL

Concept of a View in SQL

- A view in SQL terminology is a single table that is derived from other tables.
- These other tables can be base tables or previously defined views. A view does not necessarily exist in physical form.
- It is a virtual table, in contrast to base tables, whose tuples are always physically stored in the database.
- This limits the possible update operations that can be applied to views, but it does not provide any limitations on querying a view.

Views in SQL

- We can think of a view as a way of specifying a table that we need to reference frequently, even though it may not exist physically.
- Rather than having to specify the join of the three tables EMPLOYEE, WORKS_ON, and PROJECT every time, we can define a view that is specified as the result of these joins.

Views in SQL

Specification of Views in SQL

- The command to specify a view is `CREATE VIEW`.
- The view is given a (virtual) table name (or view name), a list of attribute names, and a query to specify the contents of the view.
- If none of the view attributes results from applying functions or arithmetic operations, we do not have to specify new attribute names for the view, since they would be the same as the names of the attributes of the defining tables in the default case.

Views in SQL

```
CREATE VIEW WORKS_ON1
AS SELECT
  FROM
  WHERE
    Fname, Lname, Pname, Hours
  EMPLOYEE, PROJECT, WORKS_ON
  Ssn = Essn AND Pno = Pnumber;
```

WORKS_ON1

Fname	Lname	Pname	Hours
-------	-------	-------	-------

```
CREATE VIEW DEPT_INFO(Dept_name, No_of_emps, Total_sal)
AS SELECT
  FROM
  WHERE
  GROUP BY
    Dname, COUNT (*), SUM (Salary)
  DEPARTMENT, EMPLOYEE
  Dnumber = Dno
  Dname;
```

DEPT_INFO

Dept_name	No_of_emps	Total_sal
-----------	------------	-----------


```
SELECT      Fname, Lname
FROM        WORKS_ON1
WHERE       Pname = 'ProductX';
```

- We can now specify SQL queries on a view—or virtual table—in the same way we specify queries involving base tables.
- For example, to retrieve the last name and first name of all employees who work on the 'ProductX' project, we can utilize the WORKS_ON1 view.
- The same query would require the specification of two joins if specified on the base relations directly.

Views in SQL

Views in SQL

- One of the main advantages of a view is to simplify the specification of certain queries.
- Views are also used as a security and authorization mechanism.
- A view is supposed to be always up-to-date; if we modify the tuples in the base tables on which the view is defined, the view must automatically reflect these changes.
- Hence, the view does not have to be realized or materialized at the time of view definition but rather at the time when we specify a query on the view.

Views in SQL

- If we do not need a view anymore, we can use the DROP VIEW command to dispose of it.

```
DROP VIEW WORKS_ON1;
```

Views in SQL

View Implementation, View Update, and Inline Views

- The problem of how a DBMS can efficiently implement a view for efficient querying is complex.
- Two main approaches have been suggested
 - Query modification
 - View materialization
- Query modification involves modifying or transforming the view query into a query on the underlying base tables.
- The disadvantage of this approach is that it is inefficient for views defined via complex queries that are time-consuming to execute, especially if multiple view queries are going to be applied to the same view within a short period of time.

Views in SQL

- View materialization involves physically creating a temporary or permanent view table when the view is first queried or created and keeping that table on the assumption that other queries on the view will follow.
- Update the view table when the base tables are updated.
- Incremental update is used for this purpose, where the DBMS can determine what new tuples must be inserted, deleted, or modified in a materialized view table.
- The view is kept as a materialized table if it is being queried.

Views in SQL

- If the view is not queried for a certain period, the system may then automatically remove the physical table and recompute it from scratch when future queries reference the view.
- Different strategies as to when a materialized view is updated are possible.
- The immediate update strategy updates a view as soon as the base tables are changed.
- The lazy update strategy updates the view when needed by a view query.
- The periodic update strategy updates the view periodically (in the latter strategy, a view query may get a result that is not up-to-date).

Views in SQL

- Issuing an INSERT, DELETE, or UPDATE command on a view table is in many cases not possible.
- An update on a view defined on a single table without any aggregate functions can be mapped to an update on the underlying base table under certain conditions.
- For a view involving joins, an update operation may be mapped to update operations on the underlying base relations in multiple ways.
- Hence, it is often not possible for the DBMS to determine which of the updates is intended.

Views in SQL

- Some view updates may not make much sense; for example, modifying the Total_sal attribute of the DEPT_INFO view.
- A view update is feasible when only one possible update on the base relations can accomplish the desired update operation on the view.
- Whenever an update on the view can be mapped to more than one update on the underlying base relations, it is usually not permitted.

Views in SQL

- A view with a single defining table is updatable if the view attributes contain the primary key of the base relation, as well as all attributes with the NOT NULL constraint that do not have default values specified.
- Views defined on multiple tables using joins are generally not updatable.
- Views defined using grouping and aggregate functions are not updatable.

Views in SQL

- The WITH CHECK OPTION should be added at the end of the view definition if a view is to be updated by INSERT, DELETE, or UPDATE statements.
- This allows the system to reject operations that violate the SQL rules for view updates.
- It is also possible to define a view table in the FROM clause of an SQL query. This is known as an in-line view.

Views in SQL

Views as Authorization Mechanisms

- Views can be used to hide certain attributes or tuples from unauthorized users.
- Suppose a certain user is only allowed to see employee information for employees who work for department 5; then we can create the following view DEPT5EMP and grant the user the privilege to query the view but not the base table EMPLOYEE itself.
- This user will only be able to retrieve employee information for employee tuples whose Dno = 5 and will not be able to see other employee tuples when the view is queried.

```
CREATE VIEW DEPT5EMP AS
SELECT *
FROM EMPLOYEE
WHERE Dno = 5;
```

```
CREATE VIEW BASIC_EMP_DATA AS
SELECT Fname, Lname, Address
FROM EMPLOYEE;
```

- A view can restrict a user to only see certain columns; for example, only the first name, last name, and address of an employee may be visible.
- By creating an appropriate view and granting certain users access to the view and not the base tables, they would be restricted to retrieving only the data specified in the view.

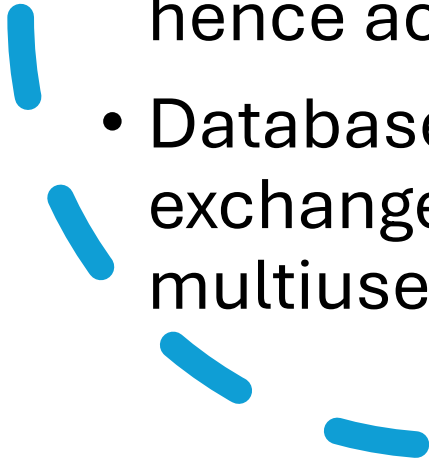
Views in SQL

Chapter 2: Transaction Processing







Introduction to Transaction Processing

Single-User versus Multiuser Systems

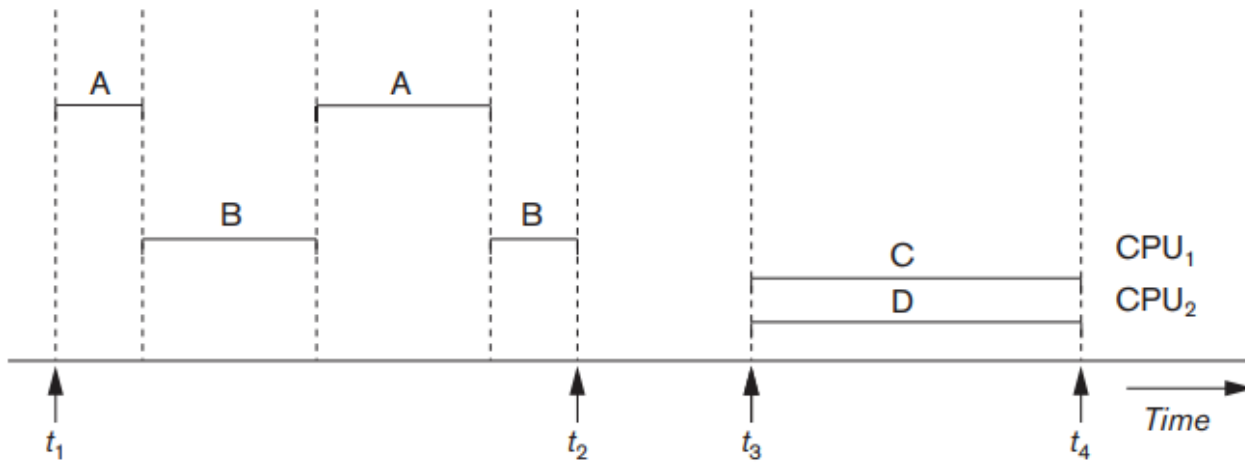
- One criterion for classifying a database system is according to the number of users who can use the system concurrently.
 - A DBMS is single-user if at most one user at a time can use the system, and it is multiuser if many users can use the system - and hence access the database - concurrently.
 - Database systems used in banks, insurance agencies, stock exchanges, supermarkets, and many other applications are multiuser systems.
- 



Introduction to Transaction Processing

- Multiple users can access databases simultaneously because of the concept of multiprogramming, which allows the operating system of the computer to execute multiple programs at the same time.
 - A single central processing unit (CPU) can only execute at most one process at a time.
 - Multiprogramming operating systems execute some commands from one process, then suspend that process and execute some commands from the next process, and so on.
 - A process is resumed at the point where it was suspended whenever it gets its turn to use the CPU again. Hence, concurrent execution of processes is actually interleaved.
- 
- 
- 
- 

Introduction to Transaction Processing



- A and B, executing concurrently in an interleaved fashion.
- If the computer system has multiple hardware processors (CPUs), parallel processing of multiple processes is possible, as illustrated by processes C and D.

Introduction to Transaction Processing

Transactions, Database Items, Read and Write Operations, and DBMS Buffers

- A transaction is an executing program that forms a logical unit of database processing.
- A transaction includes one or more database access operations—these can include insertion, deletion, modification, or retrieval operations.
- The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL.
- One way of specifying the transaction boundaries is by specifying explicit begin transaction and end transaction statements.

Introduction to Transaction Processing

- If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a read-only transaction; else it is known as a read-write transaction.
- A database is basically represented as a collection of named data items.
- The size of a data item is called its granularity.
- A data item can be a database record, an attribute, a table, a database or an entire disk block.

Introduction to Transaction Processing

- Using a simplified database model, the basic database access operations that a transaction can include are as follows:
- 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.

Introduction to Transaction Processing

Executing a `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). The size of the buffer is the same as the disk block size.
3. Copy item X from the buffer to the program variable named X.





Introduction to Transaction Processing

Executing a `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 disk block from the buffer back to disk (either immediately or at some later point in time).



Introduction to Transaction Processing

- The DBMS will maintain in the database cache a number of data buffers in main memory.
 - Each buffer typically holds the contents of one database disk block, which contains some of the database items being processed.
 - When these buffers are all occupied, and additional database disk blocks must be copied into memory, some buffer replacement policy is used to choose which of the current occupied buffers is to be replaced. (Ex. LRU)
- 
- 
- 
- 

Introduction to Transaction Processing

(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>

- A transaction includes `read_item` and `write_item` operations to access and update the database.
- The read-set of a transaction is the set of all items that the transaction reads, and the write-set is the set of all items that the transaction writes.
- For example, the read-set of T_1 is $\{X, Y\}$ and its write-set is also $\{X, Y\}$.

Introduction to Transaction Processing

Why Concurrency Control Is Needed

- Several problems can occur when concurrent transactions execute in an uncontrolled manner.
- Figure (a) shows a transaction T_1 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 .

(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>

Introduction to Transaction Processing

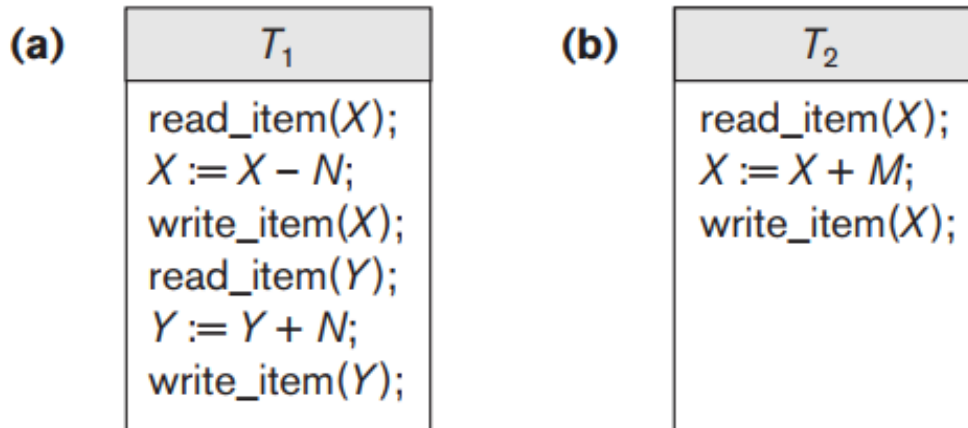


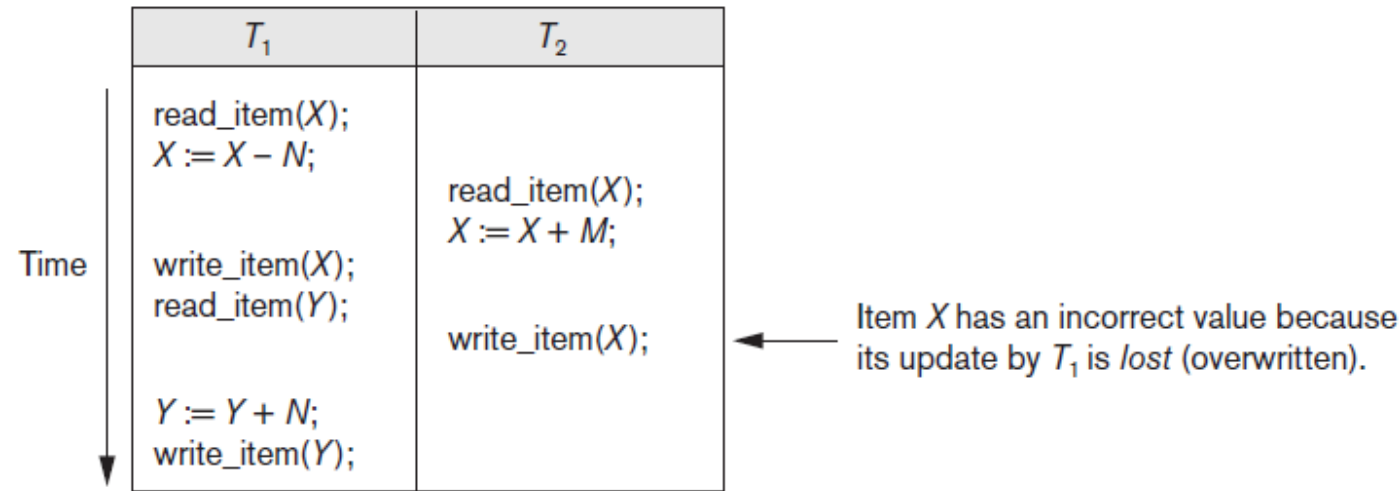
Figure (b) shows a simpler transaction T_2 that just reserves M seats on the first flight (X) referenced in transaction T_1 .

Types of problems we may encounter:

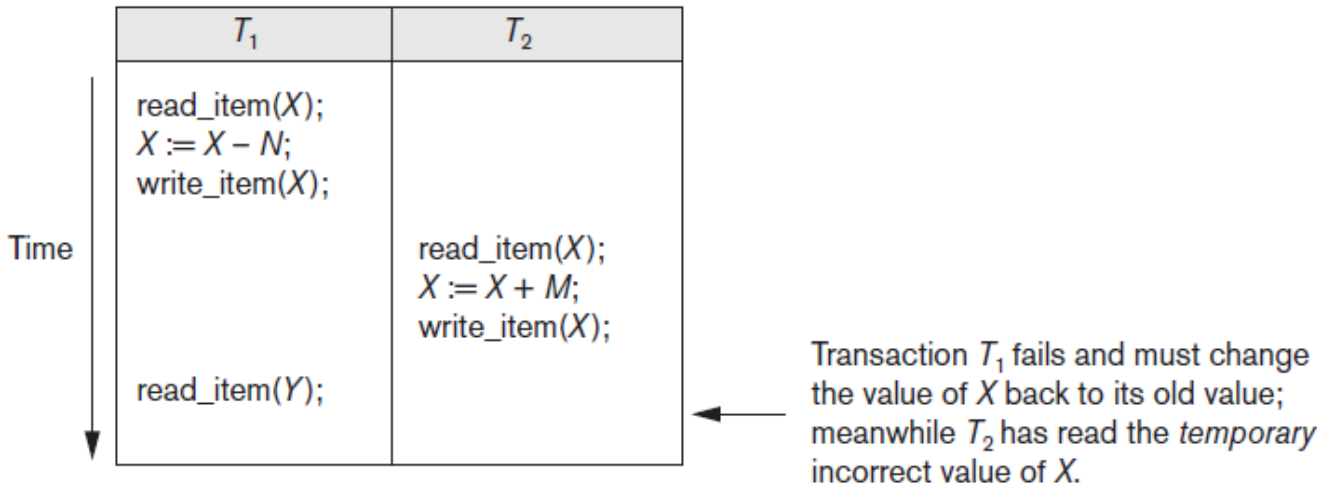
- The Lost Update Problem
- The Temporary Update (or Dirty Read) Problem.
- The Incorrect Summary Problem.
- The Unrepeatable Read Problem.

The Lost Update Problem

- This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect.
- Suppose that transactions T1 and T2 are submitted at approximately the same time and suppose that their operations are interleaved as shown in Figure.
- The final value of item X is incorrect because T2 reads the value of X before T1 changes it in the database, and hence the updated value resulting from T1 is lost.



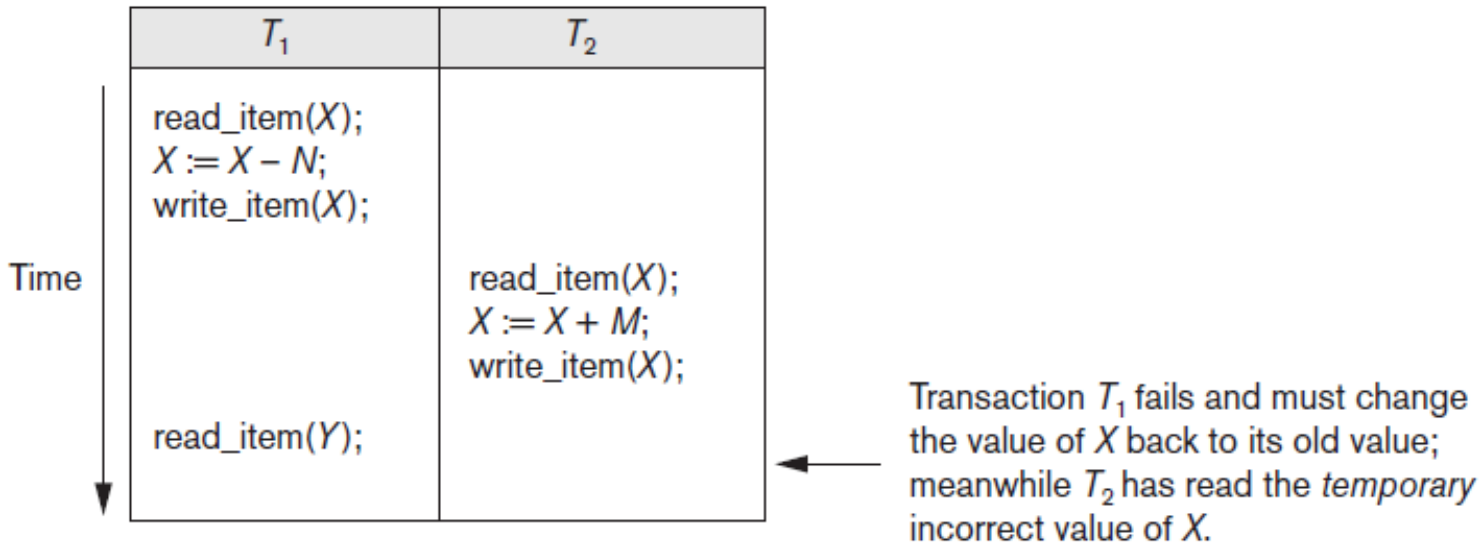
Introduction to Transaction Processing



Introduction to Transaction Processing

The Temporary Update (or Dirty Read) Problem

- This problem occurs when one transaction updates a database item and then the transaction fails for some reason.
- Meanwhile, the updated item is accessed (read) by another transaction before it is changed back (or rolled back) to its original value.
- Figure shows an example where T_1 updates item X and then fails before completion, so the system must roll back X to its original value.



Introduction to Transaction Processing

- Before it can do so, however, transaction T_2 reads the temporary value of X , which will not be recorded permanently in the database because of the failure of T_1 .
- The value of item X that is read by T_2 is called dirty data because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the dirty read problem.

T_1	T_3
<pre> read_item(X); X := X - N; write_item(X); read_item(Y); Y := Y + N; write_item(Y); </pre>	<pre> sum := 0; read_item(A); sum := sum + A; ⋮ read_item(X); sum := sum + X; read_item(Y); sum := sum + Y; </pre>

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

Introduction to Transaction Processing

The Incorrect Summary Problem

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

Introduction to Transaction Processing

The Unrepeatable Read Problem

- Another problem that may occur is called unrepeatable read, where 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. This may occur, for example, if during an airline reservation transaction, a customer inquires about seat availability on several flights.

Introduction to Transaction Processing

Why Recovery Is Needed

- The system is responsible for making sure that either all the operations in the transaction are completed successfully and saved (committed) or that the transaction does not have any effect on the database or any other transactions (aborted).
- Whole transaction is a logical unit of database processing.
- If a transaction fails after executing some of its operations but before executing all of them, the operations already executed must be undone and have no lasting effect.

Introduction to Transaction Processing

Types of Failures: Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution:

1. A computer failure (system crash)

- A hardware, software, or network error occurs in the computer system during transaction execution.
- Hardware crashes are usually media failures—for example, main memory failure.

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. Additionally, the user may interrupt the transaction during its execution.

Introduction to Transaction Processing

3. Local errors or exception conditions detected by the transaction

- During transaction execution, certain conditions may occur that necessitate cancellation of the transaction.
- For example, data for the transaction may not be found.

4. Concurrency control enforcement

- The concurrency control method may abort a transaction because it violates serializability.
- It may abort one or more transactions to resolve a state of deadlock among several transactions

Introduction to Transaction Processing

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.

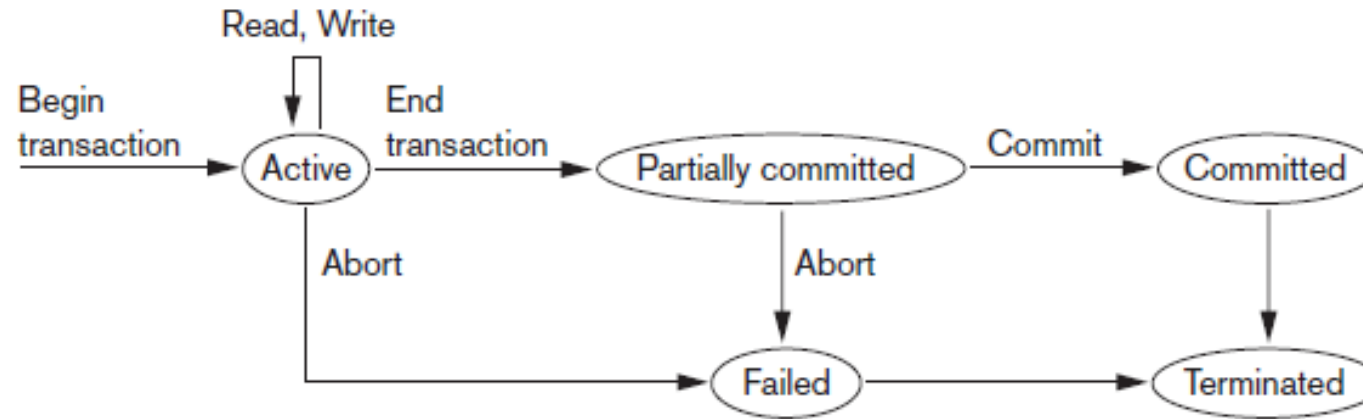
Transaction and System Concepts

Transaction States and Additional Operations

- For recovery purposes, the system needs to keep track of when each transaction starts, terminates, and commits, or aborts.
- The recovery manager of the DBMS needs to keep track of the following operations:
 - BEGIN_TRANSACTION
 - READ or WRITE
 - END_TRANSACTION
 - COMMIT_TRANSACTION
 - ROLLBACK (or ABORT)

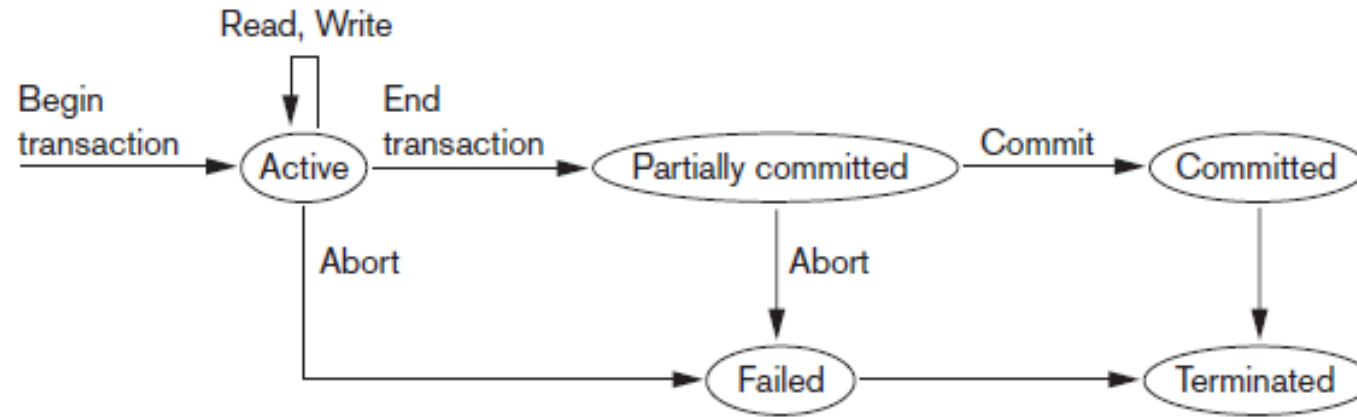
Transaction and System Concepts

-
- BEGIN_TRANSACTION
 - Marks the beginning of transaction execution
 - READ or WRITE
 - Specify read or write operations on the database items
 - END_TRANSACTION
 - Specifies that READ and WRITE transaction operations have ended
 - COMMIT_TRANSACTION
 - Signals a successful end of the transaction
 - ROLLBACK (or ABORT)
 - Signals that the transaction has ended unsuccessfully



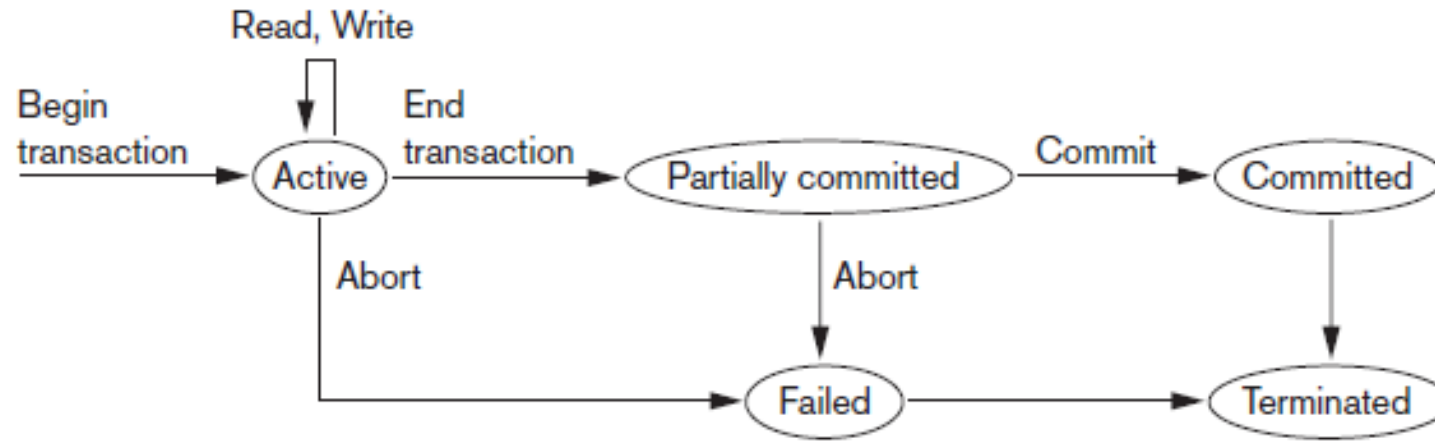
Transaction and System Concepts

- A transaction goes into an active state immediately after it starts execution, where it can execute its READ and WRITE operations.
- When the transaction ends, it moves to the partially committed state.
- At this point, some types of concurrency control protocols may do additional checks to see if the transaction can be committed or not.
- Some recovery protocols need to ensure that a system failure will not result in an inability to record the changes.



Transaction and System Concepts

- If these checks are successful, the transaction is said to have reached its commit point and enters the committed state.
- All its changes must be recorded permanently in the database, even if a system failure occurs.
- A transaction can go to the failed state if one of the checks fails or if the transaction is aborted during its active state.
- The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database.



Transaction and System Concepts

- The terminated state corresponds to the transaction leaving the system.
- The transaction information that is maintained in system tables is removed when the transaction terminates.
- Failed or aborted transactions may be restarted later

Transaction and System Concepts

The System Log

- To recover from failures, the system maintains a log to keep track of all transaction operations that affect the values of database items.
- The log is a sequential, append-only file that is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure.
- Typically, main memory buffers, called the log buffers, hold the last part of the log file, entries are first added to the log in buffer.
- When the log buffer is filled, or when certain other conditions occur, the log buffer is appended to the end of the log file on disk.

- The log file from disk is periodically backed up to archival storage (tape) to guard against catastrophic failures.
- The following are the types of entries (log records) that are written to the log file and the corresponding action for each log record.
- T refers to a unique transaction-id that is generated automatically.
 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.

Transaction and System Concepts

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.

- 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.
- This is done by tracing backward through the log and resetting all items changed by a WRITE operation of T to their old_values.
- Redo of an operation is when a transaction has its updates recorded in the log, but a failure occurs before the system could store data permanently.

Transaction and System Concepts

Commit Point of a Transaction

- 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 have been recorded in the log.
- Beyond the commit point, the transaction is said to be committed, and its effect must be permanently recorded in the database.
- The transaction then writes a commit record [commit, T] into the log.
- If a failure occurs, we can search back in the log for all transactions T that have written a [start_transaction, T] record into the log but have not written their [commit, T] record yet.
- these transactions may have to be rolled back to undo their effect on the database during the recovery process.

Transaction and System Concepts

Transaction and System Concepts

-
- It is common to keep blocks of the log file in main memory buffers (log buffer) until they are filled with log entries and then to write them back to disk only once, rather than writing to disk every time a log entry is added.
 - At the time of a system crash, only the log entries that have been written back to disk are considered in the recovery process if the contents of main memory are lost.
 - Hence, 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 buffer to disk before committing a transaction.

Transaction and System Concepts

DBMS-Specific Buffer Replacement Policies

- The DBMS cache will hold the disk pages that contain information currently being processed in main memory buffers.
- If all the buffers are occupied and new disk pages are required a page replacement policy is needed to select the particular buffers to be replaced.
- Domain Separation (DS) Method
 - Various types of disk pages exist, such as, index pages, data file pages, log file pages, and so on.
 - In this method, the cache is divided into separate domains.

Transaction and System Concepts

-
- Each domain handles one type of disk pages, and page replacements within each domain are handled via the basic LRU algorithm.
 - Does not adapt to dynamically changing loads because the number of available buffers for each domain is predetermined.
 - Several variations of the DS page replacement policy have been proposed.
 - Group LRU gives each domain a priority level and selects pages from the lowest-priority level domain first for replacement.
 - Another method dynamically changes the number of buffers in each domain based on current workload.

Transaction and System Concepts

—

- **Hot Set Method**

- This page replacement algorithm is useful in queries that have to scan a set of pages repeatedly, such as when a join operation is performed using the nested-loop method.
- If the inner loop file is loaded completely into main memory buffers without replacement (the hot set), the join will be performed efficiently because each page in the outer loop file will have to scan all the records in the inner loop file to find join matches.
- The hot set method determines for each database processing algorithm the set of disk pages that will be accessed repeatedly, and it does not replace them until their processing is completed.

Transaction and System Concepts

- **The DBMIN Method**

- This page replacement policy uses a model known as QLSM (query locality set model), which predetermines the pattern of page references for each algorithm for a particular type of database operation.
- Depending on the type of access method, the file characteristics, and the algorithm used, the QLSM will estimate the number of main memory buffers needed for each file involved in the operation.
- The DBMIN page replacement policy will calculate a locality set using QLSM for each file instance involved in the query.
- DBMIN then allocates the appropriate number of buffers to each file instance involved in the query based on the locality set for that file instance.

Desirable properties of Transactions

- Transactions should possess ACID properties; they should be enforced by the concurrency control and recovery methods of the DBMS.
- **Atomicity**
 - A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.
- **Consistency preservation**
 - A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions.
 - It should take the database from one consistent state to another.

Desirable properties of Transactions

- **Isolation**

- A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently.
- Execution of a transaction should not be interfered with by any other transactions executing concurrently.

- **Durability or permanency**

- The changes applied to the database by a committed transaction must persist in the database.
- These changes must not be lost because of any failure.

Desirable properties of Transactions

- The atomicity property requires that we execute a transaction to completion.
- It is the responsibility of the transaction recovery subsystem of a DBMS to ensure atomicity.
- The preservation of consistency is the responsibility of the programmers who write the database programs and of the DBMS module that enforces integrity constraints.
- The isolation property is enforced by the concurrency control subsystem of the DBMS.
- The durability property is the responsibility of the recovery subsystem of the DBMS.

Characterizing schedules based on recoverability

- When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from all the various transactions is known as a schedule.

Schedules (Histories) of Transactions

- A schedule (or history) S of n transactions T_1, T_2, \dots, T_n is an ordering of the operations of the transactions.
- Operations from different transactions can be interleaved in the schedule S .
- However, for each transaction T_i that participates in the schedule S , the operations of T_i in S must appear in the same order in which they occur in T_i .
- The order of operations in S is a total ordering, meaning that for any two operations in the schedule, one must occur before the other.

Characterizing schedules based on recoverability

- We are interested in the read_item and write_item operations of the transactions, as well as the commit and abort operations.
- A notation for describing a schedule uses the symbols b, r, w, e, c, and a for the operations begin_transaction, read_item, write_item, end_transaction, commit, and abort, respectively, and appends as a subscript the transaction id to each operation in the schedule.

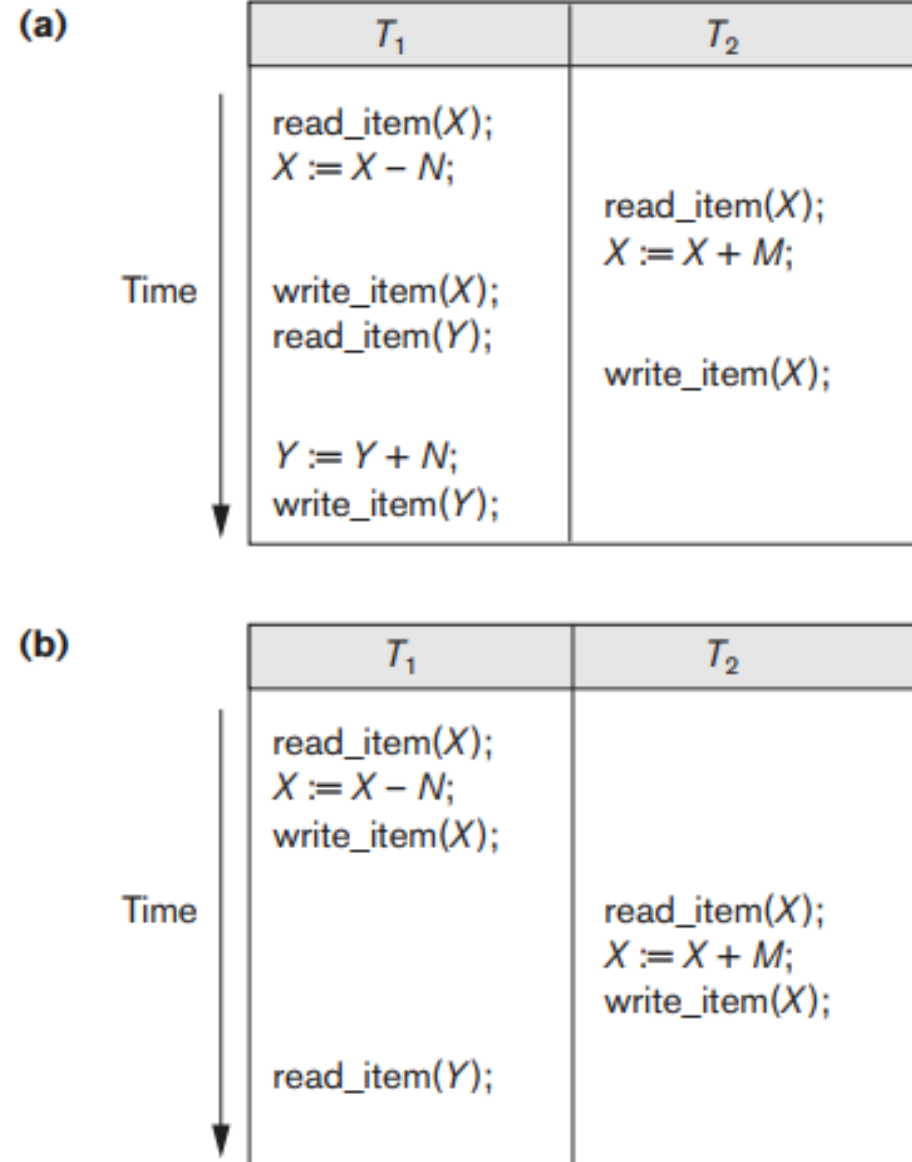
Characterizing schedules based on recoverability

- The schedule in Figure (a), which we shall call S_a , can be written as follows in this notation:

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

- Similarly, the schedule for Figure (b), which we call S_b , can be written as follows, if we assume that transaction T_1 aborted after its `read_item(Y)` operation:

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



Characterizing schedules based on recoverability

Conflicting Operations in a Schedule

- Two operations in a schedule are said to conflict if they satisfy all three of the following conditions:
 1. they belong to different transactions
 2. they access the same item X ;
 3. at least one of the operations is a $\text{write_item}(X)$.
- In schedule $S_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y)$; the 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 $w_2(X)$.

Characterizing schedules based on recoverability

- $S_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$
- 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 ; and the operations $r_1(X)$ and $w_1(X)$ do not conflict because they belong to the same transaction.
- Two operations are conflicting if changing their order can result in a different outcome.
- For example, if we change the order of the two operations $r_1(X); w_2(X)$ to $w_2(X); r_1(X)$, then the value of X that is read by transaction T_1 changes, because in the second ordering the value of X is read by $r_1(X)$ after it is changed by $w_2(X)$. This is called a read-write conflict.

Characterizing schedules based on recoverability

- The other type is called a write-write conflict and is illustrated by the case where we change the order of two operations such as $w_1(X); w_2(X)$ to $w_2(X); w_1(X)$.
- Two read operations are not conflicting because changing their order makes no difference in outcome.
- A schedule S of n transactions T_1, T_2, \dots, T_n is said to be a complete schedule if the following conditions hold:
 1. The operations in S are exactly those operations in T_1, T_2, \dots, T_n , including a commit or abort operation as the last operation for each transaction in the schedule.
 2. For any pair of operations from the same transaction T_i , their relative order of appearance in S is the same as their order of appearance in T_i .
 3. For any two conflicting operations, one of the two must occur before the other in the schedule.

Characterizing schedules based on recoverability

- For some schedules it is easy to recover from transaction and system failures, whereas for other schedules the recovery process can be quite involved.
- In some cases, it is even not possible to recover correctly after a failure.
- Hence, it is important to characterize the types of schedules for which recovery is possible, as well as those for which recovery is relatively simple.

Characterizing schedules based on recoverability

- We would like to ensure that, once a transaction T is committed, it should never be necessary to roll back T .
- This ensures that the durability property of transactions is not violated.
- The schedules that theoretically meet this criterion are called recoverable schedules.
- A schedule where a committed transaction may have to be rolled back during recovery is called nonrecoverable and hence should not be permitted by the DBMS.

Characterizing schedules based on recoverability

- A schedule S is recoverable if no transaction T in S commits until all transactions T' that have written some item X that T reads have committed.
- A transaction T reads from transaction T' in a schedule S if some item X is first written by T' and later read by T .
- In addition, T' should not have been aborted before T reads item X , and there should be no transactions that write X after T' writes it and before T reads it.

Characterizing schedules based on recoverability

- Consider the schedule S_a' given below, which is the same as schedule S_a except that two commit operations have been added to S_a .

$S_a': r_1(\mathbf{X}); r_2(X); w_1(X); r_1(Y); \mathbf{w}_2(\mathbf{X}); c_2; w_1(Y); c_1;$

- S_a' is recoverable as per definition.
- S_c is not recoverable because T_2 reads item X from T_1 , but T_2 commits before T_1 commits.
- The problem occurs if T_1 aborts after the c_2 operation in S_c ; then the value of X that T_2 read is no longer valid and T_2 must be aborted after it is committed, leading to a schedule that is not recoverable.

$S_c: r_1(X); \mathbf{w}_1(\mathbf{X}); \mathbf{r}_2(\mathbf{X}); r_1(Y); w_2(X); \mathbf{c}_2; \mathbf{a}_1;$

Characterizing schedules based on recoverability

- In a recoverable schedule, no committed transaction ever needs to be rolled back, and so the definition of a committed transaction as durable is not violated.
- However, it is possible for a phenomenon known as cascading rollback (or cascading abort) to occur in some recoverable schedules, where an uncommitted transaction must be rolled back because it read an item from a transaction that failed.
- A schedule is said to be cascadeless if every transaction in the schedule reads only items that were written by committed transactions.
- In this case, all items read will not be discarded because the transactions that wrote them have committed, so no cascading rollback will occur.

Characterizing schedules based on recoverability

- A more restrictive type of schedule, called a strict schedule, in which transactions can neither read nor write an item X until the last transaction that wrote X has committed (or aborted).
- In a strict schedule, the process of undoing a `write_item(X)` operation of an aborted transaction is simply to restore the before image (old_value or BFIM) of data item X.
- This simple procedure always works correctly for strict schedules, but it may not work for recoverable or cascadeless schedules.

Characterizing schedules based on recoverability

- For example, consider schedule S_f :

$S_f: w_1(X, 5); w_2(X, 8); a_1;$

- Suppose that the value of X was originally 9, which is the before image stored in the system log along with the $w_1(X, 5)$ operation.
- If T_1 aborts, as in S_f , the recovery procedure that restores the before image of an aborted write operation will restore the value of X to 9, even though it has already been changed to 8 by transaction T_2 , thus leading to potentially incorrect results.

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

Characterizing schedules based on Serializability

- 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' .

Characterizing schedules based on Serializability

- 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

- 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.

Characterizing schedules based on Serializability

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 $R_i(X)$ 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' .

Characterizing schedules based on Serializability

- 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.

Characterizing schedules based on Serializability

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

Characterizing schedules based on Serializability

- Relationship between view and conflict equivalence (cont):
 - Consider the following schedule of three transactions
 - T1: $r_1(X)$, $w_1(X)$; T2: $w_2(X)$; and T3: $w_3(X)$:
 - Schedule Sa: $r_1(X)$; $w_2(X)$; $w_1(X)$; $w_3(X)$; c1; c2; c3;
- In Sa, the operations $w_2(X)$ and $w_3(X)$ 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.

Characterizing schedules based on Serializability

Testing for conflict serializability: Algorithm

- 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.

Characterizing schedules based on Serializability

- Constructing the precedence graphs for schedules A and D 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).

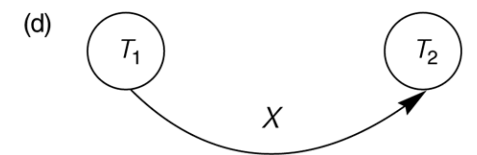
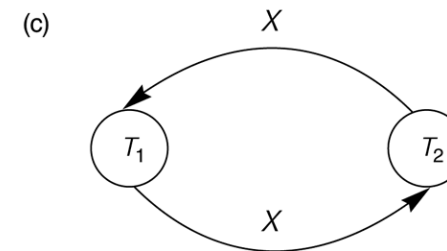
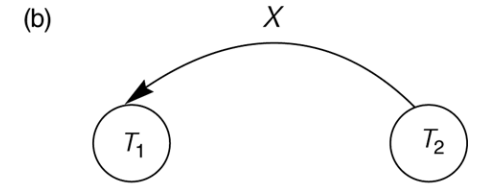
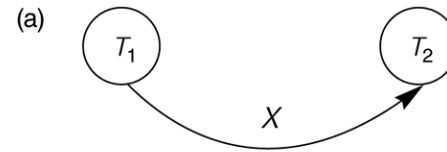


Figure 17.8

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);

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(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

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
	<code>read_item(X);</code> <code>write_item(X);</code> <code>read_item(Y);</code> <code>write_item(Y);</code>	 <code>read_item(Z);</code> <code>read_item(Y);</code> <code>write_item(Y);</code> <code>read_item(X);</code> <code>write_item(X);</code>	<code>read_item(Y);</code> <code>read_item(Z);</code> <code>write_item(Y);</code> <code>write_item(Z);</code>

Schedule F

Characterizing schedules based on Serializability

Transaction support in SQL

- A single SQL statement is always considered to be atomic—either it completes execution without an error, or it fails and leaves the database unchanged.
- Every transaction has certain characteristics attributed to it and are specified by a SET TRANSACTION statement in SQL.
- The characteristics are the access mode, the diagnostic area size, and the isolation level.
- The access mode can be specified as 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.
- A mode of READ WRITE allows select, update, insert, delete, and create commands to be executed. A mode of READ ONLY is for data retrieval.

Transaction support in SQL

- The diagnostic area size option, `DIAGNOSTIC SIZE n`, specifies an integer value `n`, which indicates 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.
- The isolation level option is specified using the statement `ISOLATION LEVEL <isolation>`, where the value for `<isolation>` can be `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ`, or `SERIALIZABLE`.

Transaction support in SQL

- The default isolation level is `SERIALIZABLE`, although some systems use `READ COMMITTED` as their default.
- The use of the term `SERIALIZABLE` here is based on not allowing violations that cause dirty read, unrepeatable read, and phantoms.
- If a transaction executes at a lower isolation level than `SERIALIZABLE`, then one or more of the following three violations may occur:
 - Dirty read
 - Nonrepeatable read
 - Phantoms.

Transaction support in SQL

- **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.

Transaction support in SQL

- **Phantoms**

- 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 r that also satisfies the WHERE-clause condition used in T1, into the table used by T1.
- The record r is called a phantom record because it was not there when T1 starts but is there when T1 ends.
- T1 may or may not see the phantom, a row that previously did not exist.
- Solution : Lock entire table or use predicate locking.

Transaction support in SQL

- READ UNCOMMITTED is the most forgiving, and SERIALIZABLE is the most restrictive in that it avoids all three of the problems.

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

Transaction support in SQL

Example: Setting isolation level.

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
    READ WRITE
    DIAGNOSTIC 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: ... ;
```

End of Module 4
